第七章：语义分析和中间代码生成

赵银亮

2012

中间代码生成

- 词法分析⇒语法分析⇒中间代码生成
- 中间代码的表示形式有多种：
  - 逆波兰表示
  - 三地址码（三元式、四元式）
  - 抽象语法树
  - P-代码
- 属性文法是实现中间代码生成的常用手段

本章内容：

- 几种中间表示
- 说明语句的翻译
- 简单算术表达式和赋值句到四元式的翻译
- 布尔表达式到四元式的翻译
- 控制语句的翻译
- 数组元素的引用
- 过程调用

7.1中间表示的概念

- 抽象语法树vs类目标代码
- 是否涉及目标机及运行时环境的信息：
  - 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

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

7.1.1 后缀式—逆波兰表示

- 一个表达式E的后缀形式可以如下定义:
  - 如果E是一个变量或常量，则E的后缀式是E自身；
  - 如果E是E_1 op E_2形式的表达式，其中op是任何二元操作符，则E的后缀式为E_1' E_2' op，其中E_1'和E_2'分别为E_1和E_2的后缀式；
  - 如果E是(E_1)形式的表达式，则E_1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
- 运算符按实际计算次序从左到右排列

对后缀式求值

- 后缀式的特点:
  - 逆波兰表示法不用括号。只要知道每个算符的目数，对于后缀式，不论从哪一端进行扫描，都能对它进行唯一分解。
- 适合于采用栈来求值:
  - 从左到右依次扫描后缀式中的各个符号，每遇到运算对象，就进栈；每碰到k目运算符，就弹出栈顶k个运算对象进行运算，并将结果进栈。结束时，栈顶为整个表达式的值

x/y^z-d*e ⇨ xyz^/de*-

把表达式翻译成后缀式的语义规则描述

产生式	语义动作
$E \rightarrow E^{(1)} op\ E^{(2)}$	$E.code := E^{(1)}.code \mid\mid E^{(2)}.code \mid\mid op$
$E \rightarrow (E^{(1)})$	$E.code := E^{(1)}.code$
$E \rightarrow id$	$E.code := id$

- E.code表示E后缀形式
- op表示任意二元操作符
- "||"表示后缀形式的连接

补充：属性文法

- 语义分析
  - 涉及到计算额外信息（语法分析之外的信息）；
  - 这些信息与程序的语义有关；
  - 编译阶段进行的语义分析叫静态语义分析；
    - 构造符号表；
    - 类型检查；
    - 类型推理；…

进行语义分析的手段

- 语义表示上的困难性
  - 缺乏标准的方法描述语言的语义；
  - 语义分析中的量和种类在不同语言间变化很大；
- 属性文法
  - 标识语言实体的属性；
  - 写出属性方程（或语义规则）表达属性计算与语法规则之间的关系；
  - 属性与属性方程的集合叫属性文法；
  - 适用于语法制导的语言（语义密切关联于语法）

属性+属性方程=属性文法

- 标识语言实体(**language entity**)的属性
  - **An entity is something that has a distinct, separate existence, though it need not be a material existence.**
- 写出属性方程（语义规则）：表明如何把这些属性的计算跟文法规则相关联；
- 属性文法如何使用？
  - 计算属性值
  - 运用属性文法进行语义分析

属性文法，另一种定义

- An attribute grammar is a context-free grammar that has been extended to provide context-sensitive information by appending attributes to some of its nonterminals.
- Each distinct symbol in the grammar has associated with it a finite, possibly empty, set of attributes.
  - Each attribute has a domain of possible values.
  - An attribute may be assigned values from its domain during parsing.
  - Attributes can be evaluated in assignments or conditions.

属性

- 属性：是任意一个跟语言构造相关的特性。
  - **A language construct is a syntactically allowable part of a program that may be formed from one or more lexical tokens in accordance with the rules of a programming language.**
- 属性举例：
  - 变量的类型；
  - 表达式的值；
  - 变量在存储器中的位置；
  - 过程(**procedure**)的目标代码；
  - 数中有效位数。

属性的特点

- 属性所包含的信息、复杂程度、能够确定其值的时间区间等都有较大的范围。
  - 编译之前确定
    - 数中有效位数; …
  - 编译过程中确定
    - …
  - 执行之前确定
    - 表达式的值;
    - 动态分配的数据结构的地址; …

为语言构造指定属性

- 给定上下文无关文法，为它的每个文法符号关联相关的“值”（称为属性），以表示与这些文法符号相关的信息。
- 使用的场合
  - 语法
  - 词法
  - 代码优化与生成

属性的binding

- 计算属性的值及将其跟语言构造建立联系的过程，属性绑定发生的时刻称为绑定时间
  - 静态：在编译过程中（执行之前）
  - 动态：在执行时
  - **变量的类型；(C/Pascal/Lisp Type Checker)**
  - **表达式的值；(Code/Constant Folding)**
  - **变量在存储器中的位置；(Static/Dynamic Allocation)**
  - **过程(procedure)的目标代码；(Static)**
  - **数中有意义数字的个数。(Runtime Env.)**

属性方程的表示

- 将属性值a与文法符号X联系起来，记为X.a
- 对于产生式规则$X_0 \rightarrow X_1X_2\ldots X_n$,有

$$X_i.a_j = f_{ij}(X_0.a_1,\ldots,X_0.a_k,X_1.a_1,\ldots,X_1.a_k,\ldots,X_n.a_1,\ldots,X_n.a_k)$$

其中，f_{ij}为数学函数，

则称为属性方程或语义规则

- 一般采用表格方式表示

属性文法的表示形式

文法规则	语义规则
产生式规则 1	关联的属性方程（多个）
…	…
产生式规则 *n*	关联的属性方程（多个）

补充完

产生式	语义动作
$E \rightarrow E^{(1)} op\ E^{(2)}$	{E.code:= $E^{(1)}$.code || $E^{(2)}$.code ||op}
$E \rightarrow (E^{(1)})$	{E.code:= $E^{(1)}$.code}
$E \rightarrow id$	{E.code:=id}

$E \rightarrow E^{(1)} op\ E^{(2)}$	E.code:= $E^{(1)}$.code || $E^{(2)}$.code ||op
$E \rightarrow (E^{(1)})$	E.code:= $E^{(1)}$.code
$E \rightarrow id$	E.code:=id

- 数组POST存放后缀式：k为下标，初值为1
- 上述语义动作另一种形式：

产生式	语义动作
$E \rightarrow E^{(1)} op\ E^{(2)}$	{POST[k]:=op; k:=k+1}
$E \rightarrow (E^{(1)})$	{}
$E \rightarrow i$	{POST[k]:=i; k:=k+1}

- 例：输入串a+b+c的分析和翻译

POST：

1	2	3	4	5
a	b	+	c	+	…

后缀式的一般情况

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

例：

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

⇒ *a*?*c*-*d*?*a*+*c*:*a**c*:*a*+*b*

⇒ a cd- ac+ ac*? ab+?

后缀式表示中注意的问题

- **exy?的语义**
  - **期望： e不等于0，返回x值，否则返回y值.**
  - **实际： x和y都要计算，但只返回其中一个的值**
- **问题**
  - **语义差异：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；**
  - **代码效率**

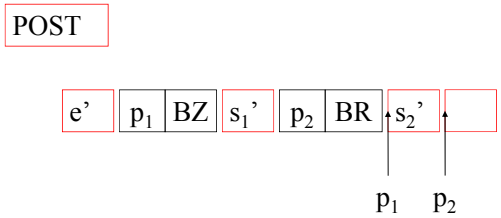
后缀式表示实现

- 将后缀式表示存放在一维向量POST[1..n]中
- 引入转移操作：
  - p BR 无条件转至POST[p]
  - e' p BZ 当e'值为0转POST[p]，其中 e'是某表达式e的后缀式表示
  - 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)

后缀式表示实现（例）

- IF e THEN s1
  - e' p2 BZ s1' p2 BR
  - e' p2 BZ s1' p2:
- IF e THEN s1 ELSE s2
  - e' p1 BZ s1' p2 BR p1:s2' p2 BR
  - e' p1 BZ s1' p2 BR p1:s2' p2:

其中e', s1', s2'是e, s1, s2的后缀式表示, p1表示s2'在POST中的开始位置，p2表示该IF语句的后继。



利用属性文法产生后缀式表示

- 属性addr：纪录运算对象的起始位置
- 向量POST和指针p

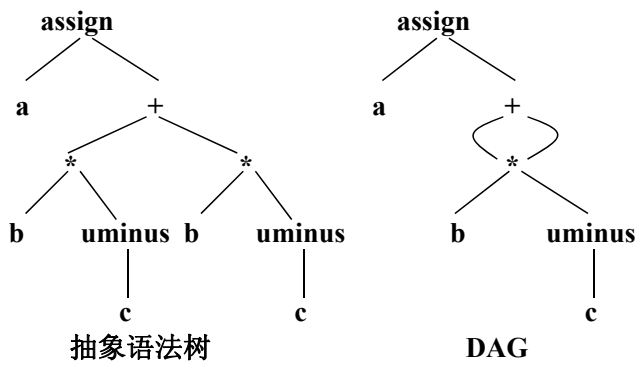
- E1→E2+T {E1.addr=E2.addr; POST[p]='+'; p++;}
- E→T {E.addr=T.addr}
- T1→T2*F {T1.addr=T2.addr; POST[p]='*'; p++;}
- T→F {T.addr=F.addr;}
- F→(E) {F.addr=E.addr;}
- F→id {F.addr=p; POST[p]=lookup(id); p++;}

步骤	符号栈	R	输入串	动作	后缀表示
0	#	a	+b*c#	shift	
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	+	b*c#	shift	a
5	#E+	b	*c#	shift	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#	shift	ab
9	# E+T*	c	#	shift	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*+

7.1.2 图表示法

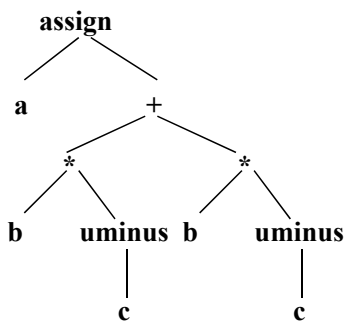
- 图表示法
  - DAG
  - 抽象语法树
- 无循环有向图(Directed Acyclic Graph，简称DAG)
  - 对表达式中的每个子表达式，DAG中都有一个结点
  - 一个**内部结点**代表一个**操作符**，它的孩子代表操作数
  - 在一个DAG中代表公共子表达式的结点具有多个父结点

a:=b*(-c)+b*(-c)的图表示法



抽象语法树　　DAG

抽象语法树与中间代码对应示例



抽象语法树

对应的代码：

T_1:=-c

T_2:=b*T_1

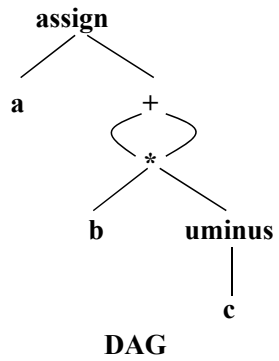
T_3:=-c

T_4:=b*T_3

T_5:=T_2+T_4

a:=T_5

例：DAG与中间代码



DAG

抽象语法树对应的代码：

T_1:=-c

T_2:=b*T_1

T_3:=-c

T_4:=b*T_3

T_5:=T_2+T_4

a:=T_5

DAG对应的代码：

T_1:=-c

T_2:=b*T_1

T_5:=T_2+T_2

a:=T_5

产生赋值语句抽象语法树的属性文法

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	E.tree=*mkNode*(uminus, E_1.tree, nil)
E→(E_1)	E.tree=E_1.tree
E→id	F.tree=*mkLeaf*(id, id.place)

7.1.3 三地址代码

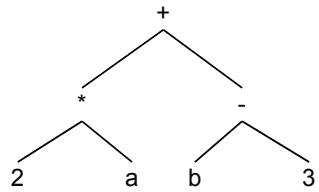
- 三地址代码

  x:=y op z
- 三地址代码可以看成是抽象语法树或DAG的一种线性表示

三地址语句示例

- x:=y op z
- x:=op y
- x:=y
- goto L
- if x relop y goto L或if a goto L
- par x和call p,n，以及返回语句return y
- x:=y[i]及x[i]:=y的索引赋值
- x:=&y, x:=*y和*x:=y的地址和指针赋值

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;

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
```

三地址码的实现：四元组

- 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,_,_,_)
```

三地址码的实现：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))

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,_,_,_)
```

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

方法1：用符号串表示代码

$E_1 \rightarrow$ **id**$=E_2$	E_1.name=**id**.strval E_1.code=E_2.code++**id**.strval||"="||E_2.name
$E \rightarrow A$	E.name=A.name; E.code=A.code
$A_1 \rightarrow A_2$+F	A_1.name=newtemp() A_1.code=A_2.code++F.code++ A_1.name||"="||A_2.name||"+"||F.name
$A \rightarrow F$	A.name=F.name; A.code=F.code
$F \rightarrow (E)$	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

方法2：用符号串表示代码

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()
{
int i,j;
float a[3,3],sum;
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);
}
```

过程中的说明语句

P → D

D → D;D | TL

T → int | float

L → id, L | id

题目：求一个3*3矩阵对角线元素之和

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

文法规则	语义规则
P → D	**offset=0**
$D_1 \rightarrow D_2;D_3$	
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**

补充：综合属性与继承属性

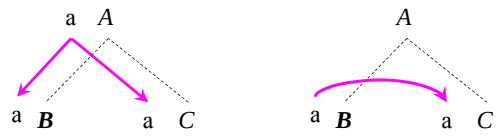
- 一个属性是综合属性是指在语法树中，该属性的值由其孩子结点的属性值计算而得。
- a是综合属性，若给定$A \rightarrow X_1X_2 \ldots X_n$,唯一与其关联的属性方程中a在等号左边

$$A.a = f(X_1.a_1,\ldots,X_1.a_k, \ldots,X_n.a_1,\ldots,X_n.a_k)$$

- **S-属性文法**
  - 所有属性都是综合属性的属性文法

继承属性

- 不是综合属性的属性是继承属性
  - 语法树中，一个结点的继承属性由此结点的父结点和或兄弟结点的某些属性确定。



(a) Inheritance from parent to siblings

(b) Inheritance from sibling to sibling

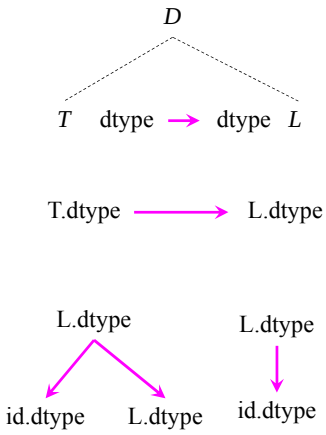
补充：依赖图

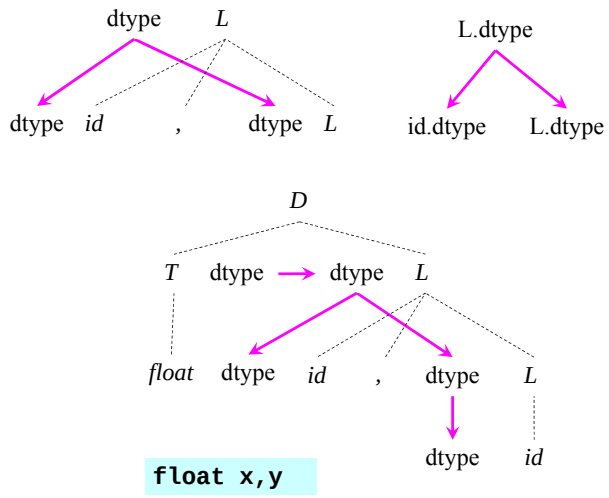
D → TL

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**





作用域

P→D

D→D;D | TL |

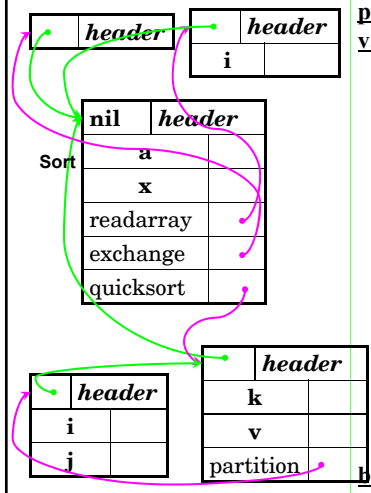
proc id;D;S |

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.
```




```
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);
  begin…end;
begin…end.
```

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。

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); push(0,s_offset)**
D→D;D	
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))** **push(pop(s_offset)+T.width, s_offset)**
N→ε	**t:=mktab(top(s_tab))**

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_.

push(t, s_tab); push(0,s_offset)

(方法1)

P→MD	addwidth(pop(s_tab), pop(s_offset))
M→ε	t:=mktab(nil); push(t, s_tab); push(0,s_offset)
D→D;D	
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)) push(pop(s_offset)+T.width, s_offset)
N→ε	t:=mktab(top(s_tab)); push(t, s_tab); push(0,s_offset)

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

(方法2)

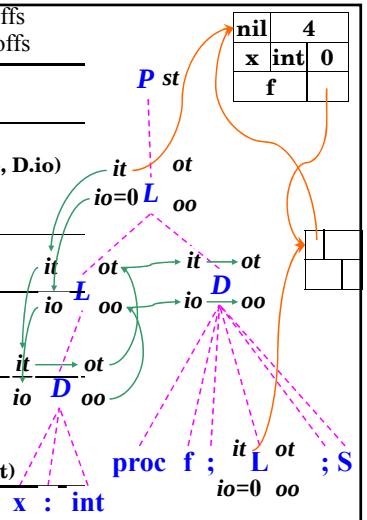
P→L	L.intab=mktab(nil); L.ioffs=0 P.symtab=L.outtab
D→ id:T	D.outtab=D.intab Enter(D.outtab, id.name, T.type, D.ioffs) D.ooffs=D.ioffs+T.width addwidth(D.outtab,D.ooffs))
L→D	D.intab=L.intab; L.outtab=D.outtab D.ioffs=L.ioffs;L.ooffs=D.ooffs
L$_1$→ L$_2$;D	L$_2$.intab=L$_1$.intab; D.intab=L$_2$.outtab L$_1$.outtab=D.outtab; L$_2$.ioffs=L$_1$.ioffs D.ioffs=L$_2$.ooffs; L$_1$.ooffs=D.ooffs
D→ proc id;L;S	L.intab=mktab(D.intab); L.ioffs=0 addwidth(L.intab,L.ooffs)) D.outtab=D.intab; D.ooffs=D.ioffs enterproc(D.intab,id.name,D.outtab)

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

(方法2)

ot outtab *io* ioffs
it inttab *oo* ooffs

P→L	L.it=mktab(nil); L.io=0 P.st=L.ot
D→ id:T	D.ot=D.it Enter(D.ot, id.name, T.type, D.io) D.oo=D.io+T.width addwidth(D.ot,D.oo))
L→D	D.it=L.it; L.ot=D.ot D.io=L.io;L.oo=D.oo
L$_1$→ L$_2$;D	L$_2$.it=L$_1$.it; D.it=L$_2$.ot L$_1$.ot=D.ot; L$_2$.io=L$_1$.io D.io=L$_2$.oo; L$_1$.oo=D.oo
D→ proc id;L;S	L.it=mktab(D.it); L.io=0 D.ot=D.it; D.oo=D.io addwidth(D.ot,D.oo)) enterproc(D.ot,id.name,L.ot)



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 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

前面介绍过的一个例子

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

7.3.1 属性及过程定义（产生四元组）

- **NewTemp()** 函数过程，返回一个新临时变量名 (一般用整数表示)；
- **Lookup(name)** 以name为名字查符号表：查到返回入口，否则加入该名字到符号表中并返回入口；
- **E.place** 表示存放E值的变量名在符号表的入口或者整数（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→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→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→(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}

去掉错误处理部分

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

7.3.2 类型转换

A→V:=E {Gen(:=,E.place,nil,V.place);}

$E \to E_1+E_2$ {E.place=NewTemp(); **Addr为实数加**
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.type=…}

$E \to E_1*E_2$ {跟上面相似，将addi和addr分别换成muli和mulr}

$E \to (E_1)$ {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=…}

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

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?

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: $l \le i \le u$
  - 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：$base+(i-l)*w$
- **A[i]**地址：$base+(i-l)*w=(base-l*w)+i*w$
- 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）

(2) 数组说明A[*l*:*u*]的表示

符号表项　　内情向量表　　元素存储区域

名字	类型	地址
A	数组	

1		base
Type/w		C
l	*u*	*d*

A[1]
A[2]
…

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

地址计算公式（续）

- 对于二维数组**A[i,j]**：
  - 下标的变化范围: $l_1 \le i \le u_1$; $l_2 \le j \le u_2$
  - 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：$(i*d_2+j)*w$

- 对于多维数组$A[i_1,i_2,\ldots,i_n]$：
  - 下标的变化范围: $l_1 \le i_1 \le u_1;\ldots; l_n \le i_n \le u_n$
  - 连续存储区的首址：*base*，每个数组元素占用单元*w*个
- 则$A[i_1,i_2,\ldots,i_n]$地址为：
- $base+((\ldots((i_1-l_1)*d_2+(i_2-l_2))*d_3+(i_3-l_3)\ )\ldots)d_n+(i_n-l_n))*w$

$$=base-((\ldots((l_1*d_2+l_2)*d_3+l_3)\ldots)d_n+l_n)*w+$$

$$i_1*d_2*\ldots*d_n+i_2*d_3*\ldots*d_n+\ldots+i_n$$

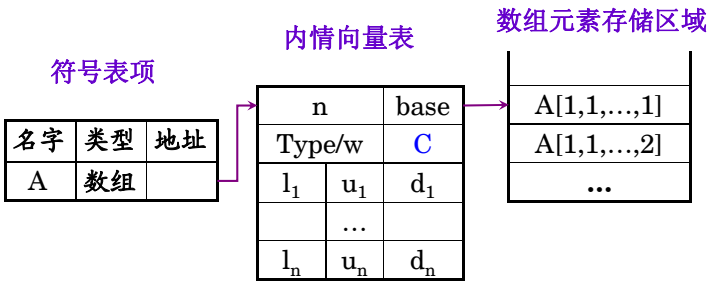
$$= base- ConstPart+$$

$$(\ldots((i_1)*d_2+i_2)*d_3+i_3)*d_4+i_4)\ldots)*d_n+i_n$$

注意：

- **CONSTPART**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

静态数组$A[l_1:u_1,\cdots, l_n:u_n]$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$A[i_1,\ldots i_k]$的地址为 **T + T1**,

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])

(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

B	数组	
A	数组	

1		base
T_B		C
l	u	d

2		$base_2$
T_A		C_2
l_1	u_1	d_1
l_2	u_2	d_2

文法允许数组元素嵌套定义，B[i]:=A[v,B[j]+w]

可变部分的计算

V → i[L] | i
L → L,E | E

⇒

V → L] | i
L → L,E | i[E

- $e_1=t_1$
- $e_m=e_{m-1}*d_m+t_m$

VARPART=$(\ldots((i_1)*d_2+i_2)*d_3+i_3)*d_4+i_4)\ldots)*d_n+i_n$

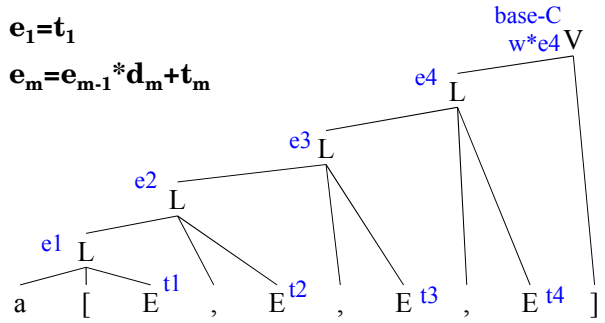
L.place

limit(i,k),k=3

对下标变量的处理

V → L] | i
L → L,E | i[E

- $e_1=t_1$
- $e_m=e_{m-1}*d_m+t_m$



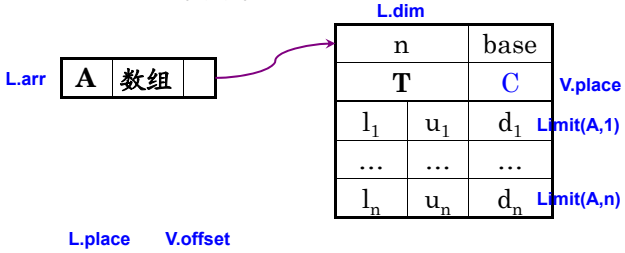
属性定义

L.arr 该数组名在符号表中的入口位置
L.dim 数组维数计数器，随着归约新的下标而增加
L.place 存放已形成的VARPART部分，可能是变量或临时变量
Limit(A,k) 函数过程，数组A的第k维长度为d_k

V.place
对于简单变量记录该变量名在符号表中的位置
对于下标变量记录保存CONSTPART的那个临时变量
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

(3) 赋值语句中的数组元素翻译

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



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; } }

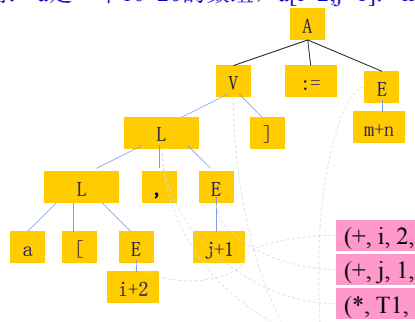
V→L]
{ t:=NewTemp(); Gen(-,acc_base(L.arr),acc_C(L.arr),t);
V.place:=t; t:=NewTemp();
Gen(*, acc_w(L.arr), L.place, t); V.offset :=t }

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

(+, i, 2, T1)
(+, j, 1, T2)
(*, T1, 20, T3)
(+, T2, T3, T4)
(-, a, 21, T5)
(*, 4, T4, T6)
(+, m, n, T7)
([]=, T7, ,T5[T6])

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

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**

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**

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) ...**

7.4.2 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

$E \rightarrow E^1 \wedge E^2$	{E.place=newtemp(); gen(∧, E^1.place, E^2.place, E.place}
$E \rightarrow E^1 \vee E^2$	{E.place=newtemp(); gen(∨, E^1.place, E^2.place, E.place}
$E \rightarrow \neg E$	{E.place=newtemp(); gen(¬ , E.place, NIL, E.place}
$E \rightarrow (E^1)$	{E.place=E^1.place}
$E \rightarrow i$	{E.place=i.name}
$E \rightarrow i^1$ rop i^2	{E.place=newtemp(); gen(rop, i^1.place, i^2.place, E.place)}

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*

```
if((p!=NULL)&&(p->val==0))…
```

- 与基本算法的差别:
  - 运算量减少
  - 适用范围广

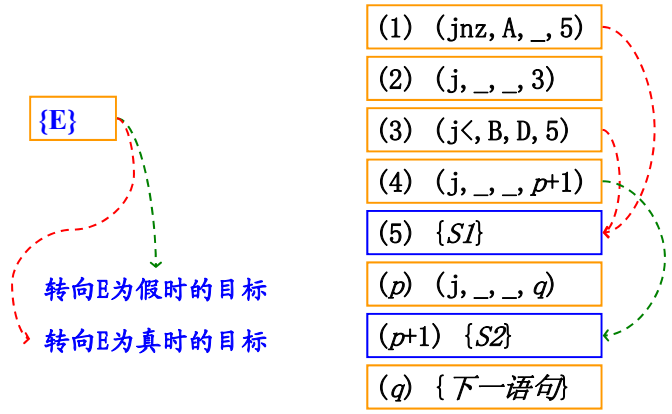
短路算法的实现问题

- 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 $E^1 \wedge E^2$ then S1 else S2
    ⇔ *if E^1 then if E^2 then S1 else S2 else S2*
  - if $E^1 \vee E^2$ then S1 else S2
    ⇔ *if E^1 then S1 else if E^2 then S1 else S2*
  - if ¬ E then S1 else S2 ⇔ *if E then S2 else S1*

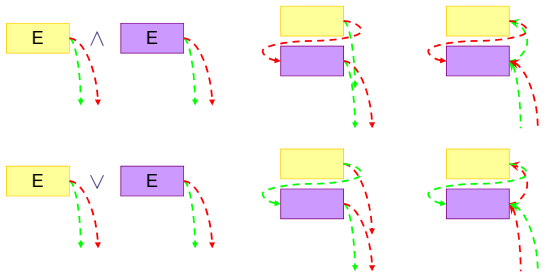
7.4.3 布尔表达式作为条件的处理

- 真出口，假出口
  - ***if E^1 then if E^2 then S1 else S2 else S2*** ⇔
    if E^1 then if E^2 then goto L-true else goto L-false else goto L-false; L-true: S1; goto L-next; L-false: S2; L-next:
  - ***if E^1 then S1 else if E^2 then S1 else S2*** ⇔
    if E^1 then goto L-true else if E^2 then goto L-true else goto L-false; L-true: S1; goto L-next; L-false: S2; L-next:
  - ***if E then S1 else S2*** ⇔
    if E then goto L-false else goto L-true; L-true: S1; goto L-next; L-false: S2; L-next:

例 if A∨B<D then S1 else S2



文法修剪



G1：

E-> E∧E|E∨E |¬E|(E)|i|i rop i

G2：

E -> E^ E|E°E|¬E|(E)|i|i rop i

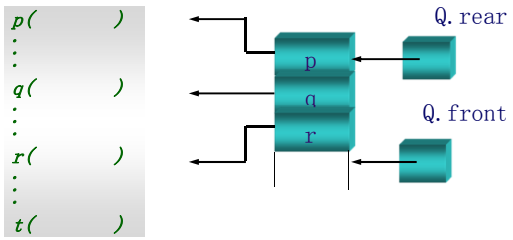
E^ -> E∧ 填E真链

E° -> E∨ 填E假链

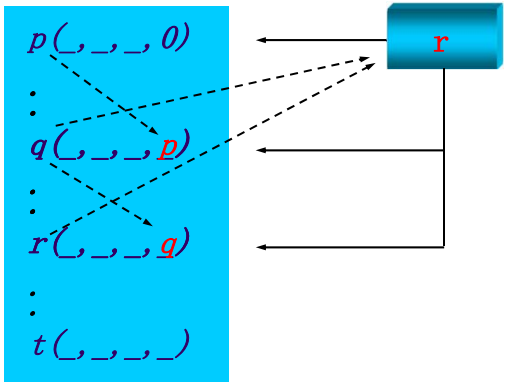
回填真假链的方法

队列法

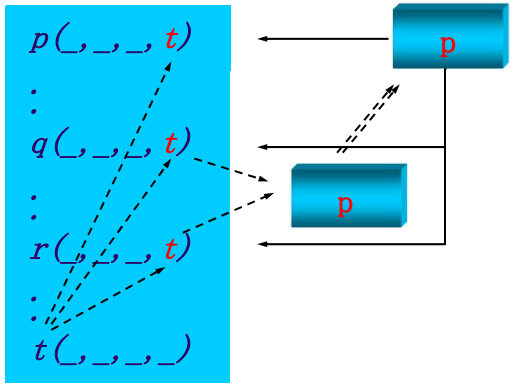
设p、q、r三条四元式均要转向t四元式



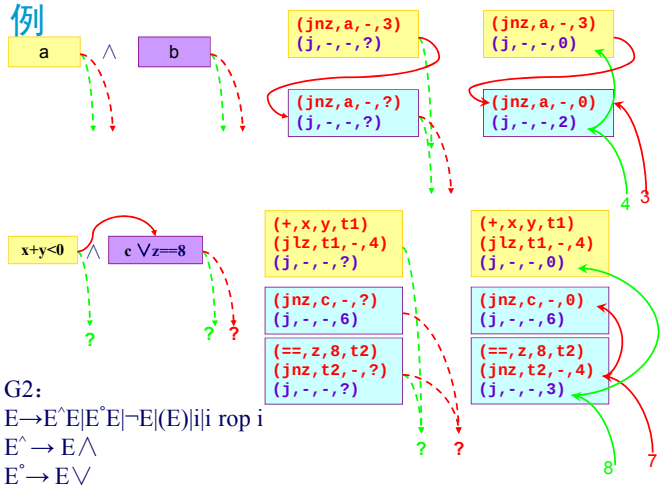
回填真假链的方法-拉链返填



拉链返填法（续）



例



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

属性及过程定义

- **nxq** 下一个四元组的编号
- **merg(p, q)** 把p链链接到q链尾，返回q
- **bp(p, t)** 把p链上所有四元组的第4区均回填为t

- **E.tc** E的代码中需回填真出口的那些四元组构成的链表
- **E.fc** E的代码中需回填假出口的那些四元组构成的链表

nxq 下一个四元组的编号

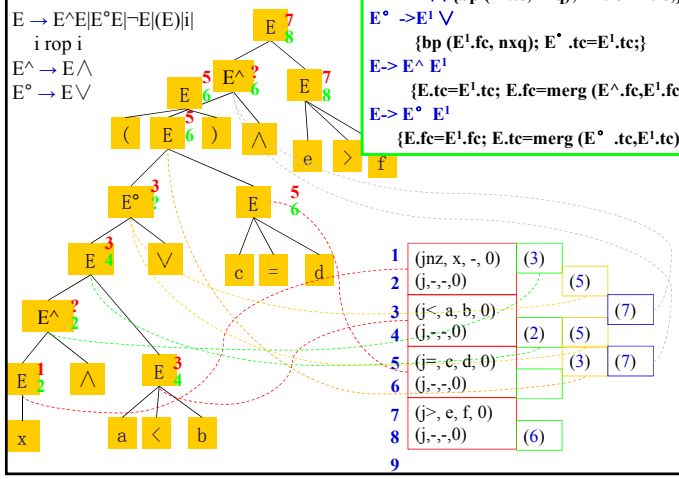
bp 回填tc或fc　　merg 合并两个tc或fc

$E \rightarrow i$	{E.tc=nxq; Gen(jnz,i,_,0); E.fc=nxq; Gen(j,-,-,0);}
$E^{\wedge} \rightarrow E^1 \wedge$	{bp (E^1.tc, nxq); $E^{\wedge}$.fc=E^1.fc;}
$E^{\circ} \rightarrow E^1 \vee$	{bp (E^1.fc, nxq); E°.tc=E^1.tc;}
$E \rightarrow E^{\wedge} E^1$	{E.tc=E^1.tc; E.fc=merg ($E^{\wedge}$.fc,E^1.fc); }
$E \rightarrow E^{\circ} E^1$	{E.fc=E^1.fc; E.tc=merg (E°.tc,E^1.tc); }
$E \rightarrow i^1$ rop i^2	{E.tc=nxq; Gen(jrop,i^1,i^2,0); E.fc=nxq; Gen(j,-,-,0);}
$E \rightarrow (E^1)$	{E.fc=E^1.fc; E.tc=E^1.tc;}
$E \rightarrow \neg E^1$	{E.tc=E^1.fc; E.fc=E^1.tc;}

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; Gen(jnz,i,_,0); E.fc=nxq; Gen(j,-,-,0);}
E^->E^1 ∧ {bp (E^1.tc, nxq); E^.fc=E^1.fc;}
E° ->E^1 ∨ {bp (E^1.fc, nxq); E°.tc=E^1.tc;}
E-> E^ E^1 {E.tc=E^1.tc; E.fc=merg (E^.fc,E^1.fc); }
E-> E° E^1 {E.fc=E^1.fc; E.tc=merg (E°.tc,E^1.tc); }



7.5 控制语句

- 标号和转移语句
- 分支与循环语句

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。

①标号先定义后引用

L1 : S1
…
Goto L1

符号表

名字	类型	…	定义否	地址
L1	标号	…	定义	P_1

遇到L1 : S1

P_1: (, , ,)
…
遇到Goto L1 → (j, _, _, P_1)

设标号对应的四元组从编号P_1开始

②标号先引用后定义

q1 goto L2
…
q2 goto L2
…
q3 L2 : S2

名字	类型	…	定义	地址
L2	标号	…	未	q1

q1 (j, _, _, 0) q3
…
…
q2 (j, _, _, q1) q3

q2 q3

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。

一般而言，带标号语句产生式

S → *Label S*
Label → i:

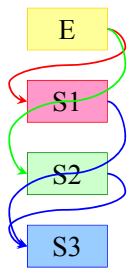
Label → i: 的语义动作:

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)。

7.5.2分支与循环语句

- *if E then S1 else S2* ⇔

  if E then goto L-true else goto L-false; L-true: S1; goto L-next; L-false: S2; L-next:

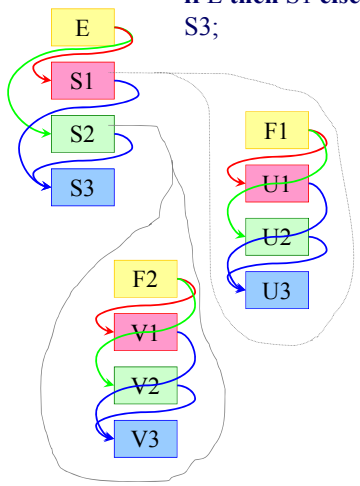


⇨ E.tc, E.fc; 回填E.tc

⇨ S1.next; 回填E.fc

⇨ S2.next;

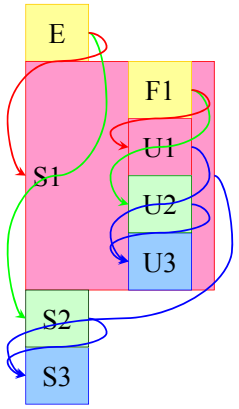
E,S1和S2代码是相邻的，共同构成S代码。S.next



if E **then** S1 **else** S2;
S3;

if E **then begin if** F1 **then**
U1 **else** U2; U3 **end else** S2;
S3;

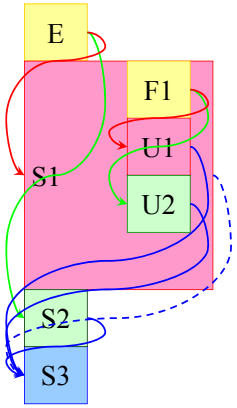
```
if E then
  begin
    if F1 then U1 else U2;
    U3
  end
else
  begin
    if F2 then V1 else V2;
    V3
end
S3;
```



规则 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

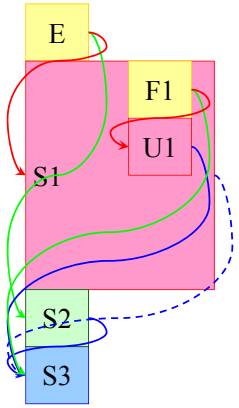
if E **then begin if** F1 **then** U1 **else** U2; U3 **end else** S2;S3;



规则 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

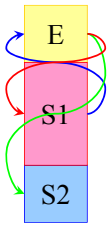
if E **then begin if** F1 **then** U1 **else** U2 **end else** S2;S3;



规则 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

if E **then begin if** F1 **then** U1 **end else** S2;S3;



规则 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)

while E do S1;S2;

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

文法修剪

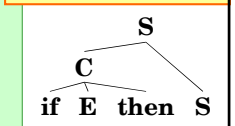
S → if E then S |
if E then S else S |
while E do S |
begin L end | A
L → L; S | S

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

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;

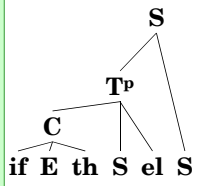
C和S后续目标相同

T^p和S后续目标相同



E真出口已知

C后续目标已知



if 语句

C和S后续目标相同

{S.chain := merg(C.chain, $S^{(1)}$.chain)}

$S \to C\ S \mid$
$T^p\ S \mid$
$W^d\ S \mid$
begin L end $\mid$
A

T^p和S后续目标相同

{S.chain := merg(T^p.chain, $S^{(1)}$.chain)}

E真出口已知

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

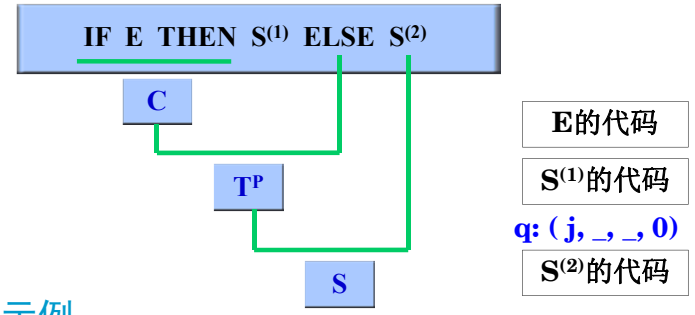
$C \to$ if E then
$T^p \to CS$ else
$W^d \to W\ E$ do
$W \to$ while
$L \to L^s\ S \mid$
S
$L^s \to L;$

C后续目标已知

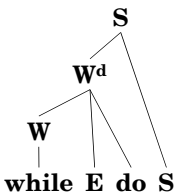
{q:=nxq; Gen(j, _, _, 0); bp(C.chain,nxq); T^p.chain:=merge(S.chain,q)}

示例

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



while 语句



$S \to C\ S \mid$
$T^p\ S \mid$
$W^d\ S \mid$
begin L end $\mid$
A
$C \to$ if E then
$T^p \to CS$ else
$W^d \to W\ E$ do
$W \to$ while
$L \to L^s\ S \mid$
S
$L^s \to L;$

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}

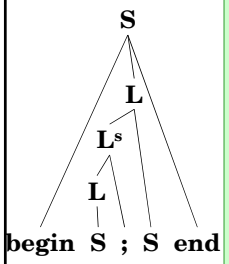
W条件已知

{ W.quad := nxq }

W循环体已知

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

复合语句



$S \to C\ S \mid$
$T^p\ S \mid$
$W^d\ S \mid$
begin L end $\mid$
A
$C \to$ if E then
$T^p \to CS$ else
$W^d \to W\ E$ do
$W \to$ while
$L \to L^s\ S \mid$
S
$L^s \to L;$

L后续就是S后续

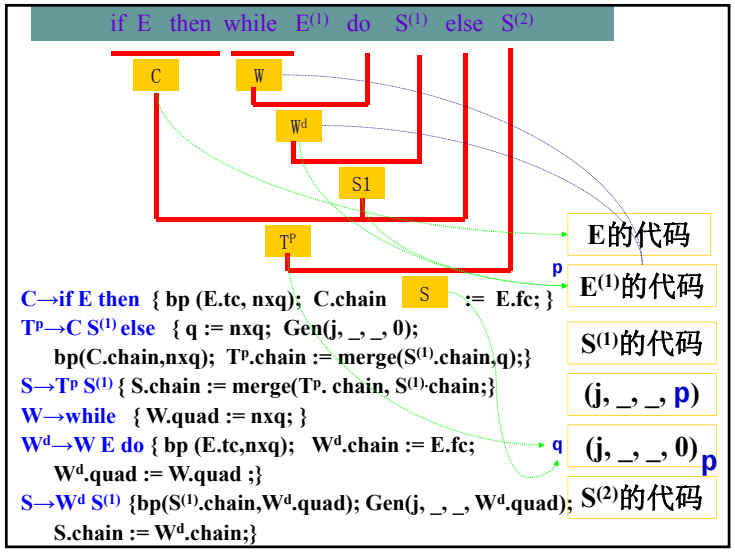
A后续为空

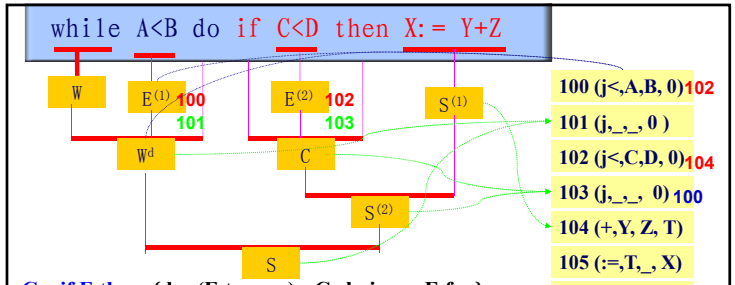
S后续就是L后续

S后续就是L后续

L的后续已知

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





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

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

```
procedure S(var a, b:integer);
…
```

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） $S \rightarrow$ CALL i(*Arglist*)

（2） *Arglist* $\rightarrow$ *Arglist*, *E*

（3） *Arglist* $\rightarrow$ *E*

困难：如何在处理实参表的过程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。

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*

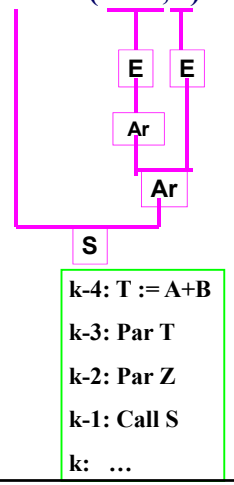
{E.place进队列arglist$^{(1)}$.queue;

arglist.queue:=arglist$^{(1)}$.queue}

arglist→ *E* {建立一个arglist.queue,

它只包含一项E.place}

CALL S (A+B,Z)



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

类型表达式

- 基本类型是类型表达式
- 类型名是类型表达式
- 用于类型表达式得到类型表达式

- 基本类型
  - **boolean, char, integer, real**
  - **type-error**
  - **void**

- 类型构造符
  - 数组 **array(I, T)**
  - 笛卡尔积 $T_1 \times T_2$
  - 指针 **pointer(T)**
  - 函数 **D→R**

- 类型系统
  - A language's type system specifies which operations are valid for which types
- 类型检查：动态; 静态
  - The goal of type checking is to ensure that operations are used with the correct types
- 良类型系统

确定标识符的类型

P → D;E

D → D;D

D → id:T {addtype(id.entry, T.type}

T → char {T.type=char}

T → integer {T.type=integer}

T → ↑T_1 {T.type=pointer(T_1.type)}

T → array[num]of T {T.type=array(num.val, T_1.type)}

类型检查

- **$E \rightarrow E_1$ mod E_2 {if E_1.type=integer and E_2.type=integer then E.type=integer else E.type=type_error}**
- **$E \rightarrow E_1[E_2]$ {if E_2.type=integer and E_1.type=array(s,t) then E.type=t else E.type=type_error}**
- **$E \rightarrow E_1\uparrow$ {if E_1.type=pointer(t) then E.type=t else E.type=type_error}**
- **$E \rightarrow E_1(E_2)$ {if E_2.type=s and E_1.type=s→t then E.type=t else E.type=type_error}**

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)}

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