第3章 语法分析

语法分析就是根据高级语言的语法规则对程序的语法结构进行分析，是编译过程的核心。它的任务是判断读入的单词符号串是否符合语言的语法规则，为语义分析和代码生成做准备。执行语法分析的程序称为语法分析程序，也称为语法分析器。

为了能够更精确地描述高级语言程序的语法结构，需要对高级语言的语法规则进行形式化描述，这种描述称为文法，适合描述高级语言语法规则的文法是上下文无关文法。因此本章首先介绍文法的相关概念。

语法分析的方法很多，不同的语法分析方法适用于不同的文法，有不同的使用场合和限制条件。语法分析不仅可以手工构造，也可以自动生成，本章最后介绍自动生成器 YACC 的基本原理和使用方法。

3.1 语法分析概述

语法分析在编译过程中处于核心地位，如图 3.1 所示。其任务是在词法分析识别出正确的单词符号串的基础上，根据语言定义的语法规则，分析并识别出各种语法成分，同时进行语法检查和错误处理。根据第 1 章的介绍，语法分析程序的输入是 token 串，输出是语法树。实际上，有时并不需要显式地构造语法树，因为很多时候，语法分析可能会和后续的翻译交错进行。

图 3.1 语法分析器在编译程序中的地位

每一种程序设计语言都有描述其语法结构的规则，如 C 语言程序由一个或多个函数构成，至少包含一个 main()函数，每个函数定义为一个复合语句，每个复合语句定义为由一对花括号{和}括起来的多个顺序执行的语句，语句有多种类型，多数语句由表达式组成，表达式由表达式、标识符、常量和运算符等构成。如果仅仅用文字这样表述，不便于计算机精确处理和判断。因此，需要对程序设计语言的语法构成规则进行形式化描述。程序设计语言的语法规则一般用上下文无关文法来描述。

上下文无关文法用递归的方式描述语法规则。语法分析的过程就是按文法规则对读入的 token 串(又称为输入符号串)进行分析的过程。token 串中的每个单词符号对应于文法中的一个符号。

根据文法可以手工或自动生成一个有效的语法分析程序，用来判断输入的符号串在语

法上是否正确。判断的依据就是对给定的输入符号串能否根据文法规则建立起一棵语法树。

按照语法树的建立方法,可以粗略地把语法分析方法分成两类:自上而下分析法和自下而上分析法。

自上而下分析法是在自左至右扫描输入符号串的过程中,从树根开始逐步向下建立语法树。使用自上而下的语法分析的困难在于表示源语言语法结构的文法需要满足特定的要求,但由于多数程序设计语言的控制流结构具有不同的关键字,如 if、while、for,因此这种方法的优势在于一旦检测出关键字,就知道哪个文法规则是唯一的选择,实现起来简单、直观,便于手工构造或自动生成语法分析器,它仍是目前常用的方法之一。常用的自上而下的分析方法有递归下降分析和预测分析两种方法,我们将在 3.3 节详细介绍。

自下而上分析法是在自左至右扫描输入符号串的过程中,沿着从树叶向树根的方向逐步建立语法树,直到树根结点。自下而上的语法分析方法对文法的限制条件少,对大多数常见的高级语言的语法分析都能使用。常用的自下而上的语法分析方法有算符优先分析和 LR 分析两种,多数商业化的编译器和语法分析的自动生成器也都采用自下而上的语法分析方法。算符优先分析方法是多数编译器中用来分析算术表达式的方法;对于几乎所有的程序设计语言,只要能够构造出它的上下文无关文法,就能够构造出识别它的 LR 语法分析器,语法分析的自动生成器 YACC 采用的是 LR 分析方法,将在 3.4 节详细介绍自下而上的语法分析方法,并在 3.5 节介绍语法分析自动生成器 YACC 的原理和使用。

3.2 上下文无关文法

对于高级程序设计语言而言,程序的语法结构是基于语法规则的,因此语法规则的定义和描述非常重要。程序设计语言的语法规则的形式化描述称为文法。本节主要介绍文法及其产生语言的方法——推导,并用语法树的方式描述推导过程。

3.2.1 文法的定义

文法(Grammar)是描述语言的语法结构的形式规则(即语法规则),这些规则必须准确而且可理解。文法是从产生语言中的句子的观点来描述一个语言,也就是说语言中的每个句子都可以用严格定义的规则来产生。

下面以自然语言为例,用语法规则来分析句子,从而得出文法的形式化定义。

例 3.1 有如下规则:

(1) <句子>→<主语><谓语>

(2) <主语>→<代词>|<名词>

(3) <代词>→我

(4) <名词>→大学生

(5) <谓语>→<动词><直接宾语>

(6) <动词>→是

(7) <直接宾语>→<代词>|<名词>

其中,"<句子>"表示该应用规则的开始;"→"表示"由……组成"或"定义为";"|"表示

“或”,具有相同左部的几个规则可以用“|”写在一起,如上述第2条和第7条规则实际各自代表了两条规则。

现在根据上述规则可以得到一个符合<句子>定义的规则的句子:我是大学生。分析过程如下。

<句子>⇒<主语><谓语> 应用规则(1)
⇒<代词><谓语> 应用规则(2)
⇒我<谓语> 应用规则(3)
⇒我<动词><直接宾语> 应用规则(5)
⇒我是<直接宾语> 应用规则(6)
⇒我是<名词> 应用规则(7)
⇒我是大学生 应用规则(4)

这说明,从<句子>出发,反复使用上述规则中“→”右边的成分替换左边的成分,产生“我是大学生”这样一个句子,从而说明按照上述规则“我是大学生”在语法上是正确的。

上述自然语言的定义就是一个文法。根据上述实例可以抽象出如下一些概念。

(1) 非终结符(Nonterminator):在上述规则中用尖括号括起来的符号,它们各自代表一个语法范畴,表示一类具有某种性质的语法单位,有时也称为语法变量。可以通过它们替换出其他句子成分,它们不会出现在最终的句子中,如例3.1中的<句子>、<主语>等。在程序设计语言中的非终结符有“算术表达式”“赋值语句”等。用 V_N 表示非终结符的集合。非终结符给出了语言的层次和结构,这种层次化结构是语法分析和翻译的关键。

(2) 终结符(Terminator):出现在最终的句子中的符号,它是一个语言的基本单位的集合,如例3.1中不带尖括号的符号“我”“是”“大学生”。在程序设计语言的语法规则中终结符就是单词符号,如关键字、标识符和界符等。用 V_T 表示终结符的集合。

$V=V_N \cup V_T$,构成文法G的字母表,是该文法中可以使用的全部符号,$V_N \cap V_T=\Phi$。

(3) 产生式(Production):按一定格式书写的、用于定义语法范畴的规则,又称为规则或生成式,说明了终结符和非终结符组合成符号串的方式。形如 $\alpha \rightarrow \beta$ 或 $\alpha ::= \beta$,称 α 为左部,$\alpha \in V^+$,α 至少包含一个非终结符;β 为右部,$\beta \in V^*$。如例3.1中“<句子>→<主语><谓语>”是一个产生式。用P表示产生式的集合,如例3.1中有9个产生式。

(4) 开始符号(Starter):是一个特殊的非终结符,至少在一个产生式的左部出现。用S表示,代表该文法定义的语言最终要得到的语法范畴,如例3.1中的<句子>。在程序设计语言中,开始符号就是<程序>,文法定义的其他语法范畴都为此服务。

由此,给出文法的形式化定义:文法G是一个四元组,$G=(V_N,V_T,P,S)$,V_N 是非终结符集,V_T 是终结符集,S是开始符号,P是产生式集合。

对例3.1的文法可表示为:$G=(V_N,V_T,P,<句子>)$,其中 $V_N=\{<句子>,<主语>,<谓语>,<直接宾语>,<代词>,<动词>,<名词>\}$,$V_T=\{我,是,大学生\}$,P就是例3.1中给出的9条规则。

例3.2 Java语言中标识符定义的文法为:

文法 $G=(V_N,V_T,P,S)$

其中:

V_N = {<标识符>,<字母>,<数字>}
V_T = {a,b,c,…,y,z,0,1,…,9}
P = {<标识符>→<字母>
<标识符>→<标识符><字母>
<标识符>→<标识符><数字>
<字母>→a
<字母>→b
⋮
<字母>→z
<数字>→0
⋮
<数字>→9
}
S = <标识符>

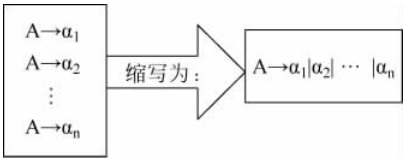
有时不用将文法 G 的四元组显式地表示出来,只将产生式写出。在书写产生式时一般有下列约定。

(1) 第一条产生式的左部是开始符号。

(2) 在产生式中,用大写英文字母表示非终结符,用小写英文字母表示终结符,用小写希腊字母(如 α、β 和 γ)代表任意的文法符号串。

(3) 如果 S 是文法 G 的开始符号,也可以将文法 G 写成 G[S]。

(4) 有时为了书写简洁,常把相同左部的多个产生式用"|"进行缩写,如:



元符号"|"读作"或",其中每个 α_i 称为 A 的一个候选式。

这种描述文法的方法称为巴科斯范式(Backus-Naur Form,BNF),"→"有时也用"::="来表示,是一种严格地表示语法规则的方法。BNF 表示语法规则的方式为:每条规则的左部是一个非终结符,右部是由非终结符和终结符组成的一个符号串,中间一般以"::="或"→"分开,具有相同左部的规则可以共用一个左部,各右部之间以竖线"|"隔开。

例 3.3 下面是一个文法的几种等价写法。

(1) G=({S,A},{a,b},P,S)
其中,P: S→aAb
A→ab
A→aAb
A→ε

(2) G: S→aAb
A→ab
A→aAb
A→ε

(3) G[S]: A→ab
A→aAb

A→ε

S→aAb

(4) G[S]：A→ab|aAb|ε

S→aAb

例 3.4 设某语言中算术表达式的语法规则定义为：

```
表达式 + 表达式是表达式
表达式 * 表达式是表达式
(表达式) 是表达式
单个数字是表达式
```

如果用 E 表示表达式，i 表示 0～9 的单个数字，则表达式的语法规则用文法表示为：

```
E→E + E
E→E * E
E→(E)
E→i
```

或者简写为：

$$E \rightarrow E+E \mid E*E \mid (E) \mid i \qquad (G3.1)$$

3.2.2 推导

在例 3.1 中，得到“我是大学生”这个符号串的方法是：从文法的开始符号出发，反复、连续使用所有可能的产生式，将一个符号串中的非终结符用某个产生式右部进行替换和展开，直到全部为终结符为止。这个过程称为推导(Derivation)。

例如，对算术表达式文法 G3.1，产生式 E→E+E 意味着允许用 E+E 代替文法符号串中出现的任何 E，以便从简单的表达式产生更复杂的表达式。“用 E+E 代替 E”这个动作可以用

$$E \Rightarrow E+E$$

来描述，读作“E 推导出 E+E”。

表达式(i+i)的推导过程可以表示为：

$$E \Rightarrow (E) \Rightarrow (E+E) \Rightarrow (i+E) \Rightarrow (i+i) \qquad (3.1)$$

抽象地说，如果 A→γ 是产生式，α 和 β 是文法的任意符号串，αAβ⇒αγβ 称为直接推导(Direct Derivation)，也称一步推导，用符号“⇒”表示“一步推导”。如果 $\alpha_1 \Rightarrow \alpha_2 \Rightarrow \cdots \Rightarrow \alpha_n$，称从 α_1 到 α_n 的整个序列为一个推导，称 α_1 推导出 α_n。用符号 $\overset{*}{\Rightarrow}$ 表示“零步或多步推导”。于是

(1) 对任何符号串有 $\alpha \overset{*}{\Rightarrow} \alpha$，并且

(2) 如果 $\alpha \overset{*}{\Rightarrow} \beta$，$\beta \overset{*}{\Rightarrow} \gamma$，那么 $\alpha \overset{*}{\Rightarrow} \gamma$。

类似地，用符号 $\overset{+}{\Rightarrow}$ 表示“一步或多步推导”，即至少经过一步推导。若有 $v \overset{+}{\Rightarrow} w$，或 v=w，则记作 $v \overset{*}{\Rightarrow} w$。于是，式(3.1)表示从 E 到(i+i)的推导过程，写作 $E \overset{*}{\Rightarrow} (i+i)$，或 $E \overset{+}{\Rightarrow} (i+i)$。

这个推导过程提供了一种证明(i+i)是一个符合文法 G3.1 的表达式的一种方法。推

导每前进一步,都要引用一条产生式规则。

推导的逆过程称为归约(Reduction)。如果 α_1 推导出 α_n,则称 α_n 可归约为 α_1。直接推导的逆过程称为直接归约(Direct Reduction)。

在推导的每一步都有两个选择:第一个是选择被替换的非终结符;第二个是选择用该非终结符的哪个候选式进行替换。如果在推导过程中的某一步有两个或多个非终结符,那么就需要决定下一步推导替换哪个非终结符。例如,在推导式(3.1)中,在得到符号串(E+E)后,也可以按如下进行:

$$(E+E)\Rightarrow(E+i)\Rightarrow(i+i) \tag{3.2}$$

式(3.2)在替换每个非终结符时所用产生式和式(3.1)一样,但有不同的替换次序。因此,从一个符号串到另一个符号串的推导过程不是唯一的。

为了理解某些分析器的工作过程,需要考虑每一步推导中非终结符的替换顺序。如果在整个推导中,每一步都是替换符号串中最左边的非终结符,这样的推导称为最左推导(Leftmost Derivation)。推导式(3.1)是最左推导。类似地可以定义最右推导(Rightmost Derivation),即在推导的每一步都替换符号串中最右边的非终结符。最右推导又称为规范推导(Canonical Derivation)。推导式(3.2)是最右推导。

最左推导的逆过程是最右归约(Rightmost Reduction),最右推导的逆过程称为最左归约(Leftmost Reduction),又称为规范归约(Canonical Reduction)。

例 3.5 对算术表达式文法 G3.1,写出(i+i)*i 的最左推导及最右推导过程。

最左推导:$E\Rightarrow E*E\Rightarrow(E)*E\Rightarrow(E+E)*E\Rightarrow(i+E)*E\Rightarrow(i+i)*E\Rightarrow(i+i)*i$

最右推导:$E\Rightarrow E*E\Rightarrow E*i\Rightarrow(E)*i\Rightarrow(E+E)*i\Rightarrow(E+i)*i\Rightarrow(i+i)*i$

3.2.3 文法产生的语言

若 S 是文法 G 的开始符号,从开始符号 S 出发推导出的符号串称为文法 G 的一个句型(Sentential Form)。即 α 是文法 G 的一个句型,当且仅当存在如下推导:$S\overset{*}{\Rightarrow}\alpha,\alpha\in V^*$。如在推导式(3.1)中,E、(E)、(E+E)、(i+E)、(i+i)都是文法 G3.1 的句型。

若 X 是文法 G 的一个句型,且 $X\in V_T^*$,则称 X 是文法 G 的一个句子(Sentence),即仅含终结符的句型是一个句子。在推导式(3.1)中,(i+i)是文法 G3.1 的句子。

把文法 G 产生的所有句子的集合称为 G 产生的语言(Language),记为 L(G),表示为:

$$L(G)=\{X\mid S\overset{*}{\Rightarrow}X,X\in V_T^*\}$$

推导是描述文法定义语言的有用方法。

如果文法 G_1 与 G_2 产生的语言相同,即 $L(G_1)=L(G_2)$,则称文法 G_1 和 G_2 是等价的(Equivalent)。在形式语言和编译理论中,文法等价是一个很重要的概念,根据这一概念,可对文法进行等价改造,以得到所需形式的文法。

例 3.6 考虑文法 $G_1=(\{S\},\{0\},S,\{S\to 0S,S\to 0\})$ 和 $G_2=(\{S\},\{0\},S,\{S\to S0,S\to 0\})$,证明它们是等价的。

对于 G_1,从开始符号开始推导,可以得到如下的句子:

$S\Rightarrow 0$

$S\Rightarrow 0S\Rightarrow 00$

$S\Rightarrow 0S\Rightarrow 00S\Rightarrow 000$

⋮

$S \Rightarrow 0S \Rightarrow 00S \Rightarrow \cdots \Rightarrow 000\cdots0$

归纳得，从 S 出发推导出的句子是由 1 个或多个 0 组成的符号串，用集合形式表示为：

$$L(G_1) = \{0^n \mid n \geqslant 1\}$$

同样，对于 G_2，从开始符号开始推导，可以得到：

$$S \Rightarrow S0 \Rightarrow S00 \Rightarrow \cdots \Rightarrow 000\cdots0$$

即

$$L(G_2) = \{0^n \mid n \geqslant 1\}$$

很显然，$G_1 \neq G_2$，但 $L(G_1) = L(G_2)$，所以文法 G_1 和 G_2 是等价的。

例 3.7 构造一个上下文无关文法 G 使得：

$$L(G) = \{a^n b^n \mid n \geqslant 1\}$$

G 中要求 a、b 的个数相同，每一次 a 的出现必然有一个 b 出现，并要求所有 a 的出现都在 b 的前面，则文法 G 可写为：

$$S \rightarrow aSb \mid ab$$

3.2.4 语法树

上面用推导的方式来考查文法定义语言的过程，但是推导不能表示句子的各个组成部分间的结构关系。本节用语法树来观察句子的构成，表示句子的层次关系。

在第 1 章中介绍了语法树，语法树就是用一棵树来表示一个句型的推导过程，有时也称为语法分析树、分析树(Parse Tree)。语法树是一棵倒立的树，根在上，枝叶在下，根结点由开始符号标记。随着推导的展开，当某个非终结符被它的候选式所替换时，这个非终结符就产生下一代新结点，候选式中自左至右的每个符号对应一个新结点，每个新结点和其父结点之间有一条连线。如 A 用产生式 A→UVW 推导时，语法树向下扩展一层，如下所示。

语法树的叶子结点由非终结符、终结符和 ε 标记。在语法树生长过程中的任意时刻，所有那些没有后代的端末结点从左到右排列起来构成一个句型。如果端末结点自左至右排列起来都是终结符，那么这棵语法树表示了这个句子的推导过程。

语法树有助于理解一个句子语法结构的层次。

例如，对算术表达式文法 G3.1，表达式(i+i)*i 的最左推导的语法树(包括推导过程)如图 3.2 所示。

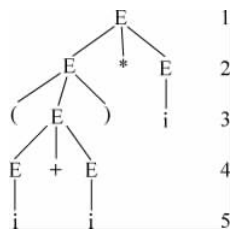


图 3.2 表达式(i+i)*i 的语法树

虽然图 3.2 表示了最左推导的语法树，但在第 4 层，到底是左边的 E 先推导出 i，还是右边的 E 先推导出 i，从语法树上反映不出来。

因此对一个句子或一个句型的推导过程不止一种，一棵语法树表示了一个句型的多种不同的推导过程，包括最左推导和最右推导。所以语法树是这些不同推导过程的共性抽象，但它不能表示非终结符替换顺序的选择。如果只考虑最左推导(或最右推导)，则可以消除推导过程中产生式应用顺序的不一致性。每棵语法树都有一个与

之对应的唯一的最左推导和唯一的最右推导。因此可以用产生语法树的方法来代替推导。

3.2.5 二义文法

然而,对给定的一个句型可能对应多棵不同的语法树,或者说,不一定只有一个最左推导或最右推导。例如,考虑算术表达式文法 G3.1,句子 i*i+i 有如下两种不同的最左推导。

(1) E ⇒ E+E ⇒ E*E+E ⇒ i*E+E ⇒ i*i+E ⇒ i*i+i

(2) E ⇒ E*E ⇒ i*E ⇒ i*E+E ⇒ i*i+E ⇒ i*i+i

因而也有两棵不同的语法树,如图 3.3(a)和图 3.3(b)所示。这两棵语法树的不同之处在于在推导过程中以不同的顺序选用不同的产生式,这说明可以用两种不同的推导过程生成同一个句子。

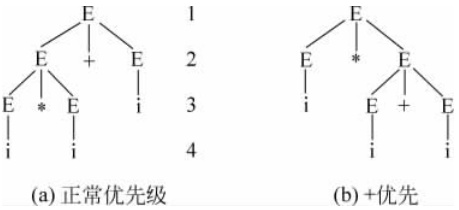


图 3.3 i*i+i 的语法树

如果一个文法存在某个句型对应两棵或两棵以上不同的语法树,则称这个文法为二义文法(Ambiguous Grammar),也就是说,二义文法是存在某个句型有不止一个最左(最右)推导的文法。对二义文法中某些句子的分析过程不是唯一的,也就不能确定某个句子应该选择哪棵语法树进行分析,所以有些程序设计语言的分析器要求处理的文法是无二义的。这样在设计文法时,可能需要使用一些附加的规则来消除二义性。

例 3.8 证明下述描述 if 语句的文法是二义文法。

设 if 语句 S 的文法 G=({E,S},{if,then,else,a,e},S,P),其中 P 为:

```
S→if E then S              (1)
S→if E then S else S       (2)
S→a                        (3)
E→e                        (4)
```

由文法可推导:S ⇒ if E then S ⇒ if E then if E then S else S

同样也可推导:S ⇒ if E then S else S ⇒ if E then if E then S else S

对于同一个句型 if E then if E then S else S,由于应用产生式的顺序不同,得到了两个不同的推导,所以该文法是二义文法。

文法的二义性并不代表语言一定是二义的。只有当产生一个语言的所有文法都是二义的,这个语言才是二义的。因为可能存在这种情况:有两个不同的文法 G 和 G′,其中一个是二义的,一个是无二义的,但它们产生的语言是相同的,这种语言也不是二义的,因为可以用无二义的文法来代替二义文法进行分析。

3.2.6 消除二义性

从图 3.3 可以看出，算术表达式文法 G3.1 对于句子 i＊i＋i 对应了两棵不同的语法树，说明 G3.1 是二义文法。该文法具有二义性是因为从文法本身来看，并不能反映运算符＋和＊的优先关系。图 3.3(a)的语法树反映了＋和＊通常的优先关系，即＊的优先级高于＋，而图 3.3(b)的语法树中反映出＋的优先级高于＊。

可以利用运算符之间的优先级和结合性来消除算术表达式文法 G3.1 的二义性。在通常的算术运算中，＊和/比＋和－的优先级高，它们都遵循左结合的原则进行运算。这样这几个运算符之间的优先级和结合性如表 3.1 所示。

表 3.1 运算符之间的优先级和结合性

优先级	结合性	运算符
1	左结合	＋、－
2	左结合	＊、/

左结合的含义是当同时出现相同优先级的运算符时，从左到右进行运算。

这样，可以引入两个非终结符号 expr 和 term，分别对应上述两个不同的优先级层次，并使用另一个非终结符 factor 来生成算术表达式的基本单元，如在算术表达式中，基本单元是单个的数字和带括号的表达式。有

```
factor→i | (expr)
```

考虑高优先级的＊和/及左结合性，可以认为 term 是由＊和/运算符分开的基本单元，但不能被低优先级的运算符分开的表达式，有：

```
term→term * factor | term/factor | factor
```

＋和－的优先级较低，算术表达式可以认为是由＋和－运算符分开的 term 列表，也就是说一个表达式可以由任何运算符分开。有：

```
expr→expr + term | expr - term | term
```

这样就得到了带优先级和结合性的算术表达式文法：

```
expr→expr + term | expr - term | term
term→term * factor | term/factor | factor
factor→i | (expr)
```

由于＋和－的优先级相同，＊和/的优先级相同，不失一般性，以后主要考虑＋和＊，并在上述文法中，用 F 代表 factor，T 代表 term，E 代表 expr。这样带优先级和结合性的算术表达式文法就可以用下面的文法 G3.2 来描述，在本书后面关于算术表达式的讨论也经常使用这个文法。

$$\begin{aligned} &E \rightarrow E + T \mid T \\ &T \rightarrow T * F \mid F \\ &F \rightarrow i \mid (E) \end{aligned} \qquad (G3.2)$$

例3.8中描述if语句的文法是二义文法,主要是因为在类似于if E then if E then S else S的语句时,前面有两个if,后面出现了一个else,else不知道和哪个if语句匹配,称为"悬空-else"。这就导致有两种理解:一种是和第一个if匹配;另一种是和第二个if匹配。

在大多数程序设计语言中对这种二义性的解决方案是:规定else总是和最近的尚未匹配的if匹配。不再修改文法的产生式,而是在分析过程中直接进行判断。

*3.2.7 Sample语言文法描述

本章主要介绍语法分析,在语法分析前,必须详细了解语言的语法规则。不同语言的语法规则不同,下面以Sample语言的语法规则为例,使用BNF形式来描述Sample语言的语法规则。

1. Sample语言中的表达式

表达式是Sample语言中唯一对数据进行处理的成分,由运算对象(数据引用或函数调用)和运算符组成。根据运算符的不同,表达式分为算术表达式、关系表达式、布尔表达式和赋值表达式四种类型。这四种类型的运算是有优先级的,算术运算的优先级比关系运算的优先级高,关系运算的优先级比布尔运算的优先级高,布尔运算的优先级比赋值运算的优先级高。Sample语言的表达式文法定义如下:

<表达式>→<算术表达式>|<关系表达式>|<布尔表达式>|<赋值表达式>

然而,就像在C语言中一样,所有的运算都可以根据优先级混合进行,因此也可以看作是同一类运算。

算术表达式在高级语言中对数据进行运算。算术运算符包括+、-、*、/、%,运算对象是指各种标识符、常量和函数调用。由于优先级和结合性的存在,表达式一般用递归规则来定义,如算术表达式由项进行加减运算构成,项由因子进行乘除运算构成,因子可以认为是单个的标识符、常量、带括号的表达式、因子取负和函数调用等。算术表达式的文法定义如下:

<算术表达式>→<项>+<算术表达式>|<项>-<算术表达式>|<项>
<项>→<因子>*<项>|<因子>/<项>|<因子>%<项>|<因子>
<因子>→(<算术表达式>)|<常量>|<变量>|<函数调用>
<常量>→<数值型常量>|<字符型常量>
<变量>→<标识符>
<函数调用>→<标识符>(<实参列表>)
<实参列表>→<实参>|ε
<实参>→<表达式>|<表达式>,<实参>

其中,<标识符>、<数值常量>、<字符常量>,以及不带尖括号的(、)和运算符都是语法分析的终结符,它们都具有词法的token值。

关系表达式的运算对象是算术表达式,运算符有大于(>)、小于(<)、大于或等于(>=)、小于或等于(<=)、等于(==)和不等于(!=)6种,关系表达式的文法定义如下:

<关系表达式>→<算术表达式><关系运算符><算术表达式>
<关系运算符>→>|<|>=|<=|==|!=

Sample语言中没有布尔量,和C语言一样,非0就表示真,因此本质上仍然是数值运

算。布尔表达式的运算对象是关系表达式，运算符有非(!)、与(&&)和或(‖)，布尔表达式的文法定义如下：

<布尔表达式>→<布尔项>‖<布尔表达式>|<布尔项>
<布尔项>→<布尔因子>&&<布尔项>|<布尔因子>
<布尔因子>→<算术表达式>|<关系表达式>|!<布尔表达式>

和C语言一样，Sample语言中赋值运算也作为一个运算符，优先级最低，含义是把赋值号右边的表达式的值赋值给左边的一个标识符。赋值表达式的文法如下：

<赋值表达式>→<标识符>=<表达式>

2. Sample语言中的语句

Sample语言的语句分为声明语句和执行语句两类。声明语句又分为变量声明、常量声明和函数声明。变量声明主要是提前声明程序中使用的变量的属性，如变量名、变量类型等，变量声明可以给变量赋初值；常量声明定义常量的值；函数声明是在主程序前事先声明本程序中所有使用的函数的属性，如函数名、返回值类型、参数个数及类型。Sample语言的常量、变量和函数声明的文法定义如下：

<语句>→<声明语句>|<执行语句>
<声明语句>→<值声明>|<函数声明>|ε
<值声明>→<常量声明>|<变量声明>
<常量声明>→const <常量类型> <常量声明表>
<常量类型>→int|char|float
<常量声明表>→<标识符>=<常量>;|<标识符> = <常量>,<常量声明表>
<变量声明>→<变量类型> <变量声明表>
<变量声明表>→<单变量声明>;|<单变量声明>,<变量声明表>
<单变量声明>→<变量>|<变量>=<表达式>
<变量类型>→int|char|float
<函数声明>→<函数类型> <标识符>(<函数声明形参列表>);
<函数类型>→int|char|float|void
<函数声明形参列表>→<函数声明形参>|ε
<函数声明形参>→<变量类型>|<变量类型>,<函数声明形参>

此处我们规定，函数声明的形参列表只声明形参的类型，不声明变量，以示和函数定义区分。

Sample语言中的执行语句包括数据处理语句、控制语句和复合语句。数据处理语句包括赋值语句和函数调用语句，主要对数据进行处理；控制语句主要有if语句、while语句、do…while语句、for语句和return语句。复合语句是由一对花括号括起来的一个或多个语句。Sample语言中的执行语句的文法定义如下：

<执行语句>→<数据处理语句>|<控制语句>|<复合语句>
<数据处理语句>→<赋值语句>|<函数调用语句>
<赋值语句>→<赋值表达式>;
<函数调用语句>→<函数调用>;
<控制语句>→<if语句>|<for语句>|<while语句>|<do while语句>|<return语句>
<复合语句>→{<语句表>}
<语句表>→<语句>|<语句><语句表>
<if语句>→if (<表达式>)<语句>|if (<表达式>) <语句> else <语句>

<for 语句>→for (<表达式>;<表达式>;<表达式>)<循环语句>
<while 语句>→while (<表达式>)<循环语句>
<do while 语句>→do <循环用复合语句> while (<表达式>);
<循环语句>→<声明语句>|<循环执行语句> | <循环用复合语句>
<循环用复合语句>→{<循环语句表>}
<循环语句表>→<循环语句>|<循环语句><循环语句表>
<循环执行语句>→<循环用 if 语句>|< for 语句>|< while 语句>|< do while 语句>|< return 语句>|< break 语句>|< continue 语句>
<循环用 if 语句>→if (<表达式>)<循环语句>| if (<表达式>) <循环语句> else <循环语句>
< return 语句>→return;| return <表达式>;
< break 语句>→break;
< continue 语句>→continue;

3. Sample 语言中的函数

在 Sample 语言中,函数分为函数声明、函数定义和函数调用。在前面声明语句部分已介绍了函数声明语句的文法,在表达式部分也介绍了函数调用的文法。函数定义就是把函数名和一段代码对应起来。函数定义包括函数返回值类型、函数名和形参列表以及对应的语句表。Sample 语言的函数定义的文法定义如下:

<函数定义>→<函数类型> <标识符>(<函数定义形参列表>) <复合语句>
<函数定义形参列表>→<函数定义形参>|ε
<函数定义形参>→<变量类型> <标识符>|<变量类型> <标识符>,<函数定义形参>

4. Sample 语言的程序

程序用来定义一个合法的 Sample 语言程序的结构,由零到多个声明语句、main()函数、零到多个函数顺序构成。每个程序必须有一个 main()函数,它是程序的入口。其文法定义为:

<程序>→<声明语句> main()<复合语句><函数块>
<函数块>→<函数定义><函数块>|ε

3.3 自上而下的语法分析

自上而下的语法分析方法就是对任何输入串(由 token 串构成的源程序),试图用一切可能的办法,从文法开始符号(根结点)出发,自上而下地为输入符号串建立一棵语法树。或者说,为输入串寻找一个最左推导。这个过程的主要难点在于:在替换一个非终结符时,如果一个非终结符有多个候选式,到底用哪个候选式来替换?我们希望每一次候选式的选择都是确定的,称为确定的自上而下的分析。自上而下分析的过程本质上是一种试探过程,是反复使用不同产生式谋求匹配输入串的过程,在试探过程中可能会出现一些问题,只有解决了这些问题,才能进行确定的分析。

3.3.1 自上而下分析方法中的问题探究

1. 确定的自上而下分析面临的问题

首先看两个例子。

例 3.9 假定有关系表达式文法 G[< REXPR >]:

(1) < REXPR >→x < ROP > y

(2) < ROP >→>= | >

构造输入符号串 x>y 的语法树。

我们希望从< REXPR >开始推导建立语法树，使其叶子结点从左到右匹配输入符号串 x>y。

首先对文法的开始符号建立根结点< REXPR >，输入指针指向输入串的第一个符号 x，用< REXPR >的产生式(此处只有一条)向下推导，语法树如图 3.4(a)所示，此时 x 已经获得匹配。接下来输入指针后移，希望用第二个子结点< ROP >去匹配输入符号>。< ROP >有两个候选式，假定先选择使用候选式< ROP >→>=进行推导，语法树如图 3.4(b)所示，此时输入串中的 x>都已匹配。输入指针后移指向下一个输入符号 y，此时 y 与语法树中< ROP >的第二个子结点=不匹配，导致分析失败。但此时并不能断定给定的符号串不能建立语法树。因为< ROP >有两个候选式，现在只是选择了其中一个使得分析失败，也许使用另一个候选式能够建立正确的语法树。所以应该回退(回溯)，重新选择< ROP >的其他候选式继续分析。

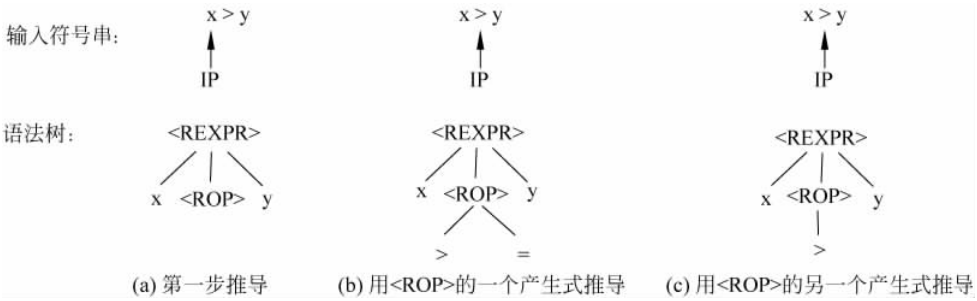


图 3.4 自上而下语法树举例 1

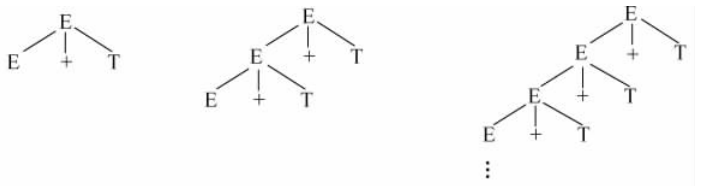
此时应把用< ROP >的第一个候选式产生的子树注销，将输入指针退回指向>。对< ROP >重新选用候选式< ROP >→>进行试探，如图 3.4(c)所示。输入串中当前符号>得到匹配，输入指针向后移动指向下一个输入符号 y。在< REXPR >的第二个子结点< ROP >完成匹配后，接着希望用< REXPR >的第三个子结点去匹配输入符号 y，< REXPR >的第三个子结点 y 正好与当前输入符号 y 匹配，推导成功，为输入符号串 x>y 建立了语法树，证明 x>y 是文法的一个句子。

例 3.10 设某语言的算术表达式文法为 G3.2，试建立输入串 i * i+i 的语法树。

按照自上而下分析方法，希望从 E 开始推导对输入串建立语法树。

首先建立根结点 E，再选用 E 的候选式 E→E+T 向下推导，得到的语法树如图 3.5(a)所示。由于采用最左推导，最左子结点仍然是一个非终结符，必须选用一个候选式继续向下扩展语法树，如果再选用 E 的候选式 E→E+T 向下推导，得到的语法树如图 3.5(b)所示；此时最左子结点仍然是一个非终结符，必须选用一个候选式继续向下推导，再选用 E→E+T，如图 3.5(c)所示。对非终结符 E 的最左推导会使语法树无休止地延伸，使分析过程陷入无限循环。

例 3.9 和例 3.10 中主要出现了两个问题，给确定的自上而下分析带来了困难。



(a) 第一步推导　(b) 第二步进行最左推导　(c) 继续进行最左推导

图 3.5　自上而下语法树举例 2

(1) 回溯(Back Track),导致分析器不稳定。当文法中存在形如 $A\rightarrow\alpha\beta_1 \mid \alpha\beta_2$ 的产生式,即某非终结符存在多个候选式的前缀相同(称为公共左因子,或左因子),则可能造成虚假匹配(即当前的匹配可能是暂时的),使得在分析过程中可能需要进行大量回溯(如例 3.9)。由于大多数编译程序的语法和语义工作是同时进行的,由于回溯,需要把已做的一些语义工作推倒重来。这样既麻烦又费时,同时使得分析器很难报告输入串出错的确切位置,也使分析器的工作过程很不稳定,时空效率都得不到保证。试探与回溯是一种穷尽一切可能的办法,效率低,代价高,在实践中的价值不大,所以使用自上而下分析时,要设法消除回溯。

(2) 左递归(Left Recursion),导致分析过程无限循环。由于文法中存在形如 $A\rightarrow A\alpha$ 的产生式(称为左递归),分析过程又使用最左推导,就会使分析过程陷入无限循环(如例 3.10)。因为当试图用 A 的右部去匹配输入串时会发现,在没有读入任何输入符号的情况下,又要求用 A 的右部去进行新的匹配。因此,使用自上而下分析时,文法应该不含左递归。

2. 回溯的消除

回溯产生的根本原因在于某个非终结符的多个候选式存在公共左因子,如非终结符 A 的产生式如下:

$$A \rightarrow \alpha\beta_1 \mid \alpha\beta_2$$

如果输入串中待分析的字前缀也为 α,此时选用 A 的哪个候选式以寻求输入串的匹配就难以确定,可能会导致回溯。因此要想进行确定的分析,必须保证文法 G 的每个非终结符的多个候选式均不含公共左因子,当使用它去匹配输入串时,能够根据它所面临的输入符号准确地指派一个候选式去进行匹配,无须试探;这时若匹配失败,则意味着输入串不是该文法的句子。

那么,如何将文法改造成符合上述要求的文法呢?改造的方法是提取公共左因子。设文法中关于 A 的候选式为:

$$A \rightarrow \delta\beta_1 \mid \delta\beta_2 \mid \cdots \mid \delta\beta_n \mid \gamma_1 \mid \gamma_2 \mid \cdots \mid \gamma_m$$(其中每个 $\gamma_i(i = 1,2,\cdots,m)$ 不以 δ 开头)

那么,可以把公共的 δ 提取出来,A 的候选式改写为:

$$A \rightarrow \delta A' \mid \gamma_1 \mid \gamma_2 \mid \cdots \mid \gamma_m$$

$$A' \rightarrow \beta_1 \mid \beta_2 \mid \cdots \mid \beta_n$$

可以证明改造后的文法和改造前的文法是等价的,因为从 A 出发,两个文法推导出的符号串是相同的。

利用改造后的文法就可以进行确定的分析了。

例 3.11 条件(if)语句的文法有两个候选式：

$$<\text{IFS}> \rightarrow \text{if B then } S_1 \text{ else } S_2$$

$$<\text{IFS}> \rightarrow \text{if B then } S_1$$

在对形如 if (a>b) then x=3 的输入符号串进行分析时，当读入输入符号 if 时，就不能立刻确定用哪个候选式去推导。通过提取公共左因子改造文法，得到：

$$<\text{IFS}> \rightarrow \text{if E then } S_1 \text{ P}$$

$$\text{P} \rightarrow \text{else } S_2 \mid \varepsilon$$

使用改造后的文法进行分析，当读入输入符号 if 时，就可以直接使用 if 语句的产生式向下分析，当 S_1 匹配成功后，根据下一个输入符号是不是 else，再决定是否选择 P 的候选式进行匹配。

3. 左递归的消除

若文法 G 中存在某个非终结符 A，对某个文法符号序列 α 存在推导 $A \overset{+}{\Rightarrow} A\alpha$，则称文法 G 是左递归的。左递归有直接左递归和间接左递归两类。若文法 G 中有形如 $A \rightarrow A\alpha$ 的产生式，则称该产生式对 A 直接左递归(Direct Left Recursion)。若文法 G 的产生式中没有形如 $A \rightarrow A\alpha$ 的产生式，但是 A 经过有限步推导可以得到 $A \overset{+}{\Rightarrow} A\alpha$，则称文法 G 间接左递归(Indirect Left Recursion)。自上而下语法分析在处理左递归文法时会陷入无限循环，因此，需要消除文法中出现的左递归。

1) 消除文法的直接左递归

消除产生式中的直接左递归是通过对产生式进行改造来实现的。根据定义，直接左递归存在于一个产生式中，将各个产生式改造后使各个非终结符不含左递归。假定关于非终结符 A 的候选式为

$$A \rightarrow A\alpha \mid \beta$$

其中，$\alpha, \beta \in (V_T \cup V_N)^*$，$\beta$ 不以 A 开头，那么，可以把 A 的候选式改写为如下的非直接左递归形式：

$$A \rightarrow \beta A'$$

$$A' \rightarrow \alpha A' \mid \varepsilon(\varepsilon \text{ 为空字})$$

改造后的文法形式和原来的文法形式是等价的，因为从 A 推导出的符号串是相同的。消除直接左递归，实际上是把直接左递归文法改成直接右递归文法，在最左推导中就不会陷入死循环。

例 3.12 文法 G3.2 是含有左递归的文法，消除左递归后，得到如下文法：

(1) $E \rightarrow TE'$

(2) $E' \rightarrow +TE' \mid \varepsilon$

(3) $T \rightarrow FT'$

(4) $T' \rightarrow *FT' \mid \varepsilon$

(5) $F \rightarrow (E) \mid i$ (G3.3)

文法 G3.3 就是在后续章节中经常使用的不含左递归的算术表达式的文法。

将上述结果推广到更一般的情形，假定文法中关于 A 的产生式如下：

$$A \rightarrow A\alpha_1 \mid A\alpha_2 \mid \cdots \mid A\alpha_m \mid \beta_1 \mid \beta_2 \mid \cdots \mid \beta_n$$

其中，$\alpha_i (i=1,2,\cdots,m)$ 都不是 ε，$\beta_j (j=1,2,\cdots,n)$ 均不以 A 开头。可以把 A 的产生式改写

为如下等价形式：

$$A \rightarrow \beta_1 A' \mid \beta_2 A' \mid \cdots \mid \beta_n A'$$
$$A' \rightarrow \alpha_1 A' \mid \alpha_2 A' \mid \cdots \mid \alpha_m A' \mid \varepsilon$$

2) 消除文法中的间接左递归

有些文法中的左递归并不是直接的，如文法 G3.4 中的 S 不是直接左递归的，但也是左递归的，因为存在推导 S⇒Ac⇒Bbc⇒Sabc。

(1) S→Ac|c

(2) A→Bb|b

(3) B→Sa|a (G3.4)

对于文法中的间接左递归，可以采用先代入再消除直接左递归的方法。

例 3.13 消除文法 G3.4 中的左递归。

(1) 代入。

将产生式 B→Sa|a 代入 A→Bb|b，有 A→(Sa|a)b|b

即 A→Sab|ab|b

将产生式 A→Sab|ab|b 代入 S→Ac|c，有 S→(Sab|ab|b)c|c

即 S→ Sabc|abc|bc|c

(2) 消除 S 的直接左递归后，得到：

S→abcS′|bcS′|cS′

S′→abcS′|ε

A→Bb|b

B→Sa|a

3.3.2 LL(1)文法

那么，是否每个非终结符 A 的多个候选式不存在公共左因子，文法也不含左递归，就可以进行确定的自上而下的语法分析呢？

考虑文法：

(1) S→Ac|Be

(2) A→db|b

(3) B→da|a

现要求对输入符号串 dbc 进行分析。

分析开始时，当要求用 S 的候选式匹配 d 时，虽然 S 的两个候选式没有公共左因子，仍不能准确地选取 S 的候选式，也就是说，不能进行确定的分析。

根据上面的讨论，并非所有没有左递归、没有公共左因子的文法都能进行确定的自上而下的分析。要对一个文法进行不带回溯的确定的自上而下的分析必须满足哪些条件呢？为方便叙述，首先给出两个概念。

1. First 集的概念及其计算方法

设文法 G 不含左递归，G 的文法符号串 α 的首终结符集 First(α)(α∈$(V_T \cup V_N)^*$)定义为

$$First(\alpha) = \{a \mid \alpha \overset{*}{\Rightarrow} a\cdots, a \in V_T\}$$

若 $\alpha \overset{*}{\Rightarrow} \varepsilon$，则规定 $\varepsilon \in First(\alpha)$。换句话说，First(α)是 α 的所有可能推导出的第一个终结符或可能的 ε，其中 α 可以是文法符号、ε 或候选式，或候选式的一部分。

由这个定义，可以根据文法 G 的候选式形式的不同，对 First 集的计算进一步细化，如算法 3.1 所示。

算法 3.1 计算某个候选式的 First 集

输入：文法 G

输出：各个候选式的 First 集

步骤：

(1) 若产生式形如 $A \rightarrow a\alpha, a \in V_T$，则 $First(A \rightarrow a\alpha) = \{a\}$。

(2) 若产生式形如 $A \rightarrow \varepsilon$，则把 ε 加入其中，$First(A \rightarrow \varepsilon) = \{\varepsilon\}$。

(3) 若产生式形如 $A \rightarrow X\alpha, X \in V_N$，则把 First(X)中非 ε 元素(记为 $First(X) \backslash \varepsilon$)加入 $First(A \rightarrow X\alpha)$中。

(4) 若有产生式形如 $A \rightarrow X_1X_2X_3 \cdots X_k\alpha$，其中 $X_1, X_2, X_3, \cdots, X_k \in V_N$。则

① 当 $X_1X_2X_3 \cdots X_i \overset{*}{\Rightarrow} \varepsilon (1 \leqslant i \leqslant k)$时，则把 $First(X_{i+1} \cdots X_k)$的所有非 ε 元素加入 $First(A \rightarrow X_1X_2X_3 \cdots X_k\alpha)$中。

② 当 $X_1X_2X_3 \cdots X_k \overset{*}{\Rightarrow} \varepsilon$ 时，则把 First(α)加入 $First(A \rightarrow X_1X_2X_3 \cdots X_k\alpha)$中。

若要求每个文法符号的 First 集，根据上述定义可知：

(1) 若文法符号 $A \in V_T$，则 $First(A) = \{A\}$。

(2) 若文法符号为非终结符 $A(A \in V_N)$，求 First 集的方法是将非终结符 A 的每个候选式的 First 集都加入到 First(A)中，即 $First(A) = \bigcup_{\forall \alpha} First(A \rightarrow \alpha)$。

例 3.14 求文法 G3.3 中各个候选式和各个非终结符的 First 集。

解：各个候选式的 First 集为

$First(E \rightarrow TE') = First(T) = First(F) = \{(, i\}$

$First(E' \rightarrow +TE') = \{+\}$

$First(E' \rightarrow \varepsilon) = \{\varepsilon\}$

$First(T \rightarrow FT') = First(F) = \{(, i\}$

$First(T' \rightarrow *FT') = \{*\}$

$First(T' \rightarrow \varepsilon) = \{\varepsilon\}$

$First(F \rightarrow (E)) = \{(\}$

$First(F \rightarrow i) = \{i\}$

各个非终结符的 First 集为

$First(E) = First(T) = First(F) = \{(, i\}$

$First(E') = \{+, \varepsilon\}$

$First(T) = \{(, i\}$

$First(T') = \{*, \varepsilon\}$

$First(F) = \{(, i\}$

根据 First 集的定义可知，如果非终结符 A 的各个候选式的首终结符集两两不相交，即

对 A 的任何两个不同的候选式 α_i 和 α_j 有

$$First(\alpha_i) \cap First(\alpha_j) = \Phi$$

那么在分析时,当非终结符 A 面临输入符号 a 时,需要选择 A 的候选式进行匹配,就可以选择 First 集中包含 a 的候选式去进行推导。

那么,如果 A 的候选式的 First 集都不包含 a,是否就一定是错误呢?请思考。

2. Follow 集的概念及其计算方法

如果给定的文法不含左递归,每个非终结符的候选式的首终结符集也两两不相交,是否就一定能进行有效的自上而下的分析呢?如果某个候选式的首终结符集含有 ε,就比较复杂。

例 3.15 使用文法 G3.3 对输入串 i+i 进行分析。

首先从开始符号 E 出发利用推导去匹配输入串,假定用输入指针 IP 来指向当前待匹配的符号。分析开始时,IP 指向 i,由于 E 只有一个候选式,且 i∈First(TE′),使用 E→TE′向下推导,建立语法树如图 3.6(a)所示。现在要从 T 出发,IP 仍指向 i,T 只有一个候选式,且 i∈First(FT′),使用 T→FT′向下推导,语法树扩展如图 3.6(b)所示。又从 F 出发,IP 仍指向 i,且 i∈First(i),使用 F→i 向下推导,使输入串的第一个符号 i 得到匹配,语法树扩展如图 3.6(c)所示。IP 后移指向+,现在希望从 T′出发去匹配+,由于+不属于 T′的任一候选式的首终结符集,无法匹配。但由于 ε∈First(T′),可以使用 T′→ε 进行自动匹配(此时 IP 指针不变),语法树扩展如图 3.6(d)所示。接下来希望从 E′出发匹配+,由于+∈First(+TE′),所以语法树扩展如图 3.6(e)所示,分析结束时语法树如图 3.6(f)所示。

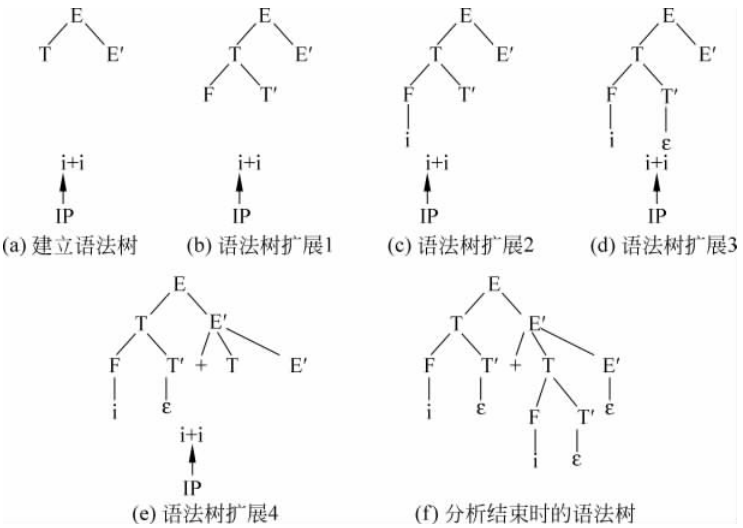


图 3.6 例 3.15 的语法分析过程

在本例的分析中,是否意味着当非终结符 A 面临某个输入符号 a 时,只要 a 不属于 A 的任一候选式的首终结符集,而 A 的某一候选式的首终结符集含有 ε,就可以使用 ε 进行自动匹配呢?仔细分析一下,在分析到图 3.6(c)时,+∉First(T′),但 T′的 First 集中含有 ε,只有当+属于 E′的某个候选式的首终结符集时才能使用 ε 产生式匹配。否则,表示出现了一个语法错误,说明不能构成句子,这就没有必要用 ε 去替换。

由此定义非终结符 A 的后随符号集 Follow(A)：假定 S 是文法 G 的开始符号，对 G 的任何非终结符 A

$$\text{Follow(A)} = \{a \mid S \overset{*}{\Rightarrow} \cdots Aa \cdots, a \in V_T\}$$

若 $S \overset{*}{\Rightarrow} \cdots A$，则规定 $\# \in \text{Follow(A)}$。也就是说，Follow(A)是所有句型中出现在紧接 A 之后的终结符号或'#'。#是输入符号串的结束标记。

利用 Follow 集的定义，使用 ε 产生式进行自动匹配的过程是：当非终结符 A 面临输入符号 a，且 a 不属于 A 的任一候选式的首终结符集，但 A 的某个候选式的首终结符集含有 ε 时，只有当 $a \in \text{Follow(A)}$，才能用 ε 产生式进行自动匹配。

文法 G 的每个非终结符 A 的 Follow 集可使用算法 3.2 来计算。

算法 3.2 计算文法 G 的非终结符 A 的 Follow 集

输入：文法 G

输出：非终结符 A 的 Follow 集

步骤：

(1) 如果 A 是开始符号，$\# \in \text{Follow(A)}$。

(2) 若有产生式 $B \to \alpha A a \beta$，$a \in V_T$，把 a 加入 Follow(A)中。

(3) 若有产生式 $B \to \alpha A X \beta$，$X \in V_N$，把 First(Xβ)中非 ε 元素(记为 First(Xβ)\ε)加入 Follow(A)中。

(4) 若 $B \to \alpha A$，或 $B \to \alpha A \beta$ 且 $\beta \overset{*}{\Rightarrow} \varepsilon$，把 Follow(B)加入 Follow(A)中。

(5) 对每一个非终结符，浏览每个产生式，连续使用上述规则，直到 A 的 Follow 集不再增大为止。

例 3.16 求文法 G3.3 中各个非终结符的 Follow 集。

解：首先改写文法 G3.3，使每个候选式单独一行，并进行编号。

(1) $E \to TE'$

(2) $E' \to +TE'$

(3) $E' \to \varepsilon$

(4) $T \to FT'$

(5) $T' \to *FT'$

(6) $T' \to \varepsilon$

(7) $F \to (E)$

(8) $F \to i$

然后根据上述算法求各个非终结符的 Follow 集。

(1) $\text{Follow(E)} = \{\#,)\}$

说明：E 是开始符号，应用规则 1；根据产生式 7，应用规则 2。

(2) $\text{Follow(E')} = \text{Follow(E)} \cup \text{Follow(E')} = \{\#,)\}$

说明：根据产生式 1，应用规则 4；根据产生式 2，应用规则 4。

(3) $\text{Follow(T)} = \text{Follow(E')} \cup (\text{First(E')} \backslash \varepsilon) \cup \text{Follow(E)} \cup (\text{First(E')} \backslash \varepsilon) = \{\#,), +\}$

说明：根据产生式 2,3，应用规则 4；根据产生式 2，应用规则 3；根据产生式 1,3，应用

规则4；根据产生式1,应用规则3。

(4) Follow(T′) =Follow(T)∪Follow(T′) ={#,),+}

说明：根据产生式4,应用规则4；根据产生式5,应用规则4。

(5) Follow(F) =(First(T′)\ ε)∪Follow(T) ∪(First(T′)\ ε)∪Follow(T′) ={*,#,),+}

说明：根据产生式4,应用规则3；根据产生式4,6,应用规则4；根据产生式5,应用规则3；根据产生式5,6,应用规则4。

3. LL(1)文法的条件

通过上述分析,一个文法要进行不带回溯的确定的自上而下分析必须满足：

(1) 文法不含左递归。

(2) 文法中每个非终结符A的各个候选式的首终结符集两两不相交。即,若

$$A \rightarrow \alpha_1 \mid \alpha_2 \mid \cdots \mid \alpha_n$$

则

$$First(\alpha_i) \cap First(\alpha_j) = \Phi \quad (i \neq j)$$

(3) 对文法的每个非终结符A,若它的某个候选式的首终结符集包含ε,则

$$First(A) \cap Follow(A) = \Phi$$

如果一个文法G满足上述三个条件,就称文法G是LL(1)文法。

这里LL(1)中第一个L表示从左到右扫描输入串,第二个L表示最左推导,1表示分析时每一步只需向前查看一个符号。

LL(1)文法可以用来描述大多数高级语言的语法结构,二义文法不是LL(1)的。

例3.17 判断下述文法是否是LL(1)文法。

S→aAS|b

A→bA|ε

解：

(1) 该文法不含左递归,满足条件1。

(2) First(S→aAS)={a}　First(S→b)={b}

First(A→bA)={b}　First(A→ε)={ε}

对于S和A,它们各自的候选式的首终结符集都不相交,满足条件2。

(3) 对于非终结符A,含有ε产生式,求其Follow集,则

Follow(A)=First(S)={a,b}

Follow(A)∩ First(A) ≠Φ

不满足条件3,因此该文法不是LL(1)文法。

针对给定的LL(1)文法,对输入串进行有效的无回溯的自上而下分析的过程是：假设要用非终结符A进行匹配,面临的输入符号为a,A的候选式为

$$A \rightarrow \alpha_1 \mid \alpha_2 \mid \cdots \mid \alpha_n$$

(1) 若$a \in First(\alpha_i)$,则指派α_i去执行匹配任务。

(2) 若a不属于任何一个候选式的首终结符集,则

① 若ε属于某个$First(\alpha_i)$,且$a \in Follow(A)$,则让A与ε自动匹配。

② 否则,a的出现是一种语法错误。

这样，根据 LL(1)文法的条件，每一步的分析都是确定的。

当给定一个 LL(1)文法时，如何实现它的分析程序呢？接下来将介绍两种确定的自上而下的语法分析方法：递归下降分析方法和预测分析方法。两种方法各有优缺点，使用递归下降分析方法编写语法分析程序，书写简单，易于理解，但只有实现分析程序所使用的高级语言支持递归过程才有意义；预测分析方法是另一种有效的自上而下的语法分析方法，需要根据文法计算和存储分析表，它显式地维护一个栈结构，而不是像递归下降分析方法中通过递归调用来隐式地维护栈结构；当文法符号较多时，存储分析表需要占用很大的空间，但利用表来控制分析的过程是固定的，因此预测分析程序可以自动生成。

3.3.3 递归下降分析方法

1. 递归下降分析方法的基本思想

对一个 LL(1)文法，可以构造一个不带回溯的自上而下的分析程序，这个分析程序是由一组递归函数(或子程序)组成的，每个函数(或子程序)对应文法的一个非终结符。这样的一个分析程序称为递归下降分析器(Recuisive Descent Parser)。如果能用某种高级语言写出所有的递归函数，也就可以用这个语言的编译系统来产生整个分析程序。

递归下降分析是直接以程序的方式模拟产生式产生语言的过程。它的基本思想是：为每个非终结符构造一个函数，每个函数的函数体按非终结符的候选式分情况展开，遇到终结符就进行比较，看是否与输入符号串匹配；遇到非终结符就调用该非终结符对应的函数。分析过程从调用文法开始符号对应的函数开始，直到所有非终结符都展开为终结符并得到匹配为止。如果分析过程中达到这一步则表明分析成功，否则表明输入符号串有语法错误。由于文法是递归定义的，因此函数也是递归的。

对应于每个非终结符 U(假定 U 的候选式为 $U\rightarrow U_1|U_2|\cdots|U_n$)的函数完成如下两项任务。

(1) 根据输入符号决定使用 U 的哪个候选式进行分析。如果当前面临的输入符号 token 在候选式 U_i 的 First 集中，即 $token\in First(U_i)$，则选择使用 U_i 进行分析。如果当前输入符号 token 不在任何一个 $First(U_i)$中，但有 $\varepsilon\in First(U_i)$，则判断该输入符号是否在 Follow(U)中，如果在，ε 将被使用。

可形式化地描述为：若非终结符 U 的候选式为 $U\rightarrow U_1|U_2|\cdots|U_n$，其递归函数原型如下。

```
void U( ){
    token = GetNextToken();      //从输入的 token 串中读取一个符号到 token 中,输入指针下移
    if (token∈First (U1) ) U1();
    else if (token∈First (U2) ) U2();
       …
    else if ( (ε∈First(U)) && (token∈Follow(U)) )
       …;
    else error();                //没有找到匹配项,出错处理
}
```

即 U 有 n 个候选式：当输入符号在 U_1 的 First 集中，就选择第一个候选式进行处理；当输入符号在 U_2 的 First 集中，就选择第二个候选式进行处理；如果输入符号不在 U 的任

何一个候选式的 First 集中,查看是否有 ε 产生式,如果有,判断面临的符号是否在 U 的 Follow 集中,否则就认为出现了语法错误。

(2) 对应于某个候选式 U_i(假定 U_i 是形如 $X_1X_2\cdots X_n$ 的候选式)的处理 $U_i()$,就是通过顺序处理该候选式 U_i 的每个符号 X_i 来完成其功能的。每个 X_i 的处理要根据 X_i 是终结符或非终结符来确定:若 X_i 是非终结符,就调用该非终结符对应的函数 $X_i()$;若 X_i 是终结符,就直接调用 match()函数进行比较,如果 X_i 和读入的符号匹配,就读入下一个输入符号,如果不匹配,则报告错误。

可形式化地描述为:对于 U 的每个候选式 $U_i=X_1X_2\cdots X_n$ 的处理过程是依次处理每个右部符号串 $X_1X_2\cdots X_n$,其处理过程的原型如下。

```
void Ui( )
{
  if (X1∈VN)                          //处理 X1
       X1( );
  else
       match(token );
  if (X2∈VN)                          //处理 X2
       X2( );
  else
       match(token );
  …
  if (Xn∈VN)                          //处理 Xn
       Xn( );
  else
       match(token );
}
```

match()函数的功能是判断当前输入符号 token 是否与文法推导中出现的符号 X_i 相等,若相等,读取下一个输入符号,否则出错。

```
void match (char * token)
  {
    if (Xi == token) {                //与当前输入符号相同,即匹配
          token = GetNextToken( );    //取下一个符号到 token 中
          return;                     //匹配时直接返回
      }
    else error ( );                   //不匹配,进行出错处理
  }
```

例 3.18 编写文法 G3.3 对应的递归下降分析程序。

(1) $E\rightarrow TE'$

(2) $E'\rightarrow +TE'\mid\varepsilon$

(3) $T\rightarrow FT'$

(4) $T'\rightarrow *FT'\mid\varepsilon$

(5) $F\rightarrow(E)\mid i$ (G3.3)

文法 G3.3 的递归下降分析程序如表 3.2 所示。对非终结符 E,只有一个候选式为 $E\rightarrow TE'$,处理函数就是顺序处理右部的两个符号 T 和 E′,两个符号都是非终结符,所以直接调

用这两个非终结符对应的函数 T()和 E′()即可，函数 E()就是根据这一原则设计的。对于 E′有两个候选式：第一个候选式的首终结符为＋，第二个候选式为 ε。这就是说，当 E′面临输入符号＋时就进入第一个候选式工作，而当面临任何其他输入符号时，E′就自动认为获得了匹配。递归函数 E′()就是根据这一原则设计的。

表 3.2 文法 G3.3 的递归下降分析程序

对非终结符 E，候选式为 E→TE′：

```
void E( )
{
     T( );
     E'( );
}
```

对非终结符 T，候选式为 T→FT′

```
void T( )
  {
    F( );
    T'( );
  }
```

对非终结符 E′，候选式为 E′→＋TE′|ε

```
void E'( )
  {
    if (token == '+')
      {
        match ('+');
        T( );
        E'( );
      }
}
```

对非终结符 F，候选式为 F→i|(E)

```
void F( )
  {
    if (token == '(')
      {
        match ('(');
        E( );
        if (token == ')')  match (')');
        else error ( );
      }
    else if (token == 'i')  match ('i');
      else error ( );
  }
```

对非终结符 T′，候选式为 T′→＊FT′|ε

```
void T'( )
  {
    if (token == '*')
      {
        match ('*');
        F( );
        T'( );
      }
  }
```

对于规模比较小的语言，递归下降分析法是很有效的方法，它简单灵活，容易构造，特别适合手工构造语法分析器。其缺点是程序与文法直接相关，对文法的任何改变均需对程序进行相应的修改，另外，由于递归调用多，导致程序运行速度慢，占用空间多。尽管这样，它还是许多高级语言，如 Pascal、C 等编译系统常常采用的语法分析方法。

递归下降分析器也可以用状态转换图（又称语法图）来设计。对于语法分析器，状态转换图的画法是：每个非终结符都对应一个状态转换图，边上的标记是终结符和非终结符。对每个非终结符 A 执行如下操作。

（1）创建一个开始状态和一个终态。

（2）对每个产生式 $A\rightarrow X_1X_2\cdots X_n$，创建一条从开始状态到终止状态的路径，边上的标记分别为 $X_1, X_2, \cdots, X_n$。

文法 G3.3 的状态转换图如图 3.7 所示。在状态转换图上，标有终结符的转换意味着如果该终结符与当前输入符号相同，就进行相应的状态转换；标有非终结符 A 的转换就是对与 A 对应的函数的调用。

根据状态转换图很容易写出递归的语法分析程序。开始，语法分析器进入开始符号的

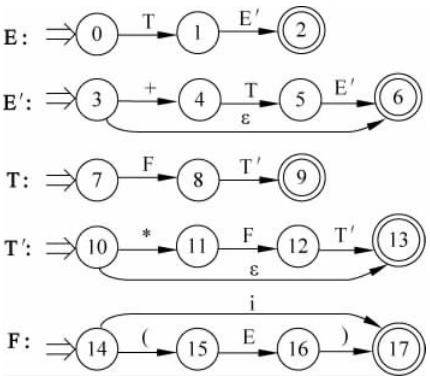
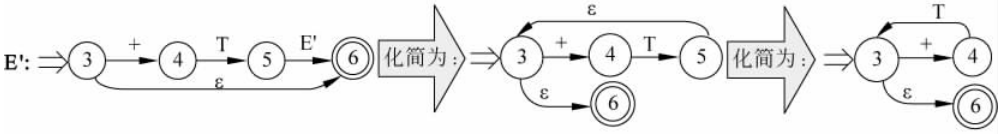


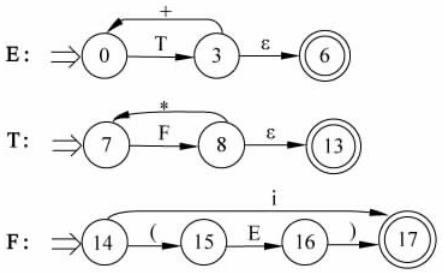
图 3.7　文法 G3.3 的状态转换图

状态转换图的开始状态(如图 3.7 中的状态 0),输入指针指向输入符号串的第一个符号。如果经过一些动作后,语法分析器进入某个状态 s,如果在状态转换图上状态 s 到 t 的边上标有终结符 a,当前输入符号正好是 a,则语法分析器读入该符号并将输入指针向右移动一位指向下一个输入符号,语法分析器进入状态 t;如果边上的标记为非终结符 A,则语法分析器进入 A 的状态转换图的初始状态,不读入任何输入符号,即不移动输入指针,一旦语法分析器到达 A 的终结状态时,则立刻返回状态 t;如果边上标有 ε,语法分析器就直接进入状态 t,而不移动输入指针。根据图写出的递归函数与表 3.2 的函数相同。

可以对图 3.7 所示的状态转换图进行化简。如首先对 E′的图进行化简,化简的方法是:在 E′中,状态 5 到状态 6 标记为 E′,和开始状态 3 的标记相同,因此可以直接从状态 5 画弧指向状态 3,不读入任何符号,得到图 3.8 (a)中的中间的状态转换图;从状态 5 不读入任何符号到状态 3,则可以直接从状态 4 到状态 3,标记为 T,得到图 3.8 (a)中的最右边的状态转换图;再将 E′的状态转换图代入到 E 中,即得到 E 的化简后的状态转换图,同理可得到 T′、T 的化简后的状态转换图,最终的状态转换图如图 3.8 所示。



(a) 文法G3.3中E'的状态转换图的化简



(b) 文法G3.3中E，T，F的状态转换图

图 3.8　文法 G3.3 的化简的状态转换图

图 3.8 中状态转换图的工作是以一种相互递归的方式进行的；因此，每个状态转换图的作用就如同一个递归函数。根据简化后的状态转换图，写出递归函数如下。

```
void E( )
{
  T( );
  while (token == '+')
    {
      match ('+');
      T();
    }
}
```

```
void T()
{
    F();
  while (token == '*')
    {
      match ('*');
      F();
    }
}
```

F 的状态转换图没有变，所以其递归函数也不变。这样就可以从前面的递归下降分析程序中删除了函数 E′()和 T′()，减少了函数的数量。

2. 递归下降分析器的设计

只要一个语言的语法规则的文法描述符合 LL(1)文法的要求，那么该语言的语法分析程序就可以使用递归下降分析方法来实现，而且可以保证分析是确定的。本节主要介绍用递归下降分析方法手工实现 Sample 语言的语法分析器。

Sample 语言的语法成分包括：

(1) 程序只有一个 main()函数，它是整个程序的入口，没有返回值，也不带参数。

(2) 程序中可以定义 0 到多个函数，可以带 0 到多个形式参数，需要定义返回值类型，没有返回值就用 void，函数定义必须放在 main()函数之后，它必须在 main()函数之前声明。

(3) 带类型的简单变量、常量的声明，可以是全局和局部定义的，变量可以赋初值；全局常量和变量在 main()函数之前声明，局部常量和变量在各个复合语句内声明。

(4) 函数声明只能是全局的，必须在 main()函数之前，可以带 0 个或多个形式参数，需要定义返回类型；函数声明和函数定义的返回值类型、参数个数和参数类型必须匹配。

(5) 表达式类型有算术表达式、关系表达式、布尔表达式和赋值表达式。

(6) 语句有表达式语句(主要是赋值语句、函数调用语句等)，if 语句，while 语句，do…while 语句，for 语句，复合语句，return 语句，continue 语句和 break 语句。函数调用语句可以带 0 到多个实际参数，但必须和函数声明的参数个数一致。

Sample 语言的文法的描述参见 3.2.7 节。递归下降分析方法的主要思想是根据文法的产生式，从开始符号开始自上而下进行分析。递归下降语法分析器的输入是词法分析输出的 token 文件，通过分析检查输入的 token 串是否符合文法要求，输出是语法树。递归下降语法分析器语法分析程序的接口如图 3.9 所示。

图 3.9 递归下降语法分析器语法分析程序的接口

递归下降分析从文法的开始符号向下分析。Sample 语言的开始符号是<程序>。根据文法描述可知,Sample 语言程序是按照一定的递归规则构成的,因此,语法分析程序是按照这个递归规则进行处理的。一个完整的 Sample 语言程序(见 1.5.4 节)由 0 到多个声明语句、main()函数、后跟 0 到多个函数定义构成。声明语句包括常量声明、变量声明和函数声明,它们之间没有顺序,有全局声明和局部声明,全局声明必须在 main()函数之前,局部声明在复合语句中。常量声明由 const 开头,因此当程序开始读入的符号如果是 const,就进入常量声明语句的处理;变量声明和函数声明必须要读到标识符后的一个符号才能区分,如果是(,则是函数声明,如果是等号或者逗号,则是变量声明;由于规定 main()函数由 main()开头,没有返回值,也没有形式参数,后面直接跟上一个复合语句,当复合语句结束后,如果继续读入了函数类型(int、char、float、void),则表明后面还有函数定义。整个流程如图 3.10 所示。

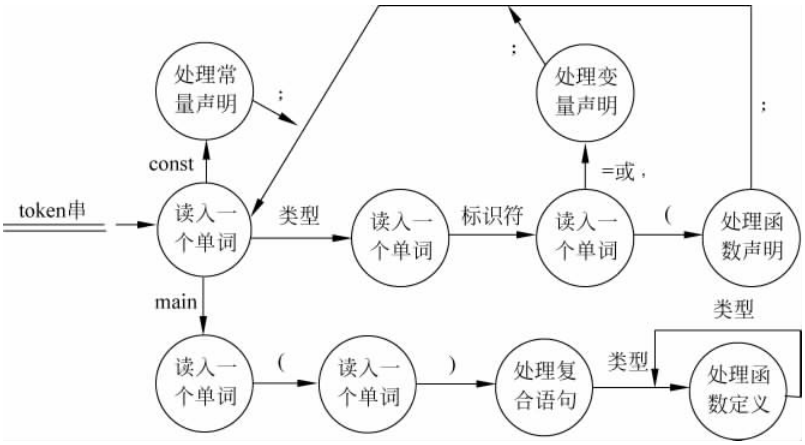


图 3.10 语法分析程序的处理流程

下面是根据该处理流程描述的递归下降语法分析器总控程序的伪代码。

```
void parser( ) /* 语法分析总控程序 */
{
    token = GetNextToken();                                          //从输入串中读取一个
                                                                     //符号到 token 中,输入
                                                                     //指针后移一个单词
    while(token!= ''main ''){
        if (token 是"const")
            ConstDeclareAnalyzer();                                  //处理常量声明
        else if (token 是"int"、"char"、"float"、"void"中的一个) {
              token = GetNextToken();
              if(token 不是标识符) SyntaxError();                    //调用错误处理
              token = GetNextToken();
              if(token 是"(") FunctionDeclareAnalyzer();             //处理函数声明
              else if(token 是" = "或",") VariableDeclareAnalyzer(); //处理变量声明
              else SyntaxError ();                                   //调用错误处理
              }
        }
        token = GetNextToken();
```

```
        if (token 不是"(" ) SyntaxError();                    //调用错误处理
        token = GetNextToken();
        if (token 不是")" ) SyntaxError();                    //调用错误处理
        CompoundSentenceAnalyzer();                           //处理复合语句
        token = GetNextToken();
        while (token 是"int"、"char"、"float"、"void"中的一个) {
            FunctionDefinitionAnalyzer();
            token = GetNextToken();
        }
}                                                             //整个程序结束
```

在这个流程中，GetNextToken()函数的功能是从输入的 token 串中取出一个单词符号，指针后移一个单词。SyntaxError()函数的功能是语法分析错误处理，报告出现了语法分析的哪一类错误，一般需要带参数，表示出现了哪一类错误，此处只是为了简化介绍，就不带参数。

接下来，需要继续对常量声明、变量声明、复合语句等的处理流程根据文法定义进一步细化，以分析复合语句为例来介绍。复合语句的文法定义是：

<复合语句>→{<语句表>}
<语句表>→<语句>|<语句><语句表>

也就是说，复合语句是以左花括号开头、右花括号结束的，当读到{表示进入复合语句的处理，后面就是各种语句，再读入一个单词，根据这个单词对语句进行分类；不同的单词进行不同的语句的处理：读入 const 就开始处理局部常量声明；读入各种类型(int、char、float)单词，就开始处理局部变量声明；读入 for、while、do、if 等关键词，就进入对应的语句的处理；读入{进入复合语句的处理；读入其他合法单词就开始处理表达式语句，否则就进入错误处理。处理完一个语句后再读入一个单词，如果单词不是}，表示该复合语句中还有其他语句，返回来继续判断是哪一类语句，否则读入了}，结束该复合语句。处理流程如图 3.11所示。

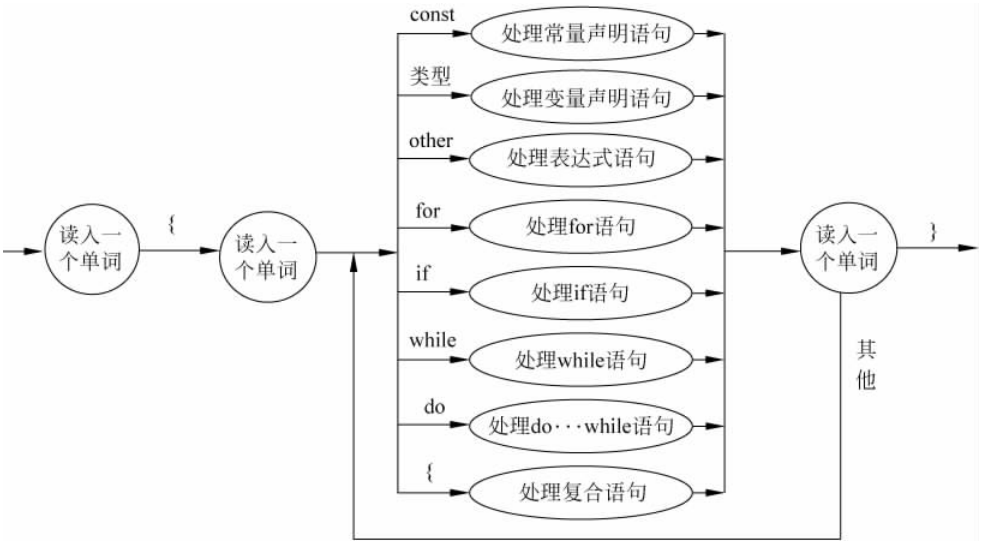


图 3.11 复合语句的处理流程

在上述处理流程中,每一个处理都是检查后续的 token 串组成的序列是否符合文法要求,各种语句的处理可以是嵌套的,同时还需要调用表达式的文法处理。

下面以 if 语句为例说明利用语法图来实现语法分析程序中的语句。假定 Sample 语言中 if 语句的文法定义为:

<if 语句>→if (<表达式>)<复合语句> | if (<表达式>) <复合语句> else <复合语句>

该文法前面部分是相同的,含有回溯,首先提取公共左因子得到< if 语句>的文法定义为:

<if 语句>→if (<表达式>)<复合语句> < if Tail >
< if Tail >→else <复合语句>|ε

根据该文法,首先画出语法图,如图 3.12 所示。

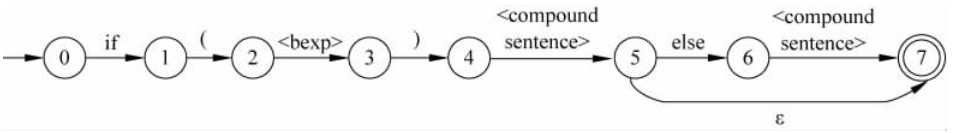


图 3.12 if 语句的语法图

根据语法图,可以写出递归下降的分析程序。

```
ifs( ) { /* 当读取的单词符号是 if 时,才调用该函数 */
    token = GetNextToken();              //从输入串中读取一个单词符号,指针后移一个单词
    if (token 不是"if" )                 // 如果调用函数已经读取了 if,此处不再读入和判断
        SyntaxError();                   //调用错误处理
    token = GetNextToken();
    if (token 不是"(" ) SyntaxError();   //调用错误处理
    token = GetNextToken();
    BoolExpressionAnalyzer();            //调用分析表达式的函数
    token = GetNextToken();
    if (token 不是")" ) SyntaxError();   //调用错误处理
    CompoundSentenceAnalyzer();          //处理复合语句
    token = GetNextToken();
    if(token 是"else") { /* 带有 else 部分时处理 else 部分 */
        token = GetNextToken();
        CompoundSentenceAnalyzer();      //处理 else 后的复合语句
    }
}
```

3.3.4 预测分析方法

预测分析方法(Forecasting Parse)是自上而下分析的另一种有效方法,要求文法必须是 LL(1)文法,分析过程将按照自左至右的顺序读入输入符号串,并在此过程中产生一个句子的最左推导。它采用表驱动的方式,通过显式地维护一个状态栈和一个二维分析表,在总控程序的控制下实现分析过程。按此方式执行语法分析任务的程序称为预测分析程序,或预测分析器。

1. 预测分析的工作过程

从逻辑上来看，预测分析器由三个部分组成：总控程序、栈(STACK)和预测分析表 M。另有一个输入缓冲区，产生一个输出流，如图 3.13 所示。输入缓冲区中存放待分析的串，以 # 标记输入串的结束；输出流是分析过程输出的结果。

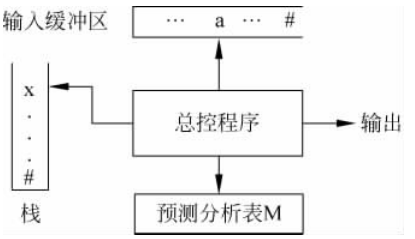


图 3.13 预测分析器模型

下面对这几个部分进行简要说明：

(1) 栈用于存放分析过程中的文法符号序列。

(2) 预测分析表 M 是一个二维数组，与文法有关。该表的行表示所有的非终结符，列表示所有的终结符和 #(注：# 不是文法的终结符，如果该语言中 # 是个合法字符，可以改用其他符号，把它当成输入串的结束符有利于简化分析算法的描述)。表项元素 M[A,a]存放着一条关于 A 的产生式，表明当用非终结符 A 向下推导时，如果面临输入符号 a，应选用的候选式；当 M[A,a]为空时，表明用 A 为左部向下推导时不应该面临输入符号 a，因此表中内容为空表示出现语法错误。表 3.3 是文法 G3.3 的预测分析表。

表 3.3 文法 G3.3 的预测分析表

	i	+	*	(	)	#
E	E→TE′			E→TE′		
E′		E′→+TE′			E′→ε	E′→ε
T	T→FT′			T→FT′		
T′		T′→ε	T′→*FT′		T′→ε	T′→ε
F	F→i			F→(E)		

(3) 总控程序。分析器对每一个输入串的分析均在总控程序的控制下工作。总控程序的工作就是控制分析器读入输入符号，根据预测分析表的内容对栈进行操作。

总控程序的工作过程如图 3.14 所示，与文法无关。首先将 # 和文法的开始符号压入栈中，然后总是根据栈顶符号 X 和当前的输入符号 a 工作。

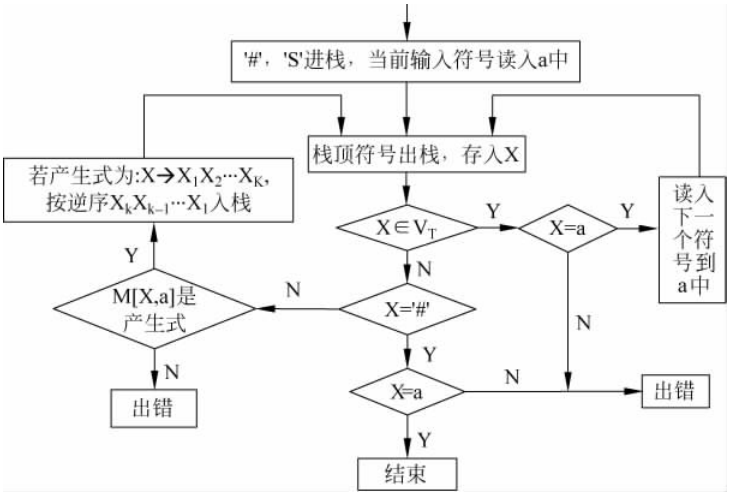


图 3.14 预测分析器总控程序的工作过程

如果栈顶符号是终结符,并且和 a 相等,则匹配成功,读取下一个输入符号;如果栈顶符号和输入符号都是#,则分析成功;如果栈顶符号是非终结符,则查预测分析表,如果 M[X,a]中存放有产生式"$X\rightarrow Y_1Y_2\cdots Y_n$",则将其按 $Y_n\cdots Y_2Y_1$ 顺序压入堆栈;算法结束,要么成功匹配一个输入符号串,要么出错。

图中符号说明如下。

"#":句子括号即输入串的结束符。

"S":文法的开始符号。

"X":存放当前栈顶符号的工作单元。

"a":存放当前输入符号的工作单元。

总控程序的工作过程可以用算法 3.3 来描述。

算法 3.3 预测分析方法

输入:一个待分析的输入符号串 w,以"#"作为结束标志;文法 G 的预测分析表 M

输出:如果 $w\in L(G)$,输出 w 的最左推导,否则提示语法错误

步骤:

(1) 将"#"和文法开始符号压入堆栈。

(2) 设 IP 为输入指针,指向输入符号串的第一个符号 a,a=w[IP]。

(3) 弹出栈顶符号,存入 X。

(4)
```
while ( X != "#" ) {                    //栈非空
    if ( X == a ) {                     //读取下个符号
      IP = IP + 1;
      a = w[IP];
      }
    else if (X∈VT),error();
    else if (X∈VN) {                    //查看分析表 M
            if (M[X, a]是一个出错标记) error();
            else {
              if (M[X,a] = "X→X1X2…Xn") {
                输出产生式;
                把 XnXn-1…X1 压入栈中;   //产生式右部反序压栈.若右部为 ε,不压入
              }
      }
      弹出栈顶符号,存入 X;
  }
```

例 3.19 使用表 3.3 对输入串 i+i*i# 进行预测分析。

使用表 3.3 进行预测分析的过程及栈的变化如表 3.4 所示。输入指针指向剩余输入串最左边的符号。语法分析器跟踪的是输入串的最左推导,即推导所使用的产生式正好就是最左推导使用的那些产生式。

表 3.4 符号串 i+i*i# 的预测分析过程

步骤	分析栈	剩余输入串	推导所用产生式或匹配
1	#E	i+i*i#	$E\rightarrow TE'$

续表

步骤	分析栈	剩余输入串	推导所用产生式或匹配
2	#E′T	i+i*i#	T→FT′
3	#E′T′F	i+i*i#	F→i
4	#E′T′i	i+i*i#	i匹配
5	#E′T′	+i*i#	T′→ε
6	#E′	+i*i#	E′→+TE′
7	#E′T+	+i*i#	+匹配
8	#E′T	i*i#	T→FT′
9	#E′T′F	i*i#	F→i
10	#E′T′i	i*i#	i匹配
11	#E′T′	*i#	T′→*FT′
12	#E′T′F*	*i#	*匹配
13	#E′T′F	i#	F→i
14	#E′T′i	i#	i匹配
15	#E′T′	#	T′→ε
16	#E′	#	E′→ε
17	#	#	接受

2. 预测分析表的构造

上面介绍的预测分析方法中，对于不同的输入符号串和文法，预测分析的总控程序都是相同的，不同的只是分析表，分析表确定了在分析过程中选用哪个候选式来进行推导。也就是说，总控程序是通用的，与文法无关，分析表是和文法相关的。

对实现预测分析器来说，总控程序相对比较简单，容易实现，关键在于构造预测分析表。

对于任意给定的LL(1)文法G，构造预测分析表M的算法思想是：假定A→α是A的一个候选式，a∈First(α)，那么，当A在栈顶且a是当前输入符号时，α应被当作是A的唯一匹配，M[A,a]中应放进产生式A→α；当α=ε或$\alpha \overset{*}{\Rightarrow} \varepsilon$且当前输入符号a∈Follow(A)(a可能是终结符或#)时，A→α就认为已自动得到匹配。因此，应把A→α放进M[A,a]中。根据这个思想，可以得到下面的构造预测分析表M的算法。

算法3.4 预测分析表的构造算法

输入：文法G

输出：预测分析表M

步骤：

(1) 对文法G的每个产生式A→α执行第2步和第3步。

(2) 对每个终结符a∈First(α)，则把A→α加至M[A,a]中。

(3) 若ε∈First(α)，则对任何b∈Follow(A)，把A→α加入M[A,b]中。

(4) 把所有无定义的M[A,a]标上“出错标志”。

例3.20 为文法G3.3构造预测分析表。

(1) 构造预测分析表需要先判断该文法是否是LL(1)文法。首先G3.3不含左递归，每个非终结符的各个候选式的First集互不相交，对E′、T′含有ε产生式，First(E′)∩

Follow(E′)= Φ,First(T′)∩Follow(T′)= Φ,因此文法 G3.3 为 LL(1)文法。

(2) 首先计算各个候选式的 First 集,如果某候选式的 First 集中含有 ε,再计算候选式左边的非终结符的 Follow 集。

(3) 建立一个以非终结符为行,终结符和 # 为列的空表格,根据算法 3.4 的步骤进行填表。填表过程如下。

由于 First(E→ TE′)={(,i },因此产生式 E→ TE′应放入 E 所对应的行,(和 i 对应的列。

由于 First(E′→+TE′)={+},因此产生式 E′→+TE′应放入 E′所对应的行,+对应的列。

由于 First(E′→ε)={ε},因此,计算 E′的 Follow 集,由于 Follow(E′)={),# },因此 E′→ε 应填入 E′对应的行,)和 # 对应的列。

由于 First(T→ FT′)={(,i},因此产生式 T→ FT′应放入 T 所对应的行,(和 i 对应的列。

由于 First(T′→ * FT′)={ * },因此产生式 T′→ * FT′应放入 T′所对应的行, * 对应的列。

由于 First(T′→ε)={ε},因此,计算 T′的 Follow 集,由于 Follow(T′)={+,),# },因此 T′→ε 应填入 T′对应的行,+、)和 # 对应的列。

由于 First(F→(E))={(},因此产生式 F→(E)应放入 F 所对应的行,(对应的列。

由于 First(F→i)={i},因此产生式 F→i 应放入 F 所对应的行,i 对应的列。

最终结果如表 3.2 所示。

例 3.21 综合题。已知某语言中程序的文法 G 为:

```
<PROGRAM>→begin <STL> end
<STL>→<STMT>|<STL>;<STMT>
<STMT>→<NCONDITION>|<CONDITION>
<NCONDITION>→a
<CONDITION>→<IFS>|<IFS> else <STMT>
<IFS>→<IFCLAUSE><NCONDITION>
<IFCLAUSE>→if c then
```

(1) 将 G 改写为等价的 LL(1)文法,并给以证明。

(2) 构造改写后的文法的预测分析表。

(3) 判断输入串 begin if c then a else a end 是否为文法 G 的句子。

解:

(1) 将文法简化为 G[P]。

P→bTd

T→S|T; S

S→N|C

N→a

C→I|IeS

I→ZN

Z→ict

相应地，输入串化简为 bictaead，即判定该符号串是否是一个句子。

① 消去 G[P]中的左递归 T 和公共左因子 I，等价的文法 G′[P]为：

P→bTd

T→SF

F→；SF|ε

S→N|C

N→a

C→ID

D→eS |ε

I→ZN

Z→ict

② 计算每个候选式的 First 集，如果含有 ε，计算其左部非终结符 Follow 集，结果如下：

First(P→bTd)＝{b}

First(T→SF)＝{a,i}

First(S→N)＝{a}

First(S→C)＝{i}

First(F→；SF)＝{；}

First(F→ε)＝{ε}

First(N→a)＝{a}

First(C→ID)＝{i}

First(I→ZN)＝{i}

First(D→eS)＝{e}

First(D→ε)＝{ε}

First(Z→ict)＝{i}

由于只有 F 和 D 含有 ε 产生式，因此只需计算 Follow(F)和 Follow(D)。

Follow(F)＝{d}

Follow(D)＝{；，d}

③ 判断：由于该文法已不含左递归，S 的候选式有两个，其中：

First(S→N)∩ First(S→C)＝{a}∩{i}＝Φ

F 和 D 均含有为 ε 产生式，根据条件 3，有：

First(F)∩Follow(F)＝{；}∩{d}＝Φ

First(D)∩Follow(D)＝{e}∩{；，d}＝Φ

所以文法 G′[P]是 LL(1)的文法。

(2) 构造改写后的文法 G′[P]的预测分析表，如表 3.5 所示。

表 3.5 G′[P]的预测分析表

	b	d	；	a	e	i	c	t	#
P	P→bTd								
T				T→SF		T→SF			

续表

	b	d	;	a	e	i	c	t	#
F		F→ε	F→; SF						
S				S→N		S→C			
N				N→a					
C						C→ID			
I						I→ZN			
D		D→ε	D→ε		D→eS				
Z						Z→ict			

(3) 用表 3.5 对输入串 bictaead 进行分析的步骤如表 3.6 所示。

表 3.6 用表 3.5 对输入串 bictaead 进行分析的步骤

步 骤	符 号 栈	剩余输入串	规 则
1	#P	bictaead#	P→bTd
2	#dTb	bictaead#	匹配
3	#dT	ictaead#	T→SF
4	#dFS	ictaead#	S→C
5	#dFC	ictaead#	C→ID
6	#dFDI	ictaead#	I→ZN
7	#dFDNZ	ictaead#	Z→ict
8	#dFDNtci	ictaead#	匹配
9	#dFDNtc	ctaead#	匹配
10	#dFDNt	taead#	匹配
11	#dFDN	aead#	N→a
12	#dFDa	aead#	匹配
13	#dFD	ead#	D→eS
14	#dFSe	ead#	匹配
15	#dFS	ad#	S→N
16	#dFN	ad#	N→a
17	#dFa	ad#	匹配
18	#dF	d#	F→ε
19	#d	d#	匹配
20	#	#	接受

到达接受状态,分析成功,说明输入串 bictaead 是文法 G′的句子,从而得到输入串 begin if c then a else a end 是文法 G 的句子。

3.4 自下而上的语法分析

3.3 节介绍了自上而下的语法分析,其目的是从文法的开始符号出发,根据语法规则建立一棵以文法开始符号为根、以被分析符号串为叶子结点的语法树。

自下而上语法分析的目的仍然是构造一棵语法树。它构造的过程是先以被分析符号串

的各个符号为叶子结点，根据文法规则，以产生式左部的非终结符为父结点，逐步向上构造子树，最后得到以文法开始符号为根的语法树。本节重点介绍这种方法中的一些基本概念。

3.4.1 自下而上分析方法概述

1. “移进-归约”分析方法

在3.2节介绍过归约的基本概念，它是推导的逆过程。自下而上语法分析的基本思想是“移进-归约”。设置一个栈，将输入符号串（指的是从词法分析器送来的token串）中的单词符号逐个移入栈中，边移入边分析，一旦栈顶形成某个产生式的右部时，就用该产生式左部的非终结符代替，这个过程称为归约。重复这一过程，直到归约到栈中只剩下文法的开始符号，就认为该输入符号串符合文法规则，是文法的句子，分析成功，否则出错。

例3.22 假设一文法G为：

(1) S→aAbB

(2) A→c|Ac

(3) B→d (G3.5)

试对输入符号串accbd进行分析，检查该符号串是否是文法G的一个句子。

具体分析过程如表3.7所示。分析前设置一个分析栈，并将“#”压入栈底。接着第一个输入符号a进栈，a不是任何产生式的右部，因此继续将c移进栈，此时栈顶的c已形成产生式A→c的右部，于是把栈顶的c归约为A（表中第4步）；再移进下一个c，栈顶的两个符号Ac形成了产生式A→Ac的右部，将其归约为A（表中第6步）；继续移进b，此时栈顶的符号串aAb、Ab或b都不是任何产生式的右部，因此继续移进d，而d已形成了产生式B→d的右部，因此将d归约为B（表中第9步）；此时，栈顶的符号串aAbB恰好是第一个产生式的右部，直接把它归约为开始符号S。分析成功，说明输入串accbd是文法G的一个句子。

表3.7 输入串accbd的自下而上分析过程

步 骤	分 析 栈	输 入 串	分析动作
1	#	accbd#	预备
2	#a	ccbd#	移进
3	#ac	cbd#	移进
4	#aA	cbd#	归约(A→c)
5	#aAc	bd#	移进
6	#aA	bd#	归约(A→Ac)
7	#aAb	d#	移进
8	#aAbd	#	移进
9	#aAbB	#	归约(B→d)
10	#S	#	归约(S→aAbB)

在上述分析过程中，每一步归约都是将栈顶的符号串归约为产生式左部的符号，也就是说进行归约的符号串总是出现在分析栈的栈顶而不会出现在栈的中间。把栈顶的这样一串符号称为“可归约串”。

上述过程共用了10步，分别用了4个产生式进行了4次归约。初看起来，这种移进-归约很简单，其实不然。在本例中的第6步，如果不是将Ac归约为A，而是将c归约为A，使

分析栈中的符号串为 aAA,这样显然达不到归约为 S 的目的,从而也就无法得知输入串 accbd 是一个合法的句子。由此可以看出,可归约串必定是某个产生式的右部,但是构成某个产生式右部的栈顶符号串不一定是可归约串。

从上述分析可以看出自下而上分析的关键问题。

(1) 判断栈顶符号是否形成了某个产生式的右部。

(2) 如果栈顶形成了多个产生式的右部,决定选用哪个产生式进行归约。

不同的自下而上的语法分析方法对上述两个问题的定义和处理方法不同。

在上述分析过程中,共进行了四种操作。

① 移进:把输入符号串中的当前符号移进栈。

② 归约:发现栈顶已形成可归约串,用适当的产生式的左部去替换这个串。

③ 接受:宣布分析成功,可以看成是"归约"的一种特殊形式,是栈顶为开始符号 S、输入串已读入完毕的一种特殊状态。

④ 出错处理:是指发现栈顶的内容与输入串相悖,分析工作无法正常进行。此时需调用出错处理程序进行诊察和校正,并对栈顶内容和输入符号进行调整。

上述语法分析过程可以看成是自下而上构造语法树的过程,每一步归约都可以画出一棵子树来,随着归约的完成,这些子树被连成一棵完整的语法树。根据表 3.7 的分析过程构造语法树的过程如图 3.15 所示。

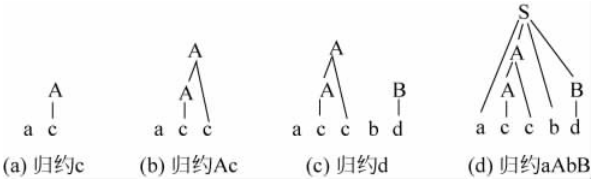


图 3.15 句型 accbd 的语法树的自下而上构造过程

2. 规范归约、短语和句柄

假设有一文法 G,开始符号为 S,如果有

$$S \overset{*}{\Rightarrow} x\beta y \text{ 且 } A \overset{+}{\Rightarrow} \beta \quad x,y,\beta \in (V_T \cup V_N)^*, A \in V_N$$

则称 β 是句型 xβy 相对于非终结符 A 的短语(Phrase)。特别地,如果

$$A \rightarrow \beta$$

则称 β 是句型 xβy 的直接短语(Direct Phrase)。位于一个句型的最左直接短语称为该句型的句柄(Handle)。

注意"短语"的定义,只有 $A \overset{+}{\Rightarrow} \beta$ 或 $A \rightarrow \beta$ 不一定意味着 β 是一个短语,必须有 $S \overset{*}{\Rightarrow} x\beta y$ 这一条件,即 xβy 必须是一个句型,离开句型来讨论短语没有意义。

例 3.23 考虑文法 G3.2

(1) E→E+T|T

(2) T→T * F|F

(3) F→i|(E) (G3.2)

给定句型 $i_1+i_2*i_3$,判断其短语、直接短语、句柄。

由于存在推导:

$E \Rightarrow E+T \Rightarrow T+T \Rightarrow F+T \Rightarrow i_1+T \Rightarrow i_1+T*F \Rightarrow i_1+F*F \Rightarrow i_1+i_2*F \Rightarrow i_1+i_2*i_3$

$E \Rightarrow T \Rightarrow F \Rightarrow i_1$　　$T \Rightarrow F \Rightarrow i_2$　　$F \Rightarrow i_3$　　$T \Rightarrow T*F \Rightarrow i_2*i_3$

所以可以看出，i_1，i_2，i_3，i_2*i_3，$i_1+i_2*i_3$ 是句型 $i_1+i_2*i_3$ 的短语；直接短语有 i_1，i_2，i_3；句柄是 i_1。i_1+i_2 不是短语，因为不存在从 E 到 $T*i_3$ 的推导。

根据定义来判断句型的短语和句柄比较困难。如果使用语法树来表示一个句型，则句型中的句柄和短语就一目了然。一棵语法树的一棵子树是由该树的某个结点（作为子树的根）连同它的所有子孙组成的。一个子树的所有叶子结点自左至右排列起来形成一个相对于子树根的短语。只有父子两代结点形成的子树的叶子结点自左至右排列起来形成相对于子树根的直接短语；一个句型的句柄是这个句型所对应的语法树中最左边那个构成直接短语的子树的叶子结点自左至右的排列。

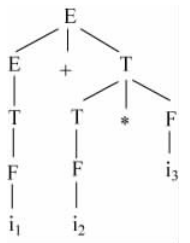


图 3.16　句型 $i_1+i_2*i_3$ 的语法树

图 3.16 是句型 $i_1+i_2*i_3$ 的语法树。从该语法树可以得出例 3.23的结论。

在例 3.22 的归约过程中，每一步归约的都是当前句型的句柄。若一个文法无二义性，则该文法的某句型中的句柄就是唯一的。在例 3.22 中的第 6 步，Ac 是句柄，而 c 不是句柄。

例 3.24　利用寻找句柄的方式对输入符号串 i+i*i# 进行归约。

在进行归约的每一步，都是先寻找句柄，并用相应产生式的左部符号进行替换，归约过程如表 3.8 所示。

表 3.8　i+i*i# 的归约过程

步　骤	栈中符号	输入缓冲区	句　柄	归约规则
(1)	#	i+i*i#		准备
(2)	#i	+i*i#		移进 i
(3)	#F	+i*i#	i	归约 F→i
(4)	#T	+i*i#	F	归约 T→F
(5)	#E	+i*i#	T	归约 E→T
(6)	#E+	i*i#		移进+
(7)	#E+i	*i#		移进 i
(8)	#E+F	*i#	i	归约 F→i
(9)	#E+T	*i#	F	归约 T→F
(10)	#E+T*	i#		移进 *
(11)	#E+T*i	#		移进 i
(12)	#E+T*F	#	i	归约 F→i
(13)	#E+T	#	T*F	归约 T→T*F
(14)	#E	#	E+T	归约 E→E+T
(15)		接受		

下面将前面介绍的几个概念进行形式化。假定 α 是文法 G 的一个句子，称序列 α_n，

α_{n-1},…,α_0 是 α 的一个规范归约,如果此序列满足如下条件。

(1) $\alpha_n=\alpha$。

(2) α_0 为文法的开始符号,即 $\alpha_0=S$。

(3) 对任何 $i(0<i\leqslant n)$,α_{i-1}是从 α_i 经把句柄替换为相应产生式的左部符号而得到的。

在上述例子中,序列 accbd,aAcbd,aAbd,aAbB,S 构成句子 accbd 的一个规范归约。简单地讲,在归约过程中始终对句柄进行归约而形成的序列称为规范归约,规范归约也称最左归约。由最左归约所得到的句型称为规范句型。

将上述规范归约过程的顺序倒过来,得到:

$$S\Rightarrow aAbB\Rightarrow aAbd\Rightarrow aAcbd\Rightarrow accbd$$

该过程和句型的最右推导一致,因此,最右推导又称为规范推导,如果文法 G 是无二义的,规范推导(最右推导)的逆过程必是规范归约(最左归约)。

注意句柄的"最左"特征,这一点对于"移进-归约"来说很重要,因为句柄的"最左"性和分析栈的栈顶两者是相关的。由于句型中的非终结符由归约产生,而句柄在句型的最左边,所以,在一个规范句型中,句柄的右边只可能出现终结符,不可能出现非终结符。基于这一点,可用句柄来刻画"移进-归约"过程的"可归约串"。因此,规范归约的实质是,在移进过程中,当发现栈顶呈现句柄时就用相应产生式的左部符号进行替换(即归约)。

为了加深对"句柄"和"归约"这些重要概念的理解,使用修剪语法树的方法来进一步阐明自下而上的分析过程。

例如,对图 3.15(d)所示的语法树,采用修剪语法树的方法来实现归约,也即每次寻找当前语法树的句柄(在语法树中用虚线勾出),然后将句柄中的树叶剪去(即实现一次归约),得到一个新的句型;再寻找新的句型中的句柄,这样不断地修剪下去,当剪到只剩下根结点时,就完成了整个归约过程,如图 3.17 所示。

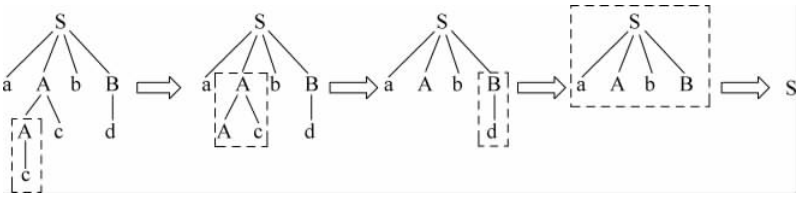


图 3.17 修剪语法树实现归约

本节简单地讨论了"规范归约"和"句柄"这两个基本概念,但并没有解决如何寻找句柄、如何归约的问题。事实上,规范归约的中心问题就是如何寻找或确定一个句型的句柄。不同的寻找句柄的方法形成了不同的自下而上的分析方法。

3.4.2 算符优先分析

在算术表达式的求值过程中,运算次序是先乘除后加减,这说明了乘除运算的优先级高于加减运算的优先级;乘和除优先级相同,加和减优先级相同;在相同优先级的情况下,出现在左边的运算符先运算,这称为左结合。如果计算的每一步做一个运算,那么求值过程的每一步都是唯一的。这说明运算的次序只与运算符有关,而与运算对象无关。算符优先分析法的思想源于这种表达式的分析,因此算符优先分析的关键是定义文法 G 中相邻算符之

间的优先关系，即给出算符之间的优先级和同一级别的结合性质，以指示表达式的计算次序。

算符优先分析法是一种简单且直观的自下而上分析方法，它特别适合分析程序设计语言中的各类表达式。在归约过程中起决定作用的是相邻运算符的优先级。但并不是所有文法都能用算符优先分析方法进行分析，只有算符优先文法(Operator Precedence Grammar，OPG)才能进行算符优先分析。

1. 算符优先文法

一个文法，如果它的任何产生式的右部都不含两个相继(并列)的非终结符，即不含如下形式的产生式：

$$P \rightarrow \cdots QR \cdots$$

则称该文法 G 为算符文法(Operator Grammar)，也称 OG 文法。

例如，文法 E→E+E|E*E|(E)|i，其中任何一个产生式都不包含两个非终结符相邻的情况，因此该文法是算符文法。

在后面的定义中，a、b 代表任意终结符；P、Q、R 代表任意非终结符；“…”代表由终结符和非终结符组成的任意序列，包括空字。

假定 G 是不含 ε-产生式的算符文法，终结符对 a、b 之间的优先关系定义如下。

(1) $a \doteq b$ 当且仅当文法 G 中含有形如 P→…ab…或 P→…aQb…的产生式。

(2) $a \lessdot b$ 当且仅当 G 中含有形如 P→…aR…的产生式，且 $R \overset{+}{\Rightarrow} b\cdots$ 或 $R \overset{+}{\Rightarrow} Qb\cdots$。

(3) $a \gtrdot b$ 当且仅当 G 中含有形如 P→…Rb…的产生式，且 $R \overset{+}{\Rightarrow} \cdots a$ 或 $R \overset{+}{\Rightarrow} \cdots aQ$。

假定 G 是一个不含 ε-产生式的算符文法，任何终结符对(a，b)如果至多只满足下述三种优先关系之一：

$$a \doteq b, a \lessdot b, a \gtrdot b$$

则称 G 是一个算符优先文法，简称 OPG 文法。算符优先文法是无二义性的。

例 3.25 对文法：E→E+E|E*E|(E)|i，证明该文法不是 OPG 文法。

证明：首先该文法是不含 ε-产生式的算符文法。

因为 E→E+E，E⇒E*E，所以有 $+ \lessdot *$，语法子树如图 3.18(a)所示。

又因为 E→E*E，E⇒E+E，所以有 $+ \gtrdot *$，语法子树如图 3.18(b)所示。

在+和*之间同时存在两种优先关系，所以该文法不是 OPG 文法。

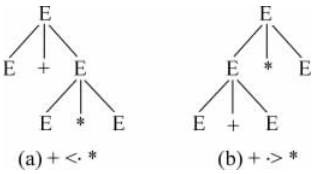


图 3.18 文法 E→E+E|E*E|(E)|i 语法子树

2. 算符优先表的构造

如果 G 是一个算符优先文法，则可以根据终结符之间的优先关系进行语法分析，终结符之间的优先关系可以指导句柄的选取。因此，如果要对算符优先文法 G 进行优先关系分析，则必须首先求出各个终结符对之间的优先关系。用表格形式来表示文法中各终结符之间的优先关系，这种表称为算符优先关系表(Operator Precedence Relation Table)。

1) 按照定义手工构造算符优先关系表

例 3.26 根据定义计算表达式文法 G3.2 的算符优先关系表。

(1) E→E+T|T

(2) T→T*F|F

(3) F→(E)|i (G3.2)

第一步,首先规定:'#'作为句子的起始和终止界符,为了分析过程的确定性,把'#'号作为终结符对待。为保证语法分析的进行,拓展文法 G,增加一个产生式:(0)E'→#E#,即必须有#⋖a 和 b⋗#成立,其中 a 为任何从 E 推导出的所有句型中的第一个终结符,b 为任何从 E 推导出的所有句型中的最后一个终结符。

第二步,根据定义计算。根据第 3 个产生式 F→(E),有(≐);根据产生式 E→E+T 和 T→T*F,有+⋖*;根据产生式 T→T*F 和 F⇒(E),有*⋖(;根据第 3 个产生式 F→(E),且 E⇒E+T,有(⋖+和+⋗);……;总之,根据优先关系的定义,用文法产生式进行推导,可以得到各个可能相邻的终结符对之间的优先关系。

第三步,对文法中的任意两个不可能相邻的终结符,它们之间无优先关系,在表中以空白表示。最后得到文法 G3.2 的算符优先关系表如表 3.9 所示。

表 3.9 文法 G3.2 的算符优先关系表

	+	*	i	(	)	#
+	⋗	⋖	⋖	⋖	⋗	⋗
*	⋗	⋗	⋖	⋖	⋗	⋗
i	⋗	⋗			⋗	⋗
(	⋖	⋖	⋖	⋖	≐	
)	⋗	⋗			⋗	⋗
#	⋖	⋖	⋖	⋖		≐

说明:

(1) 在优先表中,空白部分表示两个符号之间没有优先关系,如果在符号串中相继出现表示出现了一种语法错误。

(2) 相同终结符之间的优先关系不一定是≐,如表 3.9 中,*⋗*,在此表示了结合性。

(3) 如果 a≐b,不一定 b≐a,如表 3.9 中,(≐),而)与(之间无优先关系。

(4) 如果 a⋖b,不一定 b⋗a,即不具有对称性,因为只定义相邻运算符之间的优先关系,a,b 相邻时,不一定 b,a 相邻,如表 3.9 中,(⋖i,i 与(之间没有优先关系。

(5) 如果 a⋖b,b⋖c,不一定 a⋖c,即不具有传递性,也就是说,若 a,b 相邻、b,c 相邻时,不一定 a,c 相邻。

2) 利用算法构造算符优先关系表

为了实现算符优先关系表的自动生成,首先定义一个非终结符的 FirstVT 和 LastVT 集。

对文法 G 的任一非终结符 P,定义两个集合:

FirstVT(P)={a|$P\overset{+}{\Rightarrow}$a…或 $P\overset{+}{\Rightarrow}$Qa…,a∈$V_T$ 而 Q∈V_N},即 P 能推导出的第一个终结符构成的集合。

LastVT(P)$=\{a|P\overset{+}{\Rightarrow}\cdots a$ 或 $P\overset{+}{\Rightarrow}\cdots aQ, a\in V_T$ 而 $Q\in V_N\}$，即 P 能推导出的最后一个终结符构成的集合。

有了这两个集合，就可以通过优先关系的定义，并检查文法的产生式来求各终结符对之间的优先关系。

① $\doteq$关系：若有形如 P→…ab…或 P→…aQb…的产生式，则 a $\doteq$ b。可直接查看产生式得到。

② $\lessdot$关系：若有形如 Q→…aP…的产生式，对任何 b∈FirstVT(P)，有 a $\lessdot$ b。

③ $\gtrdot$关系：若有形如 Q→…Pb…的产生式，对任何 a∈LastVT(P)，有 a $\gtrdot$ b。

由此可知，有了 FirstVT 和 LastVT 的定义，只要给出求文法的非终结符的 FirstVT 和 LastVT 集合的算法，就可以自动构造文法的优先关系表。

进一步，分析 FirstVT 的定义，计算非终结符 P 的 FirstVT 集的算法基于下面两条规则。

规则 1：若有产生式 P→a…或 P→Qa…，则 a∈FirstVT(P)，其中 P、Q∈V_N，a∈V_T。

规则 2：若有产生式 P→Q…，且 a∈FirstVT(Q)，则 a∈FirstVT(P)。

在实现 FirstVT 集的计算时，首先建立一个布尔数组 F[P,a]，表的行是所有的非终结符，列是所有的终结符，其初值为全 0；然后通过上面的规则 1 对数组 F 进行初始化；再利用规则 2 修改数组 F，即如果发现 a∈FirstVT(P)，修改 F[P,a]=1；最后在二维数组 F 中，每行中元素为 1 对应的终结符构成的集合就是该行对应的非终结符的 FirstVT 集，即 FirstVT(P)=$\{a|F[P,a]=1\}$。该计算过程可通过建立一个栈来实现，形式化描述如算法 3.5 所示。

算法 3.5 FirstVT 集的构造

输入：文法 G

输出：每个非终结符的 FirstVT 集

步骤：

(1) 对每个非终结符 P 和终结符 a 设置 F[P,a]=0。

(2) 对每个形如 P→a…或 P→Qa…的产生式，有

```
F[P, a] = 1;          //应用规则 1
PUSH(P, a);           // (P, a)压栈
```

(3) 当堆栈非空，将栈顶元素弹出至(Q,a)。

对每条形如 P→Q…的产生式，应用规则 2，有

```
F[P, a] = 1;
PUSH(P, a);           // (P, a)压栈
```

例 3.27 构造文法 G3.2 中每个非终结符的 FirstVT 集。

首先，建立一个 3 行 5 列的数组 F，置全部元素为 0。

其次，应用规则 1，用形如 P→a…或 P→Qa…的产生式，对数组 F 初始化并压栈，如表 3.10所示。

最后，当栈非空时，应用规则 2，对栈进行操作，并寻找形如 P→Q…的产生式，修改数组 F 的值。得到的结果如表 3.11 所示。因此，有

表 3.10 应用规则 1 后的数组 F

	+	*	(	)	i
E	1				
T		1			
F			1		1

表 3.11 最终结果数组 F

	+	*	(	)	i
E	1	1	1		1
T		1	1		1
F			1		1

FirstVT(E)={ +, *, (, i }

FirstVT(T)={ *, (, i }

FirstVT(F)={ (, i }

同理,可以根据 LastVT 集的定义来构造计算 LastVT 集的算法。LastVT 集基于下面两条规则。

规则 1:若有产生式 P→…a 或 P→…aQ,则 a∈LastVT (P),其中 P、Q∈V_N,a∈V_T。

规则 2:若 a∈LastVT(Q),且有产生式 P→…Q,则 a∈LastVT(P)。

当计算出每个非终结符的 FirstVT 集和 LastVT 集,就能够构造文法 G 的算符优先关系表,可用算法 3.6 来形式化地描述。

算法 3.6 构造文法 G 的算符优先关系表

输入:文法 G 及文法 G 的每个非终结符的 FirstVT 集、LastVT 集

输出:文法 G 的算法优先关系表

步骤:

对文法 G 中的每个产生式 P→$X_1X_2\cdots X_n$,对 i=1 到 n−1,检查相邻的文法符号的下述四种情况。

(1) 如果 $X_i\in V_T$ 且 $X_{i+1}\in V_T$,则 $X_i \doteq X_{i+1}$; /* P→…ab… */

(2) 如果 i≤n−2,且 $X_i\in V_T$,$X_{i+2}\in V_T$,$X_{i+1}\in V_N$,

则 $X_i \doteq X_{i+2}$; /* P→…aQb… */

(3) 如果 $X_i\in V_T$,且 $X_{i+1}\in V_N$,

则对每一个 a∈FirstVT(X_{i+1}),设置 $X_i \lessdot a$; /* P→…aR… */

(4) 如果 $X_i\in V_N$,且 $X_{i+1}\in V_T$,

则对每一个 a∈LastVT (X_i),设置 $a \gtrdot X_{i+1}$; /* P→…Rb… */

例 3.28 利用算法 3.6 求文法 G3.2 的算符优先关系表。

解:(1) 同手工构造一样,对文法进行拓展,增加产生式:(0)E′→#E#。

(2) 计算各终结符对之间的优先关系。

① 计算'≐'关系。

由产生式(0)E′→#E# 和(3)F→(E),根据算法中的第 2 种情况,可得'#'≐'#','('≐')'成立。

② 计算每个非终结符的 FirstVT 集和 LastVT 集。

FirstVT(E′)={#} LastVT(E′)={#}

FirstVT(E)={+, *, (, i} LastVT(E)={+, *,), i}

FirstVT(T)={*,(,i}　　LastVT(T)={*,),i}

FirstVT(F)={(,i}　　LastVT(F)={),i}

③ 计算'⋖'关系(利用算法3.6的第3种情况,逐条扫描产生式,寻找形如"P→…aR…"的产生式,则 a⋖FirstVT(R))。

由 E′→#E#,得到#⋖FirstVT(E)。

由 E→E+T,得到+⋖FirstVT(T)。

由 T→T*F,得到*⋖FirstVT(F)。

由 F→(E),得到(⋖FirstVT(E)。

④ 计算'⋗'关系(利用算法3.6的第4种情况,逐条扫描产生式,寻找形如"…Rb…"的产生式,则 LastVT(R)⋗b)。

由 E′→#E#,得到 LastVT(E)⋗#。

由 E→E+T,得到 LastVT(E)⋗+。

由 T→T*F,得到 LastVT(T)⋗*。

由 F→(E),得到 LastVT(E)⋗)。

从而构造出文法 G3.2 的算符优先关系表如表3.9所示。

3. 算符优先分析过程

有了算符优先关系表,就可以对任意给定的符号串进行算符优先分析,进而判定输入符号串是否为该文法的句子。

算符优先分析方法通过比较相邻终结符间的优先关系来进行分析,仍然采用"移进-归约"方式,不断移进输入符号,识别可归约串,并进行归约。但是,由于利用算符优先分析法,仅考虑了终结符之间的优先关系,没有考虑非终结符之间的优先关系,所以每次归约的并不一定是当前句型的句柄。实际上,算符优先分析法不是用句柄来刻画"可归约串",而是用最左素短语(Leftmost Prime Phrase)来刻画"可归约串"。

所谓素短语(Prime Phrase)是指这样的一个短语:它至少含有一个终结符,并且除它自身外,不含有更小的素短语。所谓最左素短语是指处于句型最左边的那个素短语。

从定义可以看出,最左素短语必须具备如下三个条件。

(1) 至少包含一个终结符。

(2) 除自身外不包含其他素短语(最小性)。

(3) 在句型中具有最左性。

例 3.29 对文法 G3.2 求句型 T+T*F+i 的短语、素短语和最左素短语。

句型 T+T*F+i 的语法树如图3.19所示。根据语法树可知,T+T*F+i,T+T*F,T,T*F,i 都是该句型的短语;由素短语的定义和最左素短语必须具备的条件可知,只有 i 和 T*F 为素短语,T+T*F(含素短语 T*F)、T+T*F+i(含素短语 T*F 和 i)和 T(不含终结符)都不是素短语,T*F 为最左素短语。

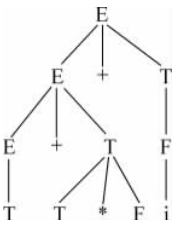


图 3.19 句型 T+T*F+i 的语法树

算符优先文法中的句型(括在两个#号之间)可以写成更一般的形式:

$$\#N_1a_1N_2a_2\cdots N_na_nN_{n+1}\#\qquad(3.3)$$

其中 $a_i(1\leqslant i\leqslant n)$是终结符,$N_i(1\leqslant i\leqslant n+1)$是可有可无的非终结符。也就是说,句型中含有 n 个终结符,任何两个终结符之间最多只有一个非终结符。任何算符优先文法的句型都具有这种结构形式。

算符优先分析方法基于下面的定理:一个算符优先文法 G 的任何句型[见式(3.3)]的最左素短语是满足如下条件的最左子串 $N_ia_i\cdots N_ja_jN_{j+1}$。

$$a_{i-1}\lessdot a_i$$

$$a_i\doteq a_{i+1},\cdots,a_{j-1}\doteq a_j$$

$$a_j\gtrdot a_{j+1}$$

即

$$a_{i-1}\lessdot a_i\doteq a_{i+1},\cdots,a_{j-1}\doteq a_j\gtrdot a_{j+1}$$

此定理告诉我们,出现在 a_i 和 a_j 之间的终结符一定属于该素短语。从语法树和最左素短语的定义可知,在算符优先分析中,每次归约的都是当前句型的最左素短语,它无法归约由单个非终结符组成的可归约串(如 E→T),因为单个非终结符不能构成最左素短语。

算符优先分析的实质是在归约时用优先关系来指导最左素短语的选择,优先性低于$\lessdot$用来标识最左素短语的头,优先性高于$\gtrdot$用来标识最左素短语的尾。

实现算符优先分析过程仍然采用移进-归约方式,使用一个符号栈和一个输入缓冲区,当前句型表示为:

符号栈内容 + 输入缓冲区内容 = # 当前句型 #

算符优先分析过程描述如下。

(1) 开始:符号栈中为#,输入缓冲区为:输入串#(以#结束)

(2) 移进-归约。

① 从左向右扫描输入符号并移进栈,查找算符优先关系表,直至找到某个 j 满足 $a_j\gtrdot a_{j+1}$时为止。

② 从 a_j 开始往左扫描符号栈,直至找到某个 i 满足 $a_{i-1}\lessdot a_i$ 为止。

③ $N_ia_i\cdots N_ja_jN_{j+1}$形式的子串就构成最左素短语,用相应产生式进行归约。

(3) 结束:如果符号栈中为#S,输入缓冲区为#,分析成功;否则失败。

在此注意,$N_ia_i\cdots N_ja_jN_{j+1}$,构成的最左素短语和某个产生式的右部可能不完全相同,此处只需要终结符相同,非终结符的位置相同即可归约。这也是算符优先分析方法和别的方法不同的地方。

例 3.30 利用算符优先分析方法判断输入符号串 i+i*i# 是否是文法 G3.2 的句子。

文法 G3.2 的算符优先关系表如表 3.9 所示,i+i*i#的算符优先分析过程如表 3.12 所示。

表 3.12 i+i*i#的算符优先分析过程

步 骤	栈	输入缓冲区	最左素短语	说 明
(1)	#	i+i*i#		初始状态
(2)	#i	+i*i#		#$\lessdot$i,i 入栈
(3)	#F	+i*i#	i	#$\lessdot$i$\gtrdot$+,用 F→i 归约
(4)	#F+	i*i#		#$\lessdot$+,+入栈
(5)	#F+i	*i#		+$\lessdot$i,i 入栈
(6)	#F+F	*i#	i	+$\lessdot$i$\gtrdot$*,用 F→i 归约
(7)	#F+F*	i#		+$\lessdot$*,*入栈

续表

步　骤	栈	输入缓冲区	最左素短语	说　明
(8)	#F+F*i	#		*<·i,i 入栈
(9)	#F+F*F	#	i	*<·i·>#,用 F→i 归约
(10)	#F+T	#	F*F	+<·*·>#,用 T→T*F 归约
(11)	#E	#	F+T	#<·+·>#,用 E→E+T 归约

从该过程可以看出,算符优先分析不是一种严格的规范归约。在整个归约过程中,归约只检查句型中自左至右的终结符序列的优先关系,不涉及终结符之间可能存在的非终结符,即实际上可以认为这些非终结符是相同的。在寻找最左素短语时,只要自左到右终结符和非终结符的位置相同、对应的终结符相同即可。

算符优先分析过程可用算法 3.7 进行形式化描述。在算法中使用了一个符号栈 S,用来存放在分析过程中使用的终结符和非终结符,top 指示栈顶的位置。

算法 3.7　利用算符优先关系表进行分析的过程

输入:文法 G 的算符优先关系表、输入符号串 w

输出:如果 w 是文法的句子,输出归约过程,否则输出错误

步骤:

(1) top=1; S[top]='#';　　//初始化

(2) 把当前输入符号读进 a;

(3) 如果 S[top]∈V_T,则设 j=top,否则设 j=top−1;　// j 指向终结符

　　//终结符之间最多只有一非终结符,故若 S[top]∉V_T 则 S[top−1]∈V_T

(4) 比较栈顶符号 S[j]与输入符号 a 的优先关系

① 如果 S[j]<· a 或 S[j]=· a,则　　//栈顶 S[j]<· a 或 S[j]=· a

　　//则 a 入栈,栈顶上移

　　top=top+1;

　　S[top]=a;

② 如果 S[j]·> a　　//S[j]·> a,找到最左素短语

　　//的尾

　　循环执行下述操作,直到 S[j]<· Q。

　　Q=S[j];　　//循环向前查找最左素短

　　//语的头,用 j 记录

　　如果(S[j−1]∈V_T),则 j=j−1,否则 j=j−2;　//查找栈中的终结符

　　//循环退出时找到了最左

　　//素短语的头

　　把 S[j+1]…S[top]归约为某个非终结符 N;

　　top=j+1;　　//最左素短语出栈

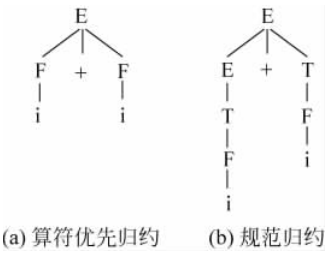
　　S[top]=N;　　//将归约后的非终结符 N

　　//入栈

(5) 若栈中为#S,且 a='#',则分析成功,否则 a 入栈,转(2)。

此算法工作过程中,若出现j减1后其值小于或等于0,则意味着输入串有错。在正确的情况下,算法工作结束时符号栈将呈现#S#。

在文法G3.2中,用算符优先分析方法分析句子i+i,归约过程是:先将第一个最左素短语i归为F,然后把第二次归约的最左素短语i(第二个i)也归为F,第三次把最左素短语F+F归约为E,语法树如图3.20(a)所示。对规范归约来说,其归约过程是:先把第一个i归为F,接着将F归为T,再将T归为E;然后重复相同的过程把第二个i归为F,再将F归为T;最后将E+T归为E,语法树如图3.20(b)所示。



(a) 算符优先归约 (b) 规范归约

图3.20 句子i+i的两种归约的语法树

因此,算符优先分析比规范归约要快,因为算符优先分析只与终结符之间的优先关系有关,非终结符对归约没有影响,甚至对非终结符可直接跳过不进行归约,即跳过了所有形如P→Q的右部仅含单个非终结符的产生式,跳过的非终结符不进入符号栈。如i+i*i的LL(1)分析过程需要17步,如表3.4所示,规范归约需要14步,如表3.8所示,算符优先分析过程只需要11步,如表3.12所示。算符优先分析也有缺点,有可能把本来不构成句子的输入串也误认为是句子,但这种缺点易于从技术上加以弥补。

***4. 算符优先函数**

用优先关系表来表示每对终结符之间的优先关系存储量大、查找费时。实际使用中,一般不直接使用优先关系表,而是使用优先函数。如果给每个终结符赋一个值(即定义终结符的一个函数f),值的大小反映其优先关系,则终结符对a、b之间的优先关系就转换为两个优先函数f(a)与f(b)的值的比较。

一个终结符在栈中(左)与在输入串中(右)的优先值是不同的。例如,既存在着+⋗)又存在着)⋗+。因此,对一个终结符a而言,它应该有一个左优先数f(a)和一个右优先数g(a),这样就定义了每个终结符的一对函数值。

根据一个文法的算符优先关系表,将每个终结符θ与两个自然数$f(\theta)$和$g(\theta)$对应,如果$f(\theta)$和$g(\theta)$的选择满足如下关系:

若$\theta_1 \lessdot \theta_2$,则$f(\theta_1)<g(\theta_2)$;

若$\theta_1 \doteq \theta_2$,则$f(\theta_1)=g(\theta_2)$;

若$\theta_1 \gtrdot \theta_2$,则$f(\theta_1)>g(\theta_2)$。

则称f和g为优先函数(Precedence Function)。其中,f称为入栈优先函数,g称为比较优先函数。

定义了优先函数后,算符优先分析法中两个终结符之间优先关系的比较就可用优先函数来代替了,这既便于做比较运算,又能节省存储空间。虽然实现容易,但优先函数有一个缺点,就是原先不存在优先关系的两个终结符,由于与自然数相对应就变成可比较的了,这样可能会掩盖输入串中的错误。但可以通过检查栈顶符号θ和输入符号a的具体内容来发现那些原先不能构成正确句子的情形。

注意,由于优先函数与自然数对应,对给定的文法,如果存在优先函数,一定存在多个,即f和g的选择不是唯一的;也有许多优先关系表不存在对应的优先函数。例如,表3.13给出的优先关系表就不存在优先函数。

表 3.13 不存在优先函数的优先关系表

	a	b
a	$\doteq$	$\gtrdot$
b	$\doteq$	$\doteq$

在表 3.13 中，假定存在 f 和 g，则应有：

$$f(a)=g(a) \qquad f(a)>g(b) \qquad f(b)=g(a) \qquad f(b)=g(b)$$

这将导致如下矛盾：

$$f(a)>g(b)=f(b)=g(a)=f(a)$$

如果某文法的优先函数存在，那么，根据优先关系表构造优先函数 f 和 g 的一个简单方法是关系图法。关系图法就是用图的方式来表示两个函数 f 和 g 的关系。用它求优先函数的过程可以描述为算法 3.8。

算法 3.8 根据算符优先关系表构造优先函数

输入：文法 G 的算符优先关系表

输出：优先函数 f 和 g

步骤：

(1) 对所有终结符 a(包括 #)，用有下标的 f_a、g_a 为结点名，画出全部 n 个终结符所对应的 2n 个结点。

(2) 若 $a \gtrdot b$ 或 $a \doteq b$，则画一条从 f_a 到 g_b 的带箭头的弧线；若 $a \lessdot b$ 或 $a \doteq b$，则画一条从 g_b 到 f_a 的带箭头的弧线。

(3) 检查构造的图中是否存在环路。如果存在环路，就不存在优先函数；如果不存在环路，就存在优先函数。

(4) 当存在优先函数时，对每个结点都赋予一个数，此数等于从该结点出发所能到达的结点(包括出发结点自身在内)的个数，赋给 f_a 的数作为 f(a)，赋给 g_b 的数作为 g(b)。

例 3.31 利用算法 3.8，求下面的文法 G 的优先函数。

(1) E→E+T|T。

(2) T→T * F|F。

(3) F→i。

(1) 根据算法 3.6 可以求得文法 G 的算符优先关系表，如表 3.14 所示。

表 3.14 文法 G 的算符优先关系表

	+	*	i	#
+	$\gtrdot$	$\lessdot$	$\lessdot$	$\gtrdot$
*	$\gtrdot$	$\gtrdot$	$\lessdot$	$\gtrdot$
i	$\gtrdot$	$\gtrdot$		$\gtrdot$
#	$\lessdot$	$\lessdot$	$\lessdot$	$\doteq$

(2) 构造表 3.14 的关系图，先构造用终结符作为下标的 8 个结点：f_+、f_*、f_i、$f_\#$、g_+、

g_*、g_i、$g_\#$，然后根据算符优先关系表画相应的弧线，如查表得到$+ \gtrdot +$，则从 f_+ 到 g_+ 画一条带箭头的弧线；又如查表得到$+ \lessdot *$，则从 g_* 到 f_+ 画一条带箭头的弧线，……，算符优先关系表中总共有 15 个优先关系，则总共要画 15 条弧线，如图 3.21 所示。

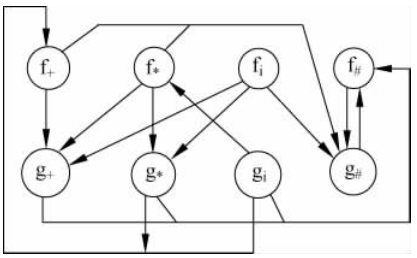


图 3.21　文法 G 的关系图

(3) 由图 3.21 所示的关系图，可以求各终结符的优先函数，如 f_+ 的值为 4，是从 f_+ 出发沿箭头指向的方向能到达的结点数(自身也算在内)；g_i 的值为 7，是从 g_i 出发沿箭头指向的方向能到达的结点数(自身也算在内)，……。最终如表 3.15 所示。

表 3.15　文法 G3.2 的优先函数

	+	*	i	#
f	4	6	6	2
g	3	5	7	2

3.4.3　LR 分析法

LR 分析法是一种自下而上符合规范归约的语法分析方法，L 表示从左到右扫描输入符号串，R 表示构造一个最右推导的逆过程。其功能强大，适用于一大类文法。LR 分析法比递归下降分析法、LL(1)分析法和算符优先分析法对文法的限制要少得多，对大多数用无二义的上下文无关文法描述的语言都可以用 LR 分析器予以识别，而且速度快，并能准确、及时地指出输入串的任何语法错误及出错位置。它的缺点是对于一个实用的程序设计语言的分析器的构造工作量相当大，实现复杂。

LR 分析法分为四种：第一种是 LR(0)分析，它使用简单的方法构造分析表，分析表不大，分析简单，容易实现，功能最弱，局限性很大，只对无冲突的文法有效，但它是进行其他 LR 分析的基础；第二种是简单的 LR 方法(SLR)，它是在 LR(0)分析的基础上，向前查看一个输入符号，这种方法较易实现，分析表和 LR(0)分析表大小相同，分析能力强于 LR(0)，有较高的实用价值；第三种是规范的 LR 方法，分析能力最强，适用于大多数上下文无关文法，但分析表体积庞大，代价很高；第四种是向前看的 LR 方法(LALR)，其分析能力和代价介于 SLR 和规范的 LR 之间，它可用于大多数程序设计语言的文法，并可高效地实现。

1. LR 分析概述

1) LR 分析的基本思想

在 3.4.1 节中已经讨论过，自下而上分析是一种移进-归约过程，在分析过程中，若栈顶符号串形成句柄就进行归约，因此自下而上分析法的关键是在分析过程中如何确定句柄。

在LR分析法中，根据当前分析栈中的符号串（通常以状态表示）和向右顺序查看输入串的K（本节中K=0和1）个符号就可唯一地确定分析动作是移进还是归约，以及用哪个产生式归约，因而也就能唯一地确定句柄。

LR分析的基本思想是：在规范归约过程中，一方面用栈存放已移进和归约出的整个符号串，即记住“历史”；另一方面，LR分析器还要面对“现实”的当前输入符号，再根据所用产生式推测未来可能碰到的输入符号，即对未来的“展望”。当某可归约符号串出现在栈顶时，需要根据已记载的“历史”“展望”和“现实”的输入符号三方面的内容来决定栈顶的符号串是否构成了真正的句柄，是否能够进行归约。

2）LR分析器的构成

LR分析器由三部分构成：LR分析程序、分析栈（包括文法符号栈和相应的状态栈）和分析表，如图3.22所示。

（1）LR分析程序即总控程序，也称为驱动程序，用于控制分析器的动作。对所有的LR分析器，LR分析程序都是相同的。其工作过程很简单，它的任何一步动作都是根据栈顶状态和当前输入符号去查分析表，完成分析表中规定的动作。

（2）分析栈的结构如图3.23所示。它将“历史”和“展望”综合成“状态”。栈里的每个状态概括了从分析开始直到某一归约阶段的全部历史和展望资料，分析时不必像算符优先分析法那样必须翻阅栈中的内容才能决定是否要进行归约，只需根据栈顶状态和现行输入符号就可以唯一决定下一个动作。显然，文法符号栈是多余的，它已经概括到状态栈里了，保留在这里是为了让大家更加明确归约过程。S_0和#是分析开始前预先放入栈里的初始状态和句子括号；栈顶状态为S_m，符号串$X_1X_2\cdots X_m$是至今已移进-归约出的文法符号串。

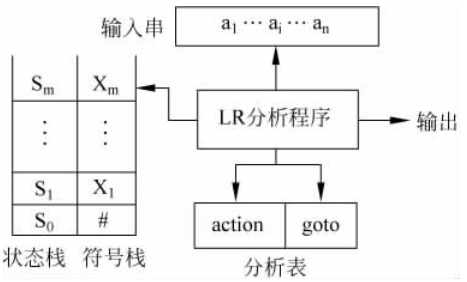


图3.22 LR分析器框图

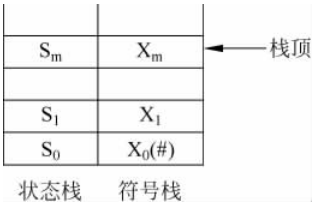


图3.23 LR分析器中分析栈的结构

（3）分析表是LR分析器的核心部分。不同文法的分析表不同，同一个文法采用的LR分析方法不同时，分析表也不同。分析表分为action（动作）表和goto（状态转换）表两部分，以二维数组表示，如文法G3.2的一个LR分析表如表3.16所示。为了在归约时使用文法的产生式编号，将文法G3.2改写为G3.2′：

（1）E → E+T

（2）E → T

（3）T → T * F

（4）T → F

（5）F → (E)

（6）F → i （G3.2′）

在LR分析表中,action[S,a]表示当状态为S面临输入符号a时应采取的动作。每一项action[S,a]规定了如下四种动作之一。

- 移进:表中用S_i来表示,当前输入符号a进符号栈,下一输入符号变成当前输入符号,当前状态i(即S_i的下标)入栈。
- 归约:表中用r_j来表示,按第j个产生式进行归约(j为文法G3.2′的编号)。
- 接受:表中用acc来表示,分析成功,停止分析器的工作。
- 报错:表中的空白部分,表示发现源程序含有错误,调用出错处理程序。

在LR分析表中,goto[S,X]表示状态S面对文法符号X时的下一状态(X是终结符和非终结符,显然,goto[S,X]定义了一个以文法符号为字母表的DFA。为了减少分析表的占用空间,在表示各个分析表时,已将X为终结符号的goto表与action表合并)。

表3.16 文法G3.2的LR分析表

状态	action						goto		
	i	+	*	(	)	#	E	T	F
0	S_5			S_4			1	2	3
1		S_6				acc			
2		r_2	S_7		r_2	r_2			
3		r_4	r_4		r_4	r_4			
4	S_5			S_4			8	2	3
5		r_6	r_6		r_6	r_6			
6	S_5			S_4		r_1		9	3
7	S_5			S_4					10
8		S_6			S_{11}				
9		r_1	S_7		r_1	r_1			
10		r_3	r_3		r_3	r_3			
11		r_5	r_5		r_5	r_5			

3) LR分析过程

为了更好地理解分析过程,在分析过程中,仍然将符号栈的变化展示出来,这样LR的分析过程就可以用下面的三元式的变化来表示:

(状态栈,符号栈,剩余输入符号串)

这个三元式分别表示分析过程中某时刻的状态栈、符号栈和输入符号串的内容。

初始时,将状态S_0和#压入状态栈和符号栈。此时的三元式为:

$$(S_0,\#,a_1a_2\cdots a_n\#)$$

分析过程中任一时刻可以用如下三元式来表示:

$$(S_0S_1\cdots S_m,\#X_1X_2\cdots X_m,a_ia_{i+1}\cdots a_n\#)$$

分析器下一步的动作是根据栈顶状态S_m和当前输入符号a_i查action表,根据表中的内容完成相应的动作,从而引起三元式的变化。算法描述如算法3.9所示。

算法3.9 LR分析过程

输入:LR分析表、输入符号串a

输出:若a是文法的句子,输出归约过程,否则输出错误

步骤：

(1) 将状态 0 和 # 压入状态栈和符号栈。

(2) 根据栈顶状态 S_m 和当前输入符号 a_i 查 action 表，分四种情况。

① 移进。在 action 表中，若 action[S_m，a_i]=S_k，执行的动作是：当前输入符号 a_i 进符号栈，下一输入符号变为当前输入符号，状态 k 进状态栈。三元式变为

$$(S_0S_1\cdots S_mk,\ \#X_1X_2\cdots X_ma_i,\ a_{i+1}\cdots a_n\#)$$

② 归约。若 action[S_m，a_i]=r_j，按某个产生式 j：A→β 进行归约，假定产生式的右端 β 的长度为 r，则两个栈顶的 r 个元素同时出栈。将归约后的符号 A 进符号栈；根据 goto 表，把(S_{m-r}，A)的下一状态 k=goto[S_{m-r}，A]进状态栈。三元式变为

$$(S_0S_1\cdots S_{m-r}k,\ \#X_1X_2\cdots X_{m-r}A,\ a_ia_{i+1}\cdots a_n\#)$$

归约的动作不改变现行输入符号，执行归约的动作意味着 $\beta(X_{m-r+1}\cdots X_m)$ 已呈现于栈顶且是一个相对于 A 的句柄。

③ 接受。若 action[S_m，a_i]=acc，则宣布分析成功，分析器停止工作。三元式不再变化。

④ 报错。若 action[S_m，a_i]为空白，此时就发现了源程序中的错误，调用出错处理程序。三元式变化过程终止。

(3) LR 分析程序按上述方式查表控制三元式的变化，直至执行“接受”或“报错”为止。

例 3.32 利用表 3.16 的分析表，对输入串 i+i*i# 进行 LR 分析。

首先将状态 0 和 # 压入栈中，然后根据栈顶的状态和当前输入符号 a_i 查 action 表，状态栈、符号栈和当前输入串的变化情况具体如表 3.17 所示。

表 3.17 输入串 i+i*i# 的 LR 分析过程

序号	状态栈	符号栈	产生式	输入串	说明
1	0	#		i+i*i#	0 和 # 进栈
2	05	#i		+i*i#	i 和 S_5 进栈
3	03	#F	F→i	+i*i#	i 和 S_5 退栈，F 和 S_3 进栈
4	02	#T	T→F	+i*i#	F 和 S_3 退栈，T 和 S_2 进栈
5	01	#E	E→T	+i*i#	T 和 S_2 退栈，E 和 S_1 进栈
6	016	#E+		i*i#	+ 和 S_6 进栈
7	0165	#E+i		*i#	i 和 S_5 进栈
8	0163	#E+F	F→i	*i#	i 和 S_5 退栈，F 和 S_3 进栈
9	0169	#E+T	T→F	*i#	F 和 S_3 退栈，T 和 S_9 进栈
10	01697	#E+T*		i#	* 和 S_7 进栈
11	016975	#E+T*i		#	i 和 S_5 进栈
12	01697<u>10</u>	#E+T*F	F→i	#	i 和 S_5 退栈，F 和 S_{10} 进栈
13	0169	#E+T	T→T*F	#	F*T 和 S_{10}，S_7，S_9 退栈，T 和 S_9 进栈
14	01	#E	E→E+T	#	T+E 和 S_9，S_6，S_1 退栈，E 和 S_1 进栈

从这个分析过程可以看出，对 LR 文法，当分析器对输入串进行自左至右扫描时，一旦

句柄呈现于栈顶,就能及时对它进行归约。

在3.4.1节讨论的规范归约中,由栈中的文法符号和当前的输入符号来识别句柄。对一个LR分析器来说,栈顶的状态包含了分析所需要的一切"历史"和"展望"信息,因此LR分析器不需要扫描整个栈就知道什么时候句柄出现在栈顶。因此,可以用一个有穷自动机来确定栈顶的句柄。LR分析表的goto函数实质上就是这样的有穷自动机。

一个文法,如果能构造一个LR分析表,且它的每个入口均是唯一确定的,则把这个文法称为LR文法。并非所有上下文无关文法都是LR文法,但多数程序语言都可用LR文法来描述。

在有些情况下,LR分析器需要"展望"和实际检查未来的k个输入符号才能决定是采取"移进"还是"归约"。一般而言,一个文法如果能用一个每步最多向前检查k个输入符号的LR分析器进行分析,则这个文法就称为LR(k)文法。

虽然LR分析法有很多种,但各种LR分析法中的分析程序和分析表的形式都是相同的,差别在于分析表的内容,不同的文法和不同的LR分析法,其分析表都是不同的。因此,进行LR分析的关键是分析表的构造,下面分别介绍四种不同的LR分析法中分析表的构造。

2. LR(0)分析

1) 活前缀和项目

在第2章介绍过字的前缀,它是指该字的任意首部。例如,字abc的前缀有ε、a、ab或abc。活前缀(Viable Prefix)是指规范句型的一个前缀,它不含句柄之后的任何符号(即在规范句型中,句柄之前的部分和句柄构成的前缀)。即对于文法G,若有规范推导 $S\overset{*}{\Rightarrow}\delta A\omega$,且可继续规范推导出 $S\overset{*}{\Rightarrow}\delta\alpha\beta\omega$,其中,$\delta\in V^*$,$A\in V_N$,$\alpha\in V^+$,$\omega\in V_T^*$,则 $\alpha\beta$ 是 $\delta\alpha\beta\omega$ 的句柄,$\delta\alpha\beta$ 的任何前缀都是 $\delta\alpha\beta\omega$ 的活前缀。因为句柄是活前缀的后缀,识别活前缀就可以找到句柄;找到了句柄,就可以对句柄归约。

对一个文法G,可以构造一个DFA来识别G的所有规范句型的活前缀。在此基础上,将它自动转换成LR分析表。

在LR分析的任何时候,栈里的文法符号(自栈底向上)$X_1X_2\cdots X_m$应该构成活前缀,把输入串的剩余部分匹配于其后即应成为规范句型(如果整个输入串为一个句子的话)。因此,在规范归约过程中的任何时刻只要已分析过的部分(即在符号栈中的符号串)一直保持为可归约成某个活前缀,就表明输入串已被分析过的部分没有发现语法错误。加上输入串的剩余部分,恰好就是活前缀所属的规范句型。一旦栈顶出现句柄,就被归约成某个产生式的左部符号,所以活前缀不包括句柄之后的任何符号。用"项目"来表示分析过程中已经分析过的部分。

为了表征句柄与活前缀间的关系,即句柄是否已在当前活前缀中出现,以及句柄中已有多少符号出现在栈中,需引入LR(0)项目的概念。文法G的一个LR(0)项目(简称项目)是在G的某个产生式右部的某个位置添加一个圆点。例如,产生式A→XYZ对应有四个项目:

(1) A→·XYZ;

(2) A→X·YZ;

(3) A→XY·Z;

(4) A→XYZ·。

产生式 A→ε 只对应一个项目 A→·。一个项目指明了在分析过程中的某个时刻,已经看到产生式所能推出的字符串的多大一部分。如上例中第一个项目意味着,希望能从输入串中看到 XYZ 推出的符号串;第二个项目意味着,已经从输入串中看到从 X 推出的符号串,希望能从后面的输入串进一步看到从 YZ 推出的符号串;最后一个项目表示,已经从输入串中看到从 XYZ 推出的全部符号串,此时可以将 XYZ 归约为 A。

若干个项目组成的集合,称为项目集。例如,对于上述产生式的四个项目即构成一个项目集。

例 3.33 求文法 G3.6 的所有项目。

0. S′ → E
1. E → aA|bB
2. A → cA|d
3. B → cB|d (G3.6)

针对该文法的每个产生式写出对应的项目:

1. S′→·E	2. S′→E·	
3. E→·aA	4. E→a·A	5. E→aA·
6. A→·cA	7. A→c·A	8. A→cA·
9. A→·d	10. A→d·	
11. E→·bB	12. E→b·B	13. E→bB·
14. B→·cB	15. B→c·B	16. B→cB·
17. B→·d	18. B→d·	

项目中的圆点用来指示识别位置,圆点之左是在分析栈栈顶的已识别部分,圆点之右是期待从输入符号串中识别的符号串(可以把圆点理解为栈内外的分界点)。

如果项目 i 和项目 j 出自同一产生式,而且项目 j 的圆点只落后于项目 i 一个位置,则称项目 j 是项目 i 的后继项目。如在例 3.33 中,项目 2 是项目 1 的后继项目,项目 4 是项目 3 的后继项目。在一个项目中紧跟在圆点后面的符号称为该项目的后继符号,表示下一时刻将会遇到的符号。

可以根据圆点所在的位置和后继符号的类型把项目分为以下几种。

(1) 归约项目:凡圆点在最右端(即后继符号为空)的项目,如 A→α·,表明一个产生式的右部已分析完,句柄已形成,可以归约。

(2) 接受项目:对文法的开始符号 S′的归约项目,如 S′→α·,表明已分析成功。

(3) 移进项目:后继符号为终结符的项目,如 A→α·aβ(其中 a 为终结符),分析动作是把 a 移进符号栈。

(4) 待约项目:后继符号为非终结符的项目,如 A→α·Bβ(其中 B 为非终结符),它表明所对应的项目等待将非终结符 B 所能推出的串归约为 B,才能继续向后分析。

由此可知,句柄、LR(0)项目与活前缀间的关系有如下三种。

(1) 当句柄 α 已完全出现在规范句型的活前缀之中,即 α 作为活前缀的一个后缀出现于分析栈的栈顶,则相应的 LR(0)项目为 A→α·,并将其称为归约项目,因为此时应按产生式 A→α 归约活前缀中的句柄 α。

(2) 当句柄的一个真前缀 β_1 已出现于分析栈的栈顶,即活前缀中仅含有句柄的一部分符号,则相应的 LR(0)项目为 $A\rightarrow\beta_1\cdot\beta_2$,此时期望能从余留的输入串形成句柄的后缀 β_2。于是,若 β_2 形如 $X\beta$,当 $X\in V_T$ 时,相应的分析动作自然是将正扫描的输入符号移进栈中,故将相应的 LR(0)项目 $A\rightarrow\beta_1\cdot X\beta$ 称为移进项目;而当 $X\in V_N$ 时,期望通过从余留的输入符号中归约出非终结符号 X,故将相应的 LR(0)项目 $A\rightarrow\beta_1\cdot X\beta$ 称为待约项目。

(3) 当活前缀中不含有句柄 α 的任何符号时,相应的 LR(0)项目为 $A\rightarrow\cdot\alpha$,显然它是上述第二类 LR(0)项目当 $\beta_1=\varepsilon$ 时的特殊情形。

现在将每一个项目看作一个状态,构造识别一个文法所有活前缀的 DFA。这个 DFA 的所有状态(项目集)称为这个文法的 LR(0)项目集规范族。LR(0)项目集规范族是构造 LR(0)分析表的基础。

为了构造文法 G 的 LR(0)项目集规范族,使终结状态易于识别,首先对原文法进行拓广。设原文法 G 的开始符号为 S,增加产生式 $S'\rightarrow S$,得到拓广文法 G',S' 为 G' 的开始符号。拓广文法是为了对某些右部含有开始符号的文法,在归约过程中能分清是已归约到文法的最初开始符,还是文法右部出现的开始符号,拓广文法的开始符号 S' 只在左部出现,确保了不会混淆。在拓广文法 G' 中,有且仅有一个接受项目 $S'\rightarrow S\cdot$,这就是唯一的接受态。

构造 LR(0)项目集规范族的方法有两种。

(1) 列出拓广文法的所有项目,构造其 NFA,再用第 3 章介绍的子集法将其确定化为 DFA。这种方法工作量较大。

(2) 使用类似于第 3 章的闭包和状态转换函数的概念,直接进行构造。

2) 通过构造 NFA,并确定化的方法来构造识别活前缀的 DFA

根据第 3 章中构造 DFA 的思想,构造 LR(0)项目集规范族和识别活前缀的 DFA,也可先构造识别文法 G 的项目集规范族和识别活前缀的 NFA,然后利用第 3 章介绍的子集法将 NFA 进行确定化。

构造识别文法 G 的活前缀的 NFA 的基本思想是:根据后继关系将某状态和其后继状态用弧线连接起来,步骤如下。

(1) 写出文法的所有项目,每个项目作为一个状态。

(2) 规定项目 1:$S'\rightarrow\cdot S$ 为 NFA 的唯一初态。

(3) 如果状态 i 和状态 j 出自同一产生式,状态 j 是状态 i 的后继状态,假定状态 i 为

$$X\rightarrow X_1\cdots X_{i-1}\cdot X_i\cdots X_n$$

而状态 j 为

$$X\rightarrow X_1\cdots X_i\cdot X_{i+1}\cdots X_n$$

那么①如果 X_i 是终结符 a,则从状态 i 画一条弧到状态 j,标记为 a。②如果 X_i 是非终结符 A,则画两种类型的弧线:从状态 i 画一条弧到状态 j,标记为 A;从状态 i 画 ε 弧到所有的 $A\rightarrow\cdot\beta$ 的状态(所有圆点出现在产生式右部最左边的、左部是非终结符 A 的项目)。

(4) 归约项目表示结束状态(句柄识别态),用双圈表示,双圈外有 * 号者表示句子接受态。

例 3.34 构造文法 G3.6 的 LR(0)项目集规范族。

对于文法 G3.6,首先写出文法 G 的所有项目,如例 3.33 所示,每个项目就是一个状

态；根据上面介绍的步骤构造识别文法 G 的所有活前缀的 NFA，如图 3.24 所示，图中的状态编号与项目编号对应。

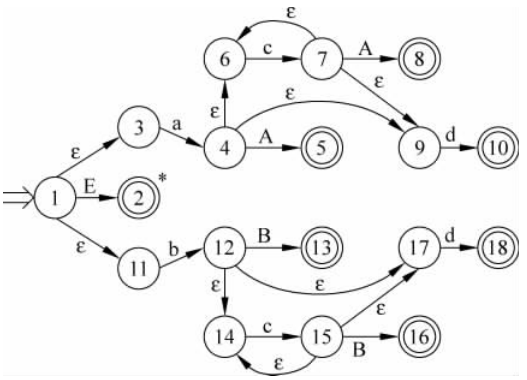


图 3.24 识别文法 G3.6 活前缀的 NFA

然后可以使用第 3 章介绍的子集法，把 NFA 确定化，得到一个以项目集为状态的 DFA，它是建立 LR 分析表的基础。图 3.25 是图 3.24 对应的 DFA。

该 DFA 中的每个状态用一个方框来表示，方框中的每个状态代表原来 NFA 的状态集合，即已经把 NFA 的多个状态合并成了 DFA 中的一个状态，并将其中包含的 NFA 的状态都写明了。

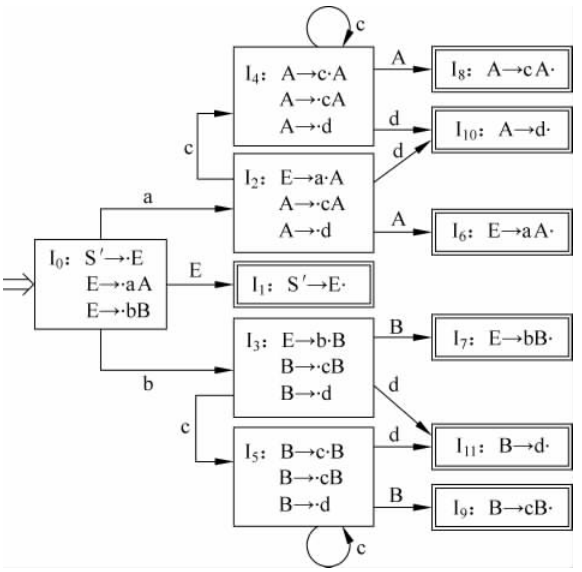


图 3.25 识别文法 G3.6 的活前缀的 DFA

3）使用闭包和状态转换函数来构造识别活前缀的 DFA

为了能直接从文法生成 LR(0)项目集规范族，设 I 是文法 G′的任一项目集，首先来介绍与项目集 I 有关的两个概念。

(1) 项目集 I 的闭包，表示为 Closure(I)，是从 I 出发由下面两条规则构造的项目集：

- 初始时，把 I 的每个项目都加入到 Closure(I)中；
- 如果 A→α·Bβ 在 Closure(I)中，将所有不在 Closure(I)中的形如 B→·γ 的项目加

入 Closure(I)中；重复执行这条规则，直至没有更多的项目可加入到 Closure(I)为止。

在构造 Closure(I)时需要注意，对任何非终结符 B，若某个圆点在左边的项目 B→·γ 进入到 Closure(I)，则 B 的所有形如 B→·β 的项目也要加入 Closure(I)中。

例 3.35 对于文法 G3.6，设 I={A→c·A}，则 Closure(I)={A→c·A，A→·cA，A→·d}，这就是图 3.25 中 I_4 的项目集。

(2) 项目集 I 的状态转换函数，表示为 GO(I,X)，也称为 I 的后继状态，它表示在状态 I 面临文法符号 X(终结符或非终结符)应该转移到的状态。函数 GO(I,X)定义为

$$GO(I, X) = Closure(\ move(I, X)\ )$$

其中，move(I, X)={任何形如 A→αX·β 的项目 | A→α·Xβ∈I}，即 A→αX·β 是 A→α·Xβ 的后继项目，它们源于同一个产生式，仅圆点相差一个位置，即由项目集 I 出发经标记为 X 的弧，到达的状态为 GO(I, X)。

我们说一个项目 A→β_1·β_2 对活前缀 $\alpha\beta_1$ 是有效的，其条件就是存在规范推导 $S\overset{*}{\Rightarrow}\alpha A\omega\Rightarrow\alpha\beta_1\beta_2\omega$。一般而言，同一个项目可能对好几个活前缀都是有效的。若归约项目 A→β_1·对活前缀 $\alpha\beta_1$ 是有效的，则它告诉我们应把符号串 β_1 归约为 A，即把活前缀 $\alpha\beta_1$ 变成 αA；若移进项目 A→β_1·β_2 对活前缀 $\alpha\beta_1$ 是有效的，则它告诉我们句柄尚未形成，下一步动作应该是移进。直观上说，若 I 是对某个活前缀 γ 有效的项目集，则 GO(I,X)就是对 γX 有效的项目集。

例 3.36 对文法 G3.6，令 I={S′→·E，E→·aA，E→·bB}，即图 3.25 中的项目集 I_0，求 GO(I,a)。

GO(I,a)就是检查 I 中所有那些圆点之后紧跟着 a 的项目，如项目 E→·aA，把这个项目的圆点右移一位，得到项目 E→a·A，于是 move(I, X)={E→a·A}，再对它求闭包，得到 GO(I,a)=Closure({ E→a·A })={E→a·A，A→·cA，A→·d}，就是图 3.25 中的项目集 I_2。

通过项目集的闭包和状态转换函数可以很容易地构造拓广文法 G′的 LR(0)项目集规范族和识别活前缀的 DFA，步骤如下。

(1) 设项目集 Closure({S′→·S})为该 DFA 的初态。

(2) 对初态集或其他已构造出的项目集使用状态转换函数 GO(I,X)，求出新的项目集，X 为项目集 I 的所有后继符号，并在 I 和 GO(I,X)之间添加弧线，标记为 X。重复该步骤直到不出现新的项目集为止。整个过程可以描述为算法 3.10。

算法 3.10 构造 LR(0)项目集规范族和识别活前缀的 DFA

输入：文法 G

输出：文法 G 的 LR(0)项目集规范族及识别活前缀的 DFA

步骤：

(1) 拓广文法，添加产生式 0：S′→S，并写出全部项目。

(2) 设 C= {Closure ({S′→·S}) }。

(3) 对 C 中的每个项目集 I 和每个文法符号 X，求 GO(I, X)。

- 如果 GO(I, X)≠Φ 且 GO(I, X)∉C，把 GO(I, X)加入 C 中。
- 在 I 和 GO(I, X)之间添加标记为 X 的弧线。

(4) 重复执行(3)，直到 C 中项目集不再增加。

根据算法 3.10,重新构造文法 G3.6 的项目集规范族和识别活前缀的 DFA,结果如图 3.25所示。项目集规范族 C 中共有 12 个项目集,GO()函数将它们连接成一个识别文法 G3.6 的活前缀的 DFA,其中 I_0 为初态,I_1 为接受态。显然两种构造方法产生的结果是相同的。

4) LR(0)分析表的构造

LR(0)分析表是 LR(0)分析器的重要组成部分,它是 LR 分析器完成动作的依据,可以根据该文法的识别活前缀的 DFA 来构造。

LR(0)分析表用一个二维数组表示,行是所有的状态编号,列是文法符号和#号。分析表的内容由两部分组成,一部分为 action(动作)表,它表示当前状态面临某个输入符号应做的动作是移进、归约、接受或出错,动作表的列只包含终结符和#;另一部分为 goto(转换)表,它表示在当前状态下面临文法符号时应转向的下一个状态,goto 表的列只包含非终结符(其实终结符的转换已经包含在 action 表中了)。

一个项目集中可能包含移进项目、归约项目、待约项目或接受项目四种项目中的一种或几种,但是一个 LR(0)项目集中不能有下列情况存在。

(1) 移进项目和归约项目同时存在。

若项目集形如{A→α·aβ,B→γ·},这时不管面临哪个输入符号都不能确定移进 a 还是把 γ 归约为 B,因为 LR(0)分析是不向前查看符号的,所以对归约的项目不管当前符号是什么都应归约。在一个项目集中同时存在移进和归约项目时称该状态含有移进-归约冲突(Shift-Reduce Conflict)。

(2) 归约项目和归约项目同时存在。

若项目集形如{A→β·,B→γ·},这时不管面临哪个输入符号都不能确定归约为 A,还是归约为 B。在一个项目集中同时存在两个或两个以上归约项目时称该状态含有归约-归约冲突(Reduce-Reduce Conflict)。

如果一个文法的 LR(0)项目集规范族中不存在移进-归约或归约-归约冲突时,称这个文法为 LR(0)文法。

对于 LR(0)文法,可直接从它的项目集规范族 C 和识别活前缀的 DFA 构造出 LR(0)分析表,算法描述如算法 3.11 所示。假定 $C=\{I_0,I_1,\cdots,I_n\}$,为简单起见,直接用 k 表示项目集 I_k 对应的状态,令包含项目 $S'\rightarrow\cdot S$ 的状态为分析器的初态。

算法 3.11 构造 LR(0)分析表

输入:文法 G 的项目集规范族和识别活前缀的 DFA

输出:LR(0)分析表

步骤:

(1) 若项目 $A\rightarrow\alpha\cdot X\beta\in I_k$ 且 $GO(I_k, X)=I_j$:

若 $X\in V_T$,则置 action[k, X]=S_j,即将(j,a)进栈;

若 $X\in V_N$,则置 goto[k, X]=j。

(2) 若项目 $A\rightarrow\alpha\cdot\in I_k$,则对任何 $a\in V_T$(或结束符#),置 action[k, a]=r_j(设 $A\rightarrow\alpha$ 是文法 G′的第 j 个产生式),即用 $A\rightarrow\alpha$ 归约。

(3) 若项目的 $S'\rightarrow S\cdot\in I_k$,则置 action[k, #]=acc,即接受。

(4) 分析表中凡不能用步骤(1)~(3)填入的空白均置为“出错标志”。

由于LR(0)文法的项目集规范族的每个项目集不含冲突项目,因此按上述方法构造的分析表的每个入口都是唯一的(即不含多重定义)。称如此构造的分析表是一张LR(0)分析表,使用LR(0)分析表的分析器称作LR(0)分析器。

例3.37 构造文法G3.6的LR(0)分析表。

首先,文法G3.6是一个已经拓广后的文法。将各个产生式进行编号,改写为G3.6′。

(0) S′ → E
(1) E → aA
(2) E → bB
(3) A → cA
(4) A → d
(5) B → cB
(6) B → d (G3.6′)

其次,根据算法3.10构造项目集规范族和识别活前缀的DFA,如图3.25所示。从图中可以看出,所有项目集中均不含冲突项目,因此这个文法是一个LR(0)文法。

最后,根据算法3.11,得到LR(0)分析表如表3.18所示。

表3.18 文法G3.6的LR(0)分析表

状态	action					goto		
	a	b	c	d	#	E	A	B
0	S_2	S_3				1		
1					acc			
2			S_4	S_{10}			6	
3			S_5	S_{11}				7
4			S_4	S_{10}			8	
5			S_5	S_{11}				9
6	r_1	r_1	r_1	r_1	r_1			
7	r_2	r_2	r_2	r_2	r_2			
8	r_3	r_3	r_3	r_3	r_3			
9	r_5	r_5	r_5	r_5	r_5			
10	r_4	r_4	r_4	r_4	r_4			
11	r_6	r_6	r_6	r_6	r_6			

例3.38 考查表达式文法G3.2的拓广文法G3.7。

(0) S′→E
(1) E→E+T
(2) E→T
(3) T→T * F
(4) T→F
(5) F→(E)
(6) F→i (G3.7)

构造该文法的 LR(0)项目集规范族及识别活前缀的 DFA，如图 3.26 所示。

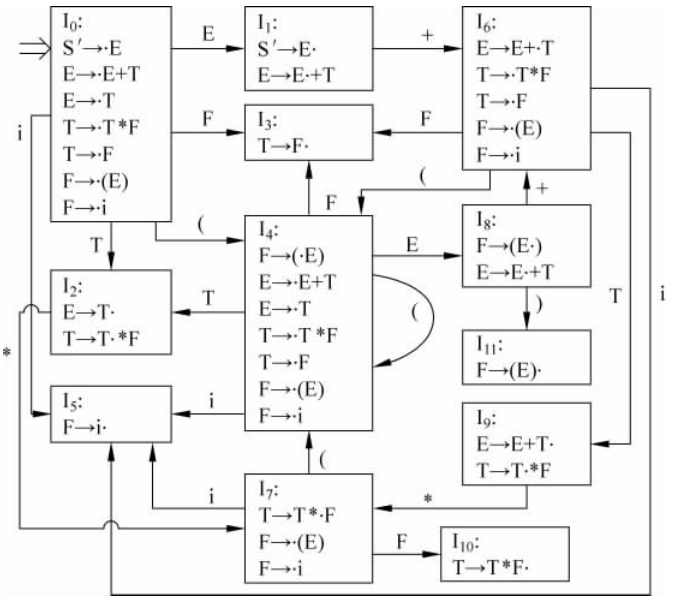


图 3.26 识别文法 G3.7 的活前缀的 DFA

在这 12 个项目集中，I_1，I_2，I_9 中存在移进-归约冲突，因而文法 G3.7 不是 LR(0)文法，不能构造 LR(0)分析表。下一小节将介绍 SLR 分析法来解决该冲突，对文法进行分析。

3. SLR(1)分析

1) SLR(1)文法

只有当一个文法 G 是 LR(0)文法，即识别 G 的活前缀的 DFA 中的每个状态都不出现冲突时，才能构造 LR(0)分析表。由于大多数实用的程序设计语言的文法都不能满足 LR(0)文法的条件，本节将介绍对于 LR(0)项目集规范族中有冲突的项目集用向前查看一个符号的办法来解决冲突。这种办法将能满足一部分文法的要求，因为只对有冲突的状态才向前查看一个符号，即查看 Follow 集，以确定完成哪个动作，称这种分析方法为简单的 LR(1)分析法，用 SLR(1)表示。

假定一个 LR(0)项目集规范族中有如下形式的项目集(状态)I：

$$I=\{X\rightarrow\alpha\cdot b\beta,\ A\rightarrow\gamma\cdot,\ B\rightarrow\delta\cdot\}$$

其中，α，β，γ，δ 为文法符号串，b 为终结符。在这个项目集中，第一个项目是移进项目，第二、三个项目是归约项目。在该项目集中含有移进-归约冲突和归约-归约冲突。那么只有在所有含有 A 或 B 的句型中，直接跟在 A 或 B 后的可能的终结符的集合(即 Follow(A)和 Follow(B))互不相交，且都不包含 b 时，才能唯一确定下一个动作，也就是说只有满足下面的条件：

$\text{Follow}(A)\cap\text{Follow}(B)=\Phi$

$\text{Follow}(A)\cap\{b\}=\Phi$

$\text{Follow}(B)\cap\{b\}=\Phi$

当在状态I面临某输入符号a时,才能采取如下“移进-归约”策略。

(1) 若a=b,则移进。

(2) 若a∈Follow(A),则用产生式A→γ进行归约。

(3) 若a∈Follow(B),则用产生式B→δ进行归约。

(4) 此外,报错。

通常而言,假设LR(0)项目集规范族中的项目集I中有m个移进项目:$A_1\rightarrow\alpha_1\cdot a_1\beta_1$,$A_2\rightarrow\alpha_2\cdot a_2\beta_2$,…,$A_m\rightarrow\alpha_m\cdot a_m\beta_m$和n个归约项目:$B_1\rightarrow\gamma_1\cdot$,$B_2\rightarrow\gamma_2\cdot$,…,$B_n\rightarrow\gamma_n\cdot$,那么,只要集合$\{a_1,a_2,\cdots,a_m\}$和Follow($B_1$),Follow($B_2$),…,Follow($B_n$)两两不相交,就可以通过检查当前输入符号a属于上述n+1个集合中的哪一个集合来解决冲突。即

(1) 若a∈$\{a_1,a_2,\cdots,a_m\}$,则移进。

(2) 若a∈Follow(B_i),i=1,2,…,n,则用产生式$B_i\rightarrow\gamma_i$进行归约。

(3) 此外,报错。

冲突性动作的这种解决方法叫作SLR(1)方法。如果某文法的LR(0)项目集规范族的项目集中存在的冲突都能用SLR(1)方法解决,称这个文法是SLR(1)文法,所构造的分析表为SLR(1)分析表。数字1的意思是,在分析过程中最多向前看一个符号。使用SLR(1)分析表的分析器称为SLR(1)分析器。

考查例3.38中的三个含有冲突的项目集。

在I_1中:$S'\rightarrow E\cdot$

$E\rightarrow E\cdot+T$

由于Follow(S')=$\{\#\}$,而$S'\rightarrow E\cdot$是唯一的接受项目,所以当且仅当面临句子的结束符#时,句子才被接受。又因$\{\#\}\cap\{+\}=\Phi$,因此I_1中的冲突可解决。

在I_2中:$E\rightarrow T\cdot$

$T\rightarrow T\cdot *F$

由于Follow(E)=$\{+,),\#\}$,Follow(E)$\cap\{*\}=\{+,),\#\}\cap\{*\}=\Phi$,因此面临输入符为+,)或#时,用产生式E→T进行归约;当面临输入符为*时,移进;其他情况则报错。

在I_9中:$E\rightarrow E+T\cdot$

$T\rightarrow T\cdot *F$

与I_2类似,由于Follow(E)$\cap\{*\}=\{+,),\#\}\cap\{*\}=\Phi$,因此面临输入符为+,)或#时,用产生式E→E+T进行归约;当面临输入符为*时,移进;其他情况则报错。

综上,例3.38中的冲突均可用SLR(1)方法解决。因此文法G3.7是SLR(1)文法。

2) SLR(1)分析表的构造

SLR(1)分析表的构造与LR(0)分析表的构造类似,只是需要在含有冲突的项目集中分别进行处理。

首先,构造出文法的LR(0)项目集规范族及识别活前缀的DFA,寻找DFA中有冲突的项目集,并对冲突项目集中计算归约项目左部非终结符的Follow集。假定项目集规范族C=$\{I_0,I_1,\cdots,I_n\}$,其中I_k为项目集的名字,k表示状态,令包含$S'\rightarrow\cdot S$项目的状态k为分析器的初态。那么,SLR(1)分析表的构造算法如算法3.12所示。

算法 3.12 构造文法的 SLR(1)分析表

输入：文法 G

输出：SLR(1)分析表

步骤：

(1) 构造文法 G 的 LR(0)项目集规范族及识别活前缀的 DFA。

(2) 判断 DFA 中的每个状态是否有冲突。

(3) 对每个冲突状态，计算归约项目左部符号的 Follow 集。

(4) 检查每个状态和每条边。

- 若项目 $A\rightarrow\alpha\cdot X\beta\in I_k$ 且 $GO(I_k,X)=I_j$：如果 $X\in V_T$，则置 action[k,X]=S_j，即将(j,a)进栈；如果 $X\in V_N$，则置 goto[k,X]=j。
- 若项目 $A\rightarrow\alpha\cdot\in I_k$，则对任何 $a\in V_T$(或结束符 #)，若 $a\in Follow(A)$时，置 action[k,a]=r_j(设 $A\rightarrow\alpha$ 是文法 G'的第 j 个产生式)，即用 $A\rightarrow\alpha$ 归约。
- 若项目 $S'\rightarrow S\cdot\in I_k$，则置 action[k,#]=acc，即接受。
- 分析表中凡不能用上面步骤填入的空白均置为“出错标志”。

按照该方法构造含有 action 和 goto 两部分的分析表，如果表的每个入口不含多重定义，则称它为文法 G 的一张 SLR(1)分析表。使用 SLR(1)分析表的分析器称为 SLR(1)分析器。

例 3.39 对例 3.38 中的文法 G3.7 构造 SLR(1)分析表，如表 3.19 所示。

表 3.19 对例 3.38 中的文法 G3.7 的 SLR(1)分析表

状态	action						goto		
	i	+	*	(	)	#	E	T	F
0	S_5			S_4			1	2	3
1		S_6				acc			
2		r_2	S_7		r_2	r_2			
3		r_4	r_4		r_4	r_4			
4	S_5			S_4			8	2	3
5		r_6	r_6		r_6	r_6			
6	S_5			S_4		r_1		9	3
7	S_5			S_4					10
8		S_6			S_{11}				
9		r_1	S_7		r_1	r_1			
10		r_3	r_3		r_3	r_3			
11		r_5	r_5		r_5	r_5			

尽管采用 SLR(1)方法能够对某些 LR(0)项目集规范族中存在冲突的项目集，通过向前查看一个符号的办法得到解决，但是实际上大多数实用的程序设计语言的文法也不能满足 SLR(1)文法的条件。若按上述方法构造的分析表存在多重定义的入口(即含有动作冲突)，则说明文法不是 SLR(1)的。这种情况下，不能用上述算法构造分析表。

例 3.40 给定文法 G3.8,是已拓广的文法:

(0) S′→S

(1) S→aAd

(2) S→bAc

(3) S→aec

(4) S→bed

(5) A→e (G3.8)

用 S′→·S 作为初态集的项目,构造识别文法 G3.8 的活前缀的 DFA,如图 3.27 所示。可以发现在项目集 I_5 和 I_7 中存在移进-归约冲突。

I_5: S→ae·c A→e·

I_7: S→be·d A→e·

归约项目左部非终结符的 Follow(A)={c,d}。

在 I_5 中,Follow(A)∩{c}={c,d}∩{c}≠Φ。

在 I_7 中,Follow(A)∩{d}={c,d}∩{d}≠Φ。

因此,I_5,I_7 中的冲突不能用 SLR(1)方法解决,而文法 G3.8 是非二义的,这就意味着该文法不是 SLR(1)的,不能求 SLR(1)分析表,即该文法不能使用 SLR(1)分析方法。下一小节将介绍 LR(1)方法来解决这种冲突。

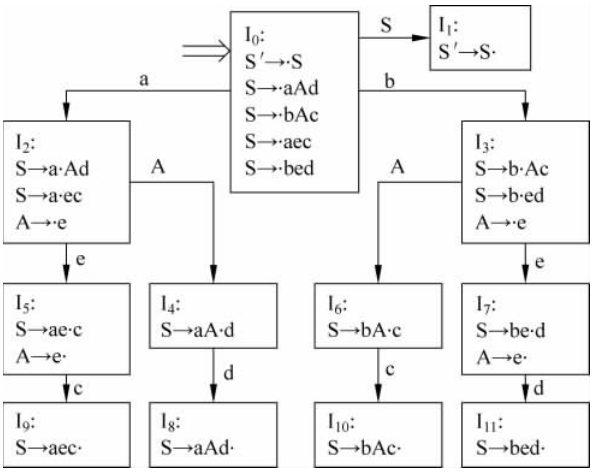


图 3.27 识别文法 G3.8 的活前缀的 DFA

*4. LR(1)分析

1) LR(1)分析的基本概念

由于用 SLR(1)方法解决动作冲突时,对于归约项目 A→α·,只要当前面临的输入符号 a∈Follow(A)时,就确定采用产生式 A→α 进行归约,但是如果栈中的符号串为 βα,归约后变为 βA,再移进当前符号 a,则栈里变为 βAa,而实际上 βAa 未必为文法规范句型的活前缀。因此,在这种情况下,用 A→α 进行归约是无效的。

例如,在识别表达式文法 G3.7 的活前缀 DFA 中,如图 3.26 所示,项目集 I_2 存在移进-归约冲突,即{E→T·,T→T·*F},若栈顶状态为 2,栈中符号为 #T,当前输入符为),

)∈Follow(E),这时按 SLR(1)方法应该用产生式 E→T 进行归约,归约后栈顶符号为#E,而再加当前符号)后,栈中为#E)不是文法 G3.7 规范句型的活前缀。

因此可以看出 SLR(1)方法虽然相对 LR(0)有所改进,但仍然存在着无效归约,也说明 SLR(1)方法向前查看一个符号的方法仍不够确切,LR(1)方法恰好可以解决 SLR(1)方法在某些情况下存在的无效归约问题。

可以设想让每个状态含有更多的"展望"信息,这些信息将有助于克服动作冲突和排除用 A→α 所进行的无效归约,在必要时对状态进行分裂,使得 LR 分析器的每个状态能够确切地指出 α 后跟哪些终结符时才允许把 α 归约为 A。

这就需要重新定义项目,使得每个项目都附带有 k 个终结符。现在每个项目的一般形式为

$$[A \to \alpha \cdot \beta, a_1 a_2 \cdots a_k]$$

其中,A→α·β 是一个 LR(0)项目,$a_i \in V_T^*$。这样的一个项目称为一个 LR(k)项目,项目中的 $a_1 a_2 \cdots a_k$ 称为它的向前搜索字符串(或展望串)。向前搜索字符串仅对归约项目[A→α·,$a_1 a_2 \cdots a_k$]有意义,对于任何移进或待约项目不起作用。归约项目[A→α·,$a_1 a_2 \cdots a_k$]意味着当它所属的状态呈现在栈顶且后续的 k 个输入符号为 $a_1 a_2 \cdots a_k$ 时,才可以把栈顶的句柄 α 归约为 A。这里,只讨论 k≤1 的情形,因为对多数程序语言的语法来说,向前搜索(展望)一个符号就可以确定"移进"还是"归约"。这样,归约项目都形如[A→α·,a],搜索字符 a∈Follow(A)。

一个 LR(1)项目[A→α·β,a]对活前缀 γ 是有效的,其含义是如果存在规范推导

$$S \overset{*}{\Rightarrow} \delta A \omega \Rightarrow \delta \alpha \beta \omega$$

其中,γ=δα,a 是 ω 的第一个符号,或者当 ω 为 ε 时,a 为#时。

例 3.41 考虑文法

(1) S→BB

(2) B→bB|a (G3.9)

项目[B→b·B,b]对活前缀 γ=bbb 是有效的,因为根据上述定义有 $S \overset{*}{\Rightarrow} bbBba \Rightarrow bbbBba$,其中 δ=bb,A=B,α=b,β=B,ω=ba。

2) LR(1)项目集规范族的构造

构造有效的 LR(1)项目集规范族本质上和构造 LR(0)项目集规范族的方法相同,也需要两个函数:Closure(I)和 GO(I, X),它们和 LR(0)文法中的这两个函数有区别。

(1) 项目集 I 的闭包 Closure(I),可按如下方式构造。

- 将 I 中的所有项目都加入 Closure(I)。
- 若项目[A→α·Bβ, a]∈Closure(I),B→γ 是一个产生式,那么对于任何 b∈First(βa),如果[B→·γ, b]原来不在 Closure(I)中,则把它加进去。重复执行该过程,直到 Closure(I)不再增大为止。

(2) 令 I 是一个项目集,X 是一个文法符号,则转换函数 GO(I, X)定义为

$$GO(I, X) = Closure(move(I, X))$$

其中,move(I, X) ={任何形如[A→αX·β, a]的项目|[A→α·Xβ, a]∈I}。

利用 GO()函数和 Closure()函数,可以求文法 G 的 LR(1)项目集规范族和识别活前缀的 DFA,构造方法可描述为算法 3.13。

算法 3.13 构造 LR(1)项目集规范族及识别活前缀的 DFA

输入：文法 G

输出：文法 G 的 LR(1)项目集规范族和识别活前缀的 DFA

步骤：

(1) 拓广文法，写出所有的 LR(1)项目。

(2) C={Closure ({[S′→ ·S, #]})}。

(3) 对 C 中的每个项目集 I 和 G′的每个文法符号 X，求 GO(I,X)。

- 如果 GO(I,X)≠Φ 且 GO(I,X)∉C，把 GO(I,X)加入 C 中。
- 在 I 和 Go(I,X)之间添加标记为 X 的弧线。

(4) 重复执行(3)的动作，直到 C 不再增大。

例 3.42 在例 3.41 中，识别活前缀的 DFA 中的冲突不能用 SLR(1)方法解决，利用算法 3.13 来构造 LR(1)项目集规范族及识别活前缀的 DFA。

考虑例 3.41 的 I_5、I_7 中的冲突不能用 SLR(1)方法解决，利用算法 3.13 来构造 LR(1)项目集规范族，如图 3.28 所示。这样 LR(1)项目集规范族有效地解决了 I_5、I_7 中的移进-归约冲突。由于归约项目的搜索字符集合与移进项目的移进符号不相交，所以在 I_5 中，当面临输入符为 d 时归约，为 c 时移进，而在 I_7 中，当面临输入符为 c 时归约，为 d 时移进。冲突可以全部解决，因此该文法是 LR(1)文法。

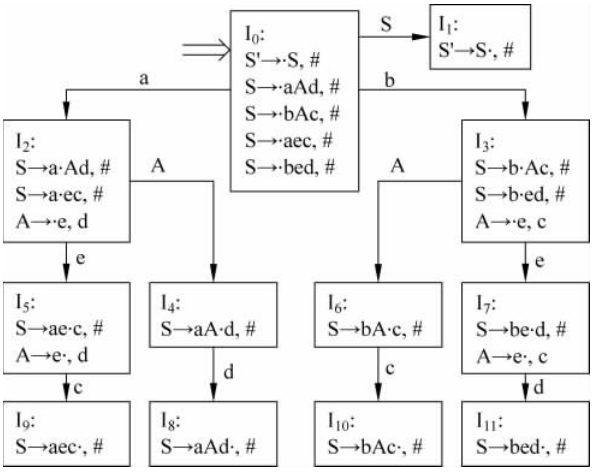


图 3.28 识别文法 G3.8 的 LR(1)项目活前缀的 DFA

3) LR(1)分析表的构造

根据文法的 LR(1)项目集规范族 C 及识别活前缀的 DFA，可以构造 LR(1)分析表。假定 C={I_0,I_1,…,I_n}，I_k 的下标 k 为分析表的状态，含有[S′→ ·S,#]的状态为分析器的初态。LR(1)分析表可按算法 3.14 构造。

算法 3.14 构造 LR(1)分析表

输入：文法 G 的 LR(1)项目集规范族和识别活前缀的 DFA

输出：LR(1)分析表

步骤：

(1) 若项目[A→α·aβ,b]∈I_k,且GO(I_k,a)=I_j,其中a∈V_T,则置action[k,a]=S_j,即把输入符号a和状态j分别移入文法符号栈和状态栈。

(2) 若项目[A→α·,a]∈I_k,其中a∈V_T,则置action[k,a]=r_j,即用产生式A→α进行归约,j是在文法中对产生式A→α的编号。

(3) 若项目[S′→S·,#]∈I_k,则置action[k,#]="acc",表示接受。

(4) 若GO(I_k,A)=I_j,其中A∈V_N,则置goto[k,A]=j,表示当栈顶符号为A时,从状态k转换到状态j。

(5) 凡不能用步骤(1)～(4)填入分析表中的元素,均置“报错标志”。

按上述算法构造的分析表,若不存在多重定义入口(即动作冲突)的情形,则称它是文法G的规范的LR(1)分析表。使用这种分析表的分析器叫作规范的LR分析器或LR(1)分析器,具有规范的LR(1)分析表的文法称为一个LR(1)文法。如果用上述方法构造的分析表出现冲突时,该文法就不是LR(1)的。

例3.43 求例3.42的LR(1)分析表。

利用算法3.14,得到例3.42的LR(1)分析表如表3.20所示。

表3.20 例3.42的LR(1)分析表

状态	action						goto	
	a	b	c	d	e	#	S	A
0	S_2	S_3					1	
1						acc		
2					S_5			4
3					S_7			6
4				S_8				
5			S_9	r_5				
6			S_{10}					
7			r_5	S_{11}				
8						r_1		
9						r_3		
10						r_2		
11						r_4		

由表3.20可以看出对LR(1)的归约项目不存在任何无效归约。但在多数情况下,同一个文法的LR(1)项目集的个数比LR(0)项目集的个数多,甚至可能多好几倍。这是因为同一个LR(0)项目集的搜索字符集合可能不同,多个搜索字符集合则对应着多个LR(1)项目集。如例3.44中的文法G3.9′是一个LR(0)文法,其LR(0)项目集规范族中只有7个状态,而LR(1)项目集规范族中有10个状态。

就文法的描述能力来说,有下面的结论:

$$LR(0) \subset SLR(1) \subset LR(1) \subset 无二义文法$$

例 3.44 将文法 G3.9 的拓广文法 G3.9′。

(0) S′→S

(1) S→BB

(2) B→bB

(3) B→a (G3.9′)

该文法的 LR(1)项目集规范族的计算方法是：用[S′→ ·S,#]作为初态集的项目，然后利用闭包和 GO 函数进行计算。项目集规范族 C 和识别活前缀的 DFA 如图 3.29 所示。

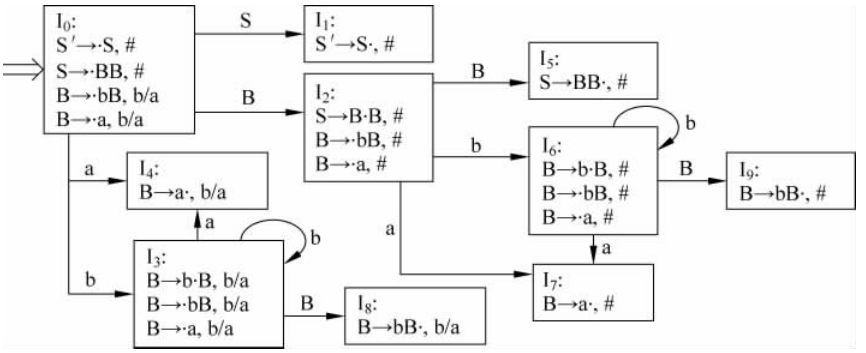


图 3.29 识别文法 G3.9′的 DFA

再根据图 3.29 以及算法 3.14，可以得到 LR(1)分析表如表 3.21 所示。

表 3.21 文法 G3.9′LR(1)分析表

状 态	action			goto	
	b	a	#	S	B
0	S_3	S_4		1	2
1			acc		
2	S_6	S_7			5
3	S_3	S_4			8
4	r_3	r_3			
5			r_1		
6	S_6	S_7			9
7			r_3		
8	r_2	r_2			
9			r_2		

*5. LALR(1)分析

LALR 方法是一种折中方法，它的分析表比 LR(1)分析表要小得多，能力也弱一些，但它能应用在一些 SLR(1)不能应用的场合。实际的编译器经常使用这种方法，大多数程序设计语言的语法结构能方便地由 LALR 文法表示。

就分析器的大小而言，SLR 和 LALR 的分析表对同一个文法有同样多的状态，而规范 LR(1)分析表要大得多。例如，对 Pascal 这样的语言，SLR 和 LALR 的分析表有几百个状态，而规范 LR(1)分析表有几千个状态。所以，使用 SLR 和 LALR 比使用 LR 要经济得多。

在LR(1)分析表中,若存在两个状态(项目集)除向前搜索字符不同外,其他部分都是相同的,称这样的两个LR(1)项目集是同心的,相同部分称为它们的心,如图3.29中的I_3和I_6是同心的。如果把同心的LR(1)项目集合并,心仍相同(心就是一个LR(0)项目集),超前搜索字符集为各同心集超前搜索字符的并集,合并同心集后go()函数自动合并。如将I_3和I_6合并后得到$I_{3,6}$:{B→b·B,a/b/#　B→·bB,a/b/#　B→·a,a/b/#}。这种LR分析法称为LALR方法。对同一个文法,LALR分析表和LR(0)、SLR分析表具有相同数目的状态。

若合并LR(1)项目集规范族中的同心集后没有产生新的冲突,称为LALR(1)项目集。合并同心集可能会推迟发现错误的时间,但错误出现的位置仍是准确的。

下面给出构造LALR分析表的算法,其基本思想是首先构造LR(1)项目集规范族及识别活前缀的DFA,如果它不存在冲突,就把同心集合并。若合并后的项目集规范族不存在归约-归约冲突(即不存在同一个项目集中有两个像A→c·和B→c·这样的产生式具有相同的搜索字符),就能按这个项目集规范族构造分析表。LALR分析表可按算法3.15构造。

算法3.15　构造LALR分析表

输入:文法G

输出:LALR分析表

步骤:

(1) 构造文法G的LR(1)项目集规范族及识别活前缀的DFA,设$C=\{I_0,I_1,\cdots,I_n\}$。

(2) 合并所有的同心集,得到LALR(1)的项目集规范族$C'=\{J_0,J_1,\cdots,J_m\}$。含有项目[S′→·S,#]的$J_k$为DFA的初态。

(3) 由C′构造action(动作)表。其方法与LR(1)分析表的构造相同。

① 若[A→α·aβ,b]∈J_k,且GO(J_k,a)=J_j,其中a∈V_T,则置action[k,a]=S_j,即把输入符号a和状态j分别移入文法符号栈和状态栈。

② 若项目[A→α·,a]∈J_k,其中a∈V_T,则置action[k,a]=r_j,r_j的含义是按产生式A→α进行归约,A→α是文法的第j个产生式。

③ 若项目[S′→S·,#]∈I_k,则置action[k,#]=acc,表示分析成功,接受。

(4) goto表的构造。对于不是同心集的项目集,转换函数的构造与LR(1)的相同;对于同心集项目,由于合并同心集后新集的转换函数也为同心集,所以,转换函数的构造也相同。

假定$I_{i1},I_{i2},\cdots,I_{in}$是同心集,合并后的新集为$J_k$,转换函数GO($I_{i1}$,X),GO($I_{i2}$,X),…,GO($I_{in}$,X)也为同心集,将其合并后记作$J_i$,因此,有GO($J_k$,X)=$J_i$,所以当X为非终结符时,GO($J_k$,X)=$J_i$,则置goto[k,X]=i,表示在k状态下遇到非终结符X时,转向状态i。

(5) 分析表中凡不能用步骤(3)、(4)填入信息的空白均填上“出错标志”。

经上述算法构造的分析表若不存在冲突,则称它为文法G的LALR分析表,存在这种分析表的文法称为LALR文法。使用LALR分析表的分析器称为LALR分析器。

LALR与LR(1)的不同之处是当输入串有误时,LR(1)能够及时发现错误,而LALR则可能还继续执行一些多余的归约动作,但绝不会执行新的移进,即LALR能够像LR(1)一样准确地指出出错的位置。

例3.45 对例3.44的文法G3.9′,求该文法的LALR(1)分析表。

根据图3.29的LR(1)项目集规范族,可发现同心集如下。

I_3: B→b·B,a/b　和　I_6: B→b·B,#
B→·bB,a/b　　B→·bB,#
B→·a,a/b　　B→·a,#
I_4: B→a·,a/b　和　I_7: B→a·,#
I_8: B→bB·,a/b　和　I_9: B→bB·,#

即I_3和I_6,I_4和I_7,I_8和I_9分别为同心集,将同心集合并后为

$I_{3,6}$: B→b·B,a/b/#　　B→·bB,a/b/#　　B→·a,a/b/#
$I_{4,7}$: B→a·,a/b/#
$I_{8,9}$: B→bB·,a/b/#

同心集合并后仍不包含冲突,因此该文法是LALR文法。

构造该文法的LALR(1)分析表的步骤是:I_3和I_6合并后用$I_{3,6}$表示,I_4和I_7合并后用$I_{4,7}$表示,I_8和I_9合并后用$I_{8,9}$表示,对文法合并同心集后的LALR(1)分析表如表3.22所示,这就和该文法的LR(0)分析表相同。

表3.22 文法G3.9′的LALR(1)分析表

状　态	action			goto	
	b	a	#	S	B
0	$S_{3,6}$	$S_{4,7}$		1	2
1			acc		
2	$S_{3,6}$	$S_{4,7}$			5
3,6	$S_{3,6}$	$S_{4,7}$			8,9
4,7	r_3	r_3	r_3		
5			r_1		
8,9	r_2	r_2	r_2		

*6. 二义文法在LR分析中的应用

对一个文法,如果它的任何移进-归约分析器都存在这样的情况:尽管栈的内容和下一个输入符号都已了解,但仍无法确定分析动作是"移进"还是"归约",或者无法从几种可能的归约中确定其一,则该文法是非LR的。LR文法肯定是无二义的,一个二义文法绝不会是LR文法;任何一个二义文法绝不是LR文法,也不是一个算符优先文法或LL(k)文法,因此任何一个二义文法不存在相应的确定的语法分析器。但是对某些二义文法,可以进行适当修改,给出优先性和结合性,从而构造出比相应非二义文法更优越的LR分析器。

例3.46 构造算术表达式的二义文法G3.1的LR(0)项目集。

$$E \to E+E \mid E*E \mid (E) \mid i \qquad (G3.1)$$

将文法G3.1拓广,写成如下形式:

(0) E′→E

(1) E→E+E

(2) E→E*E

(3) E→(E)

(4) E→i (G3.10)

定义各状态表如表3.23所示，LR(0)项目集规范族及识别活前缀的DFA如表3.24所示。

表3.23 G3.10的状态定义

I_0： E′→·E E→·E+E E→·E*E E→·(E) E→·i	I_4： E→E+·E E→·E+E E→·E*E E→·(E) E→·i	I_2： E→(·E) E→·E+E E→·E*E E→·(E) E→·i
I_3： E→i·	I_1： E′→E· E→E·+E E→E·*E	I_5： E→E*·E E→·E+E E→·(E) E→·i
I_9： E→(E)·		
I_6： E→(E·) E→E·+E E→E·*E	I_7： E→E+E· E→E·+E E→E·*E	I_8： E→E*E· E→E·+E E→E·*E

表3.24 G3.10的DFA(用矩阵表示)

	+	*	(	)	i	#	E
I_0			I_2		I_3		I_1
I_1	I_4	I_5				acc	
I_2			I_2		I_3		I_6
I_3							
I_4			I_2		I_3		I_7
I_5			I_2		I_3		I_8
I_6	I_4	I_5		I_9			
I_7	I_4	I_5					
I_8	I_4	I_5					
I_9							

从表3.23中可以看出，状态I_1、I_7和I_8中存在移进-归约冲突，现在逐个分析冲突的解决方法。

在I_1中，归约项目E′→E·实际上为接受项目。由于Follow(E′)={#}，也就是只有遇到句子的结束符号#才能接受，因而与移进项目的移进符号+，*不会冲突，所以可用SLR(1)方法解决，即当前输入符为#时则接受，遇+或*时则移进。

在 I_7 和 I_8 中,由于归约项目[E→E+E·]和[E→E*E·]的左部都为非终结符 E,而 Follow(E)={#,+,*},而移进项目均有+和*,也就存在

$$\text{Follow}(E) \cap \{+, *\} \neq \Phi$$

因而 I_7 和 I_8 中的冲突不能用 SLR(1)的方法解决,也可以证明该二义文法用 LR(k)方法仍不能解决此冲突。

然而可以定义优先关系和结合性来解决这类冲突,假如规定*优先级高于+,且它们都服从左结合,那么在 I_7 中,由于*优先级高于+,所以遇到*移进,又因+服从左结合,所以遇到+则用 E→E+E 去归约,在 I_8 中,由*优先级高于+,服从左结合,不论遇到+、*或#号都应归约。

该二义文法的 LR 分析表如表 3.25 所示。

表 3.25 对表达式二义文法的 LR 分析表

状态	action						goto
	+	*	(	)	i	#	E
0			S_2		S_3		1
1	S_4	S_5				acc	
2			S_2		S_3		6
3	r_4	r_4		r_4		r_4	
4			S_2		S_3		7
5			S_2		S_3		8
6	S_4	S_5		S_9			
7	r_1	S_5		r_1		r_1	
8	r_2	r_2		r_2		r_2	
9	r_3	r_3		r_3		r_3	

现用表 3.25 对输入表达式串 i+i*i# 进行分析,分析过程如表 3.26 所示。

表 3.26 用二义文法分析表对输入串 i+i*i# 的分析过程

步骤	状态栈	符号栈	输入串	action	goto
1	0	#	i+i*i#	S_3	
2	03	#i	+i*i#	r_4	1
3	01	#E	+i*i#	S_4	
4	014	#E+	i*i#	S_3	
5	0143	#E+i	*i#	r_4	7
6	0147	#E+E	*i#	S_5	
7	01475	#E+E*	i#	S_3	
8	014753	#E+E*i	I#	r_4	8
9	014758	#E+E*E	#	r_2	7
10	0147	#E+E	#	r_1	1
11	01	#E	#	acc	

不难发现对二义文法规定了优先关系和结合性后的 LR 分析速度比相应的非二义文法的 LR 分析速度要快一些，对输入串 i＋i＊i＃的分析，用表 3.26 比用表 3.17 少了 3 步，对于其他的二义性文法，也可用类似的方法处理，构造出无冲突的 LR 分析表。

3.5 语法分析器的自动生成工具 YACC

LR 分析法的一个主要缺点是分析表很大，因此不宜手工构造分析器，必须求助于自动产生 LR 分析程序的生成器来辅助构造语法分析器。这类工具很多，本节将介绍使用 LALR 原理的语法分析自动生成器 YACC，它实现了 3.4 节讨论的许多概念，而且应用非常广泛。

3.5.1 YACC 概述

YACC(Yet Another Compiler-Compiler)是一个著名的编译程序自动生成工具，它是 20 世纪 70 年代初期由 Johnson 等人在美国 Bell 实验室研制开发的一个基于 LALR(1)的语法分析程序构造工具。它早期作为 UNIX 系统中的一个实用程序，现在已经得到广泛应用，被用来帮助实现了几百个编译器。YACC 还不是一个完整的编译程序自动生成器，它只能生成语法分析程序，还不能产生完整的编译程序。YACC 的输入是要编写语法分析器的语言的语法描述规格说明，它基于 LALR 语法分析的原理，自动构造一个该语言的语法分析器，同时还能根据规格说明中给出的语义子程序建立规定的翻译。

一个语法分析器可用 YACC 按图 3.30 所示的方式构造出来。首先，用 YACC 规定的格式将 L 语言的规格说明(文法产生式)建立于一个源文件中，YACC 源程序以.y 为扩展名(例如 trans.y)，运行 YACC(运行方式：在命令行输入 c＞yacc trans.y，本书使用的是 bison，因此使用的命令是：c＞bison trans.y)，把文件 trans.y 翻译为一个 C 程序(上例中的 C 程序名为 trans_tab.c)，它使用的是 LALR(1)方法。程序 trans_tab.c 包含用 C 语言写的 LALR 分析器和其他用户准备的 C 语言例程。为了使 LALR 分析表少占空间，使用了紧凑技术压缩分析表的大小。

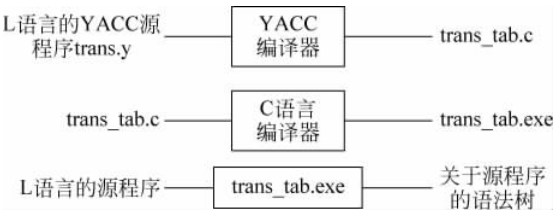


图 3.30 用 YACC 建立翻译器的过程

其次用 C 语言的编译程序将 trans_tab.c 进行编译(Windows 环境下可用 tcc 或 lcc 进行编译：c＞tcc trans_tab.c，UNIX 下使用 cc trans.tab.c-ly)，编译的结果是目标程序 trans_tab.exe(UNIX 下得到的输出是 trans.tab.out)，该目标程序是可执行程序。最后运行该程序，就能完成符合 L 语言规范的源程序的语法分析。如果还需要其他过程的话，它们可以和 trans_tab.c 一起编译或装载，就和使用 C 程序一样。

3.5.2 YACC 源文件的格式

YACC 源程序由三部分组成,格式如下:

```
说明部分
%%
翻译规则
%%
用C语言编写的辅助例程
```

其中:

(1) 说明部分包括两个可选择的部分。第一部分用%{和%}括起来,说明语义动作中使用的数据类型、全局变量、语义值的联合类型等,这部分内容包括直接放入输出文件的任何 C 代码(用%{和%}括起来,主要包括其他源代码文件的 # include 指示);另一部分用%开头,说明建立分析程序的有关记号、数据类型以及文法规则的信息,包括终结符及运算符的优先级等,这里说明的记号可以在 YACC 源程序的第二部分和第三部分中使用。

(2) 翻译规则部分位于第一个%%后面,每条规则包括修改的 BNF 格式的文法产生式以及在识别出相关的文法规则时被执行的 C 代码[即根据 LALR(1)分析算法,在归约时使用的动作]。文法规则中使用的元符号惯例是:通常,竖线用来作为候选项的分隔符(也可分别写出多个候选项)。用来分隔文法规则的左右两边的箭头符号→在 YACC 中被一个冒号取代了,而且必须用分号来结束每个文法规则。如产生式集合为

$$\text{左部} \rightarrow \text{选择 1} \mid \text{选择 2} \mid \cdots \mid \text{选择 n}$$

在 YACC 中写成

```
左部    : 选择 1        {语义动作 1}
        | 选择 2        {语义动作 2}
          ⋮                ⋮
        | 选择 n        {语义动作 n}
        ;
```

在 YACC 产生式中,加单引号的字符'c'是由单个字符 c 组成的记号;没有引号的字母数字串,若也没有声明为记号,则是非终结符。右部的各个选择之间用竖线隔开,最后一个右部的后面用分号,表示该产生式集合的结束。第一个左部非终结符是开始符号。

YACC 的语义动作是 C 语句序列。在语义动作中,符号 $$ 表示引用左部非终结符的属性值,而 $i 表示引用右部第 i 个文法符号的属性值。每当归约一个产生式时,执行与之关联的语义动作,所以语义动作一般是从各 $i 的值决定 $$ 的值。

(3) YACC 源程序的第三部分位于第二个%%后面,是一些用 C 语言编写的支持例程。在这部分中必须提供名字为 yylex( )的词法分析器[可以用 Lex 来产生 yylex( )],如果需要的话,本部分还可以加上其他过程,如错误恢复例程。

词法分析器 yylex( )返回单词符号和属性。返回的单词类别,如 DIGIT,必须在 YACC 程序的第一部分声明;属性值必须通过 YACC 定义的变量 yylval 传给分析器。

例 3.47 为说明怎样准备 YACC 的源程序,下面以构造台式计算器的翻译程序为例,

该台式计算器读入一个算术表达式,对其求值,然后打印其结果。

首先给出台式计算器识别的表达式的文法:

$$E \rightarrow E + T \mid E - T \mid T$$
$$T \rightarrow T * F \mid F$$
$$F \rightarrow (E) \mid DIGIT$$

记号 DIGIT 是 0～9 的单个的数字。由该文法编写出的 YACC 源程序如下:

```
%{
#include <ctype.h>
%}
%token DIGIT
%%
line    :expr '\n'  {printf("%d\n", $1);}
        ;
expr    :expr '+' term  { $$ = $1 + $3;}
        |expr '-' term  { $$ = $1 - $3;}
        |term
        ;
term    :term '*' factor { $$ = $1 * $3}
        |factor
        ;
factor:'('expr')'  { $$ = $2;}
        |DIGIT
        ;
%%

main() {
   return yyparse();
}

int yylex() {
    int c;
    while((c = getchar()) == ' ');          /* 跳过空格 */
    if (isdigit(c)) {
        yylval = c - '0';
        return DIGIT;
    }
    if (c == '\n') return 0;
    return c;
}

int yyerror(char *s) {
   fprintf(stderr,"%s\n",s);                /* printing the error message */
   return 1;
}
```

在这个 YACC 源程序中,说明部分的第一部分只有一个包含语句,它使得 C 的预处理程序包含标准头文件< ctype.h >,该文件中含有对函数 isdigit()的声明。说明部分的第二部分是对文法符号的说明,本例中只将 DIGIT 声明为记号。

对非终结符 E 有三个产生式：

$$E \rightarrow E + T \mid E - T \mid T$$

和它们相关的语义动作就写成

```
expr    : expr '+' term     { $$ = $1 + $3; }
        | expr '-' term     { $$ = $1 - $3; }
        | term
        ;
```

注意，在第一个产生式中，非终符 term 是右部的第三个文法符号，'+'是第二个文法符号。第一个产生式的语义动作是把右部 expr 的值和 term 的值相加，把结果赋给左部非终结符 expr，作为它的值。第三个产生式的语义动作描述省略，因为当右部只有一个文法符号时，语义动作默认就是值的复写，即语义动作是{ $$ = $1; }。

注意，在语法规则中加了一个新的开始产生式

```
line    : expr'\n'      {printf ("%d \n", $1); }
```

该产生式的含义是，这个台式计算器的输入是一个表达式后面跟一个换行字符。它的语义动作是打印表达式的十进制值并且换行。

3.5.3 YACC 的翻译规则

YACC 文法规则中的记号有两种。第一种，文法规则的单引号中的任何字符都表示它本身。因此，单字符记号就可直接被包含在这种风格的文法规则中，如例 3.47 中的运算符记号+、-和 *(以及括号记号等)；第二种，在定义部分用 YACC 的记号声明(以%token 开始)来声明符号记号，如例 3.47 中的记号 DIGIT，这样的记号被 YACC 赋予了不会与任何字符值相冲突的数字值。典型地，YACC 开始用数字 258 给记号赋值。YACC 将这些记号定义作为# define 语句插入到输入代码中。因此，在输出文件 trans_tab.c 中就可能会找到行#define DIGIT 258 作为 YACC 对源文件中的%token DIGIT 声明的对应。YACC 坚持定义所有的符号记号本身，而不是从别的地方引入一个定义，但是却有可能通过在记号声明中的记号名之后书写一个值来指定赋给记号的数字值。例如，写出%token DIGIT 18 就将给 DIGIT 赋值 18(而不是 258)。

在例 3.47 中的规则部分中，还可以看到非终结符 expr、term 和 factor 的规则。由于还需要打印出一个表达式的值，所以还有另外一个称为 line 的规则，而且将其与打印动作相结合。因为 line 的规则放在了所有规则的最前面，所以 line 被作为文法的开始符号。若不是这样，可在定义部分用%start line 来定义，这样就不必将 line 的规则放在开头了。

YACC 中的动作是由在每个文法规则中将其写作真正的 C 代码(在花括号中)来实现的。通常，尽管也有可能在一个文法规则的中间写出要执行的动作，但动作代码仍是放在每个文法规则候选式的末尾(但在竖线或分号之前)。在书写动作时，可以享受到 YACC 伪变量的好处。当识别一个文法规则时，规则中的每个符号都拥有一个值，除非它被参数改变了，该值将被认为是一个整型值(稍后将会看到这种情况)。这些值由 YACC 保存在一个与分析栈保持平行的语义栈中。每个在栈中的符号值都可通过使用以 $ 开始的伪变量来引用。$ $ 代表刚才被归约出来的非终结符的值，也就是在文法规则左边的符号。伪变量 $1、$2、$3 等都代表了文法规则右边的每个连续的符号。因此在例 3.47 中，文法规则和

动作 expr: expr'+'term{ $$ = $1+ $3; }就意味着当识别出一个符号串可用规则expr→expr+term 进行归约时,就将产生式右边 expr 的值与 term 的值相加作为左边的 expr 的值。

所有的非终结符都是通过用户提供的这些动作来得到它们的值。记号也可以被赋值,但这是在扫描器中实现的。YACC 假设记号的值已赋给了 YACC 内部定义的变量 yylval,且在识别记号时必须给 yylval 赋值。因此,在文法和动作 factor: DIGIT{ $$ = $1; }中,值 $1 指的是当识别记号时已在前面赋值给 yylval 的 DIGIT 的值。

3.5.4 YACC 的辅助程序

例 3.47 的第三个部分(辅助程序部分)包括了三个过程的定义。第一个是 main 的定义,之所以包含它是因为 YACC 输出的结果可以直接编译为可执行的程序。过程 main 调用 yyparse,yyparse 是 YACC 所产生的分析过程的名称。这个过程返回一个整型值。当分析成功时,该值为 0;当分析失败时,该值为 1(即发生一个错误,且还没有执行错误恢复)。

YACC 生成的 yyparse 过程接着又调用一个扫描程序过程,该过程为了与 Lex 词法分析程序生成器兼容,所以就假设名为 yylex(参见 3.4 节)。因此,YACC 说明还包括了 yylex 的定义。在这个特定的情况下,yylex 过程非常简单。YACC 的词法分析器用 C 语言的函数 getchar()每次读入一个输入字符,如果是数字字符,则将它的值存入变量 yylval 中,返回 DIGIT;否则,将字符本身作为记号返回。它所需要做的是返回下一个非空字符,但若这个字符是一个数字,此时就必须识别单个元字符记号 DIGIT 并返回它在变量 yylval 中的值。这里有一个例外:由于假设一行中输入一个表达式,所以当扫描程序到达输入的末尾时,输入的末尾是一个换行字符(在 C 中的'\n')。YACC 希望输入的末尾通过 yylex 由空值 0 标出(这也是 Lex 的一个惯例),所以读到'\n'时返回 0。最后定义了一个 yyerror 过程。当在分析中遇到错误时,YACC 使用这个过程打印出一个错误信息(典型地,YACC 打印串"语法错误",但这个行为可由用户改变)。

3.6 语法分析中的错误处理

如果编译程序只需要处理正确的程序,那么它的设计和实现将会大大简化。然而,不管程序员如何努力,程序总有可能出现错误。语法错误是高级语言程序设计中最容易出现的错误,因此总是期望语法分析程序能够尽可能地帮助程序员找到错误,指出错误的位置,以便程序员对源程序进行调试。然而由于有多种不同的语法分析方法,其处理和发现错误的方式可能不一样,本节主要介绍各种语法分析方法中的错误处理方式。

3.6.1 语法分析中的错误处理的一般原则

语法分析器至少应能判断出一个程序在语句构成上是否正确,即如果源程序包括语法错误,则必须指出某个错误的存在;反之,若程序中没有语法错误,分析程序不应声称有错误存在。除了这个最低要求之外,分析程序还应该对不同层次的错误做出不同的反应。通常的错误处理程序试图给出一个有意义的错误信息,尽可能地判断出错误发生的位置。有些分析程序还可以进行错误校正(Error Correction),即试图从给出的不正确的程序中推断

出正确的程序,如跳过某些单词、添加标点符号等。若语法分析器发现了错误但不做错误校正,很难生成有意义的错误信息。

语法分析中的错误处理应遵循以下原则。

(1) 发现错误为主,校正错误为辅。校正的目的是为了使语法分析能进行下去,一般我们希望语法分析器能从常见的语法错误中恢复并继续处理程序的其余部分,而不是发现错误就立即退出。

(2) 错误局部化,选择一个适当的位置恢复分析过程。分析程序应尽可能多地分析代码,更多地找到真实的错误,而不是出现错误后马上停止分析;即使跳过部分代码,也应使语法分析程序跳过的语法成分最少。

(3) 准确报告,应尽早给出错误发生的位置,否则错误位置可能会丢失;减少重复信息与株连信息,应避免出现错误级联问题(Error Cascade),这有可能会产生一个冗长的虚假的出错信息;还应避免错误的无限循环,此时即使没有任何输入也会产生一个错误信息的无限级联。

上述原则不可能同时满足,所以在实际编写语法分析器的错误处理时应做一些折中。如为了避免错误级联和无穷循环问题,分析程序应跳过一些输入符号,这与"尽可能多地分析代码"的原则相矛盾。

我们并不希望语法分析器一检测到错误就直接退出,而是希望它能一次性分析更多的代码,发现更多的错误。这就需要对源程序采用某种错误恢复方法,以便能继续分析下去。

语法分析中的错误恢复策略通常有以下几种。

(1) 应急模式(Panic Mode)的恢复,即删除符号的方法。这种方法很容易实现,完全不需要改变分析栈,并且能够保证不会进入无限循环。当读入不适当的符号之后,就删除这个符号及后继的一些符号,直到找到某个语句的分界符或特定的单词为止,如分号或者},这些单词通常称为同步词法符号(Synchronizing Token)。它们在源程序中的角色是清晰、无二义的。这种方法常常会跳过相当多的输入,不检查被跳过部分的其他错误。

(2) 短语层次的错误恢复,即通过插入、忽略、删除某个符号,使余下的输入串的某个前缀替换为另一个串,使语法分析器可以继续分析。有代表性的局部纠正方法包括将一个逗号替换为分号、删除一个多余的分号或者插入一个被遗漏的分号。在选择替换方法时必须小心以避免进入无限循环。如如果总是在当前输入符号之前插入符号,就出现了无限循环。这种方法在很多带有错误修复的编译器中广泛使用,它可以纠正任何输入串。其主要不足在于它难以处理实际错误发生在被检测到之前的情况。

(3) 添加错误产生式(Error Production)。这种方法是通过预测可能会碰到的常见错误,在当前语言的文法中加入特殊的产生式。这些产生式能够产生含有错误的语法单位。基于增加了错误产生式的文法可以构造一个语法分析器。如果分析过程中使用了某个错误产生式,语法分析器就检测到了一个预期的错误,据此生成适当的错误诊断信息。

3.6.2 自上而下语法分析的错误处理

1. 自上而下语法分析错误处理的一般方法

自上而下语法分析中经常使用应急模式来进行错误恢复。这种方式处理的基本机制是为每个递归函数定义一个同步词法符号集,在分析处理时,将同步符号集作为参数传递给分析函数,如果遇到错误,分析程序就继续向前读入输入符号,丢弃遇到的符号,直到看到某个

输入符号与同步符号集合中的某个符号相同为止，并从这里恢复分析。在做这种快速扫描时，通过不生成新的出错信息(在某种程度上)来避免错误级联。这种处理方法的效果依赖于同步符号的选取，一般从以下几方面来考虑同步符号的选取。

(1) 把 Follow(A)中的所有符号放入非终结符 A 的同步符号集。如果在遇到错误时跳过输入符号直到出现 Follow(A)中的符号，就把 A 从栈中弹出，继续分析。

(2) 加入相应语句开头的关键字。同步符号只使用 Follow 集是不够的，例如，在 C 语言中，分号作为语句的结束符号，分号就在某语句的非终结符的 Follow 集中，而作为下一个语句开头的关键字就不在其中。这样，如果在某个语句后少了一个分号，下一个语句开头的关键字就会被跳过，就会导致跳过很多个符号。

(3) 把 First 集中的符号也加入到同步集合中，这样递归下降分析程序在看到输入符号串中出现了 First 集中的符号时就可以恢复错误。

(4) 如果某个非终结符有 ε 产生式，就可以将 ε 作为默认情况，这样可以推迟错误的检测，但不能导致错误丢失。

2. 递归下降分析程序中的错误处理方式

在递归下降分析中，出现下面两种情况则说明出现了语法错误。

(1) 在推导过程中当前输入符号和文法推导的符号不相匹配。

(2) 在递归过程中调用形成死循环。

通过对 3.3.3 节中的递归下降分析过程设置同步符号集，以讨论递归下降分析器中的错误校正。除保持 match()函数之外，增加两个函数 CheckInput()(完成对 First 集的先行检查)和 ScanTo(跳过不必要的符号)：

```
CheckInput(firstset, followset)
{   if not(token in firstset) error;            /* 报告不在 First 集中 */
     ScanTo(firstset∪followset);
}
ScanTo(synchset)
{    while not(token in synchset∪{#})
        lookahead = GetNextToken( );
        /* 跳过符号,直到出现了同步符号集中的符号 */
}
```

这样 3.3.3 节中的 F 函数就可以加上错误处理(synchset 参数表示该函数的同步符号)，如下：

```
//对非终结符 F,候选式为 F→i|(E)
void F(synchset )
{
  CheckInput({'(', 'i'}, synchset);        //检查即将读入的输入符号是否是 First 集中的元素
  if (lookahead == '(') {
        match('(');
        E({')'});
        if (lookahead == ')') match(')');
        else error ( );
      }
    else if (lookahead == 'i') match('i');
```

```
    else error;
    CheckInput(synchset,{ '*', '#', ')', '+'});
                                        //检查即将读入的符号是否是 Follow 集中的元素
}
```

CheckInput()在这个过程中被调用了两次：一次是核实 First 集合的符号是输入串中的下一个输入符号；另一次是核实 Follow 集合(或 synchset)的符号是该函数退出后的下一个输入符号,这种处理将产生合理的错误信息。例如,输入串(2+-3)*4-+5 将产生两个出错信息(一个在第 1 个减号上,另一个在第 2 个加号上)。

一般情况下,在递归调用中向下传送同步符号集 synchset,同时在新的函数中再添加相应的新同步符号。在 F()过程中,当看到一个左括号之后,只有右括号在 E 的 Follow 集合时,E()才与右括号一起被调用。

3. LL(1)预测分析程序中的错误处理

预测分析中若出现下述两种情况,说明出现了语法错误。

(1) 栈顶的终结符与当前输入符号不匹配。

(2) 非终结符 A 处于栈顶,面临的输入符号是 A,但分析表中的 M[A,a]为空。

第一种情况是不常见的,这是因为就一般而言,当在输入中真正地看到输入符号时,它们只会被压入栈中。

应急模式的错误恢复也可在 LL(1)分析程序中实现,其实现方式与在递归下降分析中相似。由于 LL(1)程序是非递归的,就要求用一个新栈来保存同步符号集 synchset 参数,若不用一个额外的栈,也可静态地将同步符号集与 CheckInput()函数所采取的动作一起放入 LL(1)分析表中。在算法生成每个动作之前(当一个非终结符位于栈顶时),对 CheckInput()函数进行调用。

假设有一个位于栈顶的非终结符 A,面临一个不在 First(A)中的输入符号,就有三种处理方法。

(1) 将 A 从栈顶弹出。

(2) 看到一个可重新开始分析的输入符号之后,从输入符号串中读出该符号。

(3) 在栈中压入一个新的非终结符。

若当前输入符号是#或是在 Follow(A)中时,就选择方法 1；若当前输入符号不是#或不在 First(A)∪Follow(A)中,就选择方法 2；在特殊情况中方法 3 有时会有用,但却很少是恰当的。

例 3.48 对表 3.3 的 LL(1)分析表添加同步符号后的预测分析表如表 3.27 所示。其中,synch 表示由相应非终结符的 Follow 集构成的同步符号集。

表 3.27 文法 G3.3 的加入同步符号后的预测分析表

	i	+	*	(	)	#
E	E→TE′			E→TE′	synch	synch
E′		E′→+TE′			E′→ε	E′→ε
T	T→FT′	synch		T→FT′	synch	synch
T′		T′→ε	T′→*FT′		T′→ε	T′→ε
F	F→i	synch		F→(E)	synch	synch

分析时，若发现 M[A,a]为空，则跳过输入符号 a，若该表项为 synch，则弹出栈顶的非终结符号 A；若栈顶的终结符与输入符号不匹配，则弹出栈顶的输入符号。使用表 3.27 对输入串 *i*+i# 的分析过程前几步如表 3.28 所示。

表 3.28 加入同步符号后的部分分析过程

步 骤	分 析 栈	输 入 串	说 明
1	#E	*i*+i#	表中[E,*]为空，跳过 *
2	#E	i*+i#	E→TE′
3	#E′T	i*+i#	T→FT′
4	#E′T′F	i*+i#	F→i
5	#E′T′i	i*+i#	i 匹配，弹出 i
6	#E′T′	*+i#	T′→*FT′
7	#E′T′F*	*+i#	* 匹配，弹出 *
8	#E′T′F	+i#	M[F,+]=synch，弹出 F
9	#E′T′	+i#	T′→ε
⋮	⋮		⋮

3.6.3 自下而上语法分析的错误处理

1. 算符优先分析中的错误检测

使用算符优先分析时，在以下两种情况下会发现语法错误。

(1) 若在栈顶终结符与下一个输入符号之间不存在任何优先关系。

(2) 若找到某一“素短语”，但不存在任一产生式，其右部为此素短语。

针对上述情况，处理错误的子程序可分为几类。

(1) 在算符优先分析中，虽然非终结符的处理是隐含的，也应该在栈中为非终结符留有相应的位置。因此，当说“素短语”与某一个产生式的右部匹配时，则意味着相应的终结符相同，非终结符的位置也是相同的。即使非终结符的位置相同，出现在栈中的非终结符也不一定是一个正确的非终结符。

(2) 当发生第一种情况时，即栈顶符号与输入符号之间不存在任何优先关系，可以采取更一般的错误处理方法，即改变、插入或删除符号。如果采取改变和插入符号的方法，注意不要造成无限循环。如一直在输入端插入符号，但始终不能将栈内符号序列进行归约或将输入符号移进。一种不会陷入死循环的方法是确保在恢复后能够把当前输入符号移进栈(如果输入符号是#，确保不会移进该符号，且栈的长度最终会被缩短)。

(3) 当发生第二种情况时，就应该打印错误信息，然后确定该“素短语”与哪个产生式的右部最相似。利用该产生式报告较准确的错误信息，添加适当的符号继续分析。

算符优先关系表中的空白项，实际是没有优先关系的错误，所以必须指定一个错误恢复子程序，同一程序可用在多个地方。这样在语法分析器发现两个符号之间没有优先关系，就调用相应的错误恢复子程序进行错误处理。

例 3.49 表 3.29 是一个带有错误处理的算符优先关系表，该表中的空白项(即两个符号之间没有关系的项)被填上了 e_1，e_2，e_3，e_4，它们是错误处理程序的名字。

表 3.29 带有错误处理的算符优先关系表

	i	(	)	#
i	e_3	e_3	$\gtrdot$	$\gtrdot$
(	$\lessdot$	$\lessdot$	$\doteq$	e_4
)	e_3	e_3	$\gtrdot$	$\gtrdot$
#	$\lessdot$	$\lessdot$	e_2	e_1

这些错误处理程序的功能如下。

e_1：/* 缺少整个表达式时调用 */

把 id 插入到输入字符串中；

输出的错误信息是“缺少操作对象”。

e_2：/* 表达式以右括号开始时调用 */

从输入字符串中删除)；

输出的错误信息是“右括号不匹配”。

e_3：/* i 或)后面跟随 i 或(时调用 */

把+插入到输入字符串中；

输出的错误信息是“缺少运算符”。

e_4：/* 表达式以左括号结束时调用 */

从栈中删除(；

输出的错误信息是“缺少右括号”。

例 3.50 用例 3.49 的带错误处理的分析表分析输入符号串)i)。

分析过程如表 3.30 所示。在第 1 步中，发现表达式以右括号开始，出错，删除)，调用 e_2，给出错误信息“右括号不匹配”。在第 5 步中，# 和)之间没有优先关系，发生错误，调用错误处理程序 e_2，删除右括号，提示错误信息“右括号不匹配”。

表 3.30 对输入串)i)的带错误处理的算符优先分析过程

步 骤	栈	输入缓冲区	说 明
1	#	)i)#	初始状态
2	#	i)#	表达式以右括号开始，出错，调用 e_2
3	#i	)#	#$\lessdot$i，i 入栈
4	#F	)#	#$\lessdot$i$\gtrdot$)，用 F→i 归约
5	#F	)#	# 和)之间没有优先关系，错误，调用 e_2
6	#F	#	

2. LR 分析中的错误检测

LR 分析法在自左至右扫描输入串的过程中就能发现其中的任何错误，并能准确地指出出错位置。LR 语法分析器在访问 action 表时，若遇到一个空(或错误)的表项，将检测到一个错误，但在访问 goto 表时决不会检测到错误。与算符优先分析器不同，LR 语法分析器只要发现已扫描的输入出现一个不正确的后继符号就会立即报告错误。规范 LR 语法分析器在报告错误之前不会进行任何无效归约。SLR 语法分析器和 LALR 语法分析器在报告

错误之前可能执行几步归约，但绝不会把出错点的输入符号移进栈。

在LR分析中遇到出错时，有可能输入符号不能移进栈，又不能对栈顶符号串进行归约。处理方法有两类：第一类是使用插入、删除或修改输入符号的方法；第二类包括检测到某一个不合适的短语时，它不能与任何产生式匹配。此时，错误处理程序可能跳过其中的一些输入符号，将含有语法错误的短语分离出来。分析程序认定含有错误的符号串是由某一非终结符A所推导出的，此时该符号串的一部分已经处理，处理结果反映在栈顶的一系列状态中，剩下的未处理的符号仍在输入缓冲区中。分析程序跳过一些输入符号，直至找到某一个符号a，它能合法地跟在A的后面。同时要把栈顶的内容逐个移去，直到找到一个状态s，该状态与A有一个对应的新状态goto[s,A]，并将该新状态压栈。此时分析程序就认为它已找到A的某个匹配并已将它局部化，然后恢复正常的分析过程。

LR语法分析器主要采用短语级层次的错误恢复方式，这种方法处理比较容易，不必担心不正确的归约，实现方式是通过检查LR分析表的每个出错表项，并根据语言的使用情况确定最可能引起的错误以及程序员最容易犯的错误，然后为其编写一个适当的错误处理程序。只要在分析表的空项中填上适当的错误处理程序的指针即可。

例3.51 对简单的算术表达式文法的LR分析方法添加出错处理程序。

$$E \rightarrow E+E \mid E * E \mid (E) \mid i$$

表3.31给出了带有错误处理的LR分析表。出错处理程序的动作如下。

e_1：/*处于状态0,2,4,5时，要求输入符号为运算对象的首终结符，即i和左括号，但遇到的是+，*或#，就调用该处理程序*/

把一个假想的i压进栈，状态3进栈(即执行的是在0,2,4,5状态下面临i时的动作)，同时给出错误信息“缺少运算对象”。

e_2：/*处于状态0,1,2,4,5时，若遇右括号，就调用该处理程序*/

从输入缓冲区中删除右括号，给出错误信息“右括号不匹配”。

e_3：/*处于状态1,6时，期望一个操作符，却遇到了i或右括号，就调用该处理程序*/

将符号+压栈，状态4进栈，给出错误信息“缺少操作符”。

e_4：/*处于状态6，期望操作符或右括号，却遇到了#，就调用该处理程序*/

把右括号压入栈，状态9进栈，给出错误信息“缺少右括号”。

e_5：/*处于状态3,7,8,9时，希望输入符号为+，*或#，才能进行归约，但遇到的是i和(，就调用该处理程序*/

把一个假想的操作符+压进栈，执行归约，同时给出错误信息“缺少运算符”。

例3.52 用表3.31的带错误处理的分析表分析输入字符串i+)。

表3.31 带有错误处理的LR分析表

状 态	action						goto
	i	+	*	(	)	#	E
0	S_3	e_1	e_1	S_2	e_2	e_1	1
1	e_3	S_4	S_5	e_3	e_2	acc	
2	S_3	e_1	e_1	S_2	e_2	e_1	6

续表

状　态	action						goto
	i	+	*	(	)	#	E
3	e_5	r_4	r_4	e_5	r_4	r_4	
4	S_3	e_1	e_1	S_2	e_2	e_1	7
5	S_3	e_1	e_1	S_2	e_2	e_1	8
6	e_3	S_4	S_5	e_3	S_9	e_4	
7	e_5	r_1	S_5	e_5	r_1	e_5	
8	e_5	r_2	r_2	e_5	r_2	e_5	
9	e_5	r_3	r_3	e_5	r_3	e_5	

分析过程前几步如表3.32所示。按照LR分析过程,在第5步时,发现栈顶状态为4,面临的输入符号为),查表发现出现了错误,调用出错处理程序e_2,删除),调用给出错误信息"右括号不匹配"。继续分析到第6步时,栈顶状态4面临输入符号为#,查表发现出现了错误,调用出错处理子程序e_1,将一个假想的输入符号i压入符号栈,状态3进栈,给出错误信息"缺少操作对象"。

表3.32　对输入串i+)的带错误处理的LR分析过程

步　骤	符　号　栈	状　态　栈	输　入　串	错误信息和动作
1	#	0	i+)#	初始状态
2	#i	03	+)#	i进栈,3进栈
3	#E	01	+)#	归约i,3出栈,1进栈
4	#E+	014	)#	+进栈,4进栈
5	#E+	014	#	状态4遇到右括号,调用e_2,删除右括号,给出"右括号不匹配"的信息
6	#E+i	0143	#	状态4遇到#,调用e_1,压入一个假想的i,状态3进栈,给出"缺少操作对象"的信息
⋮	⋮	⋮	⋮	⋮

3. YACC中的错误处理

在YACC中主要使用错误产生式(Error Production)的方法进行错误处理。错误产生式就是形如A→·error α的包括了伪记号error的产生式,错误产生式可有效地允许程序员用人工方式标记出其goto项将被用作错误校正的非终结符。

首先由用户决定哪些"主要的"非终结符可能与错误处理有关,典型的选择是用于产生表达式、语句、程序块和函数(过程)的那些非终结符。然后把错误产生式A→·error α加入到文法中,其中A是主要的非终结符,α是文法符号串(可能是空串),error是YACC的保留字。YACC将从这样的产生式产生语法分析器,并把错误产生式当作普通产生式来处理。

当分析程序在分析中检测到错误时(即遇到分析表中的一个空项),它会从分析栈中弹出状态直至发现栈顶状态的项目集含有形如A→·error α的项目为止,然后把虚构的符号error移进栈。

如果α为ε时,立即归约为A并执行产生式A→error的语义动作(它可能是用户定义

的错误处理子程序),然后语法分析程序丢弃若干个输入符号,直到发现一个能恢复正常处理的输入符号为止。如果α非空,YACC在输入串上向前寻找能够归约为α的子串。如果α包含的都是终结符,那么它在输入串上寻找这样的终结符串,并把它们移进栈,这时语法分析栈的栈顶为error α,再把它归约为A,并恢复正常的语法分析。

例如,若错误产生式为

$$\text{stmt} \rightarrow \bullet\ \text{error};$$

要求语法分析程序看见错误时跳过下一个分号,好像该语句已经被看完一样。这个出错产生式的语义程序不需要处理输入,只需产生诊断信息并设置禁止生成目标代码的标记。

例3.53 对例3.47的台式计算器加上错误产生式的YACC源程序:

```
line    : line expr '\n' {printf("%d\n", $1);}
        | line '\n'
        | error '\n' {yyerror("重新输入上一行"); yyerrok; }
        ;
```

也就是说,当输入行有语法错误时,语法分析器从栈中弹出符号,直至碰到一个含有引进符号error动作的状态为止。语法分析器遇到上述错误产生式时把error移进栈,并跳过输入符号,直到发现换行符为止,此时语法分析器把换行符移进栈,把error'\n'归约成line,并输出诊断信息"重新输入上一行"。专门的YACC例程yyerrok用于将语法分析器恢复到正常操作模式。

3.7 小 结

为了能够精确地描述高级语言的语法构成规则,需要对语法规则进行形式化的描述,称为文法。适合描述高级语言语法结构的文法是上下文无关文法。在上下文无关文法中,如果从文法的开始符号出发,每次使用一个产生式的右部来替换当前符号串的左部的过程称为推导。使用推导来产生句型,只有终结符的句型称为句子,全部句子的集合称为语言。如果两个文法产生的语言相同,则这两个文法等价。推导的逆过程称为归约。得到一个句型并不规定替换产生式的顺序,因此一个句型可以由多个不同的推导过程来得到。如果推导过程中每次只替换句型中最左边的非终结符,称为最左推导;如果推导过程中每次只替换句型中最右边的非终结符,称为最右推导。如果一个文法是二义的,那一定有一个句型存在两个不同的最左推导,或者产生两棵不同的语法树。

在描述文法的基础上,可以用文法来描述高级语言的语法规则,然后根据语法规则进行分析。根据语法树生成的方向不同,语法分析主要分为自上而下和自下而上两种方法,两种方法各有优缺点,也各有适用文法。

自上而下的语法分析方法是自上而下建立语法树,具体包括递归下降分析方法和预测分析方法。要进行确定的自上而下的语法分析,要求文法必须是LL(1)文法。递归下降分析方法适合手工构造语法分析器,但要求书写语法分析器的语言具有递归实现;预测分析方法需要首先建立LL(1)分析表,由于一个实用的高级语言的LL(1)分析表很大,不太适合手工构造。在自下而上的分析方法中,从下往上建立语法树。在规范归约中,每次归约的是当前句型中的句柄;而在算符优先分析中,每次归约的是当前句型的最左素短语;在LR分

析法中,将过去归约的历史和未来将要读取的符号结合起来,使用项目的概念,每次归约的是当前句型的活前缀。算符优先分析和LR分析法是两种非常实用的两种语法分析方法,算符优先分析方法较简单,宜于手工构造,特别适合于算术表达式的分析;LR分析法的适用范围更广,宜于自动生成,是目前实现大多数编译程序语法分析器采用的方法。LR分析法有很多种,其中,LR(0)是最简单的一种,对文法的要求严格,只有满足LR(0)文法要求的文法才能使用LR(0)分析法,大多数程序设计语言都不满足LR(0)文法的要求,它们在建立项目集规范族时有些状态中会发生冲突,用不同的方法来解决这些冲突就构成不同的LR分析法,主要有SLR分析法和LR(k)分析法。语法分析的自动生成器YACC就是基于LALR分析法的语法分析工具。各种语法分析法都是为了发现源程序中的语法错误,不同的语法分析法处理错误的方法不同。

3.8 习 题

1. 语法分析器的功能是什么?其输入输出各是什么?

2. 自上而下语法分析和自下而上语法分析的主要差别是什么?

3. 自上而下语法分析面临的两个主要问题是什么?如何解决?

4. 解释下列术语:

上下文无关文法、推导、最左推导、最右推导、句型、句子、语言、文法等价、语法树、二义文法、LL(1)文法、归约、规范归约、句柄、短语、最左素短语、活前缀、项目

5. 从供选择的答案中,选出应填入_____的正确答案。

已知文法G[S]的产生式如下:

$$S \rightarrow (L) \mid a$$

$$L \rightarrow L,S \mid S$$

属于L(G[S])的句子是__A__,(a,a)是L(G[S])的句子,这个句子的最左推导是__B__,最右推导是__C__,语法树是__D__。

供选择的答案如下。

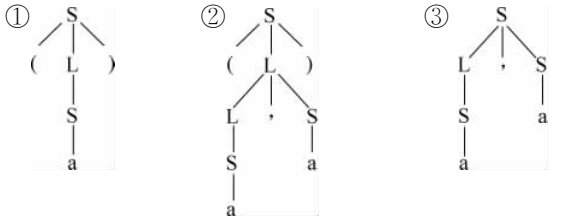
A: ① a ② a,a ③ (L) ④ (L,a)

B,C: ① $S \Rightarrow (L) \Rightarrow (L,S) \Rightarrow (L,a) \Rightarrow (S,a) \Rightarrow (a,a)$

② $S \Rightarrow (L) \Rightarrow (L,S) \Rightarrow (S,S) \Rightarrow (S,a) \Rightarrow (a,a)$

③ $S \Rightarrow (L) \Rightarrow (L,S) \Rightarrow (S,S) \Rightarrow (a,S) \Rightarrow (a,a)$

D:



6. 已知某算术表达式的文法G为:

(1) <AEXPR> → <AEXPR> + <TERM> | <TERM>

(2) <TERM>→<TERM>*<FACTOR>|<FACTOR>

(3) <FACTOR>→i|(<AEXPR>)

给出 i+i+i 和 i+i*i 的最左推导、最右推导和语法树。

7. 已知某文法 G[bexpr]：

bexpr → bexpr or bterm | bterm

bterm → bterm and bfactor | bfactor

bfactor → not bfactor | (bexpr) | true | false

(1) 请指出此文法的终结符号、非终结符号和开始符号。

(2) 试对句子 not(true or false)构造一棵语法树。

8. 试构造生成下列语言的上下文无关文法。

(1) $L_1=\{a^nb^nc^i|n\geqslant 1,i\geqslant 0\}$。

(2) $L_2=\{w|w\in\{a,b\}^+$，且 w 中 a 的个数恰好比 b 多 1$\}$。

(3) $L_3=\{w|w\in\{a,b\}^+$，且$|a|\leqslant|b|\leqslant 2|a|\}$。

(4) $L_4=\{w|$w 是不以 0 开始的奇数集$\}$。

(5) L_5 是不允许 0 开头的能被 5 整除的无符号数的集合。

(6) 语言 $L_6=\{x|x\in\{a,b,c\}^*$，x 是重复对称排列的(aabcbaa，aabbaa 等)$\}$。

(7) 用后缀方式表示的算术表达式。

(8) 由整数、标识符、四个二目运算(+、-、*、/)构成的表达式。

9. 已知某文法 G：

<AEXPR>→i|(<AEXPR>)|<AEXPR><AOP><AEXPR>

<AOP>→+|-|*|/

(1) 试用最左推导证明该文法是二义性的。

(2) 对于句子 i+i*i 构造两个相应的最右推导。

10. 对下面的陈述，正确的在陈述后的括号内画√，否则画×。

(1) 存在有左递归规则的文法是 LL(1)的。 (　　)

(2) 任何算符优先文法的句型中不会有两个相邻的非终结符号。 (　　)

(3) 算符优先文法中任何两个相邻的终结符号之间至少满足三种关系($a\doteq b$，$a\lessdot b$，$a\gtrdot b$)之一。 (　　)

(4) 任何 LL(1)文法都是无二义性的。 (　　)

(5) 每一个 SLR(1)文法也都是 LR(1)文法。 (　　)

(6) 存在一种算法，能判定任何上下文无关文法是否是 LL(1)的。 (　　)

(7) 任何一个 LL(1)文法都是一个 LR(1)文法，反之亦然。 (　　)

(8) LR(1)分析中括号中的 1 是指，在选用产生式 A→α 进行分析时，看当前读入符号是否在 First(α)中。 (　　)

11. 选择题，从供选择的答案中，选出应填入_____内的正确答案。

(1) 在编译程序中，语法分析分为自顶向下分析和自底向上分析两类。__A__和LL(1)分析法属于自顶向下分析；__B__和 LR 分析法属于自底向上分析。自顶向下分析试图为输入符号串构造一个__C__；自底向上分析试图为输入符号串构造一个__D__。采用自顶向下分析方法时，要求文法中不含有__E__。

供选择的答案如下。

A、B：①深度分析法　②宽度优先分析法
　　③算符优先分析法　④递归子程序分析法

C、D：①语法树　②有向无环图
　　③最左推导　④最右推导

E：①右递归　②左递归
　　③直接右递归　④直接左递归

(2) 自底向上语法分析采用__A__分析法，常用的是自底向上语法分析有算符优先分析法和LR分析法。LR分析法是寻找右句型的__B__；而算符优先分析法是寻找右句型的__C__。LR分析法中分析能力最强的是__D__；分析能力最弱的是__E__。

供选择的答案如下。

A：①递归　②回溯　③枚举　④移进-归约

B、C：①短语　②素短语　③最左素短语　④句柄

D、E：①SLR(1)　②LR(0)　③LR(1)　④LALR(1)

12. 试为下述文法构造一个递归下降的分析程序。

(1) 假定布尔表达式文法G[bexpr]，其产生式如下：

bexpr→bexpr or bterm|bterm

bterm→bterm and bfactor|bfactor

bfactor→not bfactor|(bexpr)|true|false

(2) 假定某语言文法G，其产生式如下：

stmt→if e then stmt stmtTail|while e do stmt|begin list end|s

stmtTail→else stmt|ε

list→listTail|stmt

listTail→; list|ε

13. 已知文法G[S]，其产生式如下：

S→(L)|a

L→ L,S|S

消除左递归，如果是LL(1)文法，构造分析表，说明对输入符号串(a,(a,a))的分析过程。

14. 求下述文法中各个非终结符的First集、Follow集，各候选式的First集。

(1) S→AB|bC

(2) A→b|ε

(3) B→aD|ε

(4) C→AD|b

(5) D→aS|c

15. 对下面的文法G：

E→TE′

E′→+E|ε

T→FT′

T′→T|ε

F→PF′

F′→*F′|ε

P→(E)|a|b|∧

(1) 计算这个文法的每个非终结符的 First 和 Follow。

(2) 证明这个文法是 LL(1)的。

(3) 构造它的预测分析表。

16. 下面文法中哪个是 LL(1)的,说明理由。

(1) S→Abc
A→a|ε
B→b|ε

(2) S→Ab
A→a|B|ε
B→b|ε

17. 已知文法 G[E]:

E→T|E+T

T→F|T*F

F→(E)|i

(1) 给出句型(T*F+i)的最右推导,并画出语法树。

(2) 给出句型(T*F+i)的短语、素短语和最左素短语。

(3) 证明 E+T*F 是文法的一个句型,指出这个句型的所有短语、直接短语和句柄。

18. 给定文法 G[S],其产生式如下:

S→(T)|a

T→T,S|S

(1) 给出输入串(a,(a,a))的最左和最右推导过程。

(2) 计算该文法各非终结符的 FirstVT 和 LastVT 集。

(3) 构造算符优先表。

(4) 计算上述文法的优先函数。

(5) 给出输入串(a,(a,a))的算符优先分析过程。

19. 给定文法:S→aS|bS|c

(1) 求出该文法对应的全部 LR(0)项目。

(2) 构造识别该文法所有活前缀的 DFA。

(3) 该文法是否是 LR(0)的? 若是,构造 LR(0)分析表。

(4) 给出输入串 ababc 的 LR 分析过程。

20. 下列文法是否为 SLR(1)文法? 若是,请构造相应的分析表。若不是,请说明理由。

(1) S→Sab|bR
R→S|a

(2) S→aSAB|BA
A→aA|B
B→b

21. 设文法 G 为

S→A

A→BA|ε

B→aB|b

(1) 证明它是 LR(1)文法。

(2) 构造它的 LR(1)分析表。

(3) 给出输入符号串 abab 的分析过程。

22. 证明下面文法是 LL(1)的但不是 SLR(1)文法。

S→AaAb|BbBa

A→ε

B→ε

23. 下面文法属于哪类 LR 文法? 试构造其分析表,并给出符号串(a,a)的分析过程。

S→(SR|a

R→,SR|)

24. 算法程序题。

(1) 构造 Sample 语言算术表达式的递归下降分析程序,或者某一控制语句的递归下降分析程序,要求如下。

① 书写出 Sample 语言语法的形式描述(BNF)。

② 消除左递归,提取左公因子。

③ 用某种高级语言书写出它的递归预测分析器。

(2) 构造 Sample 算术表达式的 LL(1)分析器。

① 书写出 Sample 语言语法的形式描述(BNF)。

② 消除左递归,提取左公因子。

③ 计算 First 集和 Follow 集。

④ 构造其 LL(1)分析表和 LL(1)驱动程序。

(3) 构造 Sample 算术表达式的算符优先分析器。

① 书写出 Sample 语言语法的形式描述(BNF)。

② 计算 FirstVT 集和 LastVT 集。

③ 构造其算符优先关系表和算符优先分析程序。

(4) 构造 Sample 算术表达式的 LR(0)分析器。

① 书写出 Sample 语言语法的形式描述(BNF)。

② 构造识别活前缀的有穷自动机。

③ 构造其 LR(0)分析表和 LR 分析程序。

(5) 使用软件工具 Yacc,构造 LALR 分析器,要求如下。

① 书写出 Sample 语言语法的形式描述(BNF)。

② 书写出 YACC 的源程序。

③ 用 YACC 生成 LALR 分析器。

④ 完成的分析程序的功能是:输入是算术表达式,输出为相应的后缀表达式;计算出算术表达式的值。