第七章：语义分析和中间代码生成

2008年秋

7.1中间表示的概念

- 抽象语法树vs类似目标代码
- 是否使用目标机和运行时环境的详细信息
  - 数据类型的尺寸；
  - 寄存器
  - 变量的存储位置
- 是否使用符号表中的全部信息
  - 作用域
  - 嵌套层次
  - 变量的偏移量

- 其它用处：
  - 用于代码分析以产生高效目标代码
  - 用于多目标编译（retargetable）

中间代码生成

- 中间代码生成位于词法分析和语法分析之后，是代码生成中的一个阶段；
- 中间代码的形式很多，如逆波兰记号、抽象语法树、三地址码（三元式、四元式）、P-代码， 等等
- 属性文法是用于中间代码生成的常用的方法。

7.1.1 后缀式－逆波兰表示

- 用于表示表达式时：
  - 变量或常量的后缀式是自身；
  - $e_1 op\ e_2$ 的后缀式是 $e_1' e_2' op$，其中e_1'，e_2' 分别是e_1和e_2的后缀式；
  - (e)的后缀式就是e的后缀式；

A+B	AB+
A+B*C	ABC*+
x/y^z-d*e	xyz^/de*-
(a=0∧b>3)∨(e ∧x≠y)	a0=b3>∧exy≠∧∨

- 运算对象出现的顺序相同；
- 运算符是按实际计算次序从左到右排列。

本章内容：

1. 中间表示形式
2. 说明语句的翻译
3. 简单算术表达式和赋值句到四元式的翻译
4. 布尔表达式到四元式的翻译
5. 控制语句的翻译
6. 数组元素的引用
7. 过程调用

对后缀式求值

使用一个栈来求值。过程：

从左到右依次扫描后缀式表示中的各个符号，每遇到一个运算对象，就压入栈中；每遇到一个运算符时，就弹出栈顶k个运算对象进行运算，并将结果压入栈顶。结束时，栈顶为整个表达式的值

x/y^z-d*e　　xyz^/de*-

后缀式的推广

- *op*为k目运算符，则*op*作用于$e_1e_2…e_k$的结果用$e_1'e_2'…e_k'op$来表示。

若用？表示if-then-else，

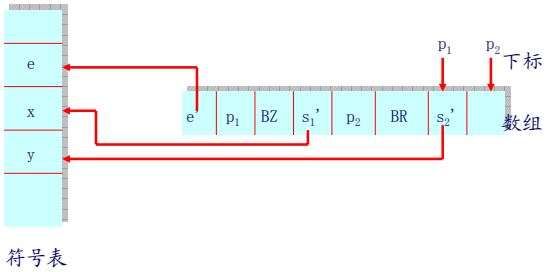
If a then if c-d then a+c else a*c else a+b

a?c-d?a+c:a*c:a+b

表示为：

a cd- ac+ ac*? ab+?

在向量POST存放的后缀式



后缀式表示中注意的问题

- **后缀式：　exy?**
- **含义：　e不等于0，取x，否则取y.**

- **这种表示法要求在任何情况下都要把x,y都计算出来，但只会用到其中一个值。**
- **语义上的差异：如果运算对象无定义或者有副作用，则后缀表示法不仅无效，而且可能是错误的；还有代码效率问题。**

属性文法表示

- 属性addr：纪录运算对象的起始位置
- 向量POST和指针p

- E1→E2+T {E1.addr=E2.addr; POST[p]=‘+’; p++;}
- E→T　　{E.addr=T.addr}
- T1→T2*F {T1.addr=T2.addr; POST[p]=‘*’; p++;}
- T→F　　{T.addr=F.addr;}
- F→(E)　　{F.addr=E.addr;}
- F→id　　{F.addr=p; POST[p]=lookup(id); p++;}

后缀式表示实现

- **后缀式存放在一个向量POST[1..n]中，并有如下转移运算:**
- **p BR　无条件转至POST[p]**
  - **e’ p BZ e’是某表达式e的后缀表示，当e’值为0转POST[p]**
  - **e1’ e2’ p BL　当e1’<e2’时转向POST[p]**
- **类似地还可以定义BN(非0转)、BP(正号转)、BM(负号转)**

IF e THEN s_1 ELSE s_2

e' p_1 BZ s_1' p_2 BR s_2'

其中e', s_1', s_2'是e, s_1, s_2的后缀表示, p_1和p_2分别表示s_2'在POST中的开始位置以及后邻的位置。

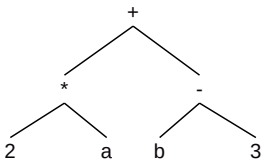
e' p_1 BZ s_1' p_2 BR p_1:s_2' p_2:

步骤	分析栈	R	输入串	动作	后缀表示
0	#	a	+b*c#	push	
1	#a	+	b*c#	归约F->id;Sub6	a
2	#F	+	b*c#	归约T->F;Sub4	a
3	#T	+	b*c#	归约E->T;Sub2	a
4	#E	+	*c#	push	a
5	#E+	b	c#	push	a
6	#E+b	*	c#	归约F->id;Sub6	ab
7	#E+F	*	c#	归约T->F;Sub4	ab
8	# E+T	*	c#	push	ab
9	# E+T*	c	#	push	ab
10	# E+T*c	#		归约F->id;Sub6	abc
11	# E+T*F	#		归约T->T*F;Sub3	abc*
12	#E+T	#		归约E->E+T;Sub1	abc*+
13	#E	#			abc*+

实现三地址码的数据结构

文法规则	语义规则
S->id:=E	S.tree=*mkNode*(:=, *mkLeaf*(id, id.place), E.tree)
E->E_1+E_2	E.tree=*mkNode*(+, E_2.tree, E_2.tree)
E->E_1*E_2	E.tree=*mkNode*(*, E_1.tree, E_2.tree)
E->-E_1	T.tree=*mkNode*(uminus, E_1.tree, nil)
E->(E_1)	E.tree=E_1.tree
E->id	F.tree=*mkLeaf*(id, id.place)

例：2*a+(b-3)




```
t1 = 2 * a
t2 = b - 3
t3 = t1 + t2
```

```
t2 = b - 3
t1 = 2 * a
t3 = t2 + t1
```

- 产生临时变量；
- 临时变量跟抽象语法树的内结点对应；
- 语句的次序；
- 操作符扩充；

7.1.3 三地址码

- `x = y` ***op*** `z`

- 以操作符为核心
  - 一个操作符***op***
  - 两个操作数y**和**z
  - 一个结果x
- x,y和z通常是内存地址
- y和z还可以是常数和常量

一个程序段的三地址代码表示：

```
read x;
if 0 < x then
  fact := 1;
  repeat
    fact := fact * x;
    x := x - 1
  until x = 0;
  write fact
end
```

```
read x;
t1 = x > 0
if_false t1 goto L1
fact = 1
label L2
t2 = fact * x
fact = t2
t3 = x - 1
x = t3
t4 = x == 0
if_false t4 goto L2
write fact
label L1
halt
```

一个程序段的三地址代码表示：

- 要产生临时变量；
- 临时变量跟抽象语法树的内结点对应；
- 语句之间可以有不同的次序；
- 操作符可有扩充；比如单目运算；

实现三地址码的数据结构

- ***四元组***
  - **(*op*, *arg1*, *arg2*, *result*)**
  - ***op*是一个二元(也可一元或零元)运算符；**
  - ***arg1*, *arg2*分别为两个运算对象，可以是变量、常数、临时变量等等(可以缺省)；**
  - **运算结果在*result*中；**
  - **跟符号表有关系。**

- 2*a+(b-3)
  - t1 = 2 * a
  - t2 = b - 3
  - t3 = t1 + t2

- (*, 2, a, T1)
- (- , b, 3, T2)
- (+, T1, T2, T3)

四元组举例：

```
read x;
t1 = x > 0
if_false t1 goto L1
fact = 1
label L2
t2 = fact * x
fact = t2
t3 = x - 1
x = t3
t4 = x == 0
if_false t4 goto L2
write fact
label L1
halt
```

```
(rd,x,_,_)
(gt,x,0,t1)
(if_f,t1,L1,_)
(asn,1,fact,_)
(lab,L2,_,_)
(mul,fact,x,t2)
(asn,t2,fact,_)
(sub,x,1,t3)
(asn,t3,x,_)
(eq,x,0,t4)
(if_f,t4,L2,_)
(wri,fact,_,_)
(lab,L1,_,_)
(halt,_,_,_)
```

产生三地址码的属性文法

- 例：
- **E->**id=**E|A**
- **A->A+F|F**
- **F->**(**E**)|num|id

- 属性name：记录临时变量名
- 属性code：记录已生成的代码

实现三地址码的数据结构2

- **三元组**
  - **(*op*, *arg1*, *arg2*)**
  - ***op*是一个二元(也可一元或零元)运算符；**
  - ***arg1*, *arg2*分别为两个运算对象：可以是变量、常数、临时变量等等(可以缺省)；也可以是某个三元式的序号；**
  - **跟符号表有关系。**

- 2*a+(b-3)
  - t1 = 2 * a
  - t2 = b - 3
  - t3 = t1 + t2

(1) (*, 2, a)
(2) (－, b, 3)
(3) (+, (1), (2))

用符号串表示代码1

语法规则	语义规则
E_1->id=E_2	E_1.name=id.strval E_1.code=E_2.code++id.strval||”=”||E_2.nam
E->A	E.name=A.name E.code=A.code
A_1->A_2+F	A_1.name=newtemp() A_1.code=A_2.code++F.code++ A_1.name||”=”||A_2.name||”+”||F.name
A->F	A.name=F.name A.code=F.code
F->(E)	F.name=E.name F.code=E.code
F->num	F.name=num.strval F.code=“”
F->id	F.name=id.strval F.code=“”

++串连接，包括1个换行；

||串连接，包括1个空格

三元组举例：

```
(0)  (rd,x,_)
(1)  (gt,x,0)
(2)  (if_f,(1),11)
(3)  (asn,1,fact)
(4)  (mul,fact,x)
(5)  (asn,(4),fact)
(6)  (sub,x,1)
(7)  (asn,(6),x)
(8)  (eq,x,0)
(9)  (if_f,(8),(4))
(10) (wri,fact,_)
(11) (halt,_,_)
```

```
(rd,x,_,_)
(gt,x,0,t1)
(if_f,t1,L1,_)
(asn,1,fact,_)
(lab,L2,_,_)
(mul,fact,x,t2)
(asn,t2,fact,_)
(sub,x,1,t3)
(asn,t3,x,_)
(eq,x,0,t4)
(if_f,t4,L2,_)
(wri,fact,_,_)
(lab,L1,_,_)
(halt,_,_,_)
```

用符号串表示代码2

语法规则	语义规则
S->id:=E	S.code=E.code||gen(id.place ‘:=‘ E.place)
E->-E_1	E.place=newtemp() E.code=E_1.code||gen(E.place ‘:=‘ ‘minus’ E_1.place)
E->E_1+E_2	E.place=newtemp() E.code=E_1.code||E_2.code||gen(E.place ‘:=‘ E_1.place ‘+’ E_2.place)
E->(E_1)	E.place=E_1.place E.code=E_1.code
E->id	E.place=id.place E.code=“”

++串连接，包括1个换行；

||串连接，包括1个空格

7.2 说明语句的翻译

例6.3 变量声明

D->TL

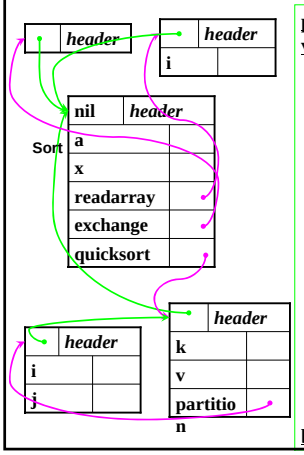
T->int | float

L->id, L | id

文法规则	语义规则
D->TL	L.dtype=T.dtype
T->int	T.dtype=integer
T->float	T.dtype=real
L_1->id, L_2	id.dtype=L_1.dtype
	L_2.dtype=L_1.dtype
L->id	id.dtype=L.dtype

题目：求一个3*3矩阵对角线元素之和

```
main()
{
float a[3,3],sum;
int i,j;
sum=0;
printf("please input
rectangle elements:\n");
for(i=0;i<3;i++)
  for(j=0;j<3;j++)
  scanf("%f",&a[i,j]);
for(i=0;i<3;i++)
  sum=sum+a[i,i];
printf("Sum of the diagonal
elements is %6.2f\n",sum);
}
```




```
program sort(input,output)
var a: array[0..10] of int;
  x:int;
  proc readarray;
  var i:int;
    begin…a….end;
  proc exchange(i,j:int);
    begin
    x:=a[i];a[i]:=a[j];a[j]=x
    end;
  proc quicksort(m,n:int);
    var k,v:int;
    func partition (y,z:int):int;
    var i,j:int;
      begin …a…v…exchange(i,j);
      d;
  begin…end;
begin…end.
```

过程中的说明语句

P->D

D->D;D|TL

T->int | float

L->id, L | id

文法规则	语义规则
P->D	offset=0
D_1->D_2;D_3	
D->TL	L.type=T.type; L.width=T.width
T->int	T.type=integer;T.width=4
T->float	T.type=real;T.width=8
L_1->id, L_2	lookup(id.name, L_1.type, offset); L_2.type=L_1.type; L_2.width=L_1.width; offset=offset+L_1.width
L->id	lookup(id.name, L.type, offset); offset=offset+L.width

题目：求一个3*3矩阵对角线元素之和

```
main()
{
float a[3,3],sum;
int i,j;
sum=0;
printf("please input
```

属性及过程的定义

- mktable(tab) 创建一个符号表，父表指针为tab；
- Enter(tab,name,type,offset)在tab所指符号表中为名字name建立一个登记项，类型type和相对地址offset填入该项中；
- addwidth(tab,width) 在tab所指符号表表头中记录下该表中所有名字占用的总宽度；
- enterproc(tab, name, newtab) 在tab所指符号表中为名字name的过程建立一个登记项，参数newtab指向name的符号表。

作用域

P→D

D→D;D|TL|

proc id;D;S|

func id;D;S

T→int | float

L→id, L | id

```
program sort(input,output)
var a: array[0..10] of int;
  x:int;
  proc readarray;
  var i:int;
    begin…a….end;
  proc exchange(i,j:int);
    begin
    x:=a[i];a[i]:=a[j];a[j]=x
    end;
  proc quicksort(m,n:int);
    var k,v:int;
    func partition (y,z:int):int;
    var i,j:int;
      begin …a…v…exchange(i,j);…end;
  begin…end;
begin…end.
```

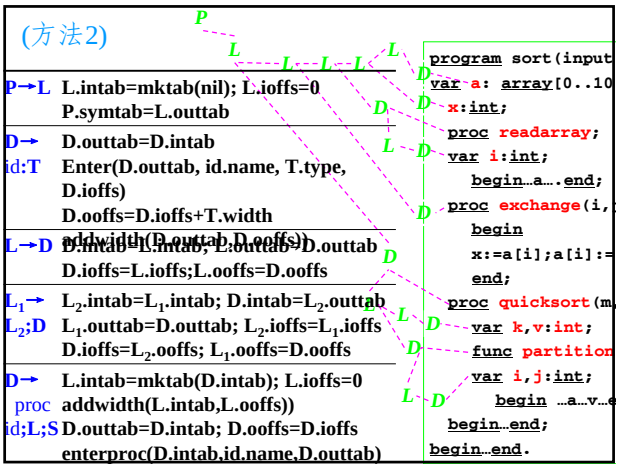
处理嵌套过程中的说明语句(方法1)

P→MD	addwidth(pop(s_tab), pop(s_offset))
M→ε	t:=mktab(nil); push(t, s_tab);
D→D;D	push(0,s_offset)
D→ proc id;ND_1;S	t:= pop(s_tab) addwidth(t,pop(s_offset)) enterproc(top(s_tab), id.name, t)
D→id:T	Enter(top(s_tab), id.name, T.type, top(s_offset))
N→ε	t:=mktab(top(s_tab)) push(t, s_tab) push(0,s_offset)

pop(s_)弹出s_栈顶元素；

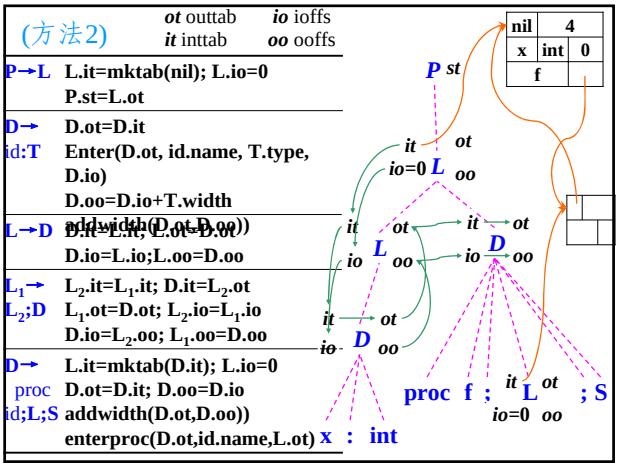
top(s_)返回s_栈顶元素；

push(e,s_)压入e到栈s_.



7.3 赋值语句的翻译

- 翻译赋值语句和算术表达式的属性文法
- 类型转换
- 数组处理



语法规则	语义规则
E_1->id=E_2	E_1.name=id.strval E_1.code=E_2.code++id.strval‖”=”‖E_2.nam
E->A	E.name=A.name E.code=A.code
A_1->A_2+F	A_1.name=newtemp() A_1.code=A_2.code++F.code++ A_1.name‖”=”‖A_2.name‖”+”‖F.name
A->F	A.name=F.name A.code=F.code
F->(E)	F.name=E.name F.code=E.code
F->num	F.name=num.strval F.code=“”
F->id	F.name=id.strval F.code=“”

前面介绍过的一个例子

说明语句的翻译的小结

- 简单变量的
  - 类型
  - 偏移量（以过程为单位）
  - 创建符号表
  - 符号表中添加登记项
- 结构型变量（在后面介绍）
  - 数组
  - 结构
  - 指针等

7.3.1 属性及过程的定义（产生四元组）

- **NewTemp()** 函数过程。每次调用时，它都回送一个代表新临时变量名的整数码作为函数值(跟符号表有关)；
- **Lookup(name)** 以name为名字查符号表：查到返回入口指针，否则填入并返回入口指针；
- **E.place** 表示存放E值的变量名在符号表的入口或者整数码（若为临时变量）；
- **Gen(op, arg1, arg2, result)** 语义过程，建立四元式 (op, arg1,arg2, result)并填入四元式表中。

例：翻译简单赋值语句为四元组的属性文法

- A→V:=E {if E.place=err or V.place=err then A.place=err else {A.place =ok; Gen(:=,E.place,nil,V.place)}}
- $E \rightarrow E_1+E_2$ {if E_1.place=err or E_2.place=err then E.place=err else {E.place=NewTemp(); Gen(+,E_1.place,E_2.place,E.place)}}
- $E \rightarrow E_1*E_2$ {if E_1.place=err or E_2.place=err then E.place=err else {E.place=NewTemp(); Gen(*,E_1.place,E_2.place,E.place)}}
- $E \rightarrow (E_1)$ {E.place=E_1.place}
- E→V {E.place=V.place}
- E→num {E.place=lexval(num)}
- V→id {V.place=(p:=Lookup(id.name))? p : err}

7.3.3 数组元素引用

- 数组说明与下标变量的语义
- 地址计算公式
- 四元式中数组元素表达形式
- 赋值语句中数组元素的翻译

去掉错误处理部分

- A→V:=E {Gen(:=,E.place,nil,V.place)}
- $E \rightarrow E_1+E_2$ {E.place=NewTemp(); Gen(+,E_1.place,E_2.place,E.place)}
- $E \rightarrow E_1*E_2$ {E.place=NewTemp(); Gen(*,E_1.place,E_2.place,E.place)}
- $E \rightarrow (E_1)$ {E.place=E_1.place}
- E→V {E.place=V.place}
- E→num {E.place=lexval(num)}
- V→id {V.place=Lookup(id.name}

数组说明与下标变量的语义

- **数组说明float a[SIZE] ;　　int i,j;**
- **数组元素引用a[i+1]=a[j*2]+3;**

- **数组说明var a: array[0..10] of int;　i,j:int;**
- **数组元素引用a[i]:=a[j]+1;**

- **数组说明real x(-13,13)**
- **integer i,j**
- **数组元素引用x[i]=x[j]+1**

- 数组元素的存放区域如何安排？
- 如何确定数组元素的偏移量？

7.3.2 类型转换

- **A→V:=E {Gen(:=,E.place,nil,V.place);}**
- **$E \rightarrow E_1+E_2$** {E.place=NewTemp();

if E_1.type=int then if E_2.type=int then
Gen(addi,E_1.place,E_2.place,E.place) else {tmp=NewTemp();
Gen(itor, E_1.place, nil, tmp); Gen(addr,tmp,E_2.place,E.place);}
else if E_2.type=int then {tmp=NewTemp(); Gen(itor, E_2.place, nil, tmp); Gen(addr,E_1.place,tmp,E.place);}
else Gen(addr, E_1.place,E_2.place,E.place);E_1.type=…}

- **$E \rightarrow E_1*E_2$ {跟上面相似，将addi和addr分别换成muli和mulr}**
- **$E \rightarrow (E_1)$ {E.place=E1.place;E.type=…}**
- **E→V {E.place=V.place; E.type=…}**
- **E→intnum|realnum {E.place=lexval(num);E.type=…}**
- **V→id {V.place=lookup(id.name;V.type=…}**

Addr为实数加

数组说明与下标变量的语义

```
a[i+1]
```

⇩

```
a + (i+1) * sizeof(int)
```

```
a[t]
```

⇩

base_address(**a**) + (***t*** - lower_bound(**a**)) * element_size(**a**)

(1) 地址计算公式

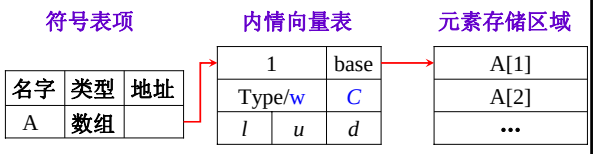
- 对于一维数组A[i]：
  - 下标的变化范围: $l \leqslant i \leqslant u$
  - 连续存储区的首址：*base*，每个数组元素占用*w*个单元
  - 则A[i]地址为：*base+(i-l)*w*

- *base+(i-l)*w=(base-l*w)+i*w*
- 固定部分：*base-l*w*　(在数组说明时即可确定)
- 变化部分：*i*w*　(在数组引用时确定，即动态确定)

- 对于多维数组A[i_1,i_2,…,i_n]：
  - 下标的变化范围: $l_1 \leqslant i_1 \leqslant u_1$;…; $l_n \leqslant i_n \leqslant u_n$
  - 连续存储区的首址：*base*，每个数组元素占用单元*w*个
  - 则A[i_1,i_2,…,i_n]地址为：

$$base+((\ldots((i_1-l_1)*d_2+(i_2-l_2))*d_3+(i_3-l_3)\ldots)d_n+(i_n-l_n))*w$$
$$=base-((\ldots((l_1*d_2+l_2)*d_3+l_3)\ldots)d_n+l_n)*w+$$
$$i_1*d_2*\ldots*d_n+i_2*d_3*\ldots*d_n+\ldots+i_n$$
$$= base- ConstPart+$$
$$(\ldots((i_1)*d_2+i_2)*d_3+i_3)*d_4+i_4)\ldots)*d_n+i_n$$

(2) 数组说明A[*l*:*u*]的表示



- 对于一维数组元素引用形式A[i]，该元素的存储位置为：
  base+(i-l) w= base - C + i*w*

注意:

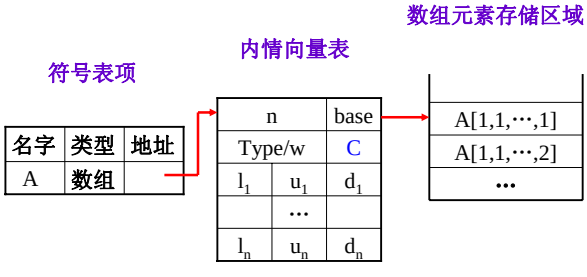
- 固定部分与数组各维的维长d_i和数组的首址base相关（在数组说明时可确定）；
- VARPART部分与下标变量的每个下标相关（在运行时才能确定）
- 计算数组元素的地址时分别计算出CONSTPART和VARPART，对前者静态算出具体值，而对后者则产生计算代码

地址计算公式（续）

- 对于二维数组A[i,j]：
  - 下标的变化范围: $l_1 \leqslant i \leqslant u_1$; $l_2 \leqslant j \leqslant u_2$
  - 连续存储区的首址：*base*，每个数组元素占用*w*个单元
  - 则A[i,j]地址为：*base+[(i-l_1)*(u_2-l_2)+j-l_2]*w*

- *base+[(i-l_1)*(u_2-l_2)+j-l_2]*w*
  =[base+(-l_1(u_2-l_2)-l_2)*w] + [(i*(u_2-l_2)+j)*w]*
- 固定部分： *base*；*(-l_1*(u_2-l_2)-l_2)*w*
- 变化部分： *(i*(u_2-l_2)+j)*w*
- 用维长表示：*(i*d_2+j)*w*

静态数组A[l_1:u_1,…, l_n:u_n]表示形式



计算可变部分

对于不变部分*Constpart*, 产生代码{ T1 := base-C };
可变部分*Varpart*, 产生代码 形如{T:=*Varpart*};
所以数组引用A[i_1,…i_k]的地址为 T + T1,

一般使用变址指令：形式为T1[T] (T1为基址，T为偏移量)
如此，四元式的形式如下：
变址取值 X:= T1[T]
(=[], T1[T], _, X)
变址存储 T1[T] := X
([]=, X, _, T1[T])

属性定义
L.arr 数组名的符号表入口
L.dim 数组维数计数器，随着归约新的下标而增加
L.place 记存业已形成的VARPART的中间结果名字在符号表中的位置，或者是一个临时变量的整数码
Limit(ARRAY,k) 函数过程，数组ARRAY的第k维长度d_k

V.place
简单变量 变量名的符号表入口
下标变量 保存CONSTPART的临时变量的整数码
V.offset
简单变量 NULL(用于区分简单变量和下标变量）
下标变量 保存VARPART的临时变量的整数码

(3) 赋值语句中的数组元素翻译

A → V:= E
V → i[L] | i
L → L,E | E
E → E+E | (E) | V

文法允许数组元素嵌套定义， A[B,C[2]+1]

A→V:=E
{if V.offset=NULL then Gen(:=,E.place,_,V.place)
else Gen([]=,E.place,_,V.place[V.offset])}

E → E_1+E_2
{ t:=NewTemp(); Gen(+,E_1.place,E_2.place,t);
E.place := t }

E → (E_1)
{ E.place := E_1.place }

E → V
{ if (V.offset=NULL) then E.place := V.place;
else { t:=NewTemp();
Gen (=[],V.place[V.offset],_,t); E.place:=t; } }

对下标表L在归约过程中需要知道数组名i的入口，以获取登记在符号表中的数组信息。

V → i[L]|i ⇒ V → L] | i
L → L,E | E L → L,E | i[E

回顾一下VARPART的计算公式，它是一个乘加式。
(…(i_1* d_2 +i_2) d_3 + i_3)…+i_{n-1})d_n+i_n

L.place
limit(i,k),k=3

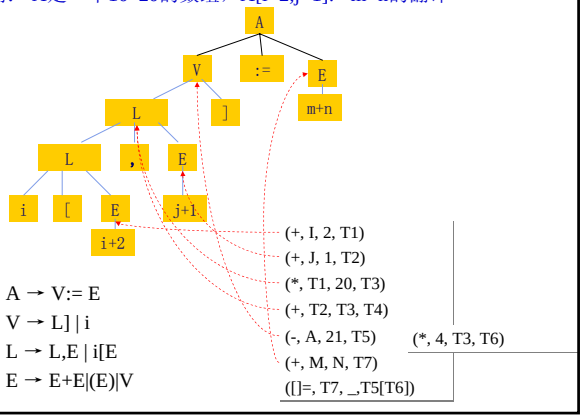
V→L]
{ t:=NewTemp(); Gen(-,acc_base(L.arr),acc_C(L.arr),t);
V.place:=t; t:=NewTemp();
Gen(*, acc_w(L.arr), L.place, t); V.offset :=t }

V → i
{ V.place:= Entry(i); V.offset:= NULL; }

L→L_1,E
{ t1:=NewTemp(); k:= L_1.dim + 1;
d_k:=Limit(L_1.arr,k); Gen(*,L_1.place,d_k,t1);
t2:= NewTemp(); Gen(+,E.place,t1,t2);
L.arr := L_1.arr; L.place := t2; L.dim := k; }

L→ i[E
{ L.place := E.place;
L.dim := 1; L.arr := Entry(i) }

例： A是一个10*20的数组，A[i+2,j+1]:= m+n的翻译



A → V:= E
V → L] | i
L → L,E | i[E
E → E+E|(E)|V

布尔表达式的语义（续）

- 布尔算符的优先级：
  - 顺序：¬，∧，∨; 其中∧和∨服从左结合。另外所有关系符的优先级相同，高于任何布尔算符，低于任何算术算符。
- 由于大部分体系结构没有内置的布尔类型，所以用0和1（或非0）分别表示False和True

7.4 布尔表达式的翻译

- 布尔表达式的语义
- 布尔表达式的求值
- 布尔表达式作为条件的处理

布尔表达式在语言中的作用

- 求值

```
logical a, b
a = .TRUE.
b = a .AND. 3 .LT. 5/2
```

- 作为控制语句中的条件
  - if(1 < i && i < 10) ...

7.4.1布尔表达式的语义

- 文法：E→E∧E | E∨E | ¬ E | (E) | i | i rop i
- C语言：
  - && || ! < == > <= >= !=
  - if(1 < i && i < 10) ...
- Pascal:
  - AND OR NOT < = > <= >=
  - if (x = 0) AND (a = 2) then...
- Fortran逻辑表达式:
  - .AND. .OR. .NOT. .LT. .EQ. .GT. .LE. .GE. .NE.
  - b = a .AND. 3 .LT. 5/2

7.4.2 布尔表达式的求值

- 基本算法: 如同算术表达式求值一样，一步步地计算各部分的值，进而计算出整个表达式的值。

例 A∨B∧C=D ⇒

(=, C, D, T1)
(∧, B, T1, T2)
(∨, A, T2, T3)

基本求值算法的实现

E→E¹∧E² {E.place=newtemp();
gen(∧, E¹.place, E².place, E.place}

E→ E¹∨E² {E.place=newtemp();
gen(∨, E¹.place, E².place, E.place}

E →┐E {E.place=newtemp();
gen(┐, E.place, NIL, E.place}

E →(E¹) {E.place=E¹.place}

E → i {E.place=i.name}

E → i¹ rop i² {E.place=newtemp();
gen(rop, i¹.place, i².place, E.place)}

7.4.3 布尔表达式作为条件的处理

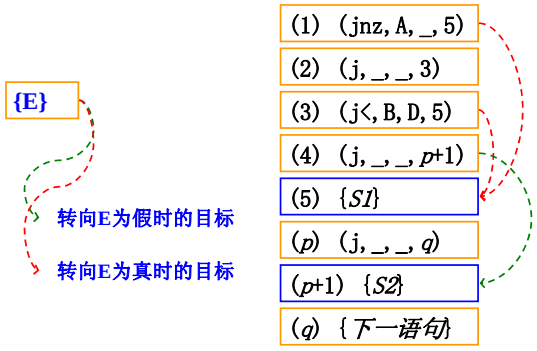
- 真出口，假出口
  - *if E¹ then if E² then S1 else S2 else S2* ⇔
    if E¹ then if E² then goto L-true else goto L-false else goto L-false; L-true: S1; goto L-next; L-false: S2; L-next:
  - *if E¹ then S1 else if E² then S1 else S2* ⇔
    if E¹ then goto L-true else if E² then goto L-true else goto L-false; L-true: S1; goto L-next; L-false: S2; L-next:
  - *if E then S2 else S1* ⇔
    if E then goto L-false else goto L-true; L-true: S1; goto L-next; L-false: S2; L-next:

布尔表达式的求值(续)

- 短路算法：
  - A∨B *if A then true else B*
  - A∧B *if A then B else false*
  - ┐A *if A then false else true*
- 与基本算法的差别：
  - 运算量减少
  - 适用范围广

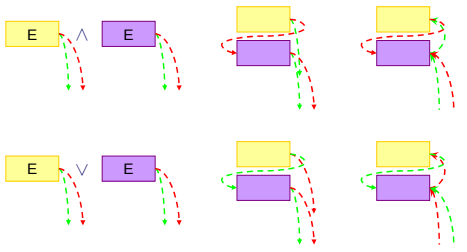
```
if((p!=NULL)&&(p->val==0))…
```

例 IF A∨B<D THEN S1 ELSE S2



短路算法的实现问题

- 求值：转换成条件语句求值
  - A∨B *if A then true else B*
  - A∧B *if A then B else false*
  - ┐A *if A then false else true*
- 作为条件：等价地转换成嵌套if语句
  - if E¹∧E² then S1 else S2
    ⇔ *if E¹ then if E² then S1 else S2 else S2*
  - if E¹ ∨ E² then S1 else S2
    ⇔ *if E¹ then S1 else if E² then S1 else S2*
  - if ┐E then S1 else S2 ⇔ *if E then S2 else S1*



G1:

E-> E∧E|E∨E |¬E|(E)|i|i rop i

G2:

E -> E^E|E° E|¬E|(E)|i|i rop i

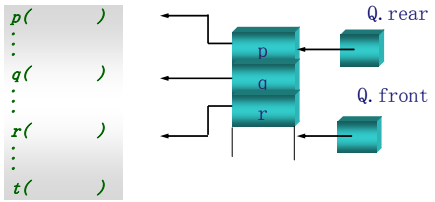
E^ -> E∧ 填E真链

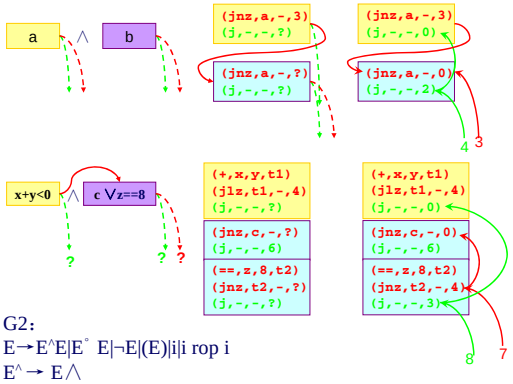
E° -> E∨ 填E假链

解决办法

设p、q、r三条四元式均要转向t四元式

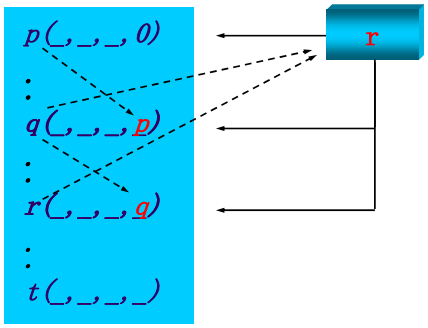
(1) 队列法





G2:
E→E^E|E° E|¬E|(E)|i|i rop i
E^ → E∧
E° → E∨

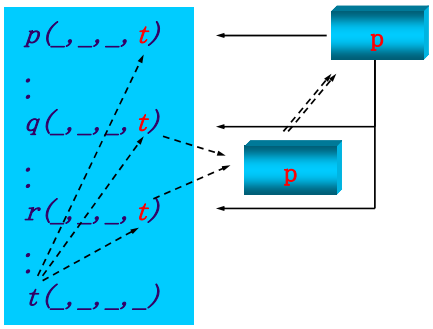
(2) 拉链-返填法



nxq 下一个四元组的编号
bp 回填tc或fc　　merge 合并两个tc或fc

E → i	**{E.tc=nxq++; Gen(jnz,i,_,0); E.fc=nxq++; Gen(j,-,-,0);}**
E^ → E¹ ∧	**{bp (E¹.tc, nxq); E^.fc=E¹.fc;}**
E° → E¹ ∨	**{bp (E¹.fc, nxq); E° .tc=E¹.tc;}**
E → E^ E¹	**{E.tc=E¹.tc; E.fc=merg (E^.fc,E¹.fc); }**
E → E° E¹	**{E.fc=E¹.fc; E.tc=merg (E° .tc,E¹.tc); }**
E → i¹ rop i²	**{E.tc=nxq++; Gen(jrop,i¹,i²,0); E.fc=nxq++; Gen(j,-,-,0);}**
E → (E¹)	**{E.fc=E¹.fc; E.tc=E¹.tc;}**
E → ¬E¹	**{E.tc=E¹.fc; E.fc=E¹.tc;}**

(2) 拉链-返填法



例：(x∧a<b ∨c=d)∧e>f

E → E^E|E° E|¬E|(E)|i|
i rop i
E^ → E∧
E° → E∨

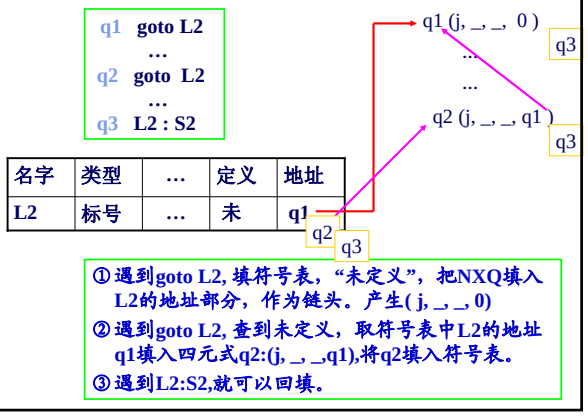
E->i {E.tc=nxq++; E.fc=nxq++; Gen(jnz,i,_,0); Gen(j,-,-,0);}
E^->E¹ ∧ {bp (E¹.tc, nxq); E^.fc=E¹.fc;}
E° ->E¹ ∨ {bp (E¹.fc, nxq); E° .tc=E¹.tc;}
E-> E^ E¹ {E.tc=E¹.tc; E.fc=merg (E^.fc,E¹.fc); }
E-> E° E¹ {E.fc=E¹.fc; E.tc=merg (E° .tc,E¹.tc); }

1 (jnz, x, -, 0) (3)
2 (j,-,-,0) (5)
3 (j<, a, b, 0) (7)
4 (j,-,-,0) (2) (5)
5 (j=, c, d, 0) (3) (7)
6 (j,-,-,0)
7 (j>, e, f, 0)
8 (j,-,-,0) (6)
9

7.5 控制语句

- 标号和转移语句
- 条件语句
- 分支语句

标号先引用后定义



① 遇到goto L2, 填符号表，“未定义”，把NXQ填入L2的地址部分，作为链头。产生(j, _, _, 0)

② 遇到goto L2, 查到未定义，取符号表中L2的地址q1填入四元式q2:(j, _, _,q1),将q2填入符号表。

③ 遇到L2:S2,就可以回填。

7.5.1标号和转移语句

- 标号的两种使用方法
- L: S
- Goto L
- 语言中允许标号先定义后使用，也允许先使用后定义。

一般而言，带标号语句产生式

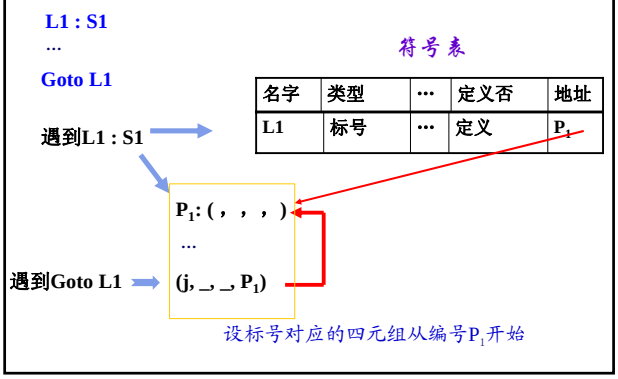
S → *Label S*

Label → i:

Label → i: 的语义动作:

1. 若i所指的标识符(假定为L)不在符号表中，则把它填入，置类型为“标号”，“定义否”为“已”，地址为NXQ。
2. 若L已在符号表中，但“类型”不为“标号”或者“定义否”为“已”，则报告出错。
3. 若L已在符号表中，则把标志“未”改为“已”，然后，把地址栏中的链头(记为q)取出，同时把NXQ填在其中，最后，执行bp(q,NXQ)。

标号先定义后引用



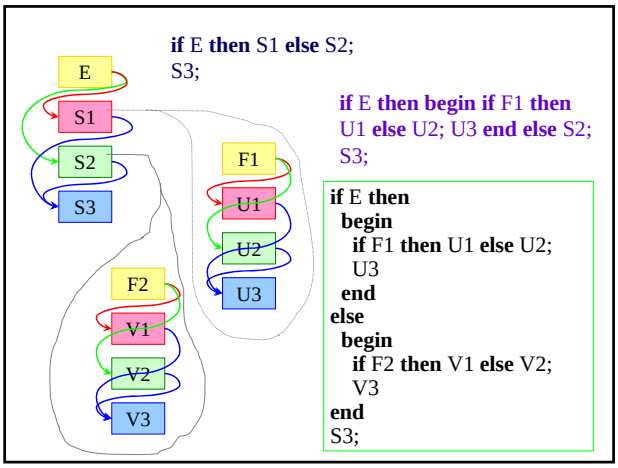
7.5.2条件语句

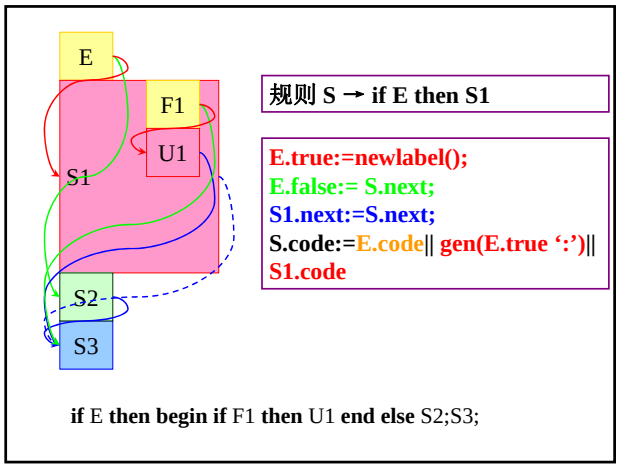
较为复杂的程序控制语句常常是嵌套的。

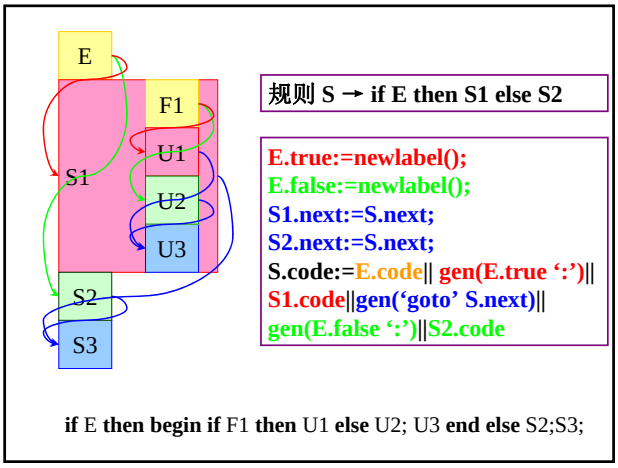
if E1 THEN | if E2 then S1 else S2 | ELSE S3

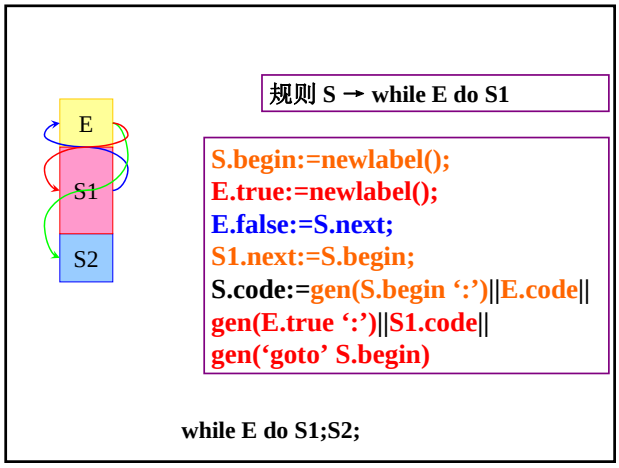
S1后有一条转移指令，转到本if语句之后。与布尔表达式中不同的是，在S2翻译之后，也不能确定这个转移地址，它要跨越S2,S3。 所以，转移地址的确定与语句所处的上下文有关。

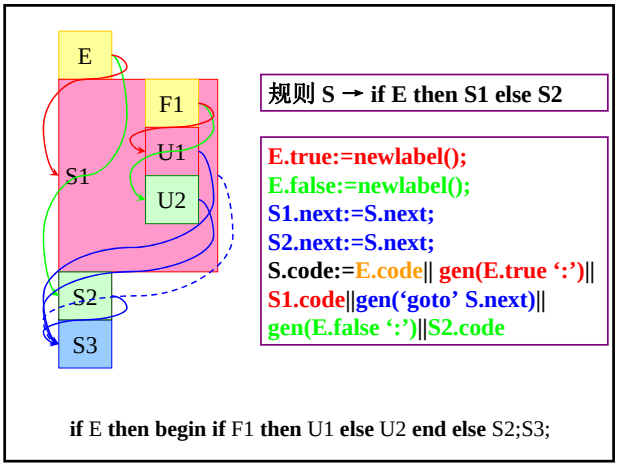
我们采用一个属性“后续链”来记录，实现当后续目标位置确定时返填











产生三地址码的属性文法

- S → if E then S1
  - E.true:=newlabel(); E.false:=S.next; S1.next:=S.next; S.code:=E.code|| gen(E.true ':')||S1.code
- S → if E then S1 else S2
  - E.true:=newlabel(); E.false:=newlabel(); S1.next:=S.next; S2.next:=S.next; S.code:=E.code|| gen(E.true ':')||S1.code||gen('goto' S.next)||gen(E.false ':')||S2.code
- S → while E do S1
  - S.begin:=newlabel();E.true:=newlabel(); E.false:=S.next;S1.next:=S.begin; S.code:=gen(S.begin ':')||E.code|| gen(E.true ':')||S1.code|| gen('goto' S.begin)

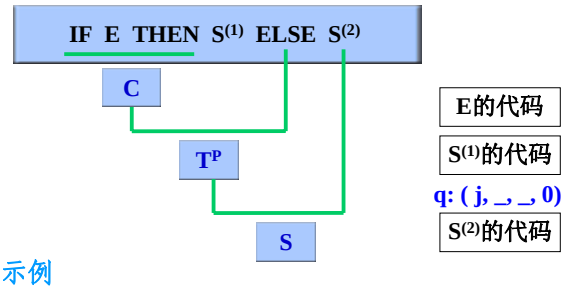
采用四元组实现时需要对其中的goto语句进行处理

嵌套条件语句的拉链返填处理

- 在四元组表示中newlabel()是目标位置的首地址：
- if E then S1 else S2语句中E的真假出口的目标位置分别是S1和S2
- if E then S1语句中E的真假出口的目标位置分别是S1和下一条语句
- if E then S1 [else S2]语句中S1和S2的后续目标位置是该if语句下一条语句
  - 嵌套时下一条语句指外层该语法单位的后续目标位置
- while E do S1语句中E的真假出口的目标位置分别是S1和下一条语句

后续目标的地址只有到那个目标被归约时才知道

C → if E then { bp (E.tc, nxq); C.chain := E.fc; }
T^p → C $S^{(1)}$ else { q := nxq; Gen(j, _, _, 0);
bp(C.chain,nxq); T^p.chain := merge($S^{(1)}$.chain,q);}
S → T^p $S^{(1)}$ { S.chain := merge(T^p. chain, $S^{(1)}$.chain;}



示例

文法修剪

S → if E then S |
if E then S else S |
while E do S |
begin L end | A
L → L; S | S

属性:
tc, fc (真假出口)
chain (后续目标)
quad (While条件)

S → C S | C和S后续目标相同
T^p S | T^p和S后续目标相同
W^d S |
begin L end |
A
C → if E then E真出口已知
T^p → CS else C后续目标已知
W^d → W E do
W → while
L → L^s S |
S
L^s → L;

while 语句

S → C S |
T^p S |
W^d S |
begin L end |
A
C → if E then
T^p → CS else
W^d → W E do
W → while
L → L^s S |
S
L^s → L;

S后续目标已确定
{ bp($S^{(1)}$.chain,W^d.quad);
Gen(j, _, _, W^d.quad);
S.chain := W^d.chain}

W循环体已知
{bp (E.tc,nxq);
W^d.chain := E.fc;
W^d.quad := W.quad}

W条件已知
{ W.quad := nxq }

if 语句

C和S后续目标相同
{S.chain := merg(
C.chain, $S^{(1)}$.chain)}

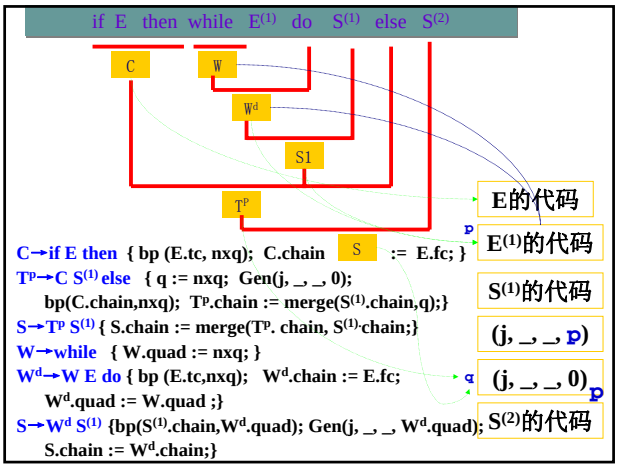
E真出口已知
{bp (E.tc, nxq);
C.chain := E.fc}

S → C S |
T^p S |
W^d S |
begin L end |
A
C → if E then
T^p → CS else
W^d → W E do
W → while
L → L^s S |
S
L^s → L;

T^p和S后续目标相同
{S.chain :=
merg(T^p.chain, $S^{(1)}$.chain)}

C后续目标已知
{q:=nxq; Gen(j, _, _, 0);
bp(C.chain,nxq);
T^p.chain:=merge(S.chain,q)}

C → if E then { bp (E.tc, nxq); C.chain := E.fc; }
S → C $S^{(1)}$ { S.chain := merg(C.chain, $S^{(1)}$.chain);}
T^p → C $S^{(1)}$ else { q := nxq; Gen(j, _, _, 0);
bp(C.chain,nxq); T^p.chain := merge($S^{(1)}$.chain,q);}
S → T^p $S^{(1)}$ { S.chain := merge(T^p. chain, $S^{(1)}$.chain;}
W → while { W.quad := nxq; }
W^d → W E do { bp (E.tc,nxq); W^d.chain := E.fc;
W^d.quad := W.quad ;}
S → W^d $S^{(1)}$ { bp($S^{(1)}$.chain,W^d.quad); Gen(j, _, _, W^d.quad);
S.chain := W^d.chain;}
L → S { L.chain := S.chain;}
L^s → L; { bp(L.chain,nxq); }
L → L^s $S^{(1)}$ { L.chain := $S^{(1)}$.chain;}
S → begin L end { S.chain := L.chain;}
S → A { S.chain := 0;} 空链



关于参数传递－传地址

约定：把实参地址逐一放在转子指令前。

如 CALL S(A+B,Z) 翻成

k-4: T := A+B

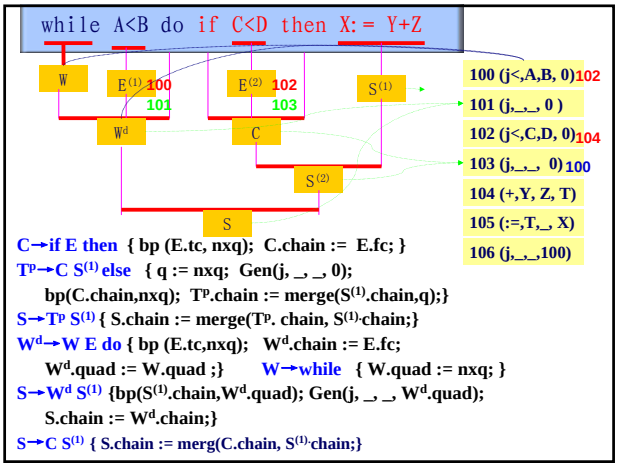
k-3: Par T

k-2: Par Z

k-1: Call S

k: …

进入子程序S之后，S就可根据返回地址k寻找到存放实参地址的单元



分析

文法G：

（1） *S* → CALL i(*Arglist*)

（2） *Arglist* → *Arglist, E*

（3） *Arglist* → *E*

困难：如何在处理实参表的过程之中记住每个实参的地 址，以便最后将它们排列在转子指令的前面。

解决：遇到第一个实参建立一个队列，后面的依次记录，要记住队列头。

7.6 过程调用

- 过程的定义
  - 代码首址记录到符号表中；
  - 形参记录到符号表中；
- 调用者与被调用者
  - 转移目标
  - 返回地址
  - 参数传递

属性文法

S→ CALL i(*Arglist*)

{for (Arglist.queue中每个元素arg)

gen(par,_,_,arg);

gen(call,_,_,entry(i))}

Arglist→ *Arglist*(1), *E*

{E.place进队列Arglist(1).queue;

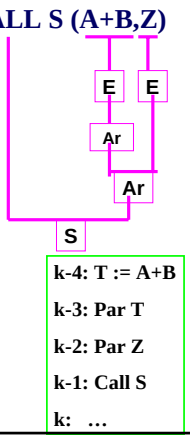
Arglist.queue:=Arglist(1).queue}

Arglist→ *E*

{建立一个Arglist.queue,它只包含一项E.place}

例

S→ CALL i(*arglist*)
{for (arglist.queue中每个元素arg)
gen(par,_,_,arg);
gen(call,_,_,entry(i))}
arglist*→ *arglist*(1), *E
{E.place进队列arglist(1).queue;
arglist.queue:=arglist(1).queue}
arglist*→ *E **{建立一个arglist.queue,**
它只包含一项E.place}



参考书

- Kenneth C.L. 编译原理与实践(英文版)，机械工业2002
- 蒋立源、康慕宁，编译原理(第二版)，西北工业大学出版社2004

本章习题

- p 217-218: 1、3、4、6、7、8

7.7 类型检查

P→ Dlist;Slist

Dlist→ Dlist;D|D

D→ **id**:T

T→ **int**|**bool**|**array [num] of** L

Slist→ Slist;S|S

S→ **if** E **then** S|**id**:=E

类型检查补充

D->**id**:T {insert(**id**.name, T.type)}
T->**int** {E.type=integer}
T->**array [num] of** T(1)
{T.type=mkTypeNode(array,**num**.size,T(1).type)}
S->**if** E **then** S {**if not** typeEqual(E.type,boolean) **then** type-error(S)}
S->**id**:=E {**if not** typeEqual(lookup(**id**.name),E.type)**then** type-error(S)}
E->E(1)+E(2) {**if not** (typeEqual(E(1).type,integer) **and** typeEqual(E(2).type,integer))**then** type-error(E);E.type=integer}
E->E(1)**or**E(2) {**if not** (typeEqual(E(1).type,boolean) **and** typeEqual(E(2).type,boolean))**then** type-error(E);E.type=boolean}
E->num {E.type=integer}
E->true {E.type=boolean}
E->id {E.type=lookup(**id**.name)}