第五章：中间代码生成

常见的中间代码结构
语法制导方法概论

1. 生成中间代码的目的

- 便于优化
  - 让生成的目标代码效率更高
  - 优化不等同于对代码的精简和算法的优化
- 便于移植
  - 编译前端：与目标机无关
  - 编译后端：与目标机相关

2. 中间代码结构

- 2.1 逆波兰式
- 2.2 抽象语法树
- 2.3 三地址中间代码

2.1 逆波兰式（后缀表达式）

- 将运算对象写在前面，把运算符写在后面，因而也称后缀式。

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 *

●逆波兰式的特点：

①逆波兰式中的变量的次序与中缀式中的变量的次序完全一致。

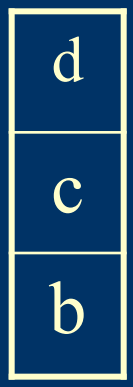
②逆波兰式中无括号。

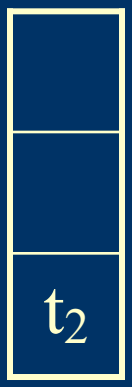
③逆波兰式中的运算符已按计算顺序排列。

- 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 bcd*+的计值过程



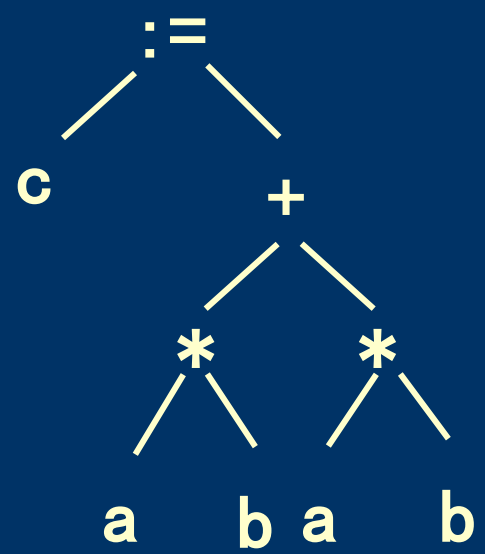


$t_1 = c*d$

$t_2 = b+t_1$

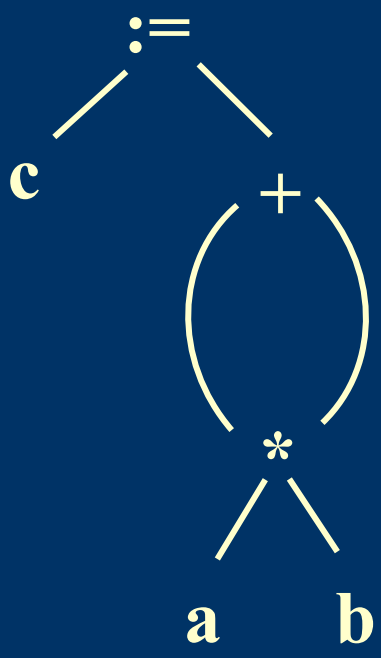
2.2 抽象语法树AGT和DAG

- 抽象语法树AGT（Abstract Grammar Tree）可以显示地表示源程序的结构，是常用的一种标准中间代码形式。
- 例如： c := a * b + a *b



- 有向无环图DAG（Directed Acyclic Graph）实际上是从抽象语法树发展而来，其主要优点是能够描述共享问题，如表达式的共享。

假设有 c := a * b + a * b.



2.3 三地址中间代码

- 所谓三地址是指操作符的两个运算分量及其该操作的运算结果的抽象地址。
- 最常见的三地址中间代码有三元式和四元式两种。

- 例如：语句 a := b * c + b / d

三元式 ：　三元式编号 (算符op，ARG_1，ARG_2)

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

四元式：(算符op，ARG_1，ARG_2，运算结果RESULT)

(1)　(* , b , c , t_1)
(2)　(/ , b , d , t_2)
(3)　(+ , t_1 , t_2 , t_3)
(4)　(:= , t_3 ,–, a)

- 三元式和四元式的比较:

- 三元式的优点是表示简单，但由于其运算结果反映在三元式的位置编号上，使得带中间代码移动或代码删除的优化工作变得复杂，所以三元式中间代码不利于优化；
- 四元式不依赖于位置，因此四元式结构的中间代码便于插入、删除和移动。

- 一般的四元式操作符应包括以下几类：

1. 算术、逻辑、关系运算符

ADDI、ADDF、SUBI、SUBF、MULTI、

MULTF、　DIVI、DIVF、MOD、AND、

OR、EQ、NE、GT、GE、LT、LE

2. I/O操作

(READI，—，　—，id)……整数输入

(READF ，—，—，id)……实数输入

(WRITE，—，— ，id)……输出id

3. 类型转换

(FLOAT, id_1, —, id_2)

…id_2 := float (id_1)

4. 赋值

(ASSIG , id_1, — id_2)

…id_2 := id_1

5. 地址加

(AADD , id_1, id_2, id_3)

…id_3 := addr (id_1) + id_2

- 6. 标号定义

(LABEL, —, —, label)….定义标号label

- 7. 条件/无条件转移

(JMP, —, — , label)…转向标号label

(JMP0, id, — , label)

…若id＝0，则转label

(JMP1, id, —, label)

…若id＝1，则转label

- 8. 过程调用

(ENTRY, Label, Size, Level)

.....子程序入口

(CALL, f, —, Result)

.....过程或函数调

- 9. 传送参数参数传递中间代码

VARACT

VALACT

FUNACT

PROACT

3. 语法制导方法

- 语法制导技术在处理和规则相关联的任务中有着重要的应用
- 语法制导就是在进行语法分析的同时要完成相应的语义动作，这些语义动作都是由一些程序组成的，要完成和用户的需求相关联的任务
- 编译器对一个串进行语法检查的同时，可以按照语法分析的程序的结构对程序进行我们所需要的操作，从而解决我们想要解决的问题。

- 语法制导方法的种类：语法制导方法依赖于具体的语法分析方法，因此可以分为自顶向下语法制导方法和自底向上语法制导方法。自顶向下语法制导方法中通常采用LL(1)分析方法为基础，而自底向上语法制导方法中通常以LR(1)分析方法为基础。
- LL(1)语法制导方法是在LL(1)文法的基础上加入语义动作符来表示相应的语义子程序，语义动作符可以加在产生式右部的任何位置。在进行LL(1)语法分析的过程中每遇到语义动作符，则调用相应的语义子程序来完成相应的语义处理。
- LR(1)语法制导方法是在LR(1)文法的基础上加入语义动作符来表示相应的语义子程序，每条产生式最多配有一个语义规则，即语义动作符只能加在产生式的最右部。在进行LR(1)语法分析的过程中，每当按照某个产生式的右部进行规约时，就调用该产生式右边的相应的语义子程序来完成相应的语义处理。

LL(1)语法制导方法的实例

- 设有如下LL(1)文法：

G:

S → A B

A → a A | b

B → b B | c

要求：应用语法制导翻译方法完成如下任务：对输入文法 G 的任意句子 L ，输出 L 中 b 串的长度。如串 abbbbc 是 G 的句子，则该处理器将输出4。

G的原始LL(1)分析表

	a	b	c	#
S	S → A B	S → A B		
A	A → a A	A → b		
B		B → b B	B → c	

G:

S → AB

A → aA | b

B → bB | c

- G 的动作文法如下：

G: S→ *#Init#* AB

A → aA | b *#Add#*

B → b *#Add#* B | c *#Out_Val#*

各动作符对应的语义子程序如下：

Init **m = 0**

Out_Val **printf (“%d”, m);**

Add **m++ ;**

G[S]:

S→ *#init#* A B

A → a A | b *#Add#*

B → b *#Add#* B | c*#Out_Val#*

G的带动作符的LL(1)分析表

	a	b	c	#
S	S → *#init#* A B	S → *#init#* A B		
A	A → a A	A → b *#Add#*		
B		B → b *#Add#* B	B → c*#Out_Val#*	

语法制导的实现过程（1）

例 G[s]：

[1] S→ *#init#* A B {a}　[2] A → a A {a}

[3] A → b *#Add#* {b}　[4] B → b*#Add#* B {b}

[5] B → c*#Out_Val#* {c}

对给定的终极符串abbbc，分析过程：

S#	abbbc #	Derivation
#init# A B #	abbbc #	# Init #
A B #	abbbc #	Derivation
aA B #	abbbc #	Match
A B #	bbbc #	Derivation
b ***#Add#***B #	bbbc #	Match
#Add# B #	bbc #	***#Add#***
B #	bbc #	

语法制导的实现过程（2）

例 G[s]：

[1] S→ *#init#* A B {a}　[2] A → a A {a}

[3] A → b*#Add#* {b}　[4] B → b*#Add#*B {b}

[5] B → c*#Out_Val#* {c}

对给定的终极符串abbbc，分析过程(续)：

B #	bbc #	Derivation
b *#Add#* B #	bbc #	Match
#Add# B #	bc #	*#Add#*
B #	bc #	Derivation
b*#Add#* B #	bc #	Match
*#Add#*B #	c #	*#Add#*
B #	c #	Derivation
c # Out_Val # #	c #	Match
# Out_Val # #	#	*#Out_Val#*
#	#	Success

LR(1)语法制导方法的实例

- 设有如下文法：

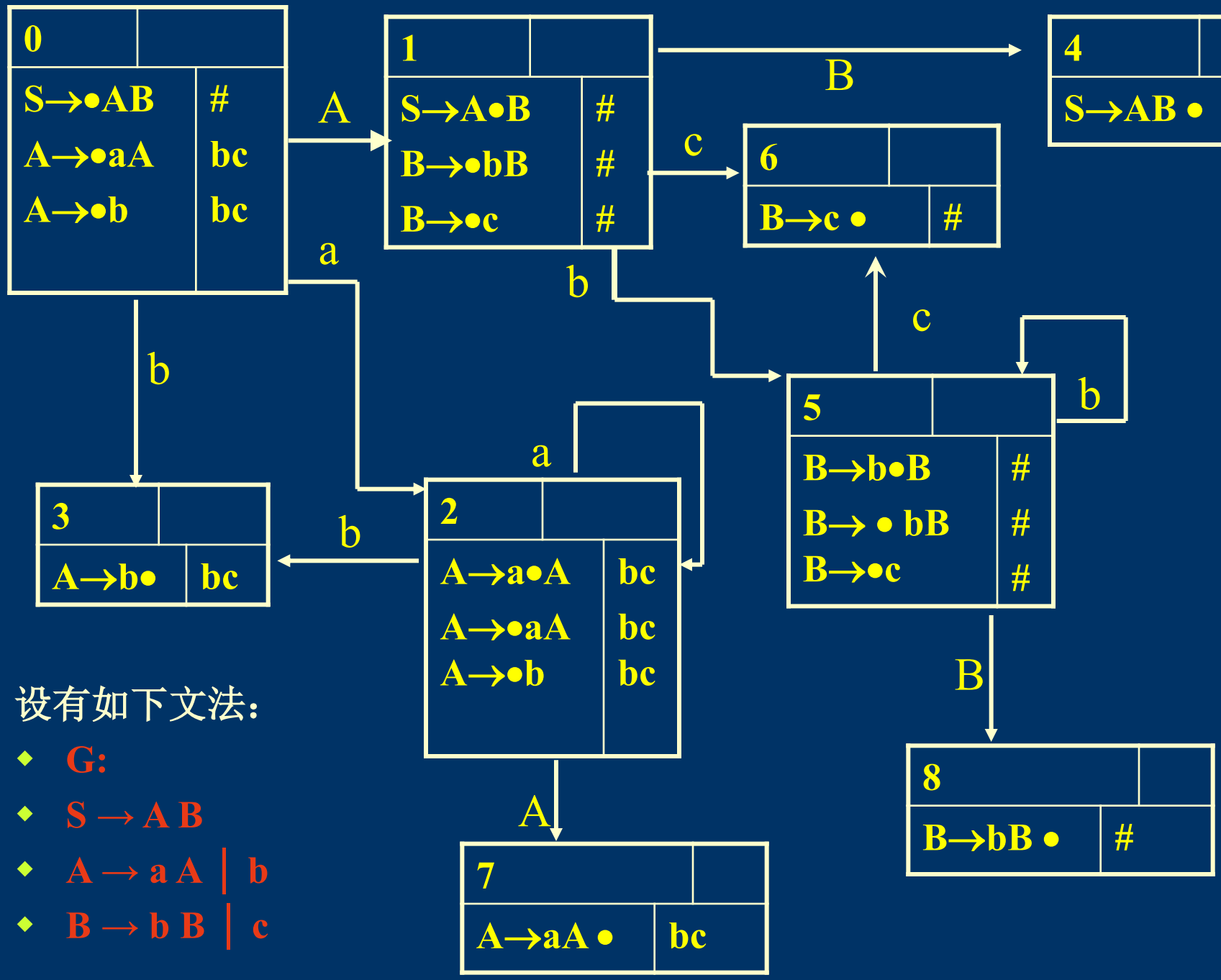
G:

S → A B

A → a A | b

B → b B | c

要求：应用LR(1)语法制导翻译方法完成如下任务：对输入文法 G 的任意句子 L ，输出 L 中 b 串的长度。如串 abbbbc 是 G 的句子，则该处理器将输出4。


0
S→•AB #
A→•aA bc
A→•b bc
1
S→A•B #
B→•bB #
B→•c #
4
S→AB • #
6
B→c • #
3
A→b• bc
2
A→a•A bc
A→•aA bc
A→•b bc
5
B→b•B #
B→ • bB #
B→•c #
8
B→bB • #
7
A→aA • bc
A
B
a
b
c
LR(1)
状
态
机
设有如下文法：
G:
S → A B
A → a A | b
B → b B | c

LR(1)分析表

	a	b	c	#	S	A	B
0	S_2	S_3				1	
1		S_5	S_6				4
2	S_2	S_3				7	
3		R_3	R_3				
4				Acc			
5		S_5	S_6				8
6				R_5			
7		R_2	R_2				
8				R_4			

动作文法如下：

G:

```
S → AB          //  #out_Val#
A → aA
    | b           //  #Add#
B → bB          //  #Add#
    | c
```

LR(1)分析过程

状态栈	符号栈	输入串	Action	GoTo
0	#	abbbc#	S2	
0,2	# a	bbbc#	S3	
0,2,3	#ab	bbc#	R3	7
0,2,7	#aA	bbc#	R2	1
0,1	#A	bbc#	S5	
0,1,5	#Ab	bc#	S5	
0,1,5,5	#Abb	c#	S6	
0,1,5,5,6	#Abbc	#	R5	8
0,1,5,5,8	#AbbB	#	R4	8
0,1,5,8	#AbB	#	R4	4
0,1,4	#AB	#	Acc	

m=0

m=1

m=2

m=3

分析表

G:

S → A B[1] // # Out_Val #

A → a A [2]

| b [3] // # Add #

B → b B [4] // # Add #

| c[5]

4. 中间代码生成中的几个问题

- 如果符号表一直保存到目标代码生成阶段，则在四元式中标识符的地址可用其在符号表中的地址表示。
- 如果符号表不保存到目标代码生成阶段，则需要把目标代码生成阶段对名字进行地址分配所需要的相关信息放到中间代码中，这些信息包括标识符抽象地址（层数、偏移）以及直接／间接访问等信息，具体地说这些信息应放在四元式中的操作分量或运算结果的地址表示中。

- 四元式中操作分量或运算结果的地址表示可以分为三大类：标号类、数值类和地址类。
  - 标号类的语义信息是相应的标号值，包括过程 ／ 函数体的入口标号；
  - 数值类的语义信息就是该数据值；
  - 地址类的语义信息由三个部分组成：层数、偏移和访问方式(分为直接访问方式和间接访问方式)。变参变量（指针变量）以及代表复杂变量地址的临时变量属于间接访问方式。

- 关于临时变量的地址表示：
  1. 临时变量没有层数概念，因此对临时变量的层数可以取任意一个负数，如取-1；
  2. 在中间代码生成阶段尽管产生大量的临时变量，但经过优化后，只有少部分临时变量能保留下来，这样在中间代码生成阶段还不能确定临时变量的Offset值，因此临时变量的地址表示中的Offset暂时取为临时变量的编号。

语义信息的提取与保存：

四元式：（算符op，ARG_1，ARG_2，运算结果RESULT）

ARG1，ARG2 ，RESULT为操作分量和运算结果的抽象地址表示，应包含相应语义信息。

如：3.5+i

(ADDF,3.5,(i.level,i.off,dir),(-1,t3,dir))

语义信息的两种表示方式：

- 指向相应符号表的指针
- 把对应分量的语义信息放在此处

语义栈Sem及其操作：在语法制导生成中间代码的过程中，要用到一个语义栈， 该栈主要用于存放运算分量和运算结果的语义信息。

- 进栈、退栈、栈的结构可自行定义

- 存放中间代码子程序GenCode

  该子程序功能是将一条四元式中间代码存放到中间代码区中，其格式如下：

  GenCode（ω , left , right , result ）

  - 该子程序主要用于表达式中间代码生成，其功能是产生一条四元式中间代码。
  - 当执行语义动作子程序GenCode时，四元式操作码的左、右运算分量的语义信息已经在语义栈Sem的次栈顶和栈顶，因此，该子程序的参数只给出操作码ω即可。

GenCode(ω)的主要工作

(1)若Sem[top–1].typ ≠ Sem[top].typ，

且Sem [top–1].typ = integer，则ω运算结果类型确定为real型，并且为左分量产生类型转换代码；

(FLOAT, **[top–1]**.FORM,-, tempArg)

(2) 若Sem[top–1].typ ≠ Sem[top].typ，

且Sem[top].typ = integer，则ω结果运算类型确定为real型，并且为右分量产生类型转换代码；

(FLOAT, **[top]**.FORM,-, tempArg)

(3) 若Sem[top–1].typ = Sem[top].typ，则ω运算结果类型确定为Sem[top-1].typ，不产生类型转换代码；

GenCode(ω)的主要工作(续)

(4) 产生（类型转换后）的中间代码:

（ω^* , **Sem[top–1].FORM** /tempArg, **Sem[top].FORM**/tempArg , **t** ）；

(5) 从语义栈Sem中删除左、右运算分量Sem[top–1] 和 Sem[top] ， 并把运算结果的语义信息压入Sem栈。