编译原理

1 绪论

1.1 程序设计语言的发展

第一代，机器语言：能够被计算机的硬件系统直接执行的指令程序。

第二代，汇编语言：将硬件指令用一些助记符表示，即符号化的机器语言。用助记符（memoni）代替操作码，用地址符号（symbol）或标号（label）代替地址码。

第三代，高级语言：从程序员的角度出发，对汇编语言进一步抽象，使用便于理解的“自然语言”表述。

1.2 高级语言的实现

编译方式：源语言为高级语言，目标语言是低级语言（汇编或机器语言）的翻译程序。如图 1.1 所示。

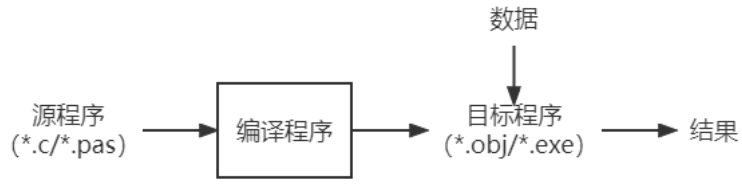


图1.1 编译方式

解释方式：一边翻译一边执行，翻译完的同时也执行完了程序。如图 1.2 所示。

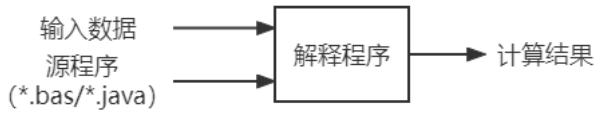


图1.2 解释方式

解释器和编译器的比较：

1. 解释器是执行系统，编译器是转换系统。
2. 解释器一次翻译目标程序只能执行一次，编译器一次翻译目标程序可执行多次。
3. 基于解释执行的程序可以动态修改自身，而基于编译执行的程序不易胜任。
4. 基于解释方式有利于人机交互。
5. 解释器执行速度更慢。
6. 解释器需要保存的信息较多，空间开销大。
7. 二者实现技术相似。

转换方式：是一种变通的方式，假如已有 B 语言的编译器，就可以把 A 语言程序转换为 B 语言程序，用 B 语言已有的编译器去编译执行。如图 1.3 所示。

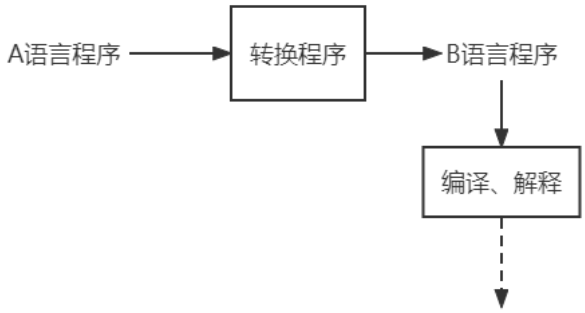


图1.3 转换方式

1.3 编译程序的组成

编译程序的组成如图 1.4 所示。

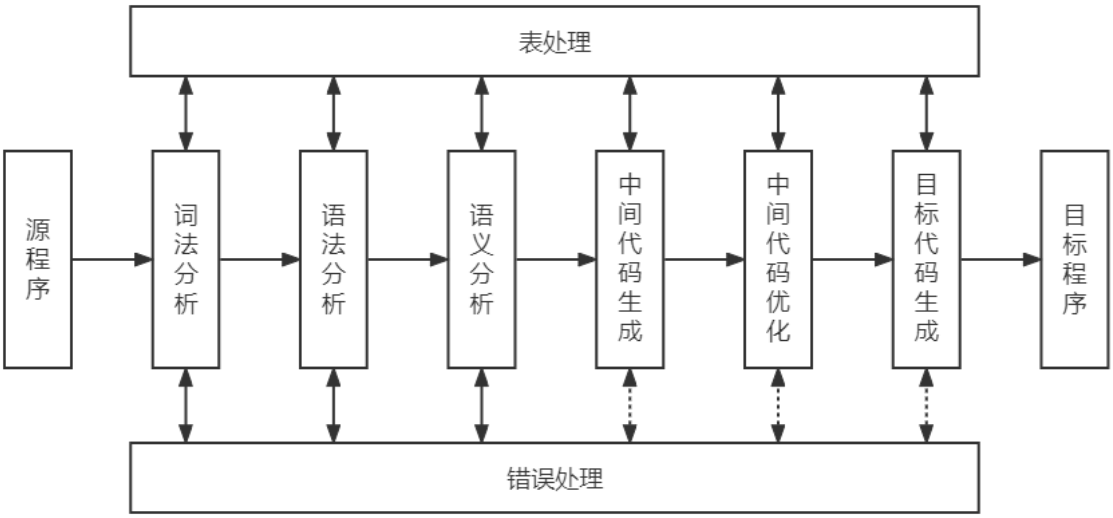


图1.4 编译程序的组成

词法分析：识别由字符组成的高级语言程序中的单词，并将其转化成一种内部表示（token）的形式，同时检查是否存在词法错误。

语法分析：根据语言定义的语法规则来验证程序中是否存在语法错误。

语义分析：检查源程序有无语义错误，为代码生成阶段收集类型信息。

中间代码生成：将源程序转换成一种称为中间代码的内部表示形式，便于优化和移植。

中间代码优化：变换或改造中间代码，使生成的目标代码更为高效，即节省时间和空间。和程序算法的高效无关，更多的是针对于程序具体运行时的内部优化，尤其针对有特殊要求的编译器。

目标代码生成：将中间代码变换为特定机器上的机器指令代码或汇编指令代码。

错误处理：当编译阶段有错误出现时，由相应的错误处理模块给出解决方案，使得编译器能够继续进行下去，并为用户提供更多的参考信息。

表处理：为了合理地管理（构造、组织、查找、更新等）表格（token 序列、符号表、类型信息表、语法信息表等），设立一些专门子程序称为表格管理程序。

1.4 编译程序的设计

编译程序的分遍：所谓“遍”就是对源程序或源程序的中间表示形式从头到尾扫描一次，并作加工处理，生成新的中间结果或目标程序。分遍就是对源程序或源程序的中间表示形式从头到尾扫描几次。

2 词法分析

2.1 词法分析概述

2.1.1 词法分析的基本功能

词法分析程序是编译程序的一部分，是整个编译过程的第一步工作。

词法分析器读取源程序的字符序列，逐个拼出单词并构造相应的内部表示，同时检查源程序中的词法错误。它的核心作用即为将字符序列转化为计算机内部表示。

单词：是指语言中具有独立含义的最小的语义单位。

2.1.2 词法分析器的接口

词法分析器有两类，一类是仅作为语法分析的子程序，如图 2.1(a) 所示；另一类是作为编译器的独立一遍处理器，如图 2.1(b) 所示。

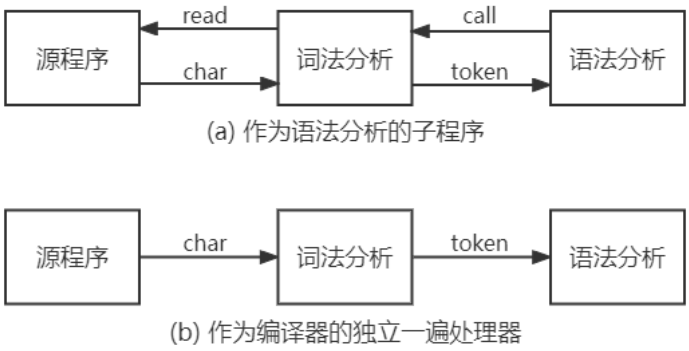


图2.1 词法分析器的接口

2.1.3 单词类型的划分

常用程序设计语言的单词可以分为以下几类：

1. 标识符：用来标识程序中各个对象的名称。它们由用户定义，用来表示变量名、常量名、数组名和函数名等。
2. 保留字：保留字一般是由语言系统自身定义的，通常是由字母组成的字符串。如 C 语言中的 `int`、`if`、`for`、`do` 等等。这些字在语言中具有固定的意义，是编译程序识别各类语法成分的依据。
3. 常量：主要包括整数常量、实数常量、字符常量、字符串常量等。
4. 特殊符号：包括运算符、界限符和控制符（格式符）。
   （1）运算符：表示程序中算术运算、逻辑运算、字符运算、赋值运算的确定的字符或字符串。如 `+`、`-`、`*`、`/`、`<`、`>=` 等。
   （2）界限符：在语言中是作为语法上的分界符号使用的，如逗号、分号、单引号等。
   （3）控制符：主要用于控制语言的格式，如回车、空格等。

注意：`true` 和 `false` 是标识符，不是常量。

2.1.4 如何实现词法分析器

1. 明确要分析的问题。
2. 利用形式化方法描述各类单词的词法规则。
3. 设计词法分析算法。

2.2 基本概念

1. 字母表（alphabet）

字母表是元素的非空有穷集合。字母表中的一个元素称为该字母表的一个**字母**（letter），也可叫做**符号**（symbol）或者**字符**（character）。字母表有时也称为**符号表**，通常用 Σ 表示。

2. 符号串

由字母表中的符号组成的任何有穷序列称为字母表上的**符号串**，一般用 $\alpha, \beta, \cdots, x, y, z$ 表示。

符号串中字符的个数称为符号串长度，用 $|\alpha|$ 表示符号串 α 的长度。

长度为 0 的符号串称为**空串**，用 ε 表示，$|\varepsilon| = 0$。对任一字母表 Σ，都有 ε 是 Σ 上的符号串。

3. 符号串的连接

设 α 和 β 均是字母表 Σ 上的符号串，α 和 β 的连接是把 β 的所有符号顺次地接在 α 的所有符号之后所得到的符号串，记为 $\alpha\beta$。

$|\alpha\beta| = |\alpha| + |\beta|$。

特别地，$\varepsilon\alpha = \alpha\varepsilon = \alpha$。

符号串的连接不满足交换律，$\alpha\beta \neq \beta\alpha$。

4. 符号串的方幂

设 α 是字母表 Σ 上的符号串，把 α 自身连接 n 次得到的符号串称作符号串 α 的 n 次幂，记作 α^n。

$$
\begin{aligned}
&\alpha^0 = \varepsilon \\
&\alpha^1 = \alpha \\
&\alpha^2 = \alpha\alpha \\
&\alpha^3 = \alpha^2\alpha = \alpha\alpha^2 = \alpha\alpha\alpha \\
&\cdots \\
&\alpha^n = \underbrace{\alpha\alpha\cdots\alpha}_{n个\alpha}
\end{aligned}
$$

5. 符号串集合

若集合 A 中的所有元素都是某字母表 Σ 上的符号串，则称 A 为该字母表上的符号串集合。

6. 符号串集合的乘积

设 A、B 是两个符号串集合，AB 表示 A 与 B 的乘积，具体定义为：

$$AB = \{xy \,|\, x \in A \text{且} y \in B\}$$

特别有：
(1) $\varnothing A = A\varnothing = \varnothing$，其中 $\varnothing$ 表示空集。
(2) $\{\varepsilon\}A = A\{\varepsilon\} = A$。

7. 符号串集合的方幂

设 A 为符号串的集合，则称 A^i 为符号串集合 A 的方幂，具体定义如下：

$$
\begin{aligned}
&A^0 = \{\varepsilon\} \\
&A^1 = A \\
&A^2 = AA \\
&\cdots \\
&A^n = \underbrace{AA\cdots A}_{n个A}
\end{aligned}
$$

8. 符号串集合的正闭包

设 A 是符号串集合，则称 A^+ 是符号串集合 A 的正闭包，其中

$$A^+ = A^1 \cup A^2 \cup \cdots \cup A^n \cup \cdots$$

9. 符号串集合的星闭包

设 A 是符号串集合，则称 A^* 是符号串集合 A 的星闭包，其中

$$A^* = A^0 \cup A^1 \cup A^2 \cup \cdots \cup A^n \cup \cdots = A^0 \cup A^+$$

2.3 正则表达式

2.3.1 正则表达式的定义

正则表达式是用来描述正则集的一种代数表达式，也称为正规表达式。

正则表达式的形式：用事先定义好的一些特定字符，以及对这些特定字符进行组合运算，形成的一个“规则字符串”。

正则表达式的作用：定义一类字符串的一种过滤逻辑。

设 Σ 为有限字母表，在 Σ 上的正则表达式可递归定义如下：

1. ε 和 $\varnothing$ 是 Σ 上的正则表达式；
2. 对任何 $a \in \Sigma$，a 是 Σ 上的正则表达式；
3. 若 r，s 都是正则表达式，则 (r)、$r|s$、$r \cdot s$、r^*、r^+ 也是正则表达式；
4. 有限次使用上述三条规则构成的表达式，称为 Σ 上的正则表达式。

正则表达式的**语义函数**：给正则表达式赋予一种语义解释的函数。不同的语义解释会使得正则表达式具有不同的语义，其操作结果也会不同。

单词的本质是字符串，在词法分析中，为了用正则表达式描述单词，我们用语义函数为正则表达式和字符串集合建立一种映射关系，使得正则表达式的语义解释被描述成字符串的形式。

在词法分析中，正则表达式 e 根据语义函数解释所得到的符号串集合称为正则表达式 e 的**正则集**，记为 $L(e)$。

若设 e、e_1、e_2 为 Σ 上的正则表达式，则 e 所对应的正则集 $L(e)$ 取值如下：

1. 当 $e = \varnothing$ 时，$L(e) = \varnothing$。
2. 当 $e = \varepsilon$ 时，$L(e) = \{\varepsilon\}$。
3. 对于 Σ 中一个字符 a，若 $e = a$，则 $L(e) = \{a\}$。
4. 当 $e = e_1 \cdot e_2$ 时，$L(e) = L(e_1)L(e_2)$。
5. 当 $e = e_1|e_2$ 时，$L(e) = L(e_1) \cup L(e_2)$。
6. $L((e)) = L(e)$。
7. $L(e^*) = L(e)^*$。
8. $L(e^+) = L(e)^+$。

正则表达式中四种运算的作用：

1. () 运算：不改变正则集本身，主要是用于确定运算优先关系。
2. | 运算：或运算。
3. · 运算：连接运算。
4. * 运算：r^* 表示对正则表达式 r 所描述的文本进行 0 到若干次循环连接。

2.3.2 正则表达式的性质

1. 运算优先级：从高到低依次为 ()、$^+$、*、·、|
2. | 运算的交换律：$A|B = B|A$
3. | 运算的结合律：$A|(B|C) = (A|B)|C$
4. 连接的结合律：$A(BC) = (AB)C$
5. 连接的分配律：$A(B|C) = AB|AC$，$(A|B)C = AC|BC$
6. 幂的等价性：$A^{**} = A^*$
7. 同一律：$A\varepsilon = \varepsilon A = A$

2.3.3 用正则表达式描述单词

标识符：$L(L|D)^*$，其中 $L = \mathrm{A}|\mathrm{B}|\cdots|\mathrm{Z}|\mathrm{a}|\mathrm{b}|\cdots|\mathrm{z}|\_$，$D = 0|1|\cdots|9$

常数：

1. 整数：$(+|-|\varepsilon)(D_1D^*)|0$，其中 $D_1 = 1|2|\cdots|9$
2. 实数：$((+|-|\varepsilon)(D_1D^*)|0).D^+$

特殊符号：

1. 保留字：$\mathrm{while}|\mathrm{if}|\mathrm{for}|\cdots$
2. 运算符：$+|-|*|\cdots$
3. 分界符：$\{|\}|;|\cdots$
4. 控制符：$\backslash\mathrm{t}|\backslash 0|\cdots$

2.3.4 正则表达式的局限性

1. 正则表达式不能用于描述重复串。
2. 正则表达式不能用于描述配对或嵌套的结构。

2.4 确定有限自动机

2.4.1 确定有限自动机的定义

确定有限自动机（Deterministic Finite Automata，DFA）为一个五元组 $M=(S,\Sigma,S_0,f,Z)$，其中：

1. S 是一个有穷状态集，它的每个元素称为一个状态。
2. Σ 是一个有穷字母表，它的每个元素称为一个输入字符。
3. $S_0\in S$，是唯一的一个初始状态（开始状态）。
4. f 是状态转换函数，$f:S\times\Sigma\to S$，并且是单值函数。$f(S_i,a)=S_k$ 表示当前状态为 S_i，遇到输入字符 a 时，自动机将唯一地转换到状态 S_k，称 S_k 为 S_i 的一个后继状态。
5. $Z\subseteq S$ 是终止状态集（可接受状态集、结束状态集），其中的每个元素称为终止状态（可接受状态、结束状态）。Z 可空。

DFA 具有确定性：

1. 初始状态唯一。
2. 状态转换函数 f 是一个单值函数，对于任何状态 $s\in S$ 和输入符号 $a\in\Sigma$，$f(s,a)$ 唯一地确定了下一个状态。
3. 没有输入为 ε 的空边，即不接受任何没有输入就进行状态转换的情况。

2.4.2 确定有限自动机的表示

2.4.2.1 状态转换图

状态转换图：用有向图表示自动机，比较直观，易于理解。

1. 结点表示状态。
   （1）非终止状态：单圆圈围住的状态标识。
   （2）终止状态：双圆圈围住的状态标识。
   （3）开始状态：由一个箭头指向的状态结点。
2. 有向边表示状态转换函数。若 $f(S_i,a)=S_k$，则由表示 S_i 的状态结点到表示 S_k 的状态结点发出一条标识为 a 的有向边。

2.4.2.2 状态转换矩阵

状态转换矩阵：用二维数组描述 DFA，易于程序实现。

行标为状态，列标为 Σ 上的所有输入字符，矩阵元素表示自动机的状态转换函数。

一般约定：第一行表示初始状态 S_0，或在初始状态的右上角标注“+”；右上角标有“*”或“-”的状态为终止状态。

2.4.2.3 陷阱状态

在图 2.2 所示的 DFA 中，状态 4 是非终止状态，而一旦到达状态 4 就再也无法离开，导致识别失败。这样的状态称为陷阱状态，也称为错误状态。

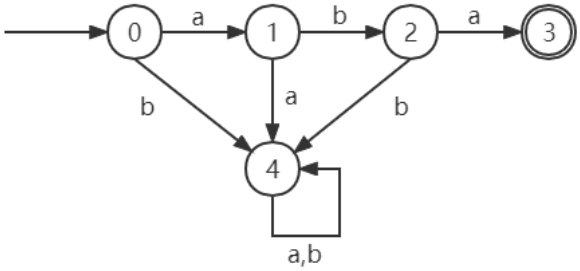


图2.2 陷阱状态示例

陷阱状态在字符串的识别过程中没有意义，为了方便起见，在状态转换图中去掉这个陷阱状态以及连到这个状态上的边；在状态转换矩阵中去掉陷阱状态，将这个状态相应的矩阵元素置空。在状态转换过程中，如果在状态转换图中没有对应的边，或者在状态转换矩阵中目的状态的矩阵元素为空，就表示进入错误状态，需要进行错误处理。

2.4.3 确定有限自动机接受的字符串

对于 Σ 中的任何字符串 $a_1a_2\cdots a_n$，若在 DFA M 中存在一条从初始结点到某一终止结点的路径，且这条路径上所有弧上的标记符连接成的字符串等于 $a_1a_2\cdots a_n$，则称该字符串可为 DFA M 所接受（识别）。

若 DFA M 的初始状态同时又是终止状态，则空字符串 ε 可为 DFA M 所接受。

DFA M 所能接受的字符串的全体称为DFA M 接受（识别）的语言，记为 $L(M)$。

对于两个 DFA M_1 和 M_2，若 $L(M_1)=L(M_2)$，则称 M_1 和 M_2 等价。

2.4.4 确定有限自动机描述单词

设 $L=\mathrm{A}|\mathrm{B}|\cdots|\mathrm{Z}|\mathrm{a}|\mathrm{b}|\cdots|\mathrm{z}|\_$，$D=0|1|\cdots|9$，$D_1=1|2|\cdots|9$，用这三个符号就可以画出各种单词的确定有限自动机。

标识符：

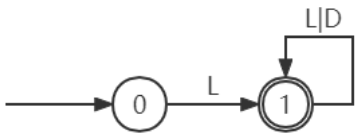


图2.3 DFA描述标识符

无符号整数：

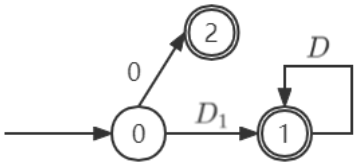


图2.4 DFA描述无符号整数

带符号整数：

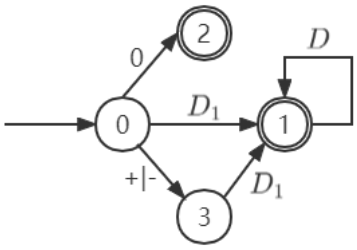


图2.5 DFA描述带符号整数

实数：

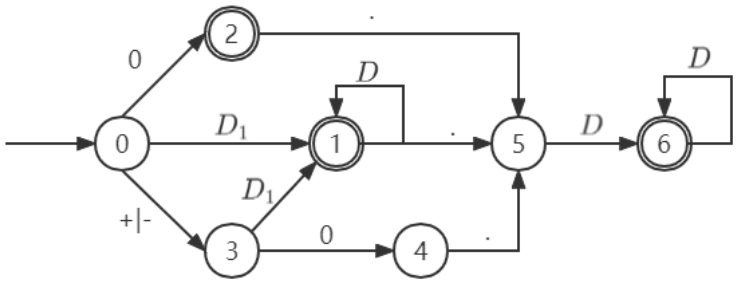


图2.6 DFA描述实数

特殊字符：

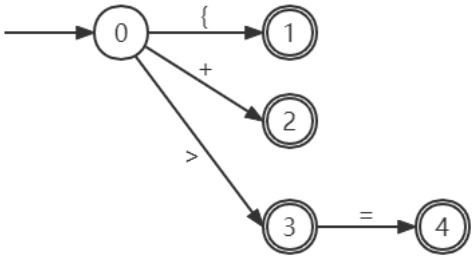


图2.7 DFA描述特殊字符

保留字：

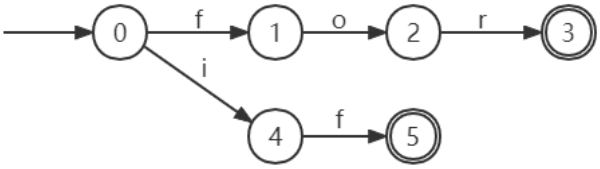


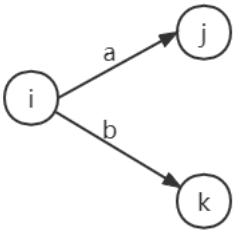
图2.8 DFA描述保留字

2.4.5 确定有限自动机的实现

2.4.5.1 直接转换法

每个状态对应一个带标号的`switch`语句，转向边对应`goto`语句。

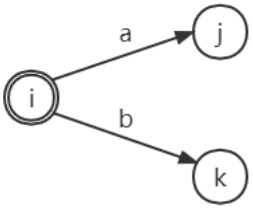
非终止状态对应的`switch`语句如图 2.9 所示。




```
Li:switch(currentChar)
{
    case 'a':
        goto Lj;
        break;
    case 'b':
        goto Lk;
        break;
    default:
        Error();
        break;
}
```

图2.9 非终止状态对应的switch语句

终止状态对应的`switch`语句如图 2.10 所示。




```
Li:switch(currentChar)
{
    case 'a':
        goto Lj;
        break;
    case 'b':
        goto Lk;
        break;
    case 'EOF':
        Accept();
        break;
    default:
        Error();
        break;
}
```

图2.10 终止状态对应的switch语句

直接转换法的特点：程序的结构一样，但程序随着状态图的不同而改变。

2.4.5.2 状态转换矩阵

1. 当前状态`state`置为初始状态。
2. 读取一个字符，保存到变量`curChar`中。
3. 如果`curChar != 'EOF'`并且`T(state, curChar) != error`，则循环执行以下操作：
   （1）将当前状态转为新的状态：`state = T(state, curChar)`。
   （2）读下一字符，保存到变量`curChar`中。
4. 如果`curChar == EOF`并且当前状态属于终止状态，则接受当前字符串，程序结束；否则报错。

基于状态转换矩阵方法的特点：程序不变，只需改变状态矩阵内容。但状态多时占用存储空间大。

2.4.6 确定有限自动机的化简

设 S_1 和 S_2 是 DFA M 的两个状态，如果对任意输入的符号串 x，从 S_1 和 S_2 出发，总是都到达接受状态或拒绝状态中，则称 S_1 和 S_2 是等价的。

两个状态等价的条件：

1. 一致性条件：S_1 和 S_2 同时为可接受状态或不可接受状态。终止状态和非终止状态是不等价的。
2. 蔓延性条件：S_1 和 S_2 对所有输入符号必须都要转换到等价状态中。

设 s 是 DFA M 的一个状态，若从开始状态没有到 s 的通路，或 s 到任意终止状态无通路，则称 s 为 M 的无关状态。

如果 DFA M 没有无关状态和等价状态，则称 M 为最小（最简）自动机。

结论：任一 DFA 都可以化为最简自动机。即任一 DFA M 都存在最简自动机 M'，使得 $L(M)=L(M')$。

DFA 化简方法——状态分离法：

1. 初始时，终止状态为一组，非终止状态为一组。

2. 对每一组进行分离。若每组中的元素接收 Σ 中的任意字符会映射到不同的组，则表示它们不等价，就可以分离出来建立新的组。
3. 重复 2，直到没有新的组产生，此时每组中的状态都是等价状态。

2.5 非确定有限自动机

2.5.1 非确定有限自动机的定义

非确定有限自动机（Nondeterministic Finite Automata，NFA）为一个五元组 $A=(S,\Sigma,S_0,f,Z)$，其中：

1. S 是一个有穷状态集，它的每个元素称为一个状态。
2. Σ 是一个有穷字母表，它的每个元素称为一个输入字符。
3. $S_0\subseteq S$，是 NFA 的初始状态集。
4. f 是状态转换函数，不要求是单值函数。$f:S\times(\Sigma\cup\{\varepsilon\})\to 2^S$，$f(S_i,a)=\{S_{j_1},S_{j_2},\cdots\}$。
5. $Z\subseteq S$ 是终止状态集（可接受状态集、结束状态集），其中的每个元素称为终止状态（可接受状态、结束状态）。Z 可空。

与 DFA 相比，NFA 的非确定性体现在以下 3 个方面：

1. 状态转换函数可为多值函数，一个状态接受同一个输入字符可以转向多个不同后继状态。
2. 允许有多个开始状态。
3. 允许有空边，即在没有任何输入的情况下允许进行状态转换。

设 A 是一个 NFA，定义 A 接受（识别）的语言 $L(A)$ 为：从任意初始状态到任意终止状态所接收的字符串的集合。

NFA 所能接受的串与 DFA 的定义是相同的，但 NFA 实现起来很困难。

2.5.2 非确定有限自动机的确定化

定义：设 A_1 和 A_2 是同一个字母表 Σ 上的自动机，如果有 $L(A_1)=L(A_2)$，则称 A_1 和 A_2 等价。

定理：对于任意一个非确定有限自动机 A，都存在一个确定有限自动机 A'，使得 $L(A)=L(A')$。

由非确定有限自动机构造出与其等价的确定有限自动机称为非确定有限自动机的确定化。

设 J 是 NFA A 状态集的子集，定义 J 的 ε 闭包 ε-$\mathrm{CLOSURE}(J)$ 为：

1. 若 $q\in J$，则 $q\in\varepsilon$-$\mathrm{CLOSURE}(J)$。
2. 若 $q\in\varepsilon$-$\mathrm{CLOSURE}(J)$，则从 q 出发经任意条 ε 边到达的任何状态 q' 都属于 ε-$\mathrm{CLOSURE}(J)$。

设 $I=\{S_1,S_2,\cdots,S_m\}$ 是 NFA 状态集的子集，对于任意的输入 $a\in\Sigma$，定义状态集 I 经过输入 a 的转换状态集合为 $I_a=\varepsilon$-$\mathrm{CLOSURE}(J)$，其中 $J=f(S_1,a)\cup f(S_2,a)\cup\cdots\cup f(S_m,a)$。

用子集法将 NFA 转化为 DFA 的思想：让 DFA 的某一个状态去记录 NFA 读入一个输入符号后可能达到的一组状态。

NFA 的确定化算法：

1. 令 $I_0=\varepsilon$-$\mathrm{CLOSURE}(S_0)$ 作为 DFA 的初始状态，其中 S_0 为 NFA 的初始状态集。
2. 若 DFA 中的每个状态都经过本步骤处理过，则转步骤 3；否则任选一个未经本步骤处理的 DFA 状态 S_i，对每一个 $a\in\Sigma$，进行下述处理：
   （1）计算 $S_j=S_{ia}$。
   （2）若 $S_j\neq\varnothing$，则令 $f(S_i,a)=S_j$。若 S_j 不为当前 DFA 的状态，则将其作为 DFA 的一个状态，转步骤 2。
3. 若 $S'=\{S_1,\cdots,S_n\}$ 是 DFA 的一个状态，且存在一个 $S_k\in S'$ 是 NFA 的终止状态，则令 S' 为 DFA 的终止状态。

2.6 正则表达式和自动机的相互转化

2.6.1 正则表达式向非确定有限自动机的转化

定理：对 Σ 上的每一个正则表达式 R，存在一个 Σ 上的非确定有限自动机 M，使得 $L(M)=L(R)$。

构造方法：

1. 构造初始状态 S 和终止状态 Z，由 S 发出指向 Z 的有向弧并标上正则表达式 R。
2. 反复利用图 2.11 所示的替换规则对正则表达式 R 依次进行分解，直至状态转换图中所有有向弧上标记的符号都是字母表 Σ 上的元素或 ε 为止。

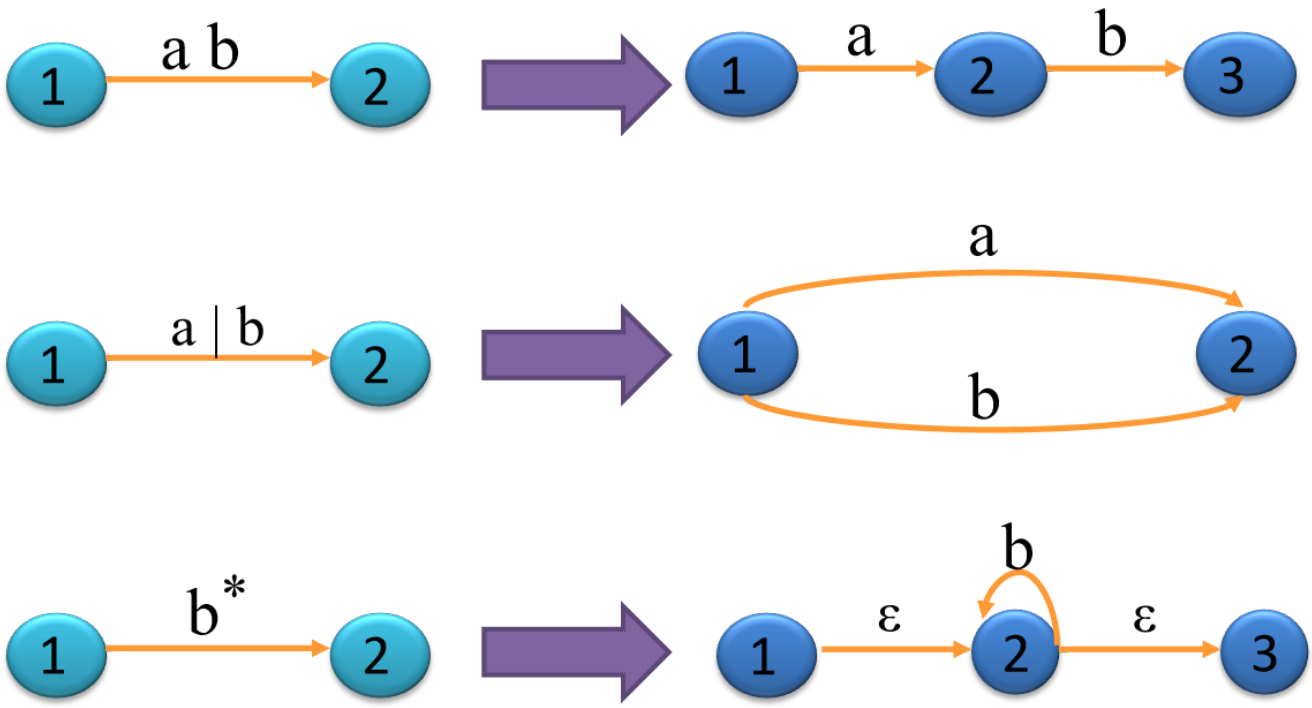


图2.11 正则表达式转化为NFA的替换规则

2.6.2 自动机向正则表达式的转换

定理：对于字母表上 Σ 上的非确定有限自动机 M，存在 Σ 上的正则表达式 R，使得 $L(R) = L(M)$。

注意：这里 M 可以是 DFA，DFA 是 NFA 的一种特例。

构造方法：

1. 在 M 的状态转换图中加入两个结点，一个是唯一的开始状态结点 S，另一个是唯一的终止状态结点 Z。从 S 出发用标有 ε 的有向弧连接到 M 的所有初始状态结点上，从 M 的所有终止状态结点用标有 ε 的有向弧连接到 Z 结点。
2. 反复利用图 2.12 的替换规则进行替换，直到状态转换图中只剩下结点 S 和 Z，在 S 指向 Z 的有向弧上所标记的正则表达式就是所求的结果。

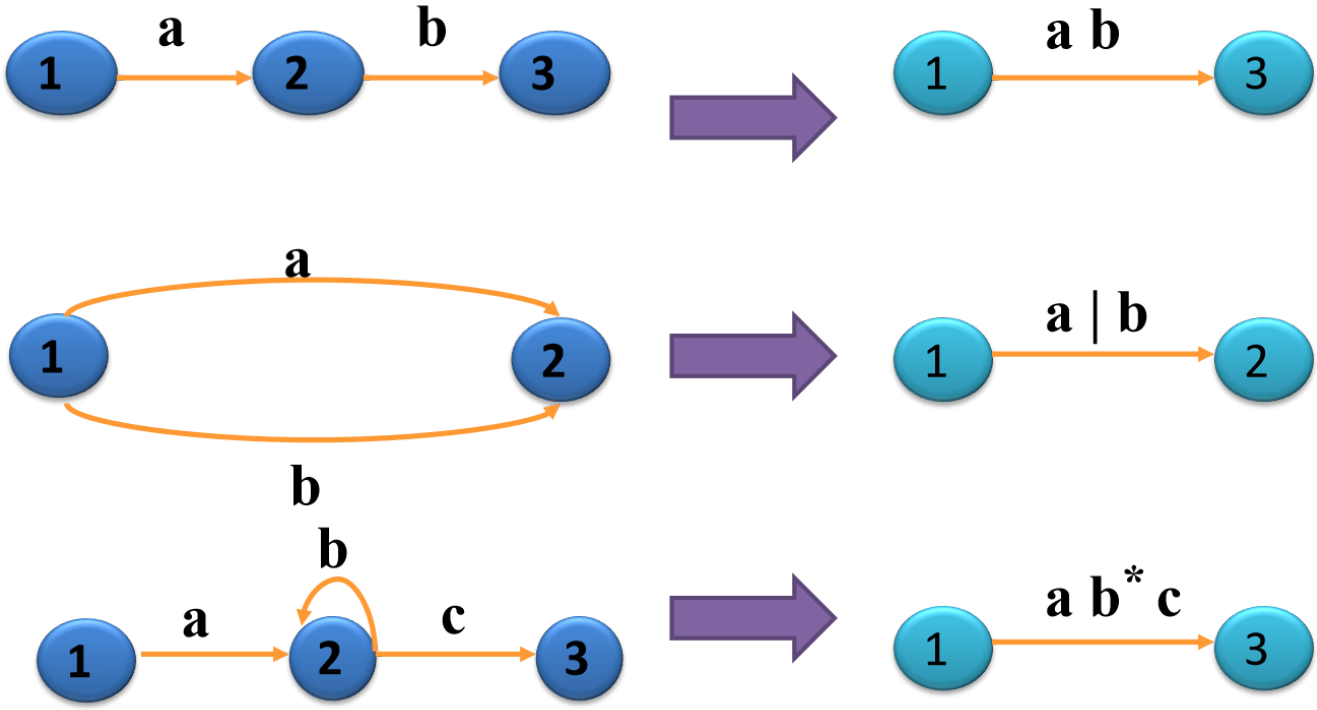


图2.12 自动机转化为正则表达式的替换规则

2.7 词法分析器的实现

2.7.1 词法分析器的设计

设计步骤：

1. 确定词法分析器的接口。
2. 确定单词的结构。
3. 给出单词的描述。
4. 设计算法。

2.7.2 单词的结构

单词的结构至少包括两部分内容：单词的类型（语法信息）、单词的内容（语义信息）。

一种可行的 token 样例：

1. token 有两个字段，分别是类型和内容。
2. 建立 3 张表：标识符索引表，常量索引表，保留字、特殊符号表。标识符索引表和常量索引表由位置编号和内容组成，保留字、特殊符号表由类型编号和内容组成。
3. 如果单词为标识符，则类型填 1，内容填该标识符在标识符索引表中的位置；如果单词为常量，则类型填 2，内容填该常量在常量索引表中的位置；如果单词为保留字或特殊符号，则类型填该保留字或特殊符号所对应的数字，内容为空。

2.7.3 词法分析器的具体实现

词法分析器的工作过程如图 2.13 所示。

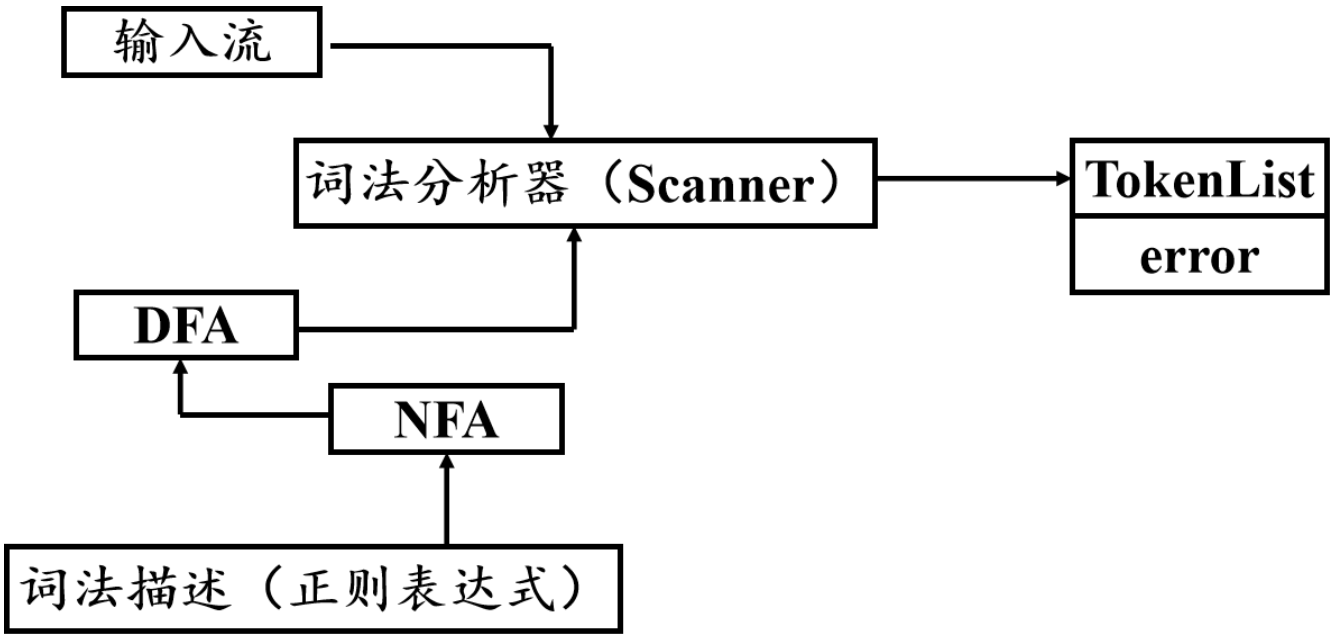


图2.13 词法分析器的工作过程

根据词法分析器的工作流程，可以将词法分析器的实现分为两部分：

1. 单词的 DFA 描述及实现。
2. 针对不同的单词生成对应的 token。

构造识别单词的有限自动机的方法如下：

1. 根据构成规则对程序语言的单词按类构造出相应的状态转换图，或将各类单词的正则表达式转换成相应的有限自动机。
2. 合并各类单词的状态转换图，构成一个能识别语言所有单词的 DFA。合并方法为：
   （1）将各类单词的状态转换图的初始状态合并为一个唯一的初始状态；

（2）化简调整状态冲突，对冲突状态重新编号；
（3）如果有必要，增加出错状态。

token 的生成：

- 标识符的 token 生成：首先查找保留字、特殊符号表，判断这个字符是否为保留字，若是则生成保留字对应的 token；若不是则查找标识符索引表，确定其在标识符索引表中的位置，生成该标识符对应的 token。
- 常量的 token 生成：查找常量索引表，确定其在常量索引表中的位置，生成该常量对应的 token。
- 特殊符号的 token 生成：查保留字、特殊符号表，确定这个特殊符号在表中的类型编码，生成该特殊符号对应的 token。

2.7.4 注意事项

1. 名字的实现方式

识别保留字的实现方法可分为两大类：一类是设置保留字表，另一类是用自动机单独来识别。

设置保留字表的主要思想是事先构造好保留字表，在进行词法分析时，把保留字也当做一般标识符来识别，然后查保留字表，若有，则把它作为保留字来处理；若没有，则按一般标识符在处理。

用自动机单独来识别保留字的主要思想是在自动机中加入识别各个保留字的状态，即把保留字和一般标识符分开来识别而不统一识别。

两种方式的比较：自动机单独识别的优点是速度快，但它使得自动机的状态数随着保留字个数的增多而急剧增加；保留字表的好处是节省空间。

2. 复合单词的识别

有一类单词是由两个或两个以上的符号组成的，这类单词的前缀也可以是一个独立的单词。例如 `++`，其前缀 `+` 也可以是一个独立的单词。

在处理此类单词时，读取到前缀单词后不能立即断定，还需要读取后续字符进一步判断。

3. 数的转换

词法分析程序应该把数字字符串转换成数。

4. 向前看若干个字符

在有些语言里，为了识别出一个单词需要向前看若干个字符。例如 `5+++a`，应该将后两个 `+` 号合在一起看做 `++` 号，因为 `5` 是一个常量，不能进行 `++` 运算。

5. 控制字符的处理

控制字符包括空格、Tab、换行符等。这些字符占用很大的空间，而且一般来说，它们只有词法意义而没有语法和语义上的意义。

若控制字符只是用来分隔源程序中不同的单词，如空格和 Tab 等，则在词法分析过程中可将它们直接删除。

换行符本身虽然没有实际意义，但对于错误处理起着重要的作用（如确定行号），所以换行符不能直接删除。

6. 注释的处理

注释没有任何语法和语义上的意义，因此在进行词法分析时可以直接将注释删除，而不必生成其 token。

3 语法分析

3.1 文法

3.1.1 文法的定义

一个文法 G 是一个四元组：$G = (V_{\mathrm{N}}, V_{\mathrm{T}}, S, P)$。其中：

- V_{N} 是一个非空的有限集合，它的每个元素称为非终极符号、非终极符或中间符，一般用大写字母表示。
- V_{T} 是一个非空的有限集合，它的每个元素称为终极符号或终极符，一般用小写字母表示。终极符号是一个语言不可再分的基本符号。
- S 是特殊的非终极符，称为文法的开始符号，$S \in V_{\mathrm{N}}$。
- P 是产生式的有限集合。

产生式也称为产生规则，是按照一定格式书写的定义语法范畴的文法规则。产生式的形式为：$\alpha \to \beta$。其中：

1. α 称为产生式的左部（头），β 称为产生式的右部（体），$\alpha, \beta \in (V_\mathrm{T} \cup V_\mathrm{N})^*$。
2. $\to$ 读作 “定义为” 或 “由……组成”。
3. 开始符号 S 必须至少在某个产生式的左部出现一次。

为了书写方便，若干个左部相同的产生式，如 $P \to \alpha_1, P \to \alpha_2, \cdots, P \to \alpha_n$，可合并为一个，缩写为 $P \to \alpha_1 | \alpha_2 | \cdots | \alpha_n$。其中，每个 a_i 称为 P 的一个候选式，符号 $|$ 读作 “或”。

3.1.2 文法的分类

0 型文法：也称为短语文法，其产生式的形式为：$\alpha \to \beta$，其中 $\alpha \in (V_\mathrm{T} \cup V_\mathrm{N})^+$，$\beta \in (V_\mathrm{T} \cup V_\mathrm{N})^*$，并且 α 至少含一个非终极符。

1 型文法：也称为上下文相关文法。它是 0 型文法的特例，要求 $|\alpha| \leqslant |\beta|$（$S \to \alpha$ 例外，但 S 不得出现在产生式右部）。其产生式的形式为：$\alpha A \beta \to \alpha \gamma \beta$，其中 $A \in V_\mathrm{N}$，$\alpha, \beta, \gamma \in (V_\mathrm{T} \cup V_\mathrm{N})^*$。

2 型文法：也称为上下文无关文法。它是 1 型文法的特例，要求产生式左部是一个非终极符。其产生式的形式为：$A \to \alpha$，其中 $A \in V_\mathrm{N}$，$\alpha \in (V_\mathrm{T} \cup V_\mathrm{N})^*$。

3 型文法：也称为正则文法、线性文法、正规文法。它是 2 型文法的特例，要求产生式的右部至多有两个符号，而且具有下面形式之一：$A \to a$ 或 $A \to aB$，其中 $A, B \in V_\mathrm{N}$，$a \in V_\mathrm{T}$。

3 型文法的描述能力与自动机、正则表达式等价。

2 型文法（上下文无关文法，Context-Free Grammer，CFG）定义为四元组 $(V_\mathrm{N}, V_\mathrm{T}, S, P)$，其中：

- V_N 是非空有限的非终极符集合。
- V_T 是非空有限的终极符集合。
- S 是开始符号，$S \in V_\mathrm{N}$。
- P 是产生式的集合，且产生式具有如下形式：$A \to X_1 X_2 \cdots X_n$，其中 $A \in V_\mathrm{N}$，$X_i \in (V_\mathrm{T} \cup V_\mathrm{N})$，右部可空。

在语法分析中使用的是 2 型文法。

3.1.3 基本概念

如果 $A \to \beta$ 是一个产生式，则有 $\alpha_1 A \alpha_2 \Rightarrow \alpha_1 \beta \alpha_2$，这时称 $\alpha_1 \beta \alpha_2$ 是由 $\alpha_1 A \alpha_2$ **直接推导**的，又称 $\alpha_1 \beta \alpha_2$ **直接规约**到 $\alpha_1 A \alpha_2$。其中 $\Rightarrow$ 表示一步推导，它的含义是，使用一条规则，代替左边的符号，产生右端的符号串。

$A \Rightarrow^+ \beta$ 表示 A 通过一步或多步可推导出 β，$A \Rightarrow^* \beta$ 表示 A 通过零步或多步可推导出 β。

设有文法 G，如果有 $S \Rightarrow^* \beta$，则称符号串 β 为 G 的**句型**。用 $\mathrm{SF}(G)$ 表示文法 G 的所有句型的集合。

设 β 为文法 G 的一个句型，且 β 只包含终极符，则称 β 为 G 的**句子**。

文法 G 的所有句子的集合称为文法 G 所定义的**语言**，记为 $L(G)$。$L(G) = \{u \,|\, S \Rightarrow^+ u, u \in V_\mathrm{T}^*\}$。

设 S 是文法的开始字符，如果有 $S \Rightarrow^* \alpha_1 A \alpha_2$，$A \Rightarrow^+ \beta$，则称 β 是句型 $\alpha_1 \beta \alpha_2$ 的一个**短语**。

设 S 是文法的开始字符，如果有 $S \Rightarrow^* \alpha_1 A \alpha_2$，$A \Rightarrow \beta$，则称 β 是句型 $\alpha_1 \beta \alpha_2$ 的一个**简单短语**。

句型中的最左简单短语称为**句柄**。

若有 $E \to E\alpha$ 形式的产生式，则称 E 是**直接左递归**。若有 $E \to \alpha E$ 形式的产生式，则称 E 是**直接右递归**。若有 $E \Rightarrow^+ E\alpha$，则称 E 是**左递归**。若有 $E \Rightarrow^+ \alpha E$，则称 E 是**右递归**。若有 $E \Rightarrow^+ \alpha_1 E \alpha_2$，则称 E 是递归。

如果进行推导时选择的是句型中的最左（右）非终极符，则称这种推导为**最左（右）推导**，用符号 $\Rightarrow_\mathrm{lm}$（$\Rightarrow_\mathrm{rm}$）表示最左（右）推导。

用最左推导方式导出的句型，称为**左句型**。用最右推导方式导出的句型，称为**右句型**（规范句型）。

结论：每个句子都有相应的最右和最左推导，但对句型此结论不成立。

3.1.4 语法分析树

语法分析树（简称分析树）用来描述句型的结构，是句型推导的一种树形表示。

设 G 是给定的语法，称满足下列条件的树为 G 的一棵语法分析树：

1. 树的每个结点都标有 G 的一个文法符号，且根结点标有初始符 S，非叶结点标有非终极符，叶结点标有终极符或非终极符或 ε。
2. 如果一个非叶节点 A 按从左到右顺序有 n 个儿子节点 $B_1, B_2, \cdots, B_n$，则 $A \to B_1 B_2 \cdots B_n$ 一定是 G 的一个产生式。

用 $\Rightarrow$ 符号进行的推导叫做线性推导，用语法分析树进行的推导叫做树型推导。线性推导与树型推导的不同：

1. 线性推导指明了推导的顺序，而树型推导没有指明推导的顺序。
2. 句型一般只有一棵分析树（如果无二义性），而线性推导可能有很多种。

语法分析树的作用：

1. 语法分析树反映出推导的过程，每一步结点的生长过程都可以对应到一步推导。
2. 语法分析树反映出串的语法结构。

若某语法分析树 T 中的某一结点 A 和它的所有分支组成树 T'，则称 T' 是 T 的一棵**子树**。

若某一结点 A 只有一层子结点，则结点 A 和与其子结点构成的子树称为**简单子树**。

语法分析树和语法概念的关系：

1. 语法树的所有叶节点对应语法符号从左到右连接起来组成的符号串是一个句型。
2. 语法树的子树所有叶节点对应语法符号从左到右连接起来组成的符号串是一个短语。
3. 简单子树所有叶节点对应语法符号从左到右连接起来组成的符号串是一个简单短语。
4. 最左简单子树叶节点对应语法符号从左到右连接起来组成的符号串是句柄。

3.1.5 二义性文法

对一个文法 G，如果至少存在一个句子有两棵（或两棵以上）不同的语法分析树，则称该句子是二义性的。包含有二义性句子的文法称为**二义性文法**，否则，该文法是无二义性的。

如果一个文法的某个句型有两种不同的最左（右）推导，则称该文法为二义性文法。

目前，不存在一个一般性的方法来判断一个现有文法是否为二义性文法。常用的经验性判断：

1. $S \to SS|a$，可以推导出 SSS 串，必有二义性。
2. $S \to S+S$，可以推导出 $S+S+S$，必有二义性。

对二义性文法进行修改，消除其二义性会导致文法的复杂程度和符号数目迅速升高。可以利用二义性文法状态少、分析快的特点，使用二义性文法，对具体问题加入语义规则，约束其二义性即可。

3.1.6 文法等价变换

3.1.6.1 增加拓广产生式

定理：对任一文法 G_1 都可以构造文法 G_2，使得 $L(G_1) = L(G_2)$，且 G_2 的开始符号唯一且不出现于任何产生式的右部。

证明：假设 S 是 G_1 的开始符号，则只要在 G_1 中扩充一条新产生式 $Z \to S$ 即可，其中 Z 是新的开始符号。令扩充后的文法为 G_2，则它显然满足定理的要求。

3.1.6.2 消除空产生式

定理：对任一文法 G_1 都可以构造文法 G_2，使得 $L(G_1) = L(G_2)$，且 G_2 中无空产生式。

根据 G_1，构造 G_2 的过程如下：

1. 令 $\beta = \{A \mid A \to \varepsilon\}$；
2. 递归扩充 β 集合，$\beta = \beta \cup \{A \mid A \Rightarrow^* \alpha, \alpha \in \beta^+\}$；
3. 从 G_1 中删除所有空产生式；
4. 从 G_1 中删除只能导出空串的非终极符对应的产生式；
5. 对于文法中任意产生式 $A \to X_1 X_2 \cdots X_{i-1} X_i X_{i+1} \cdots X_n$：
   (1) 若 $X_i \in V_{\mathrm{T}}$，不做动作；
   (2) 若 $X_i \in V_{\mathrm{N}} - \beta$，不做动作；
   (3) 若 $X_i \in \beta$，补充规则 $A \to X_1 X_2 \cdots X_{i-1} X_{i+1} \cdots X_n$。

【例3.1】 文法 $\mathrm{A} \to \mathrm{aBcD}, \mathrm{B} \to \mathrm{b} \mid \varepsilon, \mathrm{D} \to \mathrm{BB} \mid \mathrm{d}$，消除其中的空产生式。

解： 由步骤 1、2 得 $\beta=\{\mathrm{B,D}\}$，由步骤 5 可以增加下列产生式

$$\mathrm{A}\to\mathrm{acD},\mathrm{A}\to\mathrm{aBc},\mathrm{A}\to\mathrm{ac},\mathrm{D}\to\mathrm{B}$$

由步骤 3，去掉文法中的空产生式，得到等价文法如下

$$\mathrm{A}\to\mathrm{aBcD}\,|\,\mathrm{acD}\,|\,\mathrm{aBc}\,|\,\mathrm{ac},\mathrm{B}\to\mathrm{b},\mathrm{D}\to\mathrm{BB}\,|\,\mathrm{B}\,|\,\mathrm{d}$$

3.1.6.3 消除不可达产生式

如果文法中某一非终极符不出现在该文法的任意句型中，则该非终极符对应的产生式叫做**不可达产生式**。

定理：对任一文法 G_1 都可以构造文法 G_2，使得 $L(G_1)=L(G_2)$，且 G_2 中的每个非终极符必出现在它的某个句型中。

根据 G_1，构造 G_2 的过程如下：

1. 令 $\beta=\{Z\,|\,Z$是文法的开始字符$\}$;
2. 递归扩充 β 直到收敛为止，$\beta=\beta\cup\{B\,|\,A\to xBy\in G_1,B\in V_\mathrm{N},A\in\beta\}$;
3. 若一个产生式左部非终极符 $A\notin\beta$，则删除以 A 为左部的所有产生式。

3.1.6.4 消除特型产生式

左部和右部都仅由一个非终极符构成的产生式称为**特型产生式**。

定理：对任一文法 G_1 都可以构造文法 G_2，使得 $L(G_1)=L(G_2)$，且 G_2 中没有特型产生式。

构造无特型产生式的文法 G_2 的方法如下：

1. 对文法 G_1 中任意非终极符 A，求集合 $\beta_A=\{B\,|\,A\Rightarrow^+B,B\in V_\mathrm{N}\}$;
2. 若 $B\in\beta_A$，且 $B\to\alpha$ 是文法 G_1 的一个非特型产生式，则补充规则 $A\to\alpha$;
3. 去掉文法 G_1 中所有的特型产生式;
4. 去掉新的文法中的不可达产生式。

【例3.2】 文法 $\mathrm{A}\to\mathrm{B}\,|\,\mathrm{D}\,|\,\mathrm{aB},\mathrm{B}\to\mathrm{C}\,|\,\mathrm{b},\mathrm{C}\to\mathrm{c},\mathrm{D}\to\mathrm{B}\,|\,\mathrm{d}$，消除其中的特型产生式。

解： 由步骤 1 得 $\beta_\mathrm{A}=\{\mathrm{B,D,C}\}$，$\beta_\mathrm{B}=\{\mathrm{C}\}$，$\beta_\mathrm{C}=\{\}$，$\beta_\mathrm{D}=\{\mathrm{B,C}\}$。

由步骤 2，在文法中补充规则 $\mathrm{A}\to\mathrm{b}\,|\,\mathrm{d}\,|\,\mathrm{c}$，$\mathrm{B}\to\mathrm{c}$，$\mathrm{D}\to\mathrm{b}\,|\,\mathrm{c}$。

由步骤 3、4，去掉文法中的特型产生式和不可达产生式，得到如下文法：

$$\mathrm{A}\to\mathrm{aB}\,|\,\mathrm{b}\,|\,\mathrm{d}\,|\,\mathrm{c},\mathrm{B}\to\mathrm{b}\,|\,\mathrm{c}$$

3.1.6.5 消除公共前缀

若文法中某个非终极符 A 有如下形式的产生式：$A\to\alpha\beta,A\to\alpha\gamma$，则称产生式有公共前缀。

对于形如 $A\to\alpha\beta_1\,|\,\alpha\beta_2\,|\cdots|\,\alpha\beta_n\,|\,\gamma$（$\gamma$ 表示不以 α 开头的字符串）的产生式，消除公共前缀的方法为：引进非终极符 A'，将产生式替换为

$$\begin{aligned}&A\to\alpha A'\,|\,\gamma\\&A'\to\beta_1\,|\,\beta_2\,|\cdots|\,\beta_n\end{aligned}$$

3.1.6.6 消除左递归

形如 $A\to A\alpha\,|\,\beta$, $A\in V_\mathrm{N}$, $\alpha,\beta\in(V_\mathrm{T}\cup V_\mathrm{N})^*$ 的产生式称为直接左递归。

形如 $A\to B\alpha\,|\,\beta,B\to A\gamma\,|\,b$, $A,B\in V_\mathrm{N}$, $\alpha,\beta,\gamma,b\in(V_\mathrm{T}\cup V_\mathrm{N})^*$ 的产生式称为间接左递归。

对于直接左递归，简单情况下，形如 $A\to A\alpha\,|\,\beta$，可得 $A\Rightarrow\beta\alpha\alpha\cdots\alpha$，则可以转化为

$$A\to\beta A',A'\to\alpha A'\,|\,\varepsilon$$

直接左递归的一般形式为 $A\to A\alpha_1\,|\,A\alpha_2\,|\cdots|\,A\alpha_n\,|\,\beta$，转化为

$$A\to\beta A',A'\to(\alpha_1\,|\,\alpha_2\,|\cdots|\,\alpha_n)A'\,|\,\varepsilon$$

对于间接左递归，形如 $A\to B\alpha\,|\,\beta,B\to A\gamma\,|\,b$，可以转化为直接左递归：

$$A \to A\gamma\alpha \mid b\alpha \mid \beta \quad 或 \quad B \to B\alpha\gamma \mid \beta\gamma \mid b$$

再按照直接左递归的方法消除。

3.2 语法分析概述

3.2.1 语法分析的功能

语法分析的功能：按照语言的语法规则，识别并分解程序中的各种语法成分。语法分析要解决的问题是给定文法 G 和句子（程序），检查判定是否是 G 能识别的句子。

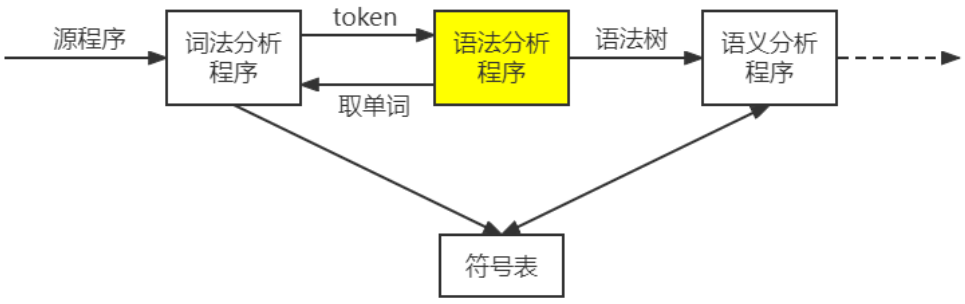


图3.1 语法分析的功能

3.2.2 语法错误处理

要求：

1. 报告出现错误的位置；
2. 修复错误并继续检查后续部分；
3. 执行开销不应太大。

处理策略：

1. 紧急方式恢复：发现错误时，分析器每次抛弃一个输入符号，直到当前输入符号属于某个指定的同步记号集合为止。
2. 短语级恢复：发现错误时，分析器对剩余输入做局部纠正，用可以使分析器继续分析的符号串代替剩余输入串的前缀。
3. 出错产生式：扩充语言的文法，增加产生错误结构的产生式。
4. 全局纠正：在处理不正确的输入串时，作尽可能少的修改，即给定不正确的输入串 x 和文法 G，获得串 y 的分析树，使把 x 变成 y 所需要的插入、删除和修改量最少。

3.3 自顶向下语法分析

3.3.1 概述

自顶向下语法分析方法，亦称面向目标的分析方法。

思想：从文法的开始符出发企图用最左推导推导出与输入的单词串完全匹配的句子。若输入的单词串是给定文法的句子，则必能推出，反之必然以推导失败而终止。

自顶向下语法分析方法分为：

1. 非确定的自顶向下语法分析方法。
2. 确定的自顶向下语法分析方法：实现方法简单、直观，便于手工构造和自动生成，是目前常用的语法分析方法。
   （1）递归下降方法
   （2）LL(1) 方法

在推导过程中，需要从文法中选择产生式来向下推导，而可选的产生式可能不止一条。如果不考虑其他信息，就需要穷举所有的产生式，遇到推不出的地方就回溯，这种方式效率很低。

为了减少回溯、提高效率，我们可以充分利用已知信息，判断出某些推导路径一定行不通，从而进行剪枝。

为了剪枝，考虑用一条产生式推导出的串，如果这个串的第一个终极符与目标串的第一个字符匹配，就可以继续推导；否则可以断定推导失败，从而进行剪枝。对于一条产生式，计算哪些终极符可以出现在这条产生式推导出的串的第一个位置，这些终极符构成一个集合。如果输入流中的第一个字符不属于这个集合，则可以断定不需要使用这条产生式进行推导，这样就达到了剪枝的目的。

设产生式为 $A \to \alpha$，当前串为 $\theta A \delta$，使用产生式推导可得 $\theta A \delta \Rightarrow \theta \alpha \delta$，则：

1. 若 $\alpha \not\Rightarrow^* \varepsilon$，则通过产生式 $A \to \alpha$ 推出的串的第一个终极符为 α 能推出的串的第一个终极符。
2. 若 $\alpha \Rightarrow^* \varepsilon$，则除了 1 中的终极符，还包括 δ 能推出的串的第一个终极符。

为此，可以定义三个集合。

3.3.2 三个集合

3.3.2.1 First集

设 $G = (V_\mathrm{T}, V_\mathrm{N}, S, P)$ 是上下文无关文法，$\beta = (V_\mathrm{T} \cup V_\mathrm{N})^*$，则

$$\mathrm{First}(\beta) = \{a \,|\, \beta \Rightarrow^* a \cdots, a \in V_\mathrm{T}\} \cup (\text{if } \beta \Rightarrow^* \varepsilon \text{ then } \{\varepsilon\} \text{ else } \varnothing)$$

$\mathrm{First}(\beta)$ 的含义是：β 经过推导，可以出现在这个串的第一个位置的终极符的集合。First 集对应于上述情况 1。

设 $X \in (V_\mathrm{T} \cup V_\mathrm{N})$，即 X 为任意字符，则 $\mathrm{First}(X)$ 的计算方法为：

- 若 $X \in V_\mathrm{T}$，则 $\mathrm{First}(X) = \{X\}$。
- 若 $X \in V_\mathrm{N}$，则 $\mathrm{First}(X) = \{a \,|\, X \to a \cdots \in P, a \in V_\mathrm{T}\}$。
- 若 $X \in V_\mathrm{N}$，且有产生式 $X \to \varepsilon$，则 $\varepsilon \in \mathrm{First}(X)$。
- 若 $X \in V_\mathrm{N}$，有产生式 $X \to Y_1 Y_2 \cdots Y_n$，且 $Y_1, Y_2, \cdots, Y_i \in V_\mathrm{N}$，则
  - 若 $Y_1, Y_2, \cdots, Y_{i-1} \Rightarrow^* \varepsilon$，则

    $$\mathrm{First}(Y_1) - \{\varepsilon\}, \mathrm{First}(Y_2) - \{\varepsilon\}, \cdots, \mathrm{First}(Y_{i-1}) - \{\varepsilon\}, \mathrm{First}(Y_i) \subseteq \mathrm{First}(X)$$

  - 若 $Y_i \Rightarrow^* \varepsilon (i = 1, 2, \cdots, n)$，则 $\{\varepsilon\} \subseteq \mathrm{First}(X)$。

设符号串 $\alpha = X_1 X_2 \cdots X_n$，则 $\mathrm{First}(\alpha)$ 的计算方法为：

- 若 $X_1, X_2, \cdots, X_{i-1} \Rightarrow^* \varepsilon$，而 $X_i \not\Rightarrow^* \varepsilon$，则

  $$\begin{aligned}\mathrm{First}(\alpha) = {} & (\mathrm{First}(X_1) - \{\varepsilon\}) \cup (\mathrm{First}(X_2) - \{\varepsilon\}) \cup \cdots \cup \\ & (\mathrm{First}(X_{i-1}) - \{\varepsilon\}) \cup \mathrm{First}(X_i)\end{aligned}$$

- 若 $X_i \Rightarrow^* \varepsilon (i = 1, 2, \cdots, n)$，则

  $$\mathrm{First}(\alpha) = \mathrm{First}(X_1) \cup \mathrm{First}(X_2) \cup \cdots \cup \mathrm{First}(X_n)$$

3.3.2.2 Follow集

设 $G = (V_\mathrm{T}, V_\mathrm{N}, S, P)$ 是上下文无关文法，$A \in V_\mathrm{N}$，S 是开始符号，则

$$\mathrm{Follow}(A) = \{a \,|\, S \Rightarrow^+ \cdots A a \cdots, a \in V_\mathrm{T}\} \cup (\text{if } S \Rightarrow^* \cdots A \text{ then } \{\#\} \text{ else } \varnothing)$$

$\mathrm{Follow}(A)$ 的含义是：能够紧跟在非终极符 A 后面的终极符的集合。Follow 集对应于上述情况 2。

Follow 集的计算方法：

1. 对所有 $A \in V_\mathrm{N}$ 且 A 非开始符号，令 $\mathrm{Follow}(A) = \{\}$；对开始符号 S，令 $\mathrm{Follow}(S) = \{\#\}$。
2. 若有产生式 $A \to xBy$：
   (1) 如果 $\varepsilon \in \mathrm{First}(y)$，则

   $$\mathrm{Follow}(B) = \mathrm{Follow}(B) \cup (\mathrm{First}(y) - \{\varepsilon\}) \cup \mathrm{Follow}(A)$$

   (2) 如果 $\varepsilon \notin \mathrm{First}(y)$，则 $\mathrm{Follow}(B) = \mathrm{Follow}(B) \cup \mathrm{First}(y)$。
3. 重复 2，直至对所有 $A \in V_\mathrm{N}$，$\mathrm{Follow}(A)$ 收敛为止。

> 说明：当 $\varepsilon \in \mathrm{First}(y)$ 时，有如下推导过程 $\alpha A \beta \Rightarrow \alpha x B y \beta \Rightarrow \alpha x B \beta$，则 $\mathrm{First}(\beta) \subseteq \mathrm{Follow}(B)$，而 $\mathrm{First}(\beta) = \mathrm{Follow}(A)$，因此有 $\mathrm{Follow}(A) \subseteq \mathrm{Follow}(B)$。

3.3.2.3 Predict集

$$\text{Predict}(A \to \beta) = \begin{cases} \text{First}(\beta) & \varepsilon \notin \text{First}(\beta) \\ (\text{First}(\beta) - \{\varepsilon\}) \cup \text{Follow}(A) & \varepsilon \in \text{First}(\beta) \end{cases}$$

$\text{Predict}(A \to \beta)$ 的含义是：使用产生式 $A \to \beta$ 推导出的串中，能出现在第一个位置的终极符的集合。Predict 集是对两种情况的综合。

【例3.3】 文法 $\text{E} \to \text{TE'}, \text{E'} \to +\text{TE'} \,|\, \varepsilon, \text{T} \to \text{FT'}, \text{T'} \to *\text{FT'} \,|\, \varepsilon, \text{F} \to \text{id} \,|\, (\text{E})$，则

文法符号	First集	Follow集
E	$\{\text{id}, (\}$	$\{), \#\}$
E'	$\{+, \varepsilon\}$	$\{), \#\}$
T	$\{\text{id}, (\}$	$\{+,), \#\}$
T'	$\{*, \varepsilon\}$	$\{+,), \#\}$
F	$\{\text{id}, (\}$	$\{*, +,), \#\}$

$\text{Predict}(\text{E} \to \text{TE'}) = \text{First}(\text{TE'}) = \text{First}(\text{T}) = \{\text{id}, (\}$

$\text{Predict}(\text{E'} \to +\text{TE'}) = \text{First}(+\text{TE'}) = \{+\}$

$\text{Predict}(\text{E'} \to \varepsilon) = (\text{First}(\varepsilon) - \{\varepsilon\}) \cup \text{Follow}(\text{E'}) = \text{Follow}(\text{E'}) = \{), \#\}$

$\text{Predict}(\text{T} \to \text{FT'}) = \text{First}(\text{FT'}) = \text{First}(\text{F}) = \{\text{id}, (\}$

$\text{Predict}(\text{T'} \to *\text{FT'}) = \text{First}(*\text{FT'}) = \{*\}$

$\text{Predict}(\text{T'} \to \varepsilon) = (\text{First}(\varepsilon) - \{\varepsilon\}) \cup \text{Follow}(\text{T'}) = \text{Follow}(\text{T'}) = \{+,), \#\}$

$\text{Predict}(\text{F} \to \text{id}) = \text{First}(\text{id}) = \{\text{id}\}$

$\text{Predict}(\text{F} \to (\text{E})) = \text{First}((\text{E})) = \{(\}$

至多有一个产生式被选择的条件是：$\text{Predict}(A \to \beta_i) \cap \text{Predict}(A \to \beta_j) = \varnothing, i \neq j$。满足该条件的文法称为 LL(1) 文法。

非 LL(1) 文法到 LL(1) 文法的等价转换：

1. 消除公共前缀。
2. 消除直接左递归。
3. 消除左递归。

注意：

1. 即使文法没有公共前缀和左递归也并不意味着文法一定是 LL(1) 文法，有时还需要其它的转换方法。
2. 通常程序设计语言的文法大都可以转换成 LL(1) 文法，但已经证明，非 LL(1) 文法是存在的，即有些文法是无法变换成 LL(1) 文法的。

3.3.3 LL(1)语法分析

基本思想：从左到右扫描，按最左推导的方式推出输入流。

LL(1) 的含义：

1. 第一个 L 表示：语法分析将按自左至右的顺序扫描输入符号串。
2. 第二个 L 表示：在分析过程中产生一个句子的最左推导。
3. (1) 表示：在分析过程中，每进行一步推导，只要查看一个输入符号便能确定当前所应选用的产生式。

LL(1) 是 LL(k) 的特例，其中的 k 表示向前看 k 个符号。

在逻辑上，一个 LL(1) 分析器由输入流、LL(1) 分析表、符号栈和驱动程序组成：

1. 输入流：待分析的符号串。
2. 符号栈：存放分析过程中的文法符号串。
3. LL(1) 分析表：用来表示相应文法的全部信息的一个矩阵（或二维数组）。
4. 驱动程序：语法分析程序。

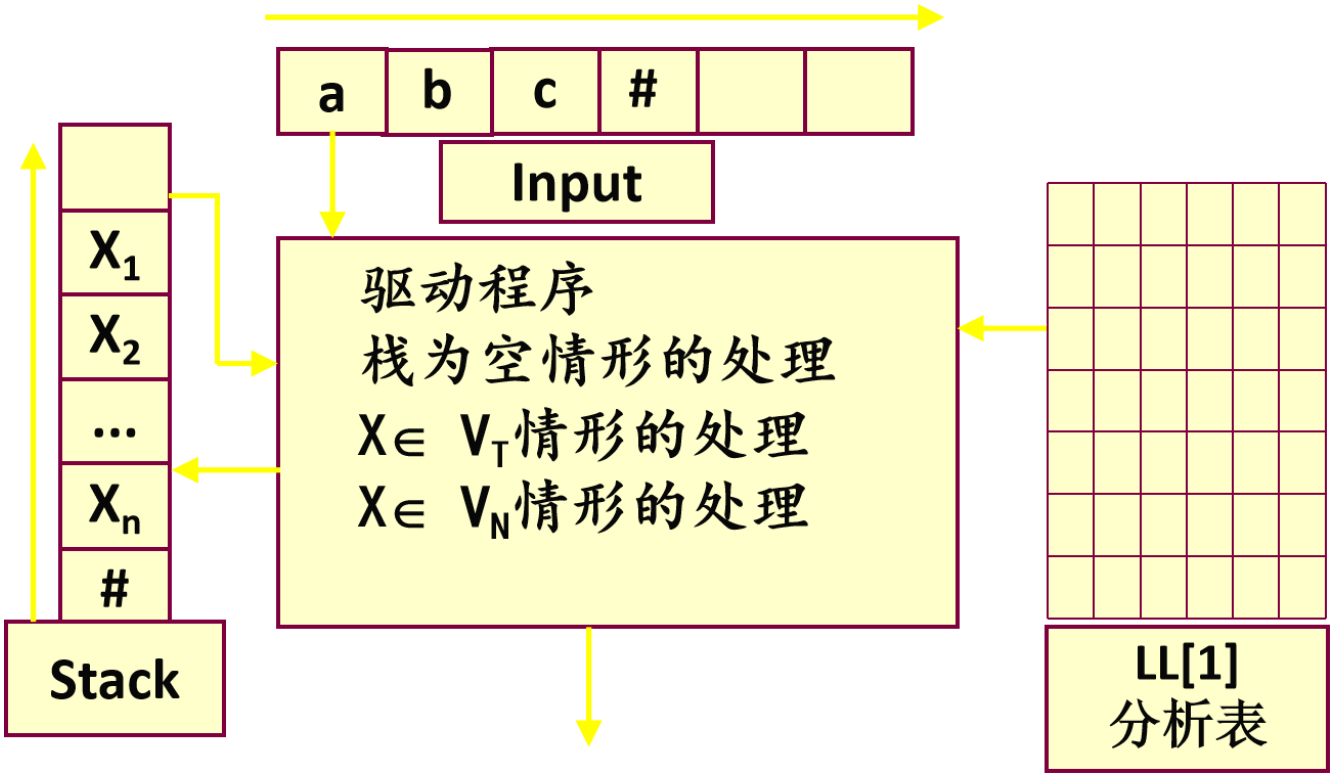


图3.2 LL(1)语法分析器的组成

将符号栈和输入流组成的二元组 $(\text{Stack}, \text{Input})$ 称为**格局**。

假设 X_1 为分析栈栈顶元素，Y_1 为输入流当前符号，则分析动作有：

- 替换：当 $X_1 \in V_\text{N}$ 时，用产生式的右部替换 X_1。
- 匹配：当 $X_1 \in V_\text{T}$ 时，将 X_1 与 Y_1 进行匹配，如果匹配成功则去掉 X_1 和 Y_1，否则报错。
- 成功：当格局为 (空, 空) 时分析成功。
- 报错。有 3 种情况：
  （1）在替换过程中，输入流的当前字符不属于任何产生式的 Predict 集。
  （2）在匹配过程中，分析栈的栈顶元素和输入流的当前值不相等。
  （3）在分析过程中，分析栈和输入流的当前值不同时为空。

LL(1) 分析表的定义：$T : V_\text{N} \times V_\text{T} \to P \cup \{\text{Error}\}$，且

$$T(A,t) = \begin{cases} A \to \alpha & t \in \text{Predict}(A \to \alpha) \\ \text{Error} & \text{否则} \end{cases}$$

LL(1) 分析表的构造方法：

1. 对文法的每一个产生式求其 Predict 集。
2. 对文法的每一个产生式 $A \to \alpha$ 进行如下处理：若 $\text{Predict}(A \to \alpha) = \{a_1, a_2, \cdots, a_n\}$，则令 $T(A, a_i) = A \to \alpha, i = 1, 2, \cdots, n$。
3. LL(1) 分析表的其它元素为 Error。

驱动程序的设计：

1. 分析的初始格局为 $(S\#, a_1 \cdots a_n\#)$。
2. 设存在格局为 $(X_1 \cdots X_m\#, a_1 \cdots a_n\#)$
   - 若 $X_1 \in V_\text{T}$ 且 $X_1 = a_1$，则匹配成功，去掉 X_1 和 a_1，得到新的格局 $(X_2 \cdots X_m\#, a_2 \cdots a_n\#)$。
   - 若 $X_1 \in V_\text{N}$ 则查表，如果 $T(X_1, a_1) = X_1 \to \alpha$，则用 α 替换 X_1，得到新的格局 $(\alpha X_2 \cdots X_m\#, a_1 \cdots a_n\#)$；如果 $T(X_1, a_1) = \text{Error}$，则报错。
   - 其他情况，报错。

驱动程序的实现：

1. 初始化：`Stack := empty; Push(#); Push(S);`，即由符号栈和输入流构成的初始格局为 $(S\#, a_1 a_2 \cdots a_n\#)$。

2. 读第一个输入字符：`Read(a);`。
3. 若当前格局为 $(\#,\#)$，则成功结束，否则转下一步。
4. 设当前格局为 $(X\cdots,a\cdots)$，则：
   - 若 $X\in V_{\mathrm{T}}$ 且 $X=a$，则`Pop(1); Read(a); goto [3];`。
   - 若 $X\in V_{\mathrm{T}}$ 且 $X\neq a$，则报错。
   - 若 $X\in V_{\mathrm{N}}$，则：

$$\begin{aligned}&\texttt{if } T(X,a)=X\to Y_1Y_2\cdots Y_n\\&\texttt{then \{Pop(1); Push(}Y_1Y_2\cdots Y_n\texttt{); goto [3];\}}\\&\texttt{else Error}\end{aligned}$$

3.3.4 递归下降法

递归下降法（Recursive-Descent Parsing）的基本原理：对每个非终极符构造相应的一个子程序（称为语法分析子程序），其功能是识别、分析该非终极符所能推导出的字符串。构造子程序的过程中，终极符产生匹配命令，非终极符产生调用命令。

因为文法是递归的，相应的子程序也是递归的，所以称这种方法为递归子程序方法或递归下降法。

为了保证推导的唯一性，对文法的要求与 LL(1) 文法相同。即对于文法 G 中任一非终极符 A，其任意两个产生式 $A\to\alpha$ 和 $A\to\beta$，都要满足下面条件：$\mathrm{Predict}(A\to\alpha)\cap\mathrm{Predict}(A\to\beta)=\varnothing$。

当产生式形如 $A\to\beta_1\mid\beta_2\mid\cdots\mid\beta_n$，则按下面的方法编写子程序 A：

$$\begin{aligned}&\texttt{procedure A()}\\&\texttt{begin}\\&\quad\texttt{if token}\in\mathrm{Predict}(A\to\beta_1)\ \texttt{then}\ \theta(\beta_1)\ \texttt{else}\\&\quad\texttt{if token}\in\mathrm{Predict}(A\to\beta_2)\ \texttt{then}\ \theta(\beta_2)\ \texttt{else}\\&\quad\cdots\\&\quad\texttt{if token}\in\mathrm{Predict}(A\to\beta_n)\ \texttt{then}\ \theta(\beta_n)\ \texttt{else}\\&\quad\texttt{error()}\\&\texttt{end}\end{aligned}$$

其中对 $\beta_i=X_1X_2\cdots X_n$，$\theta(\beta_i)=\theta'(X_1);\theta'(X_2);\cdots;\theta'(\beta_n)$。

- 如果 $X\in V_{\mathrm{N}}$，则 $\theta'(X)=X()$;
- 如果 $X\in V_{\mathrm{T}}$，则 $\theta'(X)=\mathrm{Match}(X)$;
- 如果 $X=\varepsilon$，则 $\theta'(\varepsilon)=\mathrm{skip}$（空语句）。

主程序：

```
void main()
{
    ReadToken();
    S(); // 初始字符对应的子程序
    if (token == '#')
        success();
    else
        error();
}
```

具体构建流程：

1. 求每条产生式的 Predict 集。
2. 写针对每个非终极符的函数。
3. 写主函数。

优点：构造简单。

缺点：

1. 频繁的函数调用影响效率。
2. 程序比较长。

3.4 自底向上语法分析

3.4.1 概述

自底向上语法分析的基本思想：从待分析的符号串开始，自左向右进行扫描，自下而上进行分析，通过反复查找当前句型的句柄，并将找到的句柄归约为相应产生式左部的非终极符，直到将输入串归约为文法的开始符为止。

自底向上语法分析的动作：

- 移入（Shift）：把输入流的当前字符压入分析栈。
- 规约（Reduce）：把分析栈栈顶的句柄用某一非终极符进行替换。
- 成功（Accept）。
- 失败（Error）。

核心问题：如何确定句柄。不同的找句柄方法构成不同的自底向上语法分析程序。

3.4.2 基本概念

规范句型：用最右推导导出的句型。也称右句型。

规范前缀：若存在规范句型 $\alpha\eta$，且 η 是终极符串或空串，则称 α 为**规范前缀**。

规范活前缀：若规范前缀 α 不含句柄，或者含一个句柄且具有形式 $\alpha = \alpha'\pi$（π 是句柄），则称规范前缀 α 为**规范活前缀**，简称**活前缀**。

归约规范活前缀：若活前缀 α 是含句柄的活前缀，即具有形式 $\alpha = \alpha'\pi$（π 是句柄），则称活前缀 α 为**归约规范活前缀**，简称**归约活前缀**。

派生定理：

1. 开始符号产生式的右部是归约规范活前缀。
2. 如果 $\alpha A\beta$ 是归约规范活前缀，且 $A \to \pi$ 是产生式，则 $\alpha\pi$ 也是归约规范活前缀。
3. 任何归约规范活前缀都可按上述方式被派生。

> **证明：** 开始符号产生式的右部本身就是一个句柄，因此也是归约规范活前缀，第 1 条得证。
>
> 下面证明第 2 条。
>
> 设 $\alpha A\beta\delta$ 是一个规范句型，δ 是终极符串或空串，$\alpha A\beta$ 是归约规范活前缀。对 $\alpha A\beta\delta$ 进行最右推导，得 $\alpha A\beta\delta \Rightarrow^* \alpha A\theta\delta \Rightarrow \alpha\pi\theta\delta$，其中 θ 和 δ 是终极符串或空串。
>
> 由于 $\alpha A\beta$ 是归约规范活前缀，根据定义，句柄出现在 $\alpha A\beta$ 的最右边，因此 α 一定不是句柄。则在 $\alpha\pi\theta\delta$ 中，α 不是句柄，π 是句柄，因此 $\alpha\pi$ 是归约规范活前缀。第二条得证。

一个文法的归约规范活前缀是无穷多的，一般用有限自动机来识别文法所有的归约规范活前缀。

自底向上语法分析的关键问题在于如何构造有限自动机来识别文法的归约规范活前缀。根据有限自动机的构造方法不同，可以分成不同的语法分析方法。

3.4.3 LR(0)语法分析

3.4.3.1 基本思想

符号栈中的内容为 $\#\alpha$，其中有终极符也有非终极符；输入流的内容为 $\beta\#$，全部为终极符。假如要分析的串没有语法错误，则 $\alpha\beta$ 一定是文法的一个句型。

LR(0) 方法的主要思想是，从输入流依次把符号移入符号栈，直至栈顶出现一个句柄；之后对句柄进行归约，直至栈顶不出现句柄；重复上述过程，直至最终归约为一个开始符，且输入流为空。

换句话说，符号栈中的 α 是句型 $\alpha\beta$ 中的规范活前缀，随着 β 中字符的移入，当 α 成为归约规范活前缀时，就对 α 进行归约。重复这一过程，直至符号栈中剩下一个开始符号，且输入流为空。

3.4.3.2 基本概念

LR(0) 项目：若 $A \to \alpha\beta$ 是产生式，则称 $A \to \alpha \cdot \beta$ 为 LR(0) 项目，简称**项目**。项目也可以写作 $\alpha \cdot \beta[p]$。

项目的含义：分析过程中某一个步骤的状态。

项目集的投影：假设 IS 是 LR(0) 项目集，则称 $IS_{(X)}$ 为 IS 关于 X 的投影集，定义为

$$IS_{(X)} = \{A \to \alpha X \cdot \beta \,|\, A \to \alpha \cdot X\beta \in IS, X \in (V_{\mathrm{T}} \cup V_{\mathrm{N}})\}$$

投影集的含义：项目集中每一个项目所描述的状态，处理完一个符号后所对应的状态。

项目集的闭包：假设 IS 是 LR(0) 项目集，则称 $\mathrm{CLOSURE}(IS)$ 为 IS 的闭包集，定义为

$$\mathrm{CLOSURE}(IS) = IS \cup \{A \to \cdot\pi \,|\, Y \to \beta \cdot A\eta \in \mathrm{CLOSURE}(IS), A \to \pi \in P\}$$

项目集的闭包表示分析到了项目集中每一个项目所描述的状态，还需要做的所有归约所用的产生式。

GO 函数：假设 IS 是 LR(0) 项目集，X 是任意的文法符号，则函数 $\mathrm{GO}(IS, X)$ 定义为

$$\mathrm{GO}(IS, X) = \mathrm{CLOSURE}(IS_{(X)})$$

GO 函数的含义：项目集中每一个项目所描述的状态，当处理完一个符号后还需做的所有归约。

3.4.3.3 构造LR(0)归约规范活前缀状态机

为了使 “成功” 状态易于识别，通常 LR 文法要求文法的开始符唯一且不出现于产生式的右部，因此要增加一个新的产生式 $Z \to S$（称为拓广产生式），其中 S 是原文法的开始符，而 Z 则是新符号。

LR(0) 归约规范活前缀状态机称为 LRSM_0，LRSM_0 可以识别所有的归约规范活前缀。

LRSM_0 中的每个状态将对应一个饱和项目集：其中一部分是由先驱状态分出来，称为基本项目；另一部分则是由基本项目扩展出来的，称为扩展项目或派生项目。派生项目的特点是其中的 “·” 出现在产生式右部的最左侧。

构造 LRSM_0 的步骤：

1. 构造初始状态 $IS_0 = \mathrm{CLOSURE}(\{Z \to \cdot S\})$，并给 IS_0 标上 NO。
2. 从已构造的状态机部分图中选择被标为 NO 的任意状态 IS，删除 NO，并对每个符号 $X \in (V_{\mathrm{T}} \cup V_{\mathrm{N}})$ 做如下操作：
   (1) 令 $IS_j = \mathrm{CLOSURE}(IS_{(X)})$。
   (2) 若 IS_j 非空，则：
   - 若在状态机部分图中已有与 IS_j 相同的项目集 IS_k，则在 IS 和 IS_k 之间画有向边：$IS \xrightarrow{X} IS_k$。
   - 若在状态机部分图中没有 IS_j 项目集，则将 IS_j 作为状态机的一个新的状态结点，并给 IS_j 标上 NO，同时在 IS 和 IS_j 之间画有向边：$IS \xrightarrow{X} IS_j$。
3. 重复步骤 2，直至没有被标记为 NO 的状态结点为止。

在 LR(0) 归约规范活前缀状态机中，形如 $A \to \pi\cdot$ 的项目称为归约型项目，形如 $A \to \alpha \cdot \beta$ 的项目称为移入型项目。

如果在同一个项目集中既包含移入型项目又包含归约型项目，称为移入-归约冲突。如果在同一个项目集中包含两个或两个以上归约型项目，称为归约-归约冲突。LRSM_0 中存在冲突的状态称为二义性状态。

如果状态机中任何状态都不存在冲突，则将该文法称为 LR(0) 文法。

状态机提供的信息：

1. 合法性检查信息。例如，项目 $A \to \alpha \cdot a\beta$ 表示 α 后面的输入应该是 a。
2. 移入-归约信息。如果存在形如 $A \to \alpha \cdot a\beta$ 的项目表示需要移入，如果存在形如 $A \to \pi\cdot$ 的项目表示需要归约；如果两种都存在则发生冲突，不能分析。
3. 移入-归约后的转向状态信息。

例如，文法 $\mathrm{S} \to \mathrm{E\$}, \mathrm{E} \to \mathrm{E} + \mathrm{T}, \mathrm{E} \to \mathrm{T}, \mathrm{T} \to \mathrm{id} \,|\, (\mathrm{E})$ 的 LRSM_0 如图 3.3 所示。

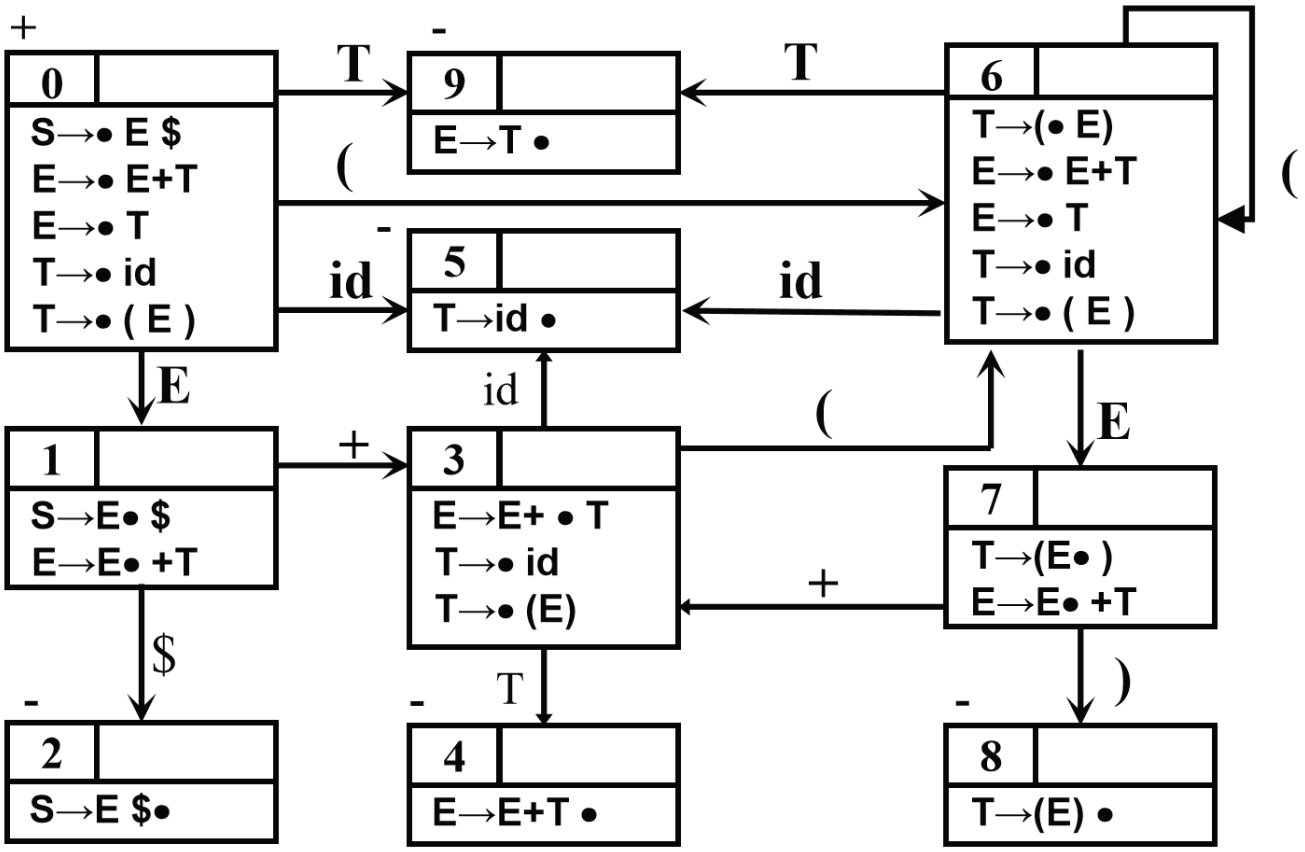


图3.3 归约规范活前缀状态机

3.4.3.4 LR(0)分析表

LR(0) 分析表的结构：

- action 矩阵：行代表状态，列代表终极符，矩阵元素表示分析动作（Shift/Reduce/Accept/Error）。
- goto 矩阵：行代表状态，列代表非终极符，矩阵元素表示移入或归约后的转向状态。

LR(0) 分析表的构造方法：假设 IS_k 为 LR(0) 项目集，则：

- 若 $A \to \alpha \cdot a\beta \in IS_k$，且 $\mathrm{GO}(IS_k, a) = IS_i, a \in V_\mathrm{T}$，则 $\mathrm{action}(IS_k, a) = S_i$，表示将状态 IS_i 和展望符 a 入栈。
- 若 $A \to \alpha \cdot \in IS_k$，则对任意符号 $a \in (V_\mathrm{T} \cup \{\#\})$，令 $\mathrm{action}(IS_k, a) = R_j$，其中产生式 $A \to \alpha$ 的编号为 j，表示用编号为 j 的产生式进行归约。
- 若 $Z \to \alpha \cdot \in IS_k$，且 Z 为拓广产生式的左部非终极符，则 $\mathrm{action}(IS_k, \#) = \mathrm{Accept}$。
- 若 $\mathrm{GO}(IS_k, A) = IS_i, A \in V_\mathrm{N}$，则 $\mathrm{goto}(IS_k, A) = i$。
- 其他情形报错，填入 Error(n)，表示出错标志，也可不填。

3.4.3.5 LR(0)驱动程序

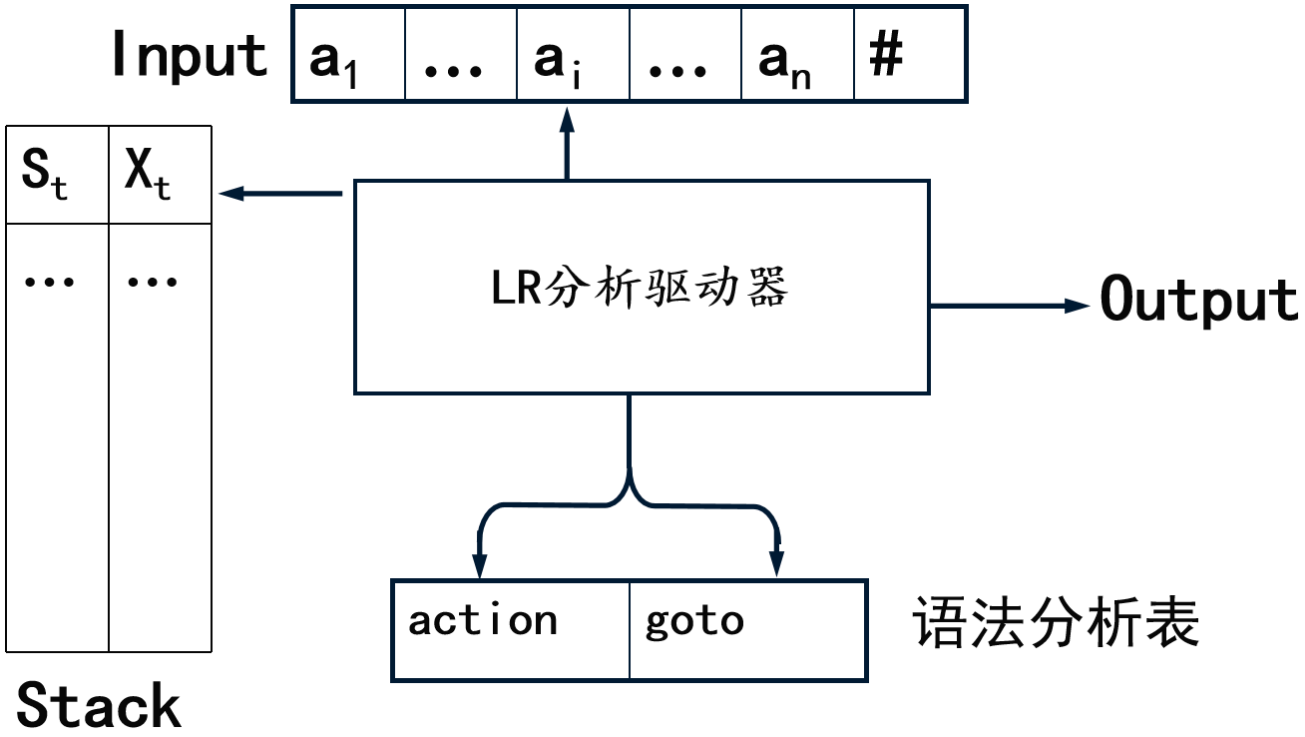


图3.4 LR(0)驱动程序

具体驱动程序算法如下：

1. 置状态栈、符号栈和输入流的开始格局为：$(\#S_1, \#, a_1a_2\cdots a_n\#)$。
2. 若当前格局为 $(\#S_1S_2\cdots S_m, \#X_1X_2\cdots X_m, a_ia_{i+1}\cdots a_n\#)$，且 $\text{action}(S_m, a_i) = S_j, a_i \in V_\text{T}$，表示要进行移入，则 a_i 入符号栈，S_j 入状态栈。移入后格局变为 $(\#S_1S_2\cdots S_mS_j, \#X_1X_2\cdots X_ma_i, a_{i+1}\cdots a_n\#)$。
3. 若当前格局为 $(\#S_1S_2\cdots S_m, \#X_1X_2\cdots X_m, a_ia_{i+1}\cdots a_n\#)$，且 $\text{action}(S_m, a_i) = R_j, a_i \in (V_\text{T} \cup \{\#\})$，则按第 j 个产生式进行归约，符号栈和状态栈相应元素出栈，归约后的文法符号入栈。假设第 j 个产生式为 $A \to \alpha$，$k = |\alpha|(\alpha = X_{m-k+1}\cdots X_m)$，则归约后的格局变为 $(\#S_1S_2\cdots S_{m-k}S, \#X_1X_2\cdots X_{m-k}A, a_ia_{i+1}\cdots a_n\#)$，其中 $S = \text{goto}(S_{m-k}, A)$。
4. 若状态栈的栈顶元素为 S_i，输入流当前值为 $\#$，且 $\text{action}(S_i, \#) = \text{Accept}$，则分析成功。
5. 若状态栈的栈顶元素为 S_i，输入流当前值为 a，且 $\text{action}(S_i, a) = \text{Error}$ 或空，则转向出错处理程序。

3.4.4 SLR(1)语法分析

LR(0) 语法分析方法的不足：对文法的要求严格，容易出现冲突状态。

SLR(1) 语法分析的基本思想：通过求 Follow 集的方法向前看一个符号，来决定语法分析的动作。

如果某个状态有如下项目集 $\{A \to \alpha\cdot, D \to \mu\cdot d\gamma\}$，则存在移入-归约冲突。可以如下解决：

1. 若当前输入符在 A 的 Follow 集中，则应用 $A \to \alpha\cdot$ 归约。
2. 若当前输入符为 d 则应移入。
3. 若当前输入符为 d，而 d 在 A 的 Follow 集中，则无法解决。

如果某个状态有如下项目集 $\{A \to \alpha\cdot, B \to \beta\cdot\}$，则存在归约-归约冲突。可以如下解决：

1. 若当前输入符在 A 的 Follow 集中，则用 $A \to \alpha\cdot$ 归约。
2. 若当前输入符在 B 的 Follow 集中，则用 $B \to \beta\cdot$ 归约。
3. 若当前输入符既在 A 的 Follow 集中又在 B 的 Follow 集中，则无法解决。

若 LRSM$_0$ 中存在状态 $\{A_1 \to \alpha_1\cdot, \cdots, A_n \to \alpha_n\cdot, B_1 \to \beta_1\cdot a_1\gamma_1, \cdots, B_m \to \beta_m\cdot a_m\gamma_m\}$，如果集合 $\text{Follow}(A_1), \cdots, \text{Follow}(A_n), \{a_1, \cdots, a_m\}$ 两两相交为空，则可按如下策略进行分析：

- 若输入符是某个 $a_i, i = 1, 2, \cdots, m$，则移入。
- 若输入符 $a \in \text{Follow}(A_i), i = 1, 2, \cdots, n$，则按项目 $A_i \to \alpha_i\cdot$ 进行归约。

如果集合交集不空，则有 SLR(1) 冲突，分两种情况：

- SLR(1) 移入-归约冲突：$\{a_1, \cdots, a_m\} \cap \text{Follow}(A_i) \neq \varnothing$。

- SLR(1) 归约-归约冲突：$\mathrm{Follow}(A_i) \cap \mathrm{Follow}(A_j) \neq \varnothing, i \neq j$。

若文法的 LRSM$_0$ 不存在 SLR(1) 冲突，则称该文法为 SLR(1) 文法。

SLR(1) 分析表的构造：假设 IS_k 为 LR(0) 项目集，则：

- 若 $A \to \alpha \cdot a\beta \in IS_k$，且 $\mathrm{GO}(IS_k, a) = IS_i, a \in V_\mathrm{T}$，则 $\mathrm{action}(IS_k, a) = S_i$，表示将状态 S_i 和展望符 a 入栈。
- 若 $A \to \alpha \cdot \in IS_k$，则对任意符号 $a \in V_\mathrm{T}, a \in \mathrm{Follow}(A)$，令 $\mathrm{action}(IS_k, a) = R_j$，其中产生式 $A \to \alpha$ 的编号为 j，表示用编号为 j 的产生式进行归约。
- 若 $Z \to \alpha \cdot \in IS_k$，且 Z 为拓广产生式的左部非终极符，则 $\mathrm{action}(IS_k, \#) = \mathrm{Accept}$。
- 若 $\mathrm{GO}(IS_k, A) = IS_i, A \in V_\mathrm{N}$，则 $\mathrm{goto}(IS_k, A) = i$。
- 其他情形报错，填入 Error(n)，表示出错标志，也可不填。

构造 SLR(1) 分析表与 LR(0) 分析表的差别在于，在第二种情况中，LR(0) 方法在所有的终极符及 $\#$ 的列上都填上 R_j，而 SLR(1) 方法仅在 $\mathrm{Follow}(A)$ 的那些列上填 R_j。

LR(0) 和 SLR(1) 分析能力对比：LR(0) 只看分析栈的内容，不考虑当前输入符；SLR(1) 考虑输入符，用 Follow 集来解决冲突，因此 SLR(1) 要比 LR(0) 分析能力强。

SLR(1) 分析的问题：SLR(1) 归约时向前看一个符号，但是不区分语法符号的不同出现。例如文法 $\mathrm{Z} \to \mathrm{BaBbBc}, \mathrm{B} \to \mathrm{d}$，B 出现了三次，第一个 B 的后继只能是 a，第二个 B 的后继只能是 b，第三个 B 的后继只能是 c，而 $\mathrm{Follow(B)} = \{\mathrm{a}, \mathrm{b}, \mathrm{c}\}$，用 SLR(1) 就失去了精度。

3.4.5 LR(1)语法分析

SLR(1) 通过求 Follow 集来决定语法分析的动作，而 Follow 集是非终极符在文法中所有位置上的跟随符，因此 SLR(1) 求展望符的方法不够精确。

LR(1) 语法分析的基本思想：对非终极符的每个不同出现求其后继符，而不是给每个非终极符求其统一的后继符。

3.4.5.1 基本概念

LR(1) 项目：$[A \to \alpha \cdot \beta, a]$，其中 a 称为展望符，$a \in (V_\mathrm{T} \cup \{\#\})$。LR(1) 项目即为 LR(0) 项目与展望符组成的二元组。

用 IS 表示 LR(1) 项目的集合，简称 LR(1) 项目集。其中，项目 $Z \to \cdot\alpha$ 的展望符为 $\#$。

LR(1) 项目集的投影：

$$IS_{(X)} = \{[A \to \alpha X \cdot \beta, a] | [A \to \alpha \cdot X\beta, a] \in IS, X \in (V_\mathrm{T} \cup V_\mathrm{N})\}$$

LR(1) 项目集的闭包：$\mathrm{CLOSURE}(IS) = IS \cup \{[B \to \cdot\gamma, b] | [A \to \alpha \cdot B\beta, a] \in \mathrm{CLOSURE}(IS), B \to \gamma \in P, b \in \mathrm{First}(\beta a)\}$

GO 函数：若 IS 是一个 LR(1) 项目集，X 是一个文法符号，则 $\mathrm{GO}(IS, X) = \mathrm{CLOSURE}(IS_{(X)})$。

3.4.5.2 构造LR(1)归约规范活前缀状态机

1. 构造初始状态 $IS_0 = \mathrm{CLOSURE}(\{[Z \to \cdot S, \#]\})$，并给 IS_0 标上 NO。
2. 从已构造的状态机部分图中选择被标为 NO 的任意状态 IS，删除 NO，并对每个符号 $X \in (V_\mathrm{T} \cup V_\mathrm{N})$ 做如下操作：
   (1) 令 $IS_j = \mathrm{CLOSURE}(IS_{(X)})$。
   (2) 若 IS_j 非空，则：
   - 若在状态机部分图中已有与 IS_j 相同的项目集 IS_k，则在 IS 和 IS_k 之间画有向边：$IS \xrightarrow{X} IS_k$。
   - 若在状态机部分图中没有 IS_j 项目集，则将 IS_j 作为状态机的一个新的状态结点，并给 IS_j 标上 NO，同时在 IS 和 IS_j 之间画有向边：$IS \xrightarrow{X} IS_j$。
3. 重复步骤 2，直至没有被标记为 NO 的状态结点为止。

LR(1) 冲突：

- LR(1) 移入-归约冲突：状态机中某个状态存在形如 $[A \to \alpha \cdot a\beta, b]$ 与 $[A \to \alpha \cdot, a]$ 的项目。
- LR(1) 归约-归约冲突：状态机中某个状态存在形如 $[A \to \alpha \cdot, a]$ 与 $[B \to \alpha \cdot, a]$ 的项目。

若状态机中无 LR(1) 冲突，则其文法为 LR(1) 文法。

3.4.5.3 构造LR(1)分析表

假设 IS_k 为 LR(1) 项目集，则：

- 若 $[A \to \alpha \cdot a\beta, b] \in IS_k$，且 $\mathrm{GO}(IS_k, a) = IS_i, a \in V_\mathrm{T}$，则 $\mathrm{action}(IS_k, a) = S_i$，表示将状态 IS_i 和展望符 a 入栈。
- 若 $[A \to \alpha \cdot, a] \in IS_k$，则 $\mathrm{action}(IS_k, a) = R_j$，其中产生式 $A \to \alpha$ 的编号为 j，表示用编号为 j 的产生式进行归约。
- 若 $[Z \to \alpha \cdot, \#] \in IS_k$，且 Z 为拓广产生式的左部非终极符，则 $\mathrm{action}(IS_k, \#) = \mathrm{Accept}$。
- 若 $\mathrm{GO}(IS_k, A) = IS_i, A \in V_\mathrm{N}$，则 $\mathrm{goto}(IS_k, A) = i$。
- 其他情形报错，填入 Error(n)，表示出错标志，也可不填。

3.4.6 LALR(1)语法分析

LR(1) 语法分析的分析能力较强，能适应大多数文法。

LR(1) 项目的缺点：状态机的状态数太多，构造分析表的工作量和所占的存储空间较大。

3.4.6.1 基本思想

LR(1) 状态机和 LR(0) 状态机从它们所表示的自动机的角度来看是等价的，只是 LR(1) 状态机中存在等价状态，可以通过合并等价状态来减少状态个数。

3.4.6.2 基本概念

项目的心：假设 $[A \to \alpha \cdot \beta, b]$ 是 LR(1) 项目，则称其中的 LR(0) 项目部分 $A \to \alpha \cdot \beta$ 为该项目的心。

状态的心：设 S 是 LR(1) 状态机的一个状态，则 S 的所有项目心之和称为状态的心，表示为 $\mathrm{Core}(S)$。

同心状态：如果 LR(1) 状态机中的两个状态具有相同的心，则称它们为同心状态。

记法约定：

- $\mathrm{SameCoreState}(S)$：所有与 S 同心的状态的集合。
- $\mathrm{Merge}(SS)$：同心状态集 SS 中所有状态项目的并集。

3.4.6.3 构造LALR(1)状态机

按 LR(1) 状态机的方式构造，但发现同心状态时不产生新状态，而是采用合并状态的方法。

LALR(1) 冲突：

- LALR(1) 移入-归约冲突：状态机中某个状态存在形如 $[A \to \alpha \cdot a\beta, b]$ 与 $[A \to \alpha \cdot, a]$ 的项目。
- LALR(1) 归约-归约冲突：状态机中某个状态存在形如 $[A \to \alpha \cdot, a]$ 与 $[B \to \alpha \cdot, a]$ 的项目。

如果状态机中不存在 LALR(1) 冲突，则其文法为 LALR(1) 文法。

可以证明：若原 LR(1) 状态机中无冲突，则合并同心状态后，可能产生归约-归约冲突，但是不可能产生移入-归约冲突。

LALR(1) 分析表和驱动程序与 LR(1) 完全相同。

3.4.7 LR类分析方法比较

从状态数方面看，各种语法分析方法的状态数有如下关系：LR(0)=SLR(1)=LALR(1)<LR(1)

从功能上看，各种语法分析方法的分析能力从小到大依次为：LR(0)<SLR(1)<LALR(1)<LR(1)

4 语义分析

4.1 语义分析概述

语言包括 4 个内容：词法、语法、语义、语用。

程序的语义是指：在为程序单元赋予一定含义时程序应该满足的性质。

语法和语义的区别：

- 语法：关于什么样的字符串才是该语言在组成结构上合法的程序的法则。
- 语义：关于结构上合法的程序的意义法则。

语义的分类：

- 静态语义：在编译阶段（compile-time）可以检查的语义。例如：标识符未声明。
- 动态语义：目标程序运行时（run-time）才能检查的语义。例如：除零、溢出错误。

在编译期间所进行的语义分析称为**静态语义分析**，主要实现程序结构相关的语义检查工作，如静态类型检查。目标代码执行期间进行的语义分析称为**动态语义分析**，主要实现程序运行时的语义检查，如动态类型检查。

静态语义分析的主要内容：

1. 一般性的语义检查
   （1）每个使用性标识符是否都有声明？在同一作用域内有无标识符被声明多次？
   （2）`case`语句的标号是否有声明？有无重复声明和重复定位错误？有无非法转入错误？
   （3）枚举类型的元素定义是否是唯一的？
   （4）下标变量`V[E]`中的`V`是不是变量标识符？`V`是不是数组类型？
   （5）结构体域名变量`V.id`中的`V`是不是变量标识符？`V`是不是结构体类型？`id`是不是该结构体类型中的成员？
   （6）表达式`y + f(10, x)`中的`f`是不是函数名？实参的个数和形参的个数是否一致？
   （7）`break`语句是否有合法的语句包围它？
2. 静态类型检查
   （1）各种条件表达式的类型是不是布尔类型？
   （2）运算符的分量的类型是否相容？
   （3）赋值语句左右部的类型是否相容？
   （4）函数形参和实参的类型是否相容？
   （5）下标表达式的类型是否为所允许的类型？
   （6）函数声明中的函数类型和返回值的类型是否一致？

语义分析的主要任务：

1. 根据声明部分建立符号表。
2. 在整个程序范围内检查常见语义错误。

4.2 符号表

4.2.1 符号表概述

符号表（symbol table）：一种供编译器保存有关源程序的各种信息的数据结构。符号表的每个条目中包含与一个标识符相关的信息，这些信息全面地反映该名字的属性及它们在编译过程中的特征。

符号表通常是一个二维表格，其中每一行称为符号表的一个表项。每个符号表项包含两部分，第一部分是标识符的名字，第二部分是标识符的属性列表。

符号表的作用：

1. 存储标识符的属性。
2. 便于检查语义错误。
3. 中间代码生成阶段用于类型检查及转换。
4. 目标代码生成阶段作为地址分配的依据。

4.2.2 符号表的总体组织

1. 多表结构：将种类相同的标识符组织在一起，构造多个表（如常量表、变量表、函数表等）。
   - 优点：每个符号表的属性个数和结构完全相同，各个表项是等长的，并且表项中的每个属性都是有效的，管理起来方便一致，空间效率高，避免属性存储空间的浪费。
   - 缺点：编译程序将同时管理若干个符号表，增加了总体管理的工作量和复杂度。并且对各类标识符的共同属性如类型的管理必须设置重复的运行机制。
2. 单表结构：将不同种类的标识符都组织在一张符号表中。
   - 优点：总体管理集中单一，且不同种类符号的共同属性可一致地管理和处理，简化了表的管理工作，只需要一个填表函数和一个查表函数。
   - 缺点：由于符号属性的不同，为完整表达各类符号的全部属性必将出现不等长的表项，以及表项中属性位置交错重叠的复杂情况，极大地增加了符号表管理的复杂度。
   - 可行的解决方法：把所有符号的可能属性作为符号表表项属性。

        - 优点：有助于降低符号表管理的复杂度。
        - 缺点：对于某些类具体符号可能增加无用的属性空间，从而增加了空间开销。
3. 折中方式：根据符号属性相似程度分类组织成若干张表，每张表中记录的符号都有比较多的相同属性。

4.2.3 符号表表项的排列

1. 线性组织：规定符号表中的表项按标识符被扫描到的先后顺序填入符号表中。
    - 优点：没有空白项，存储空间效率高。
    - 缺点：运行效率低，特别当表项数目较大后效率就非常低。
2. 排序组织：每次将标识符填入符号表时，均按某个指定的关键字对其进行排序。
   关于排序表的表项建立及符号查找，通常采用 “二分法”。
3. 散列组织：一个符号在散列表中的位置，由对该符号代码值进行某种函数操作（杂凑函数）所得到的函数值来确定。
   优缺点：效率最高，但实现上较复杂而且要消耗一些额外的存储空间。

对应于这三种表项排列方式，符号表的查表方式也可以分为三种：顺序查表法、折半查表法（二分法）、散列查表法（哈希表）。

4.3 标识符的属性表示

标识符的种类：

- 常量标识符
- 类型标识符
- 变量标识符
    - 实在变量
    - 形参变量
        - 值引用型
        - 地址引用型
- 过函标识符
    - 实在过函
    - 形式过函
- 域名标识符

4.3.1 常量标识符的属性表示

属性名	描述
Name	常量的名字
Kind	取值为 constKind，表明该标识符是常量
Type	Type = TypePtr，其中 TypePtr 是指向具体常量的类型的内部表示的指针
Value	Value = ValPtr，其中 ValPtr 是指向具体常量值的内部表示的指针

4.3.2 类型标识符的属性表示

属性名	描述
Name	类型标识符的名字
Kind	取值为 typeKind，表示标识符是类型标识符
Type	Type = TypePtr，指向类型标识符指代的类型的内部表示

4.3.3 变量标识符的属性表示

属性名	描述
Name	变量的名字
Kind	取值为 varKind，表示标识符是变量标识符
Type	Type = TypePtr，指向变量的类型的内部表示

属性名	描述
Access	变量的访问方式。dir 表示直接变量（变量的地址空间中存放的是值），indir 表示间接变量（变量的地址空间中存放的是地址）
Level	表示该变量声明所在主程序/函数/过程的层数
Off	表示该变量相对它所在主程序/函数/过程的内存块起始地址的偏移量
Value	Value = ValPtr，如果变量定义时说明了初值，则为初值的内部表示的指针，否则为空

其中 Level 与 Off 合在一起构成变量的抽象地址，Level 确定了变量所在的内存块，Off 确定了变量在这个内存块中的相对位置。

层数：约定主程序的层数为 0，在主程序中声明的变量标识符和函数标识符的层数也为 0；在 $n(n \geqslant 0)$ 层函数声明中再声明的变量标识符和函数标识符的层数为 $n+1$。

偏移量：在每一层函数中，第一个被声明的变量标识符的偏移量为 0；若第 $n(n \geqslant 1)$ 个被声明的变量标识符的偏移量为 off，且该变量的类型所占空间大小为 t，则第 $n+1$ 个被声明的变量标识符的偏移量为 $off+t$。

4.3.4 过程/函数标识符的属性表示

属性名	描述
Name	过程/函数的名字
Kind	取值为 routKind，表示过程/函数标识符
Type	函数返回值类型的内部表示。过程取值为 NULL
Class	取值为 actual 表示实在过函（有函数体），取值为 formal 表示形式过函（作为其他过函的参数）
Level	表示过程/函数的层数
Off	只对形式过函有效，表示形式过函在所属过函内存块中的偏移量
Param	表示过函的参数表指针。参数表的结构与符号表的结构相同。
Code	只对实在过函有效，表示过函定义对应生成的目标代码的起始地址，当目标代码生成时回填得到。形式过函取值为 NULL
Size	只对实在过函有效，表示过函的过程活动记录所占内存区的大小。当目标代码生成时回填得到
Forward	只对实在过函有效。取值为 true 表示超前声明，取值为 false 表示不是超前声明

形式过函不分配内存空间，Code 和 Size 属性无效。

4.3.5 域名标识符的属性表示

域名标识符是形如 `s.x` 的标识符，其中 `s` 是结构体变量标识符，`x` 是结构体中的域名。

域名标识符在程序中可以像变量一样使用，但其作用域仅限于该域名所在的结构体，因此不需要记录它的层数信息，其偏移量是相对于分配给结构体变量的起始地址的偏移量。

域名标识符属性信息的管理有 2 种方法：

1. 将域名标识符登记到符号表中。（会有同名冲突）
2. 登记到结构体类型的内部表示中。查找时需要先找到结构体，再找到域名。

4.4 类型的内部表示

类型的分类：

- 基本类型：整型（int）、实型（float，real）、字符型（char）、布尔型（bool）
- 构造类型：结构体（struct，record）、数组、枚举（enum）、指针、联合体（union）

4.4.1 基本类型的内部表示

属性名	描述

属性名	描述
Size	占用空间的大小。 `int` 型为 intSize，`bool` 型为 boolSize，`real` 型为 realSize，`char` 型为 charSize。
Kind	类型。 `int` 型为 intTy，`bool` 型为 boolTy，`real` 型为 realTy，`char` 型为 charTy。

Size 属性取值的规定：intSize、boolSize、charSize 为 1，realSize 为 2。

4.4.2 数组类型的内部表示

属性名	描述
Size	占用空间的大小。`Size = sizeof(ElemType) * (Up - Low + 1)`
Kind	取值为 arrayTy，表示数组类型
Low	数组下标的下界
Up	数组下标的上界
ElemType	数组的成分类型的内部表示指针

4.4.3 结构体和联合体类型的内部表示

属性名	描述
Size	占用空间的大小。 结构体类型为所有域的类型的 Size 的总和，联合体类型为所有域的类型的 Size 中的最大值。
Kind	structTy 表示结构体类型，unionTy 表示联合体类型
Body	域名标识符的内部表示链

Body 的内部表示如下。

属性名	描述
Name	域名
Type	指向域的类型的内部表示
Off	只对结构体类型有效，表示域相对于结构体类型分配的内存块起始地址的偏移量。 对于联合体类型而言，所有的域名标识符的起始偏移都是相同的，所以可以省略。
link	指向下一个域名标识符内部表示的指针

4.4.4 枚举类型的内部表示

属性名	描述
Size	枚举类型所占空间的大小
Kind	取值为 enumTy，表示枚举类型
ElemList	指向枚举常量表表头的指针

枚举常量表的内部表示如下。

属性名	描述
Name	枚举常量的名字
Value	枚举常量所代表的整数值

4.4.5 指针类型的内部表示

属性名	描述
Size	指针类型所占空间的大小，一般为 1
Kind	取值为 pointerTy，表示指针类型
BaseType	表示指针所指向空间的类型

4.5 抽象地址

4.5.1 地址分配原则

- 静态分配：在编译时间即为所有数据对象分配固定的地址单元，且这些地址在运行期间始终保持不变。是一种直观的方法，但不适用于动态申请空间。
- 动态分配：程序中变量分配的地址不是具体的地址，而是一个抽象地址，当程序运行时根据抽象地址分配具体的物理地址。

4.5.2 抽象地址的结构

抽象地址的形式是一个二元组，由层数和偏移组成。变量标识符和过函标识符的属性 $(\mathrm{Level}, \mathrm{Off})$ 即为抽象地址。

层数主要是针对嵌套式语言，表示的是某个函数所处的嵌套定义层数。偏移是针对过程活动记录的一个相对偏移量。

4.5.3 层数的定义

嵌套式语言中：

- 主程序设为 0 层。
- 主程序直接定义的函数和过程定义为 1 层。
- 若某函数为 L 层，则该函数直接定义的函数和过程为 $L+1$ 层。

并列式语言中：全局变量定义为 0 层，函数中的变量定义为 1 层。

4.5.4 过程活动记录

每次函数被调用时都会给函数分配一片空间，用以存储如下信息，我们把这片存储空间称作过程活动记录。

临时变量区
局部变量区
形参区
管理信息

偏移是针对过程活动记录的，通过 “起始位置+偏移量” 就可以找到对应的物理位置。

过程活动记录中存储的顺序实际上是处理的先后顺序。

4.5.5 空间分配原则

- 临时变量分配一个单元（这个只是假设，具体情况根据目标机和操作数的类型决定）
- 局部变量按类型大小分配
- 形参
  - 地址引用型形参：分配一个单元
  - 值引用型形参：按类型大小分配
  - 过/函形参：分配两个单元（入口地址，display 表信息）

4.6 符号表的局部化处理

4.6.1 源程序的局部化单位

局部化单位：程序中允许有声明的程序段。

4.6.2 标识符的作用域

一般地，在没有局部化单位的嵌套时，标识符的作用域是从声明该标识符的位置开始到其所在的局部化单位的结束。当有局部化单位嵌套时，标识符的作用域遵循 “最近嵌套原则”，且外层不可访问内层标识符。

一个局部化单位所建立的所有符号表表项称为该局部化单位的符号表。

一个局部化单位的符号表的有效范围是该局部化单位。

在程序的点 P 处，对它有效的符号表是包含 P 的那些嵌套外层的局部化单位的符号表。

在点 P 处遇使用性出现的标识符查符号表时，只能按着由内层到外层的次序（体现标识符的可视性作用域）查在此处有效的那些局部化单位的符号表，而不访问在此无效的那些符号表，这就要求编译程序合理组织符号表，使其能够在程序的每点判断出哪些局部化单位的符号表是有效的，这就是符号表局部化处理的本质。

符号表的局部化：在构造符号表时，保证标识符在其作用域内可以被访问到，而在其作用域之外访问不到。

4.6.3 标识符处理原则

遇到声明性标识符时，首先查找当前局部化单位的符号表中有无与此同名者，若有，则报告重复定义的语义错误；若无，则将该标识符的名字及其属性填入符号表。

遇到使用性出现，按由内到外的次序依次查其嵌套外层的局部化单位的符号表，若有，则根据查得的属性判断对其使用是否合法；若无，则报告有使用而无声明的语义错误。

4.6.4 符号表的组织

从符号表的组织方式上看，可分为两大类：一类是局部式，一类是全局式。

- 局部式符号表：把每个局部化单位的符号表作为一个独立的表来处理，即把每个局部符号表作为建表和查表单位，用一个 Scope 栈实现跨局部化单位的访问。
- 全局式符号表：把整个程序的符号表作为一个表，即建表和查表单位是整个符号表。

4.6.5 局部式符号表

基本思想：

- 在不同时刻，不是所有的局部符号表都有效。为了标记在某一时刻，哪些符号表是有效的，用到一个辅助数据结构 Scope 栈。
- Scope 栈用于存放当前有效的局部化单位的局部符号表的始地址。每当进入一个新的局部化区，就将它的符号表始地址存入 Scope 栈，当退出一个局部化区时，再将 Scope 栈的栈顶元素弹出。

建表过程：

1. 初始化 Scope 栈为空。
2. 每当进入新一层局部化单位，为该局部化单位创建一个新的符号表，并将表的首地址压入 Scope 栈。
3. 该局部化单位声明的标识符依次填入符号表中。
4. 每当退出一层局部化单位时，从 Scope 栈栈顶将该层符号表首地址弹出，删除该符号表。

查表过程：从 Scope 栈栈顶所记录的符号表的首地址，找到最新一层的符号表，先在该表内查找，若未找到要找的标识符，则再依次从次栈顶所记录的符号表开始查找，直到找到或所有符号表中都找不到为止。

4.6.6 全局式符号表

4.6.6.1 删除法

基本思想：

- 每当进入一个局部化区，记住本层符号表的始地址。
- 每当遇到定义性标识符时，构造其语义信息并查本层符号表，若查到同名标识符，则表示有错，否则往符号表里填写标识符及其语义信息。
- 每当遇到使用性标识符时，查整个符号表（从后往前），若查不到同名标识符，则表示有错（因为我们假定声明在前），否则找出相应语义信息并将它传给有关部分。

- 每当结束一个局部化单位时，“删除”本层符号表。

实现细节：

- 给每个局部化单位赋予一个编号。
- 给每个符号表项增加一个属性 num，表示该标识符所属的局部化单位的编号。
- 定义一个全局变量 `CurrentNum`，初值为 0，用于记录分析程序当前所在的局部化单位编号。
- 每当进入一个新的局部化单位，`CurrentNum` 加 1。
- 每当遇到标识符的声明，则检查符号表中 `num == CurrentNum` 的那些表项，是否有同名标识符，若有则出现标识符重复定义错误，否则将该标识符填入符号表中。
- 每当遇到标识符的使用，则检查符号表中 `num <= CurrentNum` 的那些表项，且按照 num 递减的顺序查找，如果都没有，则有标识符未定义错误，否则最先查到的表项为标识符对应的属性。
- 每当退出一个局部化单位，则删除所有 `num == CurrentNum` 的表项，`CurrentNum` 减 1。

4.6.6.2 驻留法

基本思想：不删除表，但采取一定措施不去查那些已无效的符号表部分。

实现细节：

- 总体思路同删除法。
- 当退出一个局部化单位时，仍保留当前局部化单位的表项，但向符号表中填入一个标记项，使得查表时越过那些无效项。
- 标记项：名字为“#”，属性为一个指针，指向跳过无效表项后的最后一个有效表项。

4.6.6.3 散列法

基本思想：删除法的一种，以散列表的方式组织符号表。

实现细节：定义适当的散列函数，以标识符的名字作为主键，计算标识符的符号表地址。当出现多个局部化单位有重名标识符时，采用外拉链的方式化解散列冲突。

5 中间代码生成

5.1 中间代码概述

中间代码是一种介于源程序语言和目标语言之间的语言。

将编译程序中词法分析、语法分析、语义分析、中间代码生成合称为编译前端，中间代码优化和目标代码生成为编译后端。

编译器的分类：

- 没有中间代码生成的编译器
- 有中间代码生成的编译器

没有中间代码生成的编译器的优点：没有中间代码生成目标代码的额外开销，可避免重复性工作，从而减少编译器的体积。

有中间代码生成的编译器的优点：

1. 可使编译程序的结构在逻辑上更为简介、清晰。
2. 在中间代码一级进行优化容易实现。
3. 编译前端只与源程序有关，编译后端只与目标机有关，易于实现编译器的移植，便于编译器开发。

5.2 中间代码的结构

5.2.1 后缀式中间代码

后缀式又称为逆波兰式，指将运算对象写在前面，把运算符写在后面。

后缀式的特点：

1. 后缀式中的变量的次序与中缀式中的变量的次序完全一致。
2. 后缀式中无括号。

3. 后缀式中的运算符已按计算顺序排列。

后缀式的最大优点是易于计算机处理。处理过程：从左到右扫描后缀式，每碰到运算对象就推进栈；碰到运算符就从栈顶弹出相应数目的运算对象施加运算，并把结果推进栈。最后的结果留在栈顶。

5.2.2 抽象语法树

抽象语法树（Abstract Grammar Tree，AGT）可以显式地表示源程序的结构，是常用的一种标准中间代码形式。

5.2.3 有向无环图

有向无环图（Directed Acyclic Graph，DAG）实际上是从抽象语法树发展而来，其主要优点是能够描述共享问题，如表达式的共享。

5.2.4 三地址中间代码

所谓三地址是指操作符的两个运算分量及其该操作的运算结果的抽象地址。

最常见的三地址中间代码有三元式和四元式两种。

三元式的结构：(三元式编号)(算符op，ARG1，ARG2)

四元式的结构：(算符op，ARG1，ARG2，运算结果RESULT)

三元式和四元式的比较：

- 三元式的优点是表示简单，但由于其运算结果反映在三元式的位置编号上，使得中间代码移动或代码删除的优化工作变得复杂，所以三元式中间代码不利于优化。
- 四元式不依赖于位置，因此四元式结构的中间代码便于插入、删除和移动。

一般的四元式操作符应包括以下几类：

1. 算术、逻辑、关系运算符：ADDI（整数加法）、ADDF（浮点数加法）、SUBI（整数减法）、SUBF（浮点数减法）、MULTI（整数乘法）、MULTF（浮点数乘法）、DIVI（整数除法）、DIVF（浮点数除法）、MOD（取余）、AND（逻辑与）、OR（逻辑或）、EQ（等于）、NE（不等于）、GT（大于）、GE（大于等于）、LT（小于）、LE（小于等于）
2. I/O 操作
   （1）整数输入：`(READI, -, -, id)`
   （2）实数输入：`(READF, -, -, id)`
   （3）输出：`(WRITE, -, -, id)`
3. 类型转换：`(FLOAT, id1, -, id2)`（`id2 = float(id1)`）
4. 赋值：`(ASSIG, id1, -, id2)`（`id2 = id1`）
5. 地址相加：`(AADD, id1, id2, id3)`（`id3 = addr(id1) + id2`）
6. 标号定义：`(LABEL, -, -, label)`
7. 条件/无条件转移
   （1）无条件转移：`(JMP, -, -, label)`（转向标号`label`）
   （2）假跳转：`(JMP0, id, -, label)`（若`id`为 0 则转向标号`label`）
   （3）真跳转：`(JMP1, id, -, label)`（若`id`为 1 则转向标号`label`）
8. 过程调用
   （1）子程序入口：`(ENTRY, Label, Size, Level)`
   （2）过函调用：`(CALL, f, -, result)`
9. 参数传递：VARACT（传递变量参数）、VALACT（传递值参数）、FUNACT、PROACT

5.3 语法制导方法

在语法分析过程中，随着源程序结构的一步步识别，同时完成对应的翻译处理。把这种在语法分析的同时进行翻译处理的办法称为语法制导方法。

语法制导方法依赖于具体的语法分析方法，因此可以分为自顶向下语法制导方法和自底向上语法制导方法。自顶向下语法制导方法中通常采用 LL(1) 分析方法为基础，而自底向上语法制导方法中通常以 LR(1) 分析方法为基础。

5.3.1 LL(1)语法制导方法

LL(1) 语法制导方法是在 LL(1) 文法的基础上加入语义动作符来表示相应的语义子程序，语义动作符可以加在产生式右部的任何位置。在进行 LL(1) 语法分析的过程中每遇到语义动作符，则调用相应的语义子程序来完成相应的语义处理。

【例】 有如下 LL(1) 文法 G：$\mathrm{S} \to \mathrm{AB}, \mathrm{A} \to \mathrm{aA} \mid \mathrm{b}, \mathrm{B} \to \mathrm{bB} \mid \mathrm{c}$。要求 LL(1) 应用语法制导方法完成如下任务：对输入文法 G 的任意句子 L，输出 L 中 b 串的长度。

语义动作符及对应的语义子程序如下：

```
void Init()
{
    m = 0;
}

void Out_Val()
{
    printf("%d", m);
}

void Add()
{
    m++;
}
```

G 的动作文法如下：

$$
\begin{aligned}
&\mathrm{S} \to \#Init\# \ \mathrm{AB} \\
&\mathrm{A} \to \mathrm{aA} \mid \mathrm{b} \ \#Add\# \\
&\mathrm{B} \to \mathrm{b} \ \#Add\# \ \mathrm{B} \mid \mathrm{c} \ \#Out\_Val\#
\end{aligned}
$$

G 的带动作符的 LL(1) 分析表如下：

	a	b	c	#
S	$\mathrm{S} \to \#Init\# \ \mathrm{AB}$	$\mathrm{S} \to \#init\# \ \mathrm{AB}$	Error	Error
A	$\mathrm{A} \to \mathrm{aA}$	$\mathrm{A} \to \mathrm{b} \ \#Add\#$	Error	Error
B	Error	$\mathrm{B} \to \mathrm{b} \ \#Add\# \ \mathrm{B}$	$\mathrm{B} \to \mathrm{c} \ \#Out\_Val\#$	Error

对给定的终极符串 abbbc，分析过程如下：

符号栈	输入流	操作
S#	abbbc#	推导
#*Init*# AB#	abbbc#	#*Init*#
AB#	abbbc#	推导
aAB#	abbbc#	匹配
AB#	bbbc#	推导
b #*Add*# B#	bbbc#	匹配
#*Add*# B#	bbc#	#*Add*#
B#	bbc#	推导
b #*Add*# B#	bbc#	匹配
#*Add*# B#	bc#	#*Add*#
B#	bc#	推导
b #*Add*# B#	bc#	匹配
#*Add*# B#	c#	#*Add*#
B#	c#	推导

符号栈	输入流	操作
c #*Out_Val*# #	c#	匹配
#*Out_Val*# #	#	#*Out_Val*#
#	#	成功

5.3.2 LR(1)语法制导方法

LR(1) 语法制导方法是在 LR(1) 文法的基础上加入语义动作符来表示相应的语义子程序，每条产生式最多配有一个语义规则，且语义动作符只能加在产生式的最右部。在进行 LR(1) 语法分析的过程中，每当按照某个产生式的右部进行规约时，就调用该产生式右边的语义子程序来完成相应的语义处理。

【例】 有如下文法 G：$\mathrm{S} \rightarrow \mathrm{AB}, \mathrm{A} \rightarrow \mathrm{aA} \mid \mathrm{b}, \mathrm{B} \rightarrow \mathrm{bB} \mid \mathrm{c}$。要求应用 LR(1) 语法制导方法完成如下任务：对输入文法 G 的任意句子 L，输出 L 中 b 串的长度。

语义动作符及对应的语义子程序如下：

```
void Out_Val()
{
    if (m != 0)
    {
        printf("%d", m);
        m = 0;
    }
}

void Add()
{
    m++;
}
```

G 的动作文法如下：

$$
\begin{aligned}
&\mathrm{S} \rightarrow \mathrm{AB}\ \#Out\_Val\# \\
&\mathrm{A} \rightarrow \mathrm{aA} \\
&\mathrm{A} \rightarrow \mathrm{b}\ \#Add\# \\
&\mathrm{B} \rightarrow \mathrm{bB}\ \#Add\# \\
&\mathrm{B} \rightarrow \mathrm{c}
\end{aligned}
$$

5.4 中间代码生成中的几个问题

5.4.1 语义信息的获取和保存

标识符等语义信息在中间代码生成和目标代码生成阶段都要用到。在中间代码生成中，用于类型检查和类型转换；在目标代码生成中，作为对名字进行地址分配的依据。

四元式 `(op, ARG1, ARG2, RESULT)` 中，`ARG1`、`ARG2`、`RESULT` 为操作分量和运算结果的抽象地址表示，应包含相应语义信息。语义信息有两种表示方式：

1. 指向相应符号表的指针
2. 把对应分量的语义信息放在此处

如果符号表一直保存到目标代码生成阶段，则在四元式中标识符的地址可用其在符号表中的地址表示。

如果符号表不保存到目标代码生成阶段，则需要把目标代码生成阶段对名字进行地址分配所需要的相关信息放到中间代码中，这些信息包括标识符抽象地址（层数、偏移）以及直接/间接访问方式等信息，具体地说这些信息应放在四元式中的操作分量或运算结果的地址表示中。

四元式中操作分量或运算结果的地址表示可以分为三大类：标号类、数值类和地址类。

- 标号类的语义信息是相应的标号值，包括过程/函数体的入口标号。
- 数值类的语义信息就是该数据值。
- 地址类的语义信息由三个部分组成：层数、偏移和访问方式（分为直接访问方式和间接访问方式）。变参变量（指针变量）以及代表复杂变量地址的临时变量属于间接访问方式。

关于临时变量的地址表示：

1. 临时变量没有层数概念，因此对临时变量的层数可以取任意一个负数，如取 -1。
2. 在中间代码生成阶段尽管产生大量的临时变量，但经过优化后，只有少部分临时变量能保留下来，这样在中间代码生成阶段还不能确定临时变量的 Offset 值，因此临时变量的地址表示中的 Offset 暂时取为临时变量的编号。

5.4.2 语义栈Sem及其操作

在语法制导生成中间代码的过程中，要用到一个语义栈， 该栈主要用于存放运算分量和运算结果的类型和地址表示。

假设语义栈 Sem 用数组实现，具体定义如下：

```
#define MAX_SIZE 100

struct SemElem
{
    ElemType typ; // 运算分量的类型
    FORM_Record FORM; // 运算分量的地址
};

SemElem SemStack[MAX_SIZE]; // Sem栈
int top; // 栈顶指针
```

语义栈的操作：

1. `push(x)`：将 `x` 的类型和地址压入 Sem 栈， `x` 为标识符或常量。
2. `pop(n)`：从 Sem 栈顶弹出 n 个元素。

常用的语义子程序：

1. `new_dir(t)`：申请一个临时变量 `t`，且 `t` 是直接寻址。
2. `new_indir(t)`：申请一个临时变量 `t`，且 `t` 是间接寻址。
3. `Generate(op, left, right, result)`：将一条四元式中间代码存放到中间代码区中。
4. `GenCode(op)`：产生一条四元式中间代码。

当执行语义动作子程序 `GenCode` 时，四元式操作码的左、右运算分量的语义信息已经在语义栈 Sem 的次栈顶和栈顶。因此，该子程序的参数只给出操作码 `op` 即可。

此处讨论的表达式涉及的操作符有加法、减法、乘法、除法和类型转换，并且运算分量类型只考虑整型和实型的情况。

`GenCode(op)` 的主要工作：

1. 若 `Sem[top - 1].typ != Sem[top].typ`，且 `Sem[top - 1].typ == integer`，则运算结果类型确定为 real 型，并且为左分量产生类型转换代码：`(FLOAT, Sem[top - 1].FORM, -, tempArg)`。
2. 若 `Sem[top - 1].typ != Sem[top].typ`，且 `Sem[top].typ == integer`，则运算结果类型确定为 real 型，并且为右分量产生类型转换代码：`(FLOAT, Sem[top].FORM, -, tempArg)`。
3. 若 `Sem[top - 1].typ == Sem[top].typ`，则运算结果类型确定为 `Sem[top - 1].typ`，不产生类型转换代码。
4. 产生（类型转换后的）中间代码：`(op*, Sem[top - 1].FORM | tempArg, Sem[top].FORM | tempArg, t)`，其中 `op*` 表示 `op` 的相应整型或实型的中间代码操作码。
5. 从语义栈 Sem 中删除左、右运算分量 `Sem[top - 1]` 和 `Sem[top]`，并把运算结果的语义信息压入 Sem 栈。

5.5 表达式的中间代码

表达式的中间代码就是正确计算表达式值的四元式序列。

表达式中的变量可以是简单、复杂变量（数组变量，结构体成员变量等），还可以是函数调用，而表达式中的运算符也可以包括算术运算符，布尔运算符等。

本节给出简单算术表达式的中间代码生成的 LL(1) 语法制导方法。

简单算术表达式的 LL(1) 文法如下所示：

$$
\begin{aligned}
&(1)\quad \mathrm{E} \to \mathrm{TE'}\\
&(2)\quad \mathrm{E'} \to \varepsilon\\
&(3)\quad \mathrm{E'} \to +\mathrm{TE'}\\
&(4)\quad \mathrm{E'} \to -\mathrm{TE'}\\
&(5)\quad \mathrm{T} \to \mathrm{PT'}\\
&(6)\quad \mathrm{T'} \to \varepsilon\\
&(7)\quad \mathrm{T'} \to *\mathrm{PT'}\\
&(8)\quad \mathrm{T'} \to /\mathrm{PT'}\\
&(9)\quad \mathrm{P} \to \mathrm{C}\\
&(10)\quad \mathrm{P} \to \mathrm{id}\\
&(11)\quad \mathrm{P} \to (\mathrm{E})
\end{aligned}
$$

向其中加入语义动作符，方法为：

- 当遇到常量 C 和简单变量 id 时，把它们的语义信息压入语义栈。
- 当处理完一个运算符（+，-，*，/）的右分量时，该运算符的左、右运算分量已经分别存放在语义栈 Sem 的次栈顶和栈顶的位置，因此可以生成相应的运算符的四元式，并把运算结果的语义信息压入语义栈。

简单表达式的带有动作符的 LL(1) 文法如下所示：

$$
\begin{aligned}
&(1)\quad \mathrm{E} \to \mathrm{TE'}\\
&(2)\quad \mathrm{E'} \to \varepsilon\\
&(3)\quad \mathrm{E'} \to +\mathrm{T}\ \#GenCode(+)\#\ \mathrm{E'}\\
&(4)\quad \mathrm{E'} \to -\mathrm{T}\ \#GenCode(-)\#\ \mathrm{E'}\\
&(5)\quad \mathrm{T} \to \mathrm{PT'}\\
&(6)\quad \mathrm{T'} \to \varepsilon\\
&(7)\quad \mathrm{T'} \to *\mathrm{P}\ \#GenCode(*)\#\ \mathrm{T'}\\
&(8)\quad \mathrm{T'} \to /\mathrm{P}\ \#GenCode(/)\#\ \mathrm{T'}\\
&(9)\quad \mathrm{P} \to \mathrm{C}\ \#Push(\mathrm{C})\#\\
&(10)\quad \mathrm{P} \to \mathrm{id}\ \#Push(\mathrm{id})\#\\
&(11)\quad \mathrm{P} \to (\mathrm{E})
\end{aligned}
$$

LL(1) 分析表如下所示：

	C	id	(	)	+	−	*	/	#
E	1	1	1	Error	Error	Error	Error	Error	Error
E’	Error	Error	Error	2	3	4	Error	Error	2
T	5	5	5	Error	Error	Error	Error	Error	Error
T’	Error	Error	Error	6	6	6	7	8	6
P	9	10	11	Error	Error	Error	Error	Error	Error

【例】 表达式 `x + y * z`，其中所有变量均为整型。

```
(MULTI, y, z, t1)
(ADDI, x, t1, t2)
```

【例】 表达式 `X * 2 + A * (i + 1) / (j + 1)`，其中 `i` 和 `j` 是整型变量，其他为实型变量。

```
(FLOAT, 2, -, t1)
(MULTF, X, t1, t2)
(ADDI, i, 1, t3)
(FLOAT, t3, -, t4)
(MULTF, A, t4, t5)
(ADDI, j, 1, t6)
(FLOAT, t6, -, t7)
(DIVF, t5, t7, t8)
(ADDF, t2, t8, t9)
```

5.6 下标变量的中间代码

5.6.1 下标变量的地址计算

假设有数组类型的变量声明如下：

$$\texttt{A:array}[L_1..U_1][L_2..U_2]\cdots[L_n..U_n]\ \texttt{of T;}\quad (n\geqslant 1)$$

数组 A 占单元大小为：$\text{size}(\texttt{A})=(U_1-L_1+1)(U_2-L_2+1)\cdots(U_n-L_n+1)\alpha$，其中 α 为类型 T 所占单元数。

下标变量取址：$\texttt{A}[i_1][i_2]\cdots[i_n]=\text{addr}(\texttt{A})+(i_1-L_1)S_1+(i_2-L_2)S_2+\cdots+(i_n-L_n)S_n$，其中 S_i 为第 i 层数组变量元素空间大小。

5.6.2 下标变量的四元式结构

对于任意下标变量 $\texttt{A}[E_1][E_2]\cdots[E_k](n\geqslant k)$，其中间代码结构如下：

$$\begin{aligned}
&\texttt{A}[E_1]\text{地址}:\\
&\qquad E_1\to t_1\\
&\qquad (\text{SUBI},\ t_1,\ L_1,\ t_2)\\
&\qquad (\text{MULTI},\ t_2,\ S_1,\ t_3)\\
&\qquad (\text{AADD},\ \texttt{A},\ t_3,\ t_4)\\
&\texttt{A}[E_1][E_2]\text{地址}:\\
&\qquad E_2\to t_5\\
&\qquad (\text{SUBI},\ t_5,\ L_2,\ t_6)\\
&\qquad (\text{MULTI},\ t_6,\ S_2,\ t_7)\\
&\qquad (\text{AADD},\ t_4,\ t_7,\ t_8)\\
&\cdots\cdots\\
&\texttt{A}[E_1][E_2]\cdots[E_k]\text{地址}:\\
&\qquad E_k\to t_{4k-3}\\
&\qquad (\text{SUBI},\ t_{4k-3},\ L_k,\ t_{4k-2})\\
&\qquad (\text{MULTI},\ t_{4k-2},\ S_k,\ t_{4k-1})\\
&\qquad (\text{AADD},\ t_{4k-4},\ t_{4k-1},\ t_{4k})
\end{aligned}$$

【例】 数组定义为 `a: array[1..10][1..5] of integer`，求下表变量 `a[i + 1][j * i - 2]` 的中间代码。

```
// 计算 i + 1
(ADDI, i, 1, t1)

// 求 a[i + 1] 的地址
(SUBI, t1, 1, t2)
(MULTI, t2, 5, t3)
(AADD, a, t3, t4)

// 计算 j * i - 2
(MULTI, j, i, t5)
(SUBI, t5, 2, t6)

// 求 a[i + 1][j * i - 2] 的地址
(SUBI, t6, 1, t7)
(MULTI, t7, 1, t8)
(AADD, t4, t8, t9)
```

5.6.3 下标变量的中间代码生成

下标变量的 LL(1) 动作文法可以写成：

$$\begin{aligned}
&\text{V}\to \#Init\#\ \text{id}\ \#Push(\text{id})\#\ \text{A}\\
&\text{A}\to [\text{E}]\ \#AddNext\#\ \text{B}\\
&\text{B}\to \varepsilon\\
&\text{B}\to [\text{E}]\ \#AddNext\#\ \text{B}
\end{aligned}$$

其中：

- `Init` 用来初始化下标计数器，令 $k=0$。
- 遇到变量标识符，则调用 `Push(id)` 把其语义信息压入 Sem 栈。
- 在下标表达式之后，调用 `AddNext`，用于生成地址累加的四元式。

`AddNext` 算法：

1. 下标计数器加 1，`k = k + 1`。
2. 从语义栈中取出下标表达式结算结果 `result`。
3. 生成计算第 k 维下标需要累加的地址偏移的四元式组：

$$(\text{SUBI},\ \text{result},\ L_k,\ t_{4k-2})$$
$$(\text{MULTI},\ t_{4k-2},\ S_k,\ t_{4k-1})$$

4. 从语义栈中取出地址累加结果 t_{4k-4}。
5. 生成地址累加四元式：$(\text{AADD},\ t_{4k-4},\ t_{4k-1},\ t_{4k})$。
6. `pop(2)`。
7. 把累加后的地址结果 t_{4k} 压入语义栈。

扩展表达式的带有动作符的 LL(1) 文法：

$$
\begin{aligned}
(1)\quad & \mathrm{E} \to \mathrm{TE'} \\
(2)\quad & \mathrm{E'} \to \varepsilon \\
(3)\quad & \mathrm{E'} \to +\mathrm{T}\ \#GenCode(+)\#\ \mathrm{E'} \\
(4)\quad & \mathrm{E'} \to -\mathrm{T}\ \#GenCode(-)\#\ \mathrm{E'} \\
(5)\quad & \mathrm{T} \to \mathrm{PT'} \\
(6)\quad & \mathrm{T'} \to \varepsilon \\
(7)\quad & \mathrm{T'} \to *\mathrm{P}\ \#GenCode(*)\#\ \mathrm{T'} \\
(8)\quad & \mathrm{T'} \to /\mathrm{P}\ \#GenCode(/)\#\ \mathrm{T'} \\
(9)\quad & \mathrm{P} \to \mathrm{C}\ \#Push(\mathrm{C})\# \\
(10)\quad & \mathrm{P} \to \mathrm{id}\ \#Push(\mathrm{id})\# \\
(11)\quad & \mathrm{P} \to (\mathrm{E}) \\
(12)\quad & \mathrm{V} \to \mathrm{id}\ \#Push(\mathrm{id})\#\ \mathrm{A} \\
(13)\quad & \mathrm{A} \to \varepsilon \\
(14)\quad & \mathrm{A} \to \#Init\#\ [\mathrm{E}]\ \#AddNext\#\ \mathrm{B} \\
(15)\quad & \mathrm{B} \to \varepsilon \\
(16)\quad & \mathrm{B} \to [\mathrm{E}]\ \#AddNext\#\ \mathrm{B}
\end{aligned}
$$

LL(1) 分析表如下所示：

	C	id	(	)	+	−	*	/	[	]	#
E	1	1	1	Error	Error	Error	Error	Error	Error	Error	Error
E’	Error	Error	Error	2	3	4	Error	Error	Error	2	2
T	5	5	5	Error	Error	Error	Error	Error	Error	Error	Error
T’	Error	Error	Error	6	6	6	7	8	Error	6	6
P	9	10	11	Error	Error	Error	Error	Error	Error	Error	Error
V	Error	12	Error	Error	Error	Error	Error	Error	Error	Error	Error
A	Error	Error	Error	13	13	13	13	13	14	13	13
B	Error	Error	Error	15	15	15	15	15	16	15	15

5.7 赋值语句的中间代码

赋值语句的形式为：`Left := Right`。

赋值语句的四元式结构为：

- `Left` 的中间代码
  - 如果 `Left` 是简单变量，这部分就为空；
  - 如果 `Left` 是复杂变量（如下标变量），就要有计算变量地址的四元式。
- `Right` 的中间代码
- `(FLOAT/INTEGER, Right, -, t)`
  强制类型转换语句，当赋值号两侧类型不一致时产生类型转换四元式。
- `(ASSIG, Right(t), -, Left)`

赋值语句中间代码生成动作文法为：$\mathrm{S} \rightarrow \mathrm{V} := \mathrm{E}\ \#Assig\#$

`Assig` 需要做如下处理：

1. 从语义栈中取出赋值号左右分量的语义信息。
2. 比较类型是否相同，如果不同，则生成类型转换中间代码。
3. 生成赋值四元式：`(ASSIG, Right(t), -, Left)`

5.8 条件语句的中间代码

条件语句有 2 种形式：

1. if-then-else 型：`<S> -> if <E> then <S1> else <S2>`
2. if-then 型：`<S> -> if <E> then <S1>`

if-then-else 型的中间代码结构：

- `E` 的中间代码
- `(THEN, E.FORM, -, -)`
  作用：条件转移，产生一条假跳四元式 `(JMP0, E.result, -, ?)`，当条件为假时跳转到 `else` 语句。
- `S1` 的中间代码
- `(ELSE, -, -, -)`
  作用：
  - 无条件转到 `if` 语句的后续语句，产生一条无条件跳转指令 `(JMP, -, -, ??)`。
  - 确定条件转移的目标地址。
- `S2` 的中间代码
- `(ENDIF, -, -, -)`
  作用：确定无条件跳转的目标地址

if-then 型的中间代码结构：

- `E` 的中间代码
- `(THEN, E.FORM, -, -)`
- `S1` 的中间代码
- `(ENDIF, -, -, -)`
  作用：确定假跳的目标地址

条件语句的 LL(1) 动作文法：

$$
\begin{aligned}
&<\mathrm{S}> \rightarrow \text{if } <\mathrm{E}> \text{ then } \#ThenIf\# <\mathrm{S}> <\mathrm{ElsePart}>\ \#EndIf\# \\
&<\mathrm{ElsePart}> \rightarrow \text{else } \#ElseIf\# <\mathrm{S}> \\
&<\mathrm{ElsePart}> \rightarrow \varepsilon
\end{aligned}
$$

`ThenIf` 的主要工作：当遇到关键字 `then` 时，该语句的条件表达式计算结果的语义信息已经在语义栈的栈顶，因此可以根据 `Sem[top].FORM` 的值，检查它的类型是否为布尔类型，如果是则产生中间代码 `(THEN, Sem[top], -, -)`。

`EndIf`：遇条件语句结束符，产生中间代码 `(ENDIF, -, -, -)`。

`ElseIf`：当遇到关键字 `else` 时，产生中间代码 `(ELSE, -, -, -)`。

5.9 While语句的中间代码

While 语句的形式为：`S -> while (E) do S`

While 语句的中间代码结构：

- `(WHILE, -, -, -)`
  作用：定位，产生的目标代码为`(LABEL, -, -, inL)`，其中`inL`表示循环入口的位置。
- `E`的中间代码
- `(DO, E.FORM, -, -)`
  作用：条件转移，产生一条假跳四元式`(JMP0, E.FORM, -, outL)`，其中`outL`表示后续语句的入口位置。当条件为假时，跳出循环，跳转到 While 语句的后续语句。
- `S`的中间代码
- `(ENDWHILE, -, -, -)`
  作用：
  - 无条件转移到循环入口处，产生一条无条件跳转指令`(JMP, -, -, inL)`。
  - 确定`outL`。

While 语句的 LL(1) 动作文法：

$$<\mathrm{S}> \rightarrow \text{while } \#StartWhile\# <\mathrm{E}> \text{ do } \#DoWhile\# <\mathrm{S}> \#EndWhile\#$$

`StartWhile`：遇到关键字`while`时，产生中间代码`(WHILE, -, -, -)`。

`DoWhile`：遇到关键字`do`时，检查`E`是否为布尔类型，如果是，则产生中间代码`(DO, Sem[top].FORM, -, -)`。

`EndWhile`：遇循环结束符时，产生中间代码`(ENDWHILE, -, -, -)`。

5.10 goto语句和标号语句的中间代码

有些程序设计语言的标号是用说明语句来定义的（如 Pascal），而大多数程序设计语言的标号则是直接在语句前面使用。这两种情形的中间代码生成过程有所不同。

5.10.1 用说明语句定义的标号

- 标号声明格式：`LABEL L_1, L_2, ... , L_n`
- 转向语句：`goto L_i`
- 标号定位：`L_i: S`

标号的语义信息：所标识的代码地址，或是指其内部标号 （为其分配的一个存储单元，用来存储它所标识的代码地址）。

标号的处理原则：设立标号表，其类似于符号表。当进入一个局部化单位时，建立本层标号表，把本层的标号及其语义信息填入表中；当结束一个局部化单位时，删除本层标号表。

中间代码的生成：

- 当扫描标号声明部分时，用语义子程序`NewLabel`给每个`L_i`分配一个内部标号`LL_i`，将语义信息`(L_i, LL_i)`填入标号表。
- 对转向语句`goto L_i`，产生中间代码`(JMP, –, –, LL_i)`。
- 对标号定位`L_i: ...`，产生中间代码`(LABEL, -, -, LL_i)`。

5.10.2 不用说明语句定义的标号

- 标号定位：`L_i: S`
- 转向语句：`goto L_i`

在处理整个程序之前，需要建立一个数组`ArrayL`来记录当前遇到的所有标号及其语义信息（内部标号及一个用于表示该标号是否已经定位了的标志），初始时为空。

`ArrayL`的结构：

标号名	定位与否标志	地址/语义信息/内部标号

中间代码的生成：

- 每当遇到转向语句`goto L_i`，查`ArrayL`：

    - 如果没有查到该标号，则产生一条缺欠（需回填）转移地址的中间代码 `(JMP, –, –, ?)`，并把标号 `L_i`、JMP 四元式的地址（作为 `L_i` 的语义信息）以及表示该标号为未定位的标记，添加到 `ArrayL`。
    - 若查到标号 `L_i`：
        - 若 `L_i` 是已经定位的了，则从 `ArrayL` 中取出它的地址 `LL_i`，然后产生中间代码 `(JMP, -, -, LL_i)`。
        - 若 `L_i` 未定位，则从 `ArrayL` 中取出它的地址 `LL_i`，然后产生需回填转移地址的中间代码 `(JMP, -, -, LL_i)`，并将 `ArrayL(L_i)` 的地址填入上述中间代码编号。
- 每当遇到标号定位 `L_i: ...`，首先给每个 `L_i` 分配一个内部标号 `LL_i`，产生中间代码 `(LABEL, –, –, LL_i)`。然后查 `ArrayL`，如果没有标号 `L_i` 则把该标号及其相应的语义信息加入 `ArrayL`，并且标记为已定位；如果有标号 `L_i` 并标为未定位，则往对应的所有四元式回填地址；如果查到标号 `L_i` 并标为已定位，则产生标号重复定义错误。

当标号未定位时，相应四元式中的转移地址为上一条未定位的四元式的编号，`ArrayL` 中的地址字段为最后一条未定位的四元式的编号。这样，该标号的所有四元式组成一个链表，`ArrayL` 中的对应表项指向该链表的表头。当标号定位时，从表头开始遍历此链表，为每个四元式回填地址。

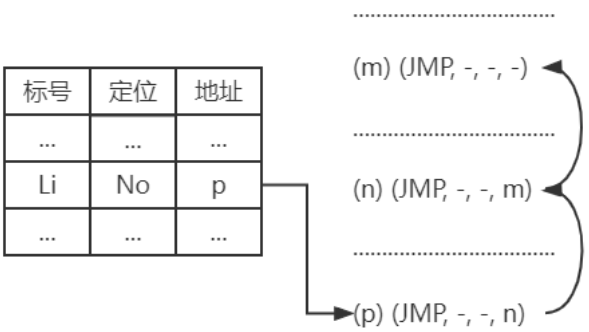


图5.1 未定位的标号

5.11 过程调用和函数调用的中间代码

过程调用和函数调用的文法：

$$
\begin{aligned}
&<\text{ProcFunCall}> \rightarrow \text{id}(<\text{ParamList}>)\\
&<\text{ParamList}> \rightarrow \varepsilon \mid <\text{ExpList}>\\
&<\text{ExpList}> \rightarrow \text{E}\ <\text{NextList}>\\
&<\text{NextList}> \rightarrow \varepsilon \mid , <\text{ExpList}>
\end{aligned}
$$

过程调用和函数调用的语法形式：`id(E_1, ... , E_n)`

过程调用和函数调用的计算过程：

1. 计算各个实在表达式的值。
2. 将计算结果传递给各个形式参数。
3. 转到过程和函数体中执行。

过程调用和函数调用的中间代码结构：

```
E_1的中间代码
E_2的中间代码
……
E_n的中间代码
(VarACT/ValACT, t_1, Off_1, Size_1)
(VarACT/ValACT, t_2, Off_2, Size_2)
……
(VarACT/ValACT, t_n, Off_n, Size_n)
(CALL, f, true/false, [Result])
```

其中：

1. 前 n 条为计算实在表达式的四元式。如果某个实在表达式是一个简单表达式或常量，相应的中间代码为空。

2. 之后的 n 条为传参四元式，`VarACT` 表示形参是变量参数，`ValACT` 表示形参是值参，`t_i` 表示实在表达式的计算结果，`Off_i` 表示形参的偏移，`Size_i` 表示形参占用空间的大小。
3. 最后一条是转子四元式，转到相应的函数体或过程体中执行。`f` 表示过程名或函数名的符号表的入口地址。`true` 表示实在过函，静态确定转向地址；`false` 表示形式过函，动态确定转向地址。`[Result]` 表示函数的返回值。

过程调用和函数调用的 LL(1) 动作文法：

$$
\begin{aligned}
&<\text{ProcFunCall}> \rightarrow \text{id}\ \#CallHead\#\ (<\text{ParamList}>)\ \#CallTail\# \\
&<\text{ParamList}> \rightarrow \varepsilon \mid <\text{ExpList}> \\
&<\text{ExpList}> \rightarrow \text{E}\ \#ActParam\#\ <\text{NextList}> \\
&<\text{NextList}> \rightarrow \varepsilon \mid ,<\text{ExpList}>
\end{aligned}
$$

`CallHead`：当遇到过程/函数名时，将其符号表地址压入 Sem 栈，令实参计数器为 0。

`ActParam`：对每个实参，产生它的中间代码，将结果的类型和语义信息压入 Sem 栈，实参计数器加 1。

`CallTail`：

1. 取出 `id` 的语义信息，包括：过函本身是形式过函还是实在过函，以及所有形参的信息。
2. 检查形、实参个数是否一致，检查形、实参类型是否相容。
3. 产生传参四元式。
4. 产生转子四元式。（需要用到第 1 步中的信息）
5. 删除当前过程/函数调用语句所占用的语义栈单元。如果是函数调用，则把返回值的类型和语义信息压入 Sem 栈。

5.12 过程/函数声明的中间代码

过程/函数声明的文法为：

$$
\begin{aligned}
&\text{ProcFunDec} \rightarrow \text{ProcDec} \mid \text{FunDec} \\
&\text{ProcDec} \rightarrow \text{Procedure id(ParamList);} \\
&\qquad\qquad\ \ \text{Declaration} \\
&\qquad\qquad\ \ \text{ProgramBody} \\
&\text{FunDec} \rightarrow \text{Function id(ParamList):Type;} \\
&\qquad\qquad\ \ \text{Declaration} \\
&\qquad\qquad\ \ \text{ProgramBody}
\end{aligned}
$$

过程/函数声明所特有的四元式有：

1. 入口：`(ENTRY, Label_Q, Size_Q, Level_Q)` 或 `(ENTRY, Q, -, -)`
2. 出口：`(ENDPROC, –, –, –)` 或 `(ENDFUNC, –, –, –)`

过程声明的形式：

```
Procedure P(ParamList);
    LabelDec    // 标号声明
    ConstDec    // 常量声明
    TypeDec     // 类型声明
    VarDec      // 变量声明
    ProFunDec1  // 私有过函声明
    ...
    ProFunDecn
    ProgramBody // 过程体
```

过程声明的中间代码结构：

```
(ENTRY, P, -, -)
ProFunDec1.tuple // 私有过函声明的中间代码
...
ProFunDecn.tuple
ProgramBody.tuple // 过程体的中间代码
(ENDPROC, -, -, -)
```

函数声明的形式：

```
Function F(ParamList): Type;
    LabelDec    // 标号声明
    ConstDec    // 常量声明
    TypeDec     // 类型声明
    VarDec      // 变量声明
    ProFunDec1  // 私有过函声明
    ...
    ProFunDecn
    ProgramBody // 过程体
```

函数声明的中间代码结构：

```
(ENTRY, F, -, -)
ProFunDec1.tuple // 私有过函声明的中间代码
...
ProFunDecn.tuple
ProgramBody.tuple // 函数体的中间代码
(ENDFUNC, -, -, -)
```

过程/函数声明的动作文法：

$$
\begin{aligned}
&\text{ProcFunDec} \rightarrow \text{ProcDec} \mid \text{FunDec} \\
&\text{ProcDec} \rightarrow \text{Procedure id } \#Entry\# \text{ (ParamList);} \\
&\qquad\qquad \text{Declaration} \\
&\qquad\qquad \text{ProgramBody } \#EndProc\# \\
&\text{FunDec} \rightarrow \text{Function id } \#Entry\# \text{ (ParamList):Type;} \\
&\qquad\qquad \text{Declaration} \\
&\qquad\qquad \text{ProgramBody } \#EndFunc\#
\end{aligned}
$$

`Entry`：给子程序`Q`分配新标号`Label_Q`，并将它填到`Q`的符号表项中，产生入口中间代码`(ENTRY, Label_Q, Size_Q, Level_Q)`或`(ENTRY, Q, -, -)`。

`EndProc`和`EndFunc`：产生出口中间代码`(ENDPROC, -, -, -)`或`(ENDFUNC, -, -, -)`。

6 中间代码优化

6.1 中间代码优化概述

优化目标：提高程序的质量，尤其是程序的运行速度。

优化要求：

1. 要保证程序的正确性。
2. 优化后要使程序的运行速度有数量级上的提高。
3. 优化要适度。

优化对象：主要针对循环内下标变量地址的计算。

中间代码优化的分类：

- 源程序阶段的优化：用户基于候选算法的时间复杂度和空间复杂度的比较而做出选择。
- 编译阶段的优化
    - 编译器前端的中间代码级的优化
        - 局部优化：基本块内的优化
            1. 常量表达式优化：合并常数项
            2. 公共表达式优化：消除重复操作
        - 非局部优化
            1. 循环优化：循环不变表达式外提
            2. 全局优化：超越循环之后需要完成
    - 编译器后端的目标代码级的优化

典型的优化方法：

- 常量表达式优化（合并常数）：取固定值的表达式，把表达式的值计算出来后替换常量表达式。
- 公共表达式优化（消除重复操作）：某一个表达式是前面出现的表达式的重复计算，可以用前面表达式的值来替换这个表达式。
- 循环不变式外提：有一个表达式的值在循环中不会改变，把它提到循环体外面，可以大大提高目标程序的执行效率。
- 数学上的优化：可以不计算的则直接删去其四元式，直接写出结果（如减 0、乘 1 等运算）；高运算强度的可以转化成低运算强度的（如乘法换成加法）。
- 表达式短路问题
- 削减程序的运算强度：用强度低的运算代替强度大的运算，通常是针对循环的。
- 消除无用语句，消除冗余代码。
- 中间变量的优化：假如两个临时变量的活动区不相交，则可以共用同一个存储单元。

6.2 基本块

基本块是指程序的一组顺序执行的语句序列，其中只有一个出口和一个入口。入口是指基本块的第一条语句，出口是指基本块的最后一条语句。

基本块的特点：

1. 对于一个基本块而言，执行时只能从它的入口进入，从出口退出。
2. 一个基本块内部的语句要么全执行，要么全不执行，不能执行其中的一部分，不能在中间转出，也不能从中间转入。
3. 基本块可以基于源代码、中间代码和目标代码。

基于四元式中间代码基本块的划分原则：

- 整个四元式序列的第一个四元式为基本块的入口四元式。
- 遇转移性四元式时，结束当前基本块，并把该四元式作为当前基本块的出口，下一条四元式作为新基本块的入口。
- 遇标号性四元式时结束当前基本块，四元式本身作为新基本块的入口。
- 遇到 `(ASSIG, A, -, X)` 时，如果 `X` 为引用型形参，则结束当前基本块，并作为该块的出口。

转移性四元式是指在生成目标代码时一定产生跳转指令的四元式。例如：

- `(JMP, -, -, L)`
- `(JMP1, E, -, L)`
- `(JMP0, E, -, L)`
- `(ENDPROC, -, -, -)`
- `(ENDFUNC, -, -, -)`
- `(THEN, E, -, -)`
- `(ELSE, -, -, -)`
- `(DO, E, -, -)`
- `(ENDWHILE, -, -, -)`

标号性四元式也称定位性四元式，起到一个定位的作用，不产生跳转指令。例如：

- `(LABEL, -, -, L)`
- `(ENTRY, Label, Size, Level)`
- `(WHILE, -, -, -)`
- `(ENDIF, -, -, -)`

程序流图是以基本块为节点的有向图。

6.3 常量表达式优化

常量表达式：任何时候都取固定常数值的表达式。

处理思想：针对每个基本块，如果一个四元式的两个分量的值已知，则由编译器将其结果计算出来，并用所求的值替换原来的运算结果变量，删掉相应的中间代码。

原理：常量定值表 ConstDef，表元素为二元组 `(Var, Val)`。如果在 ConstDef 中有元素 `(V, c)`，表示变量 `V` 此时一定取常数值 `c`，在 `V` 被更改之前出现的 `V` 均可替换成 `c`。

基本块上常量表达式的局部优化算法：

1. 基本块入口置 ConstDef 为空。

2. 读当前四元式。
3. 对当前四元式的分量利用 ConstDef 表进行值代换，得新四元式 newtuple。
4. 如果新多元式 newtuple 形如 `(op, A, B, t)`：若 `A` 和 `B` 是常数，则计算 `A op B` 的值 `v`，并将 `(t, v)` 填入 ConsDef 表。删除当前四元式。
5. 如果新多元式 newtuple 形如 `(ASSIG, A, -, B)`：如果 `A` 是常数，则把 `(B, A)` 填入 ConsDef 表，若已有 `B` 项， 只需修改其值；否则（`A` 为非常数）从 ConsDef 中删除 `B` 的登记项。
6. 重复 2~5 直到基本块结束。

6.4 公共表达式优化

设 $(\omega_1, A_1, B_1, t_1)$ 和 $(\omega_2, A_2, B_2, t_2)$ 是非赋值型运算类四元式，若 $\omega_1 = \omega_2$，并且 A_1 和 A_2、B_1 和 B_2 的值彼此相等，则称这两个四元式等价。也就是说，如果四元式的操作码和两个运算分量的值彼此相等，就说两个四元式等价。

公共表达式（可节省的公共代码，ECC）：如果在基本块中出现多个等价的四元式，则除了第一个外其他的等价的四元式称为第一个四元式的公共表达式，或称可节省的公共代码。

值编码优化方法的主要思想：对中间代码中出现的每个值确定一个编码，使得具有相同值的运算分量对应的编码相同。

编码原理：

- 用到一张值编码表，表示分量当前值的编码。
- 若当前考察的代码为 $\mathrm{dk}: (\omega, u_1, u_2, u_3)$
  - 若值编码表中已有 (u_i, m_i)，则令 $\mathrm{dk}: u_i$ 的值编码为 m_i，$i = 1, 2, 3$。
  - 否则为 $\mathrm{dk}: u_i$ 创建一个新编码 m 填入编码表。
- 若当前考察的代码为 $\mathrm{dk}: (=, u_1, -, u_2)$
  - u_1 的处理与之前相同。
  - 令 $\mathrm{dk}: u_2$ 的编码与 $\mathrm{dk}: u_1$ 的编码相同。

值编码的性质：

1. 不同分量上的相同常量一定具有相同值编码。
2. 不同分量上的相同变量未必具有相同编码。
3. 不同常量的编码值一定不同，对变量来说并非如此。
4. 每当一个变量 `x` 被赋值，`x` 将得到一个新的编码，使得后面代码中的 `x` 分量取编码值 `n`，直至 `x` 再被赋值。

在分析的过程中对应每条四元式还要生成一个编码四元式。令 $\mu(x)$ 表示任意运算分量 x 的值编码，把 $(\omega, \mu(A), \mu(B), \mu(t))$ 叫做 (ω, A, B, t) 的映象码。

基于值编码的公共表达式优化用到三张表：

1. 值编码表（ValuNum）：存放变量（或常数）及其值的编码。
2. 可用表达式代码表（UsableExpr）：存放基本块中的可用于匹配的表达式编码四元式。
3. 临时变量等价表（PAIR）：存放等价的临时变量偶对。(t_i, t_j) 表示 t_i 和 t_j 是等价的，需要用 t_i 替换 t_j。

基于值编码的公共表达式优化算法：从基本块的第一条四元式开始，执行以下操作：

- 临时变量等价表（PAIR）替换，(t_i, t_j) 需要用 t_i 替换 t_j。
- 对四元式中的运算分量和结果临时变量进行编码。
- 查值编码表进行替换，生成编码四元式（当前表达式的映象码）。
- 遇到运算型四元式，查可用表达式代码表（UsableExpr）中是否存在与编码四元式运算符、运算分量都相同的，若存在则说明当前表达式为公共表达式，删去当前四元式，把结果临时变量填入等价表 PAIR。
- 遇到赋值 `(=, a, -, b)` 时，`b` 的编码赋值为 `a` 的编码。

例：

```
a = b * c + b * c;
d = b;
e = d * c + b * c;
```

序号	中间代码	映像码	a	b	c	d	e	t1	t2	t3	t4	t5	t6	等价表	优化后
1	`*,b,c,t1`	`*,1,2,3`	-	1	2	-	-	3	-	-	-	-	-	-	不变

序号	中间代码	映像码	a	b	c	d	e	t1	t2	t3	t4	t5	t6	等价表	优化后
2	`*,b,c,t2`	`*,1,2,3`	-	1	2	-	-	3	3	-	-	-	-	`(t1,t2)`	节省
3	`+,t1,t2,t3`	`+,3,3,4`	-	1	2	-	-	3	3	4	-	-	-	-	`+,t1,t1,t3`
4	`=,t3,-,a`	-	4	1	2	-	-	3	3	4	-	-	-	-	不变
5	`=,b,-,d`	-	4	1	2	1	-	3	3	4	-	-	-	-	不变
6	`*,d,c,t4`	`*,1,2,3`	4	1	2	1	-	3	3	4	3	-	-	`(t1,t4)`	节省
7	`*,b,c,t5`	`*,1,2,3`	4	1	2	1	-	3	3	4	3	3	-	`(t1,t5)`	节省
8	`+,t4,t5,t6`	`+,3,3,4`	4	1	2	1	-	3	3	4	3	3	4	`(t3,t6)`	节省
9	`=,t6,-,e`	-	4	1	2	1	4	3	3	4	3	3	4	-	`=,t3,-,e`

6.5 循环不变式外提

循环不变式：如果一个表达式 E 在一个循环中不改变其值，则称它为该循环的不变表达式。

循环不变式外提：将循环不变式提到循环外面执行。

循环不变式外提的关键：

- 识别循环结构（循环入口、循环体、循环出口）。
- 检查循环体中哪些变量的值被改变过。
- 根据这个结果来看哪些表达式是循环不变式。
- 建立变量定值表（LoopDef），将循环体中值被改变的变量都填到表里，若某运算型四元式中两个运算分量都不出现在这个表里，就说明其值不发生改变，可以进行外提。

循环不变式外提算法：

- 对循环体四元式进行第一遍扫描，把有定值的变量填到变量定值表中。若它是一个运算型四元式 $(\omega_1, A_1, B_1, t_1)$，则把 t_1 填到表中；若为赋值型四元式 $(=, a, -, b)$，则把 b 填入表中。
- 循环不变式外提为第二遍扫描，每遇到一个运算型四元式 $(\omega_1, A_1, B_1, t_1)$，若 A_1、B_1 都不在变量定值表中，则将其提到循环体外，同时在变量定值表中删去 t_1，把 t_1 从本层变量定值表移到外层变量定值表。

注意：

1. 多层循环问题中，一个四元式从里层开始可以被外提若干次，里层变量定值表属于外层变量定值表。
2. 除法不外提。
3. 赋值绝不外提。
4. 非良性循环（循环体有函数调用或地址引用型变量的赋值）不做外提优化。
5. 运算型四元式的运算分量是间接寻址时，该四元式不外提。

7 运行时存储空间管理

7.1 运行时存储分配策略

目标程序占用的存储空间：

- 代码空间
  - 库代码空间：用于存放标准库函数的目标代码。
  - 目标代码空间：用以存放编译生成的目标程序。
- 数据空间
  - 静态区空间：有些数据对象所占用的空间可以在编译时确定，其地址可以编译进目标代码中，这些数据对象通常存放在静态区中，如静态变量和全局变量。
  - 栈区空间：存放过程活动记录。该存储区被称作一个栈，一个元素是一个过程活动记录，调用函数时压栈，函数结束时退栈。
  - 堆区空间：堆不是一个连续分配的模式，可以进行动态分配的空间管理，主要用于存放动态申请的数据对象。

栈区和堆区之间没有事先划好的界线，当目标代码运行时，栈区指针和堆区指针不断地变化，并朝着对方方向不断增长。如果这两个区相交，则表示出现了内存溢出。

语言影响空间分配策略的因素：

- 语言中是否允许函数、过程的递归调用
  语言如果允许递归调用，则函数的形参、局部量可能对应一串存储单元，不能采用静态的存储分配方式；如语言不允许有递归出现，则一个函数最多只能分配一个活动记录大小的空间。
- 当一个函数结束时局部变量是否需要保存
  一般来说，一个函数的局部变量在函数调用结束的时候所占用的存储空间都要被释放掉。但是有一类语言的特殊语句，允许函数中的某个局部变量保存着，下一次再调用这个函数的时候，这个局部变量已经具有上一次调用的值。
- 是否可以访问非局部的变量
  外层的变量、其他变量是否可以被访问，不同的语言有不同的约定。
- 函数参数的传递方式
  常用的参数传递方式是值引用和地址引用，这就决定了地址引用型实参只能是一个变量，而不能是一个常数。形参的值引用和地址引用的空间分配的方式是不一样的。
- 函数是否可以作为参数进行传递
- 函数的结果是否可以为函数
- 动态地申请存储块
- 是否显式地释放存储空间

存储管理模式的分类：

- 静态存储分配策略：在编译时为数据对象分配固定的存储空间，且存储对象的存储位置在程序的整个生命周期是固定的。
- 动态存储分配策略
  - 栈式动态存储分配策略：是一种最常见的模式，在有函数调用的时候动态地分配存储空间。若程序运行过程中有动态进行申请和释放程序空间，用栈式的就不行了，没有办法静态地在活动记录里给动态变量分配空间。
  - 堆栈混合式动态存储分配策略：活动记录的大小确定地存在栈区中，动态申请的空间在堆区中申请然后释放，同时增加一部分对堆区的管理程序

完全采用静态分配策略的语言必须满足以下约束条件：

1. 不允许递归过程。
2. 数据对象的长度和它在内存中的位置，必须是在编译时可知。
3. 不允许动态建立的数据结构（如动态数组、指针等），因为没有运行时的存储分配机制。

7.2 过程活动记录的申请和释放

过程活动记录是栈式管理中最重要的内容。栈区中通常需要设立两个指针：

1. sp：指向当前活动记录的起始位置
2. top：指向第一个可用的存储单元

过程的活动记录：为管理过程、函数的一次活动所需要的信息，目标程序要在栈区中给被调过程分配一段连续的存储空间，以便存放该过程的局部变量值、控制信息和寄存器内容等，称这段连续的存储空间为过程的活动记录，简称活动记录（Activation Record），并记为 AR。

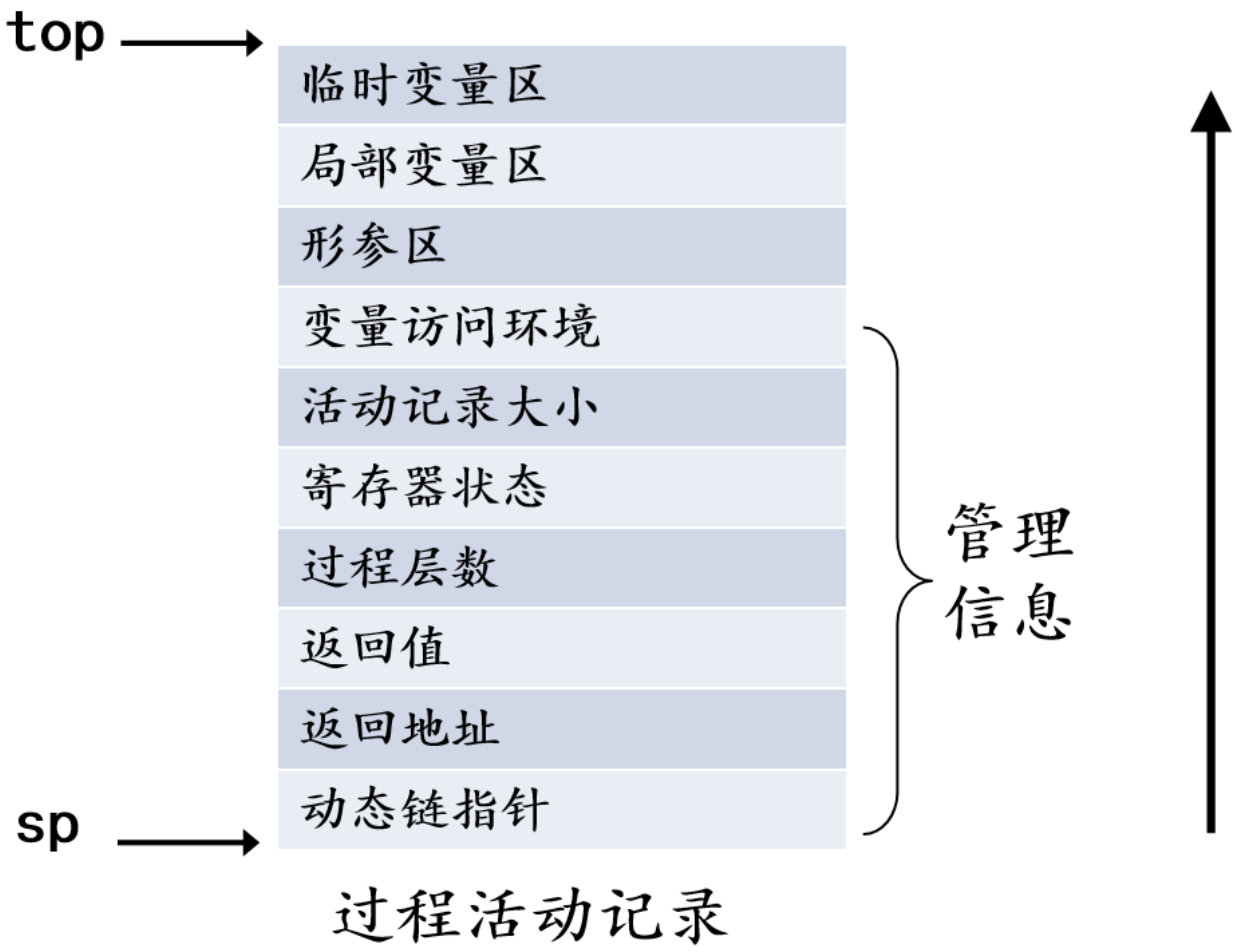


图7.1 栈式管理中的过程活动记录

临时变量区：用于存放表达式计算的中间结果。当寄存器不足以存放所有这些临时变量时，可以把它们存放在临时变量区中。

局部变量区：保存过程局部声明的数据。

形参变量区：用于存放调用过程提供的实在参数。

变量访问环境：提供非局部变量的访问环境。

活动记录大小：当过程的局部数据中没有动态数组等可变长数据时，过程的活动记录的长度能够静态确定，将其记录下来，可以用于过程调用时确定现行 AR 的始地址。

寄存器状态：保存刚好在过程调用前的机器状态信息，包括程序计数器的值和控制从这个过程返回时必须恢复的机器寄存器的值。

过程层数：辅助变量构建访问环境和变量寻址。

返回值：存放一存储空间地址，该存储空间用于存放被调用过程返回给调用过程的值。

返回地址：指向调用者调用指令的下一条指令。

动态链指针：动态链也称为控制链。用来指向调用者的活动记录，以便在过程调用返回时将当前活动记录恢复成调用者的活动记录。

注意：过程活动记录其实并没有包含过程一次执行所需的全部信息，比方说非局部数据就不在活动记录中。另外，过程运行时生成的动态变量也不在活动记录中，对它们通常采用堆式分配。

遇到函数过程调用时申请地址，具体来看在遇到 call 四元式中间代码时，要生成相应的目标代码。要做的工作有两个：

1. 产生一个新的活动记录，即 `sp = top; top = top + size;` 。
2. 保存寄存器内容，填写过程活动记录的管理信息，如返回地址、寄存器内容、动态链指针等等。

遇到 `return` 语句或遇到函数的结束语句时，要释放过程活动记录，要做的主要工作有：

1. 恢复现场，将寄存器里的值恢复。

2. 释放当前活动记录，即 `top = sp; sp = 动态链指针;` 。
3. 根据返回地址创建跳转指令。

调用链：过程名的序列，序列的头是主程序名 M。具体地说：

- (M) 是调用链。
- 若 $(M, \cdots, R)$ 是调用链，并且 R 中有 S 的调用，则 $(M, \cdots, R, S)$ 也是调用链。

对于任一过程（函数）S，其调用链不是唯一的，每个调用链对应于一个动态的过程调用序列。

用 $\mathrm{CallChain}(S) = (M, \cdots, R, S)$ 表示 S 的调用链，表示当前正在执行的是 S 的过程体，而 $M, \cdots, R$ 则是已经开始执行但被中断了的过程。

动态链：如果当前正在执行的是 S，并且 $\mathrm{CallChain}(S) = (M, N, \cdots, R, S)$，则栈的当前内容可表示为 $[\mathrm{AR}(M), \mathrm{AR}(N), \cdots, \mathrm{AR}(R), \mathrm{AR}(S)]$，称它为对应调用链 $(M, N, \cdots, R, S)$ 的**动态链**，表示为 $\mathrm{DynamicChain}(S) = [\mathrm{AR}(M), \mathrm{AR}(N), \cdots, \mathrm{AR}(R), \mathrm{AR}(S)]$。

7.3 变量地址映射

地址分配原则：

- 值引用的形参按照类型大小分配。
- 地址引用的形参分配为 1。
- 局部变量按照类型长度分配。
- 临时变量分配 1。
- 函数和过程作为形参分配 2，其中一个是实参的入口地址，另一个是先行的 display 表地址。

抽象地址的变化规律：

处理前可用的抽象地址	处理内容	处理后可用的抽象地址
`(L, off)`	LabelDec（标号声明）	`(L, off)`
`(L, off)`	ConstDec（常量声明）	`(L, off)`
`(L, off)`	TypeDec（类型声明）	`(L, off)`
`(L, off)`	`var id: T`（变量声明）	`(L, off + n_T)` （`n_T` 表示 T 类型的长度）
`(L, off)`	ProcDec（过程声明）	`(L, off)`
`(L, off)`	FuncDec（函数声明）	`(L, off)`
`(L, off)`	`Proc p()` （实在过程首部）	`(L + 1, off1)` （`off1` 表示处理完形参时的偏移量）
`(L, off)`	`Func f(): T` （实在函数首部）	`(L + 1, off1)`
`(L, off)`	`Proc P()` （形式过程）	`(L, off + 2)`
`(L, off)`	`Func F(): T` （形式函数）	`(L, off + 2)`
`(L, off)`	`var ID: T` （地址引用的形参声明）	`(L, off + 1)`
`(L, off)`	`ID: T` （值引用的形参声明）	`(L, off + n_T)`
`(L, off)`	`Proc p(` （实在过程名及左括号）	`(L + 1, d)` （`d` 表示活动记录控制信息所占的大小）
`(L, off)`	`Func f(` （实在函数名及左括号）	`(L + 1, d)`

并列式语言：相对简单，只有 0 层和 1 层，可以用两个指针来解决，`sp0`指向全局量的首地址，`sp`指向当前活动记录的首地址，根据变量的层数来确定是`sp0+off`还是`sp+off`。

嵌套式语言：若抽象地址是`(L + 1, off)`，如果`L`等于当前层，则说明它所对应的变量处于当前的活动记录中（`L + 1`层变量位于`L`层过函中）；如果`L`不等于当前层，就要找到他所对应的活动记录的首地址，为此要构造一个 display 表，把每一个活动记录活跃的静态外层的首地址都存起来，然后找到`L`所对应的那一层的首地址加上`off`就可以了。

7.4 变量访问环境

一个过程 S 在动态链中可有多个活动记录，但其中只有最新的活动记录是可访问的，称此活动记录为 S 的**活跃活动记录**，并记为 $\mathrm{LiveAR}(S)$，简写为 $\mathrm{LAR}(S)$。

过程声明链：

- 过程名序列 (M) 是过程声明链，M 是主程序名。
- 若 $(M,\cdots,P)$ 是过程声明链，且 P 中有过程 Q 的声明，则 $(M,\cdots,P,Q)$ 也是过程声明链，记为 $\mathrm{DeclaChain}(Q)=(M,\cdots,P,Q)$。

当前变量访问环境：若 $\mathrm{DeclaChain}(Q)=(M,\cdots,P,Q)$，则当前变量访问环境为 $\mathrm{VarVisitEnv}(\mathrm{LAR}(Q))=(\mathrm{LAR}(M),\cdots,\mathrm{LAR}(P),\mathrm{LAR}(Q))$。

非局部变量访问的实现：假设 Q 的变量访问环境为 $\mathrm{VarVisitEnv}(\mathrm{LAR}(Q))=(\mathrm{LAR}(M),\cdots,\mathrm{LAR}(P),\mathrm{LAR}(Q))$，在 Q 中有变量 X_Q、Y_M、Z_P，它们分别定义在过程 Q、M 和 P 中，则它们的存储单元地址可表示如下：

$$\begin{aligned}\mathrm{addr}(X_Q)&=<\mathrm{LAR}(Q)>+Offset_X\\ \mathrm{addr}(Y_M)&=<\mathrm{LAR}(M)>+Offset_Y\\ \mathrm{addr}(Z_P)&=<\mathrm{LAR}(P)>+Offset_Z\end{aligned}$$

其中，$<\mathrm{LAR}(Q)>$ 表示 $\mathrm{LAR}(Q)$ 的始地址。

结论：对于每个活动记录 AR，只要知道了它的变量访问环境 $\mathrm{VarVisitEnv}(AR)$，即可实现包括非局部变量在内的所有变量的访问。

定理：设 $(\mathrm{AR}(M),\cdots,\mathrm{AR}(P),\mathrm{AR}(Q))\in\mathrm{DynamicChain}(Q)$，且 Q 的层数为 N，则有 $\mathrm{VarVisitEnv}(\mathrm{AR}(Q))=\mathrm{VarVisitEnv}(\mathrm{AR}(P))_N\oplus\mathrm{AR}(Q)$。

结论：变量访问环境可由先行过程的变量访问环境求得。

7.5 变量访问环境的实现方法

7.5.1 局部display表

对于每个 AR 求出其变量访问环境，并把它以地址表的形式（display 表）保存在 AR 中。因为每个 AR 都自带 display 表，称这种方法为局部 display 表方法。

如果层数为 N 的过程 P 的变量访问环境为 $\mathrm{VarVisitEnv}(\mathrm{AR}(P))=[AR_0,\cdots,AR_n]$，$ar_i$ 表示 AR_i 的始地址，则 $[ar_0,\cdots,ar_n]$ 是 $\mathrm{AR}(P)$ 的 display 表。

display 表的求法：`NewAR.display = CurrentAR.display的前N项 + newsp`

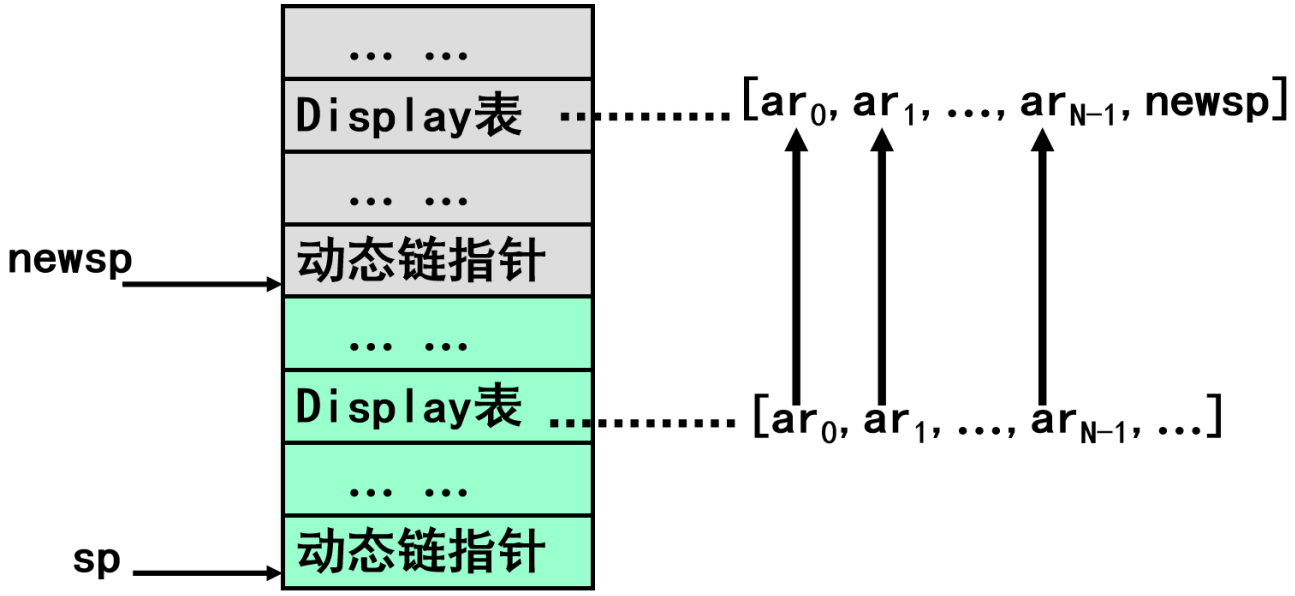


图7.2 display表的求法

对一个变量 X，其抽象地址为 `(L+1, off)`，则地址的计算方法为：

1. 当 `L == CurrentAR.level` 时， `addr(X) = sp + off` 。
2. 否则， `addr(X) == CurrentAR.display[L] + off` 。

7.5.2 全局display表

每个程序设置一个总的 display 表，其长度为最大嵌套层数（最长声明链的长度），其中 `display[i]` 存放第 `i` 层最新 AR 的指针，用 `D[i]` 表示。

该方法的理论依据：在程序的任何一点，相同层数的过程声明只能有一个有效。

在 AR 中设置一个 Resume 单元，用来临时保存某 `D[i]` 。

当层数为 `j` 的过程 Q 被调用时：

1. 将旧的 `D[j]` 的内容保存到 `NewAR(Q)` 中： `NewAR(Q).ResumeAddr = D[j];`
2. 改写 `D[j]` 的内容： `D[j] = NewAR(Q)的地址;`

当退出 Q 时，恢复原来 `D[j]` 的内容： `D[j] = CurrentAR.ResumeAddr;`

局部变量 `X(L+1, off, indir/dir)` 的访问：X 的地址为 `addr(X) = D[L] + off` 。

7.5.3 静态链

活动记录中原 display 表部分变成一个单元，称为静态链单元，存放静态链指针。静态链指针指向 display 表中当前活动记录的前驱活动记录。

静态链指针的确定：若 `k == CurrentAR.level - (NewAR.level - 1)` ，则 `NewAR.StaticChainPointer = Indir(sp, k)` ，其中 `Indir(sp, k)` 表示 `sp` 的 `k` 次间接内容。

变量 `X(L+1, off)` 的地址：

- 若 `L == CurrentAR.Level` ，则 `addr(X) = sp + off` 。
- 否则， `k = CurrentAR.Level - L` ， `addr(X) = Indir(sp, k) + off` 。

7.5.4 总结

display 表方法是用表结构表示变量访问环境。

- 局部 display 表的产生需要花空间，但返回时不需要为恢复变量访问环境做任何事情。
- 对于全局 display 表方法而言，display 表的产生需要花时间，而且返回时也需要为恢复变量访问环境而花时间，其主要优点是能节省存储单元。

静态链方法是用链表表示变量访问环境。静态链方法实际上是一种共享化的局部 display 表方法。其主要优点同全局 display 表方法是能节省存储单元。产生需要花时间，但返回时不需要为恢复变量访问环境做任何事情。

具体采用哪种方法，取决于机器条件：如果寄存器较少，则使用 display 表方法可能合适些；如果机器能提供较好的间接操作，则可选用静态链方法。

8 目标代码生成

8.1 目标代码生成概述

目标代码：在目标机上能够直接运行的可执行代码。

目标代码一般有以下两种形式：

1. 使用绝对地址的机器语言代码
   - 绝对机器语言代码即是能够立即执行的机器语言代码，代码中所有地址均已定位，编译后可直接执行。
   - 这种形式的目标代码执行速度最快，在编译过程中，通常要把整个源程序一起编译，而不能独立地编译源程序中的各个程序模块，一般只适用于需要快速编译执行的小型程序。
2. 可重定位的机器代码
   - 当程序执行时，必须由连接装入程序把它们和一些运行时子程序连接起来，并完成地址的重新定位，转换成可立即执行的机器语言代码。
   - 装入程序用来把程序加载到内存中，以便执行。装入程序可处理所有的与指定的基地址或起始地址有关的可重定位的地址，它使得可执行代码更加灵活。
   - 连接程序由于使用了可重定位的地址，这种形式代码的各个子程序模块可以分别编译，由连接程序负责将分别在不同的目标文件中编译或汇编的代码集中到一个可执行文件中。并且，库函数的代码以及程序所使用的图标和字符串等资源，也是由连接程序连接到目标文件中。

目标代码的质量与目标机的指令系统有关：

- 指令系统丰富的目标机，对给定的操作可以提供多种实现方法，不同的实现方法之间的效率差别可能会非常大。
- 目标代码的质量常用目标指令的条数和执行速度来衡量。规定：访问内存一次的代价为 1，执行一次操作的代价为 1。

8.2 虚拟机

定义一台虚拟机作为目标机器，在其上介绍目标代码生成的基本原理。

假定虚拟的目标机按字编址，拥有 1 个通用寄存器 R，每个机器字存放一条指令。

指令的格式为：`OP destination, source`。`OP` 表示操作符，`destination` 表示目的操作数，`source` 表示源操作数。`destination` 和 `source` 可以是立即数、寄存器、存储字地址，但不能同时为存储字地址。

虚拟机的寻址方式：

寻址方式	汇编形式	地址
绝对地址	`M`	M（内存单元）
寄存器	`R`	R（寄存器）
变址	`C(R)`（变址寄存器 R 的内容）	C+Content(R)
间接寄存器	`*R`（寄存器中存放的是操作数的地址）	Content(R)（内存单元）
间接变址	`*C(R)`	Content(C+Content(R))（内存单元）
立即数	`C` / `#C`	无需寻址

虚拟机的指令系统：

指令名称	指令形式	指令含义
读	`IN R`	将外部值输入到寄存器 R 中

指令名称	指令形式	指令含义
写	`OUT R`	将寄存器 R 中的值输出
取数	`LD R, A`	将地址 A 中的内容存入寄存器 R 中
存数	`ST A, R`	将寄存器 R 中的内容存入地址 A 中
加	`ADD R, A`	将寄存器 R 中的内容与地址 A 中的内容相加，结果存入寄存器 R 中
减	`SUB R, A`	将寄存器 R 中的内容与地址 A 中的内容相减，结果存入寄存器 R 中
乘	`MULT R, A`	将寄存器 R 中的内容与地址 A 中的内容相乘，结果存入寄存器 R 中
除	`DIV R, A`	将寄存器 R 中的内容与地址 A 中的内容相除，结果存入寄存器 R 中
条件真转移	`JMPT R, A`	R 中的内容为真，则转向地址 A
条件假转移	`JMPF R, A`	R 中的内容为假，则转向地址 A
无条件转移	`JMP A`	无条件地转向地址 A
取址	`LEA R, A`	将 A 的地址放入寄存器 R 中
块传送	`MOVEB A1, A2, S`	将起始地址为 A2，长度为 S 的内存区中的内容传送到起始地址为 A1 的内存区

在虚拟目标机中，取出几个寄存器作为地址计算专用的寄存器，分别为 SP、TOP、SP0；其他寄存器用 R1、R2...表示。

8.3 四元式转化为目标指令

四元式等价地转换成目标指令需要用到两个栈：

1. 标号定位栈 L1：定位性标号是为了某些转移提供地址的，需要把暂时没用到的标号存在栈中。例如 while 四元式可以对应一个嵌套的循环，在定位产生时还没产生跳转指令，它的地址还没用到，为了让后面能用到需要用栈把标号保存下来。
2. 目标指令地址栈 L2：在有些产生跳转指令的时候，转移地址暂时无法确定，例如 do 四元式，不知道后面的转移地址，则把当前目标指令地址存到栈里，在知道转移地址以后，回填这个指令地址。回填地址是编译中的一项非常重要的技术。

8.3.1 运算型四元式

运算型四元式 `(op, X, Y, T)` 生成 3 条目标指令：

```
LD R, X  ;把X取到R中
op R, Y  ;根据运算符对应的运算生成指令，结果存入R
ST T, R  ;将R中的内容存入T中
```

两点重要的解释：

1. 目标指令中 X，Y，T 对应着一个具体的地址，如 `sp+off_X` 。
2. 如果 X、Y 为间接变量，这里的寻址方式为间接。如 X 为间接变量，用 `*X` 表示它是一个间接寻址方式。

8.3.2 赋值语句四元式

赋值型的四元式 `(=, X, -, Y)` 生成 2 条目标指令：

```
LD R, X  ;把X取到寄存器中
ST Y, R  ;把寄存器中的内容存到Y中
```

8.3.3 输入输出语句四元式

输入语句四元式的一般形式为 `(READ, -, -, A)` ，生成的目标代码为：

```
IN R
ST A, R
```

输出语句四元式的一般形式为 `(WRITE, A, -, -)` ，生成的目标代码为：

```
LD R, A
OUT R
```

8.3.4 条件语句四元式

if-else 型条件语句的形式为`if <E> then <S1> else <S2>`，其中间代码形式为：

```
E的中间代码
(THEN, X, -, -)
S1的中间代码
(ELSE, -, -, -)
S2的中间代码
(ENDIF, -, -, -)
```

生成目标代码的过程如下：

1. 生成 E 的目标代码。
2. 对于四元式`(THEN, X, -, -)`，生成一条半指令，并将转移指令的地址`n+1`压入 L2 栈。

```
[n] LD R, X
[n+1] JMPF R, *  ;半条假跳指令，跳转的目标地址未知
```

3. 生成 S1 的目标代码。
4. 对于四元式`(ELSE, -, -, -)`，产生半条无条件转移指令`[m] JMP *`；然后，从 L2 栈中取出假跳指令，将无条件转移指令的下一个地址`m+1`回填，得到完整的假跳指令`[n+1] JMPF R, m+1`；最后，将`m`压入 L2 栈。
5. 生成 S2 的目标代码。
6. 对于四元式`(ENDIF, -, -, -)`，不产生目标代码，从 L2 栈中弹出无条件转移指令，将当前可用的目标代码的地址回填。

对于不带 else 子句的条件语句，没有`(ELSE, -, -, -)`和 S2 的中间代码，对应的目标代码也不用生成。四元式`(ENDIF, -, -, -)`将地址回填到`(THEN, X, -, -)`生成的假跳指令中。

8.3.5 循环语句四元式

while 语句的形式为`while <E> do <S>`，其中间代码形式为：

```
(WHILE, -, -, -)
E的中间代码
(DO, E.FORM, -, -)
S的中间代码
(ENDWHILE, -, -, -)
```

生成目标代码的过程如下：

1. 对于四元式`(WHILE, -, -, -)`，不产生目标代码，将当前目标指令地址`n+1`压入 L1 栈。
2. 生成 E 的目标代码。
3. 对于四元式`(DO, X, -, -)`，生成一条半指令，将假跳指令的地址`m+1`压入 L2 栈。

```
[m] LD R, X  ;将条件表达式的结果读入寄存器R中
[m+1] JMPF R,*  ;半条假跳指令
```

4. 生成 S 的目标代码。
5. 对于四元式`(ENDWHILE, -, -, -)`，将 L1 栈顶元素弹出，生成一条无条件转移指令`[k] JMP L1(top)`，目标地址为 L1 栈顶语句的地址；L2 栈顶出栈，将地址`k+1`回填给 L2 栈顶代码，得到完整的假跳指令`JMPF R, k+1`。

8.3.6 标号语句和goto语句的四元式

标号声明语句`Label L`不产生目标代码，只需为它分配存储单元。

标号定位语句`L: S`的四元式为`(LABEL, -, - L)`，不产生目标代码，只需将当前目标代码地址存入 L 对应的标号表单元。

goto 语句的四元式为`(goto, -, -, L)`，产生半条无条件转移指令`JMP *(L)`，对应的地址暂时为一个指向标号表 L 的指针。目标代码全部生成以后，对所有 goto 语句的目标代码，依据标号表中对应的目标代码地址进行回填。

8.3.7 过程函数声明的四元式

过程函数声明的入口四元式为`(ENTRY, Q, -, -)`，其中 Q 为过程名或函数名的符号表地址。对其四元式不产生目标代码，只需把当前指令地址填入 Q 的语义信息的 Code 项中。

设过程活动记录的首单元存放动态链指针，第二个单元存放返回地址，第三个单元存放返回值。

对于过程函数声明的出口四元式`(ENDFUNC, -, -, -)`或`(ENDPROC, -, -, -)`，生成目标代码的过程为：

1. 生成一组读取命令，用于恢复寄存器的现场信息。
2. 删除本层活动记录，使动态外层的活动记录成为当前活动记录，生成两条指令：

```
ST top, sp
LD sp, *0(top)  ;作废当前活动记录
```

3. 产生一条返回指令，根据返回地址生成一个跳转指令`JMP *1(top)`。如果是函数返回语句，要增加一个操作，将返回结果存储到活动记录`*2(top)`中。

8.3.8 过函调用的四元式

过程函数调用语句的四元式结构为：

```
E1的中间代码
E2的中间代码
……
En的中间代码
(VarACT/ValACT, t_1, Off_1, Size_1)
(VarACT/ValACT, t_2, Off_2, Size_2)
……
(VarACT/ValACT, t_n, Off_n, Size_n)
(CALL, f, true/false, [Result])
```

值参传递四元式`(ValACT, t, Offset, size)`的目标代码生成：

- 若 t 为间接变量，则生成的目标代码为：

```
LD R, *t
ST Offset(top), R
```

- 若 t 为直接变量，则生成的目标代码为：

```
LD R, t
ST Offset(top), R
```

- 若 t 为数组，则生成成组传送的目标代码为：`MOVEB t, Offset(top), size`。

变参传递四元式`(VarACT, t, Offset, size)`的目标代码生成：

- 若 t 为直接变量，则生成的目标代码为：

```
LEA R, t
ST Offset(top), R
```

- 若 t 为间接变量，则生成的目标代码为：

```
LD R, t
ST Offset(top), R
```

过函调用语句`(CALL, f, true, [Result])`的目标代码生成：

1. 生成填写变量访问环境的指令。
2. 把机器状态（寄存器内容）保存到活动记录的寄存器状态区中，一般应生成一组存的指令。
3. 填写管理信息。首先填写过程层数，从过函 f 的语义信息中取其层数，填入到`3(top)`中，生成指令为：

```
LD R, sem[f].level
ST 3(top), R
```

4. 填写动态链指针：`ST 0(top),sp`。

5. 填写返回地址：

```
[A] LD R, A+5
[A+1] ST 1(top), R
```

6. 生成过程活动记录：

```
[A+2] ST sp, top
[A+3] ST top, top + sem[f].size
```

7. 生成转向过函 f 入口的指令：`[A+4] JMP sem[f].code`。
8. 如果是函数调用，则在调用返回时把函数值读到寄存器中：

```
[A+5] LD R, 2(top)
[A+6] ST result, R
```