第五章：中间代码生成(1)

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 三元式
- 2.2 逆波兰式
- 2.3 抽象语法树
- 2.4 四元式

2.1 三元式

- 由三个部分组成:

  操作符，运算分量，运算分量

如：x+y 可以表示成（+, x, y）

例如语句:

a:=b*c+b/d

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

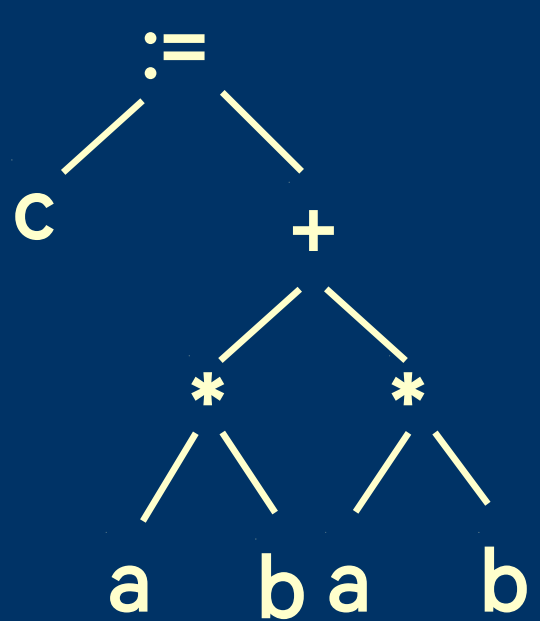
2.2 逆波兰式（后缀表达式）

- 将运算对象写在前面，把运算符写在后面，因而也称后缀式。最大特点是排好了计算顺序，而不是根据运算符的优先级。

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 *

2.3 抽象语法树

- 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



2.4 四元式

四元式: (算符op，ARG_1，ARG_2，运算结果RESULT)

例如： 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)

- 四元式的特点:

都是一些简单的运算，每一条四元式都是一个简单运算，更接近于汇编程序和机器指令，生成目标代码的时候更方便，

顺序是排好的，按照四元式的顺序来进行计算就可以，而不是按照运算符的优先级。

2.4 四元式

- 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
- READI,READF,FLOAT,ASSIG,AADD...

- 遇到时随时解释

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。

3. 语法制导方法

- 例如：有文法G

$S \rightarrow AB$

$A \rightarrow aA \mid b$

$B \rightarrow bB \mid c$

要求对输入G的任意句子L，输出b串的长度，例如abbbbc，输出4

3. 语法制导方法

则有: S→ #Init# AB

A→ aA | b #Add#

B→ b #Add# B | c #Out#

其中:

Init: m = 0;

Add : m++ ;

Out : print(m);

基于LL(1)的语法制导的实现过程（1）

例 G[s]：[1] S → #init# A B　　[2] A → aA　　{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：

符号栈	输入流	动作
S #	abbbc #	替换
#init# A B #	abbbc #	#Init# → m=0
A B #	abbbc #	替换
aA B #	abbbc #	Match
A B #	bbbc #	替换
b #Add#B #	bbbc #	Match
#Add# B #	bbc #	#Add# → m++
B #	bbc #	替换

基于LL(1)的语法制导的实现过程（2）

例 G[s]：[1] S → *#init#* A B {a b}　　[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)：

符号栈	输入流	动作
B #	bbc #	替换
b #Add# B #	bbc #	Match
#Add# B #	bc #	#Add#→
B #	bc #	替换
b#Add# B #	bc #	Match
#Add#B #	c #	#Add#→
B #	c #	替换
c #Out# #	c #	Match
#Out# #	#	#Out# print(m)
#	#	Success

4. 中间代码生成中的几个问题

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

4. 中间代码生成中的几个问题

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，把相关的分析程序的一些中间结果都要存放到语义栈中。

- 进栈、退栈、栈的结构可自行定义

5. 表达式的中间代码生成

算数表达式文法：

【1】 E→T
【2】 E→E+T
【3】 E→E-T
【4】 T→F
【5】 T→T*F
【6】 T→T/F
【7】 F→(E)
【8】 F→i

5.1 基于LR(1)方法

【1】E→T 空

【2】E→E+T #inc 【3】E→E-T #dec

#inc中要执行的动作:

1. 对sem(s-1)和sem(s-2)进行类型检查（是否相同、是否相容、是否需要类型转换_必要的话产生转换四元式）
2. new(t);
3. Gen(+,sem(s-2),sem(s-1),t)//dec为'-'号
4. 退栈： s=s-1; 或者是 pop(2)
5. 进栈: sem(s-1)=t; 或者是push(t)

5.1 基于LR(1)方法

【4】T→F 空

【5】T→T*F #MUL 【6】T→T/F # Dev

#MUL中要执行的动作:

1. 对sem(s-1)和sem(s-2)进行类型检查（是否相同、是否相容、是否需要类型转换_必要的话产生转换四元式）
2. new(t);
3. Gen(*,sem(s-2),sem(s-1),t)//Dev为'/'号
4. 退栈： s=s-1;
5. 进栈: sem(s-1)=t

5.1 基于LR(1)方法

【7】F→(E) 空

【8】F→i#Push

#Push中要执行的动作:

sem(s)=i; s=s+1;

试给出a+b/c-d*f的分析过程