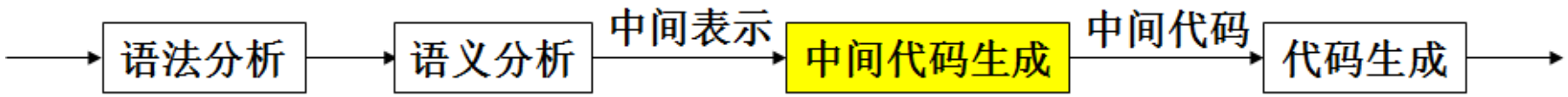
第8章　中间代码生成

重庆大学　葛亮

知识点：三地址代码
语句的翻译
布尔表达式的翻译
回填技术

中间代码生成

- 中间代码生成程序的任务
  - 把经语法分析、语义分析后得到的源程序的中间表示形式翻译成中间代码表示。
- 采用中间代码作为过渡的优点
  - 便于编译程序的建立和移植
  - 便于进行与机器无关的代码优化工作
- 缺点
  - 增加了I/O操作、效率有所下降
- 中间代码生成程序的位置：



中间代码生成

8.1 中间代码形式

8.1.1 图形表示

- 语法树
- dag图

8.1.2 三地址代码

- 三地址语句的形式
- 三地址语句的种类
- 三地址语句的实现

8.1.1 图形表示

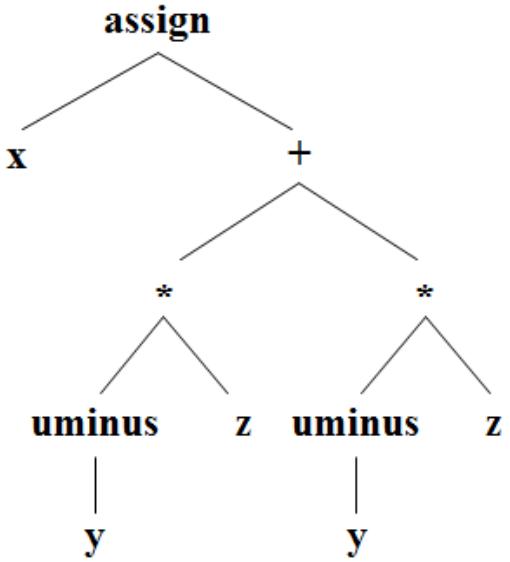
- 语法树
  - 描绘了源程序的自然层次结构。
- dag图
  - 以更紧凑的方式给出了与语法树同样的信息。
  - 在dag中，公共子表达式被标识出来了。

为赋值语句构造语法树的语法制导定义

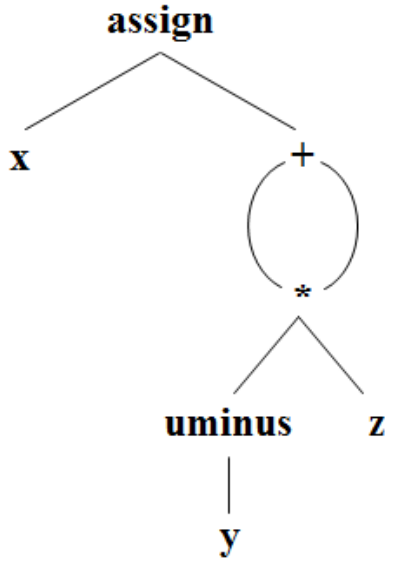
产生式	语义规则
S→id:=E	S.nptr=makenode(':=', makeleaf(id, id.entry), E.nptr)
$E \rightarrow E_1+T$	E.nptr=makenode('+', E_1.nptr, T.nptr)
E→T	E.nptr=T.nptr
$T \rightarrow T_1*F$	T.nptr=makenode('*', T_1.nptr, F.nptr)
T→F	T.nptr=F.nptr
F→(E)	F.nptr=E.nptr
F→uminus E	F.nptr=makeunode('uminus', E.nptr)
F→id	F.nptr=makeleaf(id, id.entry)
F→num	F.nptr=makeleaf(num, num.val)

赋值语句 x:=(-y)*z+(-y)*z 的图表示法

- 语法树表示：



- dag图形表示：



- 后缀式是语法树的线性表示形式。
- 对语法树进行深度优先遍历、访问子结点先于父结点、且从左向右访问子结点，得到一个包含所有树结点的序列，即后缀式。
- 在此序列中，每个树结点出现且仅出现一次；
  每个结点都是在它的所有子结点出现之后立即出现。
- 与上述语法树对应的后缀式是：
  x y uminus z * y uminus z * + assign。

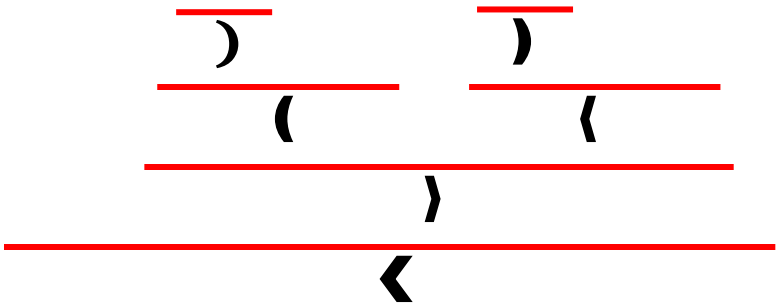
8.1.2 三地址代码

- 三地址代码：三地址语句组成的序列。
  - 类似于汇编语言的代码
  - 有赋值语句、控制语句
  - 语句可以有标号
- 三地址语句的一般形式： x:=y op z
  - x可以是名字、临时变量
  - y、z 可以是名字、常数、或临时变量
  - op 代表运算符号，如算数运算符、或逻辑运算符等
  - 语句中，最多有三个地址。
- 实现时，语句中的名字，将由指向该名字在符号表中表项的指针所代替。

三地址语句的种类及形式

- 简单赋值语句
  - x:=y op z
  - x:=op y
  - x:=y
- 含有变址的赋值语句
  - x:=y[i]
  - x[i]:=y
- 含有地址和指针的赋值语句
  - x:=&y
  - x:=*y
  - *x:=y
- 转移语句
  - goto L
  - if x relop y goto L
- 过程调用语句
  - param x
  - call p, n
- 返回语句
  - return y

赋值语句 x:=(-y)*z+(-y)*z 的三地址代码



- 对应语法树的代码

  t_1:=-y

  t_2:=t_1*z

  t_3:=-y

  t_4:=t_3*z

  t_5:=t_2+t_4

  x:=t_5

- 对应dag的代码

  t_1:=-y

  t_2:=t_1*z

  t_5:=t_2+t_2

  a:=t_5

三地址语句的实现——四元式

- 四元式

  （op，arg_1，arg_2，result）如：x:=y+z　('+', y, z, x)

  （op，arg_1，　，result）如：x:=-y　('uminus', y, , x)

  （param，arg_1，　，　）　如：param x　(param, x)

  （goto，　，　，语句标号）如：goto L　(goto, , , L)

- 赋值语句 x:=(-y)*z+(-y)*z 的四元式表示

	op	arg_1	arg_2	result
(0)	uminus	y		t_1
(1)	*	t_1	z	t_2
(2)	uminus	y		t_3
(3)	*	t_3	z	t_4
(4)	+	t_2	t_4	t_5
(5)	:=	t_5		x

三地址语句的实现——三元式

- 三元式：（op，arg_1，arg_2）
  - 为避免把临时变量名也存入符号表，不引入临时变量
  - 一个语句计算出来的中间结果直接提供给引用它的语句
  - 用计算中间结果的语句的指针代替存放中间结果的临时变量
- 赋值语句 x:=(-y)*z+(-y)*z 的三元式表示

语句序号	op	arg1	arg2
(0)	uminus	y	
(1)	*	(0)	z
(2)	uminus	y	
(3)	*	(2)	z
(4)	+	(1)	(3)
(5)	assign	x	(4)

语句x[i]:=y和x:=y[i]的三元式序列

- 语句 x[i]:=y

语句序号	op	arg1	arg2
(0)	[]=	x	i
(1)	assign	(0)	y

- 语句 x:=y[i]

语句序号	op	arg1	arg2
(0)	=[]	y	i
(1)	assign	x	(0)

8.2 赋值语句的翻译

- 假定赋值语句出现的环境可用下面的文法描述：

P→ M D; S

M→ ε

D→ D; D | D→ id: T | D→ proc id; N D; S

N→ ε

T→integer | real
| array [num] of T_1
| ↑T_1
| record L D end

N→ ε

S→ id:=E

E→ E+E | E*E | -E | (E) | id | num | num.num

设计函数：
(1) p=lookup(id.name)
(2) gettype(p)
(3) newtemp()
(4) outcode(s)

8.2.1 仅涉及简单变量的赋值语句

- 文法

$S \rightarrow id := E$

$E \rightarrow E_1 + E_2$

$E \rightarrow E_1 * E_2$

$E \rightarrow -E_1$

$E \rightarrow (E_1)$

$E \rightarrow id$

$E \rightarrow num$

$E \rightarrow num.num$

- 属性 E.entry：
  记录与E相应的临时变量在符号表中的表项位置

翻译方案8. 1

S→id:=E { p=lookup(id.name);
if (p!=nil) outcode(p ′:=′ E.entry);
else error(); }

E→E_1+E_2 {E.entry=newtemp();
outcode(E.entry ′:=′ E_1.entry ′+′ E_2.entry)}

E→E_1*E_2 {E.entry=newtemp();
outcode(E.entry ′:=′ E_1.entry ′*′ E_2.entry)}

E→-E_1 { E.entry=newtemp();
outcode(E.entry ′:=′ ′uminus′ E_1.entry) }

E→(E_1) { E.entry=E_1.entry }

E→id { p=lookup(id.name);
if (p!=nil) E.entry=p;
else error(); }

思考？
E→num ??
E→num.num ??

同时进行类型检查的翻译方案

- 假设，仅考虑类型 integer 和 real
- $E \rightarrow E_1 + E_2$ 的类型检查动作：

```
{ if (E1.type==integer) && (E2.type==integer)
      E.type=integer;
  else E.type =real; }
```

$E \rightarrow E_1+E_2$ 带有类型检查的语义动作

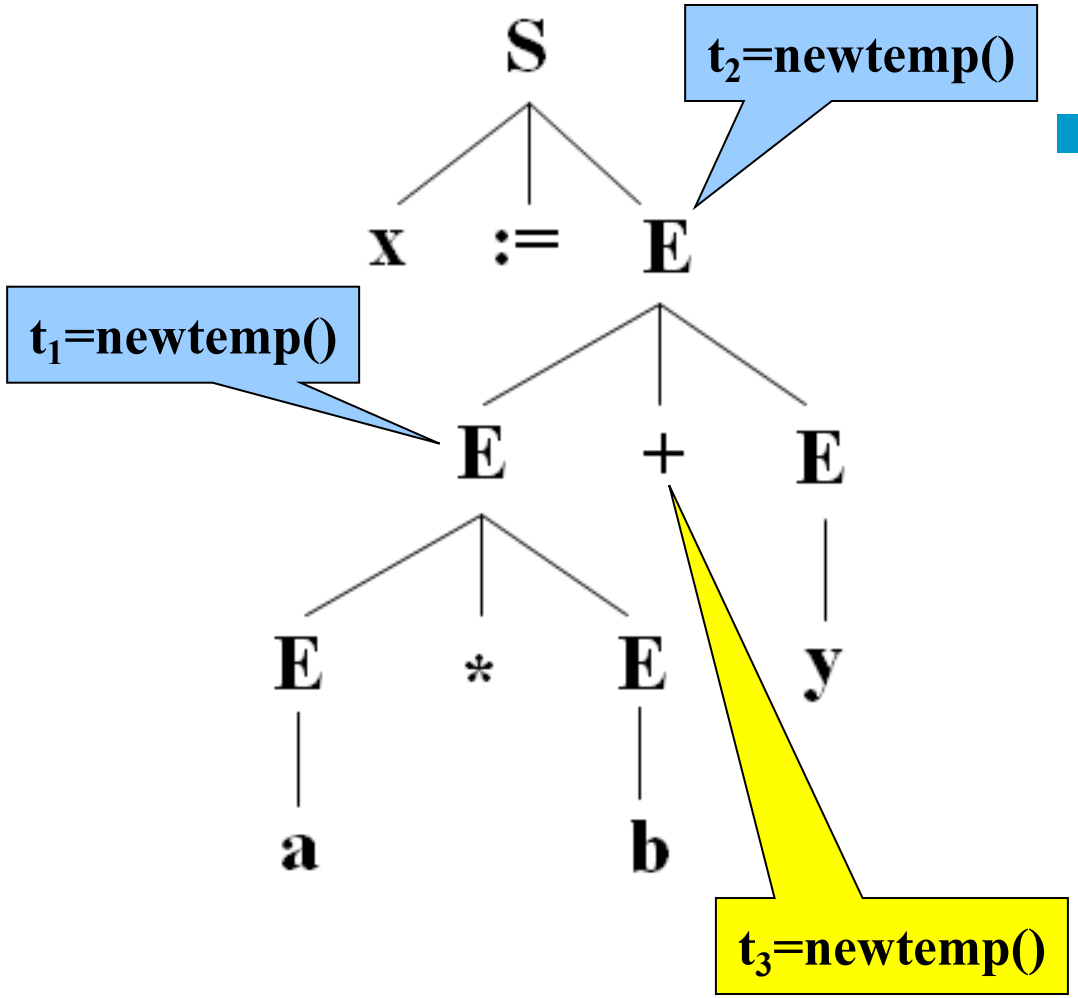
```
{ E.entry=newtemp();
  if (E1.type==integer) && (E2.type==integer) {
        outcode(E.entry ':=' E1.entry '+' E2.entry);
        E.type=integer; };
  else if (E1.type==real) && (E2.type==real) {
         outcode(E.entry ':=' E1.entry 'real+' E2.entry);
         E.type=real; };
  else if (E1.type==integer) && (E2.type==real) {
         u=newtemp();
         outcode(u ':=' 'inttoreal' E1.entry);
         outcode(E.entry ':=' u 'real+' E2.entry);
         E.type=real; };
  else if (E1.type==real) && (E2.type==integer) {
        u=newtemp();
        outcode(u ':=' 'inttoreal' E2.pace);
        outcode(E.entry ':=' E1.entry 'real+' u);
        E.type=real; };
  else E.type=type_error;   }
```

S→id:=E 带有类型检查的语义动作

```
p=lookup(id.name);
if  (p==nil) {
        error();
}else{
        t=gettype(p);
        if (t==E.type) {
                outcode(p ':=' E.entry);
                S.type=void;   };
        else if  (t==real) && (E.type==integer) {
                u=newtemp();
                outcode(u ':='  'inttoreal'  E.entry);
                outcode(p ':=' u);
                S.type=void;   }
        else  S.type=type_error;
  };
```

翻译赋值语句 x:=a*b+y

- 假定x和y的类型为real，a和b的类型为integer



- 三地址代码：

t_1:= a int* b

t_3:= inttoreal t_1

t_2:= t_3 real+ y

x:= t_2

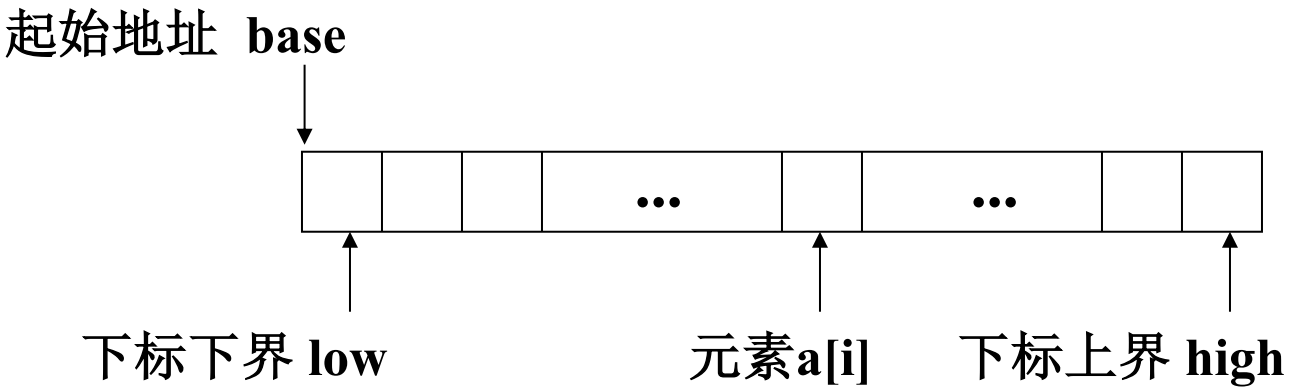
8.2.2 涉及数组元素的赋值语句

1.计算数组元素的地址

- 数组元素存储在一个连续的存储块中，根据数组元素的下标可以快速地查找每个元素。
- 数组空间起始地址：base
- 每个元素的域宽：w

- 一维数组 A[i]
- 二维数组 A[i, j]
- k 维数组 $A[i_1, i_2, \ldots, i_k]$

一维数组--A[i]的地址



数组元素个数：high-low+1

数组元素A[i]的位置：

$$base + (i - low) \times w$$

$$= i \times w + base - low \times w \quad \text{常数C}$$

二维数组--A[i, j]的地址

- 二维数组 A[m, n]

$$\begin{matrix} a_{11} & a_{12} & \cdots & a_{1j} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2j} & \cdots & a_{2n} \\ \cdots & & & & & \\ a_{i1} & a_{i2} & \cdots & a_{ij} & \cdots & a_{in} \\ \cdots & & & & & \\ a_{m1} & a_{m2} & \cdots & a_{mj} & \cdots & a_{mn} \end{matrix}$$

存储方式：

按行优先存放

按列优先存放

每维的下界：low_1、low_2

每维的上界：$high_1$、$high_2$

每维的长度：$m=high_1-low_1+1$

$n=high_2-low_2+1$

数组元素A[i, j]的位置：

$$base + ((i-low_1) \times n + (j-low_2)) \times w$$

$$= (i \times n + j) \times w + base - (low_1 \times n + low_2) \times w$$

常数C

k维数组--A[i_1, i_2, ..., i_k]的地址

每维的下界：low_1、low_2、...、low_k

每维的长度：n_1、n_2、...、n_k

存储方式：按行存放

数组元素A[i_1, i_2, ..., i_k]的位置：

$$((\ldots((i_1\times n_2+i_2)\times n_3+i_3)\ldots)\times n_k+i_k)\times w$$

$$+\ base - ((\ldots((low_1\times n_2+low_2)\times n_3+low_3)\ldots)\times n_k+low_k)\times w$$

递归计算：

$e_1=i_1$

$e_2=e_1\times n_2+i_2$

$e_3=e_2\times n_3+i_3$

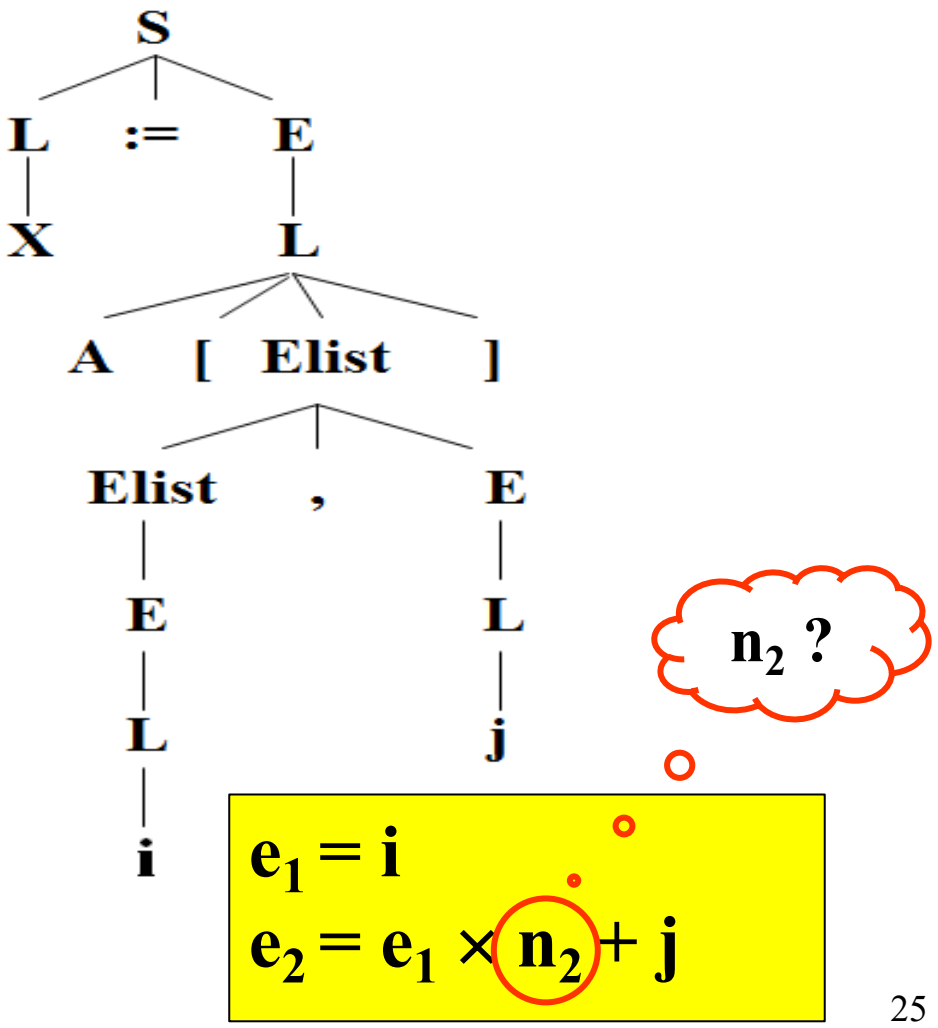
...

$e_k=e_{k-1}\times n_k+i_k$

2. 涉及数组元素的赋值语句的翻译——L属性定义

- 赋值语句的文法：
  - (1) S→L:=E
  - (2) L→id
  - (3) L→id [Elist]
  - (4) Elist→E
  - (5) Elist→Elist$_1$, E
  - (6) E→E$_1$+E$_2$
  - (7) E→(E$_1$)
  - (8) E→L

语句 X:=A [i, j] 的分析树



属性及函数设计

- L 综合属性 L.entry 和 L.offset（符号表入口指针）
  - 简单变量：

    L.offset=null　　L.entry=变量在符号表中的入口指针
  - 数组元素：（临时变量在符号表中的入口指针）

    L.offset=计算公式第一项 ($e_m \times w$)　L.entry=计算公式第二项 (base-C)
- E 综合属性E.entry，保存E值的变量在符号表中的位置
- Elist 继承属性array， 综合属性ndim，entry
  - Elist.array：数组名在符号表中的位置
  - Elist.ndim：目前已经识别出的下标表达式的个数
  - Elist.entry：保存递推公式中e_m值的临时变量在符号表中的位置
- 函数
  - getaddr(array)：根据 指针array访问符号表，返回该表项中存放的数组空间的起始位置 base。
  - limit(array, j)：返回array指向的数组的第 j 维的长度。
  - invariant(array)：返回array指向的数组的地址计算公式中的常量C。

翻译方案8.2（--L属性定义）

S→L:=E { if (L.offset==null) /* L是简单变量 */
outcode(L.entry ':=' E.entry);
else outcode(L.entry '[' L.offset ']' ':=' E.entry); }

L→id { L.entry=id.entry; L.offset=null; }

L→id[{ Elist.array=id.entry; }
Elist] { L.entry=newtemp();
outcode(L.entry ':=' getaddr(Elist.array) '-'
invariant(Elist.array));
L.offset=newtemp();
outcode(L.offset ':=' w '×' Elist.entry); }

数组元素A[i, j]的位置：
base + ((i-low_1) × n + (j-low_2)) × w
= (i×n+j)×w + base - (low_1×n+low_2)×w

Elist→E { Elist.entry=E.entry;
Elist.ndim=1; }

$e_1=i_1$

翻译方案8.2（续）

$e_2=e_1\times n_2+i_2$
$e_3=e_2\times n_3+i_3$
…
$e_k=e_{k-1}\times n_k+i_k$

Elist→ { $Elist_1$.array=Elist.array; }

$Elist_1$,E { t=newtemp(); m=$Elist_1$.ndim+1;
outcode(t ′:=′ $Elist_1$.entry ′×′ limit($Elist_1$.array,m));
outcode(t ′:=′ t ′+′ E.entry);
Elist.entry=t;
Elist.ndim=m; }

E→E_1+E_2 { E.entry=newtemp();
outcode(E.entry ′:=′ E_1.entry ′+′ E_2.entry) }

E→(E_1) { E.entry=E_1.entry }

E→L { if (L.offset == null) E.entry=L.entry;
else { E.entry=newtemp();
outcode(E.entry ′:=′ L.entry ′[′ L.offset ′]′);
}
}

示例：翻译语句 x:=A[y, z]

已知：

A是一个10×20的数组，即 $n_1=10$，$n_2=20$；

设数组元素的域宽 $w=4$；

设数组的第一个元素为：A[1,1]，

则有 $low_1=1$，$low_2=1$

所以：

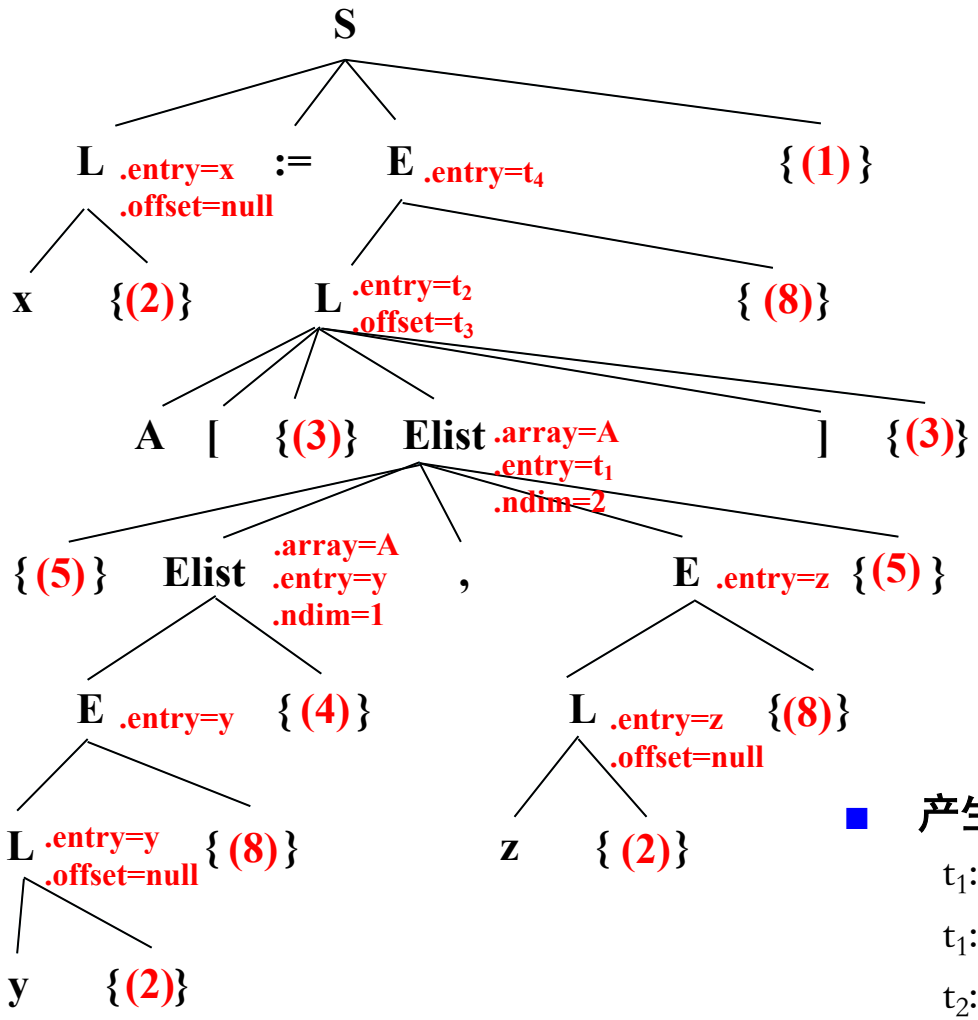
$$(low_1 \times n_2 + low_2) \times w = (1 \times 20 + 1) \times 4 = 84$$

要求：

将赋值语句 x:=A[y,z] 翻译为三地址代码。

赋值语句 x:=A[y, z]的翻译

A是一个10*20的整型数组



(1) $S \to L:=E$ {if ($L.offset==$null) /*L是简单变量*/
outcode($L.entry$':='$E.entry$);
else outcode($L.entry$'['$L.offset$']':='$E.entry$);}

(2) $L \to$ id{ $L.entry=$id.$entry$; $L.offset=$null; }

(3) $L \to$ id [{$Elist.array=$id.$entry$;} /*继承属性*/
$Elist$]{$L.entry=newtemp()$;
outcode($L.entry$':='getaddr($Elist.array$)'-'
invariant($Elist.array$));
$L.offset=newtemp()$;
outcode($L.offset$':='w'×'$Elist.entry$);}

(4) $Elist \to E${$Elist.entry=E.entry$;$Elist.ndim=1$;}

(5) $Elist \to$ { $Elist_1.array=Elist.array$; } /*继承属性*/
$Elist_1$,E{$t=newtemp()$;
$m=Elist_1.ndim+1$;
outcode(t':='$Elist_1.entry$'×'limit($Elist.array$,m));
outcode(t':='t'+'$E.entry$);
$Elist.entry=t$;
$Elist.ndim=m$ }

(6) $E \to E_1+E_2${$E.entry=newtemp()$;
outcode($E.entry$':='$E_1.entry$'+'$E_2.entry$);}

(7) $E \to (E_1)${$E.entry=E_1.entry$}

(8) $E \to L$ {if ($L.offset==$null) /*L是简单变量*/
$E.entry=L.entry$;
else{$E.entry=newtemp()$;
outcode($E.entry$':='$L.entry$'['$L.offset$']'); }
}

(翻译方案8

- **产生三地址代码：**

t_1:=y×20 limit(A, 2)=20

t_1:=t_1+z

t_2:=A-84 invariant(A) = (low_1*n+low_2)*w
= (1*20+1)*4
= 84

t_3:=4×t_1

t_4:=t_2[t_3]

x:=t_4

涉及数组元素的赋值语句的翻译
——S属性定义

- 赋值语句的文法：
  - (1) S→L:=E
  - (2) L→id
  - (3) L→id [Elist]
  - (4) Elist→E
  - (5) Elist→Elist$_1$, E
  - (6) E→E$_1$+E$_2$
  - (7) E→(E$_1$)
  - (8) E→L

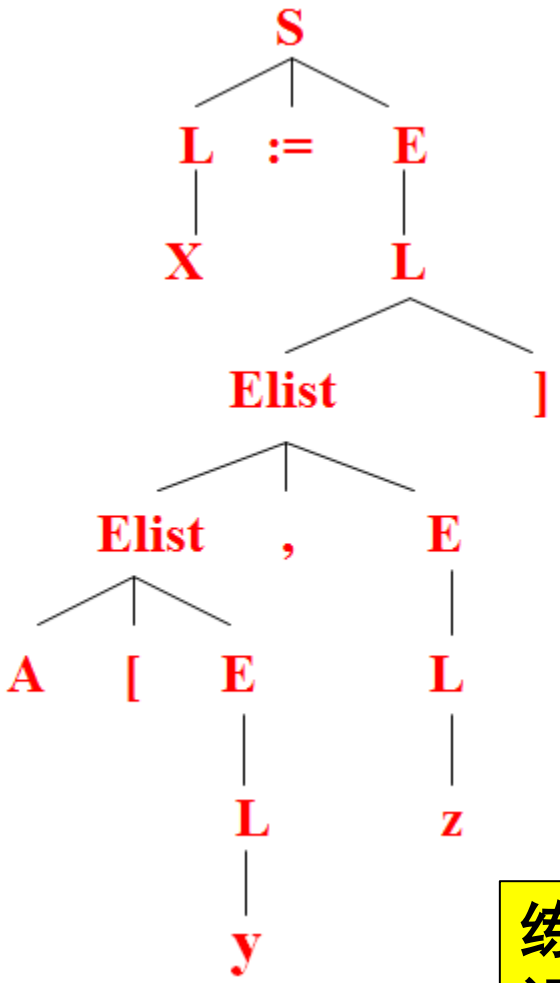
改写文法：

(3) L→ Elist]

(4) Elist→ id[E

(5) Elist→Elist$_1$, E

语句 X:=A [y, z]的分析树



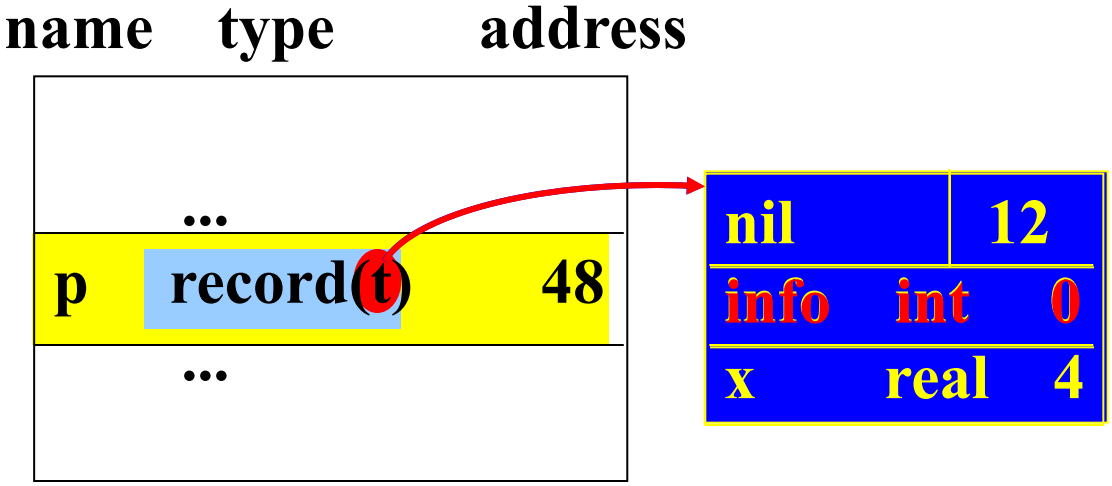
L.entry
L.offset
E.entry
Elist.array
Elist.ndim
Elist.entry

练习：
设计翻译方案

3. 记录结构中域的访问

- 声明：

```
p: record
     info: integer;
     x: real
   end;
```



- 引用

```
p.info:=p.info+1;
```

- 编译器的动作

  lookup(p)

  Gettype

  根据t，找到记录的符号表

  根据info在表中找

访问记录域的翻译动作

$L \rightarrow L_1.L_2$ {

```
    L.entry=newtemp( );
    if (L1.offset==null)
        L.entry= L1.entry;
    else
        outcode( L.entry ':=' L1.entry '[' L1.offset ']' );
    L.offset=newtemp( );
    if (L2.offset==null)
        L.offset= L2.entry;
    else
        outcode( L.offset ':=' L2.entry '[' L2.offset ']' );
}
```

练习

- 翻译语句：

  x:=A[i, j].info

其中，A是一个10*20的Element结构体数组；
Element结构体的成员有：info和x

8.3 布尔表达式的翻译

- 布尔表达式的作用
  - 计算逻辑值
  - 用作控制语句中的条件表达式
- 产生布尔表达式的文法

  E→ E or E

  E→ E and E

  E→ not E

  E→ (E)

  E→ id relop id

  E→ true

  E→ false

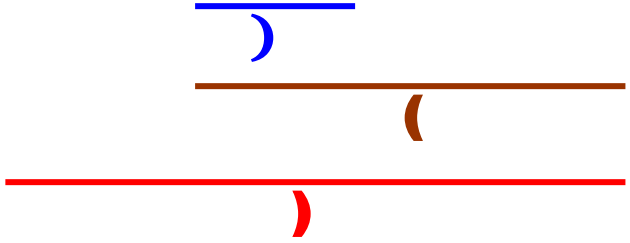
8.3.1 翻译布尔表达式的方法

- 布尔表达式的真值的表示方法
  - 数值表示法：
    - 1 — true　　0 — false
    - 非0 — true　　0 — false
  - 控制流表示法：

    利用控制流到达程序中的位置来表示 true 或 false
- 布尔运算符的短路运算
  - 短路运算，如C、C++、java支持，Pascal不支持
  - Ada语言，非短路运算符：and，or
    短路运算符：and then，or else
- 布尔表达式的翻译方法
  - 数值表示法
  - 控制流表示法

8.3.2 数值表示法

- 布尔表达式的求值类似于算术表达式的求值
- 例如： a or not b and c



- 三地址代码

  t_1:=not b

  t_2:= t_1 and c

  t_3:=a or t_2

- 关系表达式 x>y

  等价于：

  if x>y

  then 1

  else 0

- x>y 的三地址代码：

  100： if x>y goto 103

  101： t:=0

  102： goto 104

  103： t:=1

  104：

语义动作中变量、属性及函数说明

- 变量nextstat：写指针，指示输出序列中下一条三地址语句的位置。
- 属性E.entry：存放布尔表达式E的真值的临时变量在符号表中的入口位置。
- 函数outcode(s)：根据nextstat的指示将三地址语句写到输出序列中。
  - outcode(s)输出一条三地址语句之后， nextstat自动加1。

数值表示法翻译方案

$E \rightarrow E_1$ or E_2 { E.entry=newtemp();
outcode(E.entry ':=' E_1.entry 'or' E_2.entry); }

$E \rightarrow E_1$ and E_2 { E.entry=newtemp();
outcode(E.entry ':=' E_1.entry 'and' E_2.entry); }

$E \rightarrow$ not E_1 { E.entry=newtemp();
outcode(E.entry ':=' 'not' E_1.entry); }

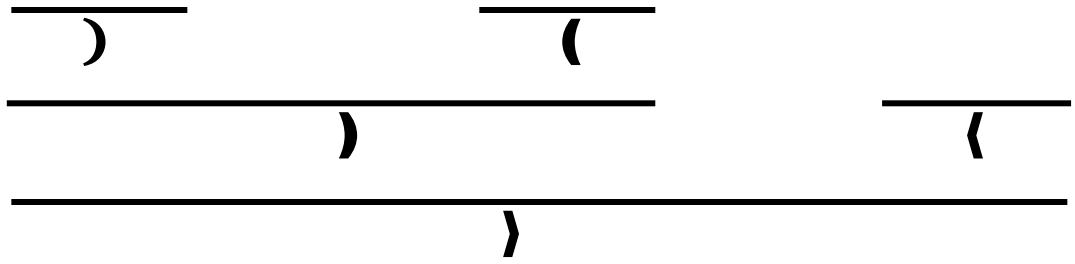
$E \rightarrow (E_1)$ { E.entry=E_1.entry); }

$E \rightarrow id_1$ relop id_2 { E.entry=newtemp();
outcode('if' id_1.entry relop.op id_2.entry 'goto' nextstat+3);
outcode(E.entry ':=' '0');
outcode('goto' nextstat+2);
outcode(E.entry ':=' '1'); }

E→ true { E.entry=newtemp(); outcode(E.entry ':=' '1') ; }

E→ false { E.entry=newtemp(); outcode(E.entry ':=' '0') ; }

举例：a>b and c>d or e<f



100: if a>b goto 103

101: t_1:=0

102: goto 104

103: t_1:=1

104: if c>d goto 107

105: t_2:=0

106: goto 108

107: t_2:=1

108: t_3:=t_1 and t_2

109: if e<f goto 112

110: t_4:=0

111: goto 113

112: t_4:=1

113: t_5:=t_3 or t_4

8.3.3 控制流表示法及回填技术

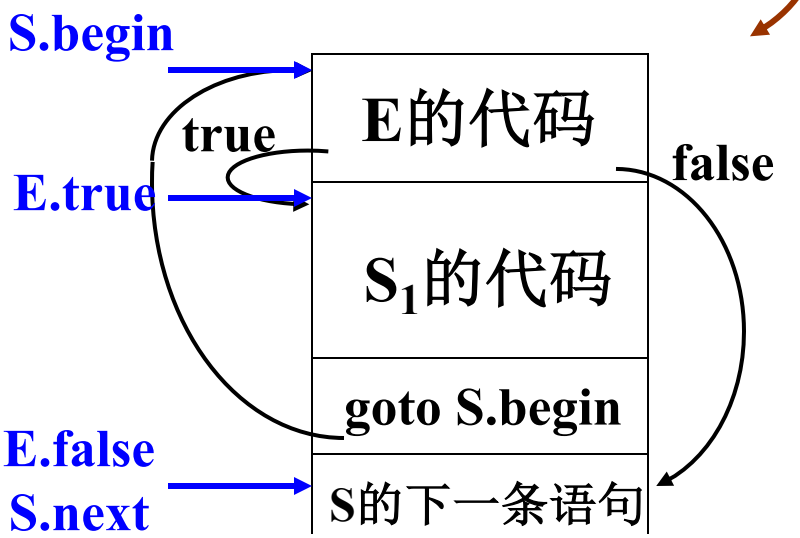
- 控制语句

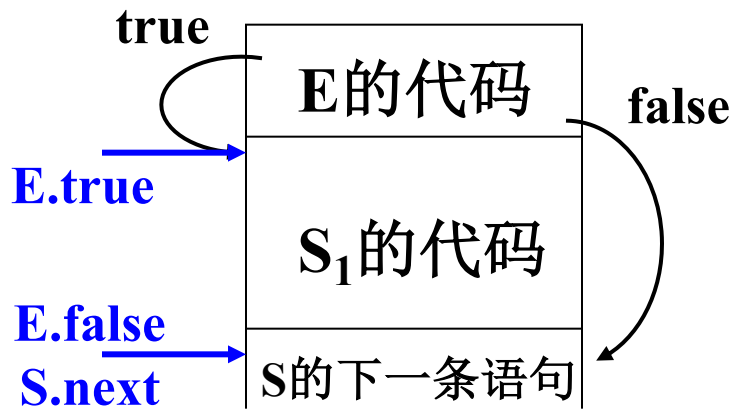
  S→ if E then S_1

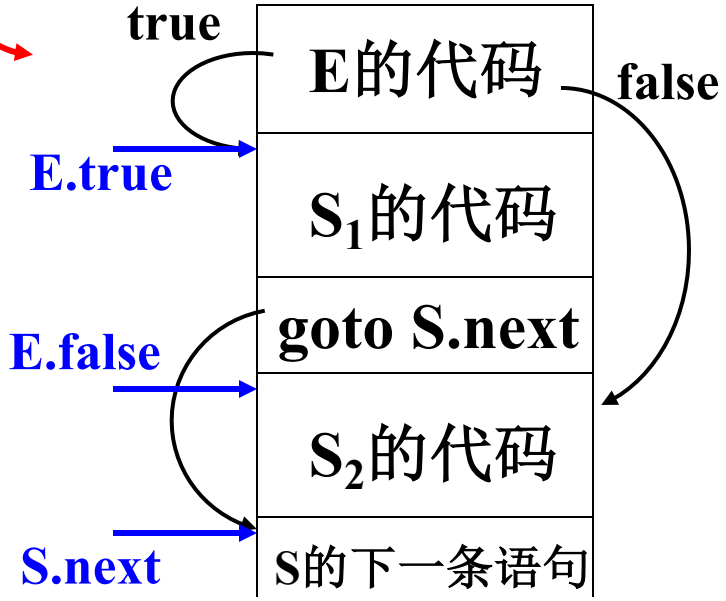
  | if E then S_1 else S_2

  | while E do S_1

- 控制语句的代码结构







变量、属性及函数说明

- 函数newlable()：产生并返回一个新的语句标号。
- 继承属性：三地址语句标号
  - E.true：E的值为真时应执行的第一条语句的标号
  - E.false：E的值为假时应执行的第一条语句的标号
  - S.next：紧跟在语句S之后的下一条三地址语句的标号
  - S.begin：语句S的第一条三地址语句的标号

控制流表示法翻译布尔表达式

- 布尔表达式被翻译为一系列条件转移和无条件转移三地址语句
- 这些语句转移到的位置是E.true、E.false之一
- 例如 a<b 翻译为：

  if a<b goto E.true

  goto E.false
- 属性说明
  - 继承属性

    E.true：E为真时转移到的三地址语句的标号

    E.false： E为假时转移到的三地址语句的标号

控制流翻译方法的基本思想

- 条件表达式 x>y 翻译为：

  if x>y goto E.true

  goto E.false

- $E \rightarrow id_1$ relop id_2

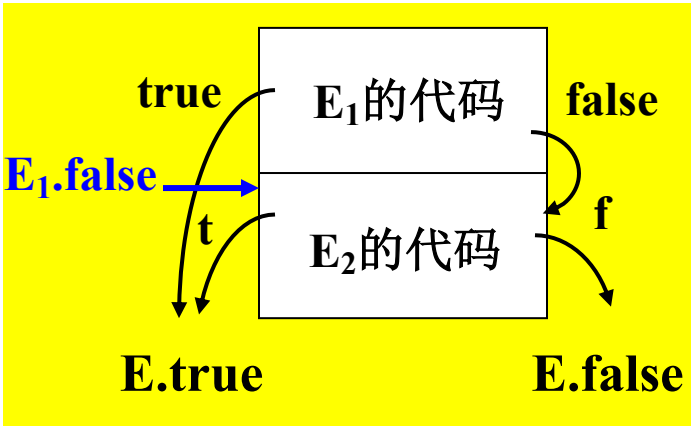
  'if' id_1.entry relop.op id_2.entry 'goto' E.true

  'goto' E.false

- 将布尔表达式E翻译为一系列条件转移和无条件转移三地址语句。
  - 转移语句转移到的位置是 E.true 或者 E.false
  - E的值为真或为假时，控制转移到的位置

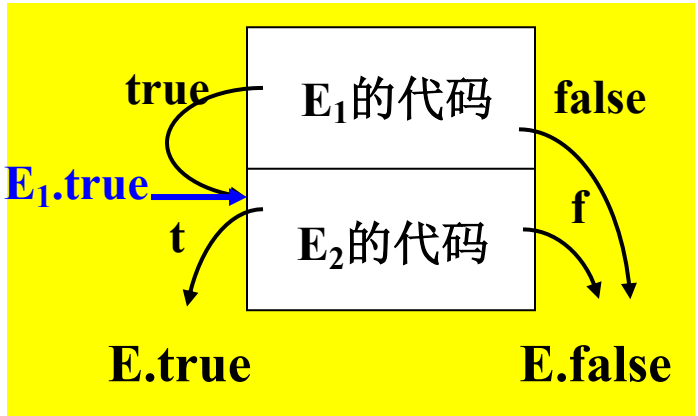
布尔表达式的代码结构（短路运算）

$E \rightarrow E_1$ or E_2



$E \rightarrow$ not E_1

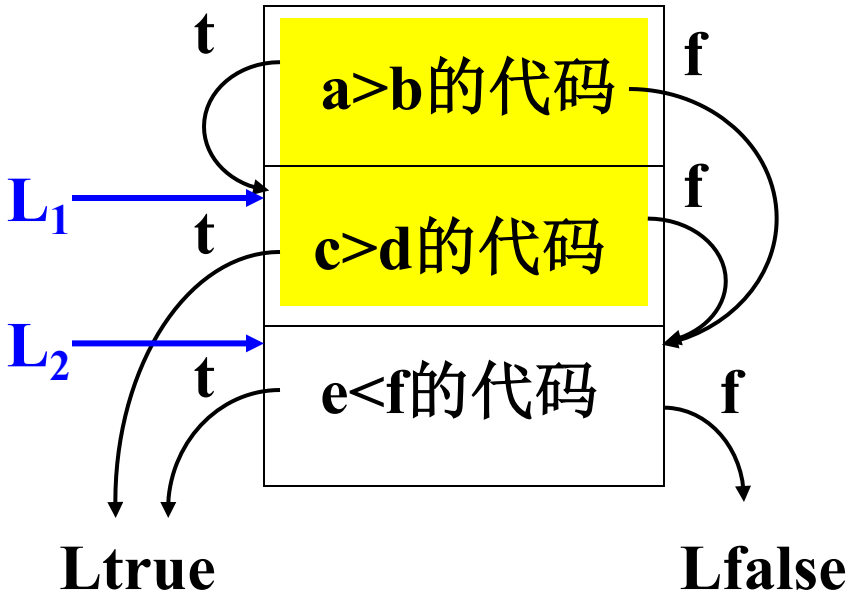
$E \rightarrow E_1$ and E_2



$E \rightarrow id_1$ relop id_2

'if' id_1.entry relop.op id_2.entry 'goto' E.true
'goto' E.false

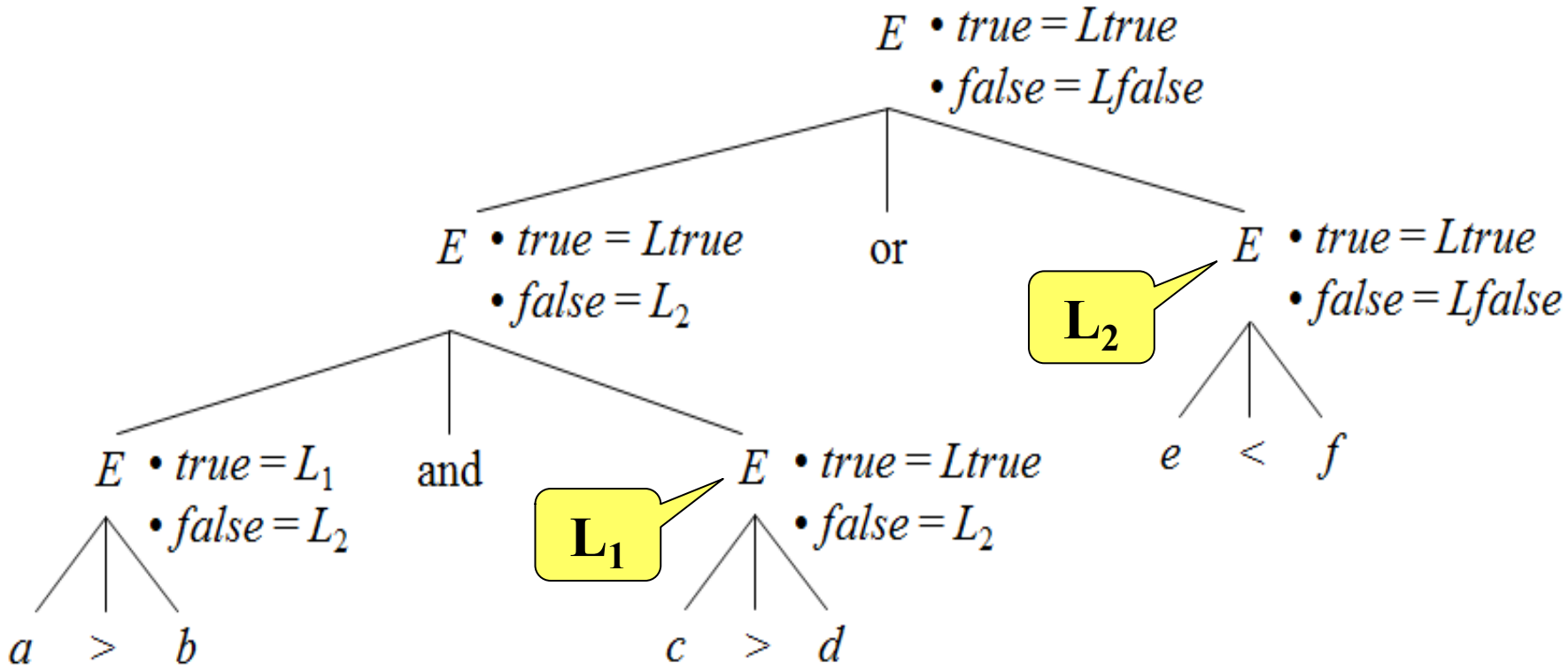
例：a>b and c>d or e<f 的代码结构及三地址语句



if a>b goto L_1
goto L_2
L_1: if c>d goto Ltrue
goto L_2
L_2: if e<f goto Ltrue
goto Lfalse

用控制流表示法翻译布尔表达式

a>b and c>d or e<f



控制流表示法翻译布尔表达式

- 布尔表达式的真假出口位置不但与表达式本身的结构有关，还与表达式出现的上下文有关。
- 考虑表达式“a>b or c>d”和“a>b and c>d”，“a>b”的真假出口依赖于：
  - 布尔表达式的结构
  - 布尔表达式所在控制语句的结构
- 两遍扫描的翻译技术

  Pass 1. 生成分析树

  Pass 2. 为分析树加注释——翻译
- 可否在一遍扫描过程中，同时完成分析和翻译？

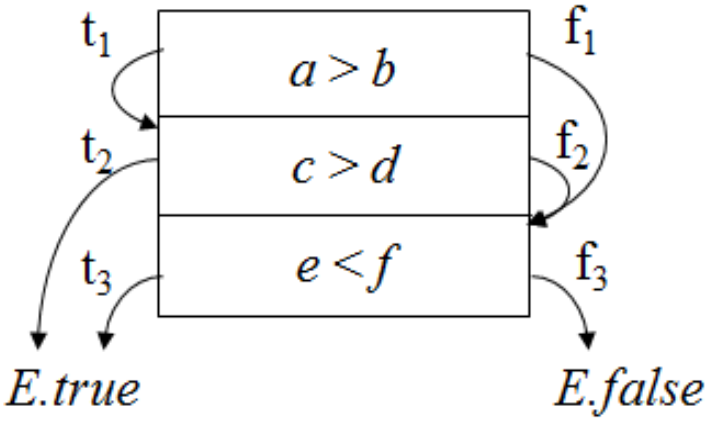
  问题：当生成某些转移指令时，目标地址可能还不知道。

控制流表示法翻译布尔表达式——回填技术

- 先产生没有填写目标标号的转移指令；
- 建立一个链表，把转向这个目标的所有转移指令的标号填入该链表；
- 目标地址确定后，再把目标地址填入该链表中记录的所有转移指令中。

例：用回填技术翻译 a>b and c>d or e<f

$$\frac{\underset{(1)}{a>b}\ \text{and}\ \underset{(2)}{c>d}}{(3)}\ \text{or}\ \underset{(4)}{e<f}\ \Big/\ (5)$$



.t={102}
.f={101, 103}

a>b and c>d

.t={100}
.f={101}

100: if a>b goto 102
101: goto 104

.t={102}
.f={103}

102: if c>d goto t_2
103: goto 104

.t={104}
.f={105}

104: if e<f goto t_3
105: goto f_3

a>b and c>d or e<f

.t={102, 104}
.f={105}

利用回填技术翻译布尔表达式

- 布尔表达式文法

$E \rightarrow E_1$ or M E_2

$E \rightarrow E_1$ and M E_2

$E \rightarrow$ not E_1

$E \rightarrow (E_1)$

$E \rightarrow id_1$ relop id_2

$E \rightarrow$ true

$E \rightarrow$ false

$M \rightarrow \varepsilon$

- 说明
  - 三地址语句用四元式表示
  - 四元式存放在数组中
  - 数组下标：三地址语句的标号
- 变量nextquad：记录将要产生的下一条三地址语句在四元式数组中的位置
- 标记非终结符号M
  - 标识E_2的开始位置
  - 属性M.quad，记录E_2的第一条三地址语句的地址
  - $M \rightarrow \varepsilon$ 的动作：
    M.quad=nextquad

属性定义及函数说明

- 综合属性
  - E.truelist：记录转移到E的真出口的指令链表的指针
  - E.falselist：记录转移到E的假出口的指令链表的指针
  - M.quad：M所标识的三地址语句的地址
- 函数
  - makelist(i)：建立新链表，其中只包括待回填的转移指令在数组中的位置 i，返回所建链表的指针。
  - merge(p_1,p_2)：合并由指针p_1和p_2所指向的两个链表，返回结果链表的指针。
  - backpatch(p,i)：用目标地址 i 回填 p所指链表中记录的每一条转移指令。
  - outcode(S)：产生一条三地址语句S，并写入输出数组中，该函数执行完后，变量 nextquad 加 1。

布尔表达式的翻译方案

$E \rightarrow E_1$ or ME_2 { backpatch(E_1.falselist, M.quad);
E.truelist= merge(E_1.truelist, E_2.truelist);
E.falselist=E_2.falselist; }

$E \rightarrow E_1$ and ME_2 { backpatch(E_1.truelist, M.quad);
E.truelist=E_2.truelist;
E.falselist= merge(E_1,falselist, E_2.falselist); }

$E \rightarrow$ not E_1 { E.truelist=E_1.falselist; E.falselist=E_1.truelist; }

$E \rightarrow (E_1)$ { E.truelist=E_1.truelist; E.falselist=E_1.falselist; }

$E \rightarrow id_1$ relop id_2 { E.truelist=makelist(nextquad);
E.falselist=makelist(nextquad+1);
outcode('if' id_1.entry relop.op id_2.entry 'goto -');
outcode('goto —'); }

$E \rightarrow$ true { E.truelist=makelist(nextquad); outcode('goto -'); }

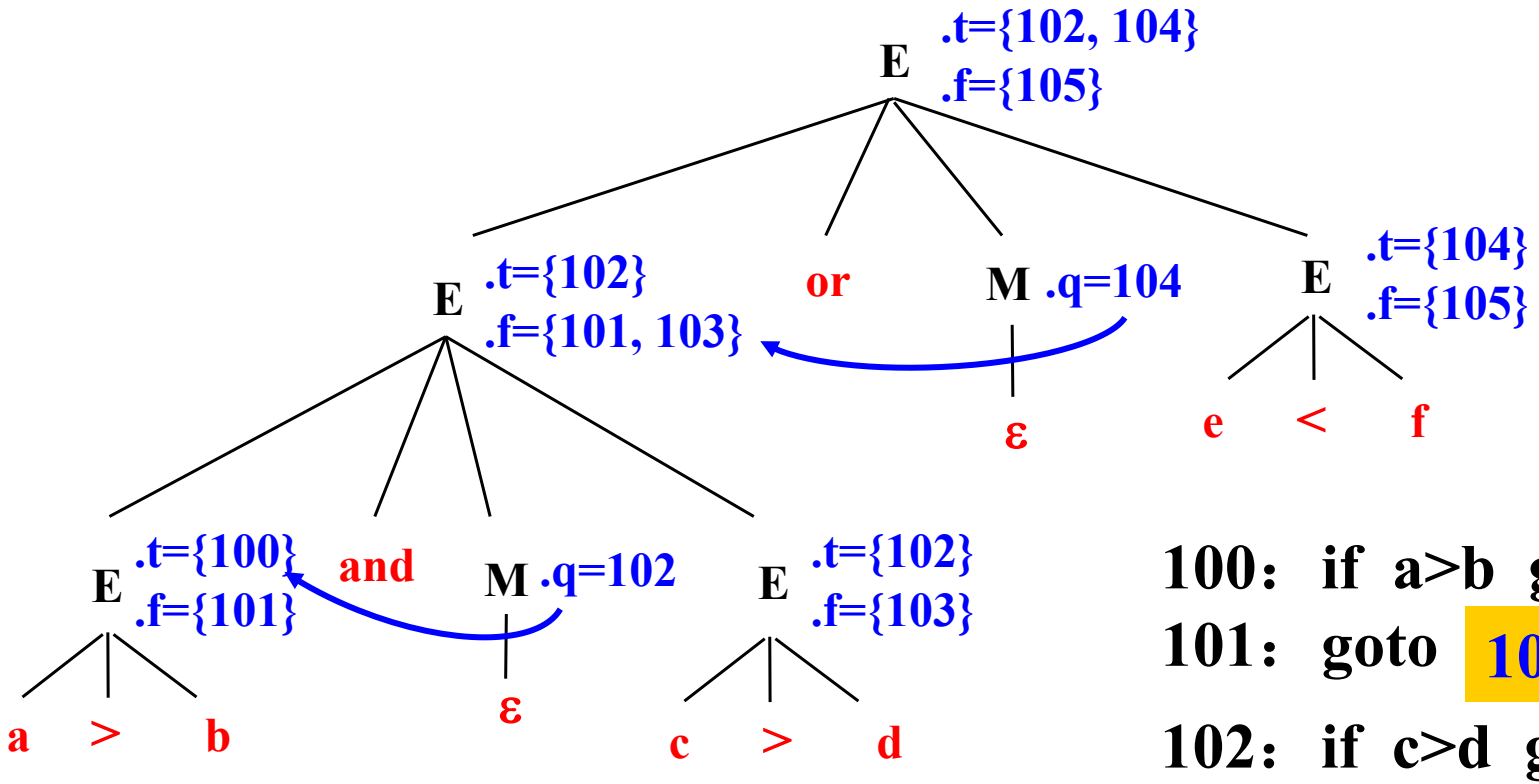
$E \rightarrow$ false { E.flaselist=makelist(nextquad); outcode('goto -'); }

$M \rightarrow \varepsilon$ { M.quad=nextquad; }

利用翻译方案翻译布尔表达式

a>b and c>d or e<f

假定**nextquad**的当前值为**100**



100：if a>b goto 102
101：goto 104
102：if c>d goto —
103：goto 104
104：if e<f goto —
105：goto —

8.4 控制语句的翻译

- 文法

S→if E then M S_1

S→if E then M_1 S_1 N else M_2 S_2

S→while M_1 E do M_2 S_1

S→begin Slist end

S→A

Slist→$Slist_1$; M S

Slist→S

M→ε

N→ε

转移到下一条语句的指令链表的指针

属性：

E.truelist

E.falselist

M.quad

S.nextlist

Slist.nextlist

N.nextlist

变量：nextquad

函数：

makelist(i)

backpatch(p, i)

merge(p_1, p_2)

outcode(s)

▲ 记录变量 nextquad 的当前，以便回填转移到此的指令

◆ 产生一条不完整的goto指令，并记录下它的位置

控制语句的翻译方案

S→if E then M S_1 { backpatch(E.truelist, M.quad);
S.nextlist=merge(E.falselist, S_1.nextlist); }

S→if E then M_1 S_1 N else M_2 S_2
{ backpatch(E.truelist, M_1.quad);
backpatch(E.falselist, M_2.quad);
S.nextlist=merge(S_1.nextlist, N.nextlist, S_2.nextlist) ; }

M→ε { M.quad=nextquad }

N→ε { N.nextlist=makelist(nextquad); outcode('goto —'); }

S→while M_1 E do M_2 S_1 { backpatch(S_1.nextlist, M_1.quad);
backpatch(E.truelist, M_2.quad);
S.nextlist=E.falselist;
outcode('goto' M_1.quad); }

S→begin Slist end { S.nextlist=Slist.nextlist ; }

S→A { S.nextlist=makelist() ; }

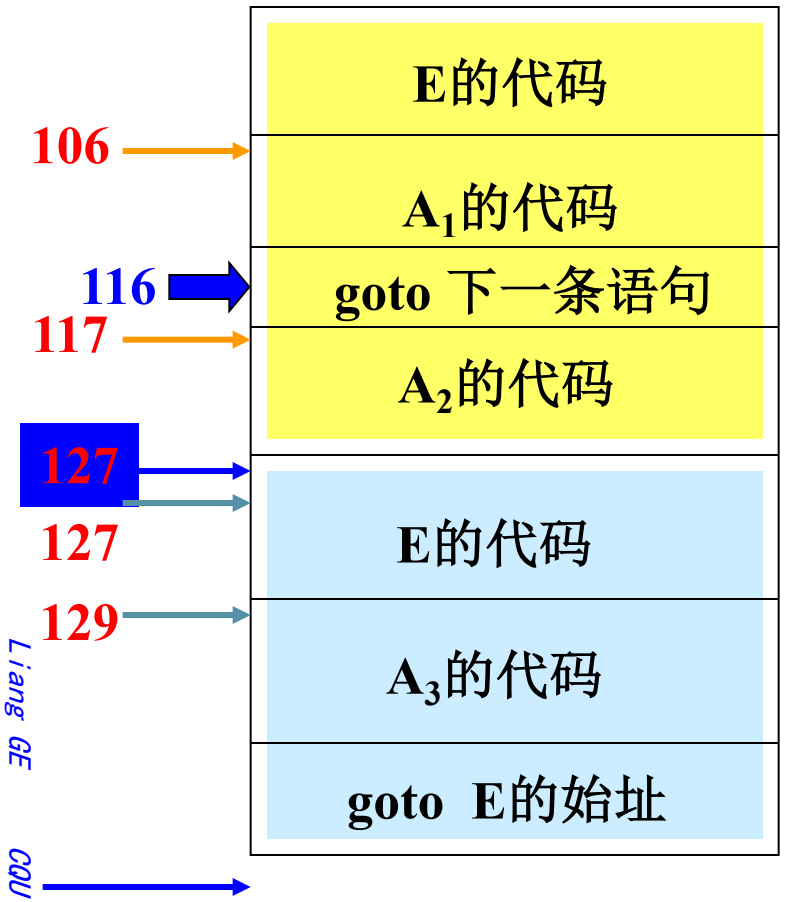
Slist → Slist $_1$; M S { backpatch(Slist$_1$.nextlist, M.quad);
Slist.nextlist=S.nextlist }

Slist →S {Slist.nextlist=S.nextlist }

例：

if a>b and c>d or e<f then 106 A_1 else 116 117 A_2;

while 127 a<b 127 do 129 A_3



.t={102,104}
.f={105}

100: if a>b goto 102
101: goto 104
102: if c>d goto 106
103: goto 104
104: if e<f goto 106
105: goto 117
106:
…
115: A_1的代码

.n={116}

116: goto 127
117:
…
126: A_2的代码

.t={127}
.f={128}

127: if a<b goto 129
128: goto —
129:
…
138: A_3的代码
139: goto 127

8.5 goto语句的翻译

- goto 语句的一般形式
  - goto lable
  - if expr goto lable
- 语句标号的出现形式
  - 定义性出现，形式为 lable：stmt
  - 引用性出现，作为转移目标出现在 goto语句中
- 程序中应用形式：
- 标号的声明
  - Pascal要求使用前先声明
  - C语言，不要求
- 符号表中的语句标号

先定义后引用：	先引用后定义：
lable: stme; … goto lable	goto lable; … lable: stme;

名字	类型	定义标志	地址
L	Label	F	-1

名字	类型	定义标志	地址
L	Label	T	*V*

标号引用性出现时的处理（goto L;）

- 根据标识符*L*在符号表中进行查找。
- 对于要求语句标号先声明后使用的语言（如Pascal）
  - 若未找到，则报告“标号未定义”的错误；
  - 若找到，则检查其类型是否为标号
    - 若不是，则报告类型错误；
    - 若是，则进一步检查该标号的“定义标志”
      - 若“定义标志”是‘T’，说明之前已识别出标号*L*的定义，其“地址”域中记录的是它所标识的语句的第一条三地址语句的位置*V*，此时直接生成四元式（goto, —, —, *V*）即可；
      - 若“定义标志”是‘F’，说明标号*L*在程序体中还未定义，则生成待回填的goto语句，并将它插入与该目标地址相关的语句链的链首。

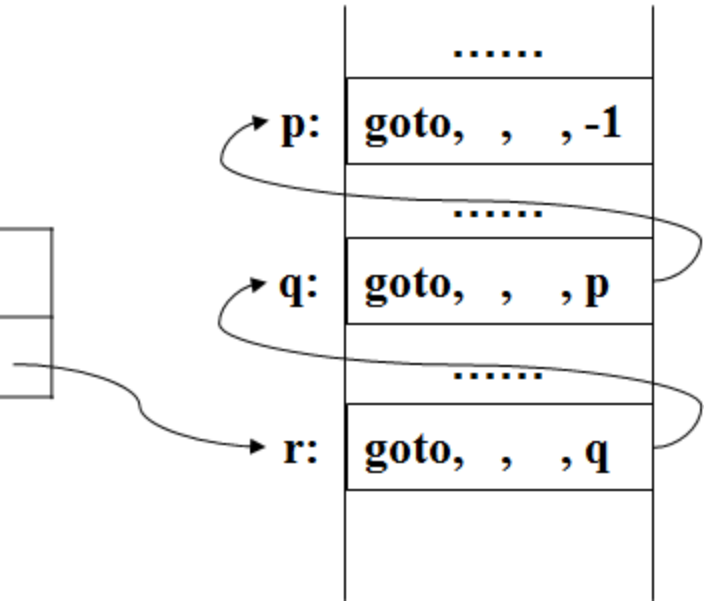
标号引用性出现时的处理（续1）

- 对于不要求语句标号先声明的语言（如C）
  - 若未找到，则先将标号L插入符号表中
    - “定义标志”设置为‘F’，表示“标号未定义”；
    - 将全局变量 nextquad 的值写入“地址”域；
    - 生成四元式：（goto，-，-，-1）。
  - 若找到，则进一步检查该标号的“定义标志”
    - 若“定义标志”是‘T’，说明之前已识别出标号*L*的定义，其“地址”域中记录的是它所标识的语句的第一条三地址语句的位置*V*，此时直接生成四元式（goto, —, —, *V*）即可；
    - 若“定义标志”是‘F’，说明标号*L*在程序体中还未定义，则生成待回填的goto语句，并将它插入与该目标地址相关的语句链的链首。

标号引用性出现时的处理（续2）

- 待回填的语句链

名字	类型	定义标志	地址
L	Label	F	*r*



标号定义性出现时的处理（L：S）

- 根据标号L查找符号表；
- 对于要求语句标号先声明后使用的语言（如Pascal）
  - 若未找到，则报告“标号未定义”的错误；
  - 若找到，则检查其类型是否为“标号”
    - 若不是，则报告类型错误；
    - 若是，则进一步检查其“定义标志”。
      - “定义标志”是‘T’，则报告“标号重复定义”的错误；
      - 若“定义标志”是‘F’，则将“定义标志”改为‘T’，判断地址域是否为空
        - 若为空，说明标号L是首次出现，并且是定义性出现，则将全程变量nextquad的值V（即语句S的第一条三地址语句在四元式数组中的位置）写入地址域中。
        - 若不空，说明之前已经有标号L的引用性出现，存在待回填语句链，此时，编译程序首先将nextquad的值V回填到该链表中记录的所有语句中，然后再将V写入L的“地址”域中。

标号定义性出现时的处理（续1）

- 对于不要求语句标号先声明后使用的语言（如C）
  - 若未找到，说明标号L是首次出现，并且是定义性出现，则将L插入符号表中，
    - 设置其类型为Label
    - “定义标志”为‘T’
    - 将全程变量nextquad的值V写入地址域中。

名字	类型	定义标志	地址
L	Label	T	*V*

  - 若找到，则处理过程与上述Pascal编译程序的一样。

其他语句

- break
  - 用于强行退出当前结构，不再执行结构体中剩余的语句；
  - 控制直接转移到当前结构的下一条语句的位置。
- continue
  - 用于停止执行当前的循环；
  - 控制转移到循环起始处，开始下一次循环。

小结

- 中间语言
  - 图形表示
    - 树、dag
  - 三地址代码
    - 三地址语句的形式：x:=y op z
    - 三地址语句的种类
      - 简单赋值语句
      - 涉及数组元素的赋值语句
      - 涉及指针的赋值语句
      - 转移语句
      - 过程调用语句
    - 三地址语句的具体实现
      - 三元式、四元式、间接三元式

小结（续1）

- 赋值语句的翻译
  - 文法（赋值语句出现的环境）
  - 仅涉及简单变量的赋值语句的翻译
  - 涉及数组元素的赋值语句的翻译
    - 计算数组元素的地址
  - 访问记录中的域
- 布尔表达式的翻译
  - 数值方法
  - 控制流方法：代码结构
  - 回填技术
    - 思想、问题、方法
    - 与链表操作有关的函数
      - **makelist**
      - **merge**
      - **backpatch**
    - 属性设计
    - 布尔表达式的翻译
- 控制语句的翻译