第8章(Part 2) 中间代码生成

1 中间代码的形式

2 语句的翻译

简单赋值语句、布尔表达式、控制结构、说明语句、数组和结构

1 中间代码形式

中间代码

中间代码(Intermediate code)
(Intermediate representation)
(Intermediate language)

是源程序的一种内部表示
复杂性介于源语言和目标机语言之间

中间代码的作用：

使编译程序的逻辑结构更加简单明确
利于进行与目标机无关的优化
利于在不同目标机上实现同一种语言

中间代码形式

中间代码的形式:

逆波兰式

AST (Abstract Syntax Tree, 抽象语法树)

DAG (Directed Acyclic Graph, 有向无环图)

三元式

四元式 (Three-Address Code, 三地址码)

Bytecode (Java编译器输出, Java 虚拟机输入)

SSA (Static Single Assignment form, 静态单赋值形式)

中间代码的层次

中间代码按照其与高级语言和机器语言的接近程度，可以分成以下三个层次：

高级：最接近高级语言，保留了大部分源语言的结构。

中级：介于二者之间，与源语言和机器语言都有一定差异。

低级：最接近机器语言，能够反映目标机的系统结构，因而经常依赖于目标机。

不同层次的中间代码举例

源语言 (高级语言)	中间代码 (高级)	中间代码 (中级)	中间代码 (低级)
float a[10][20]; **a[i][j+2];**	**t1 = a[i, j+2]**	**t1 = j + 2** **t2 = i * 20** **t3 = t1 + t2** **t4 = 4 * t3** **t5 = addr a** **t6 = t5 + t4** **t7 = *t6**	**r1 = [fp - 4]** **r2 = [r1 + 2]** **r3 = [fp - 8]** **r4 = r3 * 20** **r5 = r4 + r2** **r6 = 4 * r5** **r7 = fp – 216** **f1 = [r7 + r6]**

逆波兰式

逆波兰式，也称做**后缀式**，是最简单的一种中间代码表示形式，早在编译程序出现之前，它就用于表示算术表达式，是波兰逻辑学家卢卡西维奇发明的。

程序设计语言中的表示	逆波兰式
a+b	ab+
a+b*c	abc*+
(a+b)*c	ab+c*
a:=b*c+b*d	abc*bd*+:=

逆波兰式

- 由于逆波兰式表示上的简洁和求值的方便，特别适用于解释执行的程序设计语言的中间表示，也方便具有堆栈体系的计算机的目标代码生成 。
- 逆波兰表示很容易扩充到表达式以外的范围。

程序设计语言中的表示	逆波兰式
GOTO L	L jump
if E then S1 else S2	ES1S2￥ (￥表示if-then-else)

三元式

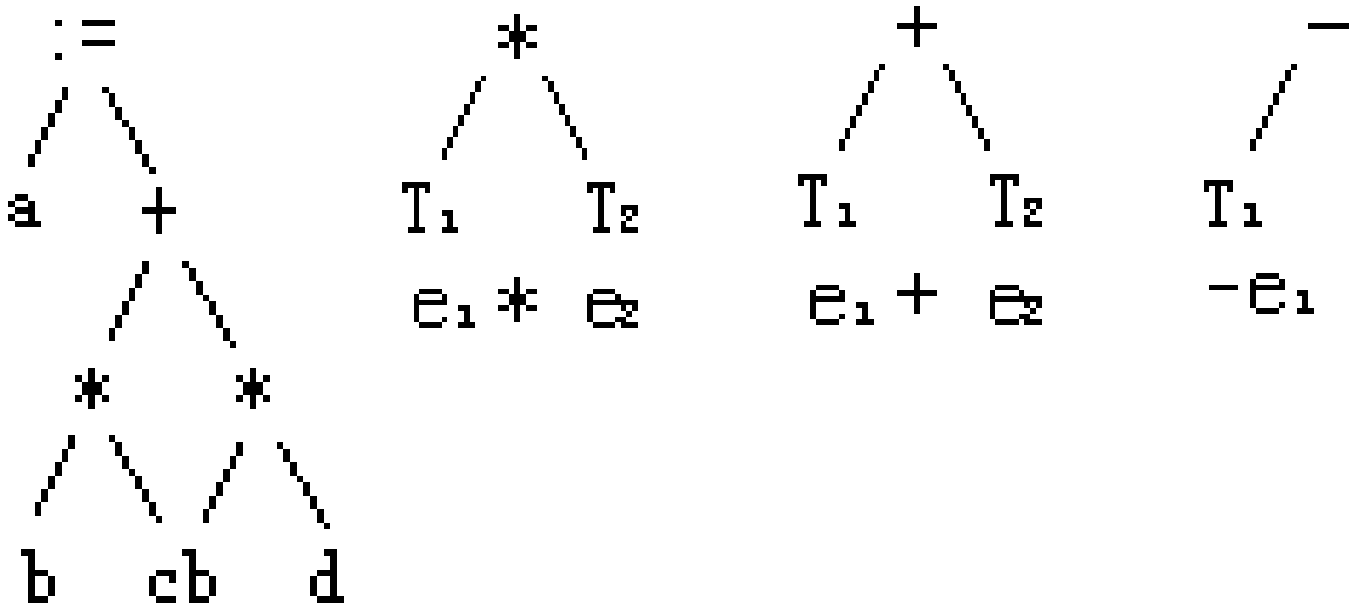
将表达式及语句表示成一组三元式。**每个三元式由三个部分组成：算符op，第一运算对象ARG1和第二运算对象ARG2。**运算对象可能是源程序中的变量，也可能是某个三元式的结果，用三元式的编号表示

例如a:=b*c+b*d的三元式表示为：

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

抽象语法树表示

三元式也可表示成树形表示



四元式

四元式是一种比较普遍采用的中间代码形式。**四元式的四个组成成分是：算符op，第一和第二运算对象ARG1和ARG2及运算结果RESULT。**运算对象和运算结果有时指用户自己定义的变量，有时指编译程序引进的临时变量。

例如, a:=b*c+b*d的四元式表示如下：

（1）（*，　b，　c，　t1）
（2）（*，　b，　d，　t2）
（3）（+，　t1，　t2，　t3）
（4）（：=，t3，　—，　a）

四元式

- **四元式和三元式的主要不同在于**，四元式对中间结果的引用必须通过给定的名字，而三元式是通过产生中间结果的三元式编号。也就是说，四元式之间的联系是通过临时变量实现的。
- **四元式表示很类似于三地址指令**。有时为了更直观，也把四元式的形式写成简单赋值形式。如：

（1）t1：=b*c

（2）t2：=b*d

（3）t3：=t1+t2

（4）a：=t3

将(jump，－，－，L)写成 **goto L**

将(jrop，B，C，L)写成 **if B rop C goto L**

例： A + B * (C - D) + E / (C - D) ^N

逆波兰　A B C D - * + E C D – N ^ / +

四元式　(1) (- C D T1)

(2) (* B T1 T2)

(3) (+ A T2 T3)

(4) (- C D T4)

(5) (^ T4 N T5)

(6) (/ E T5 T6)

(7) (+ T3 T6 T7)

三元式

(1) (- C D)

(2) (* B (1))

(3) (+ A (2))

(4) (- C D)

(5) (^ (4) N)

(6) (/ E (5))

(7) (+ (3) (6))

2 语句的翻译

简单赋值语句的翻译

在四元式中，使用**变量名字**本身表示运算对象ARG1和ARG2，用ti表示RESULT。在实际实现中，它们或者是一个指针，指向符号表的某一登录项，或者是一个临时变量的整数码。

对**id**表示的单词定义一属性**id.name**，用做语义变量；

用**Lookup(id.name)**表示审查id.name是否出现在符号表中，如在，则返回一指向该登录项的指针，否则返回nil。

语义过程**emit**表示输出四元式到输出文件上。语义过程newtemp表示生成一个临时变量，每调用一次，生成一新的临时变量。

语义变量**E.place**，表示存放E值的变量名在符号表的登录项或一整数码。

赋值语句的四元式翻译

(1) S→id：=E {p:=lookup(id.name);
if p≠nil *then*
emit(p ':=' E.place)
else error }

(2) E→E1+E2 {E.place:=newtemp；
emit(E.place ':=' E1.place '+' E2.place)}

(3) E→E1*E2 {E.place:=newtemp；
emit(E.place ':=' E1.place '*' E2.place)}

(4) E→－E1 {E.placeE:=newtemp;
Emit(E.place ':=' 'uminus' E1.place)}

(5) E→(E1) {E.place:=E1.place}

(6) E→id {p:=lookup(id.name);
if p ≠ nil *then*
E.placeE:=p
else error }

```
E→E1*E2 {E.place:=newtemp;
        if E1.type=int AND E2.type=int then
            begin emit(E.place, ':=' , E1.place, ' *i ' , E2.place);
                  E.type:=int
            end
            else if  E1.type=real AND E2.type=real then
            begin emit (E.place, ':=' , E1.place, ' *r ' , E2.place);
                  E.type:=real
            end
            else if E1.type=int /* and E2.type=real*/then
            begin t:=newtemp;
                  emit(t, ':=', ' itr ', E1.place);
                  emit(E.place, ':=',  t , ' *r ' , E2.place);
                  E.type:=real
            end
            else    /*E1·type=real and E2．type=int*/
            begin t:=newtemp;
                  emit(t, ':=', 'itr',  E2.place);
                  emit(E.place, ':=', E1.place, ' *r ' , t);
                  E.type:=real
            end;
          }
```

类型转换的语义处理

布尔表达式的翻译

布尔表达式的作用：计算逻辑值；用做改变控制流语句中的条件表达式。

布尔表达式是由布尔算符and，or和not施于布尔变量或**关系表达式**而成。

用文法描述如下：

E→E and E | E or E | not E | id rop id | true | false

其中，rop是关系符，如＜＝，＜，＞＝，＞，≠等。

布尔表达式的翻译方法

计算布尔表达式值的两种方法：

一、如同计算算术表达式一样，步步计算出各部分的真假值，最后计算出整个表达式的值。

如： 1 or (not 0 and 0) or 0

＝1 or (1 and 0) or 0

＝1 or 0 or 0

＝1 or 0

＝1

约定：布尔算符的优先顺序（从高到低）为not 、and、or，并且and和or服从左结合。

二、采取某种优化措施，只计算部分表达式。

如：要计算A or B，若计算出A的值为1，那么B的值就无需再计算了，因为不管B的值为何结果，A or B的值都为1。

若计算A and B，…

布尔表达式的翻译方法

布尔表达式**a or b and not c**翻译成的四元式序列为：

(1) t1：=not c
(2) t2：=b and t1
(3) t3：=a or t2

对于像**a < b**这样的关系表达式，可看成等价的条件语句if a < b then 1 else 0，它翻译成的四元式序列为：

(1) if a < b goto (4)
(2) t:=0
(3) goto (5)
(4) t:=1
(5) …

用数值表示布尔值的翻译方案

E→E1 or E2

{E.place:=newtemp；
emit（E.place′:=′E1.place　′or′E2.place）}

E→E1 and E2

{E.Place:=newtemp；
emit（E.　place′:=′E1.place ′and′E2.place）}

E→not E1

{E.Place:=newtemp：；
emit（E.place′:=′not′E1.place）}

E→(E1)

{E.place:=E1.place}

E→id1 rop id2

{E.place:=newtemp；
emit（′if′id1.place rop id2.place′goto′nextstat+3）；
emit（E.place′:=″0′）；
emit（′goto′nextstat+2）；
emit（E.place ′:=″1′）}

E→true

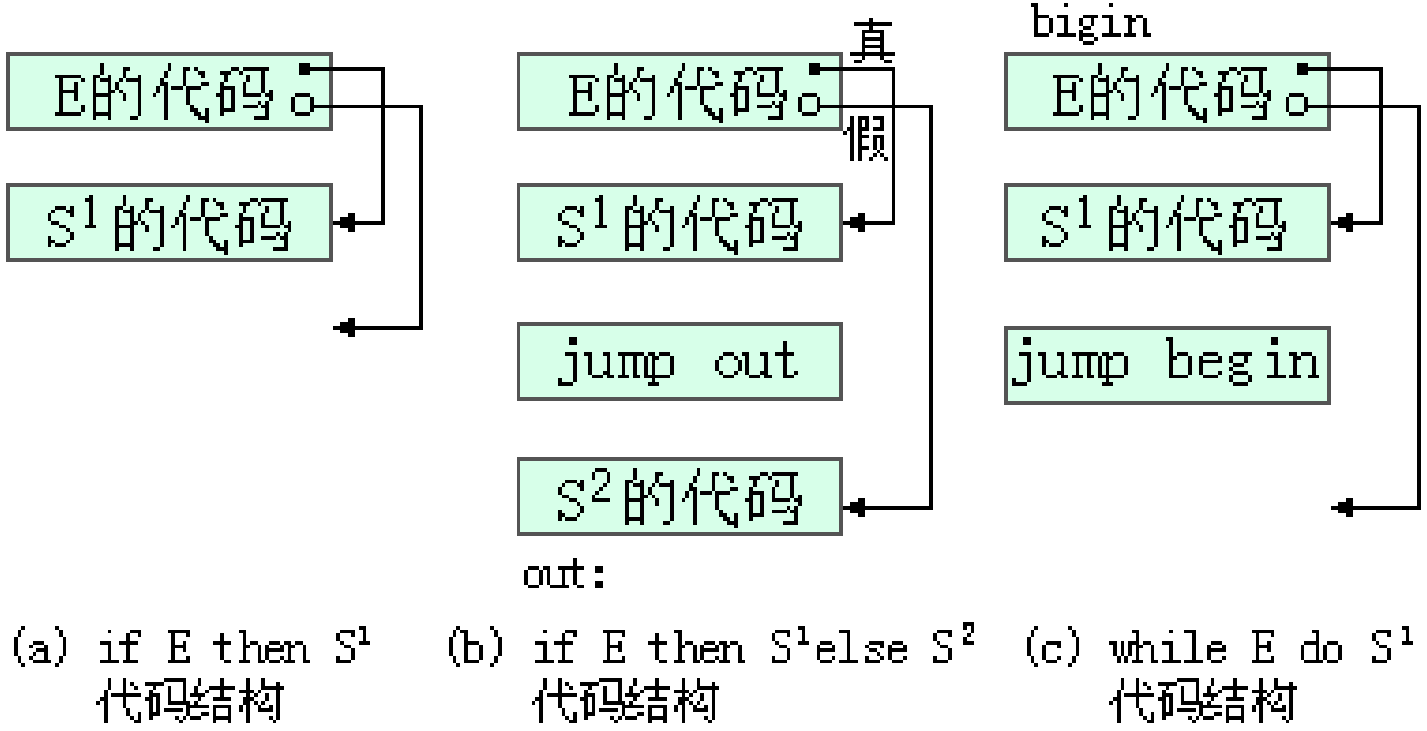
{E.place:=newtemp；
emit（E.place′ : =″1′）}

E→false

{E.place:=newtemp；
emit（E.place′:=″0′）}

控制语句中布尔表达式的翻译

S→if E then S1 | if E then S1 else S2 | while E do S1



(a) if E then S^1 代码结构　(b) if E then S^1 else S^2 代码结构　(c) while E do S^1 代码结构

只把条件转移的布尔表达式 E 翻译成仅含条件转和无条件转的四元式。

即，布尔表达式 E：**a rop b** 翻译成

if a rop b goto E.true（E 为真的出口转移地址）

goto E.false（E 为假的出口转移地址）

例如 a<b or c<d and e<f 翻译成如下四元式：

(1) if a<b goto **E.true**

(2) goto (3)

(3) if c<d goto (5)

(4) goto **E.false**

(5) if e<f goto **E.true**

(6) goto **E.false**

（不是最优！）

if a < b or c < d and e > f then S1 else S2的四元式序列为:

(1) if a < b goto **(7)**
/* (7) 是整个布尔表达式的真出口*/
(2) goto (3)
(3) if c < d goto **(5)**
(4) goto **(p+1)**
/* (p+1) 是整个布尔表达式的假出口*/
(5) if e > f goto **(7)**
(6) goto **(p+1)**
(7) (关于S1的四元式)
…
(p) goto **(q)**
(p+1) (关于S2的四元式)
…
(q)

为了记录需回填地址的四元式，常采用一种"拉链"的办法。把需回填E．true的四元式拉成一链，把需回填E．false的四元式拉成一链，分别称做"真"链和"假"链。

拉链返填：

```
          ……
（10）    goto   L
          ……
（20）    goto   L
          ……
（30）    goto   L
          ……
（40）L：……
```

```
（10）goto   0
      ……        链尾（10）
（20）goto   10
      ……
（30）goto   20
      ……        链头（30）
（40）L：……
```

语句 if a<b or c<d and e<f then S^1 else S^2 的四元式序列：

(1) if a<b goto (7) (E.true) (1)和(5）拉链（真）
(2) goto (3)

(3) if c<d goto (5)

(4) goto (p+1) (E.false) (4) 和(6) 拉链（假）
(5) if e<f goto (7)
(6) goto (p+1)
(7)(S^1 的四元式
……
(p-1) ……）
(p) goto (q)
(p+1) (S^2 的四元式
……
(q-1) ……）
(q)

语义描述使用的量：

E.true“真”链， E.false“假”链

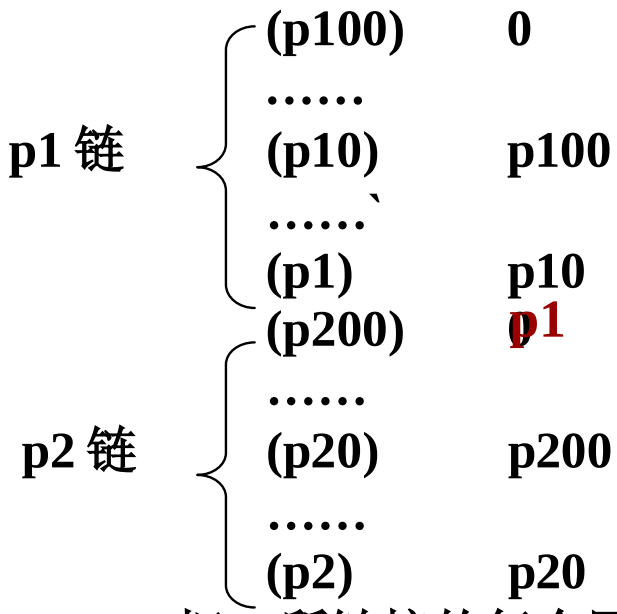
E.codebegin E 的第一个四元式

nextstat 下一四元式地址

过程 emit( ) 输出一条四元式，而后 nextstat+1

merge(p1,p2)

例如：merge(p1,p2)



backpatch(p, t)：把 p 所链接的每个四元式的第四区段都填为 t。

自下而上分析中的一种翻译方案：

(1) $E \rightarrow E^1$ or E^2 {backpatch(E^1.false, E^2.Codebegin);
E.Codebegin:= E^1.codebegin;
E.true:=merge(E^1.true, E^2.true)
E.false:= E^2.false}

(2) $E \rightarrow E^1$ and E^2 {backpatch(E^1.true, E^2.codebegin);
E.Codebegin:= E^1.codebegin;
E.true:= E^2.true;
E.false:= merge(E^1.false, E^2false);}

(3) $E \rightarrow$ not E^1 {E.true:= E^1.false;
E.Codebegin:= E^1.codebegin;
E.false:= E^1.true

(4) E→(E[1]) { E.true:= E[1].true;
E.Codebegin:= E[1].codebegin;
E.false:= E[1].false}

(5) E→id1 rop id2 { E.true:=nextstat;
E.Codebegin:=nextstat;
E.false:=nextstat+1;
emit(‘if’ id1.place ‘rop’ id2.place ‘goto’–);
emit(‘goto’-)}

(6) E→ture { E.true:=nextstat;
E.codebegin:=nextstat;
emit(‘goto’-)}

a<b or c<d and e<f 翻译成四元式序列的过程：

100： if a<b goto ---

101： goto 102

102: if c<d goto 104

103: goto ---

104: if e<f goto 100

105: goto 103

“真”链首E.trure 为104

“假”链首E.false 为105

控制结构的翻译

1.条件转移

G[S]

（1） S→ if E then S
（2） | if E then S else S
（3） | while E do S
（4） | begin L end
（5） | A
（6） L→L;S
（7） | S

其中各非终结符号的意义是：

S--语句
L--语句串
A--赋值句
E--布尔表达式

条件转移

G'[S]:

(1) $S \rightarrow C\ S^1$

(2) $|T^p\ S^2$

(3) $|W^d\ S^3$

(4) |Begin L end

(5) |A

(6) $L \rightarrow L^s\ S^1$

(7) $|S^2$

(8) C →if E then

(9) $T^p \rightarrow C\ S$ else

(10) $W^d \rightarrow W\ E$ do

(11) W →while

(12) $L^s \rightarrow L$;

条件转移

G'[S]:

(1) S →C S^1 {S.chain:=merge(C.chain, S^1.chain)}

(2) |T^p S^2 {S.chain:=merge(T^p.chain, S^2.chain)}

(3) |W^d S^3 {Backpatch(S^3.chain, W^d.codebegin)

emit('goto' W^d.codebegin)

S.chain:=W^d.chain}

(4) |Begin L end {S.chain:=L.chain }

(5) |A {S.chain:=0}

(6) L →L^s S^1 {L.chain:= S^1.chain }

(7) |S^2 {L.chain:= S^2.chain }

条件转移

G'[S]:

(8) C →if E then {backpatch(E.true, nextstat)
C.chain:=E.false}

(9) T^p → C S else {q:=nextstat
emit('goto'---)
backpatch(C.chain, nextstat)
T^p.chain:=merge(S.chain, q)}

(10) W^d →W E do {backpatch(E.true, nextstat)
W^d.chain:=E.false
W^d.codebegin:= W.codebegin}

(11) W →while {W.codebegin:=nextstat}

(12) L^s →L; {backpatch(L.chain, nextstat)}

if E then S^1

(1) $S \rightarrow C\ S^1$ {S.chain:=merge(C.chain, S^1.chain)}

(8) $C \rightarrow$ if E then {backpatch(E.true, nextstat)
C.chain:=E.false}

if E then S^1 else S^2

(1)	$S \rightarrow C\ S^1$	{S.chain:=merge(C.chain, S^1.chain)}
(2)	$S \rightarrow T^p\ S^2$	{S.chain:=merge(T^p.chain, S^2.chain)}
(8)	C→if E then	{backpatch(E.true, nextstat)
		C.chain:=E.false}
(9)	$T^p \rightarrow C\ S$ else	{q:=nextstat
		emit('goto'---)
		backpatch(C.chain, nextstat)
		T^p.chain:=merge(S.chain, q)}

while E do S

(1) $S \rightarrow W^d S^3$ {Backpatch(S^3.chain, W^d.codebegin)
emit('goto', W^d.codebegin)
S.chain:=W^d.chain}

(10) $W^d \rightarrow W$ E do {backpatch(E.true, nextstat)
W^d.chain:=E.false
W^d.codebegin:= W.codebegin}

(11) W →while {W.codebegin:=nextstat}

语句串

(4)	$S \rightarrow A$	{S.chain:=0}
(5)	$L \rightarrow L^s S^1$	{L.chain:= S^1.chain }
(6)	$L \rightarrow S^2$	{L.chain:= S^2.chain }
(12)	$L^s \rightarrow L;$	{backpatch(L.chain, nextstat)}

while（A＜B）do if（C＜D）then X:=Y+Z

翻译成如下的一串四元式:

```
100   if A＜B goto 102
101   goto  107
102   if C＜D goto 104
103   goto 100
104   T：=Y+Z
105   X：=T
106   goto 100
107
```

2. 开关语句

开关语句的形式为：

```
switch E of
case V1：S1
case V2：S2
…
case Vn-1：Sn-1
default：Sn
end
```

直观上看，**case**语句翻译成如下的一连串条件转移语句：

```
t:=E;
L1: if t≠V1 goto L2;
S1;
goto next;
L2: if t≠V2 goto L3;
S2;
goto next;
…
Ln-1: if t≠Vn-1 goto Ln;
Sn-1;
goto next;
Ln: Sn;
next:
```

建立二元组表的方法

二元组的结构：第一元为Vi的值，第二元为Vi对应的语句Si的标号；

翻译过程：当读入关键字switch时，产生新的标号test、next和一个临时变量t；然后产生计算E值的代码，将E值放入t；读入of时产生四元式goto test。每当读入关键字case时，产生标号Li，填入符号表（设位置为Pi），将Pi连同case后的Vi，顺序存放到二元组表存储区；然后产生Si的相关代码和goto next；读到end时，则产生形成n个分支的代码。

计算E值、并将其放入临时变量t中的中间代码

goto test

L_1: S_1的中间代码

goto next

…

L_n: Sn的中间代码

goto next

test: if t=V_1 goto L_1

…

if t=V_{n-1} goto L_{n-1}

goto L_n

next:

也可将if t=V_i goto L_i 写成(case, V_i, L_i, ---)的形式

开关语句的四元式代码

3. for循环语句

如：for i：=E1 step E2 until E3 do S1

该循环句等价于：

```
i:=E1;
goto OVER;
AGAIN:  i:=i+E2;
OVER:   if i≤E3 then
begin S1; goto AGAIN end;
```

可用以下文法来描述：

F1→for i:=E1

F2 →F1 step E2

F3 →F2 until E3

S →F3 do S1

F1→for i:=E1 {emit(enry(i), ‘:=’, E1.place);
F1.place:=entry(i); F1.chain:=nextstat;
emit(‘goto’ ---); F1.codebegin:=nextstat}

F2 →F1 step E2 {F2.codebegin:= F1.codebegin;
F2.place:=F1.place;
emit(F1.place, ‘:=’, E2.place, ‘+’, F1.palce);
backpatch(F1.chain, nextstat);}

F3 →F2 until E3 {F2.codebegin:= F1.codebegin;
q:=nextstat;
emit(‘if’, F2.place, ‘≤’, E3.place, ‘goto’, q+2);
F3.chain:=nextstat; emit(‘goto’)---}

S →F3 do S1 {S1的代码; emit(‘goto’ F3.codebegin)
backpatch(S1.chain, F3.codebegin);
S.chain:=F3.chain}

例如循环语句 for I:=1 step 1 until N do M:=M+1,
四元式序列如下:

100 I:=1
101 goto 103
102 I:= I+1
103 if I≤N goto 105
104 goto 108
105 T:=M+I
106 M:=T
107 goto 102
108

4.出口语句

由于转移的目标在**break、exit**语句之后的循环外才能确定，因而一遍扫描的编译中也要用回填技术才能给出转移目标。

编译程序对每个循环可使用“**循环描述符**”来记录一些必要的信息：指向该循环的直接外层循环描述符的指针，循环名在符号表中的位置和exit转移的目标。

5. goto语句

多数程序设计语言中的转移是通过标号和goto语句实现的。带标号语句的形式是**L：S**；goto语句的形式是**goto L**。

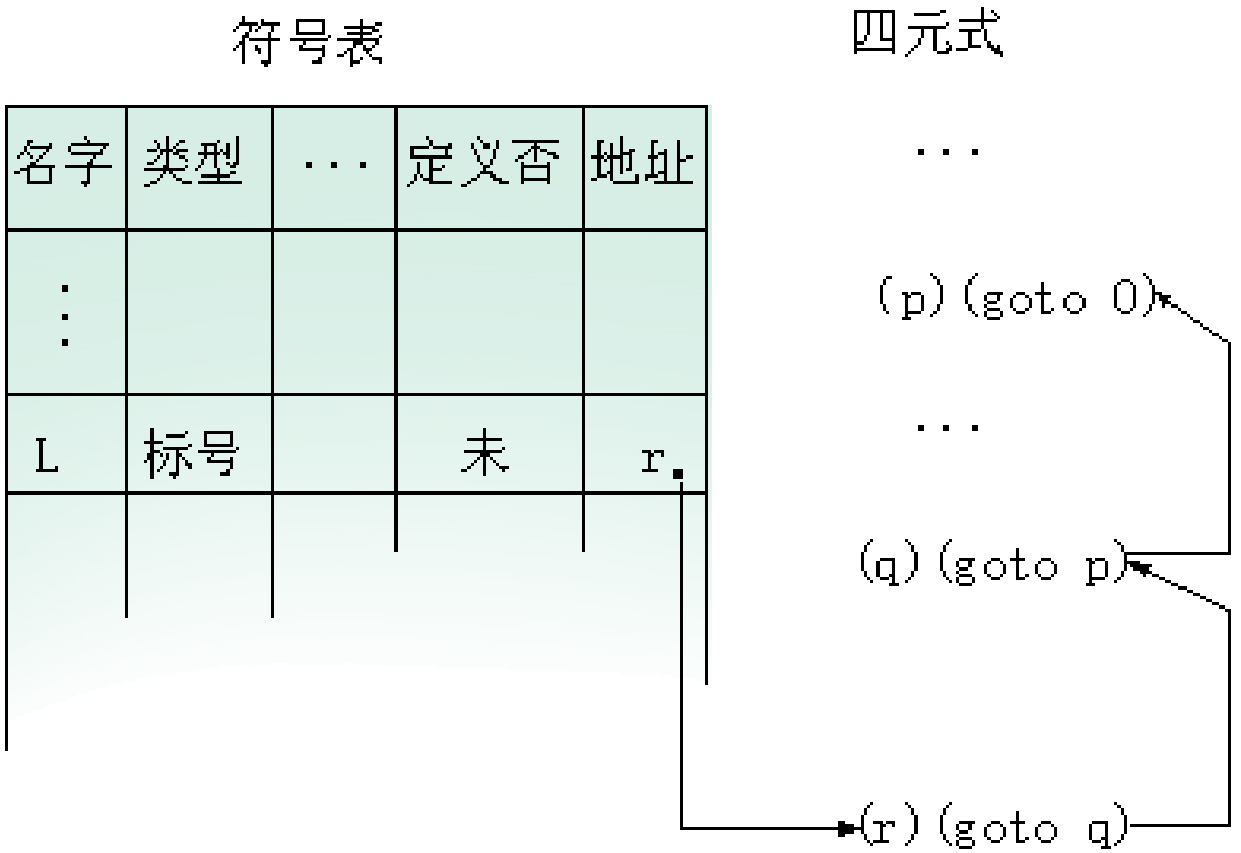
如果goto L是一个向上转移的语句，那么，当编译程序碰到这个语句时，L必是已定义了的。通过对L查找符号表获得它的定义地址p，编译程序可立即产生出相应于这个goto L的四元式如（j，－，－，p）。

如果goto L是一个向下转移的语句，也就是说，标号L尚未定义，那么，若L是第一次出现，则把它填进符号表中并标志上“**未定义**”。同时只能产生一个不完全的四元式（goto －），它的转移目标须待L定义时再**回填**进去。

可采用**拉链**的方法记录以L为转移目标的四元式的地址，以便L定义时对这些四元式进行回填。

建链的方法是：若goto L中的标号L尚未在符号表中出现，则把L填入表中，置L的"定义否"标志为"未"，把nextstat填进L的地址栏中作为新链首，然后，产生四元式（goto 0），其中0为链尾标志。若L已在符号表中出现（但"定义否"标志为"未"），则把它的地址栏中的编号（记为q）取出，把nextstat填进该栏作新链首，然后，产生四元式（goto q）。

未定义标号的引用链



一旦标号L定义时，将根据这条链**回填那些待填转移目标的四元式**。若用下面的产生式来定义标号语句

S→label S

label→i：

则当用label→i：进行归约时，应做如下的语义动作：

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nextstat。

2.若L已在符号表中但“类型”不为“标号”或“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，把地址栏中的链首（记为q）取出，同时把nextstat填在其中，最后，执行回填。

6. 过程调用的四元式生成

一个描述过程调用的文法：

(1) S → call i (<arglist>)

{for 队列arglist.queue 的每一项 p do

GEN(par, ---, ---, p);

GEN(call, ---, ---, entry(i))}

(2) <arglist> →<srglist>[1], E

{把E.place 排在arglist[1].queue的末端；

arglist.queue:= arglist[1].queue}

(3) <arglist> →E {新建一个arglist.queue，它只包含一项E.place}

数据队列queue用来记录每个实在参数的地址。

说明语句的翻译

1.简单说明语句的翻译

程序设计语言中最简单的说明句的语法描述为：

D→integer <namelist> | real <namelist>

<namelise>→<namelist>，id | id

上述文法可以改写成：

D→D1, id

| integer id

| real id

简单说明语句的翻译

对应的语义动作如下：

(1) D→integer id **{enter(id,int);**
D.ATT：=int}

(2) D→real id **{enter(id, real);**
D.ATT：=real}

(3) D→D1, id **{enter(id, D1.ATT);**
D.ATT：=D1.ATT}

其中，语义变量D.ATT，用以记录说明句所引入的名字的性质(int还是real)，过程enter(id, A)把名字id和性质A登录在符号表中。

2. 过程中说明语句的翻译

过程的说明部分的翻译，就是为过程的局部名字建立符号表项(安排存储)，此时要记录名字和存储的**相对地址**。过程的说明还允许**嵌套**。

如：

D→real id {enter(id, real, offset);
D.att:=real;
D.width:=8;
offset:=offset+D.width}

允许过程嵌套的语言:

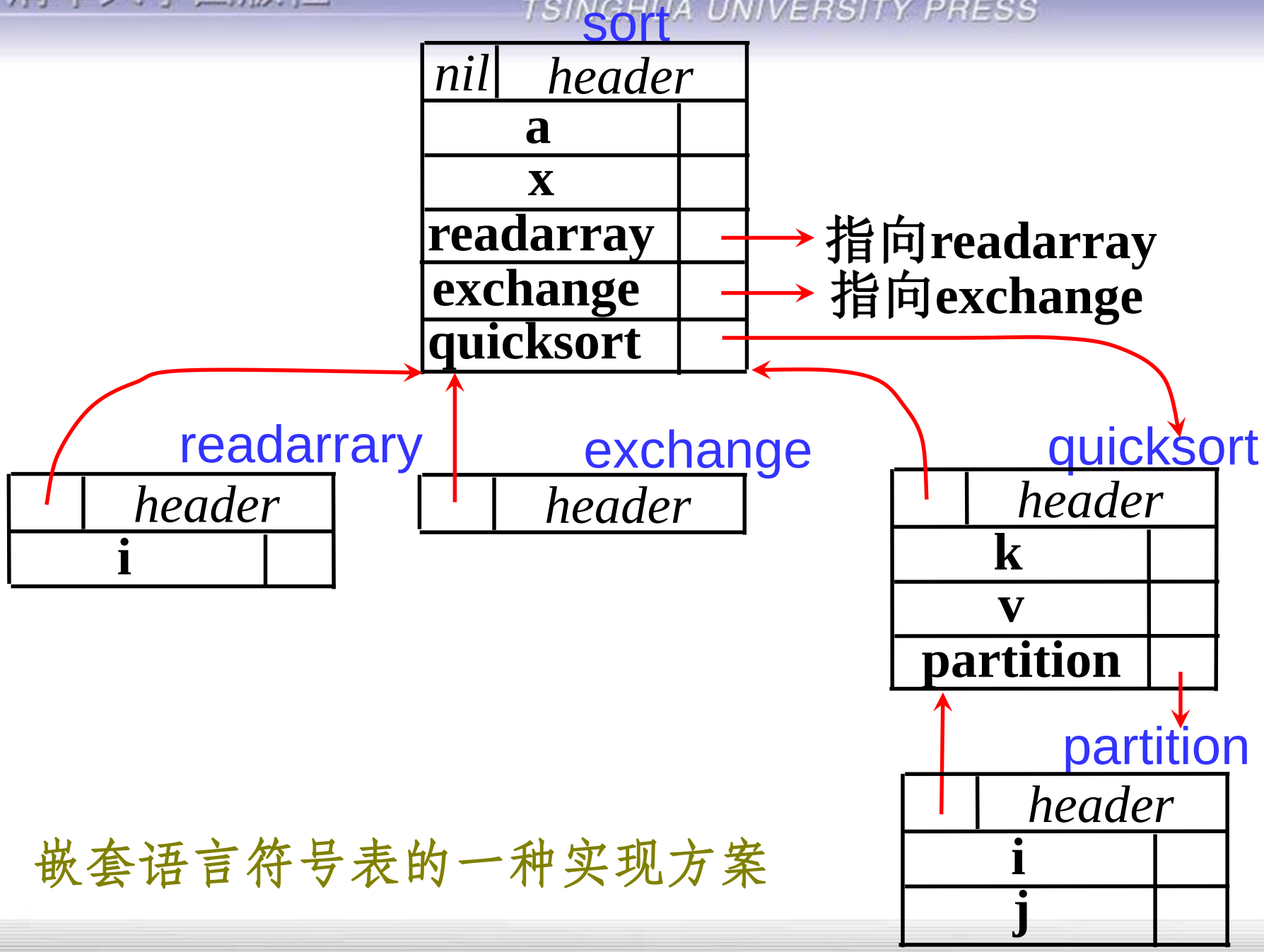
$P \rightarrow D$

$D \rightarrow D\ ;\ D \mid id : T \mid proc\ id;\ D;\ S$

为每个过程建立一个新的符号表;
新的表有一个指针指向其外围过程的符号表;
过程名作为外围过程的局部名。

例：包含嵌套过程的程序

P. 197



嵌套语言符号表的一种实现方案

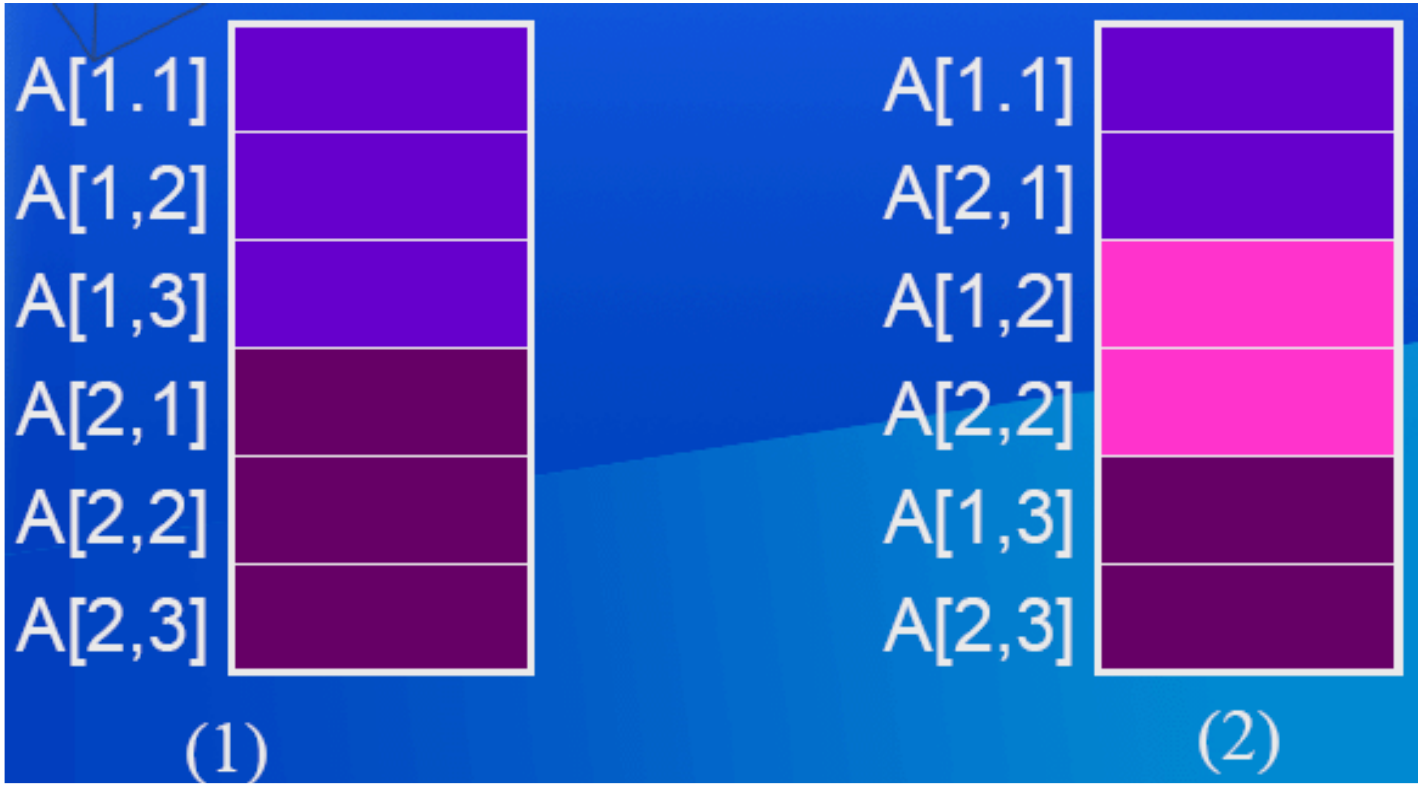
数组和结构的翻译

数组的翻译

多维数组在一维内存结构中的存放:

(1)**按行存放**　　(2)**按列存放**

例如A： array[1:2 , 1:3]



数组元素地址的计算

一般而言，设A是一个n维数组：array [l1:u1 , l2:u2 , …, ln:un]；每一维尺寸为：di= ui–li+ 1；a为数组的首地址，每个元素占m个单元。则A[i1,i2,…,in]的地址为：

D=a+((i1-l1)d2d3…dn+(i2-l2)d3d4…dn+…+(in-1-ln-1)dn+(in-ln))m

分解后可得：**D=CONSPART+VARPART**

其中：**CONSPART= a –C**

C=((…((l1d2+l2)d3+l3)d4…+ln-1)dn+ln)m

VARPART=((…((i1d2+i2)d3+i3)d4+…in-1)dn+in)m

内情向量表

对于**VARPART**部分，只有在运行时才能计算，为此，在编译时必须把数组的有关信息记录在一张内情向量表中。

l_1	u_1	d_1
l_2	u_2	d_2
…	…	…
l_n	u_n	d_n
n	C	
TYPE	a	

树组引用的中间代码形式

- 把**VARPART**计算在某一“变址”单元**T**中，用**CONSPART**作为“基址”，然后可用如下方式访问：**CONSPART [T]**;
- CONSPART=a-C的值可存放在**T1**中，那么可用**T1[T]**来表示数组元素的地址。
- 则数组引用时的四元式为：

  (=[ ] , T1[T] , _ , X) // X:=T1[T]
- 树组元素赋值的四元式为：

  ([ ]= , X , _ , T1[T]) // T1[T]:=X

若A是一个10*20数组，那么X:=A[I, J]的四元式序列为：

(* , I , 20 , T1)

(+ , J, T1, T1)

(-, A , 21 , T2)

(=[ ] , T2[T1] , --- T3)

(:= , T3, --- , X)

其中，(=[ ], T1[T], ---, X)相当于X:= T1[T]，

([ ]= , X, --- ,T1[T])相当于T1[T]:= X。

More for Array Translation

1. 文法

(1) $S \rightarrow L := E$
(2) $E \rightarrow E_1 + E_2$
(3) $E \rightarrow (E_1)$
(4) $E \rightarrow L$
(5) $L \rightarrow Elist\ ]$
(6) $L \rightarrow id$
(7) $Elist \rightarrow Elist_1, E$
(8) $Elist \rightarrow id\ [\ E$

2. 属性

L.place, E.place, Elist.place, id.place:
对应的数据对象的符号表入口地址

L.offset { null: 表示L是简单变量
否则：下标变量的偏移量，即$w*e_k$

Elist.array: 数组的符号表入口地址(相当base)。
Elist.ndim: 当前处理下标的计数(维数)。

3. 翻译模式

(1) $S \rightarrow L := E$

{ if L.offset = null then // 简单变量
emit(L.place ':=' E.place)
else //数组元素
emit(L.place '[' L.offset ']' ':=' E.place) }

(2) $E \rightarrow E_1 + E_2$

{ E.place := newtemp;
emit(E.place ':=' E_1.place '+' E_2.place) }

(3) $E \rightarrow (E_1)$

{ E.place := E_1.place }

(4)E → L

```
{ if  L.offset = null  then
      E.place := L.place
  else  begin
      E.place := newtemp;
      emit(E.place ':=' L.place'[' L.offset ']')
  end }
```

(5) L → Elist]

{ L.place := *newtemp*;
emit(L.place ':=' Elist.array '-', c_k);
L.offset := newtemp;
emit(L.offset ':=' w '*' Elist.place) }

(6) L → id

{ L.place := id.place;
L.offset := null }

(7) Elist → $Elist_1$, E

```
{ t := newtemp;
  m:= Elist1.ndim + 1;
  emit(t ':=' Elist1.ndim '*'
                limit(Elist1.array, m));
  emit(t ':=' t '+' E.place);
  Elist.array := Elist1.array;
  Elist.place := t;
  Elist.ndim := m }
```

(8) Elist → id [E

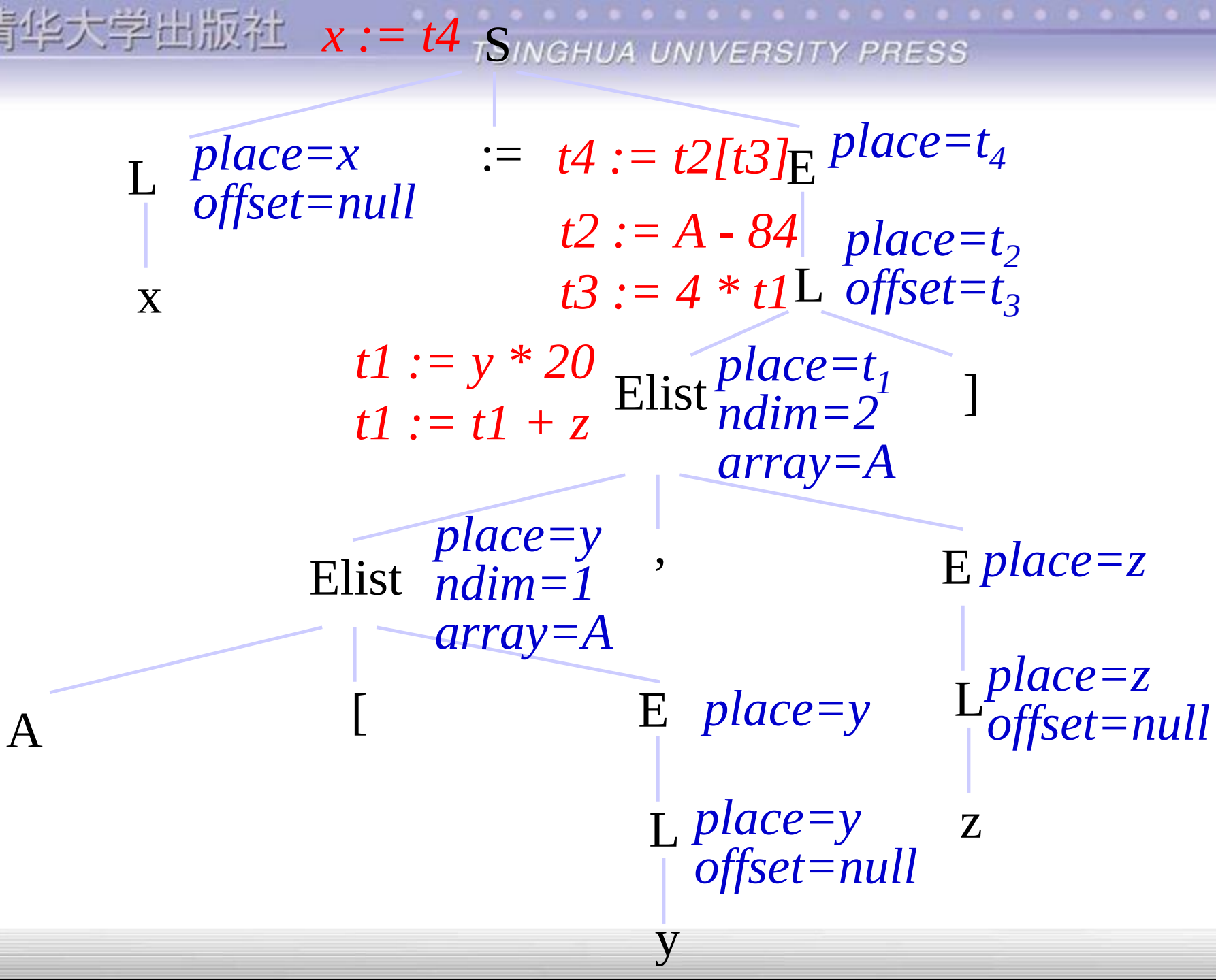
{ Elist.place := E.place;
E.ndim := 1;
Elist.array := id.place }

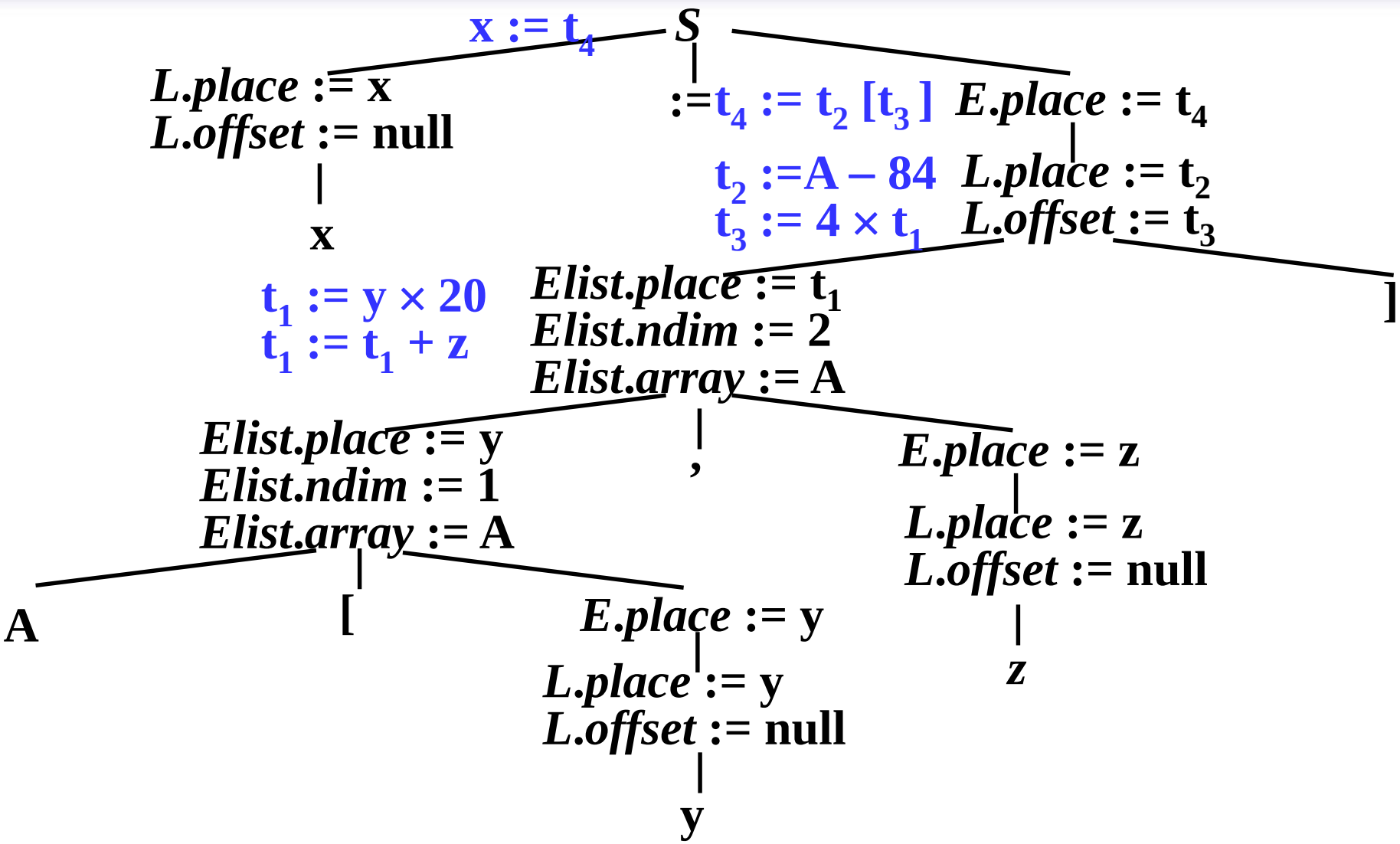
例：设A为一个10*20的数组, n_1=10, n_2=20, w=4, 数组第一个元素为A[1,1]。

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

赋值语句x:=A[y,z]翻译为三地址序列：

```
t1 := y * 20
t1 := t1 + z
t2 := A - 84
t3 := 4 * t1
t4 := t2[t3]
x := t4
```


x := t4
S
L
place=x
offset=null
:=
t4 := t2[t3]
E
place=t4
x
t2 := A - 84
t3 := 4 * t1
place=t2
L
offset=t3
t1 := y * 20
t1 := t1 + z
Elist
place=t1
ndim=2
array=A
]
Elist
place=y
ndim=1
array=A
,
E
place=z
A
[
E
place=y
L
place=z
offset=null
L
place=y
offset=null
z
y


x := t4
S
L.place := x
L.offset := null
x
:=t4 := t2 [t3]
E.place := t4
t2 :=A – 84
t3 := 4 × t1
L.place := t2
L.offset := t3
t1 := y × 20
t1 := t1 + z
Elist.place := t1
Elist.ndim := 2
Elist.array := A
]
Elist.place := y
Elist.ndim := 1
Elist.array := A
,
E.place := z
L.place := z
L.offset := null
z
A
[
E.place := y
L.place := y
L.offset := null
y

x := A[y, z]的注释分析树

结构的翻译

结构是由已知类型的数据组合起来的一种数据类型。如，

```
struct date{
    int date;
    char month-name[4];
    int year;
};
```

结构说明的文法描述如下：

<type> → struct {f_1};

| int

| char

| pointer

f_1 →f_1; f | f

f →<type> i | <type> i[n]

中间代码生成时，需记录所有**分量的信息**：

分量1	分量2	… …	分量n

每个分量中记录**名字、类型、长度**等属性。

语义过程**FILN(name, L)**和**FILO(name, L)**分别将分量名表中名字为name的项的len和offset属性赋L值。

offset:此分量之前的分量长度之和。

f →<type> i {f.name:=i.name; f.len:=type.len; FILN(i.name, f.len)}

f → <type> i[n] {f.name:=i.name; f.len:=type.len * n.val; FILN(i.name, f.len}

f_1 → f {FILO(f.name, 0); f1.len:=f.len}

f_1 →$f_1^{(1)}$; f {FILO(f.name, $f_1^{(1)}$.len); f1.len:= $f_1^{(1)}$.len+f.len}

<type> → struct {f_1}; {type.len:= f1.len}

<type> → int {type.len:= 1}

<type> → char {type.len:= 4}

<type> → pointer {type.len:= 4}

习题

1. 请将表达式-(a+b)*(c+d)-(a+b+c) 表示成逆波兰式、抽象语法树和四元式形式。

2. 请将下列语句

while (A<B) do if (C>D and E<F) then X:=Y+Z else X:=Y-Z翻译成四元式。

3. 令A为一个二维数组A[1:10, 10:20]，写出赋值语句**A[i+1, j+1]:=a*b-10**的四元式序列。