第 7 章　中间代码生成

7.1　导入

7.1.1　为什么要研究语义分析

词法分析是基于有限状态自动机，扫描识别字符串序列中的语素（关键字、标识符、运算符、常数、界符等），为下一部的语法分析做基础。

语法分析是基于自底向上或自顶向下的分析方法，检查词法分析处理过后的语素集合是否能被该编程语言产生式匹配的过程，这个过程会隐式的生成语法树。在语法分析里，我们可以检查代码更大粒度的正确性。

符号表被很多人认为是语义分析的一部分（剩下的一块拼图就是本章的中间代码生成）。在平时编译阶段，对于类型检查，如将整型当作数组类型使用；控制流，如 C 语言中大括号的使用是否前后呼应；重复定义，如不能在一个函数中重复定义两次同一个变量；作用域，如全局变量和局部变量的使用等，这些问题，通过语义分析可以找出。

自然语言的语义与程序语言的语义是不一样的，严格意义上讲，自然语言的语义会非常丰富，而程序语言的语义相对固定。语义分析的目的是利用语法分析的结果，把源语言转换成机器可读的形式，称为中间代码，这个过程也称为语义分析。生成中间代码的过程，要去定义一些语义动作，这些语义动作就是语法分析每一个局部的结果，后面会有例子。

7.1.2　为什么需要生成中间代码

在程序编译过程中，可执行代码与具体的操作系统、硬件等相关，而生成的中间代码与操作系统、硬件设施无关，具有较好的移植性，方便了编译器系统的开发。另一方面，也便于进行优化操作，如程序中的 1+2，在经过优化后直接变成 3，这样的话，就节省了一步，如果是在循环或实时系统中，这样的优化就十分关键

另一个原因就是直接把源语言转换成汇编指令，这种方式的好处是效率高，缺点是无法把中间过程进行分解，系统复杂，不易调试。常用手段是两小步的方法，先把源语言转换成中间代码，然后再把中间代码转换成目标语言。

概括性的总结一下这一小节:

1. 易于修改，对于复杂的系统来讲，一般用模块化进行开发，把大任务或复杂问题简化，这里有一个重要的概念——耦合，我们希望模块间的耦合度不要太大，加入中间代码也是降低模块间耦合度的一种方式，使用中间代码这种特殊的表达形式来作为中间衔接，而不是程序接口这种简单复杂的形式，易于开发、调试、维护。
2. 跨平台，易于迁移。
3. 首先生成中间代码给了我们更多的优化算法介入的空间，因为越抽象底层的代码优化起来难度更大，

先生成可转为汇编的中间语言，让我们既朝底层更进一步，却也保留了一些优化的可能性，毕竟汇编确实看起来让人觉得有些繁琐。

7.2 中间代码生成

7.2.1 常用的中间代码形式

中间代码可以理解成一个源语言句子表示成机器容易理解的形式，这种形式可以很容易地转化成目标语言。

设有赋值语句：$x = a * b + c$，则有以下中间语言：

（1）逆波兰式：$xab * c+ =$

在编写计算器程序时，使用的就是逆波兰式的形式。以 a+b 为例，我们平时写的 a+b 这种形式，是中缀式，而逆波兰式形式得到的是 ab+，又称后缀式，将操作项放在前面，运算符放在后面。

（2）四元式

```
(1) ( *   a   b   t1 )
(2) ( +   t1  c   t2 )
(3) ( =   t2  _   x )
```

或

```
(1) t1 = a * b
(2) t2 = t1 + c
(3) x = t2
```

四元式表示一元或二元运算，可以顺序地把程序执行的过程表达出来。第一条四元式表示，用 * 对 a 和 b 进行操作，结果赋给 t1。第二条表示，t1 与 c 相加，结果赋给 t2。第三条表示，把 t2 的值赋给 x。这个过程与赋值语句 x = a * b + c 是一样的。四元式也可以简单地写成二元运算的形式，左边和右边框表示的含义是一样的，一般写成左边这种方式，比较规整。

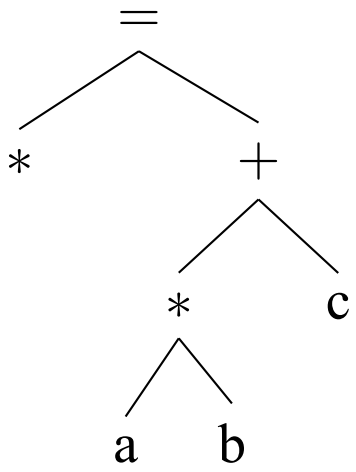
四元式形式具有更强的表达能力和可读性，因此在后续中间代码生成中使用更多。

（3）三元式

三元式和四元式的区别在于，四元式的最后一项叫结果变量，而三元式没有，是用①来表示计算的结果。

```
① (*   a   b)
② (+   ①   c)
③ (=   ②   x)
```

（4）语义树



把 a*b 进行计算，结果在与 c 相加，最终把结果赋给 x。

这四种方式是常用的形式，还有很多其它方式，这里不一一列举。这四种方式都是非常有效的中间代码表示形式，把程序语言表示成一种便于计算机计算的方式。这四种方式也各有优缺点，简单总结一下：

> 逆波兰式**简单**；四元式**清楚**；
> 三元式**节省**；语义树**直观**；

7.2.2 各种语法成分的中间代码设计

逆波兰式

给定一个表达式，可以把它写成逆波兰式的形式。逆波兰式把运算符放在操作数的后面比如：

$$a+b \Longrightarrow ab+$$

表达式的逆波兰式设计

设 pos(E) 为表达式 E 的逆波兰式；则：

①$pos(E1\omega E2) = pos(E1)pos(E2)\omega$

②$pos((E)) = pos(E)$

③$pos(i) = i$

其中，ω 是运算符，i 是运算对象（变量或常量）

【注】（1）上述三个定义式，为逆波兰翻译法则；（2）定义式中的 ω 为 E1ωE2 中最后运算的算符！例如，式 a+b*c 中，E1 相当于 a，E2 相当于 b*c，ω 为 +。

例 7.1 表达式逆波兰式翻译示例

$x*(a+b/d)<(-e+5)$，求该表达式的逆波兰式 pos()?

因为表达式中 < 的优先级最低，所以 ω 为 <，E1 为 x*(a+b/d)，E2 为 (-e+5)，所以得 pos(x*(a+b/d))pos((-e+5))<。pos(x*(a+b/d)) 中 * 最后运算，得 x pos((a+b/d))*；pos((-e+5)) 根据式②，得 pos(-e+5)。pos((a+b/d)) 中

+ 最后算，得 a pos(b/d)+，pos(b/d) 再得到 bd/。pos(-e+5) 里最后算的是 +，得 pos(-e)5+，在这里把-e 的-定义取负，是一个单目运算，于是 pos(-e) 得 e- 。最终得到如下所示逆波兰式。

$$
\begin{aligned}
&\because pos(x*(a+b/d)<(-e+5)) \\
&\quad = pos(x*(a+b/d))\ pos(-e+5)< \\
&\quad = x\ pos((a+b/d))\ *pos(-e+5)< \\
&\quad = x\ a\ pos((b/d))\ +*pos(-e+5)< \\
&\quad = x\ a\ bd/\ +*pos(-e+5)< \\
&\quad = x\ a\ bd/\ +*pos(-e)5+< \\
&\quad = x\ a\ bd/\ +*e-5+< \\
&\therefore pos() = xabd/+*e-5+<
\end{aligned}
$$

注

单目运算符的逆波兰式，$pos(-e) = e-$，
请大家和双目运算符加以区分

例 7.2 其它语法成分的逆波兰式设计示例

定义 7.1 单目运算“-”的逆波兰式

pos(-E) $\Longrightarrow$ pos(E)- 或 0 pos(E) -

定义 7.2 赋值语句的逆波兰式

v=E $\Longrightarrow$ v pos(E) =

例 7.3 x=(a+b)/-e*5，pos()?

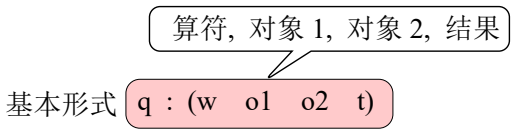
首先，= 优先级最低，得 x pos((a+b)/-e*5)=。pos((a+b)/-e*5) 里最后操作的是 *，得 pos((a+b)/-e)pos(5)*。接下来的过程省略，结果如图所示

$$
\begin{aligned}
&\because pos(x=(a+b)/-e*5) \\
&\quad = x\ pos((a+b)/-e*5)= \\
&\quad = x\ pos((a+b)/-e)pos(5)*= \\
&\quad = x\ pos(a+b)pos(-e)/\ 5*= \\
&\quad = xab+e-/\ 5*= \\
&\therefore pos() = xab+e-/\ 5*=
\end{aligned}
$$

四元式

逆波兰式更多是用来计算表达式，其它比如说 if 语句、while 语句不太容易用逆波兰式，因为不是简单的代数运算。还有很多标志符没有操作数，语义很复杂，这时候编译器更常用四元式。

1、表达式的四元式设计：



四元式如 q 所示，包括四个部分：算符、对象 1、对象 2 和结果，分别用 ω、o1、o2 和 t 表示。o1 和 o2 用 ω 进行计算，结果存到 t。

设 quat(E)，res(E) 分别为表达式 E 的四元式和结果变量。

① $quat(E_1, \omega E_2) = quat(E_1)$ $quat(E_2)$ $q:(\omega\ \ res(E_1)\ \ res(E_2)\ \ ti)$

② $quat((E)) = quat(E)$

③ $quat(i) =$ 空 ， $res(i) = i$

其中: ω(运算符); i 运算对象 (变量或常数)

规则①，求 E1ωE2 的四元式 quat(E1ωE2)，先写出 E1、E2 的四元式 quat(E1)、quat(E2)，结果分别是 res(E1)、res(E2)。这两个结果经过 ω 计算，最终放到一个变量 ti，ti 就是整个表达式的结果 res(E1ωE2)。

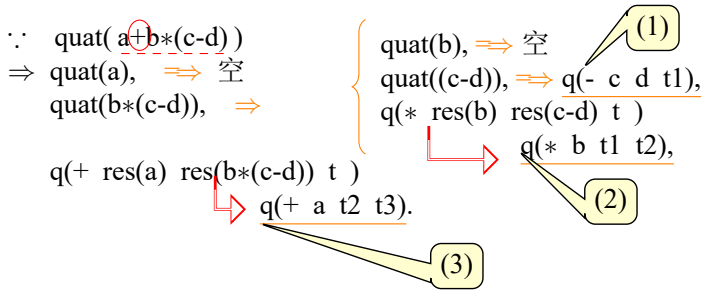
【注】

（1）方框内的三个定义式，为四元式翻译法则；

（2）定义式中的 ω 为 E1ωE2 中最后运算的算符！

例 7.4 表达式四元式翻译示例：

a+b*(c-d)



ω 是 +，E1 的四元式是 quat(a),E2 的四元式是 quat(b*(c-d))，最后一条语句是 q(+ res(a) res(b*(c-d)) t)，E1 的四元式是 res(a),E2 的四元式结果是 res(b*(c-d))，把它们做加法运算，将结果存到变量 t。

quat(a) 是空，quat(b*(c-d)) 需要根据准则继续求。ω 是 *，quat(b) 是空，quat((c-d)) 是 q(- c d t1), 对于 q(* res(b) res(c-d) t)，res(b) 是 b，res(c-d) 是 t1，将它们进行计算存在 t2，有 q(* b t1 t2)。

对于 q(+ res(a) res(b*(c-d)) t),res(a) 是 a，res(b*(c-d)) 是 t2，把结果放在 t3，有 q(+ a t2 t3)。

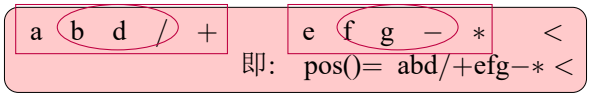
按最终结果的生成顺序，可得：

(1) (- c d t1)
(2) (* b t1 t2)
(3) (+ a t2 t3)

例 7.5 表达式逆波兰式和四元式最简翻译算法示例：

(a+b/d)<e*(f-g)，pos()?，Quat?

逆波兰式生成要点：运算对象顺序不变，运算符紧跟运算对象之后！则：



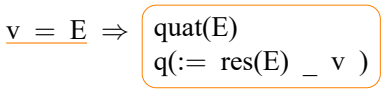
四元式生成要点：按照运算法则，依次生成四元式！则：

(1) (/ b d t1)
(2) (+ a t1 t2)
(3) (- f g t3)
(4) (* e t3 t4)
(5) (< t2 t4 t5)

四元式实际上和逆波兰式对应，先算 b/d，结果和 a 相加，然后算 (f-g)，结果和 e 相乘，最终把这两个结果 t2 和 t4 用 < 比较，比较的结果存到 t5。

2、赋值语句的四元式设计

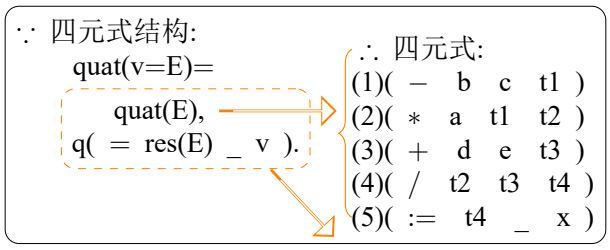
设有赋值语句：v=E



先算表达式右侧的 E，再进行赋值。

例 7.6 x=a*(b-c)/(d+e)

按照上面的规则，先求 quat(E)，把结果赋值给 v。Quat(E) 按计算的顺序写下来，最后用 t4 代替 res(E)。结果如下图所示：



3、转向语句与语句标号的四元式设计

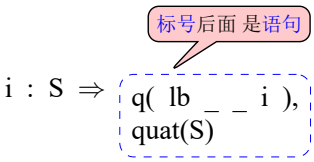
语句标号为转向语句提供转入语句标识，二者用标号相关联。

（1）设有转向语句：goto i

则有 goto i ⇒ q(gt _ _ i)

其中 gt 表示 goto

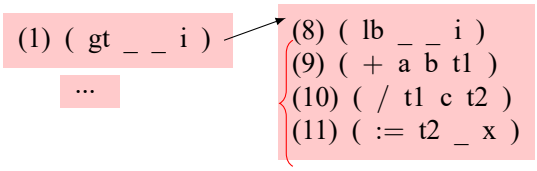
（2）设有标号语句：i：S



其中，i 表示标号，lb 表示 label，S 是后面的语句。

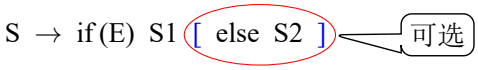
例 7.7 goto i；... i：x=(a+b)/c；

则有四元式序列

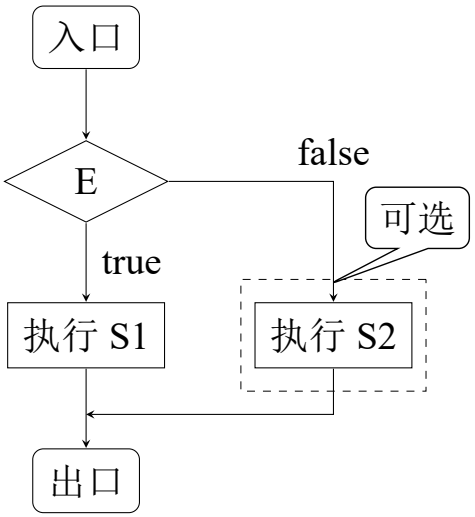


4、条件语句的四元式设计

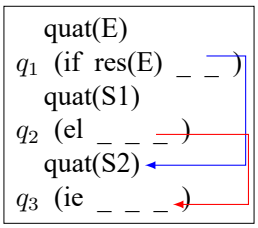
（1）文法：



（2）语义结构：从入口进，判断 E，为真则执行 S1，为假则执行 S2，S2 路径是可选的，最终都到达出口。



（3）四元式结构：



【注】

q1：当 res(E)=False，则转向 S2 入口四元式；

q2：无条件转向出口四元式；

q3：条件语句出口四元式。

首先执行的是 quat(E)，计算表达式 E 的结果，存到 res(E)。

$$q_1(\text{if res(E) \_ \_})$$

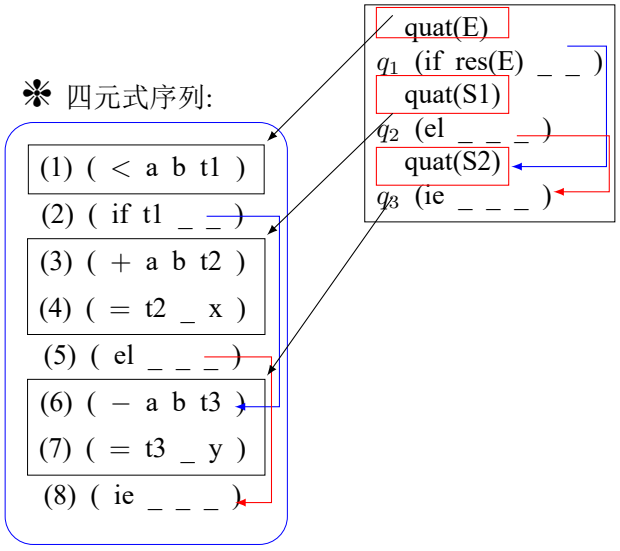
表示判断 res(E) 是否为假，若为假就跳转到 quat(S2)，若为真就顺序执行，执行 quat(S1)，执行完后跳转到出口。

例 7.8 条件语句四元式翻译示例：

if (a<b) x=a+b；

else y=a-b；

四元式序列：

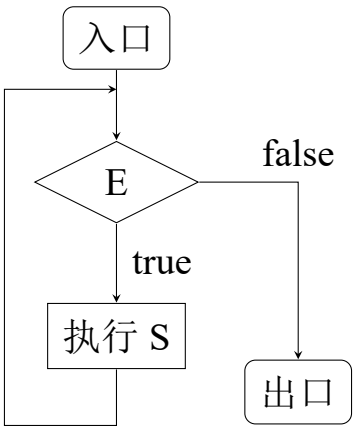


【注】如果语句中没有 else y=a-b 部分，则四元式结构中和四元式序列中的相应部分也不存在了！

5、循环语句的四元式设计：

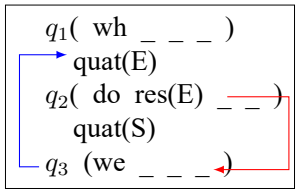
（1）文法：S → while (E) S

（2）语义结构：



逻辑是：while 语句先判断 E，若 E 为真就执行 S，再判断 E，还为真则继续执行 S，直到 E 为假则跳到出口。

（3）四元式结构：



一开始定义 q1，它是一个标识，表示 while 的入口。接下来计算 quat(E)，然后判断 res(E) 是否为假，为假则跳出循环，为真则执行 S，再通过语句 q3 返回计算 quat(E)。

【注】

q1：while 语句的入口四元式（提供转向 E 参照）；

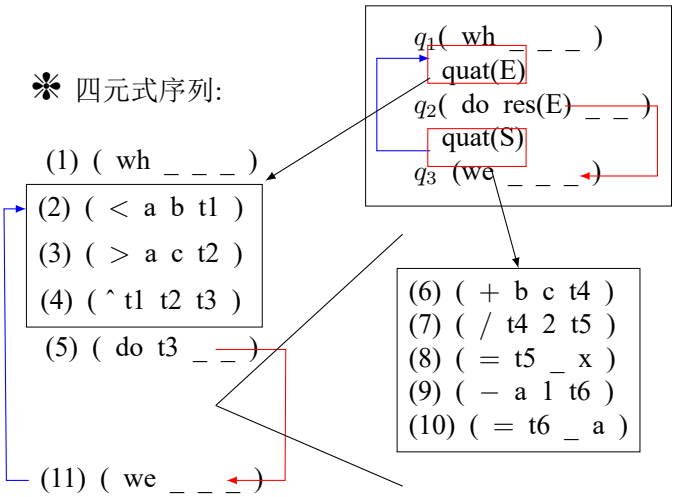
q2：当 res(E)=False 转向出口四元式；

q3：while 尾（兼循环转向 E）四元式。

例 7.9 循环语句四元式翻译示例：

while (a<b∧>c)

{x=(b+c)/2; a=a-1; }



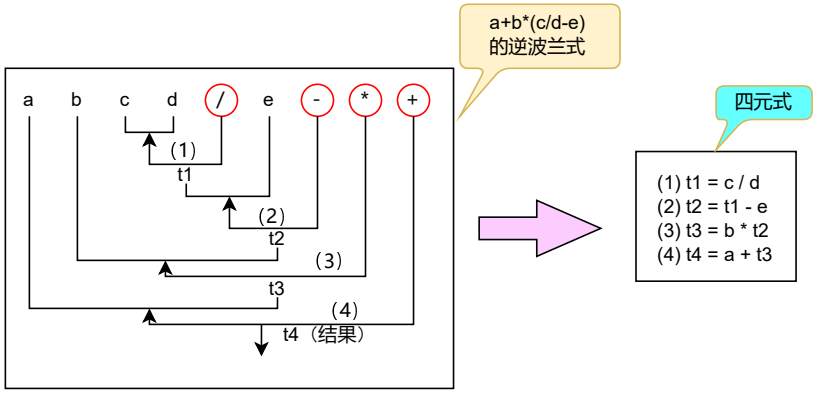
表达式 a<b∧a>c 用四元式 (2)(3)(4) 来表示，结果存到 t3。S 用四元式 (6)(7)(8)(9)(10) 表示，(6)(7)(8) 对应 x=(b+c)/2，(9)(10) 对应 a=a-1。最后是 q3，起到跳转的作用，跳转到四元式 (2)。

例 7.10 逆波兰式（四元式）计算过程示例：

【算法】设置一个“栈”，每当 NEXT(w)，重复执行：

（1）若 w= 运算对象，则压入栈中暂存：PUSH(w)；

（2）若 w= 运算符，则弹出栈顶对象计算之，并把结果压栈。



a+b*(c/d-e) 的逆波兰式是 abcd/e-*+，自左向右扫描，看到第一个运算符/，就把 c 和 d 进行运算，对应 t1，写成四元式 t1=c/d 。继续往后扫描，看到第二个运算符 - ，就把栈顶的两个元素 e 和 t1 做减法，结果存在 t2。以此类推，可以得到 t3、t4。

7.3 中间代码翻译算法

7.3.1 属性文法

定义 7.3 属性文法是上下文无关文法在语义上的扩展，可定义为如下三元组：

A=(G, V, E)；其中，G（文法），V（属性集），E（属性规则集）。

说明：

1. 属性：代表与文法符号相关的信息，这里主要指语义信息（类型、种类、值和值地址...）；文法产生式中的每个文法符号都附有若干个这样的属性。比如，描述一个人，身高、头发、颜色、出生地、年龄，都可以称为人的属性。文法符号也可以用若干个属性来描述，比如一个变量，整型/浮点型、值都是属性。
2. 属性可以进行计算和传递，属性规则就是在同一产生式中，相互关联的属性求值规则。比如，统计一个班级的总分，就是把所有同学的分数相加，如果把分数看成每一个人的属性，把所有分数相加就相当于对属性进行计算。
3. 属性分两类（按属性求值规则区分）：综合属性：其值由子女属性值来计算（自底向上求值）；继承属性：其值由父兄属性值来计算（自顶向下求值）。比如，对于继承属性，一般而言，姓氏是从父亲那“继承”的，该属性由父亲的属性决定。对于综合属性，父亲的属性需要用孩子的属性计算，以之前班级的例子为例，班级是学生一个更高层次的抽象，对于班级的成绩属性，需要把所有学生的成绩相加求平均来求。

例 7.11 属性文法构造示例：

算数表达式的属性文法；

设：X.val 为文法符号 X 的属性值；

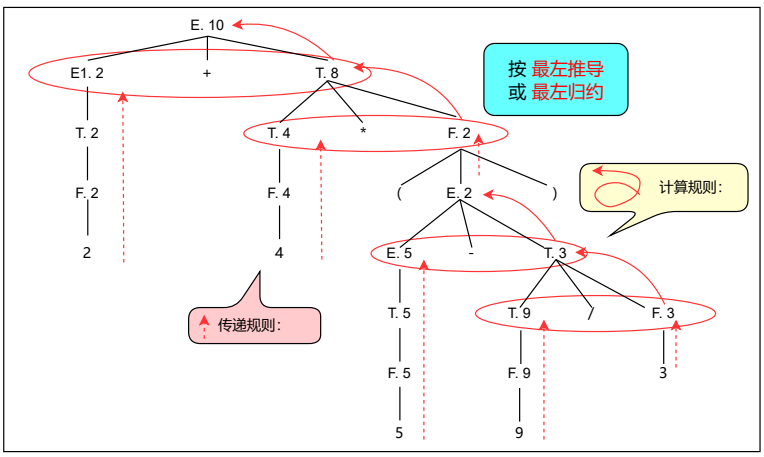
下述属性文法用于算术表达式的求值运算：

$$
\begin{aligned}
E &\rightarrow E^1 + T\ ; && \| E.val := E^1.val + T.val \\
E &\rightarrow E^1 - T\ ; && \| E.val := E^1.val - T.val \\
E &\rightarrow T\ ; && \| E.val := T.val \\
T &\rightarrow T^1 * F\ ; && \| T.val := T^1.val * F.val \\
T &\rightarrow T^1 / F\ ; && \| T.val := T^1.val / F.val \\
T &\rightarrow F\ ; && \| T.val := F.val \\
F &\rightarrow (E); && \| F.val := E.val \\
F &\rightarrow i; && \| F.val := i.val
\end{aligned}
$$

表的左侧是原始的文法，右侧是属性文法。对于 $E \rightarrow E^1 + T$ ，E^1 与 E 是不同的，其它加了上标 1 的符号同理。它对应的属性文法是右侧，将 E^1 的值加上 T 的值的结果赋给 E。-、*、/都类似。可以看出，该 X.val 属性是综合属性。对于表达式，E 相当于树的根，E^1、T 相当于树的叶子，父亲的属性值是由孩子的属性值决定的。

属性计算过程示例：

根据算术表达式属性文法及其相应的属性求值规则，2+4*(5-9/3) 的属性语法树：



如何算这个属性？基于最左推导或最左归约，传递规则是从下往上，比如 2 传给 F，F 传给 T，T 传给 E^1，然后还有相对复杂的计算，最后得到最上面的 E，也就是整个表达式对应的文法符号的值为 10。这个过程和之前描述的属性文法是一致的，父亲的属性值需要通过孩子进行计算，也就是说，如果得到了属性的值，就可以得到整个表达式的计算结果。

7.3.2 语法制导翻译技术

定义 7.4 语法制导（syntax_directed）
是指根据语言的形式文法对输入序列进行分析、翻译处理；核心技术是构造翻译文法——在源文法产生式中

插入语义动作符号（翻译子程序），借以指明属性文法中的属性求值时机和顺序。

语法制导翻译指在语法分析过程中，通过加入语义动作，来完成中间代码的生成，生成的过程称为翻译。语法制导强调所有定义、计算、建模，全都是基于语法的，在本编译原理教程中，语法是核心，决定程序的结构或系统的架构，把词法和中间代码生成连接起来。

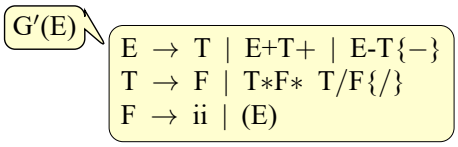
翻译文法可以看成是一种属性文法，带有属性的计算。语法制导的翻译，没有一个独立的程序来得到中间代码，中间代码的生成是贯穿在语法分析过程中的，语法分析执行完了，中间代码也就生成出来了。语法制导翻译技术由两部分构成：

语法分析技术 + 翻译文法构造技术

为叙述简便，除非临时详细指明，标识符 X 的属性一律视为符号表项指针，同时用 X 表示 X.point。接下来的大部分例子，将以算术表达式文法为例，探讨翻译文法的构造。

逆波兰式翻译文法构造示例：

例 7.12 算术表达式逆波兰式翻译文法



该文法和简单的算术表达式稍有区别，大括号里面的是语义动作，去掉及里面的内容，剩下的内容就是标准的算术表达式文法。a 的含义是输出 (a)。

符号串 a*(b+c) 的分析（翻译）过程（推导法）：

$$
\begin{aligned}
E &\Rightarrow T \Rightarrow T*F\{*\} \Rightarrow F*F\{*\} \Rightarrow a\{a\}*F\{*\} \Rightarrow a\{a\}*(E)\{*\} \\
&\Rightarrow a\{a\}*(E+T\{+\})\{*\} \Rightarrow a\{i\}*(T+T\{+\})\{*\} \\
&\overset{+}{\Rightarrow} a\{a\}*(b\{b\}+c\{c\}\{+\})\{*\}
\end{aligned}
$$

导出序列

用推导的方式，首先，E 用 T 来推导，先推导 * 乘法。然后是 T*F*，比原始文法多了 *，但还是按照文法内容完整推导出来。接下来用 $T \to F$ 把 T 推导成 F，得 $F*F\{*\}$。再将第一个 F 推导成 $a\{a\}$，第二个 F 推导成 (E)，E 就是 b+c。(E) 再推导成 $(E+T\{+\})$，需要带上 +。接下来的步骤类似。

分解导出序列：

去掉动作符号：　$a*(b+c)$.....源表达式;

去掉文法符号：　$abc+*$.....逆波兰式;

大括号 {} 是定义的一种动作，可以看作程序执行的一段代码，并不是文法真实的符号，代表的是程序要执行的一个操作。通过在语法中加入语义动作，在使用推导的方法进行语法分析后，就能得到一个动作序列，结果就是输出逆波兰式，达到生成中间代码的目的。

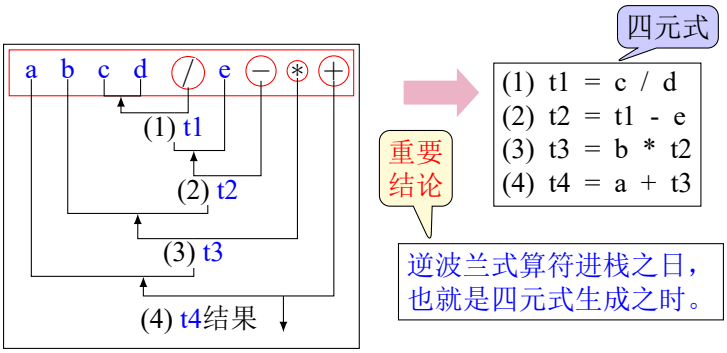
这个例子说明，通过改造文法，不需要额外地设计新的复杂程序，就能生成逆波兰式。

四元式翻译文法构造示例 1：

1、从逆波兰式的翻译到四元式的翻译：

假定：{a} 为 PUSH(a)，即把 a 压入栈（语义栈）中。

则算术表达式 a+b*(c/d-e) 的逆波兰式（生成）计算顺序：



往栈里依次压入元素 a、b、c、d，当遇到 / 时，就用栈顶的两个符号和 / 一起生成四元式 t1=c/d。同样，当遇到第二个运算符 - 时，就用栈顶符号 t1 和 e 生成第二个四元式。四元式（3）（4）同理。

{a} 代表了压栈操作，当遇到运算符，就用栈顶符号生成四元式，并把结果变量压回栈里。

四元式翻译文法构造示例 2：

2、算术表达式四元式翻译文法设计

定义：

SEM(m)——语义栈（属性传递、赋值场所），可以理解为一个指向符号表的指针，记录了所有关于这个标识符的信息；

QT[q]——四元式区，存储生成的四元式；

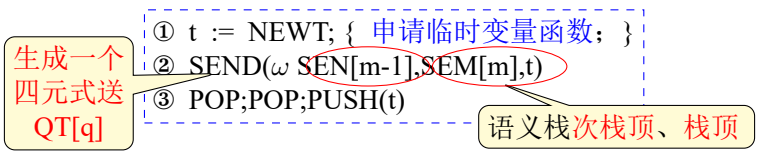
文法定义 $G''(E)$

$$E \to T \mid E+T\{GEQ(+)\} \mid E-T\{GEQ(-)\}$$
$$T \to F \mid T*F\{GEQ(*)\} \mid T/F\{GEQ(/)\}$$
$$F \to i\{PUSH(i)\} \mid (E)$$

其中：

（1）PUSH(i)——压栈函数（把当前 i 压入语义栈）；

（2）GEQ(ω)——表达式四元式生成函数：



有三个步骤：一，申请变量 t,；二，生成四元式，用 ω 进行计算，操作数是栈顶的两个元素，SEM[m-1]

是次栈顶元素，SEM[m] 是栈顶元素，把结果存在 t；三，分别是弹栈、弹栈、压栈，即把栈顶的两个元素出栈，再把结果压入栈。

动作函数（序列）执行过程示例：

算术表达式 a*(b/c-d) 的逆波兰式 abc/d-*，可得四元式动作序列：

动作序列	SEM[m]	QT[q]
PUSH(a) →	a	
PUSH(b) →	a b	
PUSH(c) →	a b c	
GEQ(/) →	a ~~b~~ ~~c~~	(1) (/ b c t1)
PUSH(d) →	a t1 d	
GEQ(-) →	a ~~t1~~ ~~d~~	(2) (- t1 d t2)
GEQ(∗) →	~~a~~ ~~t2~~	(3) (∗ a t2 t3)
	t3	

PUSH(a)、PUSH(b)、PUSH(c) 分别压栈，GEQ(/) 用栈顶两个元素 b 和 c 生成四元式，弹出 b 和 c 并把结果 t1 压栈。之后同理，最终得到结果 t3。

7.3.3 四元式翻译文法设计扩展 1

1、赋值语句四元式翻译文法：

S → vPUSH(v) = EASSI(=)；

赋值语句的表达式 v=E，它的翻译文法步骤是通过加入两个语义动作 PUSH(v) 和 ASSI(=)，其中，ASSI(=)——赋值函数，把栈顶元素赋值给次栈顶元素，然后把栈顶和次栈顶元素弹出，如下图。

$$(1)SEND(:= \ SEM[m], \_, SEM[m-1]);$$
$$(2)POP; POP;$$

2、标号、转向语句四元式翻译文法：

S → i PUSH (i) : LABEL(i) S；

S → goto i GOTO(i) ；

其中，LABEL(i)——标号函数，跳转到语义栈栈顶的元素所对应的那行代码，然后将栈顶元素弹出，如 (1)SEND(lb __ ,__ , SEM[m]);(2)pop; ，SEM[m] 是 i；GOTO(i)——转向函数，跳转到 i，如 (1)SEND(gt __ ,__ , i);

3、条件语句四元式翻译文法：

S → if(R) {IF(if)} S; [else {EL(el)}S] {IE(ie)} IF(if)——if 函数，用 if 来判断栈顶符号是否为假，SEM[m] 存的是 R 的结果；如图

$$(1)SEND(if \ SEM[m], \_, \_);$$
$$(2)POP;$$

EL(el)——else 函数，跳出 if；如图

$$(1)SEND(el\ \_,\_,\_);$$

IE(i)——ifend 函数，指示 if 的结束；如图

$$(1)SEND(ie\ \_,\_,\_);$$

循环语句四元式翻译文法：

S → while Wh() (R) DO(do) S WE(we)；

WH(wh)——循环头函数，标识 while 的开始；如图

$$(1)SEND(wh\ \_,\_,\_);$$

DO(el)——DO 函数，条件判断，判断 SEM[m] 栈顶元素的值是否为假，若为假则跳到函数尾，不为假则继续执行；如图

$$(1)SEND(do\ SEM[m],\_,\_);$$
$$(2)POP;$$

WE(we)——循环尾函数，表示条件的结束，还要再一次判断 R 的值是否为假，跳回到 while 的入口；如图

$$(1)SEND(we\ \_,\_,\_);$$

【注】文法中的 R 为关系表达式（见条件语句四元式翻译文法）

4、说明语句四元式翻译文法：

（1）

原文法：

$$D \to id:T;D \mid id:T$$
$$T \to interger \mid real \mid array[num]\ of\ T \mid \uparrow T$$

对该文法进行改造。

（2）

文法转换：

$$D \to id : T; D'$$
$$D' \to ; id : T D' \mid \varepsilon$$
$$T \to interger \mid real \mid array[num]\ of\ T \mid \uparrow T$$

（3）

翻译文法：

$$D \to \{INI()\} id \{NAME()\} : T \{ENT()\} D'$$
$$D' \to ; id \{NAME()\} : T \{ENT()\} D' \mid \varepsilon$$
$$T \to interger \{TYP(i)\} \mid$$
$$\to real \{TYP(r)\} \mid$$
$$\to array[num]\ of\ T \{TYP(a)\} \mid$$
$$\to \uparrow T \{TYP(\uparrow T)\}$$

INI()——初始化函数，在文法的开始，只能有一次。D 的产生式完成的是初始化的操作，而 D’ 不需要，这也是文法变换的一个目的，把第一个变换提取出来。

$$(1) offset := 0;$$

NAME()——标识符，名字处理函数，得到标识符对应的地址，存在该属性值；

$$(1) id.name := entry(id);$$

ENT()——填写符号表函数，根据名字、类型、偏移，把变量存入活动记录相应位置，同时 offset 要加上变量的长度；

$$(1) enter(id.name, T.type, offset);$$
$$(2) offset := offset + T.width;$$

TYP(x)——标识符类型处理函数，用 x 表示 T 的类型，x 的 width 表示 T 的 width；

$$T.type := x; \quad T.width := x.width;$$

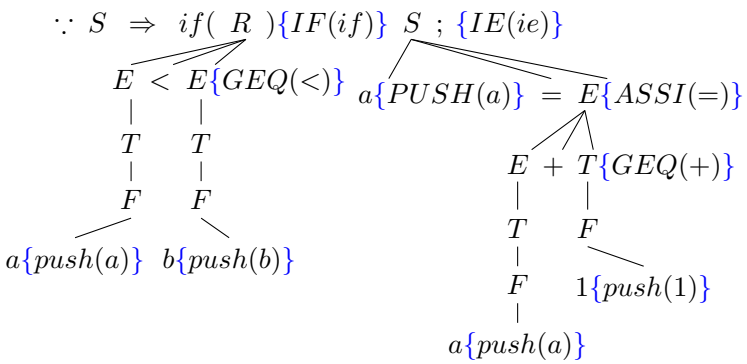
NUM()——标识符类型处理函数，将变量的值存到对应的 val 属性；

$$num.val := val(num);$$

翻译文法应用验证示例

例 7.13 已知条件语句 if(a<0) a=a+1;

（1）通过语法制导分析（属性推导树），写出动作序列：



所以 if(a<0) a=a+1 四元式翻译的动作序列依次是：push(a)，push(b)，GEQ(<)，IF(if)，PUSH(a)，push(a)，push(1)，GEQ(+)，ASSI(=)，IE(ie)

（2）根据上述条件语句四元式翻译的动作序列，if(a<0) a=a+1 的四元式生成过程：

动作序列	SEM[m]	QT[q]
PUSH(a) →	a	
PUSH(b) →	a b	
GEQ(<) →	a ~~b~~	(1) (< a b t1)
IF(if) →	~~t1~~	(2) (if t1 _ _)
PUSH(a) →	a	
PUSH(a) →	a a	
PUSH(1) →	a a 1	
GEQ(+) →	a ~~a~~ ~~1~~	(3) (+ a 1 t2)
ASSI(=) →	~~a~~ ~~t2~~	(4) (:= t2 _ a)
IE(ie) →		(3) (ie _ _ _)

7.4 中间代码翻译的实现

语法制导翻译的结构，可描述为：

扩展的**语法分析器** ⊕ **翻译文法**

语法分析器用文法对序列进行分析。翻译文法的设计，即在文法中加入语义动作，执行语法分析的过程同时执行翻译的过程。

1、自顶向下的翻译文法的要求：

（1）源文法应满足自顶向下分析要求（如 LL(1) 文法）；

（2）属性是自顶向下可求值的（属性计算不发生冲突）；

（3）动作符号可插入到产生式右部任何位置。

2、自底向上翻译文法的要求：

（1）源文法应满足自底向上分析要求（如 LR() 文法）；

（2）属性是自底向上可求值的（属性计算不发生冲突）；

（3）动作符号只能位于产生式的最右端。

7.4.1 递归子程序翻译法

例 7.14 算术表达式四元式翻译器的设计 1：

1、设置

语义栈：SEM[m]——暂存运算对象的属性值。

四元式区：QT[q]——四元式生成结果的存储区。

2、翻译文法设计

G1(E):

$$E \to T\{+T\{GEQ(+)\} \mid -T\{GEQ(-)\}\}$$
$$T \to F\{*F\{GEQ(*)\} \mid /T\{GEQ(/)\}\}$$
$$F \to i\{PUSH(i)\} \mid (E)$$

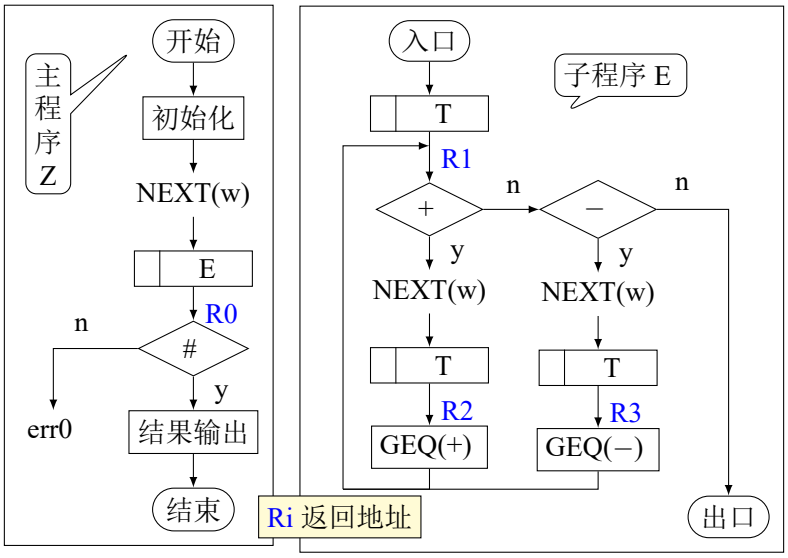
3、递归子程序的扩展

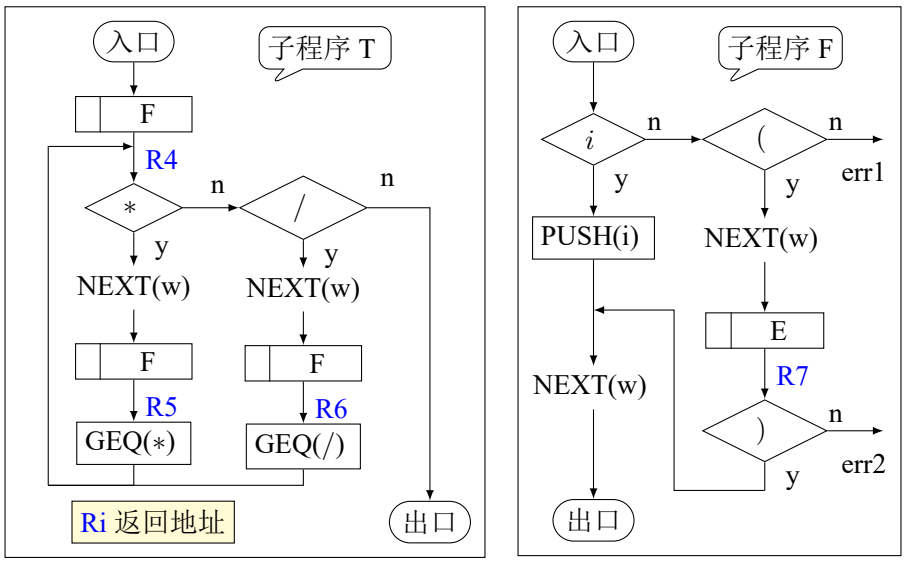
用递归子程序来分析文法，求四元式序列。要求：

①动作符号在哪，就在哪执行之；

②主控程序 (Z → E) 功能：初始化；结果输出。

算术表达式翻译文法的递归子程序：





算术表达式四元式翻译过程：

递归子程序调用算法为：入口时，把返回地址压入返回地址栈并进入；出口时，把返回地址弹出返回地址栈并返回。

设待翻译的表达式为 a*(b/c)#，执行递归子程序的过程如下：

首先，压入 Z，然后读取 w=a，调用子程序 E，进入返回地址 R0。在子程序 E 中，先压入 T，而在子程序中，又读入 F，返回 R4。在子程序 F，先判断是不是 i，是 i 则执行 PUSH(i)，把 i 压入到语义栈，然后再读取 w，跳出子程序 F，回到 R4。接下来的过程不再赘述，最终分析结束的时候，会生成两个四元式，如表所示。

递归子程序栈	返回地址栈	w	SEM[m]	QT[q]
Z E T F	R_0 R_1 R_4	a	a	
Z E T	R_0 R_1	*	a	
Z E T F	R_0 R_1 R_5	(	a	
Z E T F E T F	R_0 R_1 R_5 R_7 R_1 R_4	b	a b	
Z E T F E T	R_0 R_1 R_5 R_7 R_1	/	a b	
Z E T F E T F	R_0 R_1 R_5 R_7 R_1 R_6	c	a b c	
Z E T F E T	R_0 R_1 R_5 R_7 R_1	)	a b c	(1)(/b,c,t1)
Z E T F E	R_0 R_1 R_5 R_7	)	a t1	
Z E T F	R_0 R_1 R_5	)	a t1	
Z E T	R_0 R_1	#	a t1	(2)(*a,t1,t2)
Z E	R_0	#	t2	
Z		#	t2	
	Ok!			

7.4.2 LL(1) 翻译法

例 7.15 算术表达式四元式翻译器的设计 2：

1. 设置语法栈：SYN[n]；

   语义栈：SEM[m]；

   四元式区：QT[q]；

2. 翻译文法设计将例2.14的文法改造成右递归的形式

右递归改造文法

$$
\begin{aligned}
E &\rightarrow TE' \ (1)\\
E' &\rightarrow +T\{GEQ(+)\}\ E' \ (2) \mid -\ T\{GEQ(-)\}\ E' \ (3) \mid \varepsilon \ (4)\\
T &\rightarrow FT' \ (5)\\
T' &\rightarrow *F\{GEQ(*)\}T'(6) \mid /F\{GEQ(/)\}T'(7) \mid \varepsilon \ (8)\\
F &\rightarrow i\{PUSH(i)\} \ (9) \mid (E) \ (10)
\end{aligned}
$$

3、LL(1) 分析器的扩展

（1）当产生式（逆序）压栈是，动作符号也不例外；

（2）当动作符号位于栈顶时，执行之。

算术表达式四元式翻译过程：

首先得到 LL(1) 分析表：

\	i	+	-	*	/	(	)	#
E	①					①		
E’		②	③				④	④
T	⑤					⑤		
T’		⑧	⑧	⑥	⑦		⑧	⑧
F	⑨					⑩		

图 7.1: LL(1) 分析表

设待翻译的表达式为 a+b*c#，分析的过程如下：

栈底是 # 和 E，x 是栈顶符号 E，w 是当前要处理的符号 a。E 看到 a，是产生式①，所以 push(E)，把 E 产生式的右部逆序压栈，把 T 放到 x。T 遇到 a，把产生式 ⑤逆序压栈，此时栈顶符号是 F，F 遇到 a，把产生式 ⑨逆序压栈。压栈时，把 PUSH(i) 当成一个元素，即把 PUSH(a)、a 依次压栈。栈顶是 a，w 也是 a，做匹配操作，读取下一个符号 +。栈顶符号是 PUSH(a)，执行该语义动作，把 a 压入语义栈 SEM[m]，SYN[n] 的栈顶元素变为 T‘。把 T‘存到 x，T‘遇到 +，把产生式 ⑧逆序压栈，产生式 ⑧是空，所以不用压栈。接着是 E‘遇到 +，产生式②逆序压栈，把栈顶符号 + 存到 x，匹配 +。读取下一个符号 b，T 遇到 b，逆序压栈产生式 ⑤。接下来的过程不再赘述，最终 # 和 # 匹配，生成两个四元式，如下表。

SYN[n]	x	w	操作	SEM[m]	QT[q]
# ~~E~~	E	a	PUSH①^R		
#E’ ~~T~~	T	a	PUSH⑤^R		
#E’ T’ ~~F~~	F	a	PUSH⑨^R		
#E’ T’ PUSH(a)~~a~~	a	a	NEXT(W)		
#E’ T’ ~~PUSH(a)~~ →		+		a	
#E’ ~~T~~	T’	+	PUSH⑧^R	a	
# ~~E~~	E’	+	PUSH②^R	a	
#E’ GEQ(+) T ~~+~~	+	+	NEXT(W)	a	
#E’ GEQ(+) ~~T~~	T	b	PUSH⑤^R	a	
#E’ GEQ(+) T’ ~~F~~	F	b	PUSH⑨^R	a	

SYN[n]	x	w	操作	SEM[m]	QT[q]
#E' GEQ(+) T' PUSH(b) b	b	b	NEXT(w)	a	
#E' GEQ(+) T' PUSH(b)		*		ab	
#E' GEQ(+) T'	T'	*	PUSH⑥^R	ab	
#E' GEQ(+) T' GEQ(*) F *	*	*	NEXT(w)	ab	
#E' GEQ(+) T' GEQ(*) F	F	c	PUSH⑨^R	ab	
#E' GEQ(+) T' GEQ(*) PUSH(c) c	c	c	NEXT(w)	ab	
#E' GEQ(+) T' GEQ(*) PUSH(c)		#		abc	
#E' GEQ(+) T' GEQ(*)		#		a b c	(1)(*b,c,t1)
#E' GEQ(+) T'	T'	#		at1	
#E' GEQ(+)		#		a t1	(2)(+a,t1,t2)
# E'	E'	#		t2	
#		#	ok		

接上页

这个过程和语法分析是类似的，不同的是加入了一些语义动作，当栈顶是语义动作时，就执行相应的动作。

7.4.3 LR() 翻译法

例 7.16 算术表达式四元式翻译器的设计 3：

1、设置

语法栈：SYN[n]；

语义栈：SEM[m]；

四元式区：QT[q]；

2、

翻译文法设计（带有状态编码）

$$
\begin{aligned}
Z &\rightarrow E_1\ (0)\\
E &\rightarrow E_2\ +_3\ T_4\{GEQ(+)\}\ (1)\ |T_5\ (2)\\
T &\rightarrow T_6\ *_7\ F_8\{GEQ(*)\}\ (3)|\ F_9\ (4)\\
F &\rightarrow i_{10}\{PUSH(i)\}\ (5)\ |\ (_{11}\ E_{12}\ )_{13}\ (6)
\end{aligned}
$$

注意，自底向上翻译文法的语义动作只能在产生式的最右端。

3、LR() 分析表的扩展

LR() 分析表中的 r(i) 执行下述两种操作：

①首先执行动作符号（翻译函数）；

②然后执行归约操作（按产生式 i 归约）。

SLR(1) 分析表的构造：

＼	i	+	*	(	)	#	E	T	F
0	i10			(11			E1,2	T5,6	F9
1,2		+3				OK			
3	i10			（11				T4,6	F9
4,6		r(1)	*7		r(1)	r(1)			
5,6		r(2)	*7		r(2)	r(2)			
7	i10			（11					F8
8	r(3)	r(3)	r(3)	r(3)	r(3)				
9	r(4)	r(4)	r(4)	r(4)	r(4)				
10	r(5)	r(5)	r(5)	r(5)	r(5)				
11	i10			(11			E1,2	T5,6	F9
12,2		+3			)13				
13	r(6)	r(6)	r(6)	r(6)	r(6)				

算术表达式四元式翻译过程：设待翻译的表达式为 a*b+c#，分析过程如下：首先把 0 状态压栈，0 状态看到 a 到 10 状态，压入 a10。10 状态看到 #，执行 r(5)，归约产生式（5）。归约的时候，先执行 PUSH(a)，把 a 压入语义栈，再归约 a，压入 F9。9 状态看到 *，归约产生式（4），归约成 T。0 状态看到 T，是 T5,6 状态，5,6 状态是一起的，是确定化之后得到的。T5,6 状态看到 * 到 7 状态，随后的过程与前面类似，这里不再赘述，最终得到整个分析过程。

SYN[n]		w	动作符号	SEM[m]	QT[q]
$\#_0$	0	a			
$\#_0\ a_{10}$	10	*	PUSH(a)	a	
$\#_0\ F_9$	9	*		a	
$\#_0\ T_{5,6}$	5,6	*		a	
$\#_0\ T_{5,6}\ *_7$	7	b		a	
$\#_0\ T_{5,6}\ *_7\ b_{10}$	10	+	PUSH(b)	ab	
$\#_0\ T_{5,6}\ *_7\ F_8$	8	+	GEQ(*)	~~a b~~	(1) (* a b t1)
$\#_0\ T_{5,6}$	5,6	+		$t1$	
$\#_0\ E_{1,2}$	1,2	+		$t1$	
$\#_0\ E_{1,2}\ +_3$	3	c		$t1$	
$\#_0\ E_{1,2}\ +_3\ c_{10}$	10	#	PUSH(c)	$t1c$	
$\#_0\ E_{1,2}\ +_3\ F_9$	9	#		$t1c$	
$\#_0\ E_{1,2}\ +_3\ T_{4,6}$	4,6	#	GEQ(+)	~~t1 c~~	(2)(+ t1 c t2)
$\#_0\ E_{1,2}$	1,2	#		结束	

7.4.4 算符优先翻译法

例 7.17 算术表达式四元式翻译器的设计 4：

1、设置

语法栈：SYN[n]；

语义栈：SEM[m]；

四元式区：QT[q]；

2、翻译文法设计

翻译文法：

$$
\begin{aligned}
E &\rightarrow E + T\{GEQ(+)\} \mid T \\
T &\rightarrow T * F\{GEQ(*)\} \mid F \\
F &\rightarrow i\{PUSH(i)\} \mid E
\end{aligned}
$$

1、算符优先分析器的扩展

（1）归约时，先执行语义动作，后归约；

（2）归约时，只要产生式右部的终结符排列和栈顶 $<\bullet \ldots\ldots \bullet>$ 中的终结符匹配上即可；

（3）归约后，得到的非终结符不必入栈。

（4）算术表达式四元式翻译过程：

＼	+	*	i	(	)	#
+	>	<	<	<	>	>
*	>	>	<	<	>	>
i	>	>			>	>
(	<	<	<	<	=	>
)	>	>			>	>
#	<	<	<	<	<	OK

图 7.2: 优先级图

设待翻译的表达式为 a+b*c#，分析过程为：首先压入 #，如果关系是 <，就移进，如果是 > 就归约，用归约，先执行 PUSH(a)，再归约成 F。剩下过程不再赘述，只需注意归约时要加上语义动作。

SYN[n]	关系	W	操作	SEM[m]	QT[q]
#	<	a	移进，读		
# < a	>	+	规约，不读	a	
#	<	+	移进，读		
# < +	<	b	移进，读		
# < + < b	>	*	规约，不读	a b	
# < +	<	*	移进，读		
# < + < *	<	c	移进，读		
# < + < * < c	>	#	规约，不读	a b c	
# < + < *	>	#	规约，不读	a t1	(*,b,c,t1)
# < +	>	#	规约，不读	t2	(+,a,t1,t2)
#		#	OK		

7.4.5 翻译文法的变换问题

自底向上中间代码翻译，要求翻译文法的动作符号必须位于产生式的最右端。不满足此条件的属性翻译文法，需要通过文法变换来解决。

1. 赋值语句的属性翻译文法：

G(S)：

S → vPUSh(v) = EASSI(=);

改造文法，令 Sa → vPUSh(v)，

则有：

G‘(S)：

S → Sa = EASSI(=)；
S → vPUSh(v)；

2. 标号、转向语句属性翻译文法：

G(S)：

S → i PUSH(i) : LABEL(i) S；
S → goto i GOTO(i)；

令 Si → i PUSH(i)

SI → Si : LABEL(i) 则有：

G‘(S)：

S → SI S；
SI → Si : LABEL(i)
Si → i PUSH(i)

3. 分支语句的翻译文法

G(S)：

S → if(R) IF(if)S；
[else El(el)S] IE(ie)

令 Sif → if(R) IF(if)

则有：

G’(S)：

S → Sif S；[Sel S] IE(ie)
S → if(R) IF(if)

4. 循环语句的翻译文法

G(S)：

S → whileWH() (R) DO(do) S WE(we)

令 Swh → whileWH() Sdo → (R) DO(do)

则有：

G’(S)：

S → Swh Sdo SWE(we)
Swh → whileWH()