中国科学技术大学
University of Science and Technology of China
USTC, CHINA
ADSLAB
先进数据系统实验室

《编译原理与技术》
中间代码生成 III
（与类型相关部分）

计算机科学与技术学院
李 诚
2021-11-10

与类型相关的中间代码生成

中国科学技术大学
University of Science and Technology of China

- 符号表的组织
- 声明语句的翻译
- **数组寻址的翻译**
- **类型分析的其他应用**

数组元素的翻译

❑数组类型的声明

e.g. Pascal的数组声明，

A : array[$low_1..high_1$,…,$low_n..high_n$] of integer ;

数组元素： A[i , j, k,…] 或 A[i][j][k]…

（下界） $low_1 \le i \le high_1$ （上界） , …

e.g. C的数组声明，

int A [100][100][100];

数组元素： A[i][30][40] $0 \le i \le (100-1)$

数组元素的翻译

❑翻译的主要任务

- ❖输出(Emit)地址计算的指令
- ❖“基址[偏移]”相关的中间指令：t =b[o]，b[o]=t

数组元素的地址计算

❑一维数组A的第*i*个元素的地址计算

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

base：整个数组的基地址，也是分配给该数组的内存块的相对地址

low：下标的下界

w：每个数组元素的宽度

数组元素的地址计算

- 一维数组A的第i个元素的地址计算

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

base：整个数组的基地址，也是分配给该数组的内存块的相对地址

low：下标的下界

w：每个数组元素的宽度

可以变换成

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

$low \times w$**是常量，编译时计算，减少了运行时计算**

数组元素的地址计算

❑二维数组

A: array[1..2, 1..3] of T

❖列为主

A[1, 1], A[2, 1], A[1, 2], A[2, 2], A[1, 3], A[2, 3]

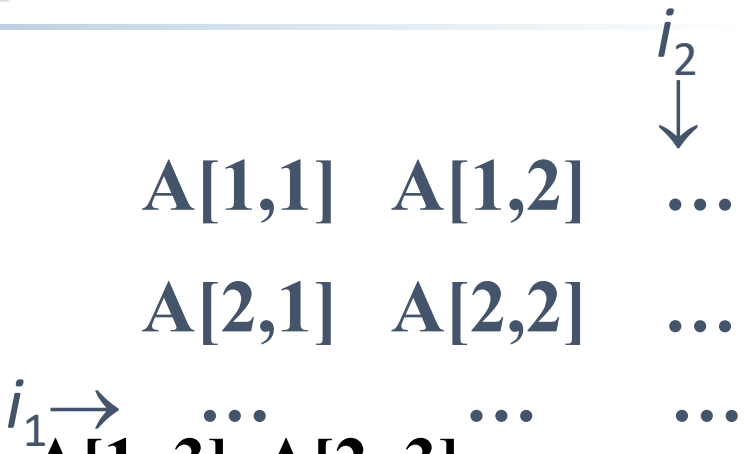
❖ 行为主

A[1, 1], A[1, 2], A[1, 3], A[2, 1], A[2, 2], A[2, 3]

数组元素的地址计算

❑二维数组

A: array[1..2, 1..3] of T



❖列为主

A[1, 1], A[2, 1], A[1, 2], A[2, 2], A[1, 3], A[2, 3]

❖行为主

A[1, 1], A[1, 2], A[1, 3], A[2, 1], A[2, 2], A[2, 3]

$base + ((i_1 - low_1) \times n_2 + (i_2 - low_2)) \times w$

（A[i_1, i_2]的地址，其中$n_2 = high_2 - low_2 + 1$）

变换成 $((i_1 \times n_2) + i_2) \times w +$

$(base - ((low_1 \times n_2) + low_2) \times w)$

数组元素的地址计算

- 多维数组下标变量A[$i_1, i_2, ..., i_k$]的地址表达式
  - 以行为主

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

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

数组元素的地址计算

❑多维数组下标变量A[$i_1, i_2, ..., i_k$]的地址表达式

❖以行为主

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

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

红色部分是数组访问翻译中的最重要的内容

递推公式：

$e_1 = i_1$

$e_m = e_{m-1} \times n_m + i_m$

数组元素地址计算翻译方案

- 下标变量访问的产生式

$S \rightarrow L := E$　　$L \rightarrow \text{id} [\ Elist\ ] \mid \text{id}$

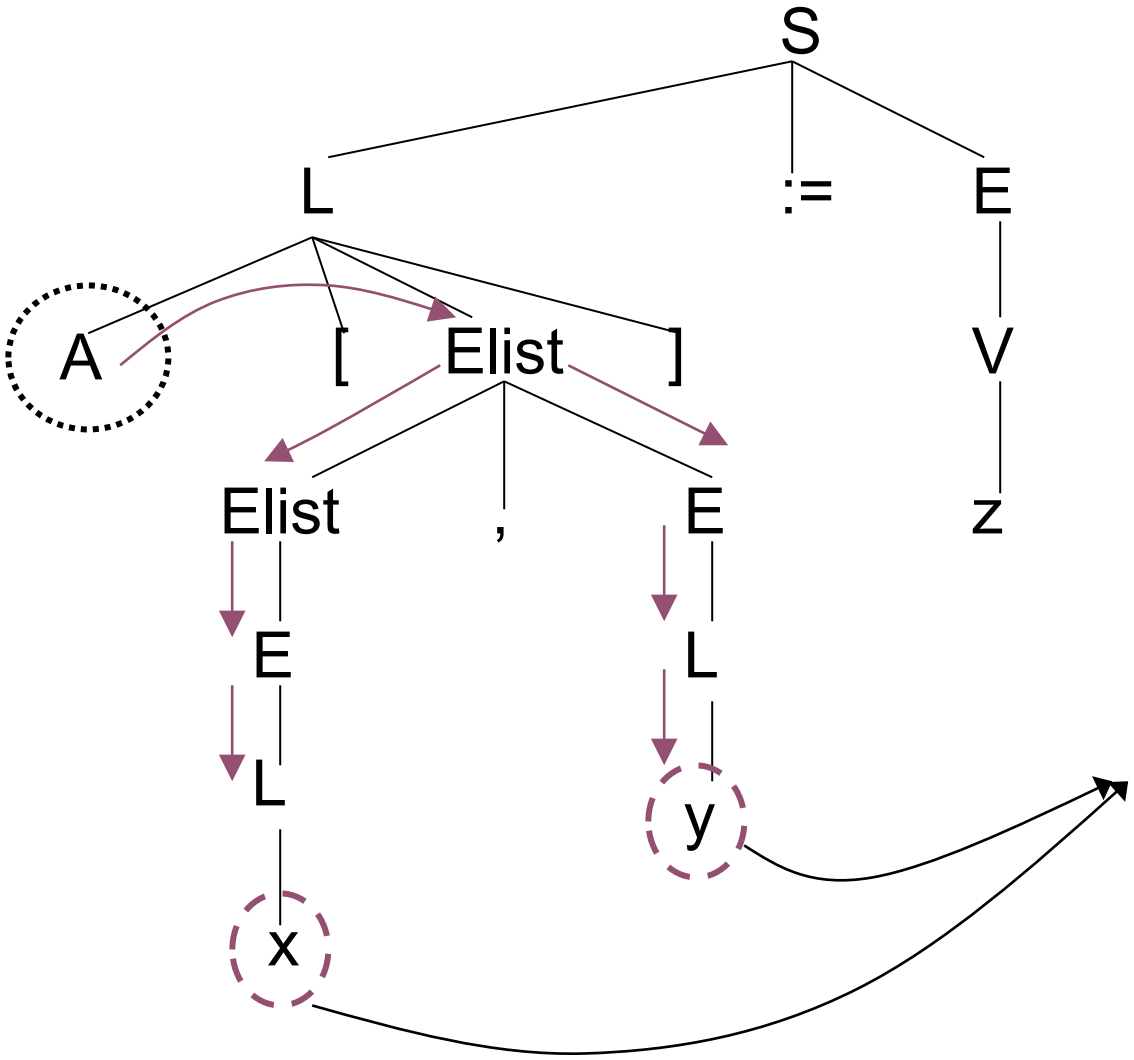
$Elist \rightarrow Elist, E \mid E$　　$E \rightarrow \text{L} \mid \ldots$

- 采用语法制导的翻译方案时存在的问题

$Elist \rightarrow Elist, E \mid E$

由Elist的结构只能得到各维的下标值，但无法获得数组的信息（如各维的长度）

A[x,y] := z的分析树



当分析到下标（表达式）x和y时，要计算地址中的“可变部分”。这时需要知晓数组A的有关的属性，如n_m，类型宽度w等，而这些信息存于在结点A处。若想使用必须定义有关继承属性来传递之。但在移进—归约分析不适合继承属性的计算！

修改文法

❑所有产生式

$S \rightarrow L := E$

$E \rightarrow E + E$

$E \rightarrow (E)$

$E \rightarrow L$

$L \rightarrow Elist\ ]$

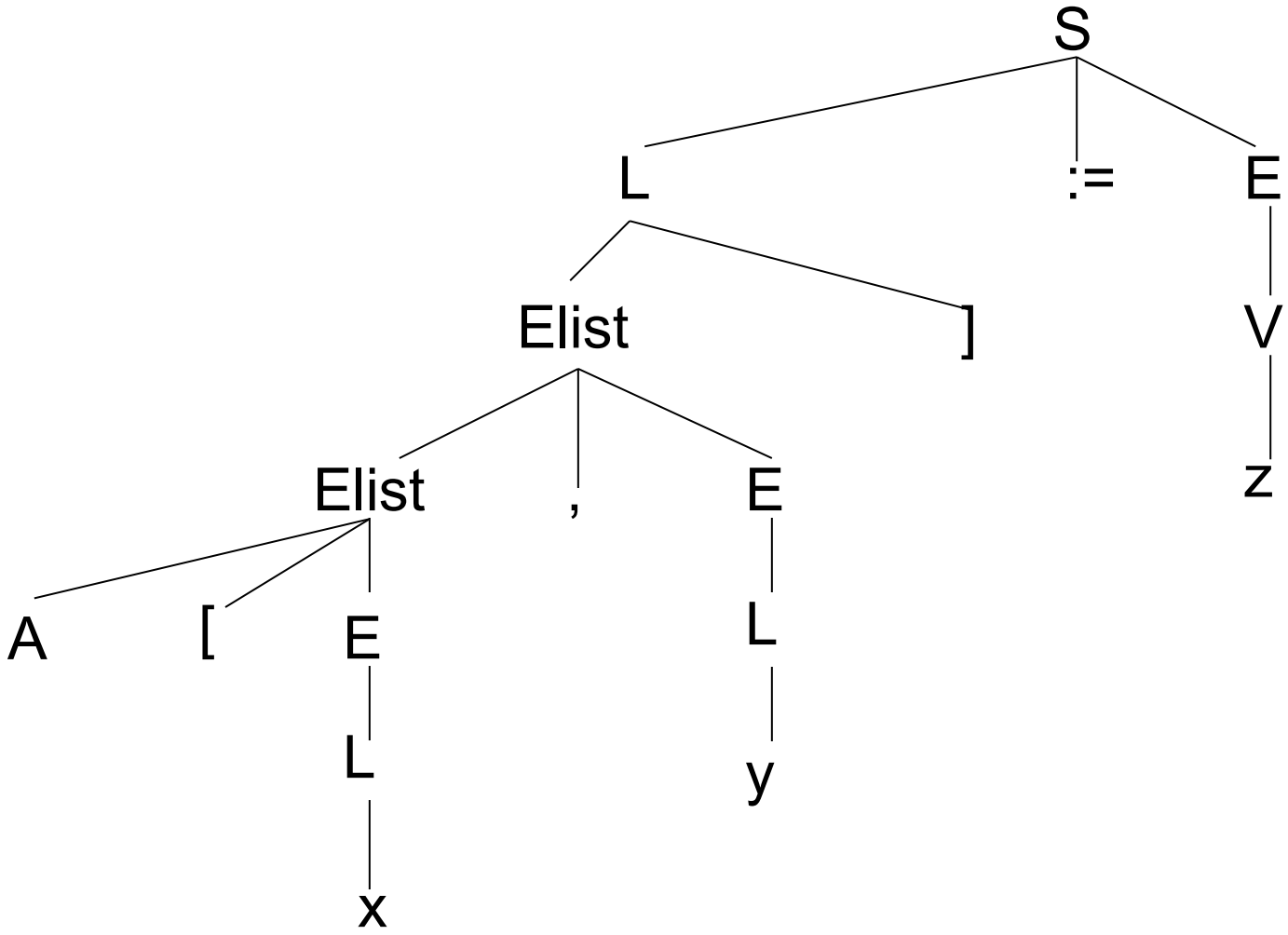
$L \rightarrow \mathrm{id}$

$Elist \rightarrow Elist, E$

$Elist \rightarrow \mathrm{id}\ [\ E$

修改文法，使数组名id成为Elist的子结点（类似于前面的类型声明），从而避免继承属性的出现

A[x,y] := z的分析树



相关符号属性定义：

L.place，L.offset ：

- 若L是简单变量，L.place为其“值”的存放场所，而L.offset为空（null）；
- 当L表示数组元素时，L.place是其地址的“常量值”部分；而此时L.offset为数组元素地址中可变部分的“值”存放场所，数组元素的表示为：
  L.place [L.offset]

相关符号属性定义：

Elist.place：存放“可变部分”值(下标计算值)的地址

Elist.array：数组名条目的指针，比如可以查询base

Elist.ndim：当前处理的维数

limit(array, j)：第j维的大小

width(array)：数组元素的宽度

invariant(array)：静态可计算的值，即紫书7.4公式

数组元素的翻译

- 翻译时重点关注三个表达式：
  - $Elist \rightarrow \text{id} [E$ ： 计算第1维
  - $Elist \rightarrow Elist_1, E$ ： 传递信息
  - $L \rightarrow Elist]$ ： 计算最终结果

数组元素的翻译

$S \rightarrow L := E$ {if *L.offset* == null then /* *L*是简单变量 */

emit (*L.place*, '= ', *E.place*)

else

/*取数组元素的左值*/

emit (*L.place* , '[', *L.offset*, ']', '=', *E.place*) }

数组元素的翻译

Elist → id [*E* {*Elist.place* = *E.place*;
/*第一维下标*/
Elist.ndim = 1;
Elist.array = id.*place* }

数组元素的翻译

$Elist \rightarrow Elist_1, E$

{

$t = newTemp();$

/*维度增加1*/

$m = Elist_1.ndim + 1;$

/* 第m维的大小*/

$n_m = limit(Elist_1.array, m);$

/*计算公式7.6 $e_{m-1} * n_m$ */

$emit\ (t, '=', Elist_1.place, '*', n_m);$

/*计算公式7.6 $e_m = e_{m-1} * n_m + i_m$*/

$emit\ (t, '=', t, '+', E.place);$

$Elist.array = Elist_1.array;$

$Elist.place = t;$

$Elist.ndim = m$

}

数组元素的翻译

$L \rightarrow Elist\ ]\ \{\ L.place = newTemp();$

/*获取数组元素地址的常量值*/

$emit\ (L.place, \text{‘=’}, invariant\ (Elist.array))\ );$

$L.offset = newTemp();$

/*获取数组元素地址的可变部分*/

$emit\ (L.offset, \text{‘=’}, Elist.place, \text{‘*’}, width(Elist.array))\}$

数组元素的翻译

$L \rightarrow$ id {*L.place* = id.*place*; *L.offset* = null }

$E \rightarrow L$ { if *L.offset* == null then /* *L*是简单变量 */

E.place = *L.place*

else begin *E.place* = *newTemp*();

emit (*E.place*, '=', *L.place*, '[', *L.offset*, ']') end }

$E \rightarrow E_1 + E_2$ {*E.place* = *newTemp*();

emit (*E.place*, '=', E_1.*place* , '+', E_2.*place*) }

$E \rightarrow (E_1)$ {*E.place* = E_1.*place* }

其他翻译同前

数组元素的翻译-举例

- 数组A的定义为：A[1…10, 1…20] of integer
- 数组的下界为1，即low为1
- 为赋值语句 x := A[y, z]生成中间代码

举例： x := A[y, z]

L.place = x
L.offset = null
|
x

A[1…10, 1…20] of integer

举例： x := A[y, z]

L.place = x
L.offset = null

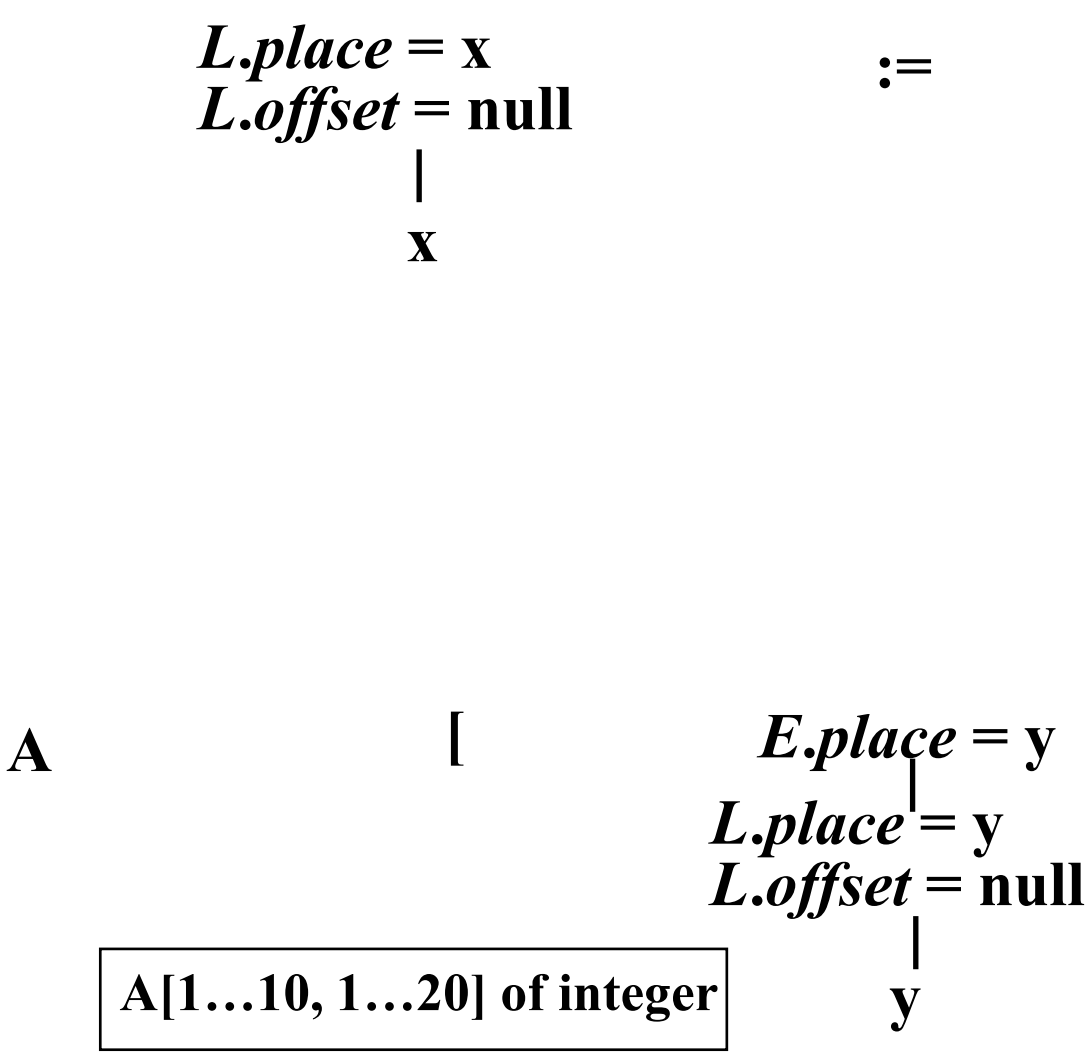
:=

|

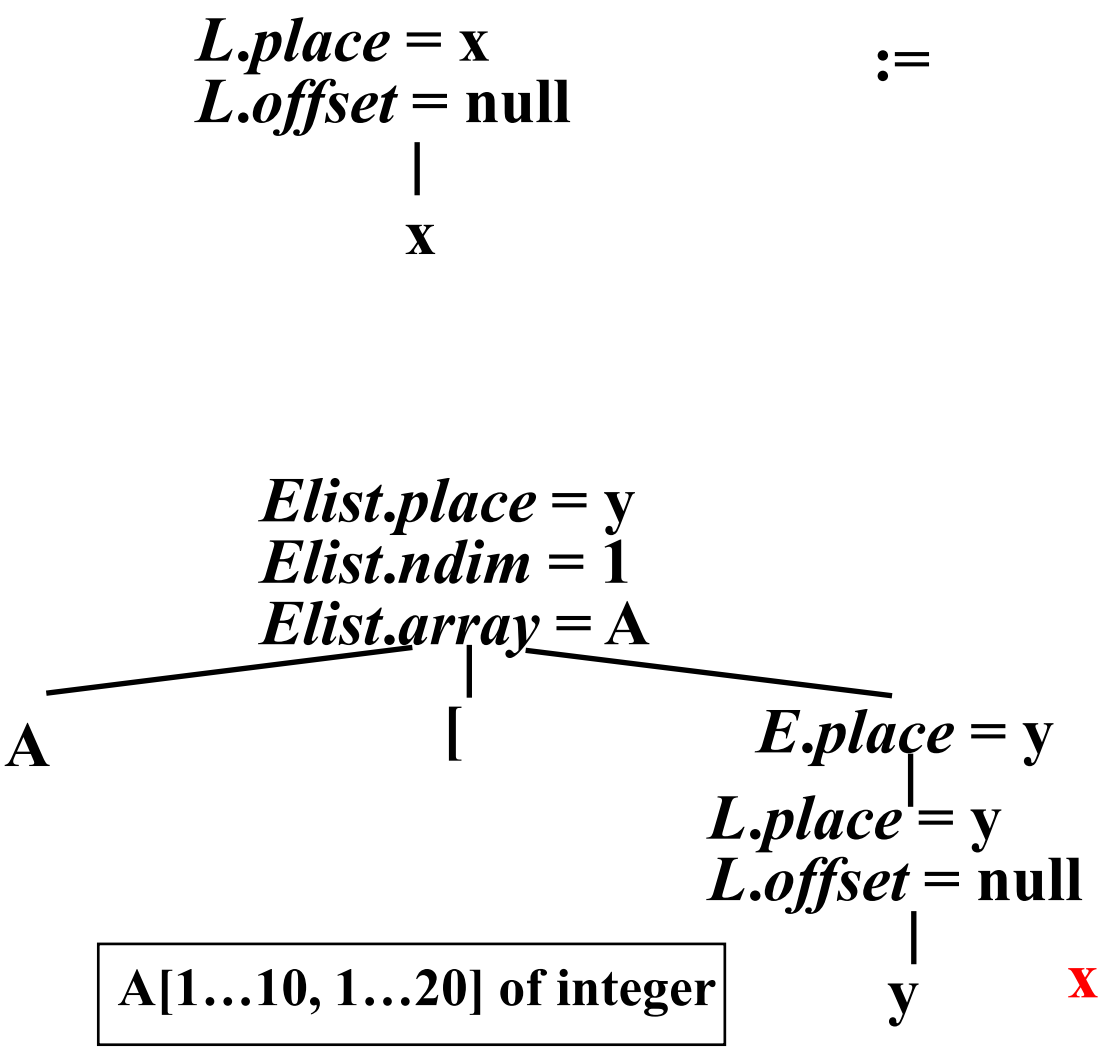
x

A[1…10, 1…20] of integer

举例： x := A[y, z]

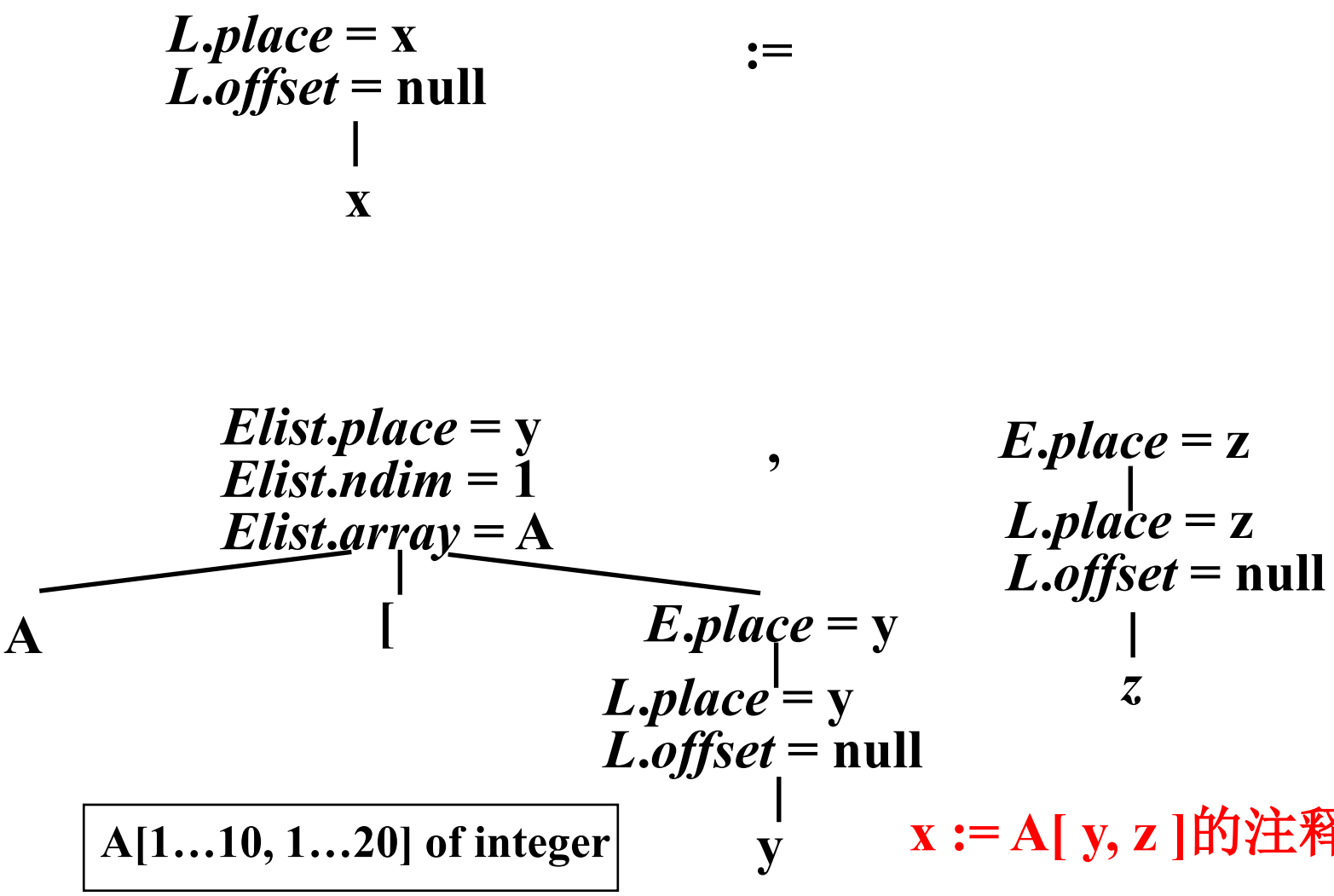

L.place = x
L.offset = null
x
:=
A
[
E.place = y
L.place = y
L.offset = null
y
A[1…10, 1…20] of integer

举例： x := A[y, z]



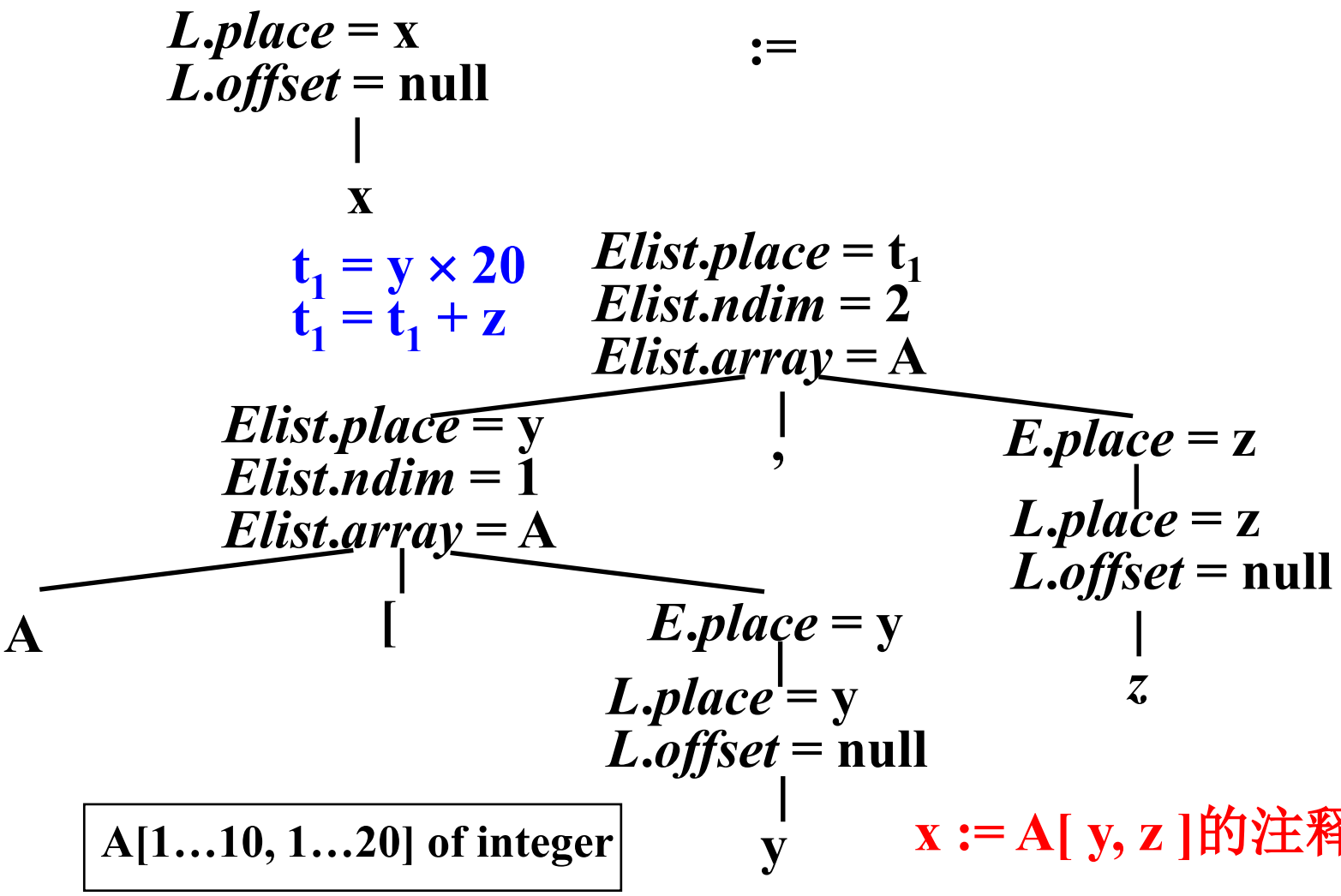
x := A[y, z]的注释分析树

举例： x := A[y, z]



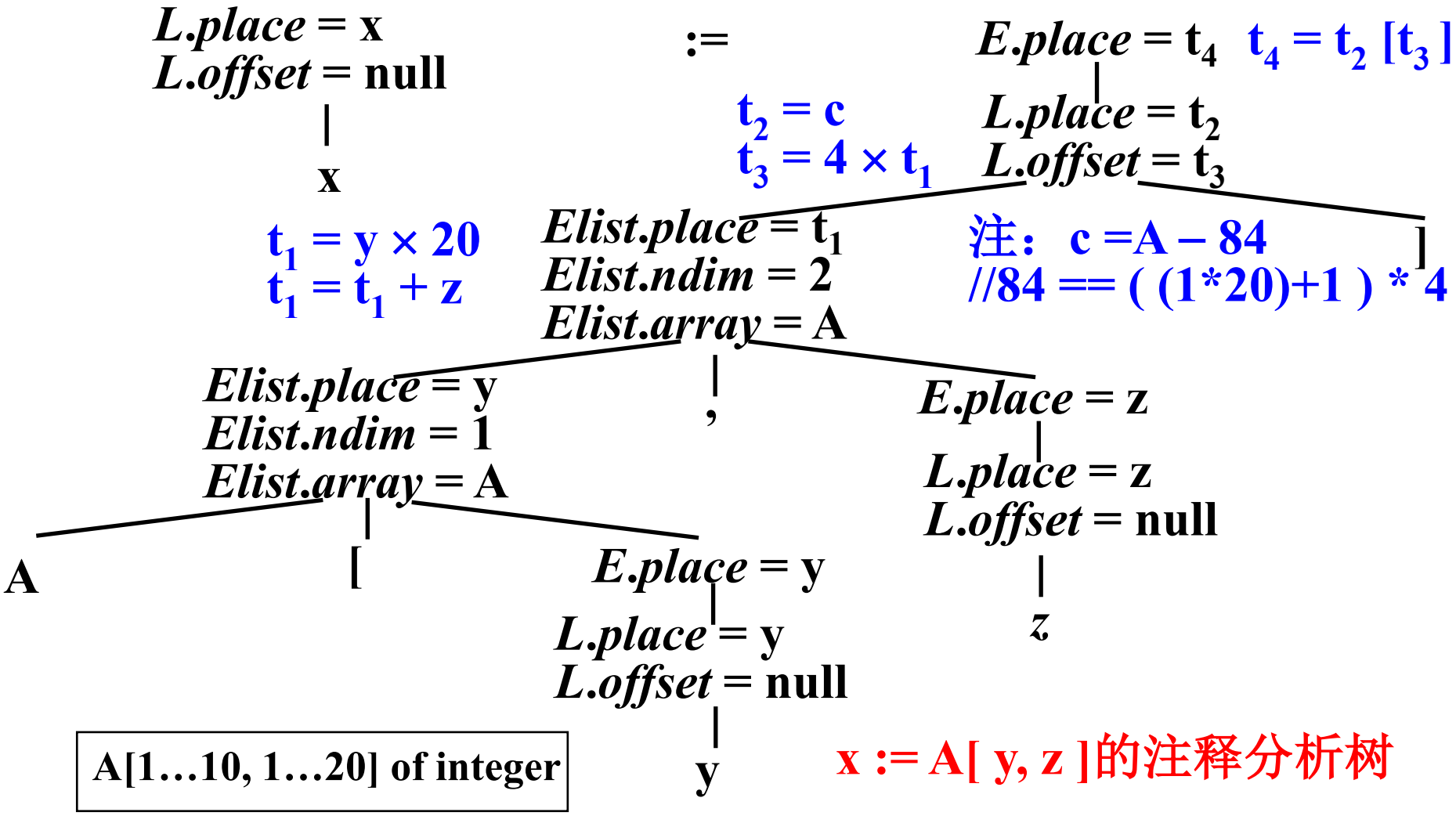
x := A[y, z]的注释分析树

举例： x := A[y, z]



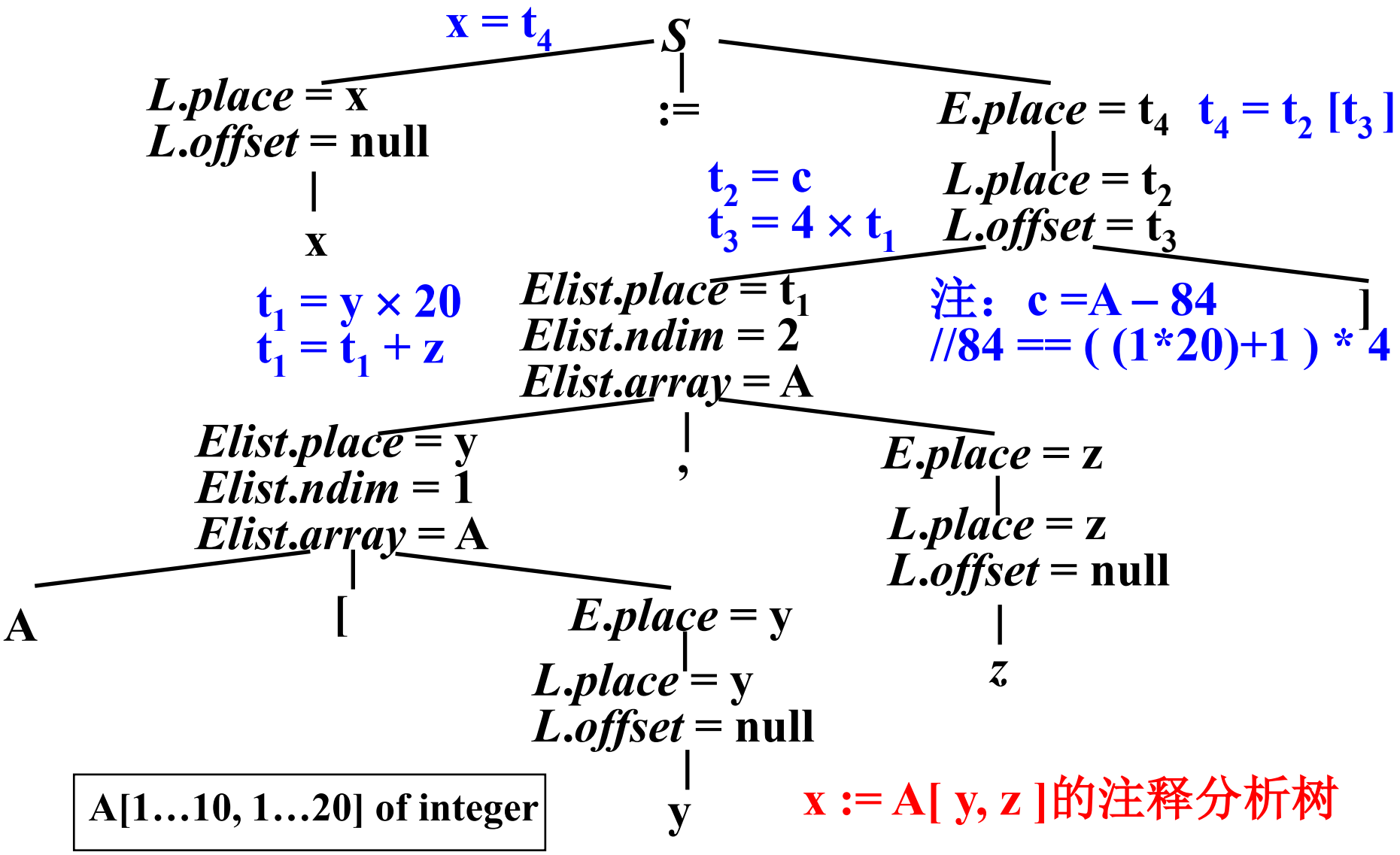
x := A[y, z]的注释分析树

举例： x := A[y, z]



x := A[y, z]的注释分析树

举例： x := A[y, z]



x := A[y, z]的注释分析树

举例：A[i, j] := B[i, j] * k

- 数组A：A[1..10, 1..20] of integer;

  数组B：B[1..10, 1..20] of integer;

  w ：4 （integer）

- TAC如下：

  (1) t_1 := i * 20

  (2) t_1 := t_1 + j

  (3) t_2 := A - 84 // 84 == ((1*20)+1) * 4

  (4) t_3 := t_1 * 4 // 以上A[i, j]的（左值）翻译

举例：A[i, j] := B[i, j] * k

TAC如下（续）：

(5) t_4 := i * 20

(6) t_4 := t_4 + j

(7) t_5 := B - 84

(8) t_6 := t_4 * 4

(9) t_7 := t_5[t_6]

//以上计算B[i,j]的右值

TAC如下（续）：

(10) t_8 := t_7 * k

//以上整个右值表达

//式计算完毕

(11) t_2[t_3] := t_8

// 完成数组元素的赋值

与类型相关的中间代码生成

中国科学技术大学
University of Science and Technology of China

- 符号表的组织
- 声明语句的翻译
- 数组寻址的翻译
- **类型分析的其他应用**

类型分析的其他应用

- 类型等价
  - 结构等价和名字等价
- 类型检查
  - 语法制导翻译方案实现
  - 函数和算符的重载
- 类型转换

结构等价(Structurally equivalence)

- 两个类型表达式完全相同（当无类型名时）

```
type link = ↑cell;
var    next  : link;
       last  : link;
       p     : ↑cell;
       q, r  : ↑cell;
```

- 两个类型表达式完全相同（当无类型名时）
  - 类型表达式树一样

```
type link = ↑cell;
var   next  : link;
      last  : link;
      p     : ↑cell;
      q, r  : ↑cell;
```

结构等价(Structurally equivalence)

- 两个类型表达式完全相同（当无类型名时）
  - 类型表达式树一样
  - 相同的类型构造符作用于相同的子表达式

```
type link = ↑cell;

var   next  : link;
      last  : link;
      p     : ↑cell;
      q, r  : ↑cell;
```

结构等价(Structurally equivalence)

- 两个类型表达式完全相同（当无类型名时）
- 有类型名时，用它们所定义的类型表达式代换它们，所得表达式完全相同(类型定义无环时)

```
type link = ↑cell;
var   next  : link;
      last  : link;
      p     : ↑cell;
      q, r  : ↑cell;
```

这里隐藏了递归检查，因此暂时不考虑有环的情况

next, last, p, q和r结构等价

结构等价(Structurally equivalence)

function sequiv(s, t) : boolean

{if s 和 t 是相同的基本类型then
  return true
else if $s == array(s_1, s_2)$ and $t == array(t_1, t_2)$ then
  return sequiv(s_1, t_1) and sequiv(s_2, t_2)
else if $s == s_1 \times s_2$ and $t == t_1 \times t_2$ then
  return sequiv(s_1, t_1) and sequiv(s_2, t_2)
else if $s == pointer\ (s_1)$ and $t == pointer(t_1)$ then
  return sequiv(s_1, t_1)
else if s == $s_1 \rightarrow s_2$ and t == $t_1 \rightarrow t_2$ then
  return squiv(s_1, t_1) and sequiv(s_2, t_2)
else return false
}

名等价(name equivalance)

- 把每个类型名看成是一个可区别的类型
- 两个类型表达式名字等价当且仅当
  - 它们是相同的基本类型
  - 不进行名字代换就能结构等价

名等价(name equivalance)

- 把每个类型名看成是一个可区别的类型
- 两个类型表达式名字等价当且仅当
  - 它们是相同的基本类型
  - 不进行名字代换就能结构等价

```
type link = ↑cell;          类型表达式

var     next    : link;     link
        last    : link;     link
        p       : ↑cell;    pointer(cell)
        q, r    : ↑cell;    pointer(cell)
```

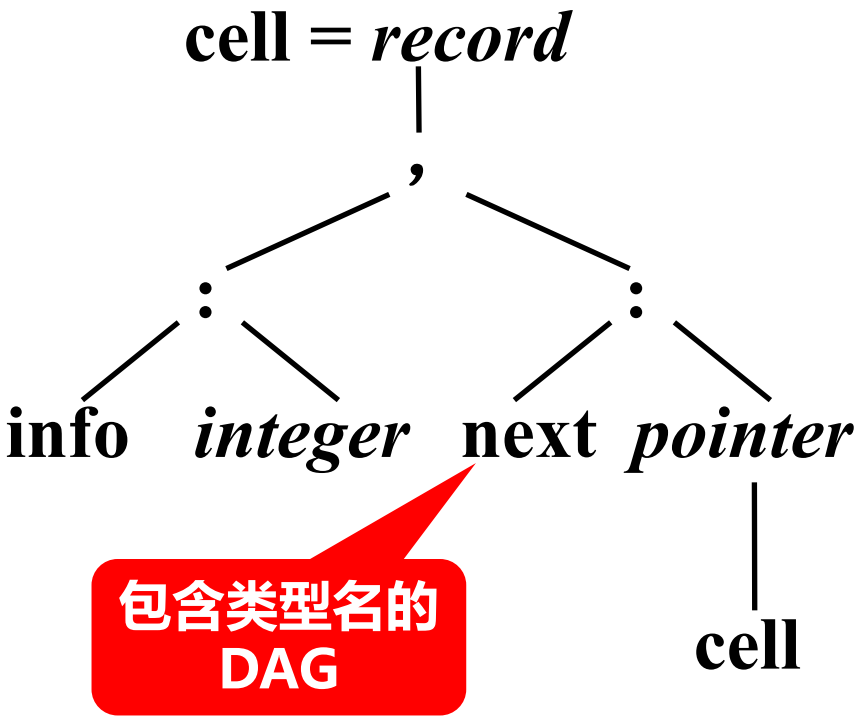
next和last名字等价

p，q和r名字等价

递归定义的类型

- Where： Linked Lists, Trees, etc
- How: records containing pointers to similar records

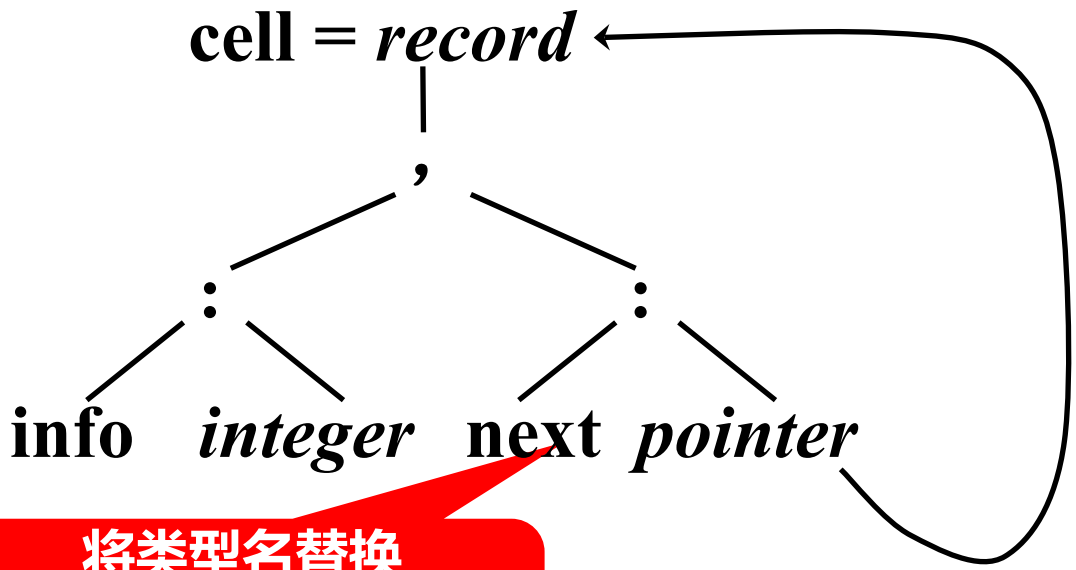
```
type link = ↑ cell ;
cell = record
          info : integer ;
          next : link
end;
```



递归定义的类型

- Where： Linked Lists, Trees, etc
- How: records containing pointers to similar records

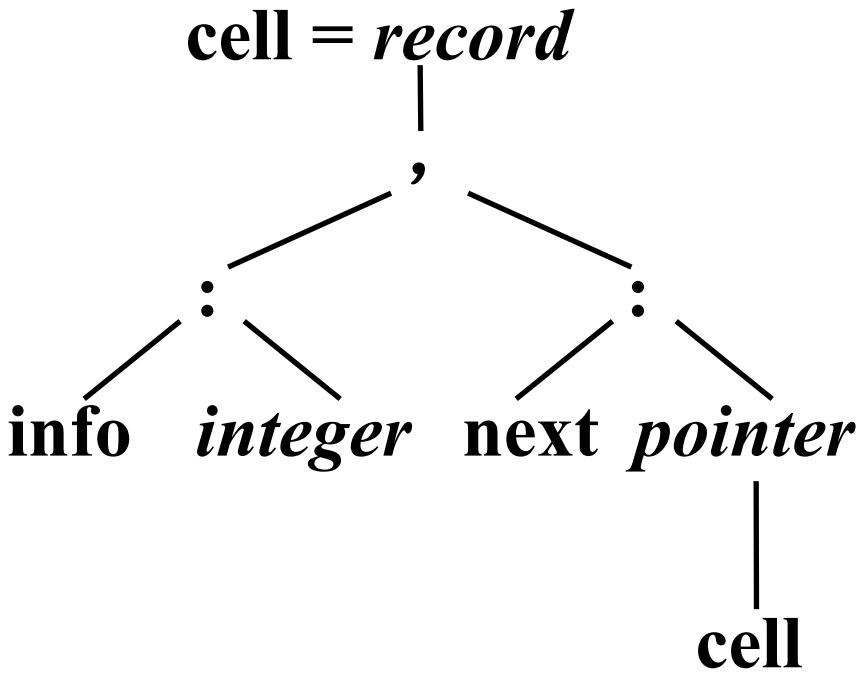
```
type link = ↑ cell ;
cell = record
          info : integer ;
          next : link
end;
```



将类型名替换
引入环，结构等价判定
有可能不终止

C语言中的递归定义的类型

C语言对除记录（结构体）以外的所有类型使用结构等价，而记录类型用的是名字等价，以避免类型图中的环



例题

在X86/Linux机器上，编译器报告最后一行有错误：

incompatible types in return

```
typedef int A1[10];                  | A2 *fun1( ) {
typedef int A2[10];                  |     return(&a);
A1 a;                                | }
typedef struct {int i;}S1;           | S2  fun2( ) {
typedef struct {int i;}S2;           |     return(s);
S1 s;                                | }
```

在C语言中，数组和结构体都是构造类型，为什么上面第2个函数有类型错误，而第1个函数却没有?

类型分析的其他应用

- 类型等价
  - 结构等价和名字等价
- 类型检查
  - 语法制导翻译方案实现
  - 函数和算符的重载
- 类型转换

一个简单的语言

$P \rightarrow D\ ;\ S$

$D \rightarrow D\ ;\ D \mid \text{id}\ :\ T$

$T \rightarrow \text{boolean} \mid \text{integer} \mid \text{array}\ [\text{num}\ ]\ \text{of}\ T \mid$

$\uparrow T \mid T\ \text{`}\rightarrow\text{'}\ T$

$S \rightarrow \text{id}\ :=\ E \mid \text{if}\ E\ \text{then}\ S \mid \text{while}\ E\ \text{do}\ S \mid S\ ;\ S$

$E \rightarrow \text{truth} \mid \text{num} \mid \text{id} \mid E\ \text{mod}\ E \mid E\ [\ E\ ] \mid$

$E\uparrow \mid E\ (E\ )$

例

```
i : integer;
j : integer;
j := i mod 2000
```

类型检查——声明语句

$D \rightarrow D ; D$

$D \rightarrow \text{id} : T \ \{addtype\ (\text{id}.entry, T.type)\}$

addtype：把类型信息填入符号表

类型检查——声明语句

$D \rightarrow D ; D$

$D \rightarrow \text{id} : T \ \{addtype\ (\text{id}.entry, T.type)\}$

$T \rightarrow \text{boolean} \quad \{T.type = boolean\}$

$T \rightarrow \text{integer} \quad \{T.type = integer\}$

$T \rightarrow \uparrow T_1 \quad \{T.type = pointer(T_1.type)\}$

类型检查——声明语句

$D \rightarrow D; D$

$D \rightarrow \text{id} : T \ \{addtype\ (\text{id}.entry, T.type)\}$

$T \rightarrow \text{boolean} \quad \{T.type = boolean\}$

$T \rightarrow \text{integer} \quad \{T.type = integer\}$

$T \rightarrow \uparrow T_1 \quad \{T.type = pointer(T_1.type)\}$

$T \rightarrow \text{array [num] of } T_1$

$\{T.type = array(\text{num}.val, T_1.type)\}$

类型检查——声明语句

$D \rightarrow D; D$

$D \rightarrow \text{id} : T \ \{addtype\ (\text{id}.entry, T.type)\}$

$T \rightarrow \text{boolean} \quad \{T.type = boolean\}$

$T \rightarrow \text{integer} \quad \{T.type = integer\}$

$T \rightarrow \uparrow T_1 \quad \{T.type = pointer(T_1.type)\}$

$T \rightarrow \text{array [num] of } T_1$

$\{T.type = array(\text{num}.val, T_1.type)\}$

$T \rightarrow T_1 \text{ ‘}\rightarrow\text{’ } T_2 \quad \{T.type = T_1.type \rightarrow T_2.type\ \}$

类型检查——表达式

$E \rightarrow \text{truth}$ $\{E.type = boolean\}$

$E \rightarrow \text{num}$ $\{E.type = integer\}$

$E \rightarrow \text{id}$ $\{E.type = lookup(\text{id}.entry)\}$

类型检查——表达式

$E \rightarrow$ truth $\quad \{E.type = boolean\}$

$E \rightarrow$ num $\quad \{E.type = integer\}$

$E \rightarrow$ id $\quad \{E.type = lookup(\text{id}.entry)\}$

$E \rightarrow E_1 \bmod E_2$

$\{E.type = \text{if } E_1.type == integer \text{ and}$

$E_2.\ type == integer \text{ then } integer$

$\text{else } type\_error\ \}$

类型检查——表达式

$$E \rightarrow E_1\ [E_2\ ]\ \{E.type = \text{if } E_2.\ type == integer \text{ and}$$

$$E_1.\ type == array(s, t) \text{ then } t$$

$$\text{else } type\_error\ \}$$

类型检查——表达式

$$E \rightarrow E_1\ [E_2\ ]\ \{E.type = \text{if } E_2.\ type == integer \text{ and } E_1.\ type == array(s, t) \text{ then } t \text{ else } type\_error\ \}$$

$$E \rightarrow E_1\uparrow\ \{E.type = \text{if } E_1.type == pointer(t) \text{ then } t \text{ else } type\_error\ \}$$

类型检查——表达式

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

$E \rightarrow E_1\uparrow$ {*E.type* = if E_1.*type* == *pointer*(*t*) then *t*
else *type_error* }

$E \rightarrow E_1\,(E_2\,)$ {*E*. *type* = if E_2 . *type* == *s* and
E_1. *type* == $s \rightarrow t$ then *t*
else *type_error* }

类型检查——语句

$$S \rightarrow \text{id} := E \{ \text{if } (id.type == E.type \;\&\&\; E.type \in \{boolean, integer\}) \; S.type = void;$$

$$\text{else } S.type = type\_error;\}$$

类型检查——语句

$S \rightarrow \text{id} := E$ { if ($id.type == E.type$ && $E.type \in$ {*boolean*, *integer*}) $S.type = void$;

else $S.type = type\_error$;}

$S \rightarrow \text{if } E \text{ then } S_1$ {$S.type = \text{if } E.type == boolean$ then $S_1.type$ else $type\_error$ }

类型检查——语句

$S \rightarrow$ **while** E **do** S_1

$$\{S.type = \text{if } E.type == boolean \text{ then } S_1.type$$

$$\text{else } type\_error \}$$

$S \rightarrow$ **while** E **do** S_1

$\{S.type = \text{if } E.type == boolean \text{ then } S_1.type$

else $type\_error\}$

$S \rightarrow S_1; S_2$

$\{S.type = \text{if } S_1.type == void \text{ and}$

$S_2.type == void \text{ then } void$

else $type\_error\}$

类型检查——程序

$$P \rightarrow D; S$$

$$\{P.\ type = \text{if } S.type == void \text{ then } void$$

$$\text{else } type\_error \}$$

类型分析的其他应用

- 类型等价
  - 结构等价和名字等价
- 类型检查
  - 语法制导翻译方案实现
  - 函数和算符的重载
- 类型转换

函数和算符的重载

❑重载符号

- ❖有多个含义，但在每个引用点的含义都是唯一的

❑例如：

- ❖加法算符+可用于不同类型，"+"是多个函数的名字，而不是一个多态函数的名字

❑重载的消除

- ❖在重载符号的引用点，其含义能确定到唯一

子表达式的可能类型集合

❑例 Ada语言的声明:

function “*” (i, j : integer) return complex;

function “*” (x, y : complex) return complex;

使得算符*重载，可能的类型包括:

integer × integer → integer --这是预定义的类型

integer × integer → complex

complex × complex → complex

2 * (3 * 5)

(3 * 5) * z z是复型

子表达式的可能类型集合

- 以函数应用为例，考虑类型检查
  - 在每个表达式都有唯一的类型时，函数应用的类型检查是：

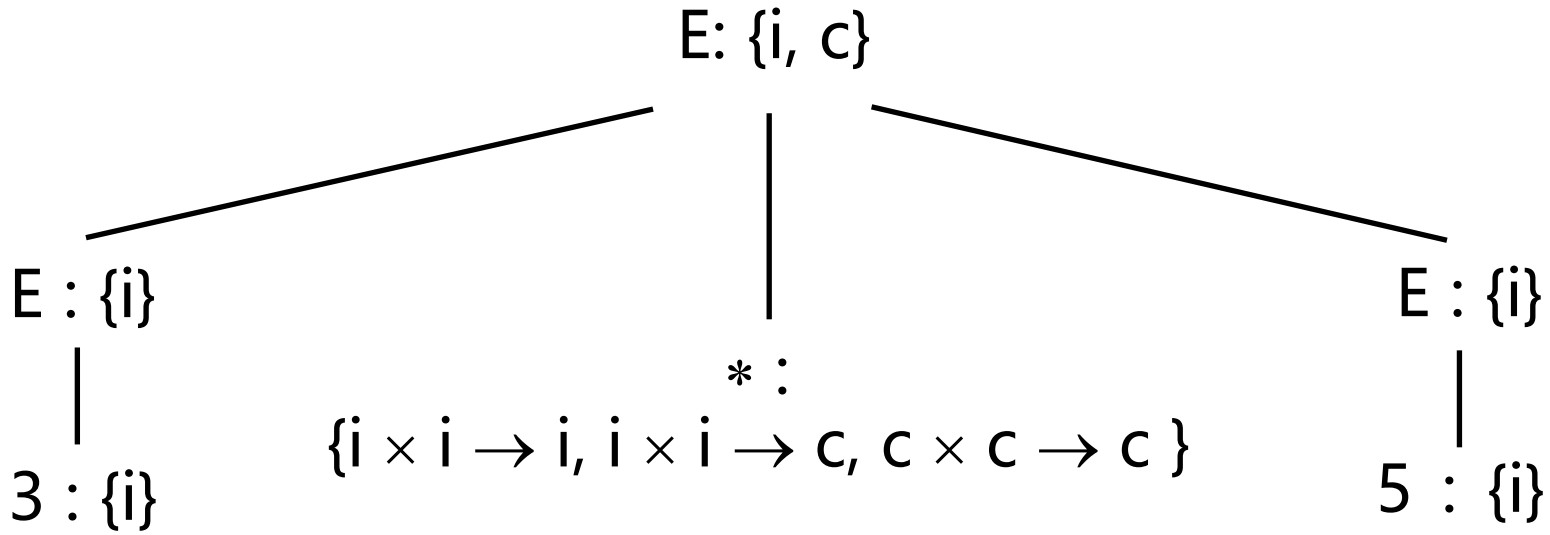
$E \rightarrow E_1(E_2)$ { E.type = if E_2.type == s and E_1.type == $s \rightarrow t$ then t else type_ error }

  - 确定表达式可能类型的集合（类型可能不唯一）

产 生 式	语 义 规 则
$E' \rightarrow E$	E′.types = E. types
$E \rightarrow id$	E. types = lookup(id. entry)
$E \rightarrow E_1(E_2)$	E. types = {t | E_2. types中存在一个s，使得s → t属于E_1.types }

子表达式的可能类型集合

- 例：表达式3 * 5可能的类型集合



缩小可能类型的集合

$E' \rightarrow E$ {E'.types = E. types

E.unique = if E'.types = {t} then t else error}

$E \rightarrow id$ {E.types = lookup(id.entry)}

缩小可能类型的集合

$E' \rightarrow E$ {E'.types = E. types

E.unique = if E'.types = {t} then t else error}

$E \rightarrow id$ {E.types = lookup(id.entry)}

$E \rightarrow E_1(E_2)$ {E.types = {s' | ∃ s ∈ E_2.types and

$s \rightarrow s' \in E_1$.types}

t = E.unique

$S = \{s \mid s \in E_2.types \text{ and } S \rightarrow t \in E_1.types\}$

E_2.unique = if S ={s} then S else error

E_1.unique = if S = {s} then S→t else error

类型分析的其他应用

- 类型等价
  - 结构等价和名字等价
- 类型检查
  - 语法制导翻译方案实现
  - 函数和算符的重载
- **类型转换**

类型转换

❑例 x = y + i * j
（x和y的类型是real，i和j的类型是integer）

中间代码

t_1 = i **int×** j
t_2 = **inttoreal** t_1
t_3 = y **real+** t_2
x = t_3

int× 和 real+ 不是类型转换，而是算符

目标机器的运算指令是区分整型和浮点型的
高级语言中的重载算符=>中间语言中的多种具体算符

类型转换的处理

- 以$E \rightarrow E_1 + E_2$为例说明
  - 判断E1 和E2的类型，看是否要进行类型转换;若需要，则分配存放转换结果的临时变量并输出类型转换指令

```
{ E.place = newTemp();
if (E1.type == integer && E2.type == integer) then begin
  emit (E.place, '=', E1.place, 'int+', E2.place);
  E.type = integer
end
else if (E1.type == integer && E2.type == real) then
  begin
  u = newTemp(); emit (u, '=', 'inttoreal', E1.place);
  emit (E.place, '=', u, 'real+', E2.place); E.type = real;
end
  . . .}
```

11.15作业

- 请参考本节中所讲的数组元素寻址翻译方法，完成以下代码的翻译：
  - A[x, y, z] = p
  - 此处， A是一个数组，定义为A[1..10, 1..10, 1..20] of integer; 每一个integer占4字节
  - 要求画出语法分析树，标注综合属性及其计算值，并给出对应的三地址码序列

中国科学技术大学
University of Science and Technology of China

USTC, CHINA
ADSLAB
先进数据系统实验室

《编译原理与技术》
中间代码生成 III

Done