第五章 语法制导翻译技术和中间代码生成

本章主要介绍：

- 语法制导翻译法
- 常见的中间代码形式
- 各种不同语法结构的语法制导翻译技术

5.1 概述

➢静态语义审查

审查每个语法结构的静态语义，即验证语法结构合法的程序，是否真正有意义。

➢执行真正的翻译

如果静态语义正确，语义处理则要执行真正的翻译，即生成程序的某种中间代码的形式或直接生成目标代码。

5.2 属性文法

1. 属性文法

(1) 属性

文法符号配备属性值，代表与其相关的信息，如类型、值、存储位置等。属性值可以在语法分析过程中计算和传递。

属性分为综合属性和继承属性。

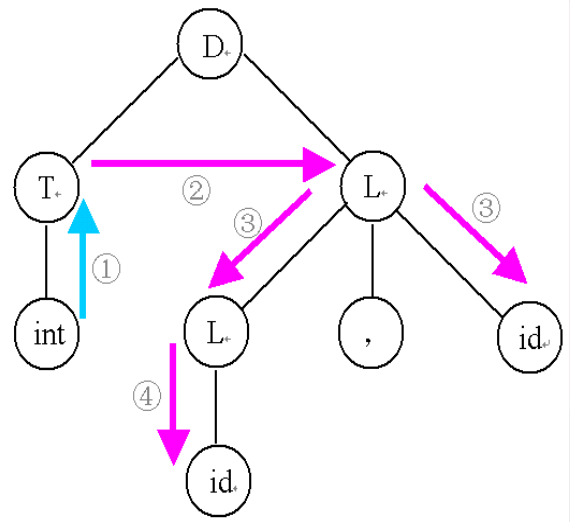
综合属性：“自下而上”

$E \rightarrow E^{(1)}+E^{(2)}$　　$E.val= E^{(1)}.val+ E^{(2)}.val$

5.2 属性文法

继承属性：“自上而下”

D→T L	L.in= T.type
T →int	T.type=integer
T →real	T.type=real
L →$L^{(1)}$, id	$L^{(1)}$.in=L.in; Fill(id.entry,L.in)
L →id	Fill(id.entry,L.in)



属性依赖：

（1）终结符只有综合属性，通过词法分析提供；

（2）非终结符既可有综合属性也可有继承属性，开始符号的继承属性作为属性计算的初始值。

5.2 属性文法

(2) 属性文法

属性文法包含一个上下文无关文法和一系列语义规则。为文法的每一个规则配备的处理属性的规则, 称为语义规则(描述语义处理的加工动作) 。

语义规则包括：属性计算、静态语义检查、符号表操作、代码生成等。

注：产生式左边的综合属性和右边的继承属性必须提供计算规则，产生式左边的继承属性和右边的综合属性不由所给产生式的属性计算规则计算，由其他产生式的属性规则计算或由参数提供。

5.3 语法制导翻译概述

语法制导的语义计算（由语法结构驱动）

输入串→语法树→按照语义规则计算属性

- 属性求值的一般性方法：依赖图
- S-属性文法和L-属性文法
- 语法制导的翻译模式
- 遍历语法树（注释语法树和带动作的语法树）
- 一遍扫描：在语法分析过程中完成语义计算

5.3 语法制导翻译概述

依赖图：描述了一棵语法树中属性实例之间的信息流

- 结点：语法树中每个结点所代表文法符号关联的属性
- 边：产生式关联的语义规则中的属性计算关系

如果不存在环，可以找到一个拓扑排序来进行属性值的计算

5.3 语法制导翻译概述

{7+8*5,3+8,6*5,…}

例如，设有简单算术表达

$$E \rightarrow E+E \mid E*E \mid (E) \mid digit$$

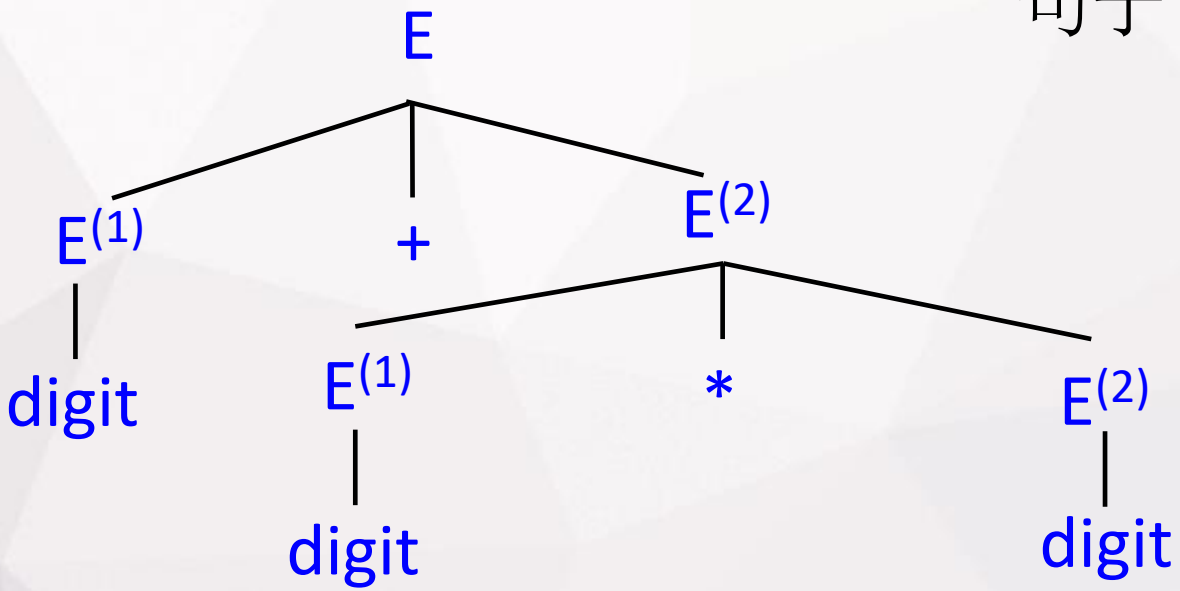
为文法每一产生式设计相应的求值的语义规则：

$1.E \rightarrow E^{(1)}+E^{(2)}$	$E.val = E^{(1)}.val + E^{(2)}.val$
$2.E \rightarrow E^{(1)}*E^{(2)}$	$E.val = E^{(1)}.val * E^{(2)}.val$
$3.E \rightarrow (E^{(1)})$	$E.val = E^{(1)}.val$
$4.E \rightarrow digit$	$E.val = Lex.digit$

5.3 语法制导翻译概述

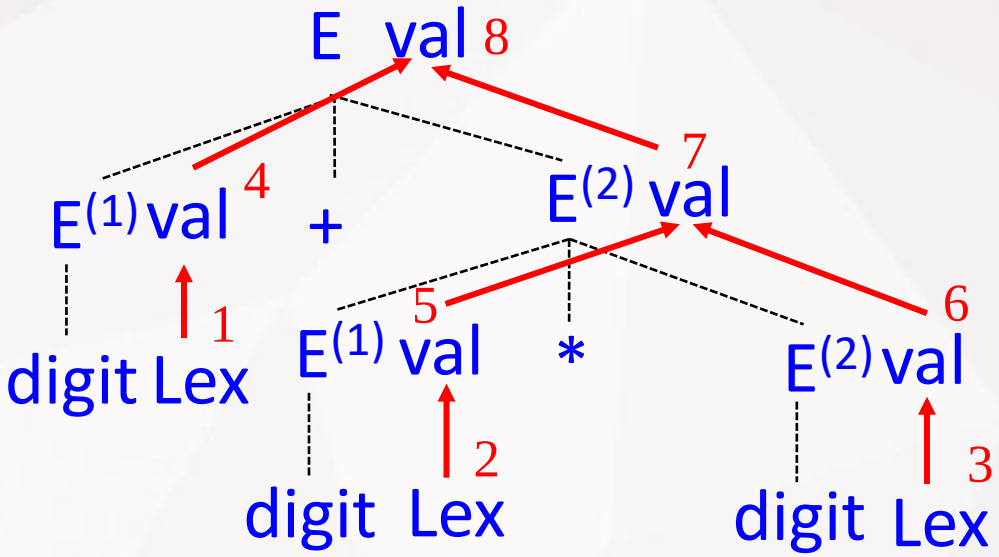
1.$E \rightarrow E^{(1)} + E^{(2)}$	$E.val = E^{(1)}.val + E^{(2)}.val$
2.$E \rightarrow E^{(1)} * E^{(2)}$	$E.val = E^{(1)}.val * E^{(2)}.val$
3.$E \rightarrow (E^{(1)})$	$E.val = E^{(1)}.val$
4.$E \rightarrow digit$	$E.val = Lex.digit$

句子 7+8*5

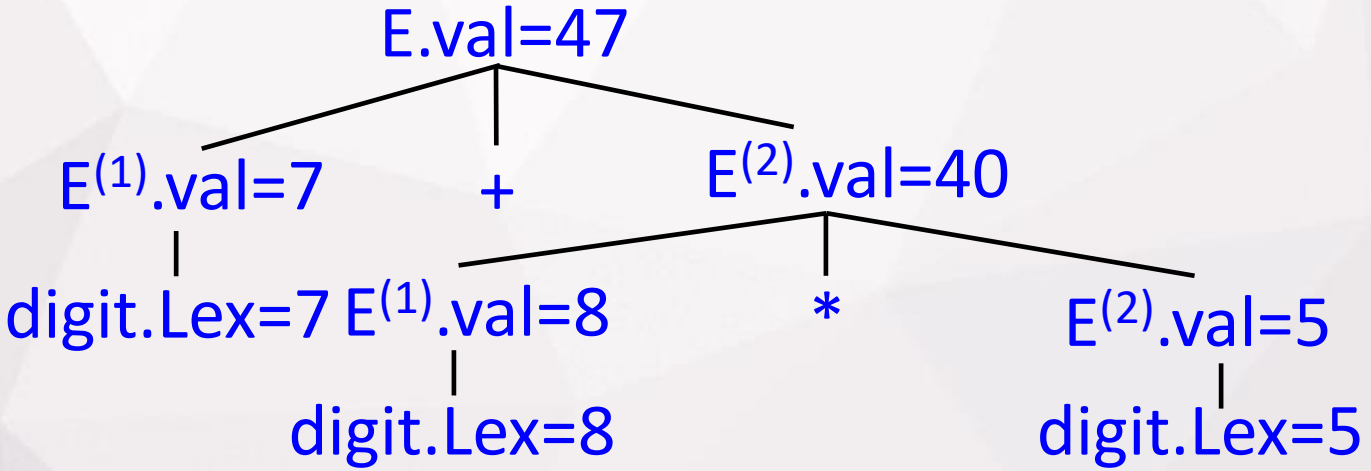


5.3 语法制导翻译概述

依赖图



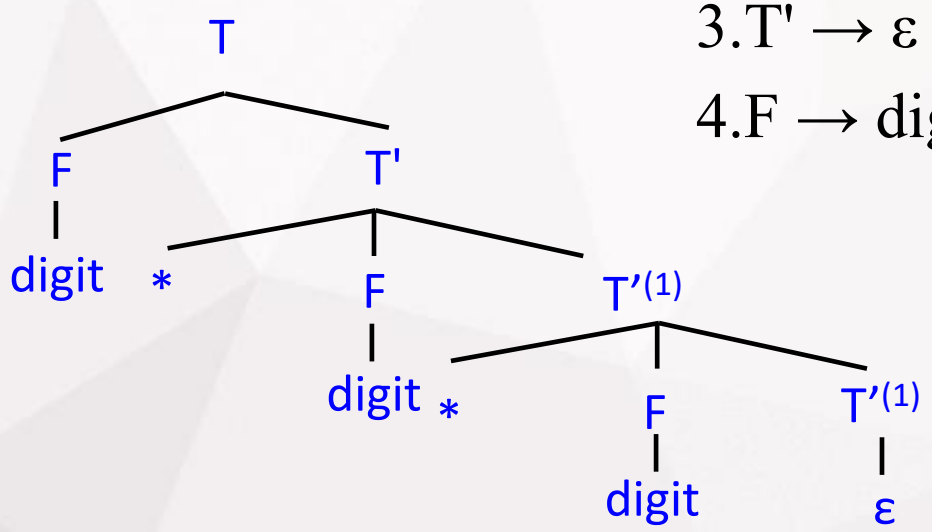
带注释的语法树



5.3 语法制导翻译概述

语法树结构和源码的抽象语法不匹配时，使用继承属性

句子 2*3*4



1.T → FT'	T'.i ＝F.v
	T.s ＝ T'.s
2.T'→ *FT'(1)	T'(1).i ＝ T'.i* F.v
	T'.s ＝ T'(1).s
3.T' → ε	T'.s ＝ T'.i
4.F → digit	F.v ＝Lex.digit

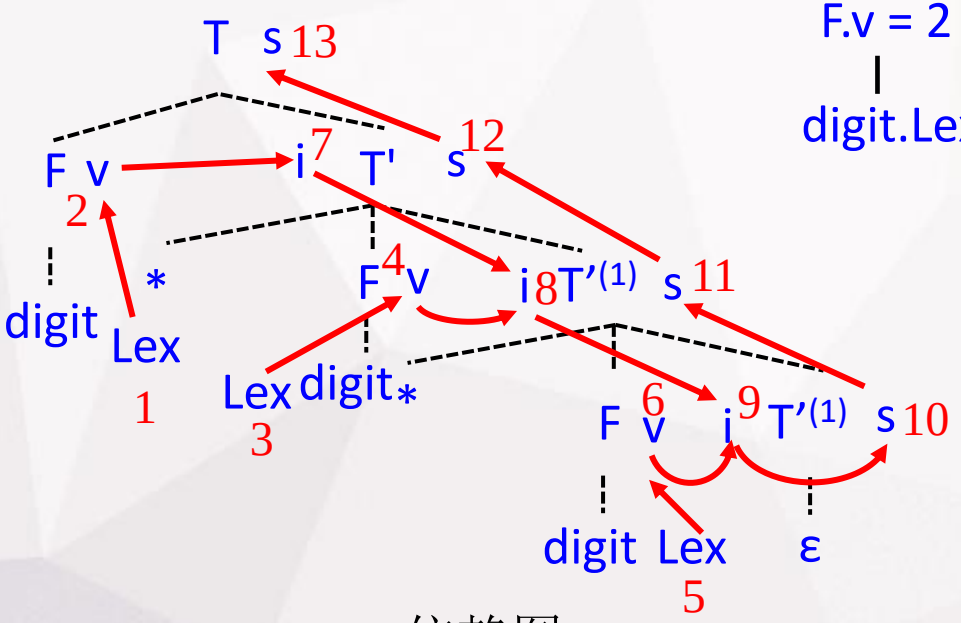
5.3 语法制导翻译概述

1.T → FT’　　T'.i =F.v，T.s = T'.s

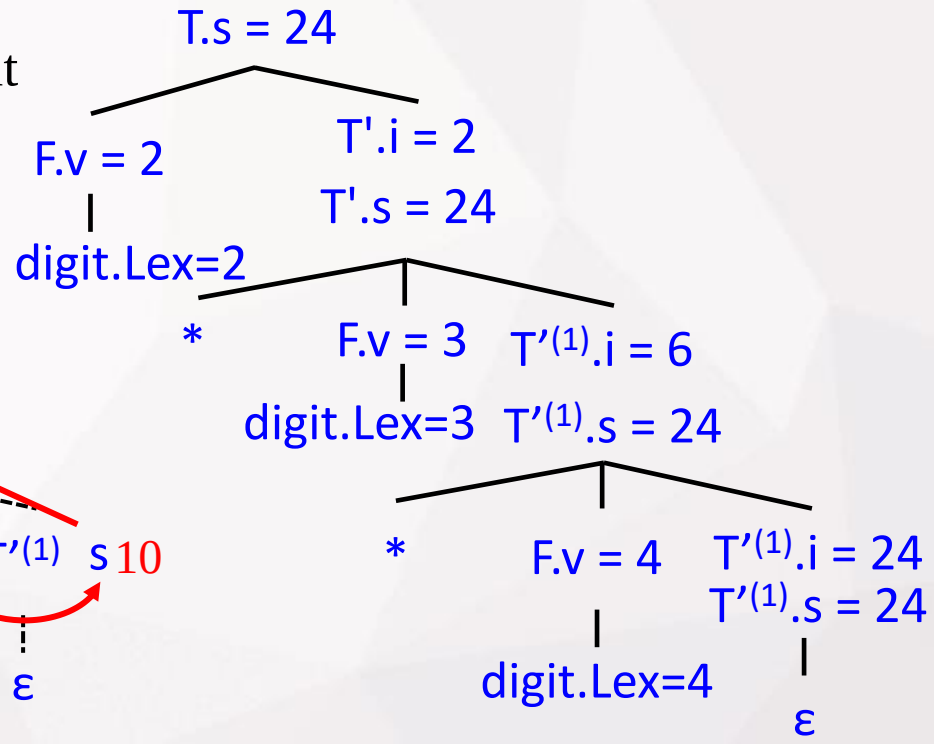
2.T'→ *FT' (1)　　T' (1).i = T'.i* F.v，T'.s = T'(1).s

3.T'→ ε　　T'.s = T'.i

4.F → digit　　F.v =Lex.digit



依赖图



带注释的语法树

5.3 语法制导翻译概述

S-属性文法：只含有综合属性的属性文法。

L-属性文法：如果对每个产生式 $A \rightarrow X_1X_2 \ldots X_n$,其语义规则中的属性或者为综合属性，或者为继承属性，若是$X_j(1 \leq j \leq n)$的继承属性，满足该继承属性仅依赖于：

（1）产生式X_j左边的$X_1X_2 \ldots X_{j-1}$的属性

（2）A的继承属性

S-属性文法是L-属性文法的一个特例。

5.3 语法制导翻译概述

A→BCD　.s-综合属性 .i-继承属性

A.s=B.i+C.s

A.s=B.i+C.s，D.i=A.i+B.s

A.s=B.s+C.s

A.s=D.i，B.i=A.s+C.s，C.i=B.s，D.i=B.i+C.i

（1）是否满足S-属性文法定义

（2）是否满足L-属性文法定义

（3）是否存在和规则一致的求值过程

5.3 语法制导翻译概述

语法制导的翻译模式：对属性文法的补充，面向实现的计算模式

- 与文法符号相关的属性和规则（称语义子动作）用{}括起来，可以插在产生式右部的任何位置
- 显式地给出了计算的顺序
- 属性文法转换成翻译模式或设计翻译模式的规则

5.3 语法制导翻译概述

1. $E \rightarrow E^{(1)} + E^{(2)}$ $E.val = E^{(1)}.val + E^{(2)}.val$
2. $E \rightarrow E^{(1)} * E^{(2)}$ $E.val = E^{(1)}.val * E^{(2)}.val$
3. $E \rightarrow (E^{(1)})$ $E.val = E^{(1)}.val$
4. $E \rightarrow digit$ $E.val = Lex.digit$

⇨

1. $E \rightarrow E^{(1)} + E^{(2)}$ $\{E.val = E^{(1)}.val + E^{(2)}.val\}$
2. $E \rightarrow E^{(1)} * E^{(2)}$ $\{E.val = E^{(1)}.val * E^{(2)}.val\}$
3. $E \rightarrow (E^{(1)})$ $\{E.val = E^{(1)}.val\}$
4. $E \rightarrow digit$ $\{E.val = Lex.digit\}$

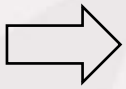
5.3 语法制导翻译概述

1.$T \rightarrow FT'$ $\quad T'.i = F.v,\ T.s = T'.s$

2.$T' \rightarrow *FT'^{(1)}$ $\quad T'^{(1)}.i = T'.i * F.v,\ T'.s = T'^{(1)}.s$

3.$T' \rightarrow \varepsilon$ $\quad T'.s = T'.i$

4.$F \rightarrow digit$ $\quad F.v = Lex.digit$



1.$T \rightarrow F\ \{T'.i = F.v\}\ T'\ \{T.s = T'.s\}$

2.$T' \rightarrow *F\ \{T'^{(1)}.i = T'.i * F.v\}\ T'^{(1)}\ \{T'.s = T'^{(1)}.s\}$

3.$T' \rightarrow \varepsilon\ \{T'.s = T'.i\}$

4.$F \rightarrow digit\ \{F.v = Lex.digit\}$

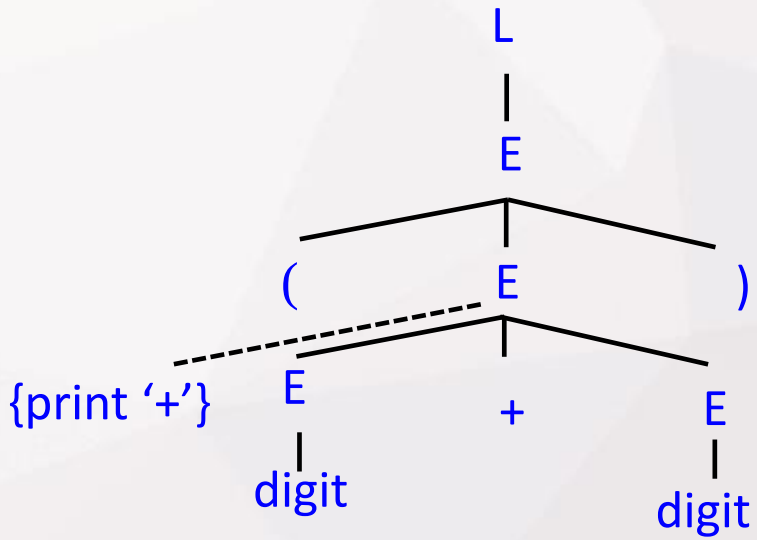
5.3 语法制导翻译概述

遍历语法树

- 注释语法树
- 构造语法树并加入动作：先根遍历

eg. 无法与语法分析同时完成

1. $L \rightarrow E$
2. $E \rightarrow \{print\ '+'\}E^{(1)}+E^{(2)}$
3. $E \rightarrow (E^{(1)})$
4. $E \rightarrow digit\ \{print\ Lex.digit\}$



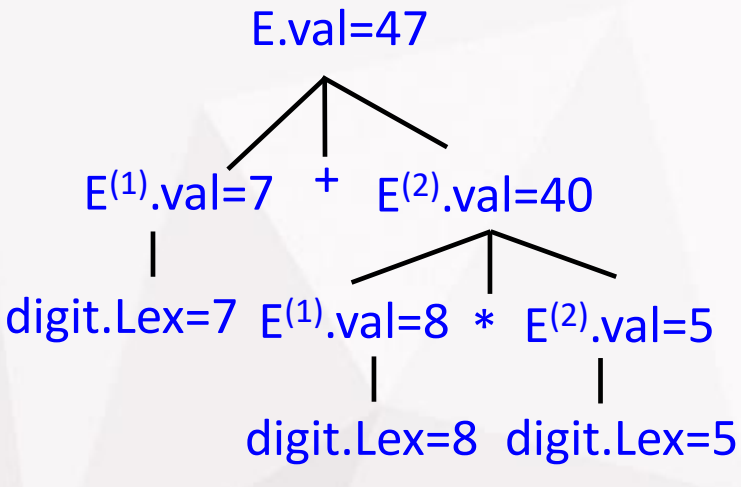
5.3 语法制导翻译概述

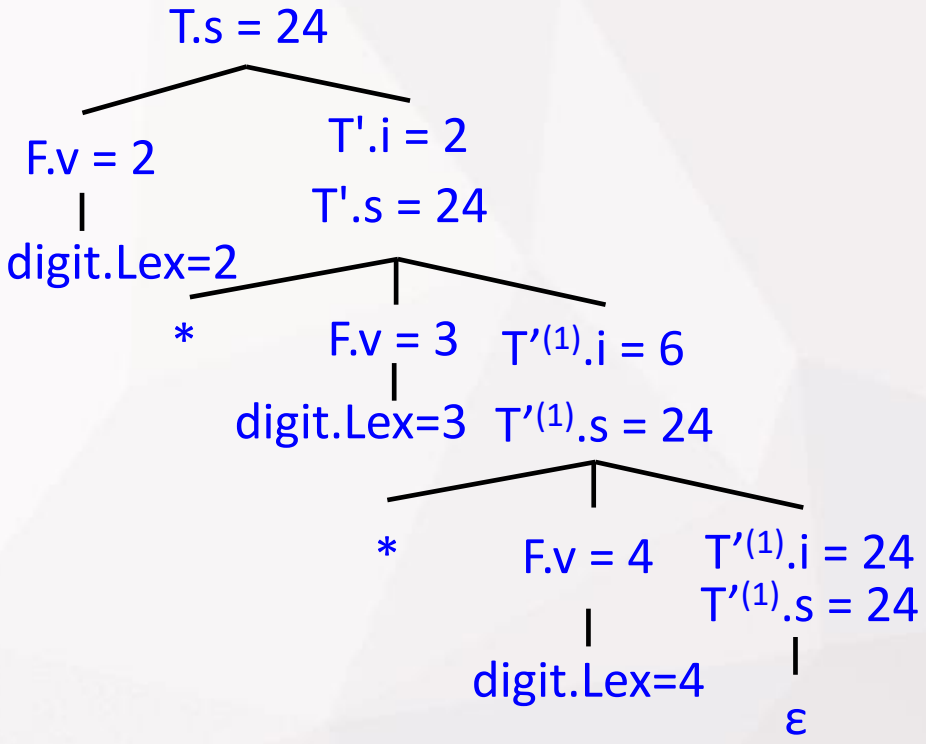
L-属性文法的树遍历

```
VisitNode (N:Node) ;
{
   if  N∈V_N  /*假设其产生式为N→X_1…X_m*/
      for i :=1 to  m  do
          if  X_i∈V_N  /*即X_i是非终结符*/
           {
                  计算X_i的所有继承属性;
                  VisitNode (X_i)
           }
     计算N的综合属性;
}
```

5.3 语法制导翻译概述

举例





5.3 语法制导翻译概述

一遍扫描，在语法分析的同时完成语义计算

- 采用的语法分析方法和属性的计算顺序
- S-属性文法适合一遍扫描的自下而上的语法分析方法
- L-属性文法适合一遍扫描的自上而下的语法分析方法

5.3 语法制导翻译概述

S-属性文法的自下而上计算

- 扩充LR分析栈，以便存放文法符号对应的语义值

S_k	X_k	X_k.val
…	…	…
S_1	X_1	X_1.val
S_0	$	—
状态栈	文法符号栈	语义值栈

- 修改总控程序：查分析表, 当用某产生式归约时, 调用相应的语义动作

5.3 语法制导翻译概述

1. $E \rightarrow E^{(1)} + E^{(2)}$ {E.val = $E^{(1)}$.val + $E^{(2)}$.val}
2. $E \rightarrow E^{(1)} * E^{(2)}$ {E.val = $E^{(1)}$.val*$E^{(2)}$.val}
3. $E \rightarrow (E^{(1)})$ {E.val = $E^{(1)}$.val}
4. $E \rightarrow$ digit {E.val = Lex.digit}

状态	ACTION						GOTO
	+	digit	*	(	)	$	E
0		s_3		s_2			1
1	s_4		s_5			acc	
2		s_3		s_2			6
3	r_4		r_4		r_4	r_4	
4		s_3		s_2			7
5		s_3		s_2			8
6	s_4		s_5		s_9		
7	r_1		s_5		r_1	r_1	
8	r_2		r_2		r_2	r_2	
9	r_3		r_3		r_3	r_3	

5.3 语法制导翻译概述

S-属性文法：自上而下计算

- 消除翻译模式中的左递归

eg1.

1. $E \rightarrow E^{(1)}+T$ {print '+'}
2. $E \rightarrow T$

⇨

1. $E \rightarrow TE'$
2. $E' \rightarrow +T$ {print '+'} E'
3. $E' \rightarrow \varepsilon$

eg2.

1. $A \rightarrow A^{(1)}Y$ {$A.s=g(A^{(1)}.s,Y.s)$}
2. $A \rightarrow X$ {$A.s=f(X.s)$}

⇨

1. $A \rightarrow X$ {$A'.i=f(X.s)$} A' {$A.s=A'.s$}
2. $A' \rightarrow Y$ {$A'^{(1)}.i=g(A'.i,Y.s)$} $A'^{(1)}$ {$A'.s=A'^{(1)}.s$}
3. $A' \rightarrow \varepsilon$ {$A'.s=A'.i$}

5.3 语法制导翻译概述

生成抽象语法树

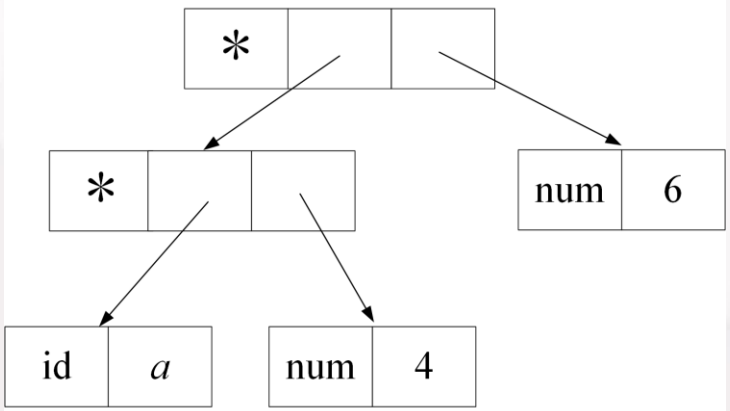
1.T → $T^{(1)}$*F　{T.nptr=mknode('*', $T^{(1)}$.nptr, F.nptr)}

2.T → F　　{T.nptr=F.nptr}

3.F → （T）　{F.nptr=T.nptr}

4.F → id　　{F.nptr=mkleaf(id, id.entry)}

5.F → num　　{F.nptr=mkleaf(num, num.val)}



*a**4*6的抽象语法树示例

5.3 语法制导翻译概述

1.T → $T^{(1)}$*F {T.nptr=mknode('*', $T^{(1)}$.nptr, F.nptr)}

2.T → F {T.nptr=F.nptr}

3.F → （T） {F.nptr=T.nptr}

4.F → id {F.nptr=mkleaf(id, id.entry)}

5.F → num {F.nptr=mkleaf(num, num.val)}

1.T → F {T'.i =F.nptr} T' {T.nptr = T'.s}

2.T'→ *F {$T'^{(1)}$.i = mknode('*', T '.i, F.nptr)} $T'^{(1)}$ {T'.s = $T'^{(1)}$.s}

3.T'→ ε {T'.s = T'.i}

4.F → （T） {F.nptr =T.nptr}

5.F → id {F.nptr=mkleaf(id, id.entry)}

6.F → num {F.nptr=mkleaf(num, num.val)}

5.3 语法制导翻译概述

L-属性文法

- 递归下降分析
- LL预测分析
- LR语法分析

5.3 语法制导翻译概述

L-属性文法

递归下降分析：以继承属性作为参数，返回综合属性

1.T → F {T'.i =F.v} T’

{T.s = T’.s}

2.T'→ *F {T'(1).i = T'.i* F.v}

T'(1) {T'.s = T'(1).s}

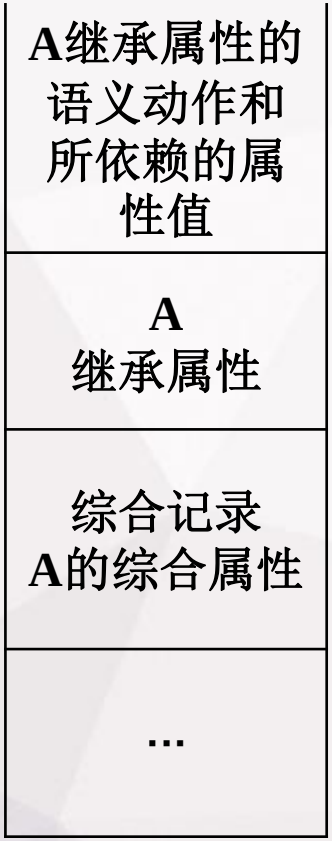
3.T'→ ε {T'.s = T'.i}

4.F → digit {F.v =Lex.digit}

```
T’( i );
{
  if  sym=‘*’
  {   scanner( );
      tmp_v = F( );
      tmp_i = i * tmp_v;
      tmp_s = T’(tmp_i);
      return tmp_s;
   }
   else  if sym ∈ Follow (T’)
          {  scanner( );
             return i ;
          else error( );
}
```

5.3 语法制导翻译概述

自上而下LL预测分析：扩展分析栈（A为V_N）

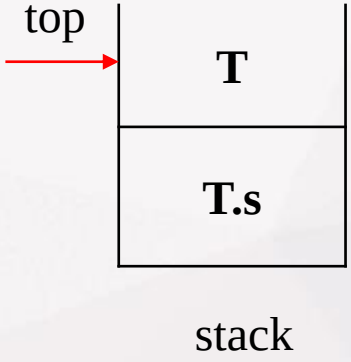


1.T → F {T'.i =F.v} T' {T.s = T'.s}

2.T'→ *F {T'(1).i = T'.i* F.v} T'(1) {T'.s = T'(1).s}

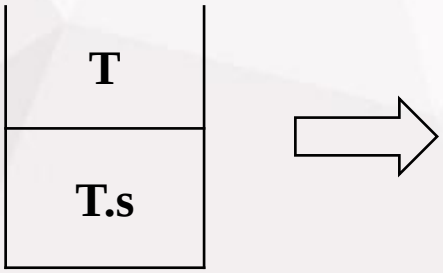
3.T'→ ε {T'.s = T'.i}

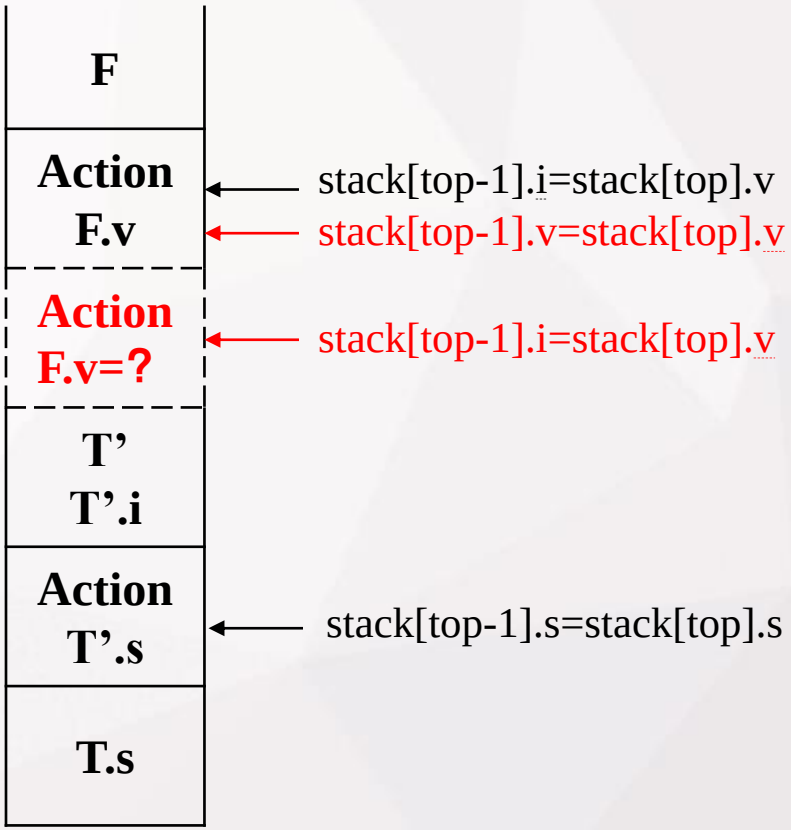
4.F → digit {F.v =Lex.digit}



5.3 语法制导翻译概述

1.T → F {T'.i =F.v} T'
{T.s = T'.s}
2.T'→ *F {T'(1).i = T'.i* F.v}
T'(1) {T'.s = T'(1).s}
3.T'→ ε {T'.s = T'.i}
4.F → digit {F.v =Lex.digit}





5.3 语法制导翻译概述

LR语法分析：添加非终结符来替代内嵌的语义子动作，将需要传递而计算得到的继承属性作为M的综合属性来传递

$$A \rightarrow \alpha \{ a \} \beta \Longrightarrow A \rightarrow \alpha M \beta , M \rightarrow \varepsilon$$

1.T → F {T'.i =F.v} T'
{T.s = T'.s}
2.T'→ *F {T'(1).i = T'.i* F.v}
T'(1) {T'.s = T'(1).s}
3.T'→ ε {T'.s = T'.i}
4.F → digit {F.v =Lex.digit}

1.T → F M T' {T.s = T'.s}
2.M → ε { M.s = F.v } （可省略）
3.T'→ *F N T'(1) {T'.s = T'(1).s}
4.N → ε { N.s = T'.i* F.v }
5.T'→ ε {T'.s = T'.i}
6.F → digit {F.v =Lex.digit}

5.4 中间语言

编译中常见的中间语言：

- 逆波兰式（后缀式）
- 三元式
- 四元式
- 树形表示（图）

5.4.1 逆波兰式

逆波兰式

逆波兰式除去了原表达式中的括号，并将运算对象写在前面，运算符写在后面，因而又称为**后缀式**。

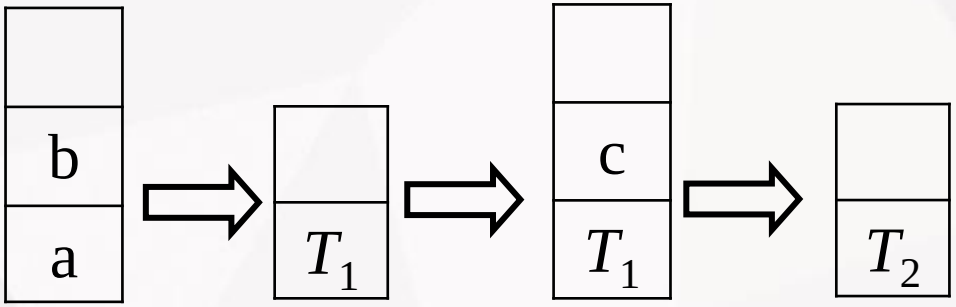
例如：	中缀表达式	逆波兰式
	a*b	ab*
	(a+b)*(c+d)	ab+cd+*

1.不再有括号,且运算符出现的顺序体现了中缀表达式 的运算顺序

2. 易于计算机处理

5.4.1 逆波兰式

逆波兰式ab+c*的处理过程：



逆波兰形式可以推广到其他语法结构：

赋值语句	逆波兰式
V=E	VE=
条件语句	逆波兰式
if E S_1 ; else S_2	E$S_1$$S_2$￥

5.4.1 逆波兰式

$E \rightarrow E + E \mid E * E \mid (E) \mid i$

$0.E' \rightarrow E$	{ 空 }
$1.E \rightarrow E^{(1)} + E^{(2)}$	{ print + }
$2.E \rightarrow E^{(1)} * E^{(2)}$	{ print * }
$3.E \rightarrow (E^{(1)})$	{ 空 }
$4.E \rightarrow i$	{ print i }

$E^{(1)}.code \parallel E^{(2)}.code \parallel +$

$E^{(1)}.code \parallel E^{(2)}.code \parallel *$

5.4.2 三元式和树形表示

三元式：（OP，arg1, arg2）

例如　a+b*c 的 三元式序列：

(1) (* , b , c)

(2) (+ , a , (1))

三元式的特点：

1. 三元式出现的顺序和语法成份的计值顺序相一致。

2. 三元式之间的联系是通过指示器实现的。

5.4.2 三元式和树形表示

间接三元式

例如　语句　X= (A+B)*C　　Y= D↑(A+B)

三元式序列

(1) (+ , A , B)

(2) (* , (1) , C)

(3) (= , X , (2))

(4) (+ , A , B)

(5) (↑, D , (4))

(6) (= , Y, (5))

间接三元式

间接码表

(1)

(2)

(3)

(1)

(4)

(5)

三元式表

(1) (+ , A , B)

(2) (* , (1) , C)

(3) (= , X , (2))

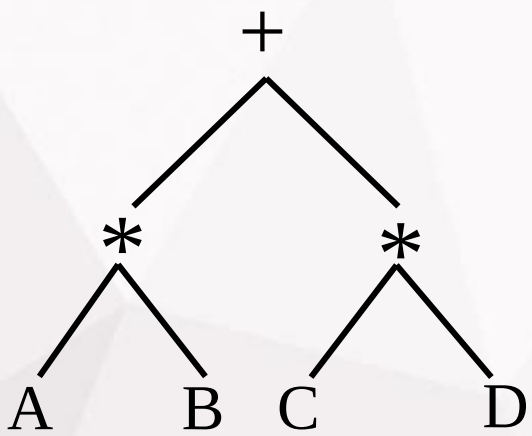
(4) (↑, D , (1))

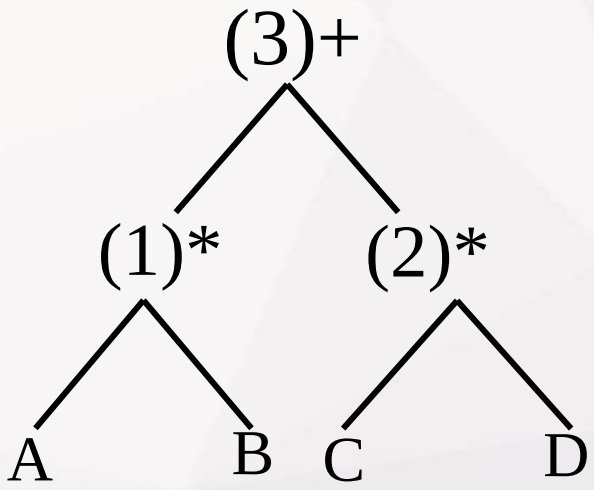
(5) (= , Y, (4))

5.4.2 三元式和树形表示

树形表示（抽象语法树和DAG图）

A * B + C*D





5.4.3 四元式

四元式：（OP，arg1, arg2, result）

X＝ a*b＋c/d 的 四元式序列

(1)	(*,	a,	b,	T_1)
(2)	(/,	c,	d,	T_2)
(3)	(+,	T_1,	T_2,	T_3)
(4)	(=,	T_3,	−,	X)

三地址代码

result :＝ arg1 OP arg2

（1）T_1＝a*b

（2）T_2＝c/d

（3）T_3＝T_1＋T_2

（4）X＝T_3

1. 四元式出现的顺序和语法成份的计值顺序相一致。
2. 易于调整和变动四元式。
3. 便于优化处理。

– 静态单赋值(SSA)，程序中的名字只有一次赋值。

```
x ← 5
x ← x - 3
if x < 3
then
    y ← x * 2
    w ← y
else
    y ← x - 3
w ← x - y
z ← x + y
```

$$
\begin{aligned}
&x_1 \leftarrow 5\\
&x_2 \leftarrow x_1 - 3\\
&\text{if } x_2 < 3\\
&\text{then}\\
&\quad y_1 \leftarrow x_2 * 2\\
&\quad w_1 \leftarrow y_1\\
&\text{else}\\
&\quad y_2 \leftarrow x_2 - 3\\
&y_3 \leftarrow \phi(y_1, y_2)\\
&w_2 \leftarrow x_2 - y_3\\
&z \leftarrow x_2 + y_3
\end{aligned}
$$

简单算术表达式和赋值语句到四元式的翻译

$A \rightarrow i = E$　　$E \rightarrow E+E \mid E*E \mid (E) \mid i$

源结构

$X = a*b+c*d$

目标结构

（1）$T_1 = a*b$

（2）$T_2 = c*d$

（3）$T_3 = T_1 + T_2$

（4）$X = T_3$

语义属性：E.place

语义函数/过程：emit（T=arg1 op arg2)

newtemp（）

lookup（i.name)

error（）

简单算术表达式和赋值语句到四元式的翻译

1. A→i = E

```
{  p = Lookup (i.name);
   if ( P != NULL )  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. $E \rightarrow E^{(1)} * E^{(2)}$

```
{  E.place = newtemp( );
   emit(E.place'='  E(1).place'*'E(2).place )
}
```

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

```
{  E.place = E(1).place;  }
```

5. E → i

```
{  p = Lookup (i.name);
   if  (p != NULL)  E.place = p; else error( );
}
```

布尔表达式到四元式的翻译

$E \rightarrow E^{(1)} \vee E^{(2)}$

$E \rightarrow E^{(1)} \wedge E^{(2)}$

$E \rightarrow \neg E^{(1)}$

$E \rightarrow (E^{(1)})$

$E \rightarrow i^{(1)}\ rop\ i^{(2)}$

$E \rightarrow true$

$E \rightarrow false$

} $E \rightarrow i$

作用: 计算逻辑值

控制语句的条件式

布尔表达式的计值方法

1. 直接仿照算术表达式的计算进行计值;

2. 用作控制流，通过转移到某个位置来表示其求值结果。

$A \vee B$ 解释成 if A then true else B

$A \wedge B$ 解释成 if A then B else false

$\neg A$ 解释成 if A then false else true

布尔表达式到四元式的翻译

直接求值

1. $E \rightarrow E^{(1)} \vee E^{(2)}$ { E.place = newtemp( );
   emit (E.place'=' $E^{(1)}$.place'$\vee$'$E^{(2)}$.place) }
2. $E \rightarrow E^{(1)} \wedge E^{(2)}$ { E.place = newtemp( );
   emit (E.place'=' $E^{(1)}$.place'$\wedge$'$E^{(2)}$.place) }
3. $E \rightarrow \neg E^{(1)}$ { E.place = newtemp( );
   emit (E.place'=' $\neg$ $E^{(1)}$.place' }
4. $E \rightarrow (E^{(1)})$ { E.place = $E^{(1)}$.place; }
5. $E \rightarrow i^{(1)}$ rop $i^{(2)}$ { E.place = newtemp( );
   emit ('if ' i $^{(1)}$.place rop.op i $^{(2)}$.place 'goto' next+3);
   emit (E.place'=' '0');
   emit (goto next+2);
   emit (E.place'=' '1'); }

next表示下一条四元式的序号

布尔表达式到四元式的翻译

6. E→ true　　{ E.place ＝ newtemp( );
emit (E.place’＝’　‘1’) }

7. E→ false　　{ E.place ＝ newtemp( );
emit (E.place’＝’　‘0’) }

或

6. E→ i　　{ E.place ＝ i . place； }

布尔表达式到四元式的翻译

例如布尔表达式 $a < b \vee c < d \wedge e > f$

100 if a < b goto 103

101 $t_1 = 0$

102 goto 104

103 $t_1 = 1$

104 if c < d goto 107

105 $t_2 = 0$

106 goto 108

107 $t_2 = 1$

108 if e > f goto 111

109 $t_3 = 0$

110 goto 112

111 $t_3 = 1$

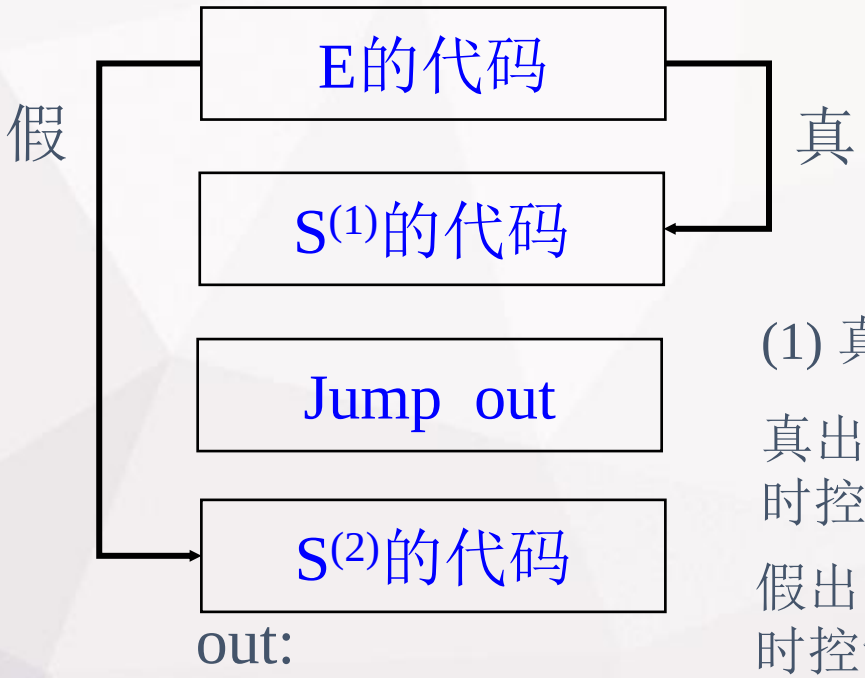
112 $t_4 = t_2 \wedge t_3$

113 $t_5 = t_1 \vee t_4$

布尔表达式到四元式的翻译

控制语句中布尔表达式的翻译

条件语句的语法为:if E then $S^{(1)}$ else $S^{(2)}$

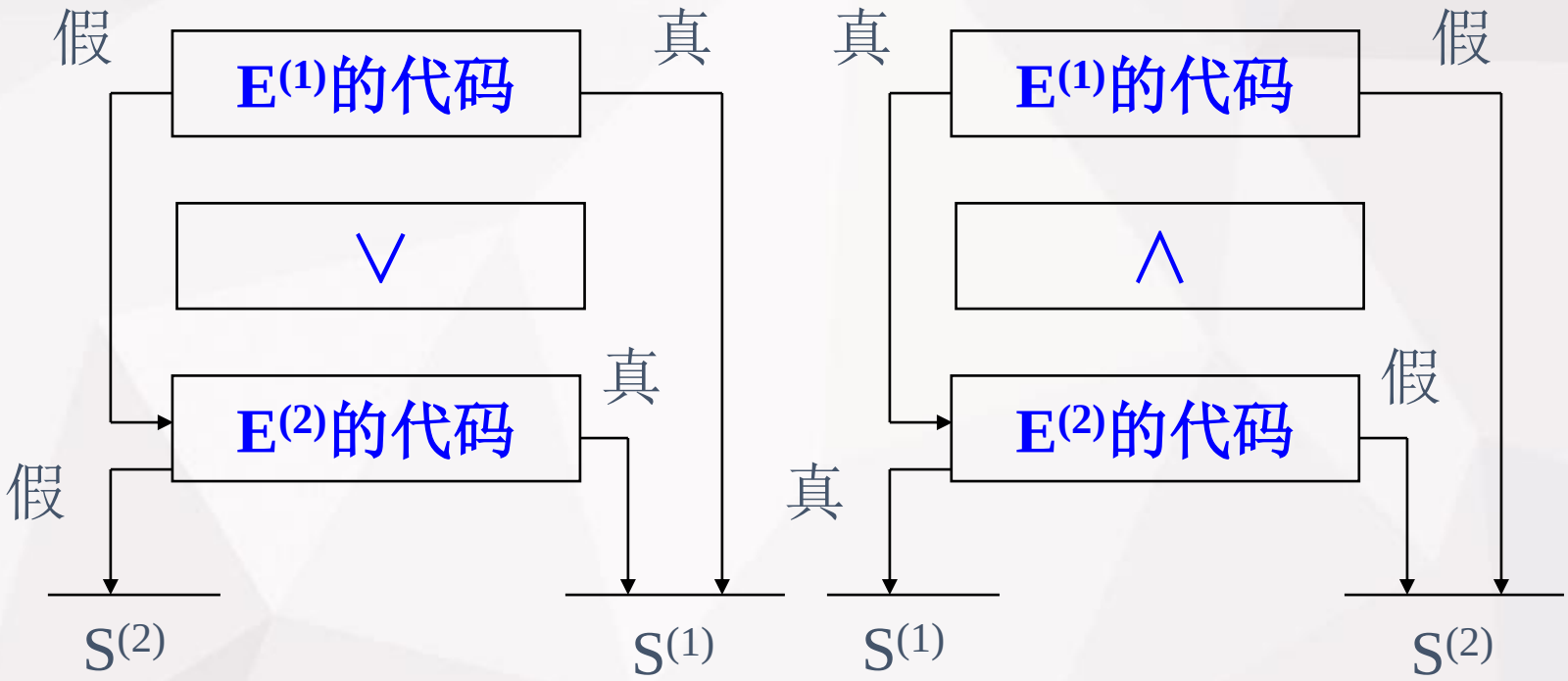


(1) 真出口和假出口

真出口表示布尔表达式E为真时控制流向的转移目标

假出口表示布尔表达式E为假时控制流向的转移目标

布尔表达式到四元式的翻译



(2) 作为条件转移的E，翻译成的代码是一串条件跳转或无条件跳转的四元式

if a rop b goto 真出口

goto 假出口

布尔表达式到四元式的翻译

例如语句 if a < b ∨ c < d ∧ e > f then $S^{(1)}$ else $S^{(2)}$

100 if a < b goto 106
101 goto 102
102 if c < d goto 104
103 goto (p+1)
104 if e > f goto 106
105 goto (p+1)
106 (关于 $S^{(1)}$ 的四元式)
……
(p) goto (q)
(p+1) (关于 $S^{(2)}$ 的四元式)
……
(q)

(3) 定义属性

E·tr: 记录表达式 E 所对应的四元式需回填真出口的四元式的地址所构成的链

E·fa: 记录表达式 E 所对应的四元式需回填假出口的四元式的地址所构成的链

E·code: 记录表达式 E 所对应第一条四元式的序号

next

布尔表达式到四元式的翻译

例如语句 if a < b ∨ c < d ∧ e > f then $S^{(1)}$ else $S^{(2)}$

100 if a < b goto 106
101 goto 102
102 if c < d goto 104
103 goto (p+1)
104 if e > f goto 106
105 goto (p+1)
106 (关于 $S^{(1)}$ 的四元式)
……
(p) goto (q)
(p+1) (关于 $S^{(2)}$ 的四元式)
……
(q)

$E^{(1)}$·tr=100
$E^{(1)}$·fa=101
$E^{(2)}$·tr=102
$E^{(2)}$·fa=103
$E^{(3)}$·tr=104
$E^{(3)}$·fa=105

(4) 定义函数
merg()
bp()

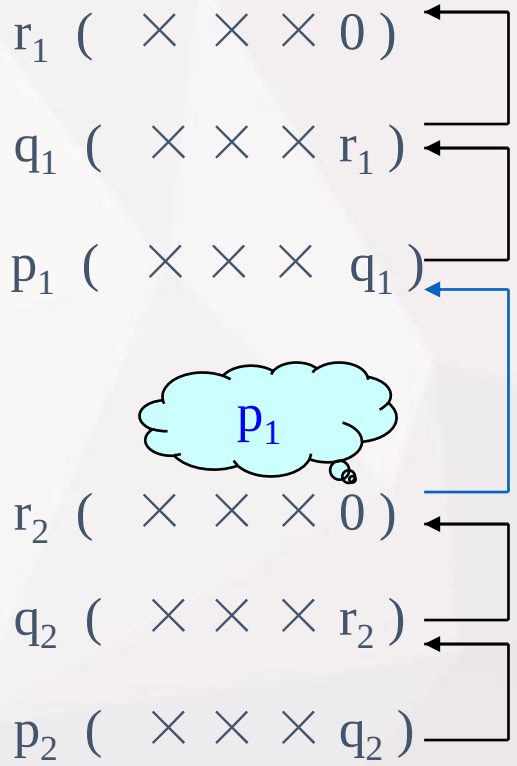
$E^{(2)}$ ∧ $E^{(3)}$归约E'
E'·tr=104
E'·fa=103和105所构成的链
$E^{(1)}$ ∨ E' 归约E时
E·tr=100和104所构成的链
E·fa= E'·fa

布尔表达式到四元式的翻译

链接函数

merg (int p_1, int p_2)

```
{
    if  p2=0  return( p1 ) ;
    else
     {  p= p2 ;
        while ( 四元式p的第四分量不为0)
        p=四元式p的第四分量;
        把p1填进四元式p的第四分量;
         return( p2 ) ;
     }
}
```



布尔表达式到四元式的翻译

回填函数

```
bp ( int p, int t )
{     q=p;
      while (q != 0 )
      {  r = 四元式 q 的第四分量;
         把 t 填进四元式 q 的第四分量;
         q=r ;
      }
 }
```

布尔表达式到四元式的翻译

bp (104, 106)

100 if a < b goto 0
101 goto 102
102 if c < d goto 104
103 goto 0
104 if e > f goto 100
105 goto 103
106 ($S^{(1)}$ 的四元式)
……
(p) goto (q)
(p+1) ($S^{(2)}$ 的四元式)
……
(q)

100 if a < b goto 106
101 goto 102
102 if c < d goto 104
103 goto 0
104 if e > f goto 106
105 goto 103
106 ($S^{(1)}$ 的四元式)
……
(p) goto (q)
(p+1) ($S^{(2)}$ 的四元式)
……
(q)

布尔表达式到四元式的翻译

1. $E \rightarrow E^{(1)} \vee E^{(2)}$

{ bp($E^{(1)}$.fa, $E^{(2)}$.code) ;
E.code= $E^{(1)}$.code ;
E.fa= $E^{(2)}$.fa ;
E.tr=merg($E^{(1)}$.tr, $E^{(2)}$.tr) ;}

2. $E \rightarrow E^{(1)} \wedge E^{(2)}$

{ bp($E^{(1)}$.tr, $E^{(2)}$.code) ;
E.code= $E^{(1)}$.code ;
E.tr= $E^{(2)}$.tr ;
E.fa=merg($E^{(1)}$.fa, $E^{(2)}$.fa) ;}

3. $E \rightarrow \neg E^{(1)}$

{ E.code= $E^{(1)}$.code ;
E.tr=$E^{(1)}$.fa ;
E.fa= $E^{(1)}$.tr ;}

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

{ E.code= $E^{(1)}$.code ;
E.tr=$E^{(1)}$.tr ;
E.fa= $E^{(1)}$.fa ;}

布尔表达式到四元式的翻译

5. E→ i (1) rop i (2)　　{ E.tr= next ;
E.code= next ;
E.fa= next+1;
emit (if i (1).place rop.op i (2).place goto _) ;
emit(goto _) ;}

6. E→ true　　{ E.tr= next ;
E.code= next ;
emit(goto _) ;}

7. E→ false　　{ E.fa= next ;
E.code= next ;
emit(goto _) ; }

6. E→ true
7. E→ false
} 6. E→ i　{ E.tr= next ; E.fa= next +1; E.code= next ;
emit(if i.place goto -);
emit(goto _) ;}

布尔表达式到四元式的翻译

布尔表达式 $a < b \vee c < d \wedge e > f$ 的翻译过程:

$a < b \vee c < d \wedge e > f$

$E^{(1)} \vee c < d \wedge e > f$

{ $E^{(1)}$.tr=100 ;
  $E^{(1)}$.fa=101 ;
  $E^{(1)}$.code=100 ; }

100 if a < b goto 0

101 goto 0

$E^{(1)} \vee E^{(2)} \wedge e > f$

{ $E^{(2)}$.tr=102 ;
  $E^{(2)}$.fa=103 ;
  $E^{(2)}$.code=102 ; }

102 if c < d goto 0

103 goto 0

布尔表达式到四元式的翻译

$a < b \vee c < d \wedge e > f$

$E^{(1)} \vee E^{(2)} \wedge E^{(3)}$

{ $E^{(3)}.tr=104$;
  $E^{(3)}.fa=105$;
  $E^{(3)}.code=104$; }

$E^{(1)} \vee E'$

{ bp($E^{(2)}.tr$, $E^{(3)}.code$)
  $E'.tr=E^{(3)}.tr=104$;
  $E'.fa=merg(E^{(2)}.fa, E^{(3)}.fa)=105$;
  $E'.code=E^{(2)}.code=102$
}

100 if a < b goto 0
101 goto 0
102 if c < d goto 0
103 goto 0
104 if e > f goto 0
105 goto 0

104

103

布尔表达式到四元式的翻译

a < b ∨ c < d ∧ e > f

E (1) ∨ E'

E

{ bp(E(1).fa, E'.code);
  E.fa=E'.fa=105 ;
  E.tr=merg(E(1).tr, E'.tr)
 E.code= E(1).code=100
}

100 if a < b goto 0

101 goto 0

102

102 if c < d goto 104

103 goto 0

100

104 if e > f goto 0

105 goto 103

布尔表达式到四元式的翻译

$a < b \vee c < d \wedge e > f$

E.tr=104

E.fa=105

E.code=100

100 if a < b goto 0

101 goto 102

102 if c < d goto 104

103 goto 0

104 if e > f goto 100

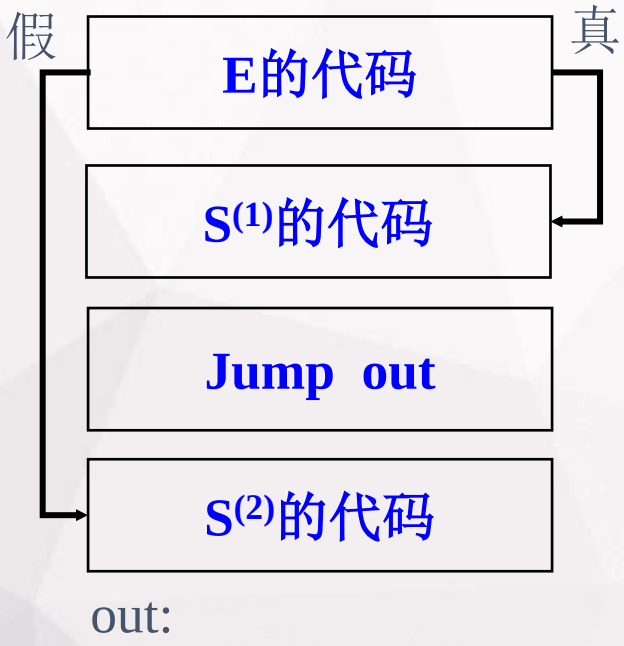
105 goto 103

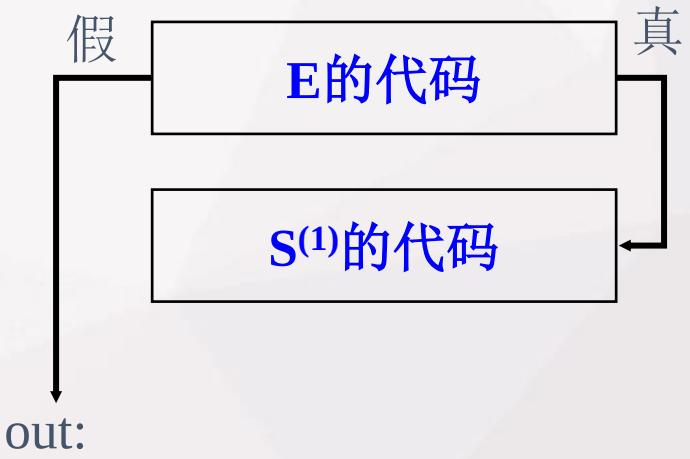
控制语句的翻译

条件语句的翻译

S→if E then S
S→if E then S else S

S→if E then M S
S→if E then M S N else M S
M→ε
N→ε





控制语句的翻译

(1) S→if E then M $S^{(1)}$ { bp(E.tr, M.s);
S.chain=merg(E.fa, $S^{(1)}$.chain)}

(2) S→if E then M $S^{(1)}$ N else $M^{(1)}$ $S^{(2)}$
{ bp(E.tr, M.s);
bp(E.fa, $M^{(1)}$.s);
tmp=merge($S^{(1)}$.chain, N.s);
S.chain=merg(tmp, $S^{(2)}$.chain)}

(3) M→ε { M.s=next; }

(4) N→ε { N.s=mklist(next);
emit('goto' —); }

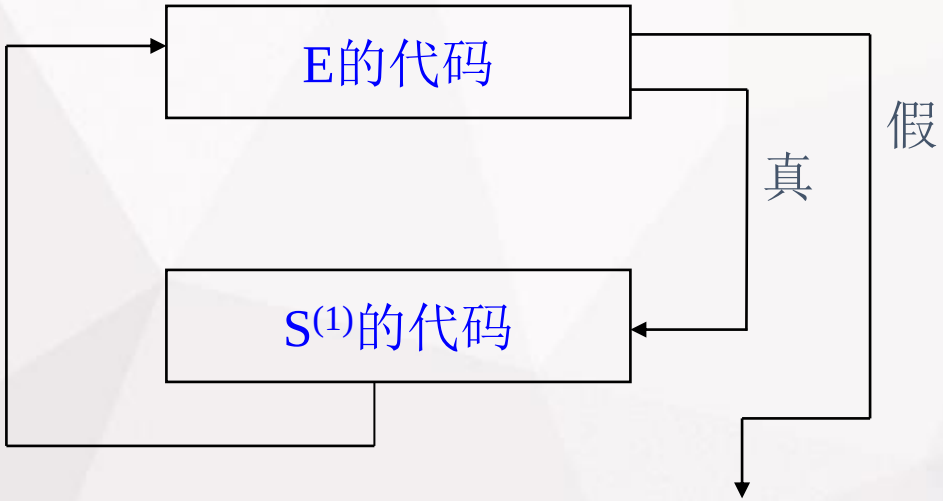
控制语句的翻译

while语句的翻译

$S \rightarrow \text{while}\ \ E\ \ \text{do}\ \ S^{(1)}$ ➡

(1) $S \rightarrow \text{while}\ \ E\ \ \text{do}\ M\ S^{(1)}$

(2) $M \rightarrow \varepsilon$



控制语句的翻译

(1) S→while E do M $S^{(1)}$

{ bp(E.tr , M.s);
bp($S^{(1)}$.chain, E.code);
S.chain=E.fa;
emit (‘goto’ E.code) }

(2) M→ε { M.s=next }

控制语句的翻译

while (A<B) do if (C<D) then x:=y+1 ;

While E do

if (C<D) then x:=y+1

{ E.tr=100=E.code;
  E.fa=101;}

100 if A<B goto —

101 goto —

While E do M

if (C<D) then x:=y+1

{ M.s=next=102 }

While E do M

if $E^{(1)}$ then x:=y+1

{ $E^{(1)}$.tr=102= $E^{(1)}$.code;
  $E^{(1)}$.fa=103;}

102 if C<D goto —

103 goto —

While $E^{(1)}$ do M

if $E^{(2)}$ then $M^{(1)}$ x:=y+1

{ $M^{(1)}$.s=next=104}

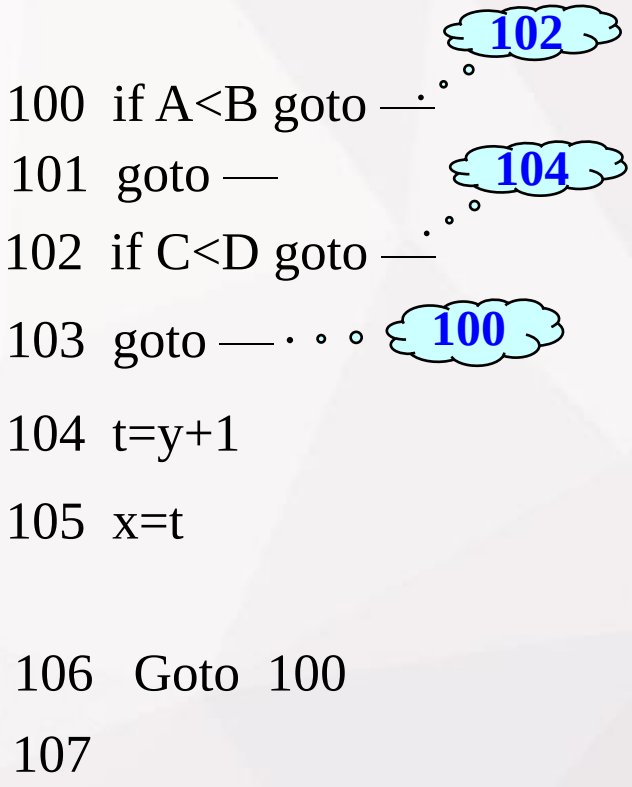
控制语句的翻译

while (A<B) do if (C<D) then x:=y+1 ;

While E do M
if $E^{(1)}$ then $M^{(1)}$ S
{ S.chain=0; }

While E do M $S^{(1)}$
{ bp($E^{(1)}$.tr, $M^{(1)}$.s);
$S^{(1)}$.chain=$E^{(1)}$.fa=103;}

S
{ bp(E.tr, M.s);
bp($S^{(1)}$.chain, E.code);
S.chain=E.fa=101;
emit('goto' E.code); }



控制语句的翻译

```
while (A<B) do
    if (C<D) then
      x:=y+1 ;
```

S.chain=101;

```
100  if A<B goto  102
101  goto —
102  if C<D goto 104
103  goto 100
104  t=y+1
105  x=t
106   Goto  100
107
```

简单说明语句的翻译

简单说明语句文法

D→ integer <namelist> | real <namelist>

<namelist>→ <namelist> , id | id

文法改写： D→D(1)， id | integer id | real id

设计语义动作如下：

（1）函数FILL(id，A)的功能是把名字id和性质A登录在符号表中。

（2）设置非终结符D的语义变量D.ATT, 记录说明语句所规定的相关名字性质

简单说明语句的翻译

（1）D→integer id

{ FILL(id, int) ; D.ATT＝int }

（2）D→real id

{ FILL(id, real) ; D.ATT=real }

（3）$D \rightarrow D^{(1)}$, id

{ FILL(id, $D^{(1)}$.ATT) ;
D.ATT=$D^{(1)}$.ATT }