编译原理

第八章语法制导翻译和中间代码生成

8.1 属性文法(Attribute Grammar)

8.2 语法制导翻译(Syntax-directed translation)

8.3 中间代码

8.4--8.8 一些语句的翻译

- 属性文法：包含一个上下文无关文法和一系列语义规则，这些语义规则附在文法的每个产生式上。
- 语法制导翻译：指在语法分析过程中，完成附加在所使用的产生式上的语义规则描述的动作。

属性文法

- 属性文法(attribute grammar)是一个三元组: A=(G,V,F),其中

G:是一个上下文无关文法

V:有穷的属性集,每个属性与文法的一个终结符或非终结符相连，这些属性代表与文法符号相关信息，如它的类型、值、代码序列、符号表内容等等。属性与变量一样，可以进行计算和传递。属性加工的过程即是语义处理的过程。

F:关于属性的断言或一组属性的计算规则(称为语义规则) . 语义规则与一个产生式相联，只引用该产生式左端或右端的终结符或非终结符相联的属性.

$E \rightarrow T^1 + T^2 \{T^1.t = \text{int AND } T^2.t = \text{int}\}$

$E \rightarrow T^1 \text{ or } T^2 \{T^1.t = \text{bool AND } T^2.t = \text{bool}\}$

$T \rightarrow \text{num } \{T.t := \text{int}\}$

$T \rightarrow \text{true } \{T.t := \text{bool}\}$

$T \rightarrow \text{false } \{T.t := \text{bool}\}$

图 8.2　类型检查的属性文法

- 每个产生式$A \rightarrow \alpha$都有一套与之相关联的语义规则，每条规则的形式为$b:=f(c_1,c_2 \ldots c_k)$，f是函数，b和$c_1,c_2 \ldots c_k$是该产生式文法符号的属性。

  （1）如果b是A的一个属性，并且$c_1,c_2 \ldots c_k$是产生式右边文法符号的属性或A的其他属性，则称b是A的综合属性;(Synthesized attribute)

  （2）如果b是产生式右边某个文法符号X的一个属性，并且$c_1,c_2 \ldots c_k$是A或产生式右边任何文法符号的属性，则称b是文法符号X的继承属性。(Inherited attribute)

- **综合属性用于“自下而上”传递信息，而继承属性用于“自上而下”传递信息。**

语法制导翻译实现

- 语法制导翻译即基于属性文法的处理过程通常是这样的：对单词符号串进行语法分析，构造语法分析树，然后根据需要遍历语法树并在语法树的各结点处按语义规则进行计算。
- 语义规则的计算可能产生代码、在符号表中存放信息、给出错误信息或执行其他动作。对输入符号串的翻译就是根据语义规则进行计算的结果。

→输入符号串

→ 语法分析树

→ 属性依赖图

→ 语义规则的计算顺序

依赖图(Dependency graph)

依赖图是一个**有向图**，用来描述分析树中的属性之间的相互**依赖关系**。

依赖图的构造算法：

for分析树中每一个结点n **do**

for 结点的文法符号的每一个属性a **do**

为a在依赖图中建立一个结点；

for 分析树中每一个结点n **do**

for结点n所用产生式对应的每一个语义规则

$b:=f(c_1,c_2,\ldots c_k)$ **do**

for i :=1 to k **do**

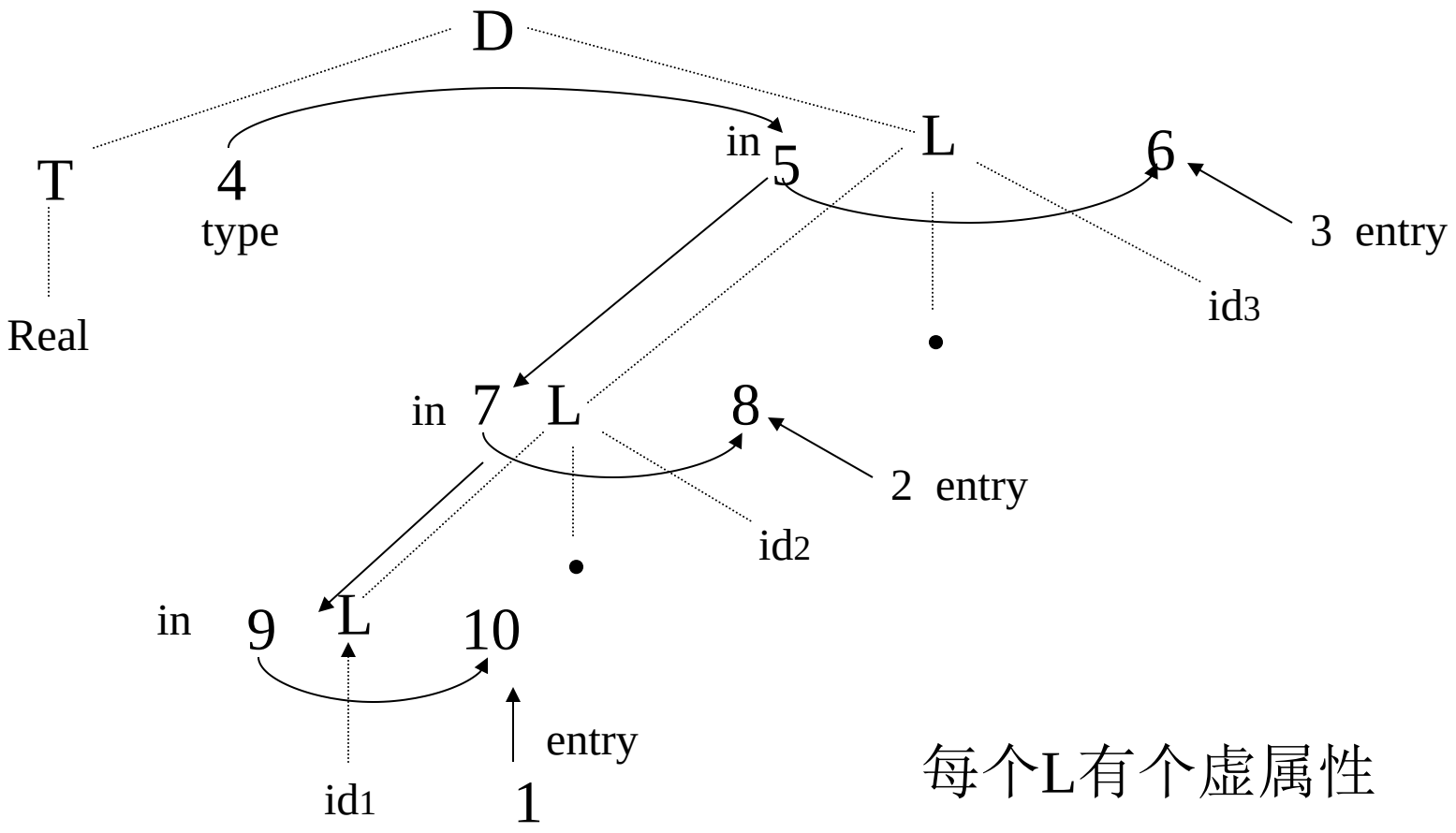
从c_i结点到b结点构造一条有向边

依赖图----例8.2

例8.2 **继承属性L.in**

产生 式	语 义 规 则
D →TL	L.in:=T.type
T→ int	T.type=integer
T →real	T.type:=real
L → L1,id	L1.in:=L.in addtype(id.entry,L.in)
L→ id	addtype(id.entry,L.in)

例8.2 **Real id1,id2,id3**分析树的依赖图



每个L有个虚属性

从依赖图的拓扑排序中，可以得到计算语义规则的顺序。用这个顺序来计算语义规则就得到输入符号串的翻译。

例8.2Real id1,id2,id3分析树的依赖图

每一条边都是从序号较低的结点指向序号较高的结点。历此，依赖图的一个拓扑排序可以从低序号到高序号顺序写出。从这个拓扑排序中我们可以得到下列程序，用a_n来代表依赖图中与序号n的结点有关的属性：

a_4: = real

a_5: = a_4

(a6) addtype (id_3.entry, a_5);

a_7: = a_5;

(a8) addtype (id_2.entry, a_7)

a_9: = a_7

(a10) addtype (id_1.entry, a_9)

这些语义规则的计算将把real类型填入到每个标识符对应的符号表项中。

属性计算方法

- 树遍历的属性计算方法
  设语法树已经建立起了，并且树中已带有开始符号的继承属性和终结符的综合属性。然后以某种次序遍历语法树，直至计算出所有属性。最常用的遍历方法是深度优先，从左到右的遍历方法。如果需要的话，可使用多次遍历（或称遍）。
- 一遍扫描的处理方法

  与树遍历的属性计算文法不同，一遍扫描的处理方法是在语法分析的同时计算属性值，而不是语法分析构造语法树之后进行属性的计算，而且无需构造实际的语法树。

- 在某些情况下可用一遍扫描实现属性文法的语义规则计算。也就是说在语法分析的同时完成语义规则的计算，无须明显地构造语法树或构造属性之间的依赖图。因为单遍实现对于编译效率非常重要。

具体的实现希望在单遍扫描中完成翻译

怎样实现这种翻译器？一个一般的属性文法的翻译器可能是很难建立的，然而有一大类属性文法的翻译器是很容易建立的

S-属性文法　适用于自底向上的计算

L-属性文法　适用于自顶向下的分析，也可用于自底向上。

S-属性文法是L-属性文法的一个特例。

- 用一遍扫描的编译程序模型来理解语法制导的翻译方法：
- 为文法中每个产生式配上一组语义规则，并且在语法分析的同时执行这些语义规则。
- 在自上而下的语法分析中，若一个产生式匹配输入串成功（对非终结符推导一次）
- 自下而上分析中，当一个产生式被用于进行归约时，此产生式相应的语义规则就被计算，完成相关的语义分析和代码产生等工作。在这种情况下，语法分析工作和语义规则的计算是穿插进行的。

S－属性文法的自下而上计算

S－属性文法，它只含有综合属性。

- 综合属性可以在分析输入符号串的同时自下而上来计算。分析器可以保存与栈中文法符号有关的综合属性值，每当进行归约时，新的属性值就由栈中正在归约的产生式右边符号的属性值来计算。

- S－属性文法的翻译器通常可借助于LR分析器实现。在S－属性文法的基础上，LR分析器可以改造为一个翻译器，在对输入串进行语法分析的同时对属性进行计算。

S_m	y. Val	y
S_{m-1}	x. Val	x
⋮	⋮	⋮
S_0	—	#
状态栈	语义值栈	符号栈

图 8.6　扩充的分析栈

产生式	语义规则
0）L→E	print（E.val）
1）E→E¹＋T	E.val：＝E¹.val＋T.val
2）E→T	E.val：＝T.val
3）T→T¹*F	T.val：＝T¹.val×F.val
4）T→F	T.val：＝F.val
5）F→（E）	F.val：＝E.val
6）F→digit	F.val:＝digit.lexval

LR分析器可以改造为一个翻译器，在对输入串进行语法分析的同时对属性进行计算。

LR分析器**增加语义栈**

２＋３*５的分析和计值过程

步骤	归约动作	状态栈	语义栈(值栈)	符号栈	留余输入串
1)		0	—	#	2+3*5#
2)		05	——	#2	+3*5#
3)	r_6	03	—2	#F	+3*5#
4)	r_4	02	—2	#T	+3*5#
5)	r_2	01	—2	#E	+3*5#
6)		016	—2—	#E+	3*5#
7)		0165	—2——	#E+3	*5#
8)	r_6	0163	—2—3	#E+F	*5#
9)	r_4	0169	—2—3	#E+T	*5#
10)		01697	—2—3—	#E+T*.	5#
11)		016975	—2—3——	#E+T*5	#
12)	r_6	01697(10)	—2—3—5	#E+T*F	#
13)	r_3	0169	—2—(15)	#E+T	#
14)	r_1	01	—(17)	#E	#
15)	接受				

L－属性文法和自顶向下翻译

一个属性文法称为L－属性文法，如果对于每个产生式$A \rightarrow X_1X_2 \ldots X_n$，其每个语义规则中的每个属性或者是综合属性，或者是X_j（$1 \leq j \leq n$）的一个继承属性且这个继承属性仅依赖于：

（1）X_j左边的符号X_1，X_2，…，X_{j-1}的属性；

（2）A的继承属性。

S－属性文法一定是L－属性文法，因为（1）、（2）限制只用于继承属性。

L－属性文法允许一次遍历就计算出所有属性值。

LL（1）这种自上而下分析文法的分析过程，从概念上说可以看成是深度优先建立语法树的过程，因此，我们可以在自上而下语法分析的同时实现L属性文法的计算。

例 8.3　将中缀表达式翻译成相应的后缀表达式的属性文法，其中 addop 表示＋或－。

$E \rightarrow E$ addop T print(addop. Lexeme) | T　　$T \rightarrow$ num print(num. val)

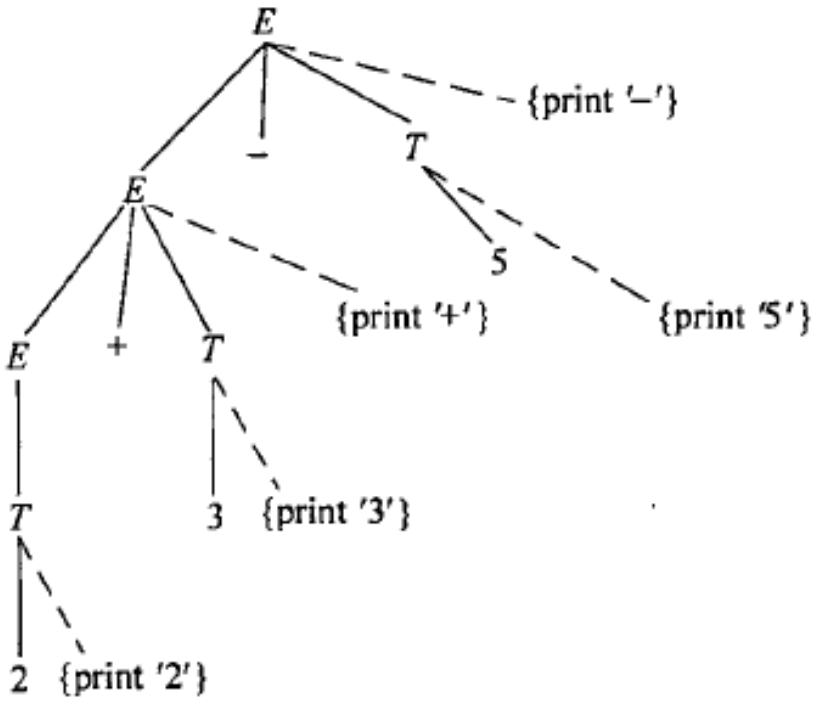


图 8.8　说明语义动作的语法树(1)

LR分析，属性计算

打印：23+5-

自上而下语法分析：

语义动作不是附在产生式右部末尾，而是T和R之间

例（中缀表达式翻译成相应的后缀表达式）

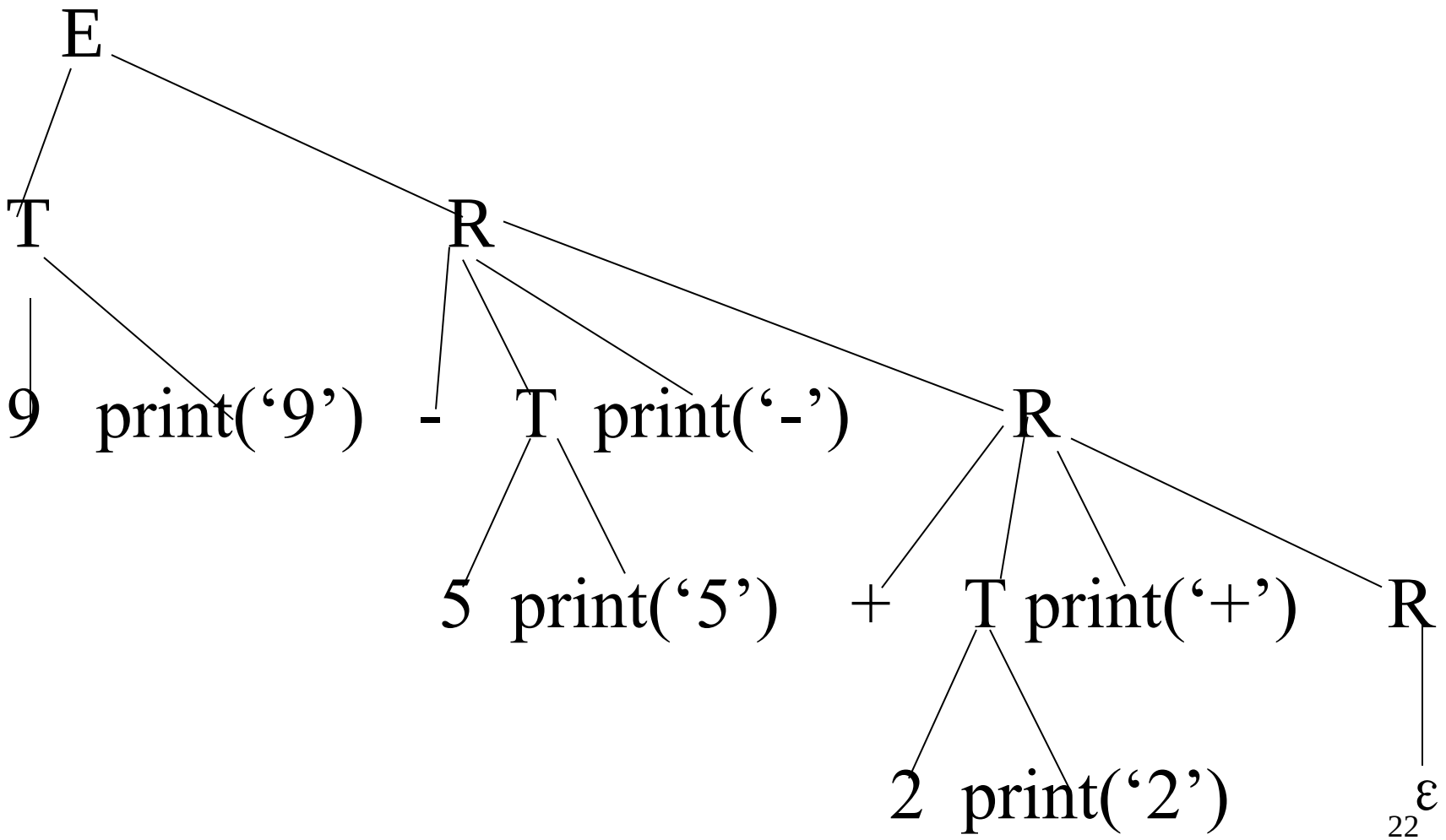
E→TR

R→addop T {print(addop. Lexeme)} R_1| ε

T→num {print(num.val)}

翻译模式（Translation schemes）适合语法制导翻译的另一种描述形式。翻译模式给出了使用语义规则进行计算的次序，可把某些实现细节表示出来。在翻译模式中，和文法符号相关的属性和语义规则（这里我们也称语义动作），用花括号｛｝括起来，插入到产生式右部的合适位置上。

输入串9－5＋2的语法树，每个语义动作都作为相应产生式左部符号的结点的儿子，按深度优先次序执行图中的动作后，打印输出95－2+。



转换左递归翻译模式的方法

- 左递归翻译方案

$A \rightarrow A_1 Y$

$\{ A.a = g(A_1.a, Y.y) \}$

$A \rightarrow X$

$\{ A.a = f(X.x) \}$

  - 假设每个文法符号有一个综合属性，用相应的小写字母表示，g 和 f 是任意函数

- 消除左递归之后，文法转换成

$A \rightarrow XR$

$R \rightarrow YR \mid \varepsilon$

- 消除左递归的翻译方案

$A \rightarrow X \quad \{ R.i = f(X.x) \}$

$\quad R \quad \{ A.a = R.s \}$

$R \rightarrow Y \quad \{ R_1.i = g(R.i, Y.y) \}$

$\quad R_1 \quad \{ R.s = R_1.s \}$

$R \rightarrow \varepsilon \quad \{ R.s = R.i \}$

  - 经过转换的翻译方案中使用了R的继承属性 i 和综合属性 s。

左递归文法翻译方案的转换

例　把带左递归的文法的翻译方案转换成不带左递归的文法的翻译方案。

$E \rightarrow E_1+T$ { 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 }

$T \rightarrow (E)$ { T.val = E.val }

$T \rightarrow num$ { T.val = num.val }

带左递归的文法的翻译方案

经过转换的不带有左递归文法的翻译方案

$E \rightarrow T$ { R.i = T.val }
R { E.val = R.s }

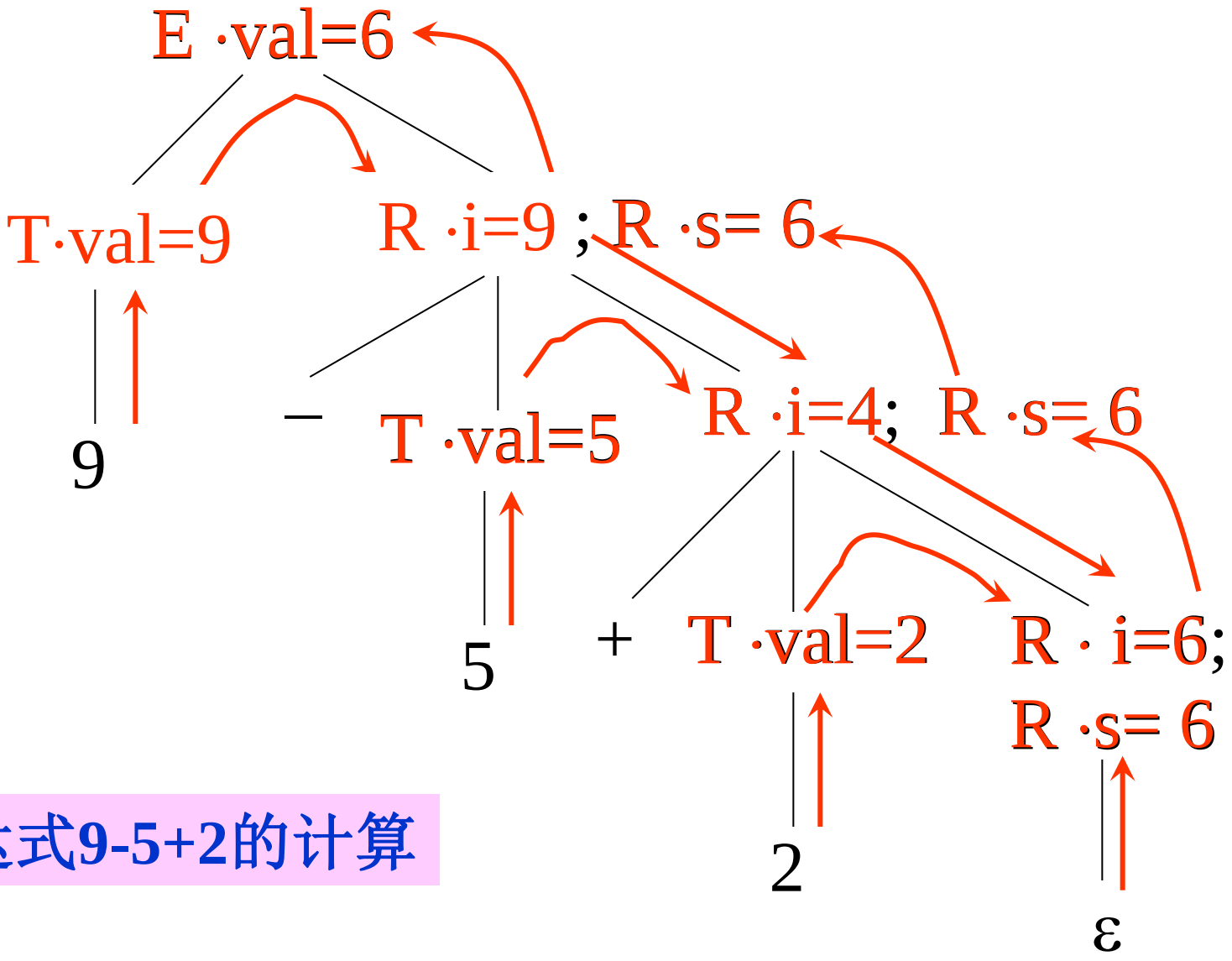
$R \rightarrow +$
T { R_1.i = R.i +T.val }
R_1 { R.s = R_1.s }

$R \rightarrow -$
T { R_1.i = R.i - T.val }
R_1 { R.s = R_1.s }

$R \rightarrow \varepsilon$ { R.s = R.i }

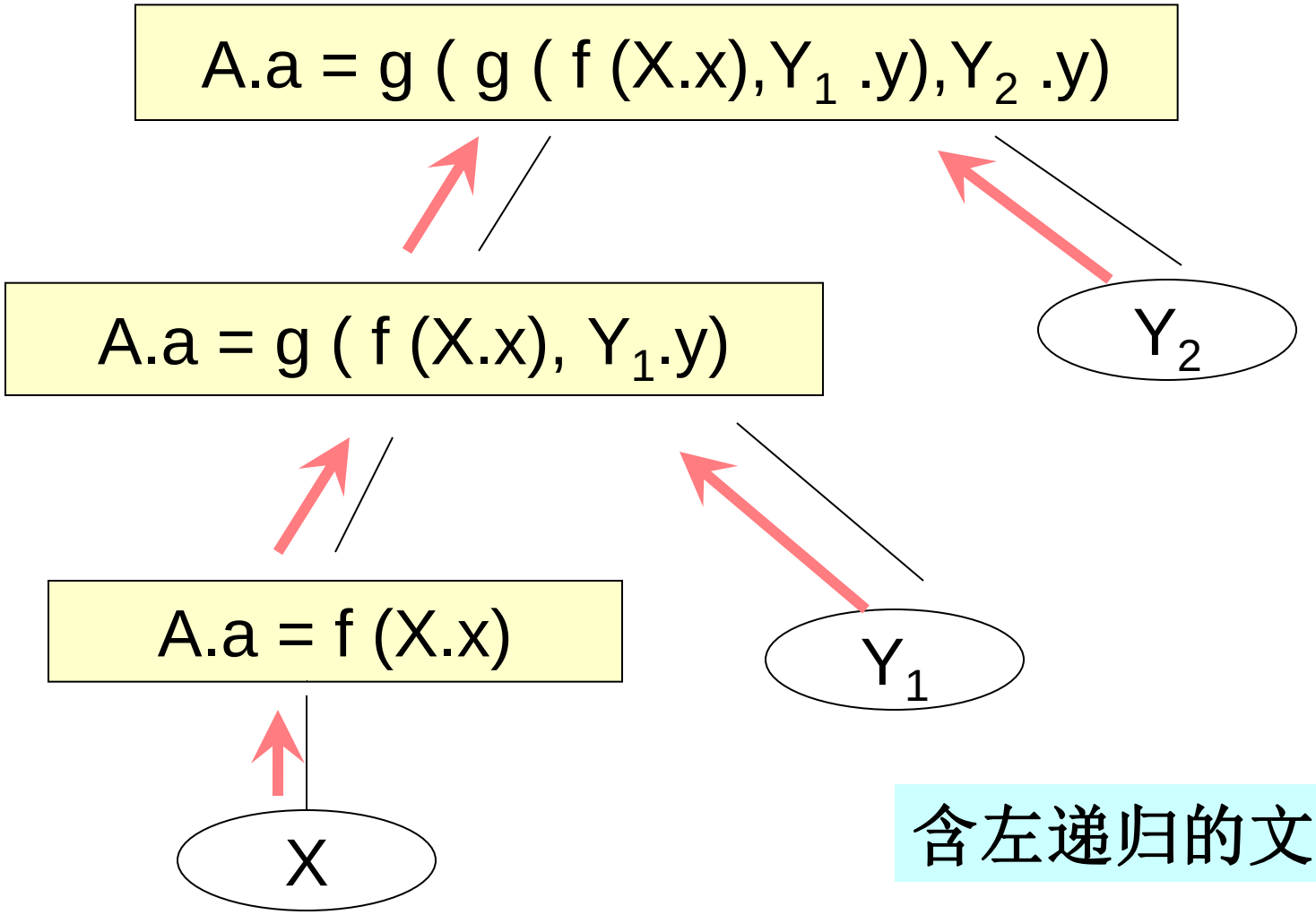
$T \rightarrow (E)$ { T.val = E.val }

T→num { T.val = num.val }

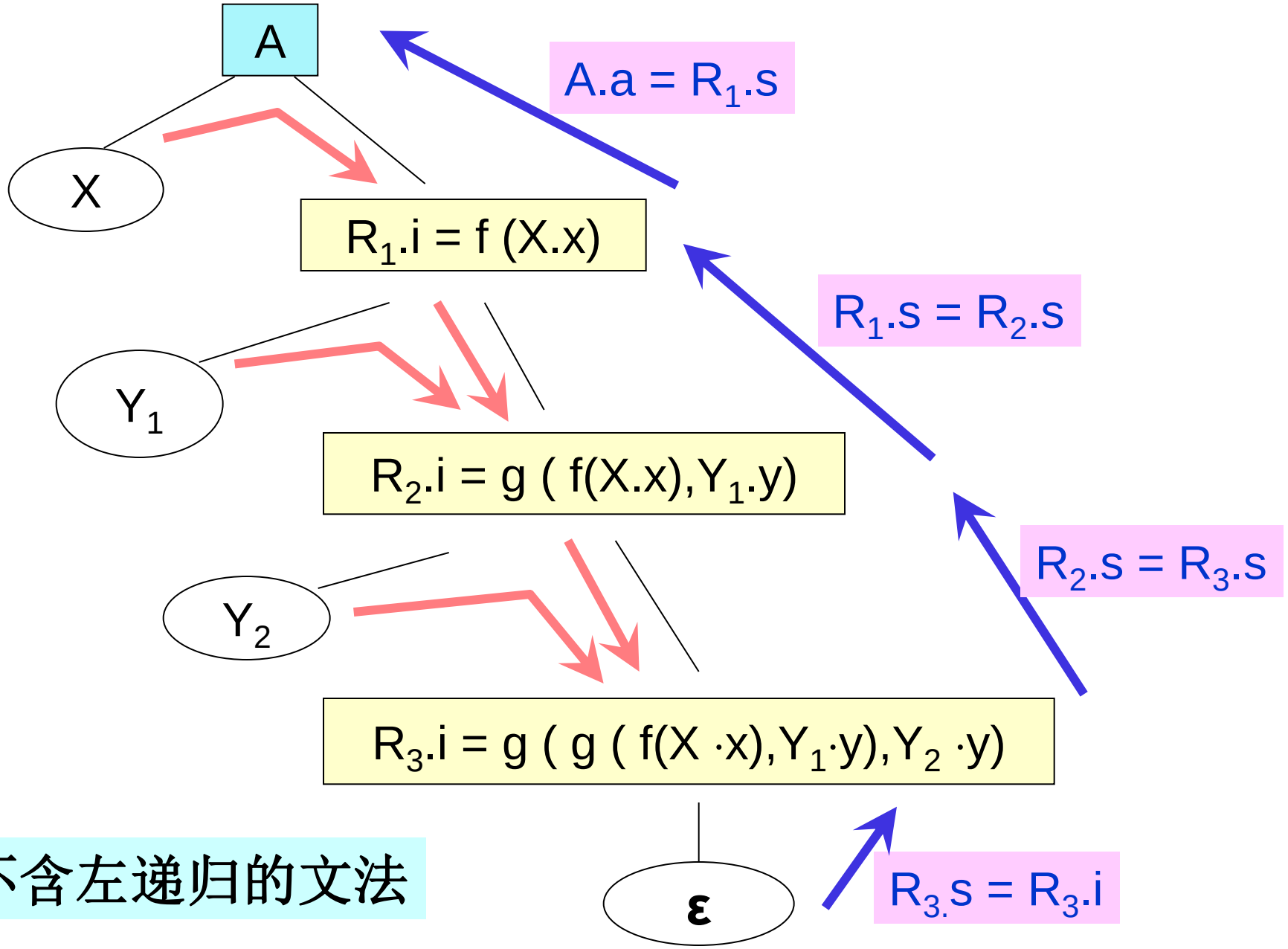


表达式9-5+2的计算

输入：XY_1Y_2



含左递归的文法


A
X
$R_1.i = f(X.x)$
Y_1
$R_2.i = g(f(X.x), Y_1.y)$
Y_2
$R_3.i = g(g(f(X \cdot x), Y_1 \cdot y), Y_2 \cdot y)$
ε
$A.a = R_1.s$
$R_1.s = R_2.s$
$R_2.s = R_3.s$
$R_3.s = R_3.i$

不含左递归的文法

自下而上计算继承属性

- 讨论在自下而上的分析过程中实现L－属性文法的方法。这种方法是S-属性文法的自下而上翻译技术的一般化。
- 如何实现自下而上计算继承属性？
- （1）从翻译模式中去掉嵌入在产生式中间的动作；
- （2）用综合属性代替继承属性

从翻译模式中去掉嵌入在产生式中间的动作，用一个非终结符代替。

- P176

$$E \to T \quad R$$
$$R \to + T \quad \{\text{print}(\text{'+'})\} R_1$$
$$R \to - T \quad \{\text{print}(\text{'-'})\} R_1$$
$$R \to \varepsilon$$
$$T \to \text{num}\{\text{print}(\text{num}, \text{val})\}$$

引入M和N，嵌入在产生式中间的动作都在后面了。

$$E \to T \quad R$$
$$R \to + \quad T \quad MR_1$$
$$R \to - \quad TNR_1$$
$$R \to \varepsilon$$
$$T \to \text{num}\{\text{print}(\text{num. val})\}$$
$$M \to \varepsilon\{\text{print}(\text{'+'})\}$$
$$N \to \varepsilon\{\text{print}(\text{'-'})\}$$

用综合属性代替继承属性

有时，改变基础文法可能避免继承属性。例如，一个Pascal的说明由一标识符序列后跟类型组成，如，m，n: integer。这样的说明的文法可由下面形式的产生式构成

D→L：T （左边L 的类型属性要继承右边T 的类型属性）

T→integer|char

L→L, id|id

如果非终结符L从第一个产生式中它的右边T中继承了类型，则我们得到的属性文法就不是L－属性的，因此，基于这个属性文法的翻译工作不能在语法分析的同时进行。

一个解决的方法是重新构造文法

使类型作为标识符表的最后一个元素：

D→id L　{addtype(id.entry, L.type)}

L→, id L1{L.type:=L1.type;addtype(id.entry,L.type)}

　　|: T {L.type := T.type}

T→integer {T.type:=int} |char{T.type:=ch}

属性文法 P177

这样，类型可以通过综合属性L.type进行传递，当通过L产生每个标识符时，它的类型就可以填入到符号表中。

语义分析

- 语义分析
- 属性文法和语法制导翻译方法和技术应用于语义分析中。

语义分析

通常包括：

（1）类型检查。验证程序中执行的每个操作是否遵守语言的类型系统的过程.，编译程序必须报告不符合类型系统的信息。

（2）控制流检查。控制流语句必须使控制转移到合法的地方。例如，在C语言中break语句使控制跳离包括该语句的最小while、for或switch语句。如果不存在包括它的这样的语句，则就报错。

（3）一致性检查。在很多场合要求对象只能被定义一次。例如Pascal语言规定同一标识符在一个分程序中只能被说明一次，同一case语句的标号不能相同，枚举类型的元素不能重复出现等等。

（4）相关名字检查。有时，同一名字必须出现两次或多次。例如，Ada 语言程序中，循环或程序块可以有一个名字，出现在这些结构的开头和结尾，编译程序必须检查这两个地方用的名字是相同的。

(5)名字的作用域分析

中间代码

- 何谓中间代码 **(Intermediate code)**
  (Intermediate representation)
  (Intermediate language)

  源程序的一种内部表示，不依赖目标机的结构，易于机械生成目标代码的中间表示。
- 为什麽要此阶段

  逻辑结构清楚；利于不同目标机上实现同一种语言；

  利于进行与机器无关的优化；这些内部形式也能用于解释。
- 中间代码的几种形式

  逆波兰　四元式　三元式

逆波兰记号

- 逆波兰记号：最简单的一种中间代码表示。最早用于表示算术表达式，现在也可以扩充到表达式以外的范围。
- 特点：把运算对象写在前面，运算符号写在后面。(后缀式)
- 图8.11 程序设计语言中的逆波兰表示

程序设计语言中的表示	逆波兰表示
$a+b$	$ab+$
$a+b*c$	$abc*+$
$(a+b)*c$	$ab+c*$
$a:=b*c+b*d$	$abc*bd*+:=$

三元式和树形表示

- 把表达式及各种语句表示成一组三元式。每个三元式由三部分组成：算符、第一运算对象、第二运算对象
- P178 例：a:=b*c+b*d

(1)	(*	*b*,	*c*)
(2)	(*	*b*,	*d*)
(3)	(+	(1),	(2))
(4)	(:=	(3),	*a*)

四元式

- 是一种普遍采用的中间代码形式。
- 有四个组成成分：算符、第一运算对象、第二运算对象、运算结果
- P179 例：a:=b*c+b*d

(1) (* , b, c, t_1)

(2) (* , b, d, t_2)

(3) (+, t_1, t_2, t_3)

(4) (:=, t_3, —, a)

例：A + B * (C - D) + E / (C - D) ^N
分别用逆波兰、三元式、四元式表示。

例：A + B * (C - D) + E / (C - D) ^N

逆波兰：　A B C D - * + E C D – N ^ / +

四元式：　(1)　(-　C　D　T1)

(2)　(*　B　T1　T2)

(3)　(+　A　T2　T3)

(4)　(-　C　D　T4)

(5)　(^　T4　N　T5)

(6)　(/　E　T5　T6)

(7)　(+　T3　T6　T7)

例： A + B * (C - D) + E / (C - D) ^N

三元式:

(1) (- C D)

(2) (* B (1))

(3) (+ A (2))

(4) (- C D)

(5) (^ (4) N)

(6) (/ E (5))

(7) (+ (3) (6))

简单赋值语句的（四元式）翻译

四元式形式　：　(op ,arg1,arg2,result) 或

result := arg1　op　arg2

语义属性：　id.name：id表示的单词，作为变量

E.Place：存放E值的变量名在符号表的登录项

函数：　lookup(id.name) ;检查是否出现在符号表中

过程：　emit(t := arg1 op　arg2);输出四元式到文件

newtemp;生成临时变量

产生式和语义描述：

(1) S →id := E

{ P:=lookup　(id.name)　;

if P ≠ nil　then emit(P　“　:=”　E.place)

else　error　}

(2) $E \rightarrow E^1 + E^2$

{E.place:= newtemp;

emit(E.place":=" E^1.place"+"E^2.place)}

(3) $E \rightarrow -E^1$

{ E.place:=newtemp;

emit(E.place":="""uminus" E^1.place)}

(4) $E \rightarrow (E^1)$

{ E.place:= E^1.place}

(5) $E \rightarrow$ id

{E.place:=newtemp;

P:=lookup(id.name);

if P≠nil then E.place:=P

else error}

- a := b * (- c + d)
- 程序执行过程

- 在赋值语句中增加类型检查和类型转换 图8.13

产生式	语义动作

$E \rightarrow E^1 * E^2$	{E. place := newtemp;
	if E^1. type = int AND E^2. type = int then
	begin emit (E. place, ':=', E^1. place, '*', E^2. place);
	E. type := int
	end
	else if E^1. type = real AND E^2. type = real then
	begin emit (E. place, ':=', E^1. place, '*', E^2. place);
	E. type := real
	end
	else if E^1. type = int /* and E^2. type = real */ then
	begin t := newtemp;
	emit(t, ':=', 'itr', E^1. place);
	emit(E. place, ':=', t, '*', E^2. place);
	E. type := real
	end
	else /* E^1. type = real and E^2. type = int */
	begin t := newtemp;
	emit(t, ':=', 'itr', E^2. place);
	emit(E. place, ':=', E^1. place, '*', t);
	E. type := real
	end;
	}

简单说明句的翻译-翻译是指在符号表中登录定义的名字和性质。

最简单的说明句的语法：

D →integer〈namelist〉 | real〈namelist〉

〈namelist〉→〈namelist〉，id | id

以上文法（自下而上分析）需先规约为namelist，一起登记进符号表

用自下而上翻译 文法改写：可以及时地把性质告诉每个id，不用成批登记

D →D[1]，id

| integer id

| real id

把名字id和性质A登陆在名表中

(1) D →integer id { enter(id,int）;D.att:=int}

(2) D →real id { enter(id,real）;D.att:=real}

(3) D → D^1 ,id { enter(id， D^1.att）；

D.att:= D^1.att }

示例：Real a，b， c

布尔表达式的翻译

- 数值表示布尔式的翻译方案

$E \rightarrow E^1$ **or** E^2	{E.place := newtemp; emit(E.place ' := ' E^1.place ' **or** ' E^2.place)}
$E \rightarrow E^1$ **and** E^2	{E.place := newtemp; emit(E.place ' := ' E^1.place ' **and** ' E^2.place)}
$E \rightarrow$ **not** E^1	{E.place := newtemp; emit (E.place ' := ' **not** ' E^1.place)}
$E \rightarrow (E^1)$	{E.place := E^1.place}

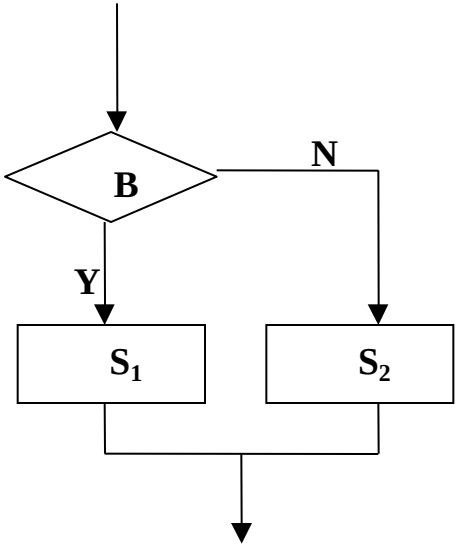
$E \rightarrow id_1$ rop id_2	{E.place := newtemp;
	emit('if' id_1.place 'rop' id_2.place ' goto ' nextstat+3);
	emit(E.place ' := ' '0');
	emit(' goto ' nextstat+2);
	emit(E.place ' := ' '1')}

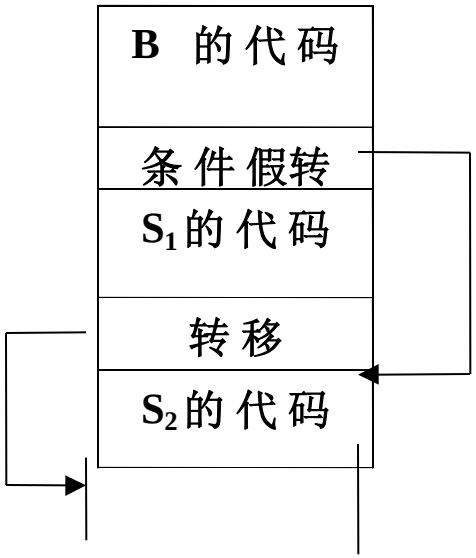
$E \rightarrow$ true	{E.place := newtemp; emit(E.place ' := ' '1')}
$E \rightarrow$ false	({E.place := newtemp; emit(E.place ' := ' '0')}

- a > b or c < d

控制语句的翻译–结构的翻译

如 if B then S1 else S_2





if $a<b$ **or** $c<d$ **and** $e>f$ then S^1 else S^2 的四元式序列为

(1) if $a<b$ goto (7)　　/* (7)是整个布尔表达式的真出口 */

(2) goto (3)

(3) if $c<d$ goto (5)

(4) goto (p+1)　　/* (p+1)是整个布尔表达式的假出口 */

(5) if $e>f$ goto (7)

(6) goto (p+1)

(7) (关于 S^1 的四元式)

⋮

(p) goto (q)

(p+1) (关于 S^2 的四元式)

⋮

(q)

练习

- P203 3、4、5