

## 第十章 语义分析和代码生成

- 10. 1 语义分析的概念
- 10. 2 栈式抽象机及其汇编指令
- 10. 3 声明的处理
- 10. 4 表达式的处理
- 10. 5 赋值语句的处理
- 10. 6 控制语句的处理
- 10. 7 过程调用和返回



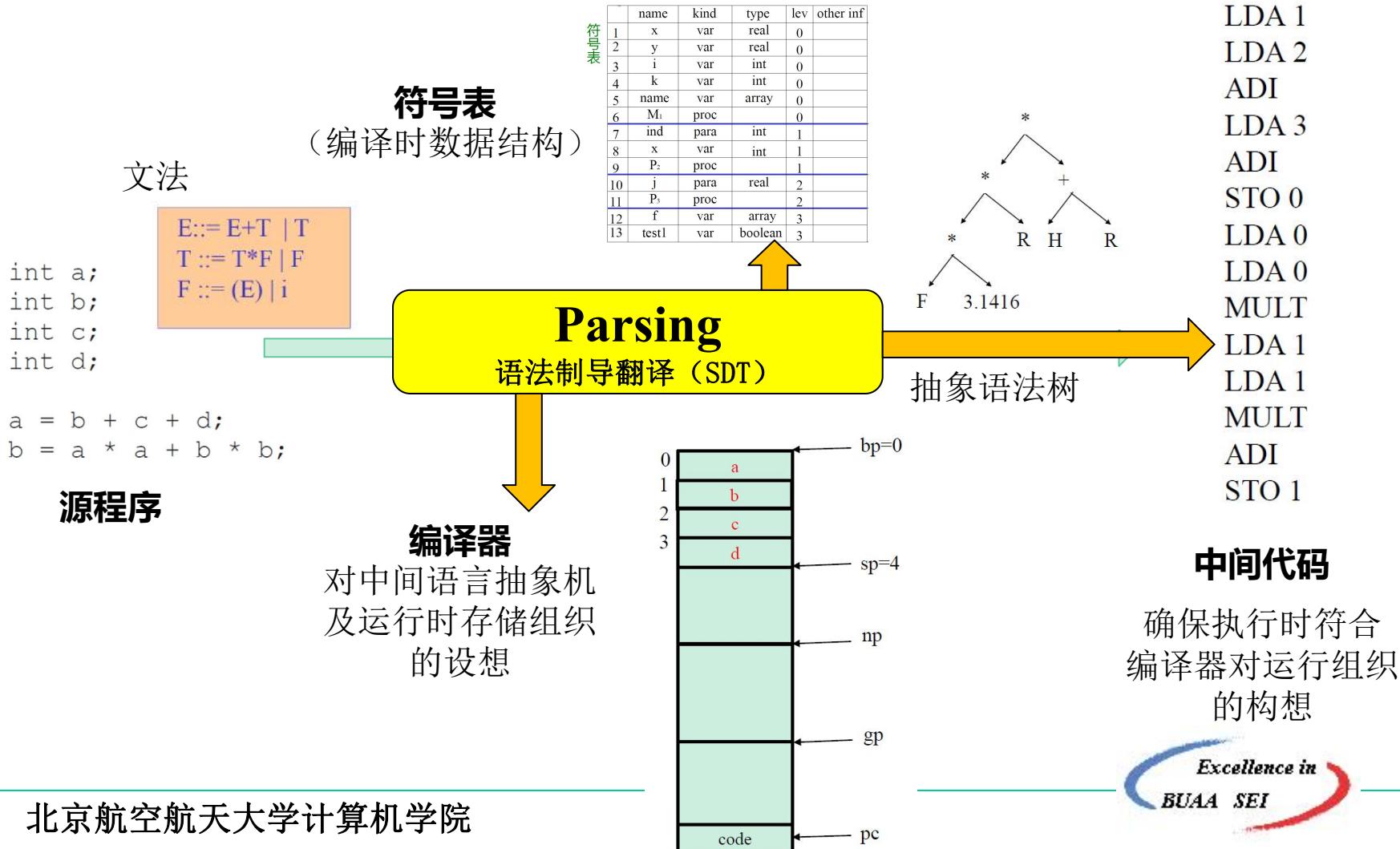
编译过程是指将高级语言程序翻译为等价的**目标程序**的过程。

习惯上是将编译过程划分为5个基本阶段：



前两节课（符号表+运行时存储）：为代码生成做“需求分析”

上一节课（语法制导翻译）：为代码生成做技术准备



## TG

**翻译文法**  
加入动作符号

@action

## ATG

**属性翻译文法**  
加入动作符号及两  
类“参数”

综合属性  
集成属性

## L-ATG

**LL(1)属性翻译文法**  
无二义、无左递归  
消除回溯

综合属性  
集成属性

## SL-ATG

**简单赋值的  
属性翻译文法**  
求值规则只能  
使用简单赋值

复杂求值函数  
放入 @f 中

a@**a+b**@**b\*c**@**c@a\***@**a+**

- 1.<说明> → Type<sub>↑t</sub> id<sub>↑n</sub> @set\_table<sub>↓t1,n1</sub> <变量表><sub>↓t2</sub>      t<sub>2,t1 := t; n<sub>1 := n;</sub></sub>
- 2.<变量表><sub>↓t2</sub> → , id<sub>↑n</sub> @set\_table<sub>↓t1,n1</sub> <变量表><sub>↓t3</sub>      t<sub>3,t1 := t<sub>2; n<sub>1 := n;</sub></sub></sub>
- 3.<变量表><sub>↓t2</sub> → ε

## 假定：

- 源语言：通用的过程语言
- 生成代码：栈式抽象机的（伪）汇编程序
- 翻译方法：自顶向下的属性翻译
- 语法成分翻译子程序参数设置：
  - 继承属性为值形参
  - 综合属性为变量形参
- 语法成分翻译动作子程序参数设置：
  - 继承属性为值形参
  - 综合属性不设形参，而作为动作子程序的返回值（由 RETURN语句返回）

## 10.1 语义分析的概念

- 1、上下文有关分析：即标识符的作用域
- 2、类型的一致性检查
- 3、语义处理：

**声明语句：**其语义是声明变量的类型等，并不要求做其他的操作。

编译程序的工作是填符号表，登录名字的特征信息，分配存储。

**执行语句：**语义是要做某种操作。

语义处理的任务：按某种操作的目标结构生成代码。



用上下文无关文法只能描述语言的语法结构，而不能描述其语义。

例如，对于有嵌套子程序结构的程序段：

**BEGIN ... BEGIN  $\alpha$ INT I  $\beta$  I END ... I ... END**

若存在文法规则：  $VAR ::= I$

**BEGIN ... <BLOCK> ... I ... END**



**BEGIN ...  $\delta$  VAR ... END**

第一次I的归约正确  
第二次I的归约错误

$\delta \in V^*$ 且不包含变量I的声明

文法规则应改为：  $INT I \beta \text{ VAR } ::= INT I \beta I$

然而上下文有关文法不仅构造困难，而且其分析器十分复杂，分析效率又低，显然是不实用的

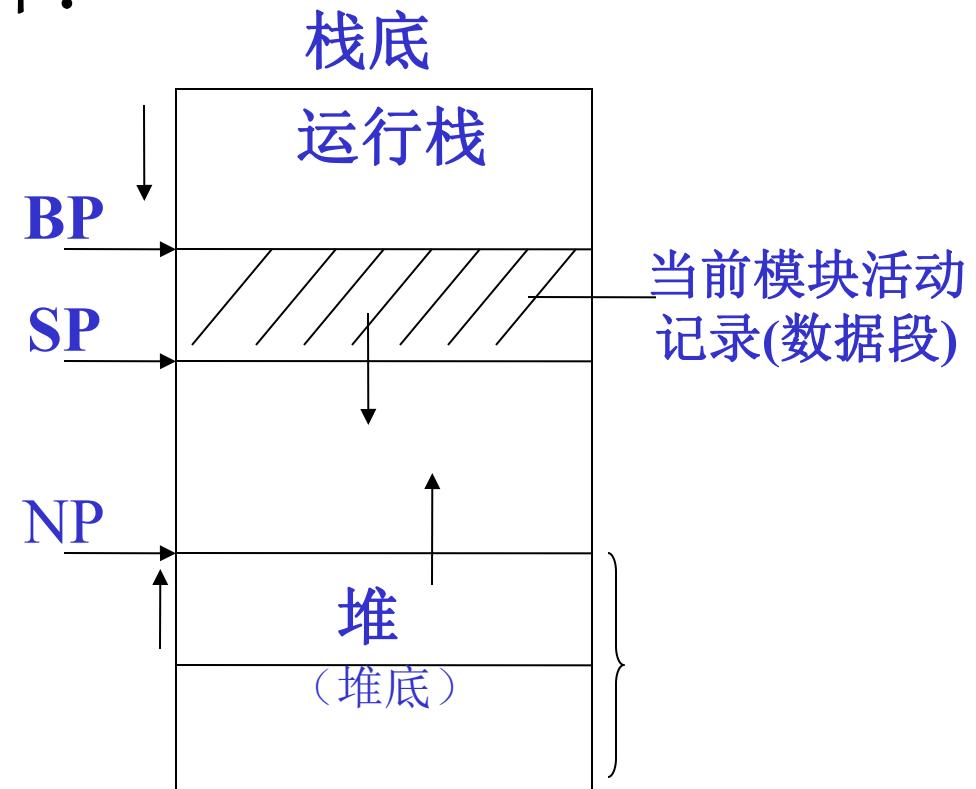
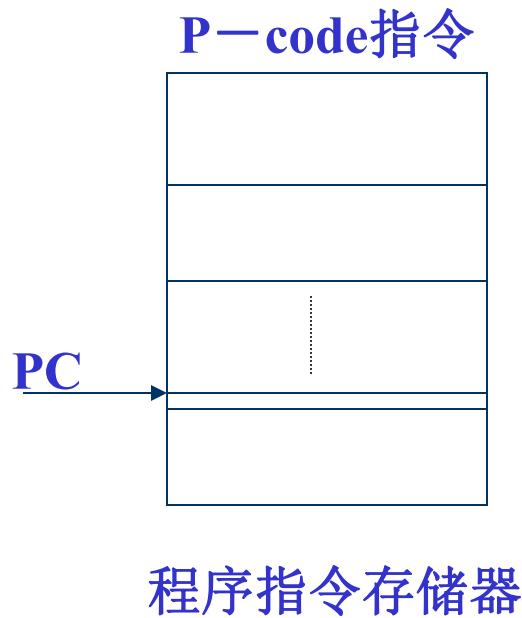
因此，通常我们把与语义相关的上下文有关信息填入符号表中，并通过查符号表中的这些信息来分析程序的语义是否正确

## 10.2 栈式抽象机及其汇编指令

栈式抽象机：由三个存储器、一个指令寄存器和多个地址寄存器组成。

存储器：  
  {  
    数据存储器  （存放AR的运行栈）  
    操作存储器  （操作数栈）  
    指令存储器

计算机的存储大致情况如下：



例：

$a := b + c;$



LDA (a)  
LOD b  
LOD c  
ADD  
STN

## 栈式抽象机指令代码如下：

指令名称	操作码	地址	指令意义
加载指令	<b>LOD</b>	<b>D</b>	将 D 的内容→栈顶
立即加载	<b>LDC</b>	常量	常量→栈顶
地址加载	<b>LDA</b>	(D)	变量 D 的地址→栈顶
存储	<b>STO</b>	<b>D</b>	存入 栈顶内容 → 变量D
间接存	<b>ST</b>	@D	将栈顶内容→D 所指单元
间接存	<b>STN</b>		将栈顶内容→次栈顶所指单元
加	<b>ADD</b>		栈顶和次栈顶内容相加，结果留栈顶
减	<b>SUB</b>		次栈顶内容减栈顶内容
乘	<b>MUL</b>		
.....			

# Compiler

指令名称	操作码	地址	指令意义
等于比较	EQL		
不等比较	NEQ		
大于比较	GRT		次栈顶内容与栈顶内容比较， 结果（1 或 0）留栈顶
小于比较	LES		
大于等于	GTE		
小于等于	LSE		
逻辑与	AND		
逻辑或	ORL		
逻辑非	NOT		
转子	JSR	lab	
分配	ALC	M	在运行栈顶分配大小为 M 的活动记录区

## 10.3 声明的处理

编译程序的任务：

• 编译程序处理声明语句要完成的主要任务为：

- 1) 分离出每一个被声明的实体，并把它们的名字填入符号表中
- 2) 把被声明实体的有关特性信息尽可能多地填入符号表中

• 对于已声明的实体，在处理对该实体的引用时要做的事情：

- 1) 检查对所声明的实体引用（种类，类型等）是否正确
- 2) 根据实体的特征信息，例如类型，所分配的目标代码地址（可能为数据区单元地址，或目标程序入口地址）生成相应的目标代码

声明有常量声明，变量（包括简单变量，数组变量和记录变量等）和过程（函数）声明等，这里主要讨论常量声明和简单变量、数组声明的处理。

## 声明的两种方式：

- (1) 类型说明符放在变量的前面。如：C语言： int a;  
在填表时已知类型和a的值（名字）：直接填入符号表。
- (2) 类型说明符放在变量的后面，如：Pascal, PL/I, Ada等，需要返填。

如PL/I声明语句：

DECLARE( X, Y(N), YTOTAL) FLOAT;

声明语句的输入文法为：

```
<declaration>→DECLARE ‘(‘<entity list>‘)’<type>
<entity list>→<entity name> | <entity name> , <entity list>
<type>→FIXED | FLOAT | CHAR
```

属性翻译文法为：

```
<declaration>→DECLARE @dec_on↑x ‘(‘<entity list> ‘)’
                <type> ↑t @fix_up↓x, t
<entity list>→<entity name> ↑n @name_defn ↓n
                | <entity name> ↑n , @name_defn ↓n <entity list>
<type> ↑t →FIXED ↑t | FLOAT ↑t | CHAR ↑t
```

## 动作程序

```

<declaration> → DECLARE @dec_on↑x ‘ (’<entity list> ’)
    <type> ↑t @fix_up↓x, t
<entity list> → <entity name>↑n @name_defn↓n
                | <entity name>↑n, @name_defn↓n <entity list>
<type> ↑t → FIXED↑t | FLOAT↑t | CHAR↑t

```

$\text{@dec\_on}_{\uparrow x}$  是把符号表当前可用表项的入口地址（指向符号表入口的指针，或称 表项下标值）赋给属性变量x。

$\text{@name\_defn}_{\downarrow n}$  是将由各实体名所得的n继承属性值，依次填入从x开始的符号表中。

注：显然应有内部计数器或内部指针，指向下一个该填的符号表项。

$\text{@fix\_up}_{\downarrow x, t}$  是将类型信息t 和相应的数据存储区分配地址填入从x位置开始的符号表中。（反填）

当然，如果声明语句中，类型说明符放在头上，就无需“反填”处理了。

## 常量声明:

常量标识符通常被看作是全局名。

```
constant integer  SYMBSIZE := 1024
```

常量声明语句的输入文法为:

```
<const del> → constant <type> <entity> := <const expr>
<type> → real | integer | string
<const expr> → <integer const> | <real const>
                | <string const>
```

## 10.3.1 常量类型声明处理

```
constant integer  SYMBSIZE := 1024
```

常量声明的ATG如下：

$\langle \text{const del} \rangle \rightarrow \text{constant } \langle \text{type} \rangle \uparrow_t \langle \text{entity} \rangle \uparrow_n := \langle \text{const expr} \rangle_{\uparrow c, s}$   
 $\qquad\qquad\qquad @\text{insert}_{\downarrow t, n, c, s};$

$\langle \text{type} \rangle \uparrow_t \rightarrow \text{real } \uparrow_t | \text{integer } \uparrow_t | \text{string } \uparrow_t$

$\langle \text{const expr} \rangle_{\uparrow c, s} \rightarrow \langle \text{integer const} \rangle_{\uparrow c, s} | \langle \text{real const} \rangle_{\uparrow c, s}$   
 $\qquad\qquad\qquad | \langle \text{string const} \rangle_{\uparrow c, s}$

翻译处理过程为：

由该文法产生的一个声明实例为：

**constant integer SYMBSIZE := 1024;**

**<const del>** → **constant <type>**  $\uparrow_t$  **<entity>**  $\uparrow_n$  :=  
**<const expr>**  $\uparrow_{c,s}$  **@insert**  $\downarrow_{t,n,c,s}$  ;

先识别类型 (integer)，将它赋给属性t；然后识别常量名字 (SYMSIZE), 将它赋给属性n；最后识别常量表达式，并将其值赋给c，其类型赋给属性s。

## ★ @insert 的功能是：

- ① 检查声明的类型t 和常量表达式的类型s 是否一致，若不一致，则输出错误信息
  - ② 把名字n，类型t 和常量表达式的值c 填入符号表中

## 10.3.2 简单变量声明处理

```
real x  
integer j  
character ( 20 ) s
```

简单变量声明语句的输入文法为：

```
<svar del> → <type> <entity>  
<type> → real | integer | character (<number>)  
          | logical
```

## 简单变量声明:

### ATG文法:

$\langle \text{svar del} \rangle \rightarrow \langle \text{type} \rangle_{\uparrow t, i} \langle \text{entity} \rangle_{\uparrow n} @\text{svardef}_{\downarrow t, i, n}$

$@\text{allocsv}_{\downarrow i};$

$\langle \text{type} \rangle_{\uparrow t, i} \rightarrow \text{real}_{\uparrow t, i} \mid \text{integer}_{\uparrow t, i} \mid \text{character}_{\uparrow t} (\langle \text{number} \rangle_{\uparrow i})$   
 $\mid \text{logical}_{\uparrow t, i}$

n: 变量名

t: 类型值

i: 该类型变量所需  
数据空间的大小

简单变量声明的例子:

**real x;**

**integer j;**

**character ( 20 ) s ;**

$\langle \text{svar del} \rangle \rightarrow \langle \text{type} \rangle \uparrow_{t,i} \langle \text{entity} \rangle \uparrow_n @\text{svardef}_{\downarrow t,i,n} @\text{allocsv}_{\downarrow i}$

$\langle \text{type} \rangle \uparrow_{t,i} \rightarrow \text{real}_{\uparrow_{t,i}} | \text{integer}_{\uparrow_{t,i}} | \text{character}_{\uparrow_t}(\langle \text{number} \rangle \uparrow_i) | \text{logical}_{\uparrow_{t,i}}$

**@svardef**动作符号是把n, i 和t 填入符号表中。

```
procedure svardef( t, i, n );
    j := tableinsert ( n, t, i );      /*将有关信息填入符号表*/
    if j = 0                         //填表时要检查是否重名
        then errmsg ( duplident , statementno);
    else if j = -1                  //符号表已满
        then errmsg( tblofflow, statementno);
end svardef;
```

```
procedure allocsv( i );
    codeptr := codeptr + i ;      //codeptr 为分配地址指针
end allocsv;
```

**@allocsv** 和 **@svardef** 可以合并

## 对于变长字符串（或其它大小可变的数据实体）

- 动态申请存储空间，将可变长实体存储在堆中
- 通过**指向存放该实体数据区的指针**来引用该实体
- 有时还应得到该实体存储空间的大小信息，一起填入符号表内

## 10.3.3 数组变量声明的处理

### 静态数组

数组的大小在编译时是已知的

建立**数组模板**(又称为**数组信息向量**) 以便以后的程序中引用该数组元素时，可按照该模板提供的信息，计算数组元素(下标变量)的存储地址。

### 动态数组

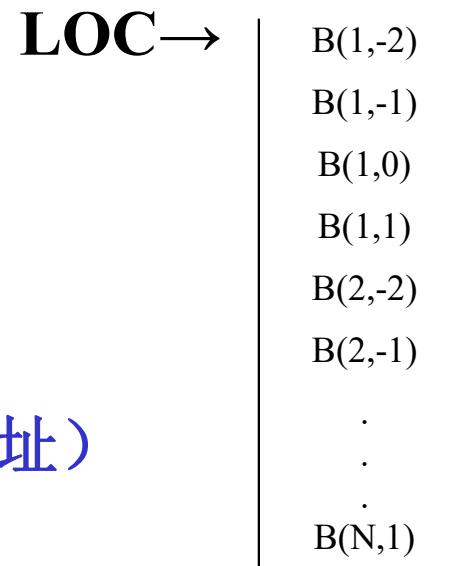
**动态数组：**大小只有在运行时才能最后确定。

在编译时仅为该模板分配一个空间，而模板本身的内容将在运行时才能填入。

大部分程序设计语言，数组元素是按行（优先）存放在存储器中的，如声明数组 **array B( N, -2: 1 ) char ;**

	-2	-1	0	1
1				
2				
3				
⋮				
N				

实际数组B各元素的存储次序为：



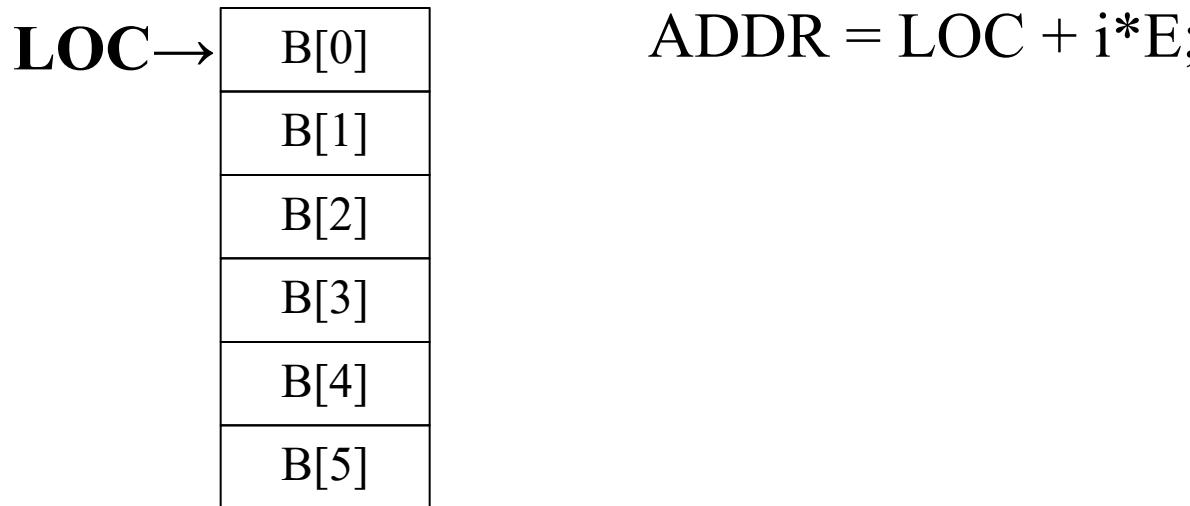
**LOC**是数组首地址  
(该数组第一个元素的地址)

\* **FORTRAN** 例外，  
它按列（优先）存放数组元素

问题：如何通过LOC, 数据B [x1, x2, … , xn] 定位元素的地址？

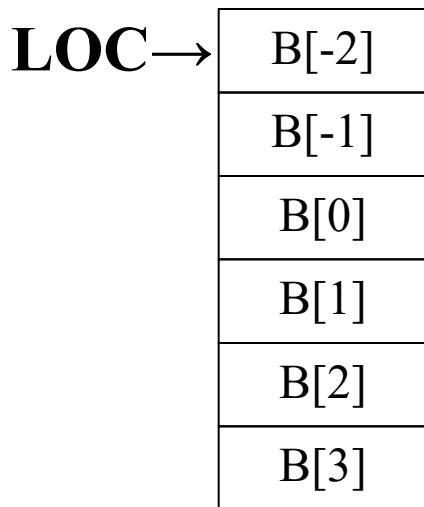
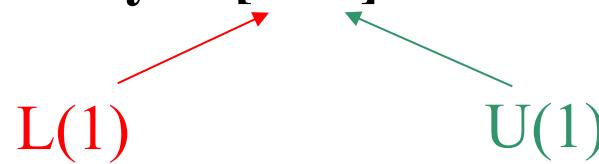
先考虑1维： Array B[5]

如何通过LOC, 定位 B[i]的地址？



问题：如何通过LOC, 数据B [x1, x2, … , xn] 定位元素的地址？

B的定义带有上下界： Array B [-2:3]



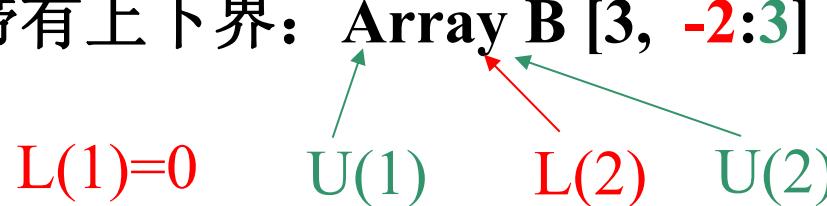
定位 B[i] 的地址？

$$\text{ADDR} = \text{LOC} + (x - L(1)) * E;$$

问题：如何通过LOC, 数据B [x1, x2, … , xn] 定位元素的地址？

二维数组B的定义带有上下界： Array B [3, -2:3]

<b>LOC</b> →	B[0, -2]
	B[0, -1]
	B[0, 0]
	B[0, 1]
	B[0, 2]
	B[0, 3]
	B[1, -2]
	B[1, -1]
	B[1, 0]
	B[1, 1]
	B[1, 2]
	B[1, 3]



定位 B[V(1), V(2)] 的地址？

$$\text{ADDR} = \text{LOC}$$

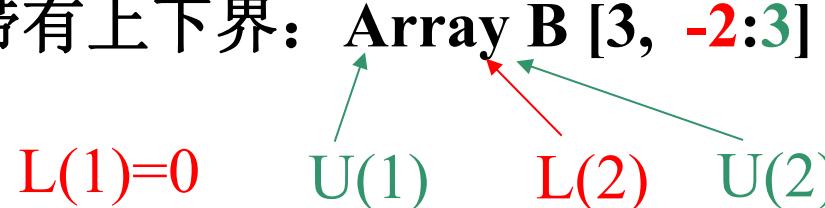
$$+ (V(1) - L(1)) * (U(2)-L(2)+1) * E$$

$$+ (V(2) - L(2)) * E$$

问题：如何通过LOC, 数据B [x1, x2, … , xn] 定位元素的地址？

二维数组B的定义带有上下界： Array B [3, -2:3]

$LOC \rightarrow$	B[0, -2]
	B[0, -1]
	B[0, 0]
	B[0, 1]
	B[0, 2]
	B[0, 3]
	B[1, -2]
	B[1, -1]
	B[1, 0]
	B[1, 1]
	B[1, 2]
	B[1, 3]



定位  $B[V(1), V(2)]$  的地址？

$$ADDR = LOC$$

$$+ (V(1) - L(1)) * P(1) * E$$

$$+ (V(2) - L(2)) * P(2) * E$$

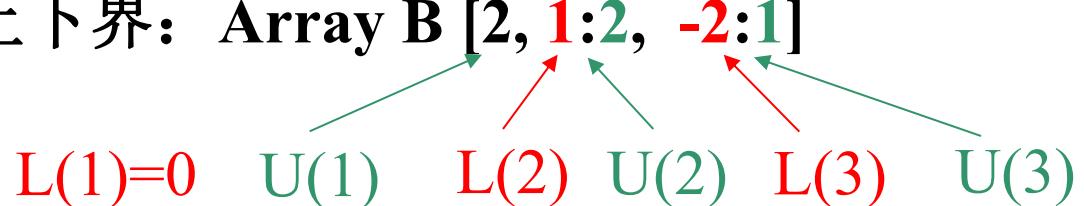
$$P(i) = 1 \quad (\text{if } i=2)$$

$$= U(i) - L(i) + 1 \quad (\text{if } i=1)$$

问题：如何通过LOC, 数据B [x1, x2, … , xn] 定位元素的地址？

三维数组B的定义带有上下界： Array B [2, 1:2, -2:1]

<b>LOC</b> →	B[0,1,-2]
	...
	B[V(1)-1,2,1]
	B[V(1),1,-2]
	...
	B[V(1),V(2)-1,1]
	B[V(1),V(2),-2]
	...
	B[V(1),V(2),V(3)-1]
	B[V(1),V(2),V(3)]



定位 B[V(1), V(2), V(3)] 的地址？

$$\text{ADDR} = \text{LOC}$$

$$\begin{aligned}
 & + (V(1) - L(1)) * (U(2)-L(2)+1) * (U(3)-L(3)+1) * E \\
 & + (V(2) - L(2)) * (U(3)-L(3)+1) * E \\
 & + (V(3) - L(3)) * E
 \end{aligned}$$

$$P(i) = 1 \quad (\text{if } i=n, \text{ 这里 } n=3)$$

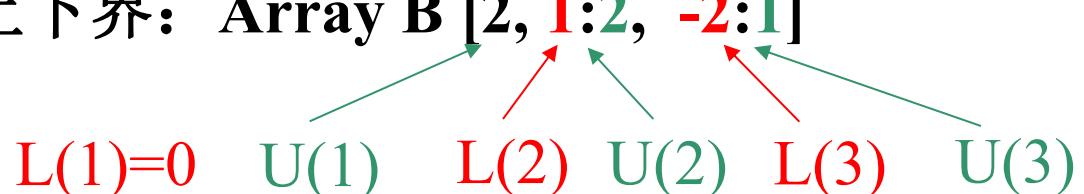
$$= (U(i+1)-L(i+1)+1)*(U(i+2)-L(i+2)+1)*...*(U(n)-L(n)+1),$$

$$= \prod_{j=i+1}^n [U(j)-L(j)+1] \quad (\text{if } 1 \leq i \leq n)$$

问题：如何通过LOC, 数据B [x1, x2, … , xn] 定位元素的地址？

三维数组B的定义带有上下界： Array B [2, 1:2, -2:1]

<b>LOC</b> →	B[0,1,-2]
	...
	B[V(1)-1,2,1]
	B[V(1),1,-2]
	...
	B[V(1),V(2)-1,1]
	B[V(1),V(2),-2]
	...
	B[V(1),V(2),V(3)-1]
	B[V(1),V(2),V(3)]



定位 B[V(1), V(2), V(3)] 的地址？

$$\text{ADDR} = \text{LOC}$$

$$+ (V(1) - L(1)) * P(1) * E$$

$$+ (V(2) - L(2)) * P(2) * E$$

$$+ (V(3) - L(3)) * P(3) * E$$

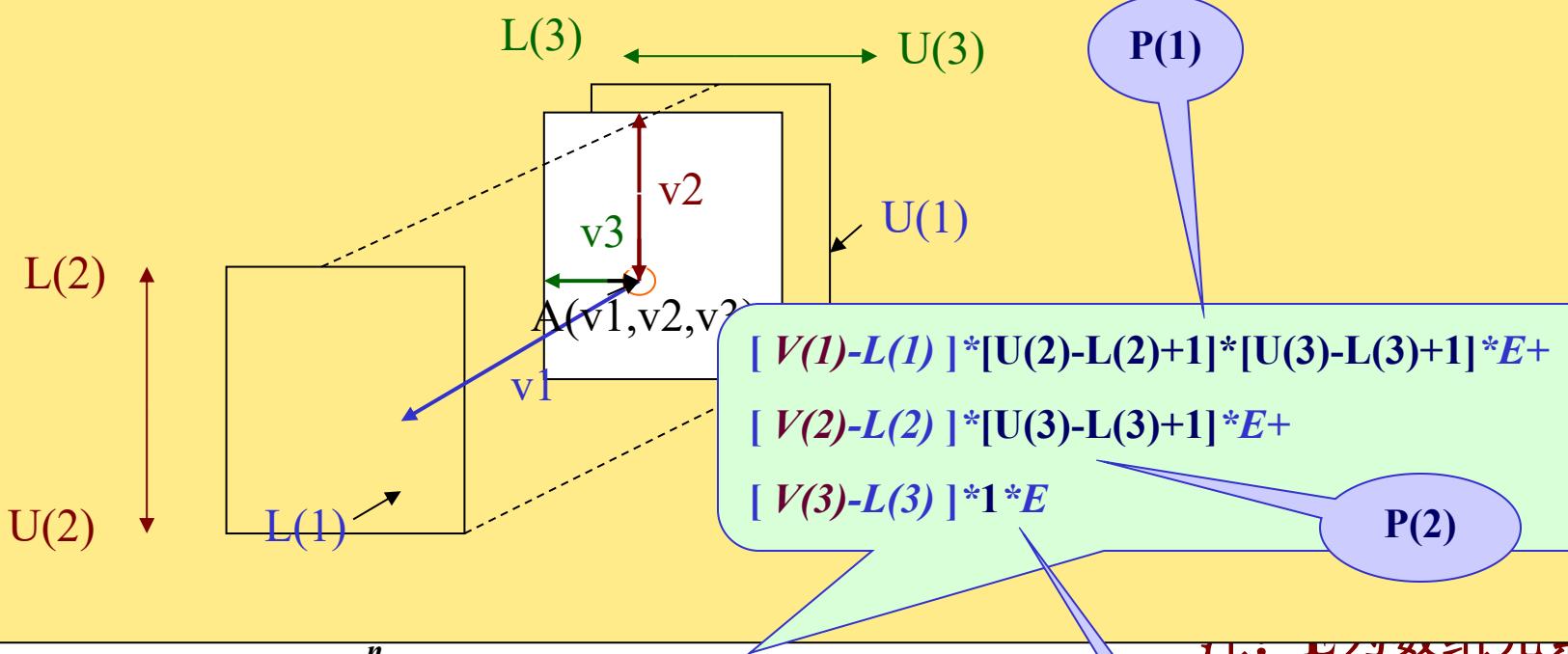
$$\text{ADDR} = \text{LOC} + \sum_{i=1}^n [V(i) - L(i)] * P(i) * E$$

$$P(i) = 1 \quad (\text{if } i=n, \text{ 这里 } n=3)$$

$$= (U(i+1) - L(i+1) + 1) * (U(i+2) - L(i+2) + 1) * \dots * (U(n) - L(n) + 1),$$

$$= \prod_{j=i+1}^n [U(j) - L(j) + 1] \quad (\text{if } 1 \leq i < 1)$$

# Compiler



其中

$$P(i) = \begin{cases} 1 & \text{当 } i = n \text{ 时} \\ \prod_{j=i+1}^n [U(j) - L(j) + 1] & \text{当 } 1 \leq i < n \text{ 时} \end{cases}$$

# 如何加速计算？

$$ADR = LOC + \sum_{i=1}^n [V(i) - L(i)] \times P(i) \times E$$

其中

$$P(i) = \begin{cases} 1 & \text{当 } i = n \text{ 时} \\ \prod_{j=i+1}^n [U(j) - L(j) + 1] & \text{当 } 1 \leq i < n \text{ 时} \end{cases}$$

可变部分

$$ADDR = LOC + \sum_{i=1}^n V(i) \times P(i) \times E$$

不变部分（可提前计算）

$$\sum_{i=1}^n L(i) \times P(i) \times E$$

# 如何加速计算？

$$ADR = LOC + \sum_{i=1}^n [V(i) - L(i)] \times P(i) \times E$$

其中

$$P(i) = \begin{cases} 1 & \text{当 } i = n \text{ 时} \\ \prod_{j=i+1}^n [U(j) - L(j) + 1] & \text{当 } 1 \leq i < n \text{ 时} \end{cases}$$

可变部分

不变部分（可提前计算）

$$ADDR = LOC + \sum_{i=1}^n V(i) \times P(i) \times E + RC$$

$$RC = - \sum_{i=1}^n L(i) \times P(i) \times E$$

可变部分                          不变部分（可提前计算）

$$\text{ADDR} = \text{LOC} + \sum_{i=1}^n V(i) \times P(i) \times E + RC$$

$$RC = -\sum_{i=1}^n L(i) \times P(i) \times E$$

**array B ( N, -2: 1 ) char ;**

以前面所举的二维数组B为例，若N = 3

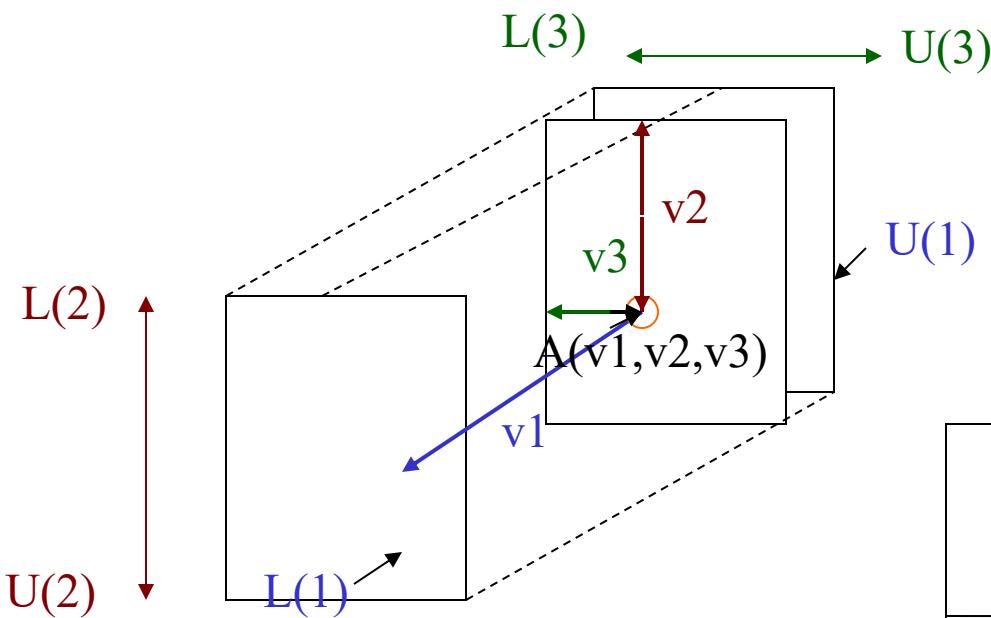
则  $P(1) = [U(2) - L(2) + 1]$                        $P(2) = 1$

$$\begin{aligned} &= 1 - (-2) + 1 \\ &= 4 \end{aligned}$$

$$\begin{aligned} RC &= -\sum_{i=1}^2 L(i)P(i) * E \\ &= -[1 \times 4 + (-2) \times 1] \times E \\ &= -2 \times E \end{aligned}$$

因此，若有数组元素B( 2 , 1 ), 则它的地址为：

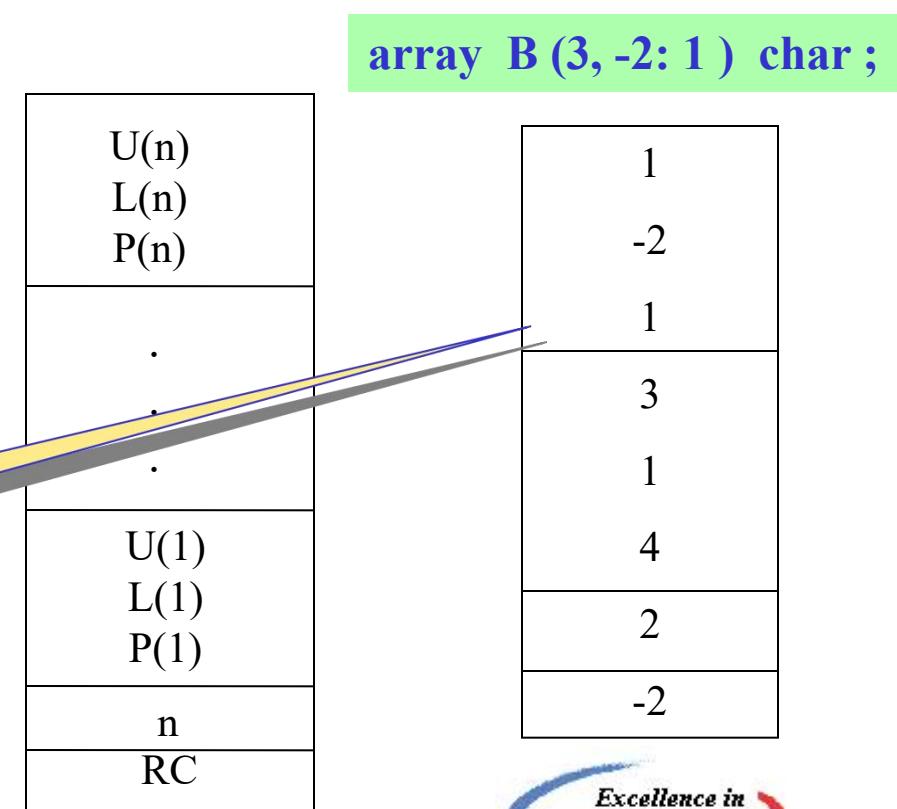
$$\begin{aligned} ADR &= LOC - 2E + \sum_{i=1}^2 V(i) \times p(i) \times E = LOC - 2E + (2 \times 4 + 1 \times 1) \times E \\ &= LOC + 7 \times E \end{aligned}$$



三维数组的例子

数组信息向量表

数组模板的一般形式如下左图所示，而对于数组B的模板如下右图所示：



我们设数组的维数为n，各维的下界和上界为L(i) 和U(i)

我们还假定n维数组元素的下标为V(1), V(2),..., V(n)

则该数组元素的地址计算公式为：

$$ADR = LOC + \sum_{i=1}^n [V(i) - L(i)] \times P(i) \times E$$

注：E为数组元素大小（字节数）

其中

$$P(i) = \begin{cases} 1 & \text{当 } i = n \text{ 时} \\ \prod_{j=i+1}^n [U(j) - L(j) + 1] & \text{当 } 1 \leq i < n \text{ 时} \end{cases}$$

可变部分

$$ADDR = LOC + \sum_{i=1}^n V(i) \times P(i) \times E + RC$$

不变部分（可提前计算）

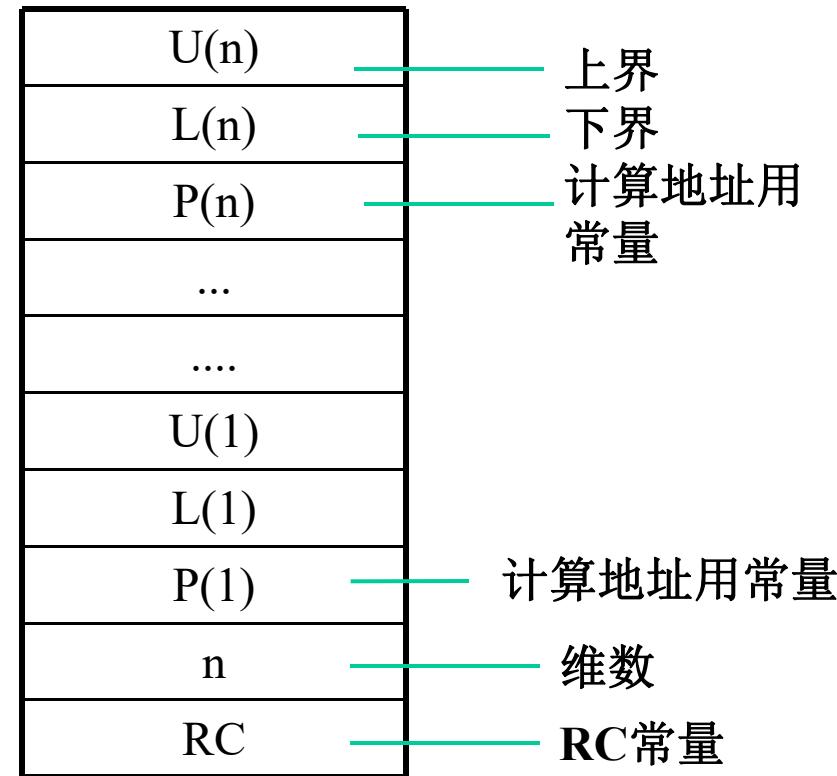
$$RC = -\sum_{i=1}^n L(i) \times P(i) \times E$$

## b) 数组信息向量表（模板）

功能： 1、用于计算下标变量地址  
2、检查下标是否越界

一般形式：

大小：  $3n + 2$



注： 1、数组模板所需的空间大小取决于数组的维数，即 $3n+2$

.:无论是常界或变界数组，在编译时就能确定数组模板的大小

2、常界数组，在编译时就可造信息向量表；而变界数组信息向量表要在目标程序运行时才能造。编译程序要生成相应的指令

array B ( N, -2: 1 ) char ;

以前面所举的二维数组B为例，若N = 3

$$P(2) = 1$$

$$\begin{aligned} P(1) &= [U(2) - L(2) + 1] \\ &= 1 - (-2) + 1 \\ &= 4 \end{aligned}$$

$$\begin{aligned} RC &= - \sum_{i=1}^2 L(i)P(i) \\ &= -[1 \times 4 + (-2) \times 1] \\ &= -2 \end{aligned}$$

数组信息向量表

1
-2
1
3
1
4
2
-2

U(2)--上界

L(2)--下界

P(2)--计算地址常量

U(1)--上界

L(1)--下界

P(1)--计算地址常量

n---维数

RC

## 数组变量声明:

如PL/1声明语句:

**array B (3, -2:3) integer**

声明语句的输入文法为:

```
<array del> → array <entity> (<sublist>)<type>
<sublist> → <subscript> | <subscript>, <sublist>
<subscript> → <integer expr>
           | <integer expr>:<integer expr>
<type> → real | integer | string
```

## 数组声明的ATG文法:

```
<array del> → array ↑k @init↑j <entity>↑n ( <sublist>↑j )
          <type>↑t @symbinsert↓j, n, t,k
<sublist> ↑j → <subscript> @dimen#↑j
          | <subscript>, <sublist>↑j @dimen#↑j
<subscript> → <integer expr>↑u @bannds↓u
          | <integer expr>↑l : @lowerbnd↓l
          <integer expr>↑u @upperbnd↓u, l
```

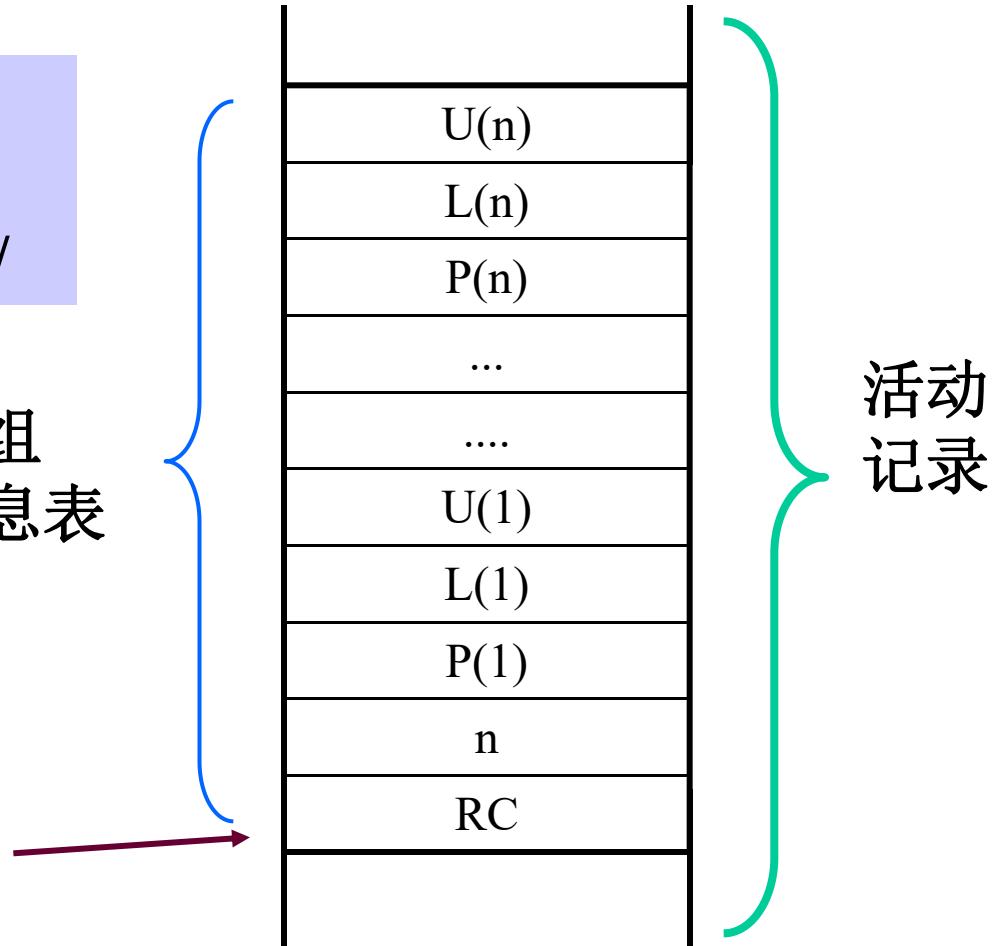
1) 动作程序 **@init** 的功能为在分配给数组模板区中保留两个存储单元，用来放 RC 和 n，并将维数计数器 j 清0。

2) **@dimen# ↑j**:     $j := j + 1$ , 即统计维数

```
1) @init:  
p:= p+2;  
j := 0; /*维数计数器*/
```

数组  
信息表

运行栈指针p



$\langle \text{subscript} \rangle \rightarrow \langle \text{integer expr} \rangle \uparrow u @\text{bannds} \downarrow u$   
 |  $\langle \text{integer expr} \rangle \uparrow l : @\text{lowerbnd} \downarrow l$   
 $\langle \text{integer expr} \rangle \uparrow u @\text{upperbnd} \downarrow u, l$

3) **@bannds** 将省略下界表达式情况的  $u \Rightarrow U(i)$ , 但应把相应的  $L(i)$  置成隐含值 1, 然后计算  $P(i)$

实际  $P(i)$  计算公式可利用  $P(i) = [U(i+1) - L(i+1) + 1] \times \underline{P(i+1)}$

$$P(i) = \begin{cases} 1 & \text{当 } i = n \text{ 时} \\ \prod_{j=i+1}^n [U(j) - L(j) + 1] & \text{当 } 1 \leq i < n \text{ 时} \end{cases}$$

注: 由于  $P(i)$  的计算要依赖于  $P(i+1)$ , 所以实际  $P(i)$  的值是反填的

4) **@lowerbnd** 把  $l \Rightarrow L(i)$   
**@upperbnd** 把  $u \Rightarrow U(i)$ , 并计算  $P(i)$

5) 最后的动作程序 **@symbinsert** 是把数组名  $n$ , 数组维数  $j$  和数组元素类型  $t$  及数组标志  $k$  填入符号表中; 为数组分配存储空间

对于变界数组：

4) **@lowerbnd<sub>↓l</sub>**

生成将  $l \Rightarrow L(i)$  的代码

**@upperbnd<sub>↓u</sub>**

生成 把  $u \Rightarrow U(i)$  的代码,

生成计算  $P(i)$  的代码;

生成将  $P(i)$  的值送模板区的代码;

5) **@symbinsert<sub>↓j, n, t,k</sub>**

a) 把  $n, j, t, k$  填入符号表中

b) 生成调用运行子程序代码（计算  $RC$ , 并将计算结果和数组名一起存入模板区；计算数组所需数据区大小，为数组分配存储空间，并将头地址填入符号表。）

- 记录、过程的声明----自学
- 作业：试设计Pascal记录变量（无变体）的属性翻译文法，并构造相应的语义动作程序。

## 10.4 表达式的处理

分析表达式的主要目的是生成计算该表达式值的代码。通常的做法是把表达式中的操作数装载（LOAD）到操作数栈（或运行栈）栈顶单元或某个寄存器中，然后执行表达式所指定的操作，而操作的结果保留在栈顶或寄存器中。

注：操作数栈即操作栈，它可以和前述的运行栈（动态存储分配）合而为一，也可单独设栈。

本章中所指的操作数栈实际应与动态运行（存储分配）栈分开。

请看下面的整型表达式ATG文法：

# *Compiler*

1.  $\langle \text{expression} \rangle \rightarrow \langle \text{expr} \rangle$
  2.  $\langle \text{expr} \rangle \rightarrow \langle \text{term} \rangle \langle \text{terms} \rangle$
  3.  $\langle \text{terms} \rangle \rightarrow \epsilon$
  4.  $\quad | + \langle \text{term} \rangle @\text{add} \langle \text{terms} \rangle$
  5.  $\quad | - \langle \text{term} \rangle @\text{sub} \langle \text{terms} \rangle$
  6.  $\langle \text{term} \rangle \rightarrow \langle \text{factor} \rangle \langle \text{factors} \rangle$
  7.  $\langle \text{factors} \rangle \rightarrow \epsilon$
  8.  $\quad | * \langle \text{factor} \rangle @\text{mul} \langle \text{factors} \rangle$
  9.  $\quad | / \langle \text{factor} \rangle @\text{div} \langle \text{factors} \rangle$
  10.  $\langle \text{factor} \rangle \rightarrow \langle \text{variable} \rangle \uparrow_n @\text{lookup} \downarrow_n \uparrow_j$   
 $\qquad \qquad \qquad @\text{push} \downarrow_j$
  11.  $\quad | \langle \text{integer} \rangle \uparrow_i @\text{pushi} \downarrow_i$
  12.  $\quad | (\langle \text{expr} \rangle)$

有关的语义动作为：

```
procedure add;  
    emit('ADD');  
end;
```

```
procedure mul;  
    emit('MUL');  
end;
```

```
procedure lookup(n);
    string n; integer j;
    j:= symblookup( n);
    /*名字n表项在符号表中的位置*/
    if j < 1
        then /*error*/
        else return (j);
end;
```

```
procedure push(j);
    integer j;
    emit('LOD', symbtbl (j).objaddr);
end;
```

```
procedure pushi(i); /*压入整数*/
    integer i;
    emitl('LDC', i) ;
end;
```

# Compiler

对于输入表达式  $x + y * 3$ :

```
<expression>
=> <expr>
=> <term><terms>
=> <factor><factors><terms>
=> <variable>↑n@lp↓n↑j@ph↓j<factors><terms>
=> <variable>↑n@lp↓n↑j@ph↓j<terms>
=> <variable>↑n@lp↓n↑j@ph↓j+<term>@add<terms>
=> <variable>↑n@lp↓n↑j@ph↓j+<factor><factors>@add<terms>
=> <variable>↑n@lp↓n↑j@ph↓j+<variable>↑n@lp↓n↑j@ph↓j<factors>@add<terms>
=> <variable>↑n@lp↓n↑j@ph↓j+<variable>↑n@lp↓n↑j@ph↓j*<factor>@mul<factors>@add<terms>
=> <variable>↑n@lp↓n↑j@ph↓j+<variable>↑n@lp↓n↑j@ph↓j*<integer>↑i@phi↓i@mul<factors>@add<terms>
=> <variable>↑n@lp↓n↑j@ph↓j+<variable>↑n@lp↓n↑j@ph↓j*<integer>↑i@phi↓i@mul@add<terms>
=> <variable>↑n@lp↓n↑j@ph↓j+<variable>↑n@lp↓n↑j@ph↓j*<integer>↑i@phi↓i@mul@add
```

- 1.<expression>→<expr>
- 2.<expr>→<term><terms>
- 3.<terms>→ε
4. | +<term>@add<terms>
5. | - <term>@sub<terms>
- 6.<term>→<factor><factors>
- 7.<factors>→ ε
8. | \*<factor>@mul<factors>
9. | /<factor>@div<factors>
- 10.<factor>→<variable><sub>↑n</sub>@lookup<sub>↓n↑j</sub> @push<sub>↓j</sub>
11. | <integer><sub>↑i</sub>@push<sub>↓i</sub>
12. | (<expr>)

LOD, < ll, on><sub>x</sub>

LOD, < ll, on><sub>y</sub>

LDC, 3

MUL

ADD

上面所述的表达式处理实际上忽略了出现在表达式中各操作数类型的不同，且变量也仅限于简单变量。

下面假定表达式中允许整型和实型混合运算，并允许在表达式中出现下标变量（数组元素）。因此应该增加有关类型一致性检查和类型转换的语义动作，也要相应产生计算下标变量地址和取下标变量值的有关指令。

## 表达式：类型检查

### 静态类型检查（static type checking）

在编译时完成类型检查，避免运行时的运行错误

### 动态类型检查（dynamic type checking）

在运行时完成类型检查

占用运行时间，影响效率（less efficient）

### 不做任何类型检查

主要靠对程序员的信任

## 类型转换

### ◆ C语言数据类型转换：

- 自动类型转换：指编译器进行的数据类型转换，不需要程序员干预。
- 强制类型转换：程序员自己在代码中进行的类型转换  
如：  
`(float) a; //将变量 a 转换为 float 类型`  
`(int)(x+y); //把表达式 x+y 的结果转换为 int 整型`  
`(float) 100; //将数值 100（默认为int类型）转换为 float 类型`

# 自动类型转换

C99标准中的转换规则：

转换前类型		转换后类型		结果类型
运算分量1	运算分量2	运算分量1	运算分量2	
long double	任一类型	long double		long double
double	任一类型	double		double
float	任一类型	float		float
unsigned long int	任一类型	unsigned long int		unsigned long int
long int	unsigned int	long int		long int
		unsigned long int		unsigned long int
long int	任一类型	long int		long int
unsigned int	任一类型	unsigned int		unsigned int
任一类型	任一类型	int		int

- 注：1. 运算分量1与运算分量2并不一定分别对应于左运算分量与右运算分量，也可相反。  
 2. 如果有上下两个规则相冲突，那么适用于位于上面的规则。  
 3. 由表可以看出，在普通算术运算中，两运算分量类型必须一致。

一个例子：

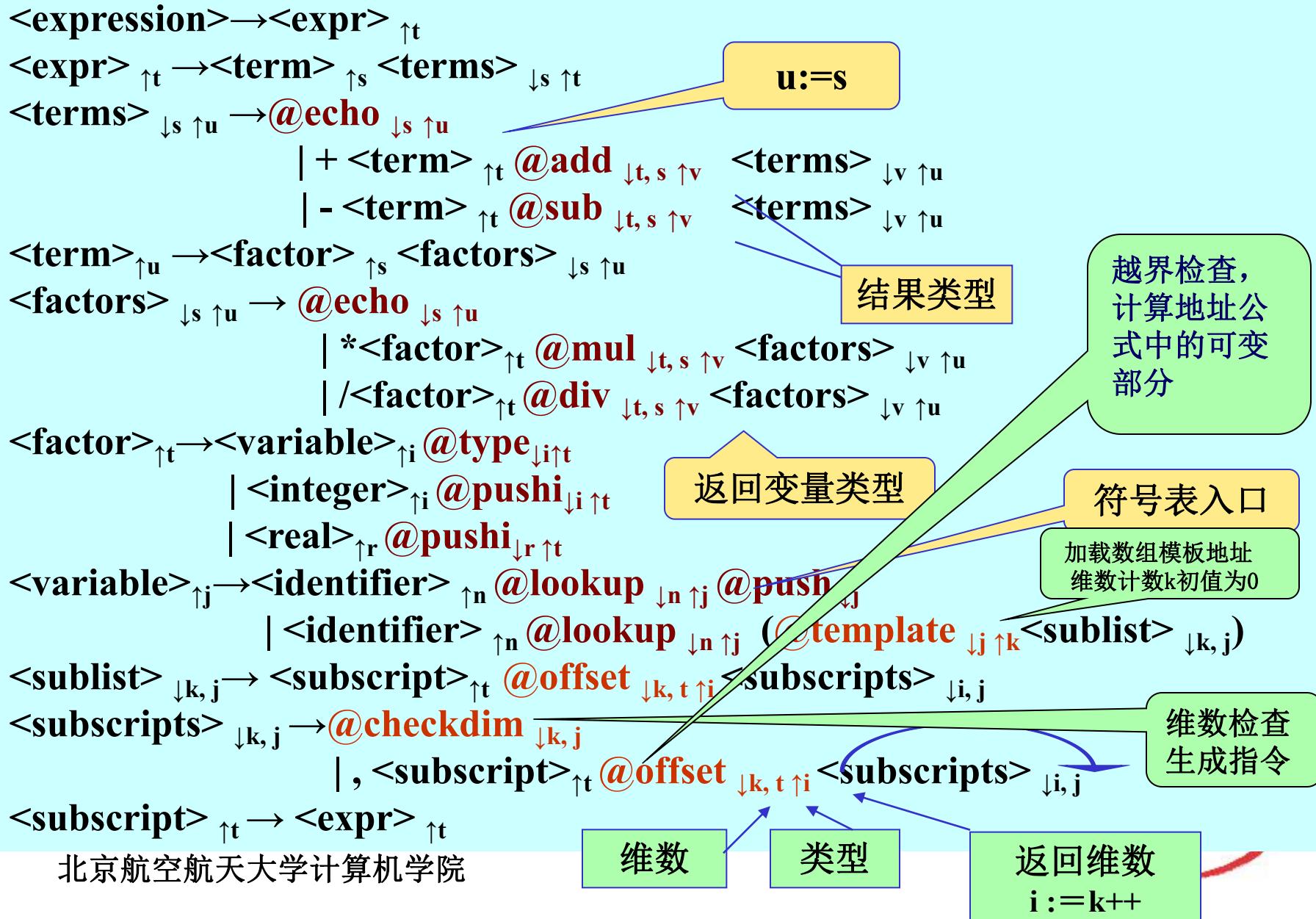
```
#include <stdio.h>
int main(){
    int sum = 103; //总数
    int count = 7; //数目
    double average; //平均数
    average = (double) sum / count;
    printf("Average is %lf!\n", average);
    return 0;
}
```

运行结果：

Average is 14.714286!

去掉强制类型转换，运行结果：  
Average is 14!

# Compiler



# Compiler

语义动作add等应作相应改变：

```
procedure add( t, s);
    string t, s;
    if t = 'real' and s = 'integer'
    then begin
        emit( 'CVN'); /*次栈顶转为实数*/
        emit( 'ADD');
        return ( 'real');
    end;
    if t = 'integer' and s = 'real'
    then begin
        emit( 'CNV'); /*栈顶转为实数*/
        emit( 'ADD');
        return ( 'real');
    end;
    emit( 'ADD');
    return ( t);
end;
```

次栈顶

栈顶

越界检查，计算地址  
公式中的可变部分

```
procedure offset( k, t );
    integer k; string t;
    k := k+1;
    if t ≠ 'integer'
    then errmsy('数组下标应为整
                 型表达式', statno);
    else emitl( 'OFS', k );
    return (k);
```

and;

```
procedure checkdim( k, j);
    integer k, j;
```

```
if k ≠ symbtbl (j).dim
then errmsy('数组维数与
             声明不匹配', statno);
else begin
    emit( 'ARR');
    emit( 'DER');
end;
```

end;

加载数组  
元素内容

```
procedure template(j);
    integer j;
```

```
emitl( 'TMP', symbtbl (j). objaddr);
k:= 0; /*维数计数器初始化*/
return(k);
end;
```

模板入口地址

★ 过程template发送一条目标机指令 ‘TMP’，该指令把数组的模板地址加载到操作数栈顶，并将下标（维数）计数器k清0。

★ offset过程要确保每一个下标都是整型，而且发送一条 ‘OFS’ 指令，该指令在运行时要完成以下功能：

1. 检查第k个下标值是否在栈顶并是否在上下界范围内

2. 使用下列递归函数，计算地址计算公式中可变部分：

$$VP(0) = 0;$$

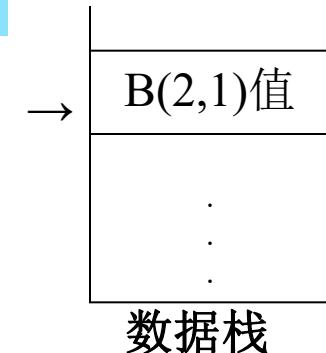
$$VP(k) = VP(k-1) + V(k) * P(k) \quad 1 \leq k \leq n$$

该VP函数是由计算公式  $\sum_{k=1}^n V(k) \times P(k)$  导出的

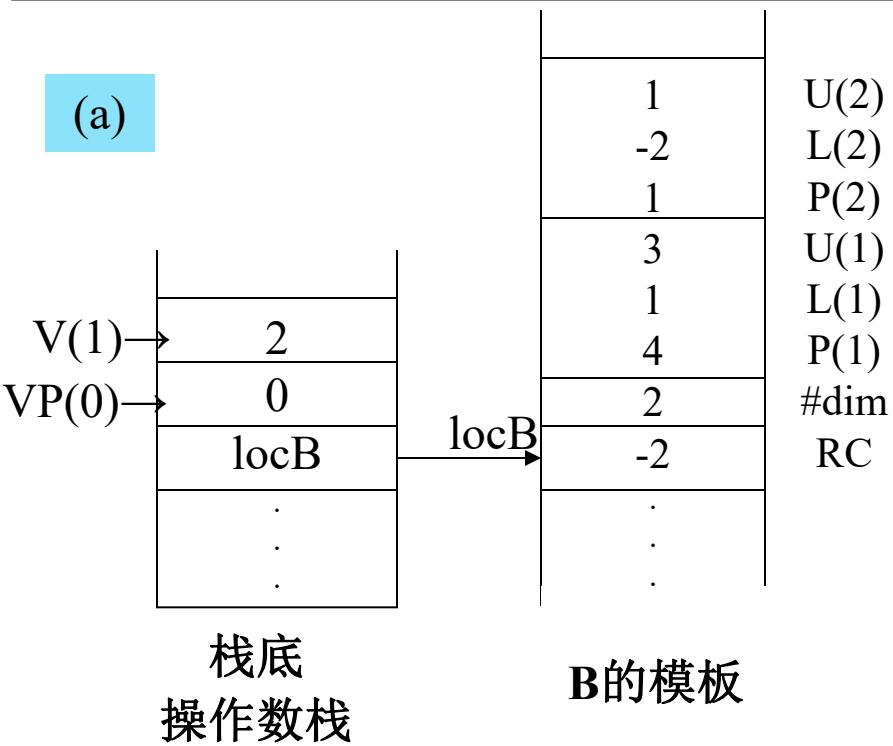
下面以数组元素B(2,1)为例，说明

- (a) 执行TMP指令并形成第一个下标值的情况
- (b) 执行第一个OFS指令并形成第二个下标值的情况
- (c) 执行第二个OFS指令及ARR指令后的情况
- (d) 执行DER指令，最后在栈顶形成下标变量B(2,1)的值

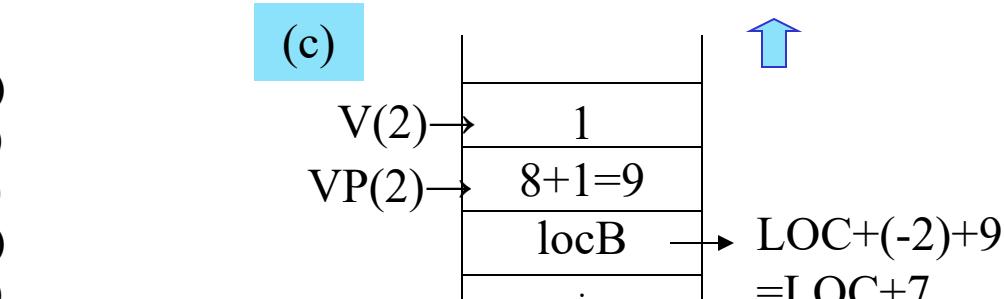
(d)



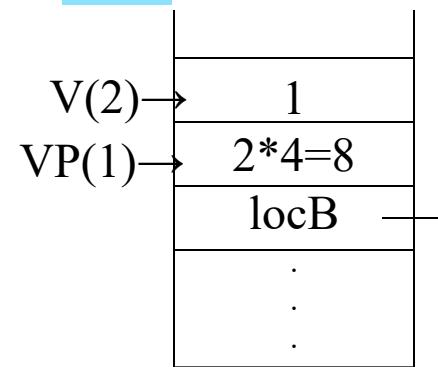
(a)



(c)



(b)



处理逻辑表达式(关系表达式) 的方法与处理算术表达式的方式基本相同。下面是逻辑表达式 $\sim(x = y \& y \neq z \mid z < x)$ 生成的指令序列：

```
LOD, (ll, on )x
LOD, (ll, on )y
EQL
LOD, (ll, on )y
LOD, (ll, on )z
NEQ
AND
LOD, (ll, on )z
LOD, (ll, on )x
LES
ORL
NOT
```

## 10.5 赋值语句的处理

X:= Y+X;

LDA ( ll, on )x  
LOD ( ll, on )y  
LOD ( ll, on )x  
ADD  
STN

$\langle \text{assignstat} \rangle \rightarrow @\text{setL}_{\uparrow L} \langle \text{variable} \rangle_{\downarrow L \uparrow t} :=$   
 $\quad @\text{resetL}_{\uparrow L} \langle \text{expr} \rangle_{\uparrow s} @\text{storin}_{\downarrow t, s};$

置“左值”特征L为真

被赋变量类型

表达式类型

类型转换，生成  
STN指令

**@setL**是设置变量为“左值”（被赋变量），即将属性L置true

**@resetL**是设置变量为非被赋变量，即把属性L置成false

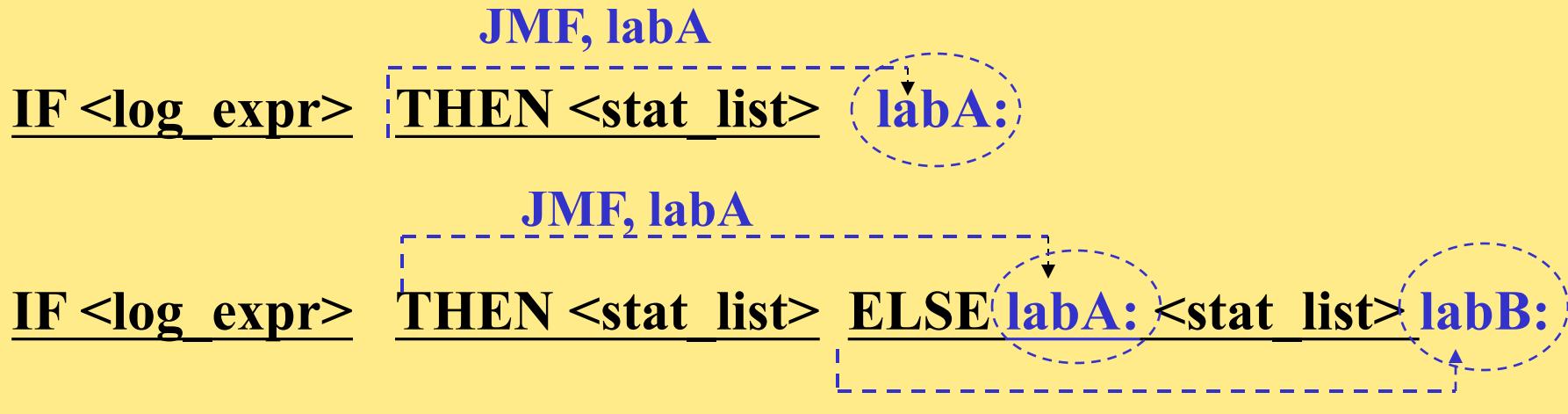
```
procedure setL;
  return (true );
end;
指示取变量地址
```

```
procedure resetL;
  return (false );
end;
指示取变量之值
```

```
procedure storin(t,s);
  string t, s;
  if t ≠ s
    then /*生成进行类型转换的指令*/
      emit('STN');
  end;
```

## 10.6 控制语句的处理

### 10.6.1 if语句



其属性翻译文法及相应的语义动作程序：

1.  $\langle \text{if\_stat} \rangle \rightarrow \langle \text{if\_head} \rangle_y \langle \text{if\_tail} \rangle_{\downarrow y}$
2.  $\langle \text{if\_head} \rangle_y \xrightarrow{\text{IF } <\text{log\_expr}> @\text{brf}_{\uparrow y}} \text{THEN } <\text{stat list}>$
3.  $\langle \text{if\_tail} \rangle_{\downarrow y} \xrightarrow{\text{@labprod}_{\downarrow y}} |\text{ELSE } @\text{br}_{\uparrow z} @\text{labprod}_{\downarrow y} <\text{stat list}> @\text{labprod}_{\downarrow z}|$

动作程序@**brf**的功能是生成JMF指令，并将转移标号返回给属性y

```
procedure brf;
  string labx;
  labx := genlab;
    /*产生一标号赋给labx*/
  emitl('JMF', labx);
  return (labx);
end;
```

动作程序@**labprod**是把从继承属性y得到的标号设置到目标程序中

```
procedure labprod(y);
  string y;
  setlab(y);
  /*在目标程序当前位置设标号*/
end;
```

动作程序@**br**是生成JMP指令，并将转移标号返回给属性z

```
procedure br;
  string labz;
  labz := genlab;
  emitl( 'JMP', labz);
  return( labz);
end;
```

1. <if\_stat> → <if\_head><sub>↑y</sub> <if\_tail><sub>↓y</sub>
2. <if\_head><sub>↑y</sub> → IF <log\_expr> @**brf**<sub>↑y</sub> THEN <stat list>
3. <if\_tail><sub>↓y</sub> → @**labprod**<sub>↓y</sub> | ELSE @**br**<sub>↑z</sub> @**labprod**<sub>↓y</sub> <stat\_list> @**labprod**<sub>↓z</sub>

## 10.6.4 for 循环语句

for 语句例子：

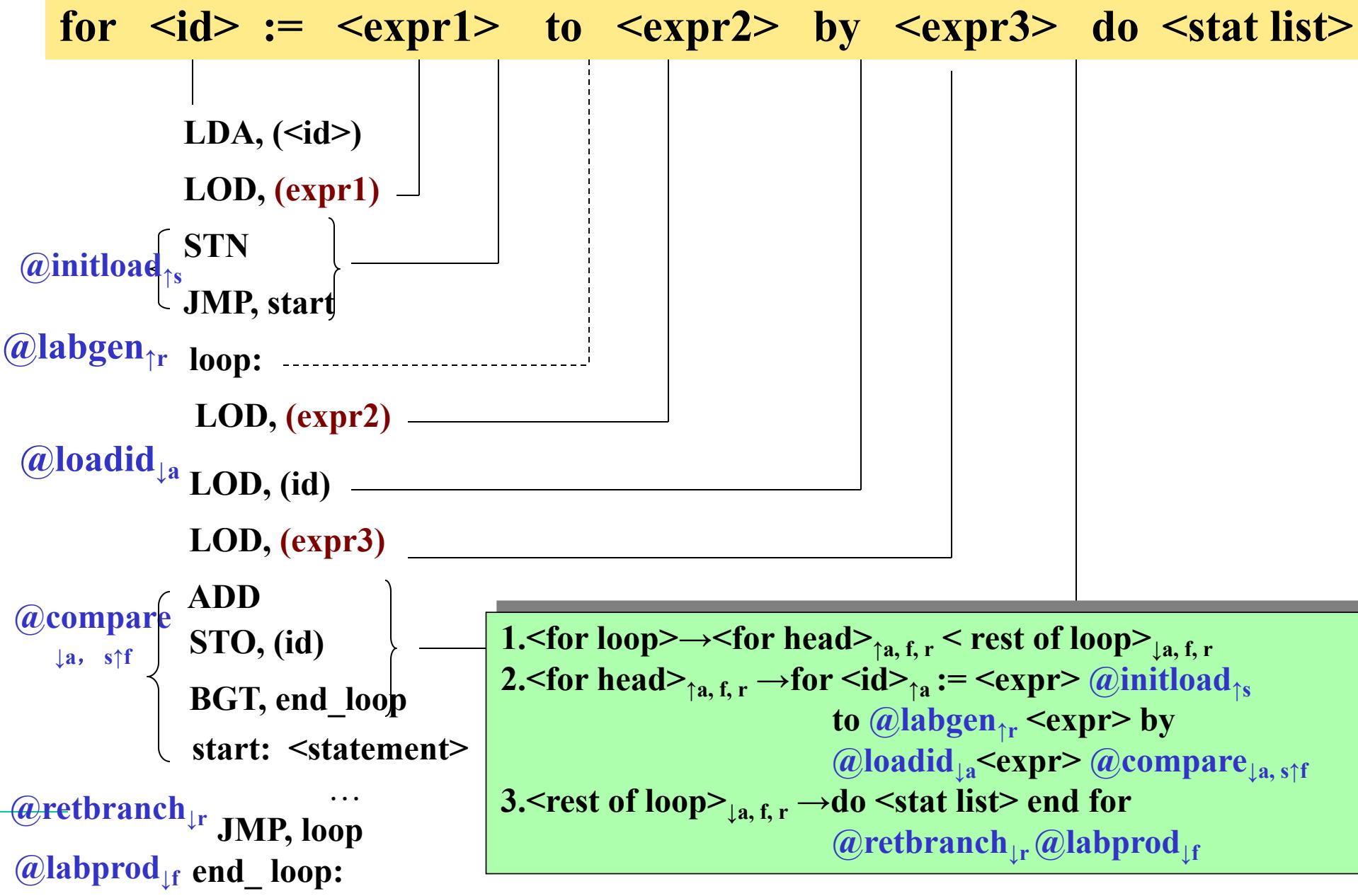
```
for i:= 1 to n by z do  
  <statement>  
  ...  
end for;
```

ATG文法

1.  $\langle \text{for loop} \rangle \rightarrow \langle \text{for head} \rangle_{\uparrow a, f, r} \langle \text{rest of loop} \rangle_{\downarrow a, f, r}$
2.  $\langle \text{for head} \rangle_{\uparrow a, f, r} \rightarrow \text{for } \langle \text{id} \rangle_{\uparrow a} := \langle \text{expr} \rangle @\text{initload}_{\uparrow s}$   
 $\quad \text{to } @\text{labgen}_{\uparrow r} \langle \text{expr} \rangle \text{ by }$   
 $\quad @\text{loadid}_{\downarrow a} \langle \text{expr} \rangle @\text{compare}_{\downarrow a, s \uparrow f}$
3.  $\langle \text{rest of loop} \rangle_{\downarrow a, f, r} \rightarrow \text{do } \langle \text{stat list} \rangle \text{ end for }$   
 $\quad @\text{retbranch}_{\downarrow r} @\text{labemit}_{\downarrow f}$

**@initload** 只生成给循环变量赋初值的指令。

# Compiler



```
procedure labgen
    string r;
    r := genlab;
    setlab(r);
    return ( r );
end;
```

```
procedure loadid( a )
    address a;
    emitl( 'LOD', a );
end;
```

```
procedure compare( a, s);
    address a;  string f, s;
    emit( 'ADD');
    emitl( 'STO', a );
    f := genlab;
    emitl( 'BGT', f );
    setlab( s );
    return( f );
end;
```

```
procedure labprod( f ) // 即labemit
    string f;
    setlab( f );
end;
```

## 10.7 过程调用和返回

### 10.7.1 参数传递的基本形式

#### 1. 传值（call by value） — 值调用

实现：

调用段（过程语句的目标程序段）：

计算实参值  $\Rightarrow$  操作数栈栈顶

被调用段（过程说明的目标程序段）：

从栈顶取得值  $\Rightarrow$  形参单元

过程体中对形参的处理：

对形参的访问等于对相应实参的访问

特点：

数据传递是单向的

如C语言，  
Ada语言的in参数，  
Pascal 的值参数。

## 2. 传地址 (call by reference) — 引用调用

实现:

调用段:

计算实参地址 => 操作数栈栈顶

被调用段:

从栈顶取得地址 => 形参单元

如: FORTRAN,  
Pascal 的变量形参。

过程体中对形参的处理:

通过对形参的间接访问来访问相应的实参

特点:

结果随时送回调用段

## 3. 传名 (call by name)

又称“名字调用”。即把实参名字传给形参。这样在过程体中引用形参时，都相当于对当时实参变量的引用。

当实参变量为下标变量时，传名和传地址调用的效果可能会完全不同。

传名参数传递方式，实现比较复杂，其目标程序运行效率较低，现已很少采用。

```
begin
    integer I;
    array A[1:10] integer;
    procedure P(x);
        integer x;
        begin
            ...
            I := I + 1;
            x := x+ 5;
            ...
        end;
    begin
        ...
        I := 1;
        P( A[I] );
        ...
    end;
end;
```

假定:  $A[1] = 1 \ A[2] = 2$

传地址:

传名:

I : 2  
A[1]: 6

I : 2  
A[I]:= A[I]+5

A[1] = 6 A[2] = 2

A[1] = 1 A[2] = 7

## 10.7.2 过程调用处理

与调用有关的动作如下：

1. 检查该过程名是否已定义（过程名和函数名不能用错） 实参和形参在类型、顺序、个数上是否一致。  
(查符号表)

2. 加载实参（值或地址）

3. 加载返回地址

4. 转入过程体入口地址

例：有过程调用：

`process_symb(symb, cursor, replacestr);`

调用该过程生成的目标代码为：

`LOD, (addr of symb )`

`LOD, (addr of cursor )`

`LOD, (addr of replacestr)`

`JSR, ( addr of process_symb)`

`<retaddr>:....`

传值调用

若实参并非上例中所示变量，而是表达式，则应生成相应计算实参表达式值的指令序列。

`JSR`指令先把返回地址压入操作数栈，然后转到被调过程入口地址。

# Compiler

设过程说明的首部有如下形式：

procedure process\_symb(string:

则过程体目标代码的  
开始处应生成以下指令，  
以存储返回地址和形参  
的值。



ALC, 4 + x /\* x为定长项空间 \*/  
STO, <actrec loc1> /\* 保存返回地址 \*/  
STO, <actrec loc4> /\* 保存replacestr \*/  
STO, <actrec loc3> /\* 保存cursor \*/  
STO, <actrec loc2> /\* 保存symb \*/

过程调用时，实参加加载指令是把实参变量内容（或地址）  
送入操作数栈顶，过程声明处理时，应先生成把操作数栈顶的实参  
送运行栈AR中形参单元的指令。

将操作数栈顶单元内容存入运行栈（动态存储分配的数据区）  
当前活动记录的形式参数单元。

可认为此时运行栈和操作数栈不是一个栈（分两个栈处理）

# Compiler

实参数计数, m 初值为0

过程名符号表入口位置

## 过程调用的ATG文法:

```
<proc call> → <call head>↑i, z @initm↑m <args>↓i, z @genjsr↓i
<call head>↑i, z → <id>↑n @lookupproc↓n↑i, z
<args>↓i, z → @chklength↓i, z | (<arg list>↓i, z)
<arg list>↓i, z → <expr>↑t @chktype↓t, i, m, z↑z <exprs>↓i, z
<exprs>↓i, z → @chklength↓i, z
          | , <expr>↑t @chktype↓t, i, m, z↑z <exprs>↓i, z
```

```
procedure lookupproc(n);
  string n; integer i, z;
  i := lookup(n); /*查符号表*/
  if i < 1
    then begin
      error(‘过程’ , n , ‘未定义’ , statno);
      errorrecovery( panic ); /*应急处理过程 */
      return ( i := 0, z:= 0);
    end
    else return( i , z:= symtbl [i].dim); /* z为形参数目*/
end;
```

# Compiler

```
procedure chktype(t, i, m, z);
  string t; integer m, i, z;
  if z < 1
    then begin
      error(‘实参数大于形参数’ , symtbl [i].name, statno);
      return ( z);
    end
    m := m+1;      /* 实参计数 */
    if t ≠ symtbl [i+m].type
      then error(‘实参和形参类型不匹配’ , symtbl [i+m].name, statno);
    z := z-1;      /* 减去已匹配的形参数 */
    return (z);    /* 剩下待匹配的形参数 */
  end;
```

```
<proc call> → <call head>↑i, z @initm↑m <args>↓i, z @genjsr↓i  
<call head>↑i, z → <id>↑n @lookupproc↓n↑i, z  
<args>↓i, z → @chklength↓i, z | (<arg list>↓i, z)  
<arg list>↓i, z → <expr>↑t @chktype↓t, i, m, z↑z <exprs>↓i, z  
<exprs>↓i, z → @chklength↓i, z  
          | , <expr>↑t @chktype↓t, i, m, z↑z <exprs>↓i, z
```

m ++, z --

LOD, (addr of symb )  
LOD, (addr of cursor )  
LOD, (addr of replacestr)  
JSR, ( addr of process\_symb)  
<retaddr>:....

**@chklenh** 应检验z最后值为0。否则表示实参数目小于形参数目。  
**@genjsr** 生成JSR指令。该指令转移地址为 symtbl [i] .addr

过程说明（定义）的ATG文法如下：



@tblinsert 是把过程名和它的形参名填入符号表中：

```

procedure tblinsert( t, n );
  string t, n; integer hloc;
  if lookup( n ) > 0
    then error( ‘名字定义重复’ , statno);
  else begin
    hloc := hashfctn(n); /*求散列函数值*/
  end;

```

```

hashtbl[hloc] := s; /*s为符号表指针
                      (下标), 为全局量*/
symbtbl [s].name:= n;
symbtbl [s].type:= t;
s := s+1;
end;

```

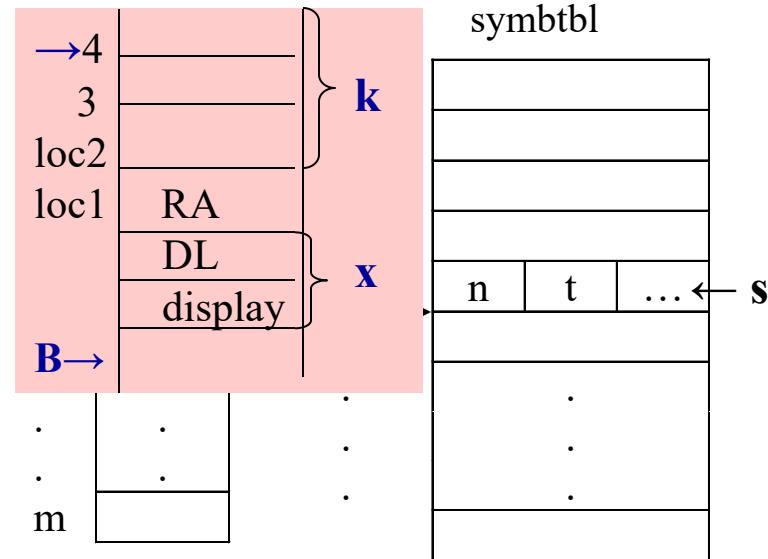


```

procedure emitstores(k);
integer k;
emitl( 'ALC', k + x +... );
emitl( 'STO', < ll, x+1 >);
    /*保存返回地址*/
for i := k + x+1 down to x+2
    /*保存参数值*/
    emitl( 'STO', < ll , i > )
end;
end;

```

**注：**实际ALC指令所分配的空间应在所有局部变量定义处理完以后，并考虑固定空间（前述‘x’）大小，反填回去。



ALC, 4 + x /\* x为定长项空间 \*/  
 STO, <actrec loc1> /\* 保存返回地址 \*/  
 STO, <actrec loc4> /\* 保存replacestr \*/  
 STO, <actrec loc3> /\* 保存cursor \*/  
 STO, <actrec loc2> /\* 保存symb \*/

## 10.7.3 返回语句和过程体结束的处理

其语义动作有：

1)若为函数过程，应将操作数栈（或运行栈）顶的函数结果值送入（存回）函数值结果单元

2)生成无条件转移返回地址的指令（JMP RA）

3)产生删除运行栈中被调用过程活动记录的指令  
(只要根据DL—活动链，把abp退回去即可)

作业：写出for语句在执行循环体之前先做循环条件测试的属性翻译文法及其处理动作程序。