第 3 章

编译器前端

编译器前端是整个编译器的第一个阶段，它负责读入被编译程序的源代码，对其进行词法分析、语法分析和语义分析，并为程序构建合适的中间表示，供编译器的后续阶段进一步处理。词法分析将程序字符流解析为记号流，并检查词法单元的正确性；语法分析读入词法分析阶段输出的记号流，并根据程序语言的语法规则，对程序进行语法检查，对于语法检查通过的程序，编译器将为其构建抽象语法树作为中间表示；语义分析遍历程序的抽象语法树，并检查抽象语法树是否满足语言的语义规范。本章对编译器前端的词法分析、语法分析和语义分析等阶段进行深入讨论。

3.1 词法分析

词法分析器的主要任务是读入源程序的字符流，生成词法记号流，同时去除源程序的注释和空白(包括空格、制表或换行符等字符)。本节深入讨论词法分析器构建的理论和技术，先介绍正则表达式和有限状态自动机的概念，再给出由正则表达式构建有限状态自动机的算法。

3.1.1 记号

词法记号(lexical token)是一个二元组(k,s)，其中第一元 k 是词法记号所属的类别，第二元 s 是组成该词法记号的具体字符序列。词法记号是词法分析阶段使用的基本数据结构，接下来将其简称为记号。

程序设计语言的记号可以归类为有限的记号类型。例如，表 3-1 给出了典型程序设计语言的一些记号。其中 ID 代表标识符记号，典型的例子包括程序可以使用的标识符(但一般不包括关键字)；INT 代表整型常量记号；REAL 代表浮点型常量记号；关键字是程序设计语言预先确定了含义的词法单元，程序员不可以对这样的词法单元重新声明它的含义，如表 3-1 中的 IF、RETURN 等都是关键字。一般地，编译器将每个运算符和分隔符也都单独归为一类。

表 3-1 典型程序设计语言的一些记号

类型 k	字符序列 s	类型 k	字符序列 s
ID	x y area dfs	COMMA	,
INT	3 65 921 684	NEQ	!=
REAL	34.8 1e67 5.5e−10	LPAREN	(
IF	if	RPAREN	)
RETURN	return		

词法分析器的主要任务是读入程序的源代码的字符流，将其分割成记号流并返回记号流。例如，对于下面一段C语言示例程序：

```
1.  float match0(char *s){ /* find a zero */
2.    if(!strncmp(s, "0.0", 3))
3.    return 0.;
4.  }
```

词法分析器将返回下列记号流：

FLOAT ID(match0) LPAREN CHAR STAR ID(s) RPAREN LBRACE IF LPAREN BANG ID(strncmp) LPAREN ID(s) COMMA STRING(0.0) COMMA INT(3) RPAREN RPAREN RETURN REAL(0.) SEMI RBRACE EOF

需要注意，由于有些记号类别(如IF)只有单一的记号，因此并不需要记录对应的字符序列s；另外，记号EOF表示结束符，代表输入的结束。

接下来将介绍如何用正则表达式来给出记号的形成规则，并基于确定有限状态自动机来实现词法分析器。

3.1.2　正则表达式

语言是字符串组成的集合，字符串是字符的有限序列，字符来自字母表。例如，Pascal语言是所有组成合法Pascal程序的字符串的集合，C语言关键字是C程序设计语言中不能作为标识符使用的所有字母、数字和字符串组成的集合。这两种语言中，前者是无限集合，后者是有限集合。这两种语言的字母表都是ASCII字符集。

利用正则表达式(Regular Expression，RE)这一数学工具，可以使用有限的描述来指明这类有可能是无限的语言，每个正则表达式给出了一个字符串集合。

对于给定的字符集

$$\Sigma=\{c_1,c_2,\cdots,c_n\}$$

正则表达式由以下规则归纳定义。

(1) 空串ϵ是正则表达式，它表示$\{\epsilon\}$。

(2) 对于任意$c\in\Sigma$，c是正则表达式，它表示语言{c}。

(3) 如果M和N都是正则表达式，则以下也是正则表达式：

① 选择：$M|N=\{M,N\}$，一个字符串属于语言M或者语言N，则它属于语言M|N，即M|N组成的串包含M和N这两个集合的字符串的并集。

② 连接：$MN=\{mn|m\in M,n\in N\}$，如果一个字符串m属于语言M，一个字符串n属于语言N，则字符串的连接mn属于MN组成的语言。

③ 闭包：$M*=\{\epsilon,M,MM,MMM,\cdots\}$，即如果一个字符串是由M中的字符串经过零次至任意多次连接运算得到，则该字符串属于M*。例如，((a|b)a)*表示无穷集合{ϵ，aa，ba，aaaa，baaa，aaba，baba，…}。

约定括号具有最高优先级，以下依次为闭包、选择和连接，则可以省略正则表达式中一些不必要的括号。例如，((a)(b)*)|(c)等价于ab*|c。

还可以引入一些更简洁的缩写形式：[abcd] 表示(a|b|c|d)；[b-g]表示[bcdefg]；[b-gM- Qkr]表示[bcdefgMNOPQkr]；M? 表示(M|ε)；M+表示(MM*)。表 3-2 总结了正则表达式的基本操作符(包括缩写形式)。正则表达式的这些缩写形式，在正则表达式库或者词法分析器自动生成工具中都得到了广泛的应用。

表 3-2 正则表达式的基本操作符

缩写形式	含义
a	表示字符本身
ε	空字符串
M|N	选择，在 M 和 N 之间选择
MN	连接
M·N	连接的另一种写法
M*	克林闭包(Kleene closure)，重复 0 次或任意多次
M+	正闭包，重复 1 次或 1 次以上
M?	M 出现 0 次或 1 次
[a-zA-Z]	字符集
.	句点，表示除换行符之外的任意单个字符
“a.+*”	引号中的字符串，表示文字字符串本身

用正则表达式可以方便地表达程序语言中记号的形成规则。例如，C 语言的标识符是由字母或下画线开头，后跟零个或多个字母、数字或下画线的串，则可以用下面的正则表达式定义 C 语言标识符：

[a-zA- Z_]([a-zA-Z0-9_])*

3.1.3 有限状态自动机

定义 有限状态自动机(Finite-state Automaton，FA)是一个五元组：

$$(S,\Sigma,\delta,s_0,S_A)$$

其中各分量的含义如下：

(1) S 是一个有限状态集。

(2) Σ 是有限字母表，通常 Σ 是有限状态机中边的标签的集合，并且该集合和具体的语言相关，例如，在 C 语言中，Σ 是 ASCII 字符集。

(3) δ 是状态转移函数，它将每个状态 $s\in S$ 和每个字符 $c\in\Sigma$ 的二元组(s，c)映射到下一状态集合 T，这意味着有限状态自动机如果在状态 s 遇到字符 c，将转移到 δ(s，c)中的任一状态上。

(4) $s_0\in S$ 是指定的起始状态，有限状态自动机的转换将从这里开始。

(5) $S_A\subseteq S$ 是接收状态集合，当有限状态自动机转换到接收状态集合 S_A 中的某个状态时，自动机可终止执行。

根据状态转移函数 δ 的值域不同，有限状态自动机可分为确定有限状态自动机(Deterministic Finite-state Automata，DFA)和非确定有限状态自动机(Non-deterministic

Finite-state Automata，NFA）。

在确定的有限状态自动机中，有

$$\delta(s,c)=\{s'\}$$

即对于状态 s 和任意输入字符 c，确定有限状态自动机只会转移到一个确定的状态 s′，而不会转移到零个或超过一个状态。确定有限状态自动机以如下方式接受或拒绝一个字符串：确定有限状态自动机从初始状态出发，对于输入字符串的每个字符，自动机都将沿着一条确定的边到达另一状态，这条边必须标有输入字符。确定有限状态自动机对输入的 n 个字符进行了 n 次状态转换后，如果自动机到达了接收状态，则自动机将接受该字符串；如果到达的不是接收状态，或者在中间某个状态，自动机找不到与输入字符匹配的转移边，那么自动机将拒绝接收这个字符串。一个自动机识别的语言是该自动机接受的字符串的集合。图 3-1 是接收由正则表达式 (a|b) * abb 给出的字符串集合的 DFA。

非确定有限状态自动机的状态转移函数满足

$$\delta(s,c)=T,\quad \text{且 } c\neq\epsilon,\quad |T|\neq 1 \tag{3.1}$$

或

$$\delta(s,\epsilon)=T \tag{3.2}$$

方程(3.1)说明非确定有限状态自动机对于非空字符 c，可以转移到多于一个状态的集合 T，这意味着此时非确定有限状态自动机需要对多条标有相同符号的转移边进行选择。方程(3.2)说明非确定有限状态自动机还可能存在标有 ϵ 的边，这种边可以在不接受输入字符的情况下进行状态转换。图 3-2 是接受由正则表达式(a|b) * abb 给出的字符串集合的 NFA。

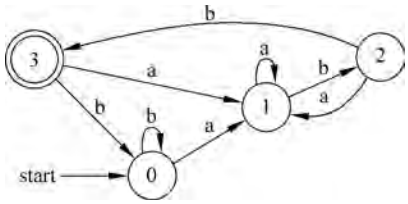


图 3-1　DFA

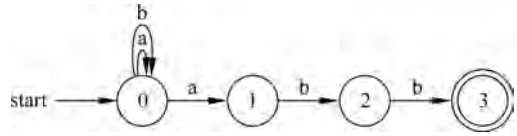


图 3-2　NFA

3.1.4　Thompson 算法

编译器要从正则表达式得到词法分析器，第一步可将正则表达式转换成一个非确定有限状态自动机。这一步骤将使用 Thompson 构造算法，该算法基于对正则表达式语法的模式匹配进行，即对于基本正则表达式(包括基本字符和空串 ϵ)，算法直接构造其对应的非确定有限状态自动机；而对于复合正则表达式(包括连接、选择和闭包)，算法以递归的方式构造其非确定有限状态自动机。

图 3-3 给出了用于 ϵ 和 a 的简单 NFA，以及用 a 和 b 的 NFA 组成 ab、a|b、a * 所需的转换。这种转换适用于任意的 NFA。

Thompson 构造算法首先为正则表达式中的每个字符构造简单的 NFA，再按照优先级和括号规定的顺序，对这些简单 NFA 应用选择、连接和闭包等转换。对于正则表达式 a(b|c) * ，

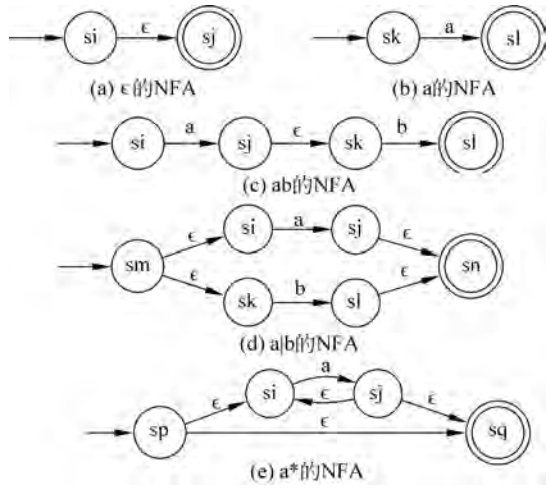


图 3-3 用于正则表达式运算符的简单 NFA

构造法首先分别为 a、b 和 c 构建简单的 NFA。因为括号的优先级最高，所以接下来为括号中的表达式 b|c 构建 NFA。又因为闭包的优先级比连接高，所以接下来为闭包(b|c)* 构建 NFA。最后将 a 和(b|c)* 的 NFA 连接起来。构造过程如图 3-4 所示。

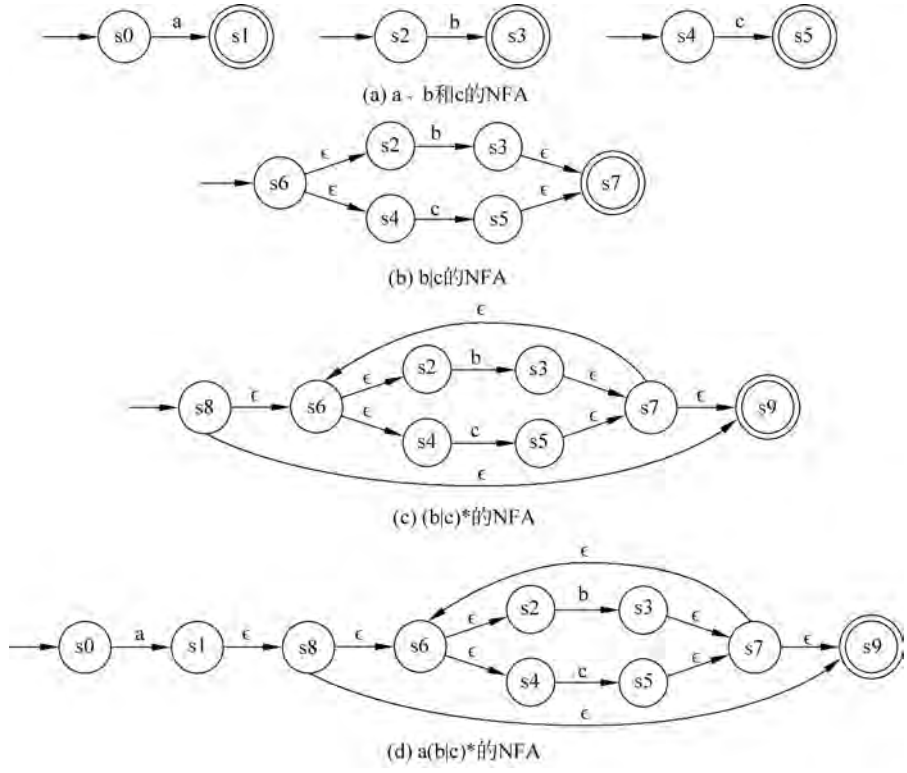


图 3-4 对 a(b|c)* 应用 Thompson 构造算法

3.1.5　子集构造算法

在非确定有限状态自动机中，从一个状态出发可能有多条标有相同符号的边，因此自动机需要进行选择，这种状态转移的不确定性导致算法在实现时往往需要用到回溯，回溯会影响程序执行效率。与非确定有限状态自动机的执行相比，确定有限状态自动机的每一步都只有一个可能的转换，因此其执行要容易得多。所以，词法分析器的下一步是使用子集构造算法将非确定有限状态自动机转换为等价的确定有限状态自动机。

算法 1 给出了子集构造算法，该算法接受非确定有限状态自动机$(N,\Sigma,\delta_N,n_0,N_A)$为输入，生成一个等价的确定有限状态自动机$(D,\Sigma,\delta_D,d_0,D_A)$，这两个自动机使用同样的字母表 Σ。公式在算法中出现时用正体表示，本书余下算法也使用此约定。

算法 1　子集构造算法

输入：非确定有限状态自动机 $(N,\Sigma,\delta_N,n_0,N_A)$
输出：确定有限状态自动机 $(D,\Sigma,\delta_D,d_0,D_A)$

```
1. procedure Subset (N, Σ, δN, n0, NA)
2.     q0 = Closure({n0} , δN)
3.     Q = {q0}
4.     W = {q0}
5.     while W ≠ Ø do
6.         从工作表 W 中删除一个状态集合 q
7.         for 对字母表 Σ 中的每个字符 c do
8.             t = Edge(q, c, δN)
9.             δD(q, c) = t
10.            if t ∉ Q then
11.                将 t 添加到 Q 和 W 中
12. procedure Edge(T, c, δ)
13.    R = Ø
14.    for 对于 S 中的每一个状态 s do
15.        R ⋃ = δ(s, c)
16.    return Closure(R, δ)
17. procedure Closure(T, δ)
18.    while 状态集合 T 仍在变化 do
19.        for T 中的每个状态 s do
20.            T ⋃ = δ(s, ε)
21.    return T
```

该算法构造了一个集合 Q，其每个元素都是状态集合 N 的一个子集，并且对应确定有限状态自动机中的一个状态。这个构造算法通过模拟非确定有限状态自动机的状态转移，来构建确定有限状态自动机。

该算法首先调用闭包函数 Closure$(\{n_0\},\delta_N)$，得到一个初始状态集合 q_0。闭包函数 Closure(T,δ) 是一个辅助函数，它计算给定的状态集合 T 仅通过 ε 转移就能到达的所有状

态集合。接着将 q_0 加入确定有限状态机的状态集合 Q。最后使用工作表算法，每次循环都从工作表 W 中删除一个状态集合 q，并对字母表 Σ 中的每个字符 c，分别计算从集合 q 中可以转移到的集合 t，并把状态转移关系记录在转移函数 δ_D 中。

子集构造算法是一个不动点算法，即集合 Q 是单调递增的，而 Q 中的每个集合只在工作表 W 上出现一次，所以算法一定会运行终止。

由于集合 Q 中的元素最多有 2^N 个，所以理论上该算法的最坏时间复杂度为 $O(2^N)$。但并不是每个子集都会出现 Q 中，所以这种情况实际很少发生。

当子集构造算法停止后，可以利用 Q 和 T 来构建 DFA。每个 $q_i \in Q$ 都用一个状态 $d_i \in D$ 来表示。如果 q_i 包含非确定有限状态自动机的某个接受状态，那么 d_i 就是确定有限状态自动机的接受状态之一。基于 q_0 构建的状态称为 d_0，即确定有限状态自动机的初始状态。

对于正则表达式 a(b|c)*，执行该算法的步骤如下：

(1) 初始化时，将 q_0 设置为ϵ-Closure(s_0)，刚好为 q_0。While 循环的第一次迭代计算ϵ-Closure($\delta(q_0,a)$)，其中包含了 6 个 NFA 状态，还计算了ϵ-Closure($\delta(q_0,b)$)和ϵ-Closure($\delta(q_0,c)$)，二者都为空集。

(2) while 循环的第二次迭代考查 q_1，生成了两个集合，称为集合 q_2 和 q_3。

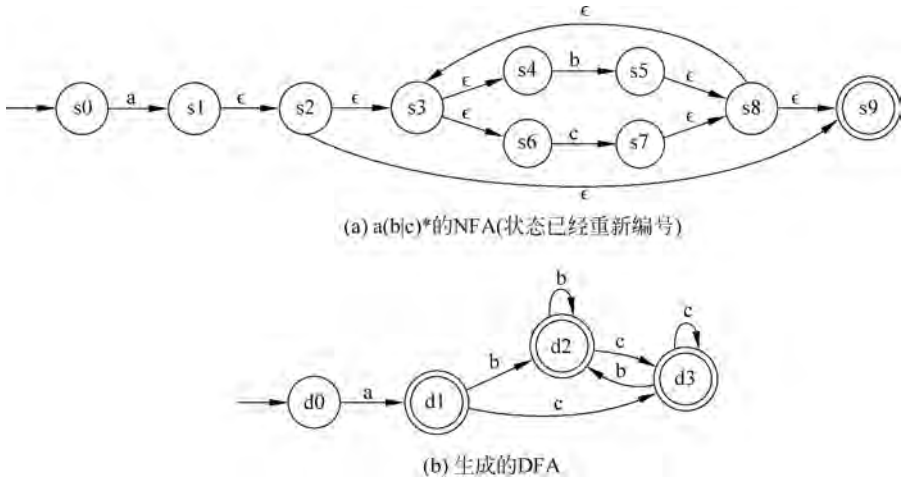
(3) while 循环的第三次迭代考查 q_2，构造出了两个集合，与集合 q_2 和 q_3 相同。

(4) while 循环的第四次迭代考查 q_3，又构造出了两个集合，与集合 q_2 和 q_3 相同。

表 3-3 详细列出了子集构造算法的各次迭代过程。图 3-5 给出了由此产生的 DFA，其状态对应于表 3-3 中得出的 DFA 状态，转移函数由生成这些状态的 δ 操作给出。由于集合 q_1、q_2 和 q_3 都包含 s_9(NFA 的接受状态)，因此在 DFA 中，这三个状态都是接受状态。

表 3-3 子集构造算法的各次迭代过程

集合名称	DFA 状态	NFA 状态	ϵ-Closure($\delta(q,*)$)		
			a	b	c
q_0	d_0	s_0	{s1, s2, s3, s4, s6, s9}		
q_1	d_1	{s1, s2, s3, s4, s6, s9}		{s5, s8, s9, s3, s4, s6}	{s7, s8, s9, s3, s4, s6}
q_2	d_2	{s5, s8, s9, s3, s4, s6}		q_2	q_3
q_3	d_3	{s7, s8, s9, s3, s4, s6}		q_2	q_3



(a) a(b|c)*的NFA(状态已经重新编号)

(b) 生成的DFA

图 3-5　对 NFA 应用子集构造法

3.1.6　Hopcroft 算法

用子集构造算法构建的确定有限状态自动机可能存在大量状态，这样会增加词法分析器占用的内存空间，所以需要对确定有限状态自动机进行化简，这个过程被称为自动机的最小化。为了最小化确定有限状态自动机 $(D,\Sigma,\delta_D,d_0,D_A)$，需要检测两个状态 $d_i,d_j\in D$ 是否等价，即二者是否对任何输入字符串都产生相同的行为。

算法 2 根据确定有限状态自动机状态的行为来找到状态中的各个等价类，并从这些等价类出发，构造一个最小的确定有限状态自动机。

算法 2　Hopcroft 算法

输入：确定有限状态自动机 $(D,\Sigma,\delta_D,d_0,D_A)$
输出：最小化后的确定有限状态自动机

```
1. procedure MinimizeDFA (D, Σ, δD, d0, DA)
2.     T = DA, D − DA
3.     P = Ø
4.     while P ≠ T do
5.         P = T
6.         T = Ø
7.         for 每个集合 p ∈ P do
8.             T = T ⋃ Split(p);
9.     return T
10. procedure Split(S)
11.     for 每个字符 c ∈ Σ do
12.         if c 将集合 S 划分为 S1 和 S2 then
13.             return {S1, S2}
14.     return S
```

这个算法的目标是构造出所有确定有限状态自动机状态的一个集合划分

$$P=p_1,p_2,\cdots,p_m$$

其构造依据是根据状态的行为分组：即对于两个状态 $d_x, d_y \in p_i$ 和任意字符 c，如果

$$d_x \xrightarrow{c} d_u$$

$$d_y \xrightarrow{c} d_v$$

则都有 $d_u, d_v \in p_j$，则状态 $d_x, d_y \in p_i$ 可看成是等价的。

在行为等价性的约束下，编译器为了最小化确定有限状态自动机，应该使每个集合 $p_i \in P$ 尽可能大。因此，算法使用的初始划分包括两个集合 $p_0 = D_A$ 和 $p_1 = D - D_A$，这样的划分确保在同一个集合中不会同时包含接受和非接受两种状态。

然后，算法重复地考查每个 $p_i \in P$，寻找 p_i 中对某个输入字符 c 具有不同行为的状态，以此来改进初始划分。划分的条件是：给定字符 $c \in \Sigma$，c 对于每个状态 $d_i \in p_s$ 都必须产生相同的行为，如果不满足这个条件，算法将围绕 c 来划分 p_i。

集合 $p_1 = \{d_i, d_j, d_k\}$ 中的各个状态是等价的，其充分必要条件为：$\forall c \in \Sigma$，各状态处理输入字符 c 时，所转移到的各个目标状态也属于同一个等价类。如图 3-6 所示，每个状态对 a 都有一个转移：$d_i \xrightarrow{a} d_x, d_j \xrightarrow{a} d_y, d_k \xrightarrow{a} d_z$。如果 d_x、d_y 和 d_z 在当前划分中属于同一个集合，那么 d_i、d_j 和 d_k 应该同时保留在 p_1 中，a 不会导致拆分 p_1。

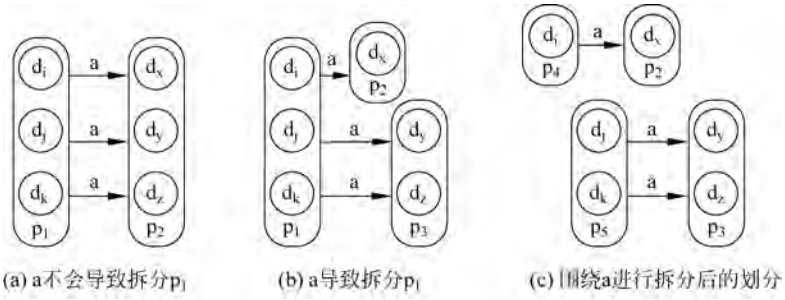


图 3-6　围绕 a 进行拆分

如果 d_x、d_y 和 d_z 属于两个或更多不同的集合，那么 a 将导致拆分 p_1。如果 $d_x \in p_2$ 且 d_y 和 $d_z \in p_3$，则必须拆分 p_1 并构造两个新集合 $p_4 = \{d_i\}$ 和 $p_5 = \{d_j, d_k\}$，以反映 DFA 对符号 a 开头的字符串的不同输出。如果 d_i 对 a 没有转移，也会导致同样的拆分。算法会考查每个状态 $p \in P$ 和每个字符 $c \in \Sigma$。如果字符 c 导致 p 发生拆分，算法会根据 p 构造两个新集合并添加到 P。重复这个过程，直至得到的划分中所有集合均无法拆分为止。

接下来依据最终的划分 P 来构造新的 DFA，首先分别创建一个状态来表示每个集合 $p \in P$，然后在这些新的状态之间增加适当的转移。对于表示 p_i 的状态，如果某个 $d_j \in p_i$ 对输入字符 c 转移到某个 $d_k \in p_m$，那么针对输入 c 添加一个转移，目标为表示 p_m 的状态。

对于正则表达式 a(b|c)*，最小化算法的第一步构造了一个初始划分 $\{\{d_0\}, \{d_1, d_2, d_3\}\}$。由于 p_1 只有一个状态，所以它不能拆分。p_2 中任何状态都没有针对输入 a 的转移。对于 b 和 c，p_2 中的每个状态都有一个转移，只是转移的目标又回到了 p_2 中。因此，Σ 中任何符号都不会导致 p_2 发生拆分，最终划分是 $\{\{d_0\}, \{d_1, d_2, d_3\}\}$。产生的最小 DFA 如图 3-7 所示。

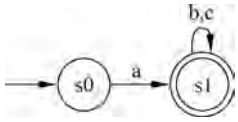


图 3-7　最小 DFA

3.2　语法分析

语法分析是编译器前端的第二个重要阶段。语法分析器的任务是读入词法分析阶段生成的记号流，判断该记号流是否满足程序设计语言的语法规则。如果语法分析器确认源程序不合法，则返回出错信息来指导程序员对源程序进行修改；如果语法分析器确认记号流符合相应语言的语法规则，它将为该程序构建一个合适的数据结构表示（一般是抽象语法树），供编译的后续阶段使用。

本节将讨论编译器实现语法分析的主要理论和技术。首先介绍描述语言语法规则的数学工具：上下文无关文法；然后分别介绍两类语法分析算法：自顶向下分析和自底向上分析。在自顶向下分析中，将重点介绍递归下降分析算法和 LL 分析算法，而在自底向上分析中，将重点介绍 LR 分析算法。

3.2.1　上下文无关文法

上下文无关文法（Context-Free Grammar，CFG）是一种形式化地描述程序设计语言语法规则的数学抽象，上下文无关文法 G 是一个四元组

$$G=(T,N,P,S)$$

其中：

（1）T 是终结符集合，终结符是组成句子的基本符号，在语法分析中，终结符一般可理解为词法记号。

（2）N 是非终结符集合，非终结符用于定义由文法生成的语言，给出了语言的层次结构。

（3）P 是一组产生式规则，产生式规则描述了将终结符和非终结符组合成串的方法。每条产生式规则都具有以下类似的语法形式

$$X \rightarrow \beta_1 \beta_2 \cdots \beta_n$$

其中 $X \in N$ 且 $\beta_i \in (T \cup N \cup \{\epsilon\})$，$1 \leqslant i \leqslant n$。

（4）$S \in N$ 是唯一的开始符号，按照惯例，P 中首先列出开始符号 S 的产生式。

下文对文法符号的表示一般使用以下约定：非终结符一般用大写字母表示；终结符一般用小写字母或数字表示。例如，下述符号是终结符。

（1）小写字母：如 a、b、c 等。

（2）运算符号：如＋、－、＊、/等。

（3）数字：如 0，1，…，9。

（4）分隔符：如标点符号或括号等。

一般用希腊字母 α、β、γ 等表示一个符号既有可能是非终结符，也有可能是终结符。接下来，在不引起混淆的情况下，把上下文无关文法简称为文法。

下面文法给出了表达式的语法

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

其中，非终结符 N={E,T,F }。E 是开始符号；终结符 T={+,-,*,/,(,),id}；产生式规则一共 3 条。

3.2.2 推导

推导(derivation)是一系列重写步骤，从语法的开始符号开始，结束于语言中的一个语句。对于给定文法 G=(T,N,P,S)，推导从 G 的开始符号 S 开始，用产生式 P 的右部替换左侧的非终结符，不断重复，直到串不出现非终结符为止，最终的串称为句子。

在推导过程中，根据选择被替换非终结符的方式的不同，可以将推导分为最左推导(leftmost derivation)和最右推导(rightmost derivation)。最左推导总是选择产生式右侧最左边的非终结符进行替换，最右推导总是选择最右边的非终结符替换。

例如，给定文法

$$E \rightarrow E+E \mid E*E \mid -E \mid (E) \mid id$$

对于句子-(id+id) 存在最左推导

$$\begin{aligned} E &\Rightarrow -E \\ &\Rightarrow -(E) \\ &\Rightarrow -(E+E) \\ &\Rightarrow -(id+E) \\ &\Rightarrow -(id+id) \end{aligned}$$

和最右推导

$$\begin{aligned} E &\Rightarrow -E \\ &\Rightarrow -(E) \\ &\Rightarrow -(E+E) \\ &\Rightarrow -(E+id) \\ &\Rightarrow -(id+id) \end{aligned}$$

3.2.3 分析树

分析树(parse tree)是对推导的树状表示，它和推导所用的顺序无关。分析树的每个内部节点代表非终结符，每个叶节点代表终结符，每一步推导代表如何从双亲节点生成它的直接孩子节点。语法分析根据给定的文法规则，对输入的字符串产生分析树，如果字符串中的各个部分能够从左至右地排在语法分析树的叶节点上，那么输入的字符串就是该文法定义的语言的一个句子，否则就不是。

给定文法 G，如果存在某个句子 s，它有两棵或多棵不同的分析树，则称文法 G 是二义性文法。这意味着二义性文法就是对同一个句子有多个最左推导或最右推导。例如，给定

文法

$$E \rightarrow E + E \mid E * E \mid -E \mid id$$

句子 id+id * id 具有两个不同的最左推导：

$$\begin{aligned} E &\Rightarrow E + E \\ &\Rightarrow id + E \\ &\Rightarrow id + E * E \\ &\Rightarrow id + id * E \\ &\Rightarrow id + id * id \end{aligned}$$

和

$$\begin{aligned} E &\Rightarrow E * E \\ &\Rightarrow E + E * E \\ &\Rightarrow id + E * E \\ &\Rightarrow id + id * E \\ &\Rightarrow id + id * id \end{aligned}$$

这两个推导对应的分析树如图 3-8 所示。

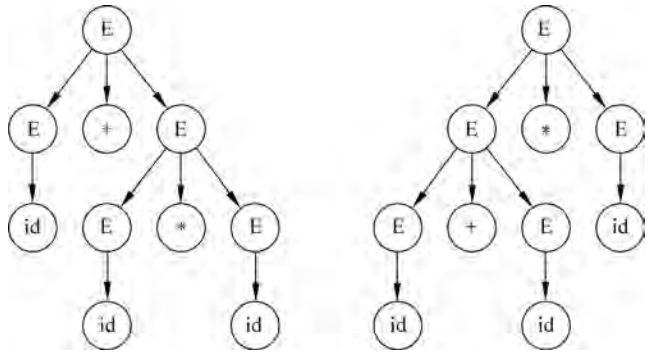


图 3-8　id+id * id 的两棵语法树

一个二义性的文法可以生成多个推导以及多个分析树。由于后续阶段会将语义关联到分析树的细节，所以存在多个分析树就意味着同一程序有多种可能的语义。从编译器的角度看，二义性文法将导致同一个程序有不同的含义，从而使程序运行的结果不唯一，故编译器设计者需要通过文法的重写解决二义性问题。

基于文法和推导的概念，语法分析器的任务就是对于给定的文法 G 和句子 s，判断是否存在对句子 s 的推导，如果存在，语法分析器将根据推导的过程构建分析树；如果不存在，语法分析器将返回语法错误信息。

3.2.4　自顶向下分析

自顶向下分析就是从文法 G 的开始符号 S 出发，不断地挑选合适的产生式对非终结符进行替换，最终展开到给定的句子。从构造分析树的角度看，语法分析器是从分析树的根

节点开始，系统化地向下扩展分析树，直到分析树的叶节点与词法分析器返回的记号流相匹配。在分析的过程中，编译器从分析树的下边缘选择一个非终结符，并选择一个以该非终结符为左部的产生式，用该产生式右侧相对应的子树来扩展该节点，如果推出的句子中的终结符与输入匹配，则可继续进行分析；如果不匹配，则需要回溯，尝试下一个产生式。这个过程一直持续到分析树的下边缘只包含终结符，且输入记号流耗尽为止。

基于最左推导的自顶向下语法分析由算法 3 给出。算法接受文法 G=(T,N,P,S)和记号流 tokens 作为输入参数，并返回语法分析的结果。算法使用变量 i 指向记号流 tokens 中下一个将要匹配的记号，栈 stack 中存放所有终结符和非终结符的序列。算法的主体是一个 while 循环，当栈顶元素是一个终结符时，将它与当前将要进行匹配的 tokens[i]比较，如果二者相等，则将该终结符弹出栈；如果二者不相等则需要进行回溯。如果栈顶元素是一个非终结符 T，则编译器将 T 弹出栈，并将该非终结符 T 的下一个没有匹配过的右部逆序压入栈。这个循环过程一直进行到栈 stack 空为止。

算法 3 基于最左推导的自顶向下分析

输入：文法 G=(T,N,P,S)和记号流 tokens
输出：语法分析的结果

```
1. procedure TopDownAnalysis(T, N, P, S, tokens)
2.     i = 0
3.     top = 0
4.     stack = [S]
5.     while stack ≠ [] do
6.       if stack[top] 是一个终结符 t then
7.           if (t == tokens[i++]) then
8.               pop()
9.           else
10.              backtrack()
11.      else   // stack[top] 是非终结符 T
12.          pop()
13.          push(β) // β 是 T 的下一个没有匹配过的右部
```

在上述算法的最后一步中，算法所选择的非终结符 T 的下一个没有匹配过的右部可能是错的，此时算法无法完成对输入的匹配，所以需要用到回溯。但在实际应用中，编译器编译大型程序时，如果反复进行回溯会大大影响编译器的效率，因此需要避免回溯来实现线性时间的分析算法。接下来将讨论对这个基本算法的改进：递归下降分析算法和 LL(1)分析算法。

1. 递归下降分析

递归下降分析(recursive decedent analysis)算法也称为预测分析(predicative analysis)算法，该算法为文法中的每个非终结符构造一个分析函数。非终结符 A 的分析函数可以识别输入流中 A 的一个实例。为了识别 A 的某个产生式右侧的非终结符 B，语法分析器递归调用 B 的分析函数。因此，递归下降语法分析器在结构上呈现为一组相互递归的过程。

为了举例说明这种算法，将为下面的文法构建一个递归下降分析器：

$$S \rightarrow \text{if E then S else S} \mid \text{begin S L} \mid \text{print E}$$

$$L \rightarrow \text{end} \mid ; \text{S L}$$

$$E \rightarrow \text{num} = \text{num}$$

递归下降语法分析器为非终结符 S、L 和 E 分别构造一个分析函数，非终结符的每个产生式对应分析函数中的一个子句。

```
enum token{
    IF,THEN,ELSE,BEGIN,END,PRINT,SEMI,NUM,EQ
};
extern enum token getToken(void);
enum token tok;

void advance(){
    tok = getToken();
}

void eat(enum token t){
    if(tok == t)
      advance();
    else error();
}

void S(){
    switch(tok){
      case IF: eat(IF); E(); eat(THEN); S();
               eat(ELSE); S(); break;
      case BEGIN: eat(BEGIN); S(); L(); break;
      case PRINT: eat(PRINT); E(); break;
      default: error();
    }
}

void L(){
    switch(tok){
      case END: eat(END); break;
      case SEMI: eat(SEMI); S(); L(); break;
      default: error();
    }
}

void E(){
    eat(NUM); eat(EQ); eat(NUM);
}
```

分析函数中调用了词法分析器的接口 getToken()来从输入流中读取下一个记号。递归下降分析器构建了分析函数 S()、L()和 E()分别分析非终结符 S、L 和 E。递归下降语法分析器的优点在于算法实现简单，有利于手工构造。

但递归下降分析算法对文法提出了苛刻的要求，即同一个非终结符的右侧产生式之间不能包括同样的非终结符。例如，如果要为如下文法构建一个递归下降语法分析器：

$$S \rightarrow E$$
$$E \rightarrow E + T \mid E - T \mid T$$
$$T \rightarrow T * F \mid T/F \mid F$$
$$F \rightarrow id \mid num \mid (E)$$

则在构建以下分析函数 E()时：

```
1. void E(){
2.     switch(tok){
3.     case ?: E(); eat(PLUS); T(); break;
4.     case ?: E(); eat(MINUS); T(); break;
5.     case ?: T(); break;
6.     default: error();
7.     }
8. }
```

遇到了一个困境：假设当前的记号是整型数字(例如 33)，则函数 E()不知道该使用哪个子句来进行递归，因为函数 E()和 T()都能接受整型数字。

因此，递归下降语法分析只适合这样的文法：每个子表达式的第一个终结符号都能够为产生式的选择提供足够信息。为此，需要形式化 FIRST 集合的概念，然后给出一个能自动构建语法分析器的算法。

2. LL(1)分析算法

在一个递归下降分析器中，非终结符 X 的分析函数 X()对 X 的每个产生式都有一个子句，因此，该函数必须根据下一个输入符号 T 来选择其中的一个子句。如果能够为每一个(X,T) 的二元组都选出正确的产生式，就能够写出一个无冲突的递归下降分析器。因此，编译器可以将需要的所有信息，用一张关于产生式的二维表来记录，此表以文法的非终结符 X 和终结符 T 作为索引，称为预测分析表(predictive parsing table)。采用这张表的分析算法称为 LL(1)分析算法，其中第一个 L 表示从左向右读入被分析的源程序，第二个 L 表示产生最左推导，(1)表示编译器需要向前看 1 个终结符号来辅助进行决策。

1) 架构

LL(1)分析算法是基于表驱动的算法，其架构如图 3-9 所示。语法分析器自动生成器读入语法，然后生成一张分析表。语法分析器工作时通过查询分析表来决定将要进行的动作。有了分析表的指导，LL(1)分析算法就可以避免回溯。

为了构造预测分析表，需要先讨论 FIRST 集合和 FOLLOW 集合。

2) FIRST 集合和 FOLLOW 集合

给定一个由终结符和非终结符组成的字符串 γ，FIRST(γ)是由可以从 γ 推导出的句子开头的所有可能终结符组成的集合。例如，在上面的文法中，令 $\gamma = T * F$，则任何可从 γ 推

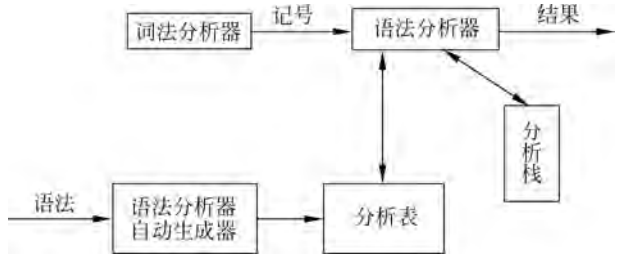


图 3-9　基于表驱动的 LL(1)分析算法架构

导出的由终结符组成的句子，都必定以 id、num 或(开始，因此，有：

$$\text{FIRST}(T * F) = \{\text{id}, \text{num}, (\}$$

FIRST 集合的计算初看并不复杂，若 $\gamma = XYZ$，则似乎可以忽略 Y 和 Z，只需计算 FIRST(X)。但是考虑下面的文法就可以看出情况并非如此：

$$X \rightarrow Y \mid a$$

$$Y \rightarrow \epsilon \mid c$$

$$Z \rightarrow XYZ \mid d$$

因为 Y 可能产生空串，所以 X 也可能产生空串，于是可以推出 FIRST(XYZ)一定包含 FIRST(Z)。因此，在计算 FIRST 集合时，必须跟踪能产生空串的符号，这种符号称为可空(nullable)符号。同时，还必须跟踪有可能跟随在可空符号之后的其他符号。

对于一个特定的文法，当给定由终结符和非终结符组成的字符串 γ 时，下述结论成立：

(1) 如果 X 可以导出空串，则 nullable(X)为真。

(2) FIRST(γ)是可从 γ 推导出的句子开头的所有可能终结符的集合。

(3) FOLLOW(X)是可直接跟随于 X 之后的终结符集合。如果存在着任一推导包含 X_t，则 $t \in$ FOLLOW(X)，当推导包含 XYZt，其中 Y 和 Z 都能推导出 ϵ 时，也有 $t \in$ FOLLOW(X)。

FIRST、FOLLOW 和 nullable 可定义为满足如下属性的最小集合：

```
对于每个终结符 Z,FIRST [Z] = Z
for 每个产生式 X → Y1Y2... Yk
    for 每个 i 从 1 到 k,每个 j 从 i+1 到 k
        if 所有 Yi都是可为空的
            then nullable[X] = true
        if Y1... Yi-1都是可为空的
            then FIRST [X] = FIRST [X] ∪ FIRST [Yi]
        if Yi+1... Yk都是可为空的
            then FOLLOW [Yi] = FOLLOW [Yi]∪FOLLOW [X]
        if Yi+1... Yj-1都是可为空的
            then FOLLOW [Yi] = FOLLOW [Yi]∪FIRST [Yj]
```

计算 FIRST、FOLLOW 和 nullable 的算法遵循的正是上述步骤，因此只需要简单地用一

个赋值语句替代每一个方程，并一直迭代到不动点，就可以计算出每个串的 FIRST、FOLLOW 和 nullable，这个迭代过程由算法 4 给出。算法接受文法 G=(T,N,P,S)作为输入，计算该文法的 FIRST、FOLLOW 和 nullable。从实现的角度看，这三个集合不必同时计算，可先计算 nullable，然后计算 FIRST，最后计算 FOLLOW。这样做可以加快算法的执行速度。

算法 4 FIRST、FOLLOW 和 nullable 的迭代计算

输入：文法 G=(T,N,P,S)
输出：FIRST、FOLLOW 和 nullable

```
1. procedure FirstFollow(T, N, P, S)
2.     将所有的 FIRST 和 FOLLOW 初始化为空集合，将所有的 nullable 都初始化为 false
3.     for 每一个终结符 Z do
4.         FIRST[Z] = {Z}
5.     repeat
6.         for 每个产生式 X → Y1 Y2 ... Yk do
7.             for 每个 i 从 1 到 k，每个 j 从 i + 1 到 k do
8.                 if 所有 Yi 都是可为空的 then
9.                     nullable[X] = true;
10.                if Y1 ... Yi-1 都是可为空的 then
11.                    FIRST[X] = FIRST[X] ∪ FIRST [Yi];
12.                if Yi+1 ... Yk 都是可为空的 then
13.                    FOLLOW [Yi] = FOLLOW [Yi] ∪ FOLLOW [X];
14.                if Yi+1 ... Yj-1 都是可为空的 then
15.                    FOLLOW [Yi] = FOLLOW [Yi] ∪ FOLLOW [Yj];
16.    until FIRST、FOLLOW 和 nullable 在此轮迭代中没有改变
```

S→ E $
E→ TE′
E′→ +TE′ | −TE′ | ε
T→ FT′
T′→ *FT′ | /FT′ | ε
F→ id | num | (E)

图 3-10 用于表达式的示例文法

将这一算法用于图 3-10 中给出的文法，首先计算这 6 个非终结符是否可为空，可以得到

$$\text{nullable}(E') = \text{nullable}(T') = \text{true}$$

即 E′ 和 T′ 是可为空的。接着将这个结果代入算法迭代计算 6 个非终结符的 FIRST 和 FOLLOW 集合，算法运行结束时，得到最终结果如表 3-4 所示。

表 3-4 nullable、FIRST 和 FOLLOW 的计算结果

非终结符	nullable	FIRST	FOLLOW
S	no	(id num	
E	no	(id num	) $
E′	yes	+ −	) $
T	no	(id num	) + − $
T′	yes	* /	) + − $
F	no	(id num	) * / + − $

可进一步将 FIRST 集合推广到符号串 X_γ 上：

$$\mathrm{FIRST}(X_\gamma)=\begin{cases}\mathrm{FIRST}[X], & \mathrm{nullable}[X]=\mathrm{no}\\ \mathrm{FIRST}[X]\cup\mathrm{FIRST}(\gamma), & \mathrm{nullable}[X]=\mathrm{yes}\end{cases}$$

并且，如果 γ 中的每个符号都是可为空的，则符号串 γ 可为空。

利用 FIRST 集合和 FOLLOW 集合，可以构造预测分析表：对每个 T∈FIRST(γ)，在表的第 X 行第 T 列，填入产生式 X→γ。此外，如果 γ 是可为空的，则对每个 T∈FOLLOW(X)，在表的第 X 行第 T 列，也填入该产生式 X→γ。

表 3-5 给出了图 3-10 中文法的预测分析表，其中省略了 num、/和－等终结符对应的列，它们和表中的其他项类似。对于一个给定的文法 G=(T,N,P,S)，我们可以使用工具自动构造出其预测分析表，这类工具被统称为语法分析器自动生成器。

根据预测分析表，可给出语法分析的 LL(1)算法，如算法 5 所示。该算法接受文法 G=(T,N,P,S)和记号流 tokens 作为输入，输出对 tokens 的分析结果。算法在分析过程中，需要根据分析表来决定产生式的选择(第 13 行)，因此避免了回溯的问题。

表 3-5　文法的预测分析表

非终结符	＋	*	id	(	)	$
S			S→E$	S→E$		
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→id	F→(E)		

算法 5　LL(1)分析算法

输入：文法 G=(T,N,P,S)和记号流 tokens
输出：对 tokens 的语法分析结果

```
1. procedure LL1(T, N, P, S, tokens)
2.     i = 0
3.     top = 0
4.     stack = [S]
5.     while stack != [] do
6.         if stack[top] 是一个终结符 t then
7.           if t == tokens[i ++] then
8.               pop()
9.           else
10.              error()
11.        else // stack[top] 是非终结符 T
12.          pop()
13.          push(table[T, tokens[i]])
```

3.2.5　自底向上分析

一个自底向上的语法分析过程对应于为输入串构造语法分析树的过程，它从叶子节点

(底部)开始构造,逐渐向上到达根节点(顶部)。可以将自底向上语法分析过程看成将一个串归约(Reduction)为文法开始符号 S 的过程。在每个规约步骤中,一个与某产生式 X→β 右部相匹配的特定子串 β,被替换为该产生式左侧的非终结符 X。

目前最重要的一类自底向上语法分析是 LR(k) 语法分析,其中符号 L 表示对输入进行从左到右的扫描,符号 R 表示构造一个最右推导序列,而符号(k)表示在进行语法分析决定时向前看 k 个输入符号。当 LR(k)分析看到的输入记号(多于 k 个单词),与正在考虑的产生式的整个右部对应时,LR(k)分析才确定使用哪一个产生式。

S'→ S $

S→ S; | id = E | print(L)

E→ id | num | E + E | (S, E)

L→E | L, E

图 3-11 直线式程序的语法

表 3-6 举例说明了用图 3-11 给出的文法,对程序

```
a = 7;
b = c + (d = 5 + 6, d)
```

进行 LR 分析的过程。

该分析器有一个栈和一个输入,输入中的前 k 个记号为向前查看的记号。根据栈的内容和向前查看的记号,分析器执行移进和归约两种动作。

(1) 移进:将第一个输入记号压入至栈顶。

(2) 归约:选择一个产生式 X→ABC,依次从栈顶弹出 C、B、A,然后将 X 压入栈。

开始时栈为空,分析器位于输入的开始。移进文件终结符 $ 的动作称为接受(Accepting),它表示分析过程成功结束。

表 3-6 列出了在每一个动作之后的栈和输入,也指明了所执行的动作,其中栈列的数字下标是 DFA 的状态编号。将栈和输入合并起来形成的一行总是构成一个最右推导。事实上,表 3-6 自下而上地给出了对输入字符串的最右推导过程。

表 3-6 一个句子的移进-归约分析

	栈	待分析字符串	输入动作
1		a=7;b=c+(d=5+6,d)	$ 移进
1	id_4	=7;b=c+(d=5+6,d)	$ 移进
1	$id_4 =_6$	7;b=c+(d=5+6,d)	$ 移进
1	$id_4 =_6 num_{10}$	;b=c+(d=5+6,d)	$ 归约 E→num
1	$id_4 =_6 E_{11}$	;b=c+(d=5+6,d)	$ 归约 S→id=E
1	S_2	;b=c+(d=5+6,d)	$ 移进
1	$S_2 ;_3$	b=c+(d=5+6,d)	$ 移进
1	$S_2 ;_3 id_4$	=c+(d=5+6,d)	$ 移进
1	$S_2 ;_3 id_4 =_6$	c+(d=5+6,d)	$ 移进
1	$S_2 ;_3 id_4 =_6 id_{20}$	+(d=5+6,d)	$ 归约 E→id
1	$S_2 ;_3 id_4 =_6 E_{11}$	+(d=5+6,d)	$ 移进
1	$S_2 ;_3 id_4 =_6 E_{11} +_{16}$	(d=5+6,d)	$ 移进
1	$S_2 ;_3 id_4 =_6 E_{11} +_{16} (_8$	d=5+6,d)	$ 移进

续表

	栈	待分析字符串	输入动作
1	$S_2;_3 id_4=_6E_{11}+_{16}(_8id_4$	=5+6,d)	$移进
1	$S_2;_3id_4=_6E_{11}+_{16}(_8id_4=_6$	5+6,d)	$移进
1	$S_2;_3 id_4=_6E_{11}+_{16}(_8 id_4=_6num_{10}$	+6,d)	$归约 E→num
1	$S_2;_3id_4=_6E_{11}+_{16}(_8id_4=_6E_{11}$	+6,d)	$移进
1	$S_2;_3id_4=_6E_{11}+_{16}(_8id_4=_6E_{11}+_{16}$	6,d)	$移进
1	$S_2;_3id_4=_6E_{11}+_{16}(_8id_4=_6E_{11}+_{16}$ num $_{10}$	,d)	$归约 E→num
1	$S_2;_3 id_4=_6E_{11}+_{16}(_8id_4=_6E_{11}+_{16}E_{17}$	,d)	$归约 E→E+E
1	$S_2;_3 id_4=_6E_{11}+_{16}(_8id_4=_6E_{11}$	,d)	$归约 S→id=E
1	$S_2;_3 id_4=_6E_{11}+_{16}(_8S_{12}$	,d)	$移进
1	$S_2;_3id_4=_6E_{11}+_{16}(_8S_{12},_{18}$	d)	$移进
1	$S_2;_3 id_4=_6E_{11}+_{16}(_8S_{12},_{18}$ id $_{20}$	)	$归约 E→id
1	$S_2;_3 id_4=_6E_{11}+_{16}(_8S_{12},_{18}E_{21}$	)	$移进
1	$S_2;_3 id_4=_6E_{11}+_{16}(_8S_{12},_{18}E_{21})_{22}$		$归约 E→(S,E)
1	$S_2;_3 id_4=_6E_{11}+_{16}E_{17}$		$归约 E→E+E
1	$S_2;_3 id_4=_6E_{11}$		$归约 S→id=E
1	$S_2;_3S_5$		$归约 S→S;S
1	S_2		$接受

1. LR 分析

LR 分析器通过确定的有限状态自动机来判断何时移进、何时归约。这种 DFA 不是作用于输入，而是作用于栈。DFA 的边用可以出现在栈中的符号(终结符和非终结符)来标记。表 3-7 是图 3-11 给出的文法的转换表。

这个转换表中的元素标有下面 4 种类型的动作。

(1) sn：移进到状态 n。

(2) gn：转换到状态 n。

(3) rk：用规则 k 规约。

(4) acc：接受。

(5) error：(用表中的空项来表示)。

为了使用该表进行分析，要将移进和转换动作看成是 DFA 边，并查看栈的内容。例如，若栈为 id=E，则 DFA 将从状态 1 依次转换到 4、6 和 11。若下一个输入记号是一个分号，例如状态 11 的“;”所在列则指出将根据规则 2 进行归约，因为文法的第二个规则是 S→id=E，于是栈顶的 3 个记号被弹出，同时 S 被压入栈顶。

在状态 11 中对于+的动作是移进，因此，如果下一个记号是+，它将从输入中被移出并压入栈中。对于每一个输入，分析器不是重新扫描栈，而是记住每一个栈元素所到达的状态。因此，分析算法通过查看栈顶状态和输入符号，从而得到对应的动作。对应的动作如下。

表 3-7 LR 分析表

状态	id	num	print	;	.	+	=	(	)	$	S	E	L
1	s4		s7								g2		
2				s3						a			
3	s4		s7								g5		
4							s6						
5				r1	r1					r1			
6	s20	s10						s8				g11	
7								s9					
8	s4		s7								g12		
9												g15	g14
10				r5	r5	r5			r5	r5			
11				r2	r2	s16				r2			
12				s3	s18								
13				r3	r3					r3			
14					s19				s13				
15					r8				r8				
16	s20	s10						s8				g17	
17				r6	r6	s16			r6	r6			
18	s20	s10						s8				g21	
19	s20	s10						s8				g23	
20				r4	r4	r4			r4	r4			
21									s22				
22				r7	r7	r7			r7	r7			
23					r9	s16			r9				

(1) 移进(n)：将 n 压入栈，并继续处理下一个输入记号。

(2) 归约(k)：从栈顶依次弹出符号，弹出符号的个数与规则 k 的右部符号个数相同。令 X 是规则 k 的左部符号，在栈顶现在所处的状态下，查看 X 得到动作“转换到 n”，则将 n 压入栈顶。

(3) 接受：停止分析，报告成功。

(4) 错误：停止分析，报告失败。

2. LR(0)分析算法

LR(k)分析器利用栈中的内容和输入中的前 k 个记号来确定下一步采取什么动作。表 3-7 说明了使用一个前看符号的情况，而当 k=2 时，这个表的每一列是由两个输入记号组成的序列，以此类推。在实际中，编译器很少需要使用 k>1 的表，一方面是因为这个表需要的存储空间非常大，另一方面是因为程序设计语言的文法都可以用 LR(1)文法来描述。

LR(0)文法是 LR(k)文法中最简单的一类，它只需查看分析栈就可完成分析，不必使用前看符号来决定进行移进还是归约。以图 3-12 中的文法为例来说明 LR(0)分析器的生成

过程。一开始，分析器的栈为空，输入是满足 S 的完整句子并以 $ 结束，即规则 S′→S$ 的右部都将出现在输入中。用一种扩展的产生式

$$S' \rightarrow \cdot\, S\$$$

来记录这一事实，其中产生式中的圆点 · 指出了分析器的当前位置。文法规则与指出其右部位置的圆点组合在一起称为项目(item)，由于正在讨论 LR(0)分析，因此也被称为 LR(0)项目。

在这个状态下，输入以 S 开始意味着它可能以产生式 S 的任何一个右部开始。用图 3-13 中的项目组成的集合来表示这种状态，其中 1 是项目集的编号。该项目集包括 3 条项目。

1) 移进动作

在状态 1，考虑当编译器移进(shift)一个终结符 x 的状态。此时栈顶为 x，并且项目 S→ · x 中的圆点将移到 x 之后。规则 S′→ · S$ 和 S→ · (L)与这个动作无关，因此忽略它们。于是得到状态 2，如图 3-14 所示。

S′→S$

S→(L)|x

L→S|L,S

图 3-12　满足 LR(0)的示例文法

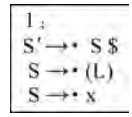


图 3-13　状态 1

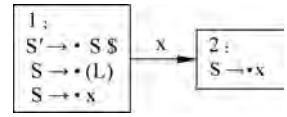


图 3-14　移进 x 到状态 2

或者在状态 1 也可考虑移进一个左括号“(”，将圆点“ · ”移到左括号的右边，得到项目 S→(· L)。此时栈顶一定为左括号“(”，并且输入中开头的应当是可从 L 推导出来的某个输入记号串，且其后跟随一个右括号“)”。通过将 L 的所有产生式都包含在项目集合中，可以确定什么样的记号可以作为输入中的开头记号。但是，在这些 L 的项目中，有一个项目的圆点“ · ”正好位于非终结符 S 之前，因此还需要包含所有 S 的产生式，最终得到的项目集合如图 3-15 所示。

2) 转换动作

如果在状态 1 已经分析过由非终结符 S 导出的记号串，则可以通过将状态 1 的第一个项目中的圆点“ · ”移到 S 之后来模拟这种情况，从而得到状态 4，如图 3-16 所示。这种情况发生在移进一个 x 求左括号，并随之用一个 S 产生式执行了规约时，然后该产生式的所有右部符号都将被弹出，并且分析器将在状态 1 对 S 执行转换(goto)动作。

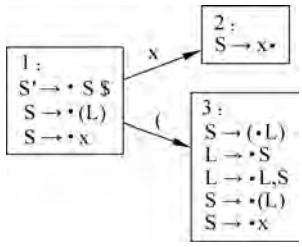


图 3-15　移进左括号到状态 3

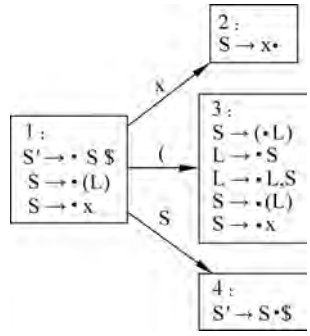


图 3-16　在状态 1 对 S 执行转换动作

3）归约动作

在状态 2 可以发现圆点“·”位于一个项的末尾，这意味着栈顶一定对应着产生式 S→x 完整的右部，并准备进行归约(Reduce)。在这种状态下，分析器可能会执行一个归约动作。

LR(0)分析算法中的两个基本操作是 Closure(I) 和 Goto(I,X)，其中 I 是一个项目集合，X 是一个文法符号(非终结符 N 或终结符 T)。当一个非终结符 X 的左侧有圆点时，函数 Closure 将更多的项目添加到项集合中；而函数 Goto 将圆点移到所有项目中的符号 X 之后。

算法 6 给出了计算 Closure(I)和 Goto(I,X)集合的算法。计算项目集 I 的 Closure(I)，首先将 I 状态中的核心项(除 S′→·S$ 外)中，所有的·不在最左端的项加入；然后对于每个项而言，如果有新的产生式，就把新的产生式加入进去。

算法 6 计算 Closure(I)和 Goto(I,X)

输入：项目集合 I,文法符号 X
输出：I 的闭包和 I 对于文法符号 X 的后继项目集合闭包

```
 1. procedure Closure(I, X)
 2.     repeat
 3.       for I 中的任意项 A → α · X β do
 4.         for 任意产生式 X → γ do
 5.           I = I ⋃ {X → ·γ}
 6.     until I 没有改变
 7.     return I
 8. procedure Goto(I, X)
 9.     将 J 设为空集
10.     for I 中的任意项 A → α · X β do
11.         将 A → αX · β 加入 J 中
12.     return Closure(J)
```

Goto(I,X)是项目集 I 对应于文法符号 X 的后继项目集闭包。首先将 J 初始化为空，然后对于项目集 I 中的每个项 A→α·Xβ，将 A→αX·β 加入集合 J，最后计算 Closure(J)。

算法 7 给出了 LR(0)分析器的构造算法。算法给文法增加一个辅助的开始产生式 S′→S$。令 B 是目前为止看到的状态集合，E 是目前为止已找到的(移进或转换)边集合。

算法 7 LR(0)分析

输入：文法 G=(T,N,P,S)
输出：文法的 LR(0) 分析器

```
 1. procedure LR0(G = (T, N, P, S))
 2.     B = {Closure(S'→·S$)}
 3.     E = Ø
 4.     repeat
 5.         for B 中的每个状态 I do
 6.          for I 中的每一项 A → α·X β do
 7.              J = Goto(I, X)
 8.              B = B ⋃ {J}
 9.              E = E ⋃ {I --X--> J}
10.   until E 和 B 在本轮迭代中没有改变
11.   return I
```

但是对于符号 $，不计算 Goto(I，$)，而是选择了接受动作。图 3-17 以图 3-12 的文法为例说明了该分析器的分析过程。

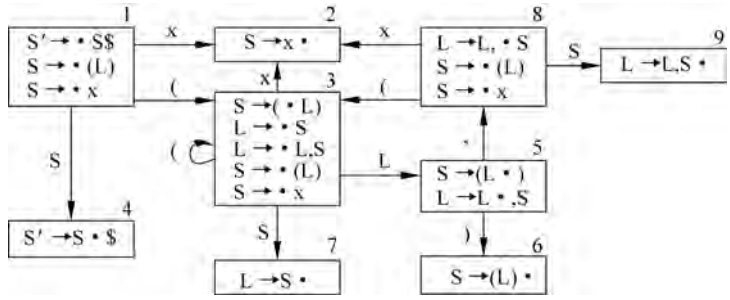


图 3-17　图 3-12 文法的 LR(0)状态

现在可以通过算法 8 来计算 LR(0) 的归约动作集合 R，并且能够为该文法构造一个分析表(表 3-8)。对于每一条边 $I \xrightarrow{X} J$，若 X 为终结符，则在表位置(I，X)中放置动作移进 J(sJ)；若 X 为非终结符，则将转换 J(gJ) 放在位置(I，X) 中。对于包含项 S′→S • $ 的每个状态 I，在位置(I，$) 中放置动作接受(a)。对于包含项 A→γ •(尾部有圆点的产生式 n)的状态，对每一个记号 Y，放置动作归约 n(rn)于(I，Y)中。

算法 8　计算归约动作集合

输入：目前为止看到的状态集合 B
输出：规约动作集合 R

```
1. procedure reduceset(B)
2.     R = {}
3.     for B 中的每个状态 I do
4.         for I 中的每个项 A →α·∈ I do
5.             R = R ∪ {(I, A → α)}
```

表 3-8　图 3-12 文法的 LR(0)分析表

状态	(	)	x	,	$	S	L
1	s3		s2			g4	
2	r2	r2	r2	r2	r2		
3	s3		s2			g7	g5
4					a		
5		s6		s8			
6	r1	r1	r1	r1	r1		
7	r3	r3	r3	r3	r3		
8	s3		s2			g9	
9	r4	r4	r4	r4	r4		

因为 LR(0)不需要向前查看，所以原则上每个状态只需要一个动作：要么移进，要么归约，但不会两者兼有。在实际应用中，由于还需要知道要移进至哪个状态，所以表 3-8 以状态号作为行，以文法符号作为列。

3. SLR 分析算法

尝试构造如图 3-18 所示文法的 LR(0)分析表，它的 LR(0)状态如图 3-19 所示。

图 3-18　表达式的示例文法

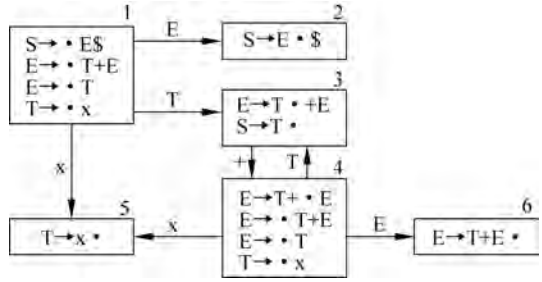


图 3-19　图 3-18 所示文法的 SLR 分析表

在状态 3，对于符号＋，有一个多重定义的项：分析器必须移进到状态 4，同时又必须用产生式 2 进行归约。这是一个冲突，它表明该文法不是 LR(0)的，因为它不能用 LR(0)分析器分析，因此需要一种能力更强的分析算法。

比 LR(0)更好的一种简单的分析器为 SLR，即 Simple LR 的简称。SLR 分析器的构造几乎与 LR(0)相同，区别是只在 FOLLOW 集合指定的地方放置归约动作。

算法 9 是在 SLR 表中放置归约动作的算法。

算法 9　计算归约动作集合

输入：目前为止看到的状态集合 T
输出：规约动作集合 R

```
1. procedure reduceset
2.    R = {}
3.    for T 中的每个状态 I do
4.       for I 中的每个项 A → α·do
5.          for FOLLOW (A)中的每个记号 X do
6.             R = R ∪ {(I, X, A → α)}
```

动作(I,X,A→α)指出，在状态 I，对于前看符号 X，分析器将用规则 A→α 进行归约。

因此，对于图 3-18 所示文法，尽管使用相同的状态图(图 3-19)，但如表 3-9 所示，在 SLR 表中放置的归约动作要少些。

SLR 文法类是其 SLR 分析表不含冲突(多重表项)的那些文法。图 3-18 所示文法即属于这一类，很多常用的程序设计语言的文法也属于这一类。

表 3-9　图 3-18 所示文法的 SLR 分析表

状态	x	+	$	E	T
1	s5			g2	g3
2			a		
3		s4	r2		
4	s5			g6	g3
5		r3	r3		
6			r1		

4. LR(1) 分析算法

比 SLR 更强大的是 LR(1)分析算法。大多数用上下文无关文法描述其语法的程序设计语言含有一个 LR(1)文法。构造 LR(1)分析表的算法与构造 LR(0)分析表的算法相似，只是项的概念要复杂一些。

一个 LR(1)项由一个文法产生式、一个右部位置(用圆点表示)和一个前看符号组成。项(A→α·β,x)指出：序列 α 在栈顶，且输入中开头的是可以从 βx 导出的符号串。

LR(1)状态是由 LR(1)的项组成的集合，并且存在着合并该前看符号的 LR(1)的 Closure 和 Goto 操作。算法 10 给出了计算 Closure(I)和 Goto(I,X)集合的算法。

算法 10　计算 Closure(I)和 Goto(I,X)

输入：项目集合 I,前看符号 z
输出：I 的闭包和后继项目集合闭包

```
1. procedure Closure(T)
2.      repeat
3.          for I 中的任意项 (A → α · Xβ, z) do
4.              for 任意产生式 X → γ do
5.                  for 任意 ω ∈ FIRST (βz) do
6.                      I = I ⋃ {(X → · γ, ω)}
7.      until I 没有改变
8.      return I
9. procedure Goto(I, X)
10.     J = {}
11.     for I 中的任意项 (A → α · Xβ, z) do
12.         将 (A → αX · β, z) 加入 J 中
13.     return Closure(J)
```

开始状态是项 (S′→ · S $,?) 的闭包，其中前看符号？具体是什么不重要，因为文件结束标志绝对不会被移进。

对于 Closure(I)的计算，对 I 中的任意项(A→α·Xβ,z)，任意产生式 X→γ，如果 ω 属于 FIRST(βz)，则将(X→· γ,ω)加入 I，直到 I 中的项不再发生变化。

对于 Goto(I,X)的计算，首先将 J 初始化为空，然后对于项目集 I 中的每个项(A→α·Xβ,z)，将(A→αX·β,z)加入集合 J，最后计算 Closure(J)。

归约动作用算法 11 来选择。动作(I,z,A→α)指出,在状态 I 看到前看符号 z 时,分析器将用规则 A→α 进行归约。

算法 11 计算归约动作集

输入: 目前为止看到的状态集合 T,前看符号 z
输出: 规约动作集合 R

```
1. procedure reduceset(T , z)
2.      R = {}
3.      for T 中的每个状态 I do
4.          for I 中的每个项 (A → α·, z) do
5.              R = R ⋃ {(I, z, A → α)}
```

图 3-20 所示文法不是 SLR,但它属于 LR(1) 文法,图 3-21 给出了该文法的 LR(1) 状态。图 3-20 所示文法中有几个项有相同的产生式,但其前看符号不同(如左图所示),可以将它们简化为右图所示。

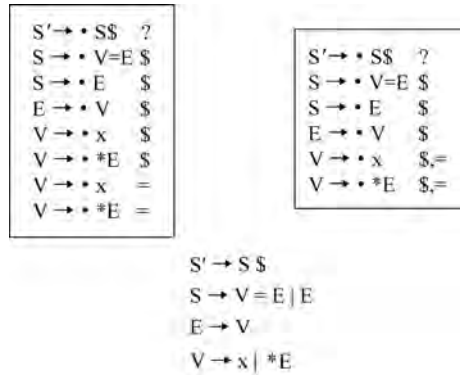


图 3-20 一个描述 C 语言中的表达式、变量和指针访问运算(*)的文法

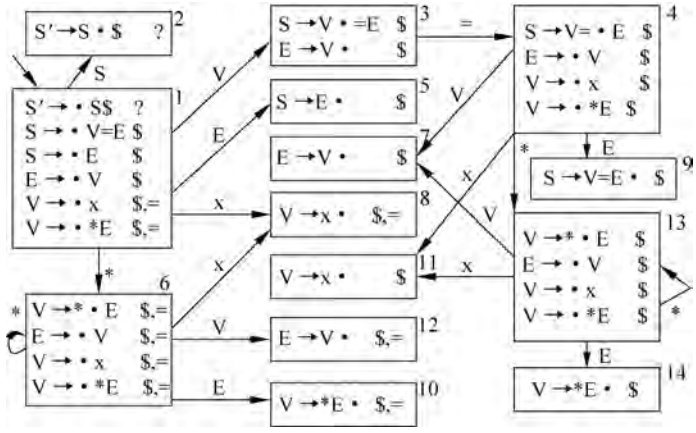


图 3-21 图 3-20 所示文法的 LR(1)状态

图 3-20 所示文法的 LR(1)状态表(表 3-10(a))是从图 3-21 导出的 LR(1) 分析表。只要在产生式的末尾有圆点,在 LR(1)表中与状态号对应的行和与项的前看符号对应的列的位置,就存在着那个产生式的一个归约动作(在这个例子中,前看符号是 $)。只要圆点位于终结符或非终结符的左边,在 LR(1)分析表中就存在相应的移进或转换动作。

表 3-10　图 3-20 所示文法的 LR(1)分析表和 LALR(1)分析表

状态	x	*	=	$	S	E	V	状态	x	*	=	$	S	E	V
1	s8	s6			g2	g5	g3	1	s8	s6			g2	g5	g3
2				a				2				a			
3			s4	r3				3			s4	r3			
4	s11	s13				g9	g7	4	s8	s6				g9	g7
5				r2				5				r2			
6	s8	s6				g10	g12	6	s8	s6				g10	g7
7				r3				7			r3	r3			
8			r4	r4				8			r4	r4			
9				r1				9				r1			
10			r5	r5				10			r5	r5			
11				r4											
12			r3	r3											
13	s11	s13				g14	g7								
14				r5											

(a) LR(1)　　(b) LALR(1)

5. LALR(1)分析算法

LR(1)分析表有很多状态,因此非常大,但是通过合并除前看符号集合外其余部分都相同的两个状态,可得到一个较小的表。由此得到的分析器称为 LALR(1)分析器,即前看 LR(1)(Look-Ahead LR(1))。

例如,在图 3-20 所示文法的 LR(1)状态(图 3-21)中,如果忽略前看符号集合,状态 6 和状态 13 的项是相同的。状态 7 和状态 12,状态 8 和状态 11 以及状态 10 和状态 14 也都如此。合并这些状态对,就得到表 3-10(b)所示的 LALR(1)分析表。

对于某些文法,LALR(1)表含有归约-归约冲突,而在 LR(1)表中却没有这种冲突。不过,实际中这种可能性很小,重要的是和 LR(1)表相比,LALR(1)分析表的状态要少得多,因此它需要的存储空间要少于 LR(1)表。

3.3　语义分析

语法正确的输入程序仍然可能包含其他错误,导致编译无法完成。为了检测这样的错误,编译器就需要进行更深层的语义分析(semantic analysis)。语义分析需要将程序结构放

到实际的上下文中进行检查，发现类型等方面的错误。语义分析根据源语言的语义规则，对语法分析阶段生成的抽象语法树的属性进行上下文相关检查；如果检查出错，则返回错误信息给程序员，供程序员修改代码；如果检查通过，编译器生成一个中间表示供编译的后续阶段使用。本节首先讨论抽象语法树的定义和生成技术，然后介绍在抽象语法树上进行语义分析的主要技术。

3.3.1 抽象语法树

语法分析树(parse tree)编码了句子的推导过程，每个输入记号对应着语法分析树中的一个叶子节点，分析期间被归约的每一个语法规则都对应着树中的一个内部节点。例如，算术表达式 15 * (3+4)对应的一棵语法分析树如图 3-22 所示。

语法分析树包含了很多不必要的信息，这些节点会占用额外的存储空间。此外，语法分析树的结构高度依赖文法细节，而各种文法转换又会引入新的非终结符和产生式，这些细节本应被限制在语法分析阶段，不应该对语义分析造成困扰。

因此，在编译器设计中，通常要对语法分析树进一步抽象，去掉无关信息，形成更紧凑的内部表示——抽象语法树(Abstract Syntax Tree，AST)。抽象语法树保留了语法分析树的基本结构，表达式的优先级和语义仍保持原样，但剔除了其中非必要的节点。对于同样的算术表达式 15 * (3+4)，其抽象语法树如图 3-23 所示。

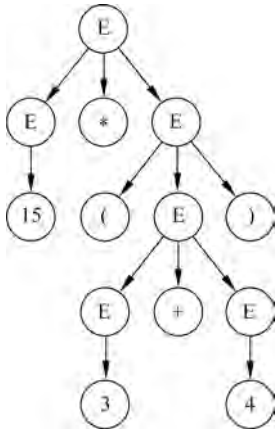


图 3-22 15 * (3+4)对应的语法分析树

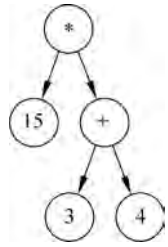


图 3-23 15 * (3+4)对应的抽象语法树

抽象语法(abstract syntax)是编译器用来表达程序语法结构的内部数据结构表示，现代编译器一般都采用抽象语法作为编译器前端(语法分析和词法分析)和后端的接口。语义分析阶段使用程序的抽象语法树分析程序中存在的语义错误。

在编译器中，为了定义抽象语法树，需要使用实现语言来定义一组数据结构。例如，对于如下的文法(做了适当抽象，略去了文法中不重要的分隔符等)：

$$E \rightarrow n \mid id \mid E + E \mid E * E$$

$$S \rightarrow id = E \mid if\ (E,S,S) \mid while\ (E,S) \mid return\ E$$

可以定义如下数据结构(类 C 语言描述)来编码该文法：

```
enum E_kind {E_INT, E_ID, E_ADD, E_TIMES};
struct Exp{
  enum E_kind kind;
};
struct Exp_Int{
  enum E_kind kind;
  int n;
};
struct Exp_Id{
  enum E_kind kind;
  char * id;
};
struct Exp_Add{
  enum E_kind kind;
  struct Exp * left;
  struct Exp * right;
};
struct Exp_Times{
  enum E_kind kind;
  struct Exp * left;
  struct Exp * right;
};
```

首先定义一个枚举类型 E_kind,它包括四个不同的枚举类型的值,分别表示表达式的四种情况。再定义一个结构体 Exp 来编码表达式产生式左部的非终结符 E,该结构体中只有一个 kind 字段。接下来,为非终结符 E 的每个右部都定义一个结构体,例如,用 Exp_Int 来编码产生式的第一个右部,即整型立即数表达式,该结构体的第一个域 kind 表示表达式的类型,第二个域 n 记录了这个整型数具体的值；用结构体 Exp_Add 来表示加法表达式,其中第一个域 kind 表示表达式的类型,第二个域 left 和第三个域 right 表示加法运算的两个表达式操作数,这两个操作数的类型都是指向结构体 Exp 的指针。

对于语句 S,可以类似地定义抽象语法树的数据结构：

```
enum S_kind {S_ASSIGN, S_IF, S_WHILE, S_RETURN};
struct Stm{
    enum S_kind kind;
};
struct Stm_Assign{
    enum S_kind kind;
    char * id;
    struct Exp * exp;
};
struct Stm_If{
  enum S_kind kind;
  struct Exp * exp;
  struct Stm * thenn;
  struct Stm * elsee;
```

```
15. };
16. struct Stm_While{
17.    enum S_kind kind;
18.    struct Exp * exp;
19.    struct Stm * body;
20. };
```

这些数据结构的含义和表达式的类似，在此不再赘述。

编译器需要实现合理的内存分配接口，为这些数据结构分配合理的内存空间。下面给出了典型的分配接口的实现：

```
 1. struct Exp_Int * Exp_Int_new(int n){
 2.    struct Exp_Int * p = malloc(sizeof( * p));
 3.    p->kind = E_INT;
 4.    p->n = n;
 5.    return p;
 6. }
 7.
 8. struct Exp_Add * Exp_Add_new(struct Exp * left, struct Exp * right){
 9.    struct Exp_Add * p = malloc(sizeof( * p));
10.    p->kind = E_ADD;
11.    p->left = left;
12.    p->right = right;
13.    return p;
14. }
15.
16. struct Stm_If * Stm_If_new(struct Exp * exp, struct Stm * thenn, struct Stm * elsee){
17.    struct Stm_If * p = malloc(sizeof( * p));
18.    p->kind = S_IF;
19.    p->exp = exp;
20.    p->thenn = thenn;
21.    p->elsee = elsee;
22.    return p;
23. }
```

这些数据结构分配接口的实现类似，都是先申请内存空间，再进行合理的初始化。实际的生产级的编译器还要考虑分配中的错误处理，以及内存的回收等更多实现细节。

有了这些表达抽象语法树的数据结构，编译器就可以在语法分析的过程中自动构建程序对应的抽象语法树。具体地，在递归下降语法分析器中，语义动作分散在实现语法分析的代码中；而在 LR 语法分析器中，编译器可以在分析器的语法动作中，加入生成语法树的代码片段，来自动生成抽象语法树。

在递归下降语法分析器中，抽象语法树自动生成的部分代码如下：

```
1. struct Exp * parse_E(){
2.     E t = parse_E_term();
3.     while(cur_token == '+'){
4.       eat('+');
5.       E t1 = parse_E_term();
```

```
6.        t = Exp_Add_new(t, t1);
7.      }
8.      return t;
9.  }
10.
11. struct Stm * parse_S(){
12.     switch(cur_token){
13.     case ID:
14.       String x = cur_token; //remember the identifier
15.       eat('=');
16.       struct Exp * e = parse_E();
17.       S s = Stm_Assign_new(x, e);
18.       return s;
19.     case IF:
20.       eat(IF);
21.       eat('(');
22.       struct Exp * e = parse_E();
23.       eat (')');
24.       eat(THEN);
25.       struct Stm * thenn = parse_S();
26.       eat(ELSE);
27.       struct Stm * elsee = parse_S();
28.       struct Stm * iff = Stm_If_new(e, thenn, elsee);
29.       return iff;
30.     }
31. }
```

以对条件语句 if 的语法分析核心过程为例，首先编译器递归调用 parse_E 得到条件语句表达式的抽象语法树 e，然后再递归调用两次 parse_S 得到条件语句两个分支 thenn 和 elsee 的抽象语法树，最后构建并返回整个语句的抽象语法树 iff。

编译器对抽象语法树的各种处理，如语义检查、中间代码生成等，本质上都是对抽象语法树的某种形式的遍历过程。后续章节将深入讨论语义检查和中间代码生成等问题，这里首先考虑输出抽象语法树，这个功能对于验证语法树构建的正确性非常重要。

```
1.  void print_exp(struct Exp * e){
2.      switch(e->kind){
3.      case E_INT:
4.        printf("%d", e->n);
5.      return;
6.      case E_ADD:
7.        printf("(");
8.        print_exp(e->left);
9.        printf(")");
10.       printf("+");
11.       printf("(");
12.       print_exp(e->right);
13.       printf(")");
14.       return;
15.     other cases: /* similar */
```

```
16.     }
17. }
18.
19. void print_stm(struct Stm * s){
20.     switch(s->kind){
21.     case S_ASSIGN:
22.       printf(" %s", s->id);
23.       printf(" = ");
24.       print_exp(s->exp);
25.       return;
26.     case S_IF:
27.       printf("if(");
28.       print_exp(e->left);
29.       printf(")");
30.       printf("then");
31.       print_stm(s->thenn);
32.       printf("else");
33.       print_stm(s->elsee);
34.       return;
35.     other cases: /* similar */
36.     }
37. }
```

函数 print_exp 和 print_stm 分别完成对表达式 E 和语句 S 的打印，它们的实现机制类似，都是基于对语法树具体形式的分类讨论，并进行必要的递归调用。

最后要注意，抽象语法树是编译器前端和后端重要的数据结构接口，程序一旦被转换为抽象语法树，则源代码可能会被丢弃，编译器后续阶段只处理抽象语法树。所以抽象语法树只有编码源代码足够多的信息，例如，它必须编码每个语法结构在源代码中的位置（文件名、行号、列号等），后续的编译检查阶段才能获得精确的源代码信息。

3.3.2 符号表

符号表（symbol table）是编译器用来保存有关源程序各种信息的重要数据结构。在分析阶段，编译器把源程序的信息逐步收集到符号表中。符号表的每个条目包含与一个标识符相关的所有必要信息，比如它的符号名、类型、存储位置、作用域和其他相关信息。符号表中的信息会在编译的后续不同阶段用到，在语义分析阶段，符号表的内容将用于语义检查和产生中间代码。在目标代码生成阶段，当对标识符进行地址分配时，符号表是地址分配的依据。

在整个编译期间，编译器对符号表的操作大致可归纳为以下 5 类。

（1）创建符号表：在编译开始时或进入一个子程序时进行。

（2）插入表项：在遇到新的标识符声明时进行。

（3）查询表项：在引用声明过的标识符时进行。

（4）修改表项：在获得新的语义值信息时进行。

（5）删除表项：删除一个或一组无用的项时进行。

根据操作的需求，可给出一个符号表实现的接口，它指明了编译器对符号表的典型需求：

```
type SymTable, K, V;
SymTable create();
void insert(SymTable, K, V);
V lookup(SymTable, K);
void update(SymTable, K, V);
void remove(SymTable, K);
```

类型 SymTable、K 和 V 分别代表符号表类型、关键字类型和值类型。接口中的 5 个函数分别完成符号表的创建、插入、查找、更新和删除操作。

由于被编译程序中标识符的规模可以非常大，且在编译器的各个阶段，每当遇到一个标识符都要查找符号表，因此编译器需要合理组织符号表，使符号表本身占据的存储空间尽量少，同时尽量提高编译期间对符号表的访问效率。

在编译器的实际构造过程中，编译器实现者需要仔细权衡时间开销、空间开销、所编译程序语言的特点等因素，来合理选择实现符号表的数据结构。如果提高对符号表的访问效率优先级更高，编译器可以使用哈希表作为符号表的具体实现。哈希表为查找操作提供了常数时间 O(1)的访问能力。在使用哈希表时，必须合理地处理碰撞，还要仔细地处理装载因子，避免哈希表浪费过多空间。而如果节约空间的优先级更高，编译器也可以使用红黑树等平衡树数据结构，尽管此时查找操作的时间复杂度变为 O(log N)，但避免了空间浪费。

很多程序设计语言中的变量都有变量作用域(scope)的概念，该变量仅在其作用域中是可见的。当编译器到达每一个作用域的开始时，需要把涉及的变量放入符号表；而到作用域结束时，需要将此作用域中的变量移除。为了处理作用域，编译器可以在哈希表上进行适当地记录，进入作用域时插入元素，退出作用域时删除元素；为了提高效率，编译器还可以采用由符号表构成的栈来处理作用域，进入作用域时插入新的符号表到栈顶，退出作用域时，删除栈顶符号表。

3.3.3　语义检查

编译器语义检查阶段的任务首先是将变量的定义与它们的各个使用联系起来，以检查程序(抽象语法树)的上下文相关的属性。例如变量在使用前先进行声明、每个表达式都有合适的类型、函数调用与函数的定义一致等。然后，编译器将抽象语法树转换成更简单的、适合于生成机器代码的中间表示。语义分析又称为上下文相关分析或类型检查。

语义分析器的输入是语法分析阶段生成的抽象语法树和程序语言的语义规则，输出是一个中间表示。传统上，大部分的程序设计语言采用自然语言来表达程序语言的语义，所以编译器的实现者必须对语言中的语义规则有全面的理解。

下面的文法：

$$P \rightarrow D\ S$$

$$D \rightarrow T\ \mathrm{id}\ ;\ D \mid \epsilon$$

$$T \rightarrow int \mid bool$$
$$S \rightarrow id = E \mid printi(E) \mid printb(E)$$
$$E \rightarrow n \mid id \mid true \mid false \mid E + E \mid E \&\& E$$

给定了一个包含变量声明的简单程序设计语言的片段。文法中的 P 表示程序，它由声明 D 和随后的语句 S 组成；变量声明 D 本质上是一个列表，包含一串由类型和变量构成的二元组；类型 T 包括整型类型 int 和布尔类型 bool 两种；语句 S 包括 3 种语句形式，即赋值语句、对整型的输出语句 printi 和对布尔型的输出语句 printb；表达式 E 包括整型加法运算和布尔型的与运算。尽管这个语言并不复杂，但它能够很好地阐释语义检查中的关键问题。

对于这个语言，可以给出对表达式 E 的类型检查实现(用类 C 的代码实现)：

```
1. enum type {INT, BOOL};
2. SymTable table;
3. enum type check_exp(struct Exp * e){
4.   switch(e->kind){
5.   case EXP_INT:
6.       return INT;
7.   case EXP_TRUE:
8.       return BOOL;
9.   case EXP_FALSE:
10.      return BOOL;
11.  case EXP_ID:
12.    enum type t = SymTable_lookup(table, id)
13.    if(id not found)
14.        error("id not found");
15.    else
16.        return t;
17.  case EXP_ADD:
18.      enum type t1 = check_exp(e->left);
19.      enum type t2 = check_exp(e->right);
20.      if(t1!= INT || t2!= INT)
21.          error ("type mismatch for +");
22.      else
23.          return INT;
24.  case EXP_AND:
25.      enum type t1 = check_exp(e->left);
26.      enum type t2 = check_exp(e->right);
27.      if(t1!= BOOL || t2!= BOOL)
28.          error ("type mismatch for &&");
29.      else
30.          return BOOL;
31.  }
32. }
```

函数 check_exp 完成对输入参数表达式 e 的类型检查，返回该表达式的类型。该函数本质上完成了对表达式 e 的后序遍历：如果 e 是一个整型常量，则返回类型 INT；如果 e 是

true 或 false，则函数返回类型 BOOL；如果 e 是一个标识符 id，则函数首先在符号表 table 中查找这个标识符，如果查找成功，则返回标识符的类型 t，如果查找失败，则输出合适的错误处理信息。对于复合表达式，函数先进行递归调用来检查两个子表达式 left 和 right 的类型 t1 和 t2，对结果进行检查后，再返回表达式的类型（如果出错，则输出合适的错误信息）。

有了对表达式 E 的类型检查函数，编译器就可以类似地实现对语句 S 的类型检查功能：

```
1. void check_stm(struct Stm * s){
2.      switch(s->kind){
3.      case STM_ASSIGN:
4.        enum type t1 = SymTable_lookup(s->id);
5.        enum type t2 = check_exp(s->exp);
6.        if(t1!= t2)
7.          error("type mismatch for =");
8.        else
9.          return INT;
10.     case STM_PRINTI:
11.       enum type t = check_exp(s->exp);
12.       if(t!= INT)
13.         error("type mismatch for printi()");
14.       else
15.         return;
16.     case STM_PRINTB:
17.       t = check_exp(s->exp);
18.       if(t!= BOOL)
19.         error("type mismatch for printb()");
20.       else
21.         return;
22.   }
23.}
```

对于赋值语句，编译器从符号表中查找被赋值变量的类型 t1，并将其与赋值右侧表达式的类型 t2 做比较；对于打印语句，编译器递归检查打印参数的类型并做适当检查。

最后，给出对变量声明 D 和程序 P 的类型检查算法：

```
1. void check_dec(enum type t, char * id){
2.      SymTable_insert(SymTable, id, t);
3. }
4.
5. void check_prog(struct Dec * d, struct Stm * s){
6.      check_dec(d);
7.      check_stm(s);
8. }
```

函数 check_dec 处理每一个变量声明，将其放入符号表 SymTable 中；函数 check_prog 先对变量声明 d 进行类型检查，再完成对语句 s 的类型检查。

尽管更实际的编程语言可能包括更复杂的语言机制，如作用域、命名空间、更复杂的类

型规则等，对其进行语义检查实现起来会更加复杂，但对其进行类型检查的基本原理和技术和上述讨论类似，编译器都要围绕符号表，仔细检查每项语言特性是否满足语言语义规范的要求。

3.4 小结

编译器前端负责读入程序的源代码，对其进行合法性检查，并编译到合适的中间表示。本章讨论了编译器前端的3个组成部分：词法分析、语法分析和语义分析。词法分析负责读入程序的字符流，并输出记号流；语法分析负责读入程序的记号流，对程序的语法合法性进行分析和检查，并构建抽象语法树的数据结构，抽象语法树是编译器前端最重要的数据结构表示；语义分析负责检查程序是否符合语言的语义规则，并把程序进一步翻译成中间表示供后续阶段处理。

3.5 深入阅读

Ravi Sethi 等提出了通过在缓冲区末尾放置一个敏感标记(sentinel)，一个不属于任何记号的字符，词法分析器就有可能只对每个记号进行一次检查，而不是对每个字符都进行检查。Bumbulis 和 Cowan 的方法只需对 DFA 中的每一次循环检查一次，当 DFA 中存在很长的路径时，这样可以减少检查的次数(相对每个字符一次)。

Conway 在介绍一个递归下降(预测)分析器的同时，描述了 FIRST 集合和提取左因子的概念。LL(k)分析理论是由 Lewis 和 Stearns 形式化的。

LR(k)语法分析方法是由 Knuth 开发的。Backhouse 给出了关于 LL 和 LR 分析法理论的介绍。Aho 等说明了利用优先级指导命令解决其中的二义性，使得确定的 LL 或 LR 语法分析引擎能够处理二义性文法。Burke 和 Fisher 发明了一种通过管理一个包含 K 个单词的队列和两个分析栈来实现错误修复的策略。

许多编译器将递归下降分析代码与语义动作混合在一起进行，Gries、Fraser 和 Hanson 给出了采用这种方法的早期编译器和现代编译器的例子。抽象语法的表示最早由 McCarthy 提出。

3.6 习题

1. 将下面正则表达式转换为上下文无关文法：

(1) ((xy * x)|(yx * y))?。

(2) ((0|1)+".")(0|1) *)|((0|1) * ".")(0|1)+)。

2. 分别为下面语言写出一个无二义性的文法。

(1) 字母表 a,b 上的回文(即无论顺读、倒读都相同的字符串)。

(2) 与正则表达式 a * b * 相匹配且 a 多于 b 的字符串。

(3) 配对的圆括号和方括号,例如 ([[]((()[()][])])。

(4) 配对的圆括号和方括号,但其中闭方括号也关闭未配对的开圆括号(一直到前一个开方括号),例如 [([](()[(][])]。提示:首先,写出圆括号配对和方括号配对的语言,并允许有额外的开圆括号;然后保证这个开圆括号必须出现在方括号内。

(5) 关键字 public、final、static、synchronized、transient 组成的所有子集和排列(无重复)。

3. 对于下面文法:

$$S \rightarrow uBDz$$
$$B \rightarrow Bv \mid w$$
$$D \rightarrow EF$$
$$E \rightarrow y \mid \epsilon$$
$$F \rightarrow x \mid \epsilon$$

(1) 计算文法的 nullable、FIRST 和 FOLLOW 集合。

(2) 构造 LL(1)分析表。

(3) 给出证据说明该文法不是 LL(1)文法。

(4) 尽可能少地修改该文法使它成为一个接受相同语言的 LL(1) 文法。

4. 给定如下的文法:

$$S \rightarrow E\$$
$$E \rightarrow id$$
$$E \rightarrow id\ (E)$$
$$E \rightarrow E + id$$

(1) 构造这个文法的 LR(0)自动机。

(2) 它是 LR(0)文法吗? 给出理由。

(3) 它是 SLR 文法吗? 给出理由。

(4) 它是 LR(1)文法吗? 给出理由。

5. 说明下面这个文法是 LALR(1),但不是 SLR:

$$S \rightarrow X\$$
$$X \rightarrow Ma \mid bMc \mid dc \mid bda$$
$$M \rightarrow d$$

6. 说明下面这个文法是 LL(1),但不是 LALR(1):

$$S \rightarrow (X \mid E\ ] \mid F\ )$$
$$X \rightarrow E\ ) \mid F\ ]$$
$$E \rightarrow A$$
$$F \rightarrow A$$
$$A \rightarrow \epsilon$$

7. 假定你正在为一种简单的、具有词法作用域的语言编写编译器。考虑如下所示的程序。

```
1. procedure main
2.     integer a,b,c;
3.     procedure f1(w,x);
4.         integer a,x,y;
5.         call f2(w,x);
6.         end;
7. procedure f2(y,z)
8.     integer a,y,z;
9.     procedure f3(m,n)
10.        integer b,m,n;
11.        c = a * b * m * n;
12.        end;
13.        call f3(c,z);
14.        end;
15.        ...
16.    call f1(a,b);
17.    end;
```

(1) 编译器处理到 11 行时,请绘制对应的符号表及其内容。

(2) 在语法分析器进入一个新的过程和退出一个过程时,需要哪些操作来管理符号表?

8. 最常见的符号表实现技术是使用哈希表,其中插入和删除的预期代价是 O(1)。

(1) 哈希表中插入和删除操作在最坏情况下的代价如何?

(2) 提议一种备选实现方案,以保证 O(1)时间内完成插入和删除操作。