编译原理

第六章 属性文法和语法制导翻译

第六章 属性文法和语法制导翻译

- 属性文法
- 基于属性文法的处理方法
- S- 属性文法的自下而上计算
- L- 属性文法和自顶向下翻译

第六章 属性文法和语法制导翻译

- 属性文法
- 基于属性文法的处理方法
- S- 属性文法的自下而上计算
- L- 属性文法和自顶向下翻译

6.2 基于属性文法的的处理方法

- 依赖图
- 树遍历
- 一遍扫描

一遍扫描的处理方法

- 一遍扫描的处理方法是在语法分析的同时计算属性值
  - 所采用的语法分析方法
  - 属性的计算次序
- L – 属性文法适合于一遍扫描的自上而下分析
- S – 属性文法适合于一遍扫描的自下而上分析

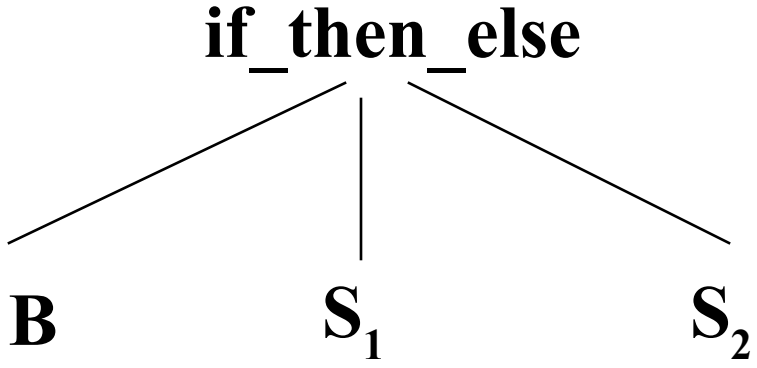
语法制导翻译法

- 所谓语法制导翻译法，直观上说就是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
- 语义规则被计算的时机
  - 在自上而下语法分析中，一个产生式匹配输入串成功时
  - 在自下而上分析中，当一个产生式被用于进行归约时

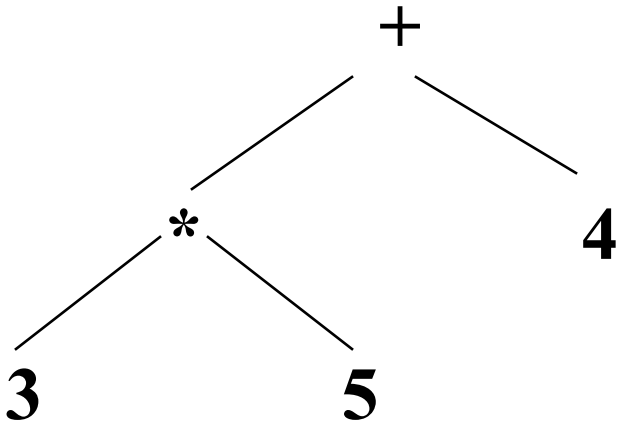
抽象语法树

- 在语法树中去掉那些对翻译不必要的信息，从而获得更有效的源程序中间表示。这种经变换后的语法树称之为抽象语法树(Abstract Syntax Tree)

- **S→if B then S_1 else S_2**



- **3*5+4**



建立表达式的抽象语法树

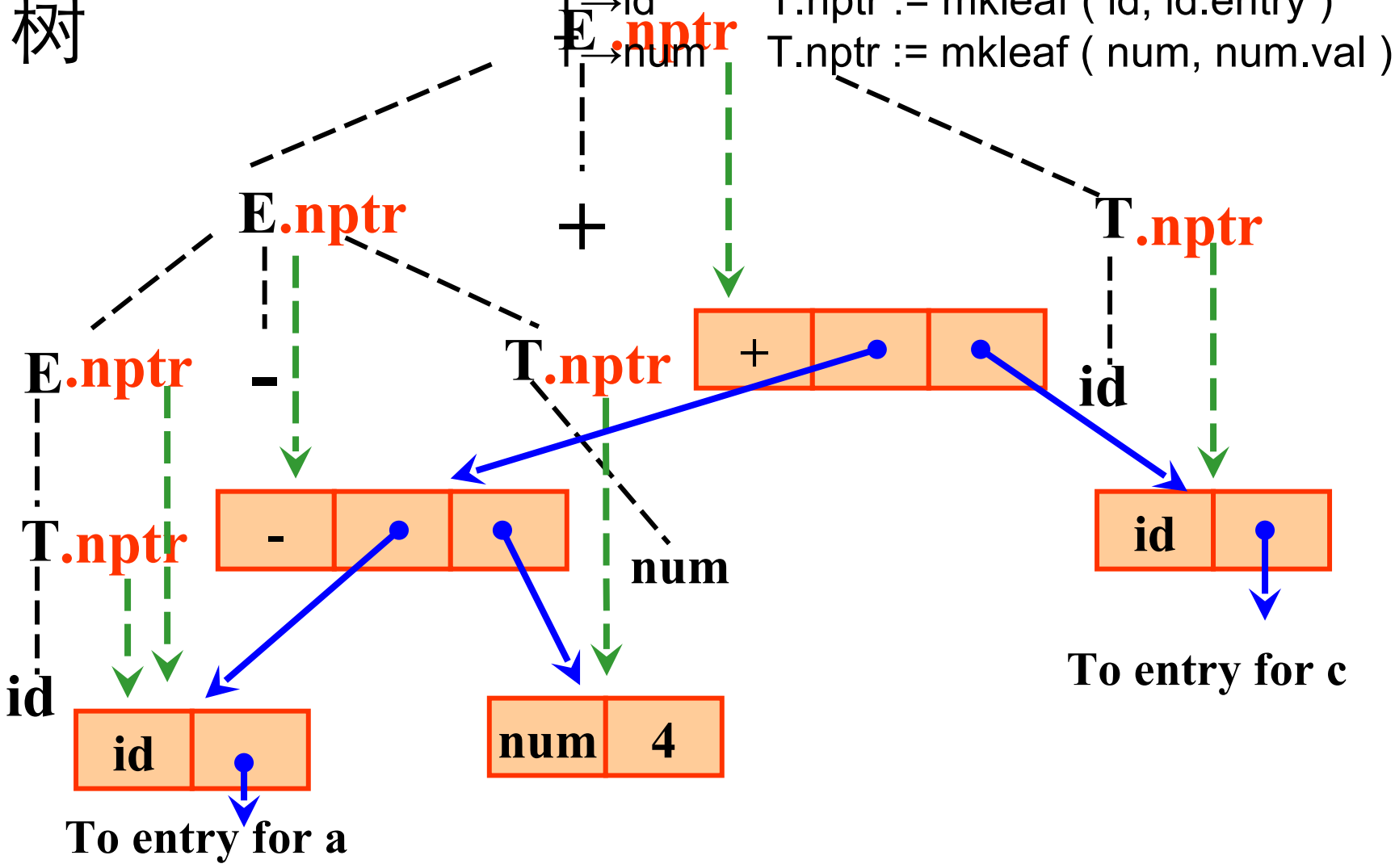
- mknode (op,left,right) 建立一个运算符号结点，标号是 op，两个域 left 和 right 分别指向左子树和右子树
- mkleaf (id,entry) 建立一个标识符结点，标号为 id，一个域 entry 指向标识符在符号表中的入口
- mkleaf (num,val) 建立一个数结点，标号为 num，一个域 val 用于存放数的值

建立抽象语法树的语义规则

产 生 式	语 义 规 则
$E \rightarrow E_1+T$	E.nptr := mknode(‘+’ , E_1.nptr, T.nptr)
$E \rightarrow E_1-T$	E.nptr := mknode(‘-’ , E_1.nptr, T.nptr)
$E \rightarrow T$	E.nptr := T.nptr
$T \rightarrow (E)$	T.nptr := E.nptr
$T \rightarrow id$	T.nptr := mkleaf (id, id.entry)
$T \rightarrow num$	T.nptr := mkleaf (num, num.val)

a – 4 + c 的抽象语法树

$E \rightarrow E_1 + T$	E.nptr := mknode(‘+’ , E_1.nptr, T.nptr)
$E \rightarrow E_1 - T$	E.nptr := mknode(‘-’ , E_1.nptr, T.nptr)
$E \rightarrow T$	E.nptr := T.nptr
$T \rightarrow id$	T.nptr := mkleaf (id, id.entry)
$T \rightarrow num$	T.nptr := mkleaf (num, num.val)



第六章 属性文法和语法制导翻译

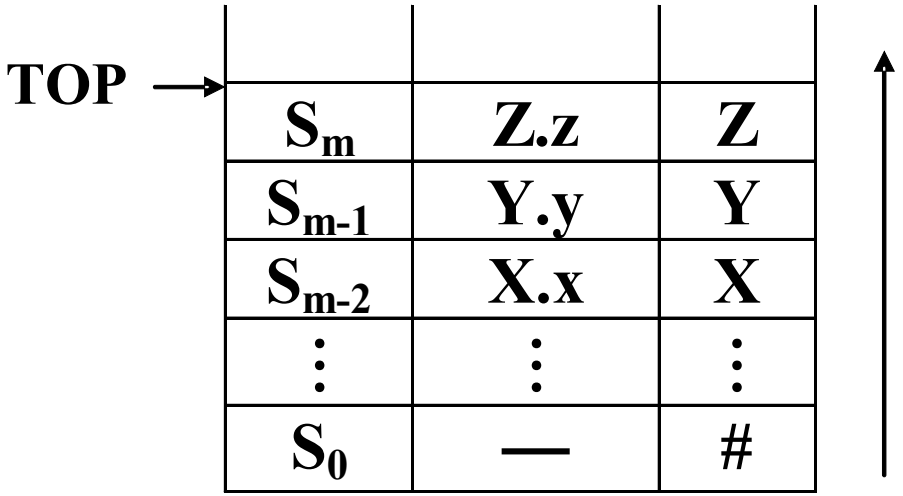
- 属性文法
- 基于属性文法的处理方法
- S- 属性文法的自下而上计算
- L- 属性文法和自顶向下翻译

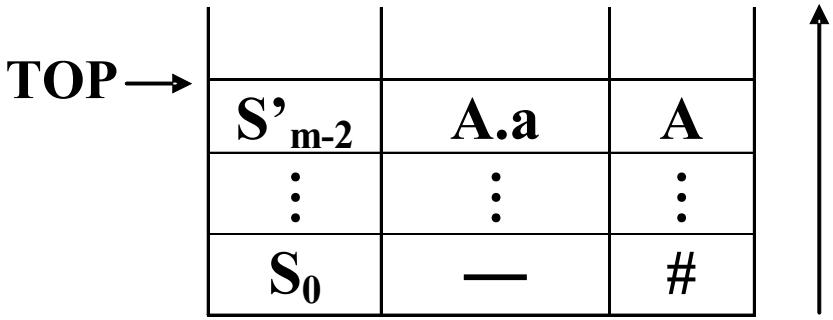
6.3 S- 属性文法的自下而上计算

- S- 属性文法：只含有综合属性
- 综合属性可以在分析输入符号串的同时由自下而上的分析器来计算
- 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- 属性文法的计算

- 在分析栈中使用一个附加的域来存放综合属性值
- 假设语义规则 A.a:=f(X.x,Y.y,Z.z) 是对应于产生式 A→XYZ 的





讨论： E 、 T 和 F 为什么没有代码段？

TOP →

S_m	Z.z	Z
S_{m-1}	Y.y	Y
S_{m-2}	X.x	X
⋮	⋮	⋮
S_0	—	#

产生式	语义规则
L→En	print(E.val)
$E \rightarrow E_1+T$	E.val := E_1.val+T.val
E→T	E.val :=T.val
$T \rightarrow T_1*F$	T.val :=T_1.val* F.val
T→F	T.val :=F.val
F→ (E)	F.val :=E.val
F→digit	F.val :=digit.lexval

产生式	代码段
L→En	print(val[top])
$E \rightarrow E_1+T$	val[ntop] := val[top-2]+val[top]
E→T	
$T \rightarrow T_1*F$	val[ntop] := val[top-2]*val[top]
T→F	
F→ (E)	val[ntop] :=val[top-1]
F→digit	

产生式	代码段
L→En	print(val[top])
$E \rightarrow E_1+T$	val[ntop] := val[top-2]+val[top]
E→T	
$T \rightarrow T_1*F$	val[ntop] := val[top-2]*val[top]
T→F	
F→ (E)	val[ntop] :=val[top-1]
F→digit	

state	sym	val	输入	用到的产生式
0	#	-	3*5+4n	
05	#3	-3	*5+4n	
03	#F	-3	*5+4n	F→digit
02	#T	-3	*5+4n	T→F
027	#T*	-3 -	5+4n	
0275	#T*5	-3 - 5	+4n	

产生式	代码段
L→En	print(val[top])
$E \rightarrow E_1+T$	val[ntop] := val[top-2]+val[top]
E→T	
$T \rightarrow T_1*F$	val[ntop] := val[top-2]*val[top]
T→F	
F→ (E)	val[ntop] :=val[top-1]
F→digit	

state	sym	val	输入	用到的产生式
0275	#T*5	-3 – 5	+4n	
02710	#T*F	-3 - 5	+4n	F→digit
02	#T	-15	+4n	T→T*F
01	#E	-15	+4n	E→T
016	#E+	-15-	4n	
0165	#E+4	-15- 4	n	

产生式	代 码 段
L→En	print(val[top])
$E \rightarrow E_1+T$	val[ntop] := val[top-2]+val[top]
E→T	
$T \rightarrow T_1*F$	val[ntop] := val[top-2]*val[top]
T→F	
F→ (E)	val[ntop] :=val[top-1]
F→digit	

state	sym	val	输入	用到的产生式
0165	#E+4	-15- 4	n	
0163	#E+F	-15- 4	n	F→digit
0169	#E+T	-15- 4	n	T→F
01	#E	-19	n	E→E+T
	#En	-19-		
	#L	-19	L→En	

小结

- 抽象语法树
- S- 属性文法
- S- 属性文法的自下而上计算
  - 一遍扫描