第七章 语法制导翻译和中间代码生成

学习目标：

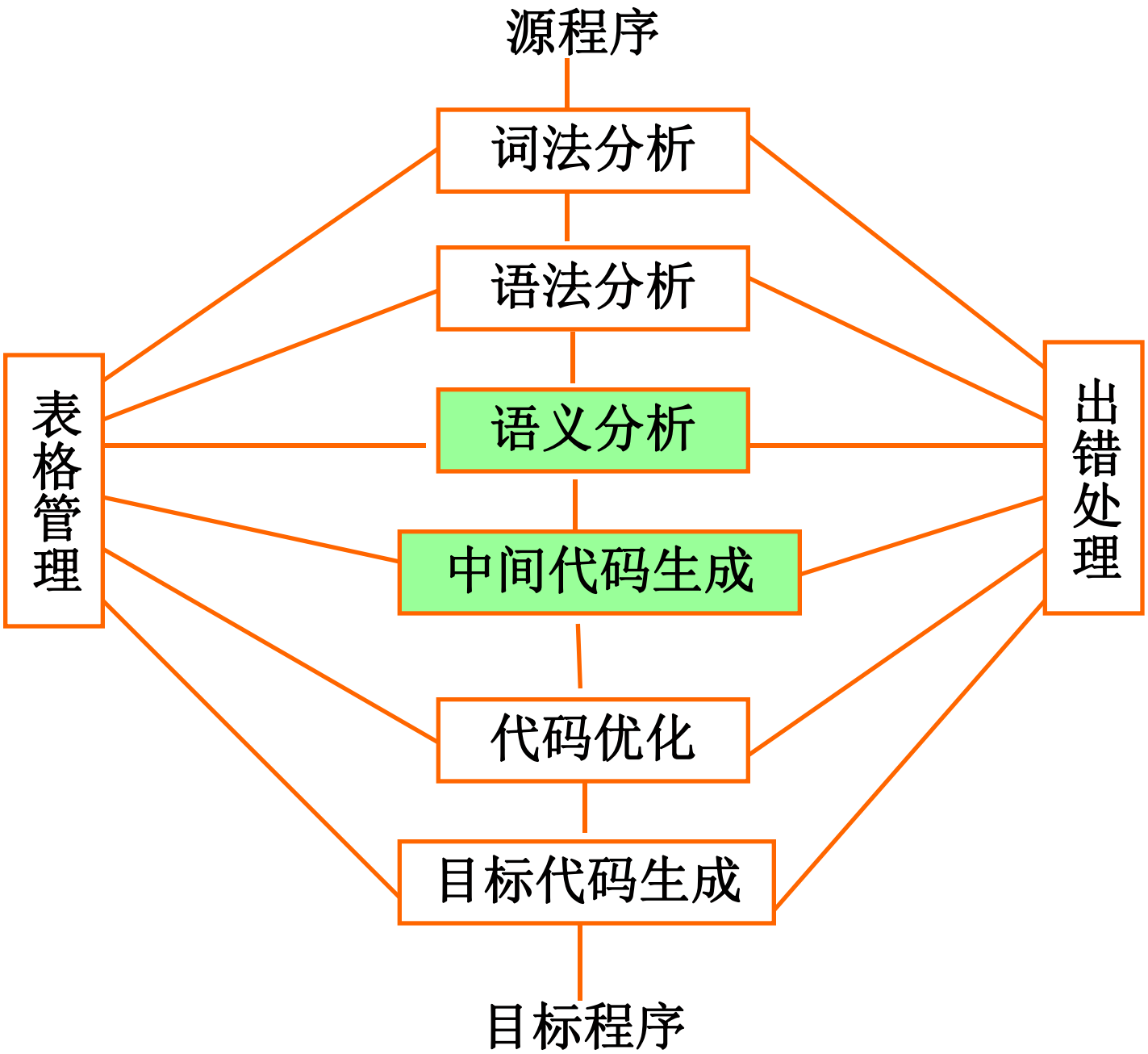
- 掌握：

  常见语法成分的中间代码形式；

  常见语法成分的属性文法或翻译方案

- 理解：

  属性文法、语法制导翻译方法


源程序
词法分析
语法分析
语义分析
中间代码生成
代码优化
目标代码生成
目标程序
表格管理
出错处理

语义分析基础

- 语义分析的内容
- 主要是类型相容检查，有以下几种：

1) 各种条件表达式的类型是不是boolean型？
2) 运算符的分量类型是否相容？
3) 赋值语句的左右部的类型是否相容？
4) 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) 函数说明中的函数类型和返回值的类型是否一致？

- 其它语义检查：

1) V[E]中的V是不是变量，而且是数组类型？
2) V.i中的V是不是变量，而且是记录类型？i是不是该记录的域名？
3) x+f(…)中的f是不是函数名？形参个数和实参个数是否一致？
4) 每个使用性标识符是否都有声明？有无标识符的重复声明？

- 在语义分析同时产生中间代码，在这种模式下，语义分析的主要功能如下：
  - 语义审查
  - 在扫描声明部分时构造标识符的符号表
  - 在扫描语句部分时产生中间代码
- 语义分析方法

语法制导翻译方法

使用属性文法为工具来说明程序设计语言的语义。

7.1 属性文法(Attribute Grammar)

- 属性

  对文法的每一个符号，引进一些属性，这些属性代表与文法符号相关的信息，如类型、值、存储位置等。

- 语义规则

  为文法的每一个产生式配备的计算属性的计算规则，称为语义规则。

- 属性文法是带属性的一种文法

  它的主要思想：

  - 首先对于每个文法符号引进相关的属性符号；
  - 其次对于每个产生式写出计算属性值的语义规则

- 属性文法的形式定义

  一个属性文法是一个三元组，A＝(G， V， F)

  - G是一个上下文无关文法；
  - V是属性的有穷集；
  - F是关于属性的断言的有穷集。

说明：

1. 每个属性与文法符号相联，N.t表示文法符号N的属性t。属性值又称语义值。存储属性值的变量又称语义变量。
2. 每个断言与文法的某个产生式相联，写在{ }内。属性的断言又称语义规则，它所描述的工作可以包括属性计算、静态语义检查、符号表的操作、代码生成等，有时写成函数或过程段。

例 完成类型检查的属性文法

1) $E \rightarrow T^1 + T^2$ {$T^1.t = int$ AND $T^2.t = int$}
2) $E \rightarrow T^1$ or T^2 {$T^1.t = bool$ AND $T^2.t = bool$}
3) $T \rightarrow num$ {$T.t := int$}
4) $T \rightarrow true$ {$T.t := bool$}
5) $T \rightarrow false$ {$T.t := bool$}

- 属性的分类：

1. 综合属性：
   - 从语法树的角度来看，如果一个结点的某一属性值是由该结点的子结点的属性值计算来的，则称该属性为综合属性。
   - 内在属性是综合属性。
   - 用于“自下而上”传递信息

2. 继承属性

- 从语法树的角度来看，若一个结点的某一属性值是由该结点的兄弟结点和（或）父结点的属性值计算来的，则称该属性为继承属性。
- 用于“自上而下”传递信息

说明：

- 终结符只有综合属性，它们由词法分析器提供
- 非终结符既有综合属性也有继承属性，但文法开始符没有继承属性

例 简单算术表达式求值的属性文法

1) L→E　　{ Print(E.val) }
2) E→E^1+T　　{ E.val :＝E^1.val +T.val }
3) E→T　　{ E.val :＝T.val }
4) T→T^1*F　　{ T.val :＝T^1.val * F.val }
5) T→F　　{ T.val :＝F.val }
6) F→(E)　　{ F.val :＝E.val }
7) F→digit　　{ F.val :＝digit.lexval }

E.val、T.val、F.val都是综合属性

终结符digit只有综合属性，它的值由词法分析提供

例 描述变量类型说明的属性文法

1) D→TL　　{ L.in:＝T.type }

2) T→int　　{ T.type:＝int }

3) T→real　　{ T.type:＝real }

4) L→L^1,id　　{ L^1.in:＝L.in;
addtype(id.entry, L.in)}

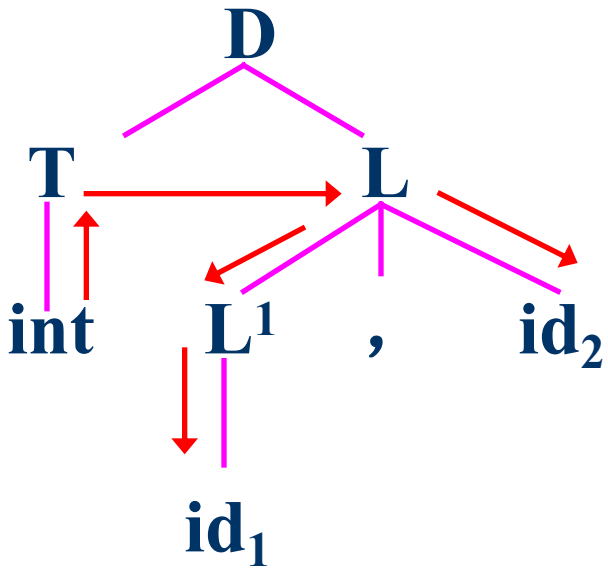
5) L→id　　{ addtype(id.entry, L.in)}

L.in是继承属性

T.type是综合属性

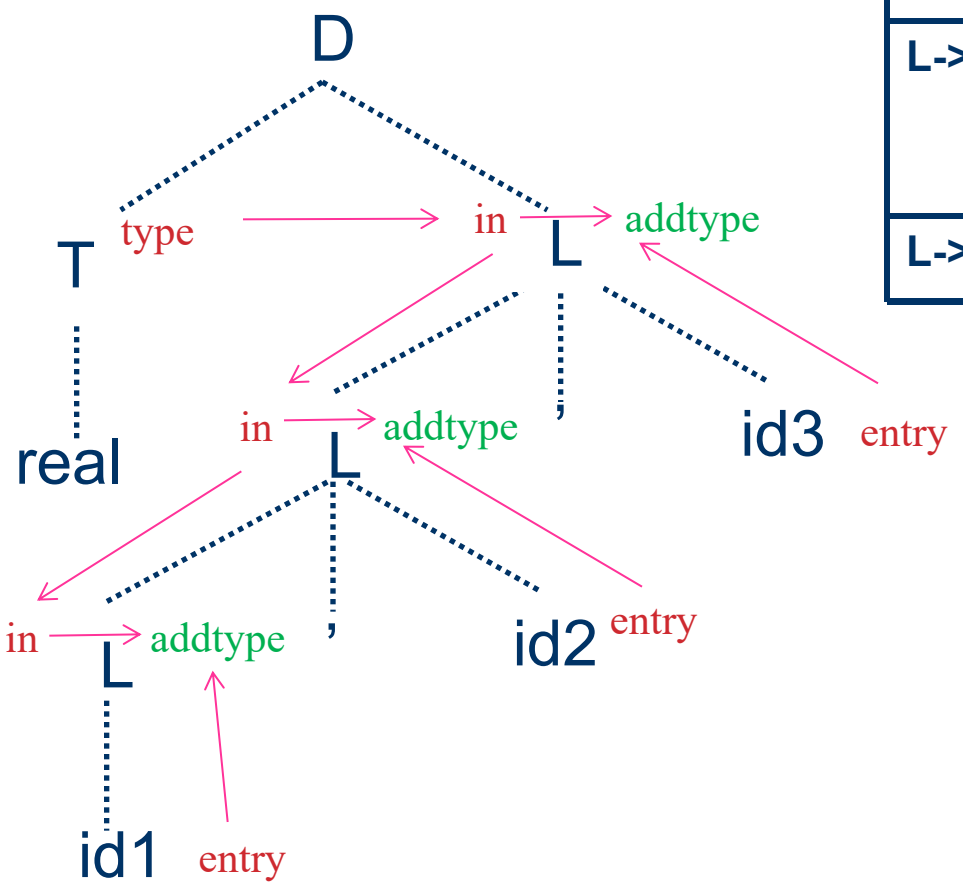
int id_1,id_2的语法树:

用→表示属性的传递情况



句子 real id1,id2,id3的依赖图

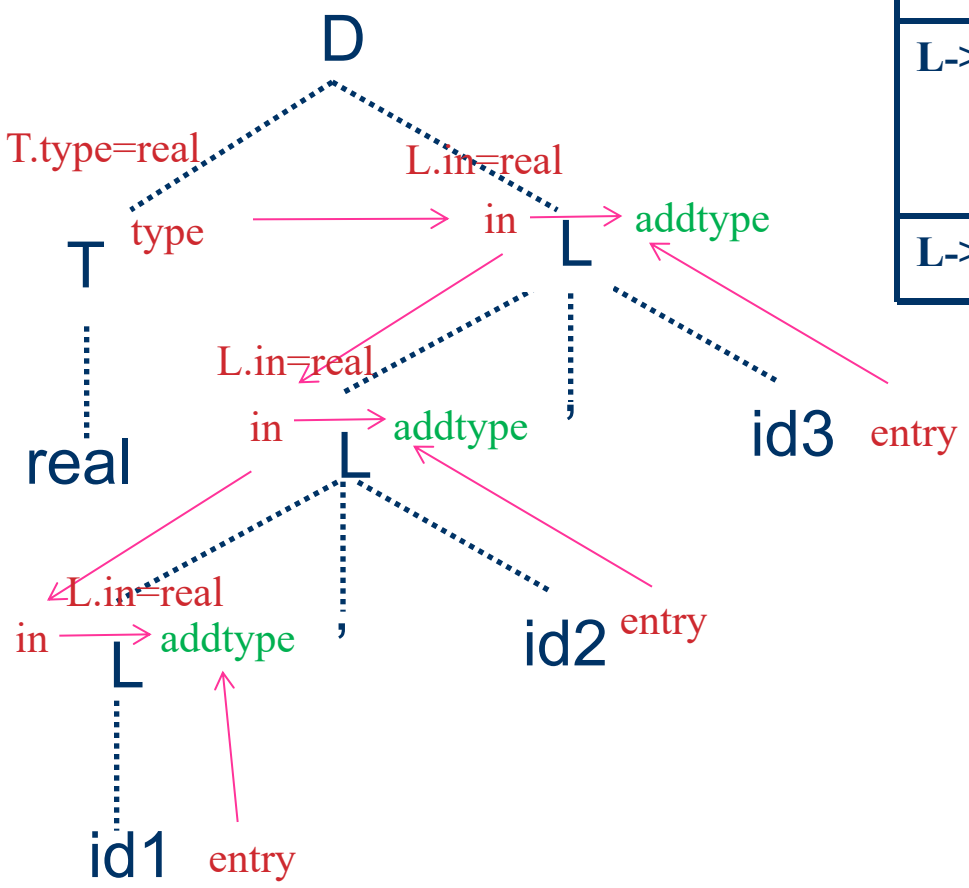
文法规则	语义规则
D->TL	L.in := T.type
T->int	T.type:= integer
T->real	T.type:=real
L->L1,id	L1.in := L.in L.s = addtype(id.entry, L.in)
L->id	addtype(id.entry, L.in)



name	type
...	...
id3	
id2	
id1	

句子 real id1,id2,id3的依赖图

文法规则	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)



name	type
...	...
id3	real
id2	real
id1	real

The Evaluation of Synthesized Attributes

- Given that a parse tree or syntax tree has been constructed by a parse, the synthesized attribute values can be computed by a single bottom-up, or postorder traversal of the tree.

- **Express this by the following code**

```
procedure PostEval(T:treenode)
begin
  for each child C of T do
      PostEval(C);
      compute all synthesized attributes of T;
end;
```

The Evaluation of Inherited Attributes

- Inherited attributes can be computed by a preorder traversal, or combined preorder/inorder traversal of the parse tree or syntax tree.

- Express this by the following code

```
procedure PreEval(T:treenode);
begin
        for each child C of T do
            compute all inherited attributes of C;
            PreEval(C);
end;
```

```
while (there is an attribute to be computed) do
    VisitNode(S);

procedure VisitNode(N:Node);
begin
    if N is non-terminal then
        for i:=0 to m do
            if X_i ∈ V_N then
                begin
                    Compute all the inherited attributes of X_i which could be computed.
                    VisitNode(X_i);
                end;
    Compute all the synthesized attributes of N which could be computed.
end;
```

Note:

- **Unlike synthesized attributes, the order in which the inherited attributes of the children are computed is important.**

- **Since inherited attributes may have dependencies among the attributes of the children.**

Example

Attribute grammar for variable declarations

Grammar rule	Semantic Rules
decl->type varlist	varlist.dtype=type.dtype
type->int	type.dtype=integer
type->float	type.dtype=real
varlist1->id,varlist2	id.dtype=varlist1.dtype varlist2.dtype=varlist1.dtype
var-list->id	id.dtype=varlist.dtype

- Assume that a parse tree has been explicitly constructed from the grammar.
- A recursive procedure that computes the *dtype* attribute at all required nodes.

```
procedure EvalType(T:treenode);
begin
    case nodekind of T of
    decl: //decl->type varlist {varlist.dtype=type.dtype}
          EvalType(type child of T);
          vallist.dtype=type.dtype;
          EvalType(varlist child of T);
    type:       //type->int          {type.dtype=integer}
                //type-> float       {type.dtype=real}
          if child of T=int then T.dtype:=integer
          else T.dtype:=real;
```

varlist: //varlist->id,varlist{id.dtype=varlist1.dtype

varlist2.dtype=varlist1.dtype}

//varlist->id{id.dtype=varlist.dtype}

assign T.dtype to first child of T;

if third child of T is not nil then

assign T.dtype to third child;

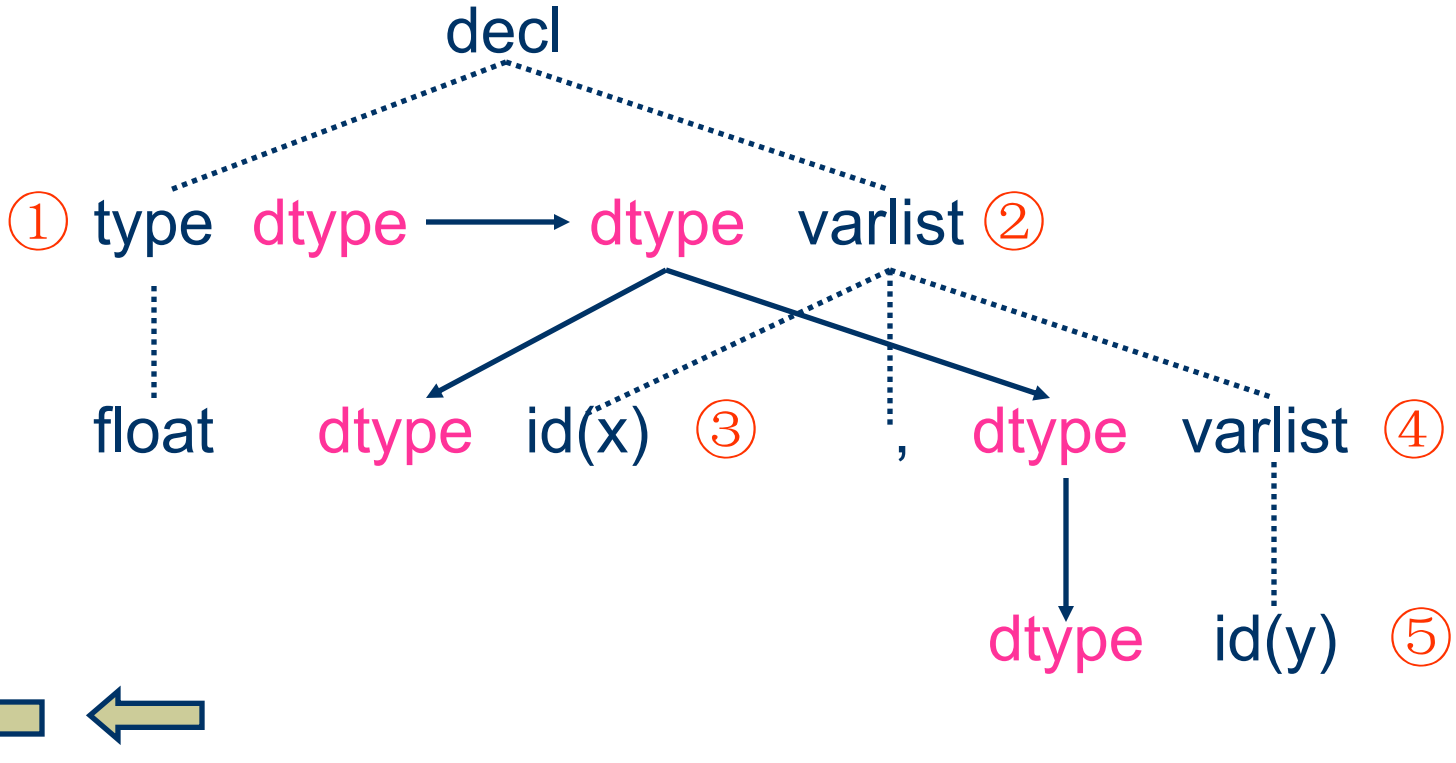
EvalType(third child of T);

end case;

end EvalType;

- **The parse tree for the string "float x,y" together with the dependency graph for the *dtype* attribute.**

We number the nodes to show the traversal order.



7.2 语法制导翻译概论

1. 语法制导翻译

- 基本思想：

  在语法分析过程中，随着分析的步步进展，每当使用一条产生式进行推导（对于自上而下分析）或归约（对于自下而上分析），就执行该产生式所对应的语义动作，完成相应的翻译工作。

- 语法制导翻译法不论对自上而下分析或自下而上分析都适用

例 简单算术表达式求值的属性文法

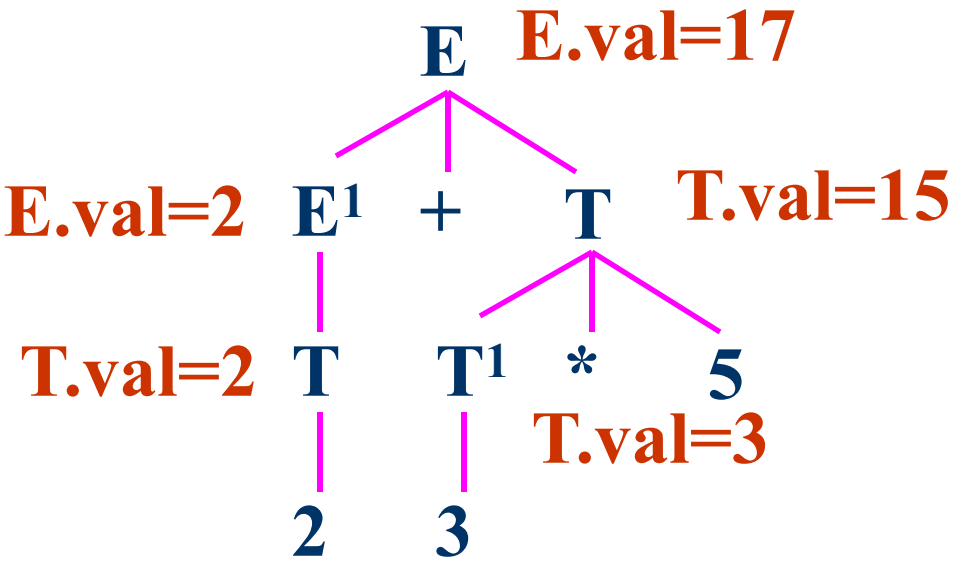
1) $E \rightarrow E^1+T$ { E.val :＝E^1.val +T.val }

2) $E \rightarrow T$ { E.val :＝T.val }

3) $T \rightarrow T^1*digit$ { T.val :＝T^1.val * digit.lexval }

4) $T \rightarrow digit$ { T.val :＝digit.lexval }

2+3*5的语法树：　　自下而上语法制导翻译过程：



一旦语法分析确认输入符号串是一个句子，它的值也同时由语义规则计算出来

2. 语法制导翻译的实现途径

以自下而上（LR分析）的语法制导翻译来说明

- 将LR分析器能力扩大，增加在归约后调用语义规则的功能
- 增加语义栈，语义值放到与符号栈同步操作的语义栈中，多项语义值可设多个语义栈，栈结构为：

状态栈	符号栈	语义栈
S_m	X_m	X_m.val
…	…	…
S_1	X_1	X_1.val
S_0	#	-

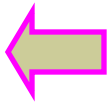
例 简单算术表达式求值的属性文法

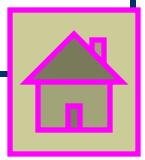
1) L →E
   {print(E.val)}
2) E→E¹+T
   { E.val :＝E¹.val +T.val }
3) E→T
   { E.val :＝T.val }
4) T→T¹*digit
{ T.val :＝T¹.val * digit.lexval }
5) T→digit
   { T.val :＝digit.lexval }

状态	ACTION				GOTO	
	d	+	*	#	E	T
0	S_3				1	2
1		S_4		acc		
2		r3	S_5	r_3		3
3		r_5	r_5	r_5		
4	S_3					7
5	S_6					
6		r_4	r_4	r_4		
7		r_2	S_5	r_2		

分析并计算2＋3*5的过程如下:

步骤	状态栈	语义栈	符号栈	剩余输入串	Action	GOTO
0	0	-	#	2＋3*5＃	S_3	
1	03	- -	#2	＋3*5＃	r_5	2
2	02	-2	#T	＋3*5＃	r_3	1
3	01	-2	#E	＋3*5＃	S_4	
4	014	-2-	#E+	3*5＃	S_3	
5	0143	-2- -	#E+3	*5＃	r_5	7
6	0147	-2-3	#E+T	*5＃	S_5	
7	01475	-2-3-	#E+T*	5＃	S_6	
8	014756	- 2-3-5	#E+T*5	＃	r_4	7
9	0147	-2-15	#E+T	#	r_2	1
10	01	-17	#E	#	acc	





7.3 中间代码的形式

- 定义：

  中间代码是一种复杂性介于源程序语言和机器语言之间的一种表示形式。

- 使用中间代码的好处：

  - 中间代码与具体机器无关
  - 对中间代码进行与机器无关的优化

- 形式：

  逆波兰记号、三元式、四元式和树形表示

7.3.1 逆波兰记号

- 逆波兰表示法

  将运算对象写在前面，把运算符写在后面，因而也称后缀式。

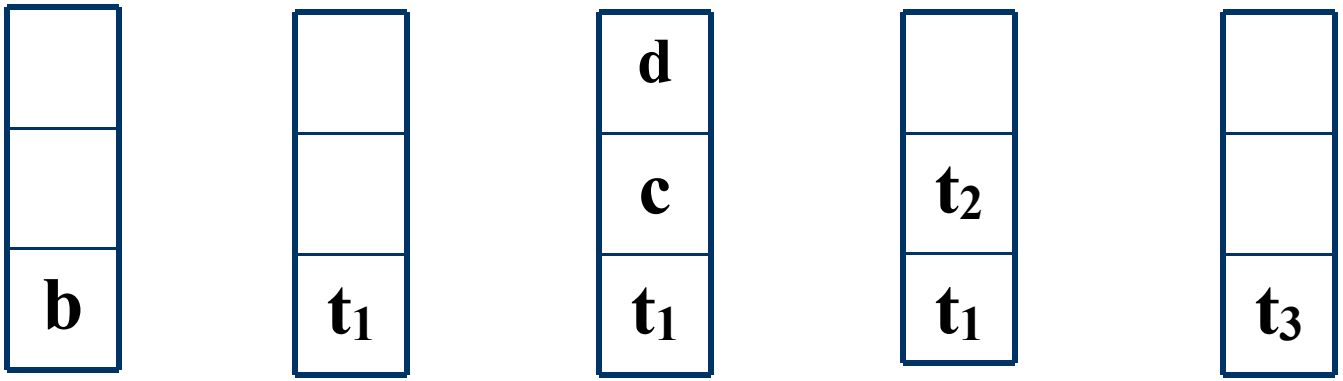
  例如：

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 *

- 后缀式的计算机处理
  - 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目数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

例：表达式－b＋c*d的后缀式 b@cd*+的计值过程



$t_1 = -b$　　$t_2 = c*d$　　$t_3 = t_1 + t_2$

- 逆波兰表示法的扩充

  逆波兰表示法很容易扩充到表达式以外的范围

  例如：

语句	逆波兰表示	备注
a:=b+c	**abc+:=**	**:=看作二目运算符**
GOTO L	**L jump**	**jump看成一目运算符，表示GOTO**
If E then S^1 else S^2	**E$S^1$$S^2$￥**	**把￥看成三目运算符，表示if –then –else**

7.3.2 三元式和树形表示

- 三元式

(算符op，第一个运算对象ARG1,第二个运算对象ARG2)

例： a :＝b*c+b*d表示为

(1) (* ,　　b,　　c　　)

(2) (* ,　　b,　　d　　)

(3) (+ ,　　(1),　　(2)　　)

(4) (:＝,　　(3),　　a　　)

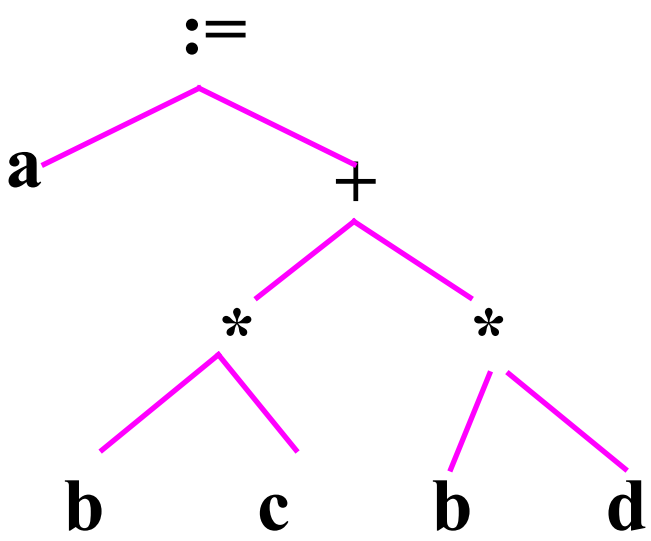
说明：

- 三元式的某些运算对象是另一个三元式的编号（代表其结果）
- 一目算符只需选用一个运算对象（ARG1）
- 多目算符可用连续几个三元式表示

- 树形表示

二目运算对应二叉子树，多目运算对应多叉子树，但通常通过引入新结点表示成二叉子树。

例如：a:＝b*c+b*d 表示成



7.3.3 四元式

- 四元式表示

四元式是一种比较普遍采用的中间代码形式

(算符op，ARG1，ARG2，运算结果RESULT）

例如：a:＝b*c+b*d的四元式表示如下：

1) (*, b, c, t_1)

2) (*, b, d, t_2)

3) (+, t_1, t_2, t_3)

4) (:＝, t_3, －, a)

其中t_i（i＝1,2,3）是编译程序引入的临时变量

- 四元式的优点：
- 四元式比三元式更便于优化。

  优化要求改变运算顺序或删除某些运算，引起编号的变化。

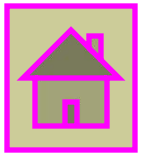
  三元式通过编号引用中间结果，编号的变化引起麻烦；四元式通过临时变量引用中间结果，编号变化无影响。
- 四元式对生成目标代码有利。

  四元式表示很类似于三地址指令，很容易转换成机器代码。

- 四元式的另一种表示

  有时为了更直观，把四元式写成简单赋值形式或更易理解的形式

四元式	直观形式
（1）（ * , b , c , t_1）	（1） t_1:＝b*c
（2）（ * , b , d , t_2）	（2） t_2:＝b*d
（3）（ +, t_1 , t_2 , t_3）	（3） t_3:＝t_1+t_2
（4）（:＝, t_3 ,－, a）	（4） a:＝t_3
(jump,－， －， L)	goto L
(jrop, B， C， L)	if B rop C goto L



7.4 基本语言成分的自下而上语法制导翻译

7.4.1 简单赋值语句的翻译

- 简单赋值语句

  是指不含复杂数据类型（如数组，记录等）的赋值语句。

- 赋值语句的语义审查包括：

  1. 每个使用性标识符是否都有声明？
  2. 运算符的分量类型是否相容？
  3. 赋值语句的左右部的类型是否相容？

- 赋值语句的翻译目标：

  在赋值语句右部表达式产生的四元式序列后加一条赋值四元式

1．属性和语义规则中用到的变量、过程和函数

属性：

- 用**id.name**表示单词id的名字。
- 用**E.place**表示存放E值的变量名在符号表的入口地址或临时变量编码。

变量、函数和过程：

- 用**nextstat**变量给出在输出序列中下一个四元式的序号
- 用**lookup（id.name）**函数审查id.name是否出现在符号表中，是则返回id的入口地址，否则返回nil。
- 用**emit**过程向输出序列输出一个四元式，emit每调用一次，nextstat的值增加1
- 用**newtemp**函数生成临时变量，每次调用生成一个新的临时变量，如$t_1, t_2, \ldots\ldots$
- 用**error**过程进行错误处理。

2．简单赋值语句的翻译（假定变量只有一种类型）

此情况下的语义审查只有：
每个使用性标识符是否都有声明？

(1) S→id:＝E

{ p:＝lookup (id.name) ;

if p≠nil then emit (:＝, E.place , - , p)

else error }

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

{ E.place:＝newtemp ;

emit (+ , E^1.place , E^2.place , E.place) }

(3) $E \rightarrow E^1 * E^2$

{ E.place:＝newtemp ;

emit (* , E^1.place , E^2.place , E.place) }

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

{ E.place:＝newtemp ;

emit (@ , E^1.place , - , E.place) }

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

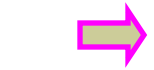
{ E.place:＝E^1.place }

(6) $E \rightarrow id$

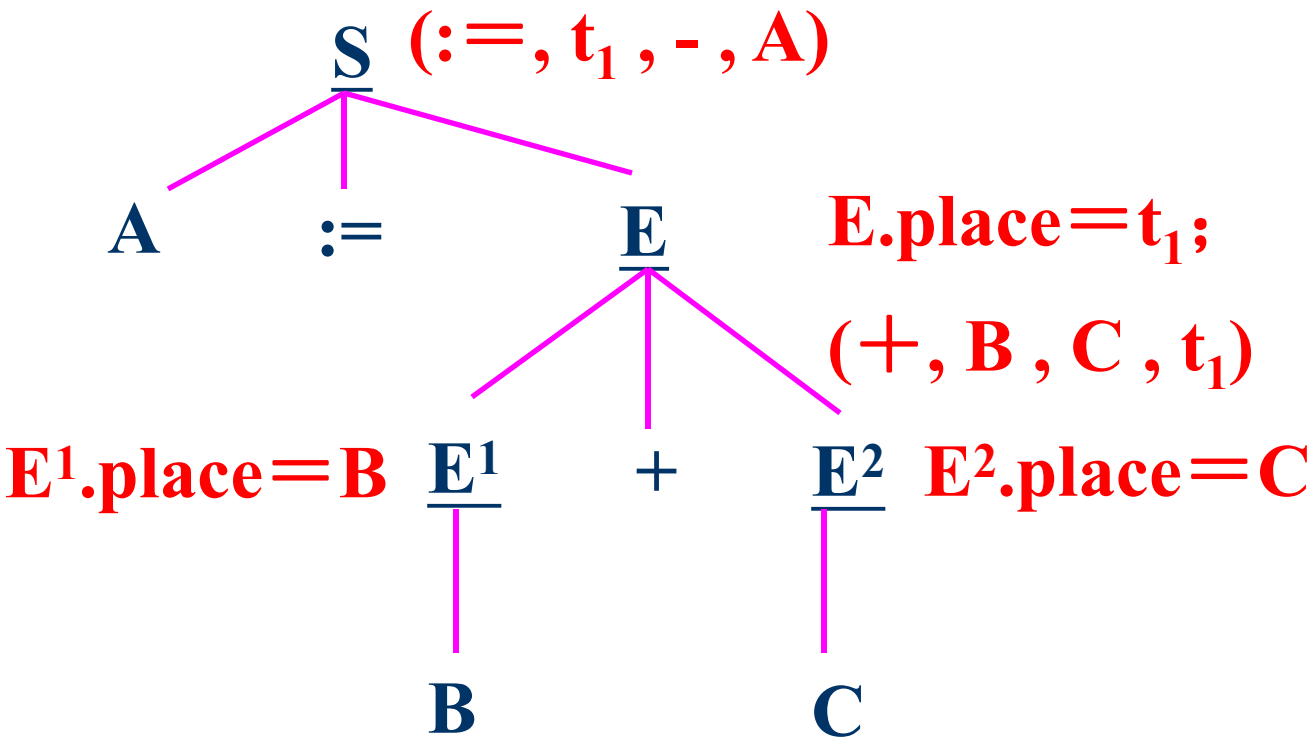
{ p:＝lookup (id.name) ;

if p≠nil then E.place:＝p

else error }



例 翻译赋值语句A:＝B+C



(为了直观，用B和C分别表示B和C在符号表的入口地址)

3 简单赋值语句的四元式翻译

- 表达式中可能出现不同类型的变量和常量
- 语义审查包括：
  1. 每个使用性标识符是否都有声明？
  2. 运算符的分量类型是否相容？
     - 若不接受不同类型的运算对象混合运算，则应指出错误；
     - 若接受混合运算则要进行类型转换处理。
- 例：假定表达式可以有混合运算，id可以是整型和实型，且当两个不同类型的id进行运算时先把整型id转换成实型，再进行运算。

用E.type表示E的类型信息，其值为int或real。

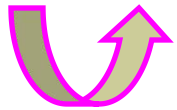
用 $+^{i}$, $*^{i}$ 表示整型运算，用 $+^{r}$, $*^{r}$ 表示实型运算。

用一目算符 itr 表示将整型量转换成实型量的运算。

产生式$E \rightarrow E^1+E^2$的包含类型属性的语义规则为：

E.place:＝newtemp ;
if E^1.type＝int AND E^2.type＝int then
begin emit ($+^i$, E^1.place , E^2.place , E.place) ; E.type:＝int end
else if E^1.type＝real AND E^2.type＝real then
begin emit ($+^r$, E^1.place , E^2.place , E.place) ;
E.type:＝real
end
else if E^1.type＝int then
begin t:＝newtemp ; emit (itr , E^1.place , - , t) ;
emit ($+^r$, t , E^2.place , E.place) ; E.type:＝real
end
else begin t:＝newtemp ; emit (itr , E^2.place , - , t) ;
emit ($+^r$, E^1.place , t , E.place) ; E.type:＝real
end ;

- 属性文法的构造
  - 属性：根据语义处理的需要，设计文法符号的相应属性（包括：属性的个数和属性的符号表示）
  - 语义规则：满足语义处理的要求，并生成相应的中间代码



7.4.2 布尔表达式的翻译

1. 布尔表达式的作用与结构

- 布尔表达式的两个作用：
  - 计算逻辑值
  - 作为控制语句(如if-then,while)的条件表达式
- 布尔表达式的语法：

<BE>→<BE> or <BE> |<BE> and <BE> | not <BE> | (<BE>) |<RE>| true | false （布尔表达式）

<RE>→<AE> relop <AE> | (<RE>) （关系表达式）

<AE>→<AE> op <AE> | -<AE> | (<AE>) | id | num （算术表达式）

其中：relop是关系算符(如<=, < ,=,≠, > , >=)

op是算术算符(+ , - , * , /)

- 只考虑如下形式的布尔表达式的翻译

  E→E or E | E and E | not E | (E) | id rop id
  |true|false

  - 布尔算符的优先顺序（从高到低）为：**not,and,or**，且**and**和**or**都服从左结合，**not**服从右结合
  - 关系算符的优先级都相同，而且高于任何布尔算符，低于任何算术算符。

2. 布尔表达式的计算方法：

采用两种方法：数值表示的直接计算与逻辑表示的短路计算

- 直接计算与算术表达式计算方法基本相同

  如：**1 or 0 and 1=1 or 0=1**

- 短路计算即布尔表达式计算到某一部分就可以得到结果，而无需对布尔表达式进行完全计算。可以用**if-then-else**来解释

A or B	**if A then 1 else B**
A and B	**if A then B else 0**
not A	**if A then 0 else 1**

3. 直接计算的语法制导翻译

如：A or B and not C被翻译成：

(not, C, - , t_1)

(and, B, t_1, t_2)

(or, A, t_2, t_3)

对关系表达式，如a<b，可翻译成如下固定的三地址代码序列：

(1) (j<, a, b, (4))

(2) (:=, 0, - , t_1)

(3) (jump, - , - , (5))

(4) (:=, 1, - , t_1)

(5) …

直接计算的翻译方案

(1)$E \rightarrow E^1$ or E^2 { E.place :＝newtemp ;
emit (or , E^1.place , E^2.place , E.place) }

(2)$E \rightarrow E^1$ and E^2 { E.place :＝newtemp ;
emit (and , E^1.place , E^2.place , E.place) }

(3)$E \rightarrow$ not E^1 { E.place :＝newtemp ;
emit (not , E^1.place ,—, E.place) }

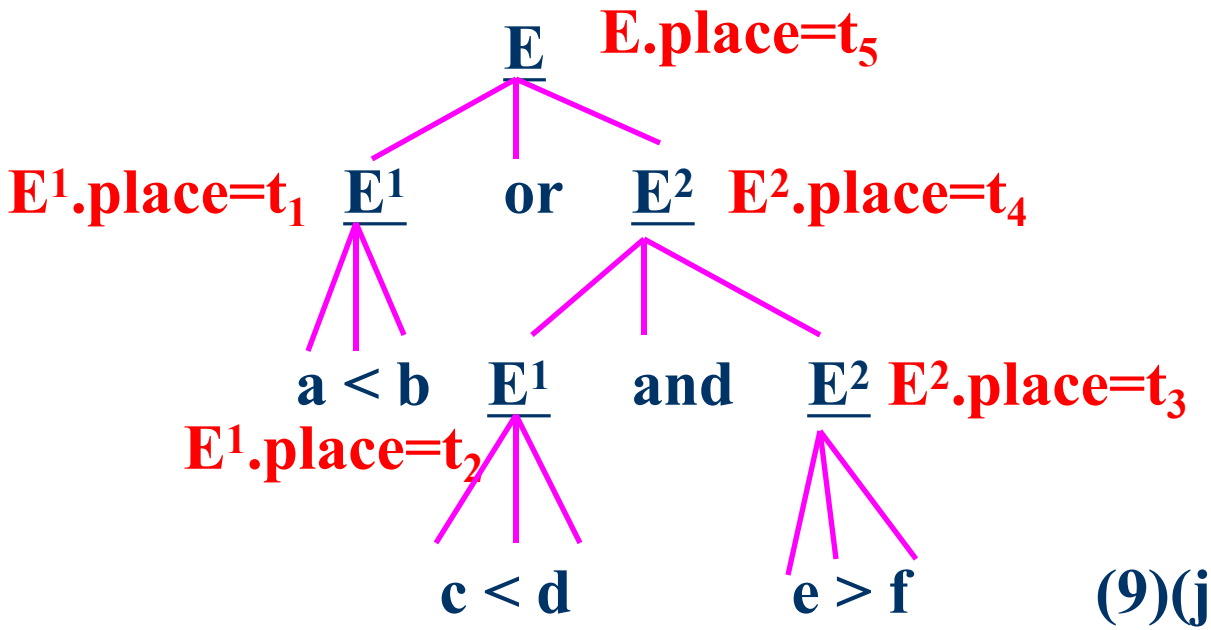
(4)$E \rightarrow (E^1)$ { E.place :＝E^1.place }

(5)$E \rightarrow id_1$ rop id_2 { E.place :＝newtemp ;
emit (jrop , id_1.place , id_2.place , nextstat+3) ;
emit (:＝, 0 ,－, E.place) ;
emit (jump ,—,—, nextstat+2) ;
emit (:＝, 1 ,－, E.place) }

(6)E→true { E.place:＝newtemp;emit(:=,1,- ,E.place) }

(7)E→false {E.place:=newtemp;emit(:=,0,- ,E.place)}

例：布尔表达式a<b or c<d and e>f的翻译



(1)(j<, a, b, (4))

(2)(:=, 0, - , t_1)

(3)(jump,- ,- ,(5))

(4)(:=, 1, - , t_1)

(5)(j<, c, d, (8))

(6)(:=, 0, - , t_2)

(7)(jump, - , - ,(9))

(8)(:=, 1, - , t_2)

(9)(j>, e, f, (12))

(10)(:=, 0, - , t_3)

(11)(jump,- ,- ,(13))

(12)(:=, 1, - , t_3)

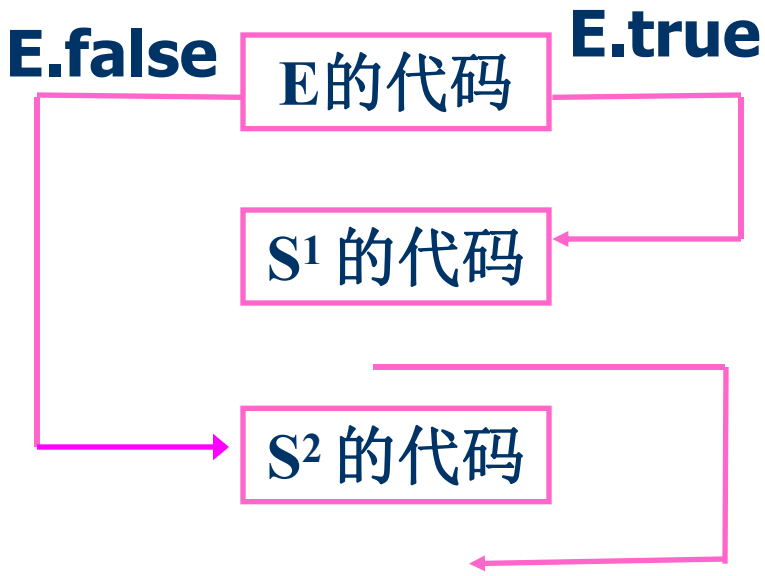
(13)(and, t_2, t_3, t_4)

(14)(or, t_1, t_4, t_5)

4. 作为条件控制的布尔表达式的翻译

- 基本翻译方法

当布尔表达式用于控制条件时，并不需要计算表达式的值，而是一旦确定了表达式为真或为假，就将控制转向相应的代码序列。



if E then S^1 else S^2

为布尔表达式E引入两个新的属性：

- **E.true**：表达式的真出口，它指向表达式为真时的转向
- **E.false**：表达式的假出口，它指向表达式为假时的转向

把E翻译成下述形式的条件转移和无条件转移的四元式序列：

1. (jnz , A , - , p)

   若A为真，则转向四元式p

2. (jrop , A , B , p)

   若A rop B为真，则转向四元式p

3. (jump , - , - , p)

   无条件转向四元式p

例：if A or B<D then S^1 else S^2翻译成如下四元式序列

(1)	(jnz , A , - , 5)	A的真出口为5
(2)	(jump , - , - , 3)	A的假出口为3
(3)	(j< , B , D , 5)	B<D的真出口为5
(4)	(jump , - , - , p+1)	B<D的假出口为(p+1)
(5)	(关于S^1的四元式序列)	
(p)	(jump , - , - , q)	跳过S^2的代码段
(p+1)	(关于S^2的四元式序列)	
(q)		

(1) - (4)是布尔式A or B<D 翻译产生的代码，全部是条件转移和无条件转移四元式，没有布尔运算。

具体说明如下：

用E.true和E.false 分别表示E的“真”和“假”出口转移目标，在翻译E时并未能确定。

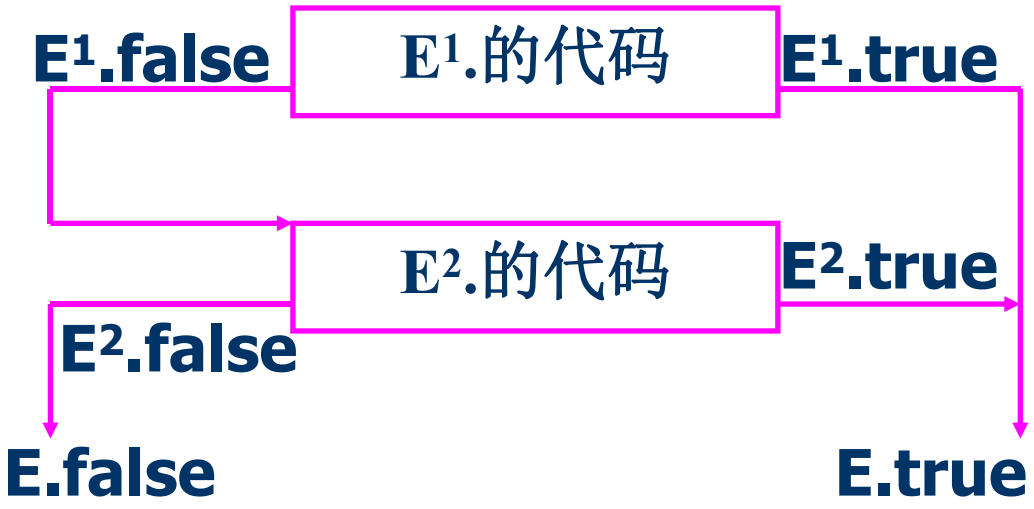
- 对于E为 a rop b 形式，生成代码如下：

(jrop , a , b , E.true)

(jump ,－,－, E.false)

以结构图表示：

➢ 对于E为 E^1 or E^2的形式，生成代码结构如下：



若E^1为真，则可知E为真，即E^1的真出口和E的真出口一样；
若E^1为假，则必须计算E^2，因此E^1的假出口应是E^2代码的第一个四元式序号；

E^2的真出口和假出口分别与E的真出口和假出口一样

➢ 对于E为 E^1 and E^2的形式，生成代码结构如下：

➢对于E为 not E^1形式，只需调换E^1的真假出口，即可得到E的真假出口。

例：E 为 a<b or c<d and e>f，翻译为四元式序列：

（1） (j<, a, b, E.true)

（2） (jump, - , - , (3))

（3） (j<, c , d , (5))

（4） (jump, - , - , E.false)

（5） (j>, e , f , E.true)

（6） (jump, - , - , E.false)

- 真假出口的拉链与回填
  - 原因

    在把布尔式翻译成一串条件转和无条件转四元式时，真假出口未能在生成四元式时确定；而且多个四元式可能有相同的出口

if a<b or c<d and e>f then S^1 else S^2

翻译为四元式序列：

（1） (j<, a, b, (7))

（2） (jump, -, -, (3))

（3） (j<, c, d, (5))

（4） (jump, -, -, (p+1))

（5） (j>, e, f, (7))

（6） (jump, -, -, (p+1))

（7） (关于S^1的四元式)

……

（p） (jump, -, -, q)

(p+1) (关于S^2的四元式)

……

(q)

说明：

- E.true和E.false不能在产生四元式的同时确定，要等将来目标明确时再回填，为此要记录这些要回填的四元式。
- 通常采用“拉链”的办法，把需要回填E.true的四元式拉成一条“真”链，把需要回填E.false的四元式拉成一条“假”链。

- 拉链方式：

若有四元式序列：

(10)…… goto E.true

……

(20)…… goto E.true

……

(30)…… goto E.true

则链接成为：

(10)…… goto (0)

……

(20)…… goto (10)

……

(30)…… goto (20)

- 把地址（30）作为链首，地址（10）作为链尾，0为链尾标志。
- 四元式的第四个区段存放链指针。
- E.true 和E.false用于存放“真”链和“假”链的链首。

- 为了完成拉链和回填工作，设计以下语义变量和过程（函数）：

1) 函数**merge (p_1, p_2)** 用于把P_1和p_2为链首的两条链合并成**1**条，返回合并后的链首值。

   其算法为：当P_2为空链时，返回P_1；当P_2不为空链时，把P_2的链尾第四区段改为P_1，返回P_2。

2) 过程**backpatch (p , t)** 用于把链首**P**所链接的每个四元式的第四区段都填为转移目标**t**。

3) 语义变量**E.codebegin**表示表达式**E**的第一个四元式的序号。

- 自下而上分析中布尔表达式的一种翻译方案

1) $E \rightarrow E^1$ or E^2

{ E.codebegin:＝E^1.codebegin ;
backpatch (E^1.false , E^2.codebegin) ;
E.true:＝merge (E^1.true , E^2.true) ;
E.false:＝E^2.false }

2) $E \rightarrow E^1$ and E^2

{ E.codebegin:＝E^1.codebigin ;
backpatch (E^1.true , E^2.codebegin) ;
E.true:＝E^2.true ;
E.false:＝merge (E^1.fasle , E^2.false) }

3) $E \rightarrow \text{not } E^1$

{ E.codebegin:＝E^1.codebigin ;

E.true:＝E^1.false ;

E.false:＝E^1.true }

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

{ E.codebegin:＝E^1.codebegin ;

E.true:＝E^1.true ;

E.false:＝E^1.false }

5) $E \rightarrow id_1\ rop\ id_2$

{ E.codebegin:＝nextstat ;

E.true:＝nextstat ;

E.false:＝nextstat+1;

emit (jrop , id_1.place , id_2.place , 0) ;

emit (jump ,－,－, 0) }

6) E→true

{ E.codebegin:＝nextstat ;

E.true:＝nextstat ;

E.false:＝0;

emit (jump ,－,－,0) }

7) E→false

{ E.codebegin:＝nextstat ;

E.false:＝nextstat ;

E.true:＝0;

emit (jump ,－,－,0) }

例 a<b or c<d and e<f 的翻译过程

假定四元式编号从100开始，

即开始时nextstat＝100

E.begin=100

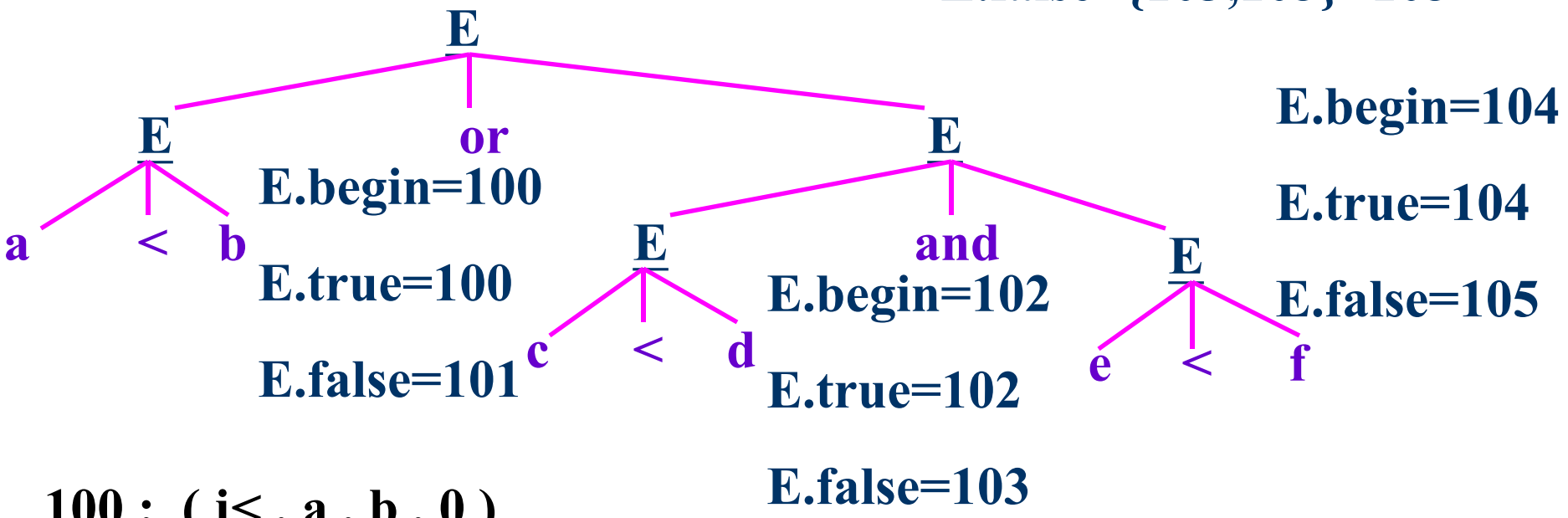
E.true={100,104}=104

E.false=105

E.begin=102

E.true=104

E.false={103,105}=105



100 : (j< , a , b , 0)

101: (jump ,—,—,0)

101(jump,－,－,102)

102: (j< , c , d , 0)

102 (j< , c , d ,104)

103: (jump ,－,－, 0)

104: (j< , e , f , 0)

104 (j< , e , f, 100)

105: (jump ,－,－,0)

105 (jump ,—,—,103)

最终结果：

100：(j< , a , b , 0)

101：(jump,—,—,102)

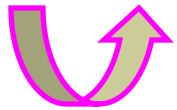
102：(j< , c , d , 104)

103：(jump,－,－, 0)

104：(j< , e , f , 100)

105：(jump,－,－,103)

“真”链首E.true＝104 , “假”链首E.false＝105。



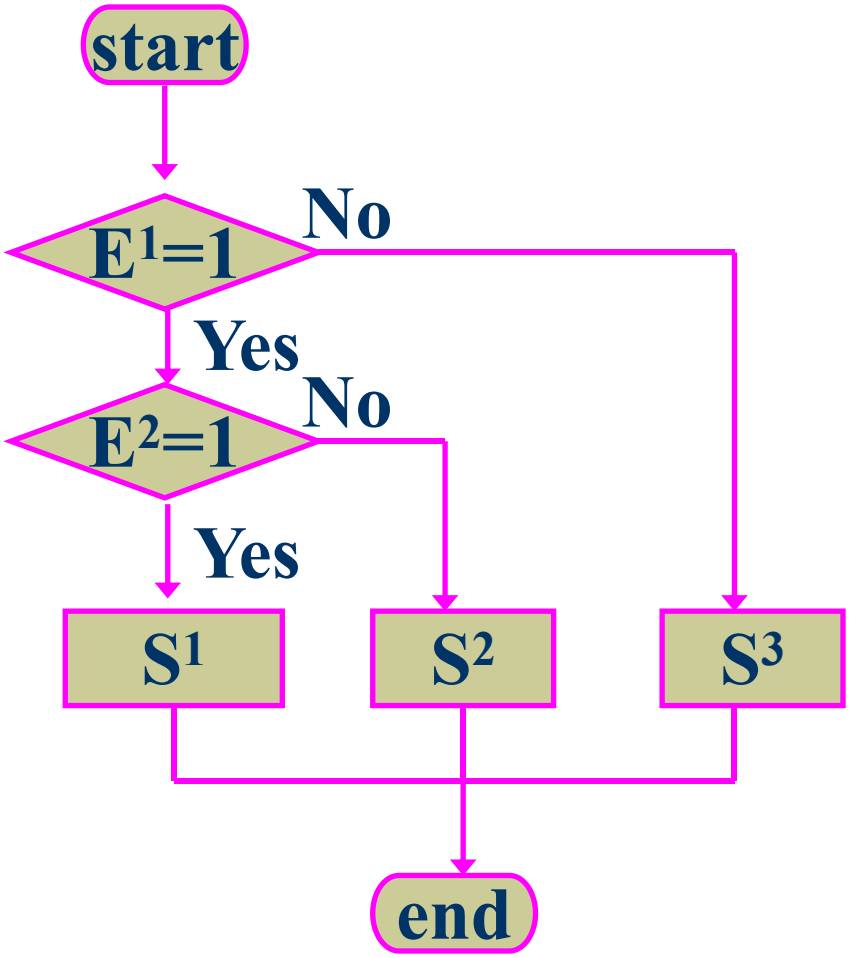
7.4.3 控制结构的翻译

- 以if 语句，while语句为例说明控制语句的翻译方法

S→ if E then S	if语句
| if E then S else S	if语句
| while E do S	while语句
| begin L end	复合语句
| A	赋值语句
A →id:=E	
L→ L ; S	语句序列
| S	语句

条件转移语句的共同特点是：根据布尔表达式取值，分别执行不同的语句序列。

问题：不同的语句序列结束后，如何使控制转向语句的结束。例如：if E^1 then if E^2 then S^1 else S^2 else S^3



参照布尔表达式的翻译方法，对非终结符S(和L)，设立语义变量S.CHAIN（和L.CHAIN），用于记住需要在翻译完S(L)后回填转移目标的一串四元式

1. 代码结构

- if E then S^1 代码结构

- if E then S^1 else S^2 代码结构

- while E do S^1 代码结构

2. 文法的改写

- 原因：在自下而上的语法制导翻译中，语义动作的执行是在使用产生式进行归约之后，并不允许在产生式的中间执行。为了能及时地执行语义动作（比如回填转移目标），需对源文法改写
- 方法：在需要执行语义动作的地方把产生式分段，引入新的非终结符来表示它
- 需要改写的产生式：

1) 把 **S→if E then S^1** 改写成

C→if E then (回填**E.true**)

S→C S^1

2) 把 $S \rightarrow$ if E then S^1 else S^2

改写成

$C \rightarrow$ if E then (回填E.true)

$T^p \rightarrow C\ S^1$ else (产生转移，回填E.false)

$S \rightarrow T^p\ S^2$

3) 把 $S \rightarrow$ while E do S^3

改写成

$W \rightarrow$ while (记住入口)

$W^d \rightarrow W$ E do (回填E.true)

$S \rightarrow W^d\ S^3$

4) 把 $L \rightarrow L$; S

改写成

$L^s \rightarrow L$; (回填前一语句的出口)

$L \rightarrow L^s$ S

源文法：

S→ if E then S

S→ if E then S else S

S→ while E do S

S→ begin L end

S→ A

L→ L ; S

L→ S

改写后的文法

(1) $S \rightarrow C\ S^1$

(2) $S \rightarrow T^p\ S^2$

(3) $S \rightarrow W^d\ S^3$

(4) S→ begin L end

(5) S→ A

(6) $L \rightarrow L^s\ S$

(7) L→ S

(8) C→ if E then

(9) $T^p \rightarrow C\ S^1$ else

(10) W→ while

(11) $W^d \rightarrow W\ E$ do

(12) $L^s \rightarrow L$;

3．安排语义动作

C→if E then

{ backpatch (E.true , nextstat) ;

C.CHAIN:=E.false }

S→C S^1　　/* if E then S^1 */

{ S.CHAIN:＝merge (C.CHAIN , S^1.CHAIN) }

T^p→C S^1 else　　/* if E then S^1 else */

{ q:=nextstat ;

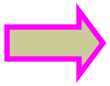
emit (jump,－,－,0) ;　　/*S^1执行完，跳离整个if语句*/

backpatch (C.CHAIN , nextstat) ;

T^p.CHAIN:＝merge (q , S^1.CHAIN) }

S→T^p S^2　　/* if E then S^1 else S^2 */

{ S.CHAIN:＝merge (T^p.CHAIN , S^2.CHAIN) }



W→while

{ W.codebegin:＝nextstat }

W^d→W E do　　/*while E do*/

{ W^d.codebeign:＝W.codebegin ;

backpatch (E.true , nextstat) ;

W^d.CHAIN:＝E.false }

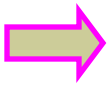
S→ W^d S^3　　/*while E do S^3 */

{ backpatch (S^3.CHAIN , W^d.codebegin) ;

emit (jump ,－,－,W^d.codebegin) ;

/*S^3执行完，跳至While语句开头*/

S.CHAIN:＝W^d.CHAIN) }



S→ begin L end

{ S.CHAIN:＝L.CHAIN }

S→ A

{ S.CHAIN:＝0 } /* 赋值句无出口，故置为空链 */

L→ S

{ L.CHAIN:＝S.CHAIN }

L^s→L ;

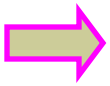
{ backpatch (L.CHAIN , nextstat) }

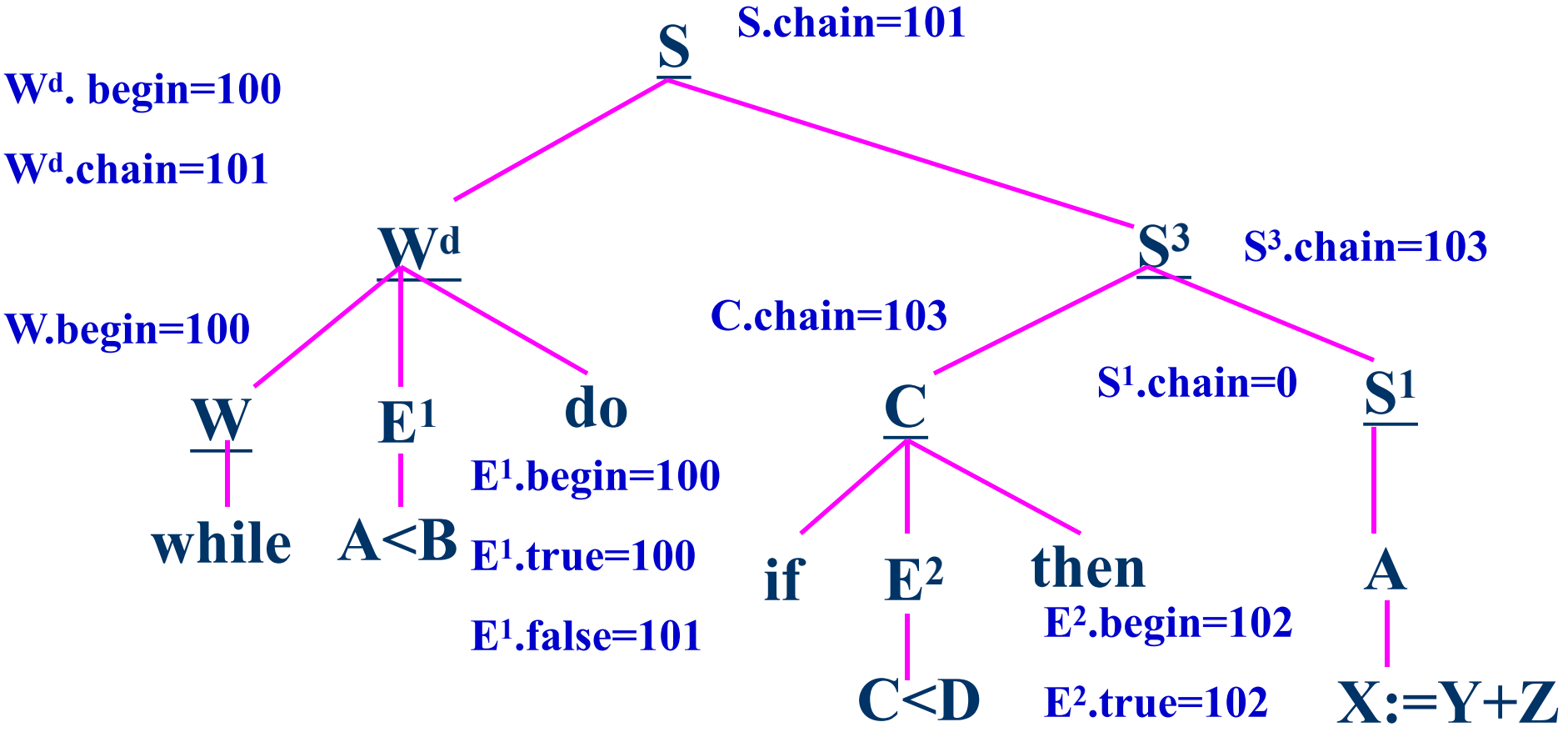
L→L^s S　　/* L;S */

{ L.CHAIN:＝S.CHAIN }

例：翻译语句 while A<B do if C<D then X:＝Y+Z

设nextstat=100





100 (j< , A , B , 0) ⇒ 100 (j< , A , B , 102)

101 (jump, — , — , 0)

102 (j< , C , D , 0) ⇒ 102 (j< , C , D , 104)

103 (jump, — , — , 0) ⇒ 103 (jump, — , — , 100)

104 (+ , Y , Z , t_1)

105 (:= , t_1 , — , X)

106 (jump, — , — , 100)

while A<B do if C<D then X:＝Y+Z 的最终翻译结果为:

100 (j< , A , B , 102)

101 (jump ,－,－, 0)

102 (j< , C , D , 104)

103 (jump ,－,－,100)

104 (＋, Y , Z , t_1)

105 (:＝, t_1 ,－, X)

106 (jump ,－,－,100)

S.CHAIN＝101

while A<B do if C<D then X:＝Y+Z 的翻译过程：

(1) 把while归为W，记住while语句的入口为100

(2) 把A<B归为E^1，产生：

100 (j< , A , B , 0)　　E^1.true＝100

101 (jump,－,－,0)　　E^1.false＝101

(3) 把W E do归为W^d，回填E^1.true 得到

100 (j< , A , B ,102)

W^d.CHAIN＝E^1.false＝101

(4) 把C<D归为E^2，产生：

102 (j< , C , D , 0)　　E^2.true＝102

103 (jump,－,－,0)　　E^2.false＝103

(5)把if E^2 then 归为C，回填E^2.true得

102 (j< , C , D , 104)

C.CHAIN＝E^2.false＝103

(6)把X:＝Y+Z归为S^1，产生：

104 (＋, Y , Z , t_1)

105 (:＝, t_1 ,－, X)

S^1.CHAIN＝0

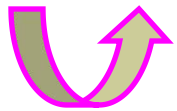
(7) 把C S^1归为S^2，S^2.CHAIN＝merge (103 , 0)＝103

(8) 把W^d S^2归为S，回填S^2.CHAIN得

103 (jump,－,－, 100)

产生四元式 106 (jump,－,－, 100)

S.CHAIN:＝W^d.CHIAN＝101



7.4.4 简单说明语句的翻译

- 说明语句的作用：定义各种形式的有名实体，如常量、变量、数组、记录、过程、子程序等
- 说明语句种类：变量说明，常量说明，类型说明，过程说明等
- 说明语句的翻译：

  简单说明语句的翻译不产生中间代码，编译程序把说明语句中定义的名字和属性登记在符号表中，用以检查名字的引用和说明是否一致

符号表

1. 符号表及其作用

- 符号表(Symbol Table)

  符号表是存放标识符信息的一种表，其中的信息表示的是标识符的属性(语义)。

- 符号表的作用

  符号表是连接声明与引用的桥梁。一个名字在声明时，相关信息被填写进符号表，而在引用时，根据符号表中的信息生成相应的可执行语句。它的作用主要有：

  - 辅助语义的正确性检查
  - 辅助代码生成

2. 符号表的设计

- 如何有效记录各类符号的属性，以便在编译的各个阶段对符号表进行快速、有效的查找、插入、修改、删除等操作，是符号表设计的基本目标。
- 符号表的组成

  表项分两部分，其中前者是标识符的名字（或在名表中的地址），而后者是属性部分（不同种类的标识符属性不同）。
- 符号表的组织方式和查找方法

  符号表的组织方式可以是数组也可以是链表等等，查找算法可以是顺序查表法、平分查表法、散列查表法等

  合理的组织和查找，将使得符号表的操作更高效

过程的说明部分：

```
CONST A=35,B=49;
VAR C,D,E;
PROCEDURE P;
VAR G
```

TABLE表中的信息

变量相对本过程基地址的偏移量

NAME:A	Kind :CONSTANT	VAL:35		
NAME:B	Kind :CONSTANT	VAL:49		
NAME:C	Kind :VARIBALE	LEVEL:LEV	ADR: DX	
NAME:D	Kind :VARIBALE	LEVEL:LEV	ADR: DX+1	
NAME:E	Kind :VARIBALE	LEVEL:LEV	ADR: DX+2	
NAME:P	Kind :PROCEDUR	LEVEL:LEV	ADR:	SIZE:4
NAME:G	Kind :VARIBALE	LEVEL:LEV+1	ADR: DX	
…	…	…	…	

符号表的生存期

- 在编译过程中，每当遇到标识符时，就要查填符号表：若是新的标识符时，就向符号表中填入一个新的表项；否则，根据情况向符号表中的已有表项增填信息（如填入分析的存储地址）或者查获信息（如进行语义检查等）
- 符号表的信息将在词法分析、语法分析的过程中陆续填入，将用于语义检查、产生中间代码以及生成目标代码等不同的阶段。

1简单说明语句

- 文法描述

  D→ integer <namelist> | real <namelist>

  <namelist> → <namelist> , id | id

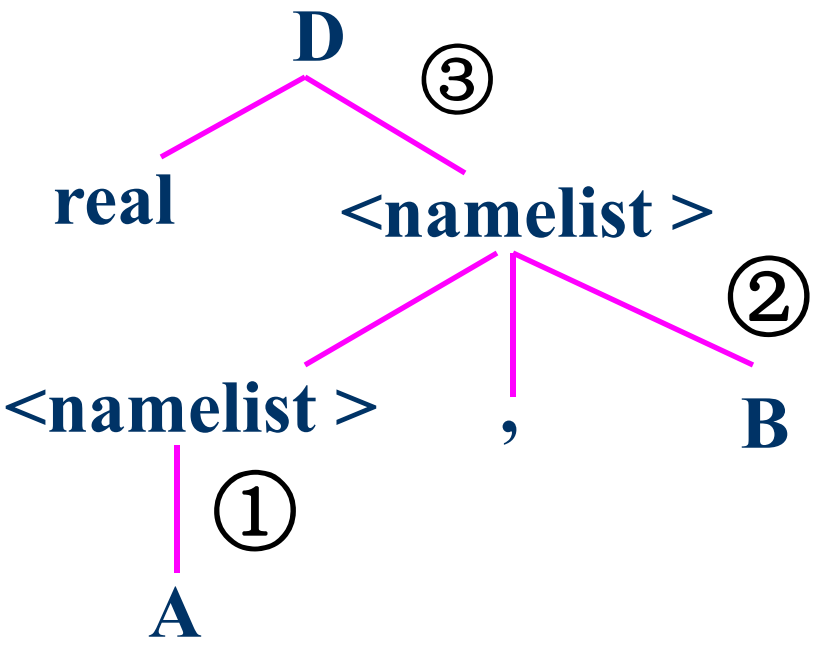
  该文法描述了以integer和real定义的一串名字

- 翻译目标

  把名字及类型信息填入符号表。

- 翻译中存在的问题：

  例:real A , B



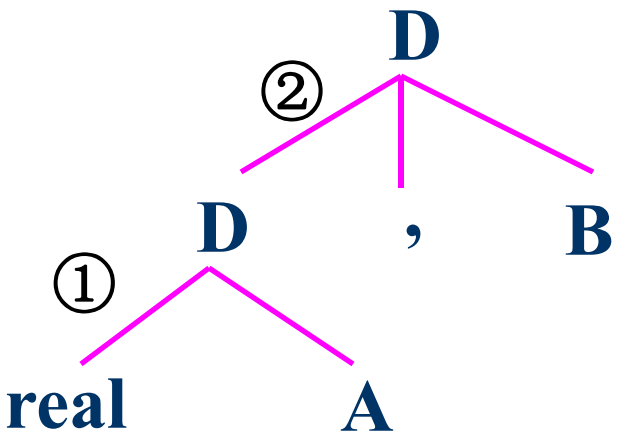
- 第①步归约A和第②步归约B时，因未有类型信息而未能填入符号表
- 只有当第③步归约real后得到类型信息才能把所有名字及类型信息一起填入符号表
- 为此必须用队列（或栈）来保存归约出的名字

2. 文法的改写

- 改写后文法：

  $D \rightarrow integer\ id \mid real\ id \mid D^1\ ,\ id$

- 句子real A ,B的规范归约过程如下：



- 在第①步归约类型real和A，即可把名字A和类型填入符号表
- 在第②步归约B时，利用已知类型信息便可把名字B和类型一起填入符号表
- 不需要另设队列（或栈）。

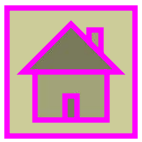
3. 语义动作

- 用到的语义变量和过程：
  - 用语义变量D.ATT记录D的性质（int还是real）
  - 用过程enter (id,ATT)把名字id和性质ATT填入符号表
- 改写后的说明语句的语义动作：

(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 }



7.5 自上向下的语法制导翻译

- 自上向下语法制导翻译的最大优点是：可根据需要在产生式右部的任何位置上调用语义动作，属性的计算更直接、方便
- 递归下降法和LL(1)分析法的易实现性使自顶向下的语法制导翻译法更受欢迎。

7.5.1 递归下降的语法制导翻译

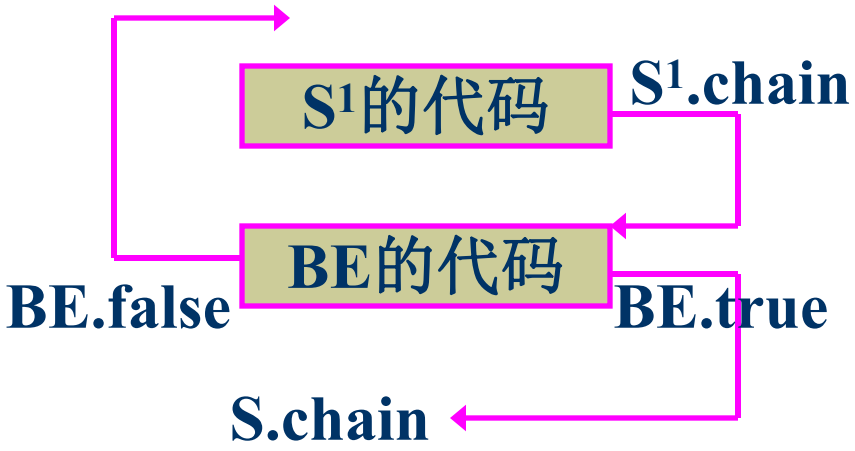
对递归下降子程序的主要修改涉及：

1. 递归子程序可以设计为函数，用于返回必要的属性
2. 适当设计子程序中的临时变量，用于保存属性值；
3. 将语义动作嵌入在子程序的适当位置，正确计算属性值，并能产生一定的四元式

例： <S> → id:=<AE> | repeat <s^1> until <BE>

```
function  S(TOKEN) : pointer;
begin
  case  TOKEN  of
  'id' : begin
         GETNEXT(TOKEN);
         if  TOKEN ≠ ':='  then  ERROR;
         GETNEXT(TOKEN);
         E.place:=AE(TOKEN);
         P:=lookup(id.name);
        if  p ≠ nil  then emit(:=,E.place,-,p)
                    else  ERROR;
        S.CHAIN:=0;
        return(S.CHAIN)
        end;
```

<S> → i:=<AE> | repeat <s1> until <BE>




```
'repeat':
        begin
        R.codebegin:=nextstat;
        GETNEXT(TOKEN);
        S1.CHAIN:=S(TOKEN);
        GETNEXT(TOKEN);
        if TOKEN ≠ 'until'  then  ERROR;
        backpatch(S1.CHAIN,nextstat);
        GETNEXT(TOKEN);
        (BE.true,BE.false):=BE(TOKEN);//调用BE返回两个出口（真、假出口）
        backpatch(BE.false,R.codebegin);
        S.CHAIN:=BE.true;
        return(S.CHAIN);
        end
end case;
end;
```

7.5.2 LL(1)语法制导翻译

- 基本思想：

  LL(1)分析法是让产生式右部逐个文法符号与输入串匹配，每当一个文法符号获得匹配，就可以执行语义动作。

- 实现办法：

  预先在源文法中的相应位置上嵌入语义动作符，当语法分析到达该位置时，调用与该动作符相应的语义动作。带有动作符的文法称为动作文法（Action Grammar）

例：文法:

L →D L | ε

D →a | b　　L代表的由a和b组成的串或空串

要构造的是这样一个语义处理器，它将输入L串并将其中b的个数打印出来。

动作文法:

L → D L
L → {Out}
D → a
D → b {Add}

动作符{Add}和{Out}对应的动作子程序分别如下：

Add : S:=S + 1
Out : Print(S)

LL(1)语法制导翻译的实现途径

- 控制程序增加识别动作符和调用语义动作的功能；每当动作符成为栈顶符号时，就执行相应的语义子程序
- 语义值的保存：增加语义栈。

  自顶向下的分析栈中的文法符号是待匹配的符号(无语义值)，一旦匹配(获得语义值)则弹出。因此用于保存语义值的语义栈必须单独操作。

动作文法:

L → D L

L → {Out}

D → a

D → b {Add}

动作符{Add}和{Out}对应的动作子程序分别如下：

Add : S:=S + 1

Out : Print(S)

输入符号串“bab”，LL(1)分析法实现上述动作文法的过程如下：

步骤	分析栈	剩余输入	产生式	动作
1	#L	bab#	L→DL	
2	#LD	bab#	D→b{A}	
3	#L{A}b	bab#	匹配b	
4	#L{A}	ab#		S:=S+1
5	#L	ab#	L→DL	
6	#LD	ab#	D→a	

动作文法:

L → D L

L → {Out}

D → a

D → b {Add}

动作符{Add}和{Out}对应的动作子程序分别如下:

Add : S:=S + 1

Out : Print(S)

7	#La	ab#	匹配a	
8	#L	b#	L→DL	
9	#LD	b#	D→b{A}	
10	#L{A}b	b#	匹配b	
11	#L{A}	#		S:=S+1
12	#L	#	L→{O}	
13	#{O}	#		Print(S)
14	#	#	接受	

例：while 语句的动作文法和语义子程序

$S \rightarrow$ while $\{w_1\}$ E $\{w_2\}$ do S^1 $\{w_3\}$

$\{w_1\}$： /* 记住入口位置 */

W.codebegin:=nextstat;

push W.codebegin;

语义栈

对应while {w1}匹配后

$\{w_2\}$：

/* E匹配后，E.true在栈顶，E.false在次栈顶 */

pop E.true;

backpatch(E.true,nextstat);

语义栈

对应while {w1}E匹配后

S → while {w_1} E {w_2} do S^1 {w_3}

{w_3}： /* S^1匹配后，S^1.CHAIN在栈顶 */

pop S^1.CHAIN;

pop E.false;

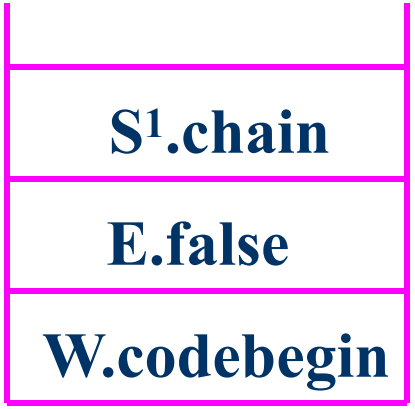
pop W.codebegin;

backpatch(S^1.CHAIN,W.codebegin);

emit(jump, -, -, W.codebegin);

S.CHAIN:=E.false;

push S.CHAIN;



语义栈

对应while {w_1}E{w_2}do S^1匹配后

语义栈

对应while {w1} E {w2} do S^1 {w3}匹配后

本章小结

属性文法和语法制导翻译：

- 简单赋值语句的翻译
- 布尔表达式的翻译
- 控制结构的翻译
- 简单说明语句的翻译

语法制导翻译总结

- 简单说明语句的翻译：不生成中间代码，只实现填表动作；
- 简单赋值语句的翻译：生成一个赋值四元式；
- 控制语句的翻译：if 条件表达式 then S
  while 条件表达式 do S
  不要忘记对语句的出口赋值：S.Chain
- 条件表达式的翻译：
  - 计算逻辑值：a<b（固定产生四个四元式
    (1) (j<,a,b,(4))
    (2) (:=,'0',-,t1)
    (3) (jump,-,-,(5))
    (4) (:=,'1',-,t1)
    (5)
  - 作为控制语句的条件表达式：翻译成条件转和无条件转四元式序列
    - E为a rop b: (jrop,a,b,E.True)
      (jump,-,-,E.False)
    - E为E1 or E2: 处理好出入口
      （三个属性：E.Codebegin、E.True、E.False）

语法制导和中间代码生成实验

- 1、文法
- 2、深刻理解语义
- 3、设计翻译目标
- 4、是否有困难，则改写文法
- 5、写出翻译子程序

语法制导翻译

- 自下而上语法制导翻译与自上而下语法制导翻译相同点：文法、语义、翻译目标相同
- 自下而上语法制导翻译与自上而下语法制导翻译不同点：

文法改写：自下而上：分割

自上而下：嵌入

编写语义子程序：语义栈独立操作（自上而下）