第十章　语义分析和代码生成

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

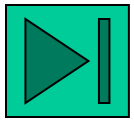
3、语义处理：

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。

语义分析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中间代码或目标代码。



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！此时最好采用上下文相关文法进行描述。

例如，对于有嵌套子程序结构的程序段：

BEGIN

若存在文法规则： **VAR ::= I** 则

BEGIN … BEGIN α INT I β VAR END … *I* … END

此时内层子程序可归约为<BLOCK>

BEGIN … I^{β} <BLOCK> … *I* … END

END

…

BEGIN … δ *VAR* … END

第一次I的归约正确
第二次I的归约错误

δ∈V*且不包含变量 *I* 的声明

∴文法规则应改为： **INT I β VAR ::= INT I β I**

上下文相关文法

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，显然不实用。

处理这种上下文有关问题的一般方法，使用专门的语义动作来补充上下文无关分析器的动作，即借助语义分析来解决。

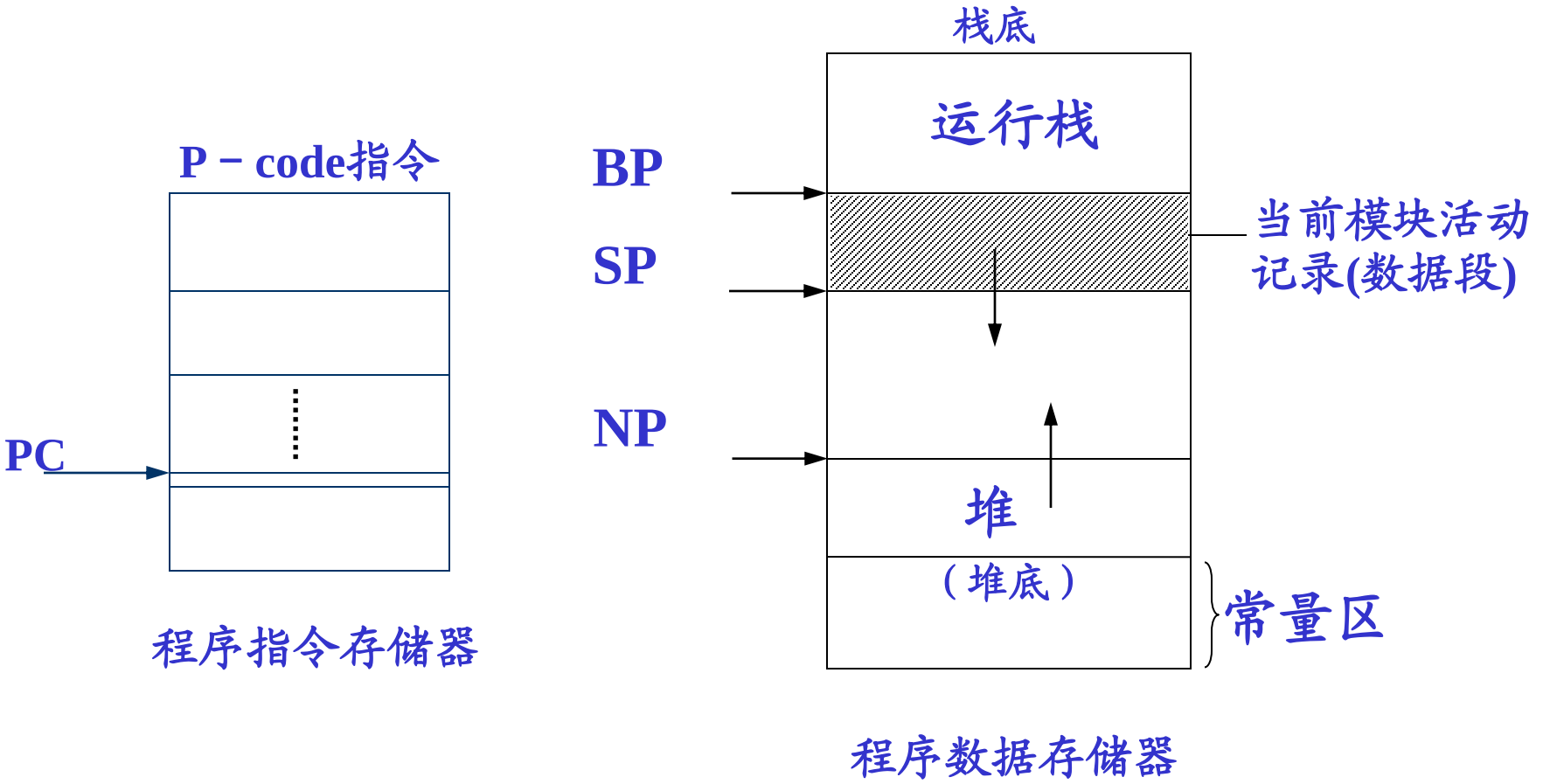
通常我们把与语义相关的上下文有关信息填入符号表中，并通过查符号表中的这些信息来分析程序的语义是否正确。

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	LOD	D	将D的内容→栈顶
立即加载	LDC	常量	常量→栈顶
地址加载	LDA	(D)	变量D的地址→栈顶
存储	STO	D	栈顶内容存入→变量D
间接存	ST	@D	将栈顶内容→D所指单元
间接存	STN		将栈顶内容→次栈顶所指单元
加	ADD		栈顶和次栈顶内容相加，结果留栈顶
减	SUB		次栈顶内容减栈顶内容
乘	MUL		
……			

指令名称	操作码	地址	指令意义
等于比较	EQL		次栈顶内容与栈顶内容比较，结果（1或0）留栈顶
不等比较	NEQ		
大于比较	GRT		
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

语义的表示:

给出语言结构的属性翻译文法来说明其语义及语义动作，并把这些语义动作插入属性翻译文法产生式中的适当位置。

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

10.3 声明的处理

语义的表示:

给出语言结构的属性翻译文法来说明其语义及语义动作，并把这些语义动作插入属性翻译文法产生式中的适当位置。

编译程序的任务:

也就是说，处理声明语句主要是做填表工作（填表前先得查表，检查是否重名）。

处理对已声明的实体的引用时主要是做查表工作。

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) 类型说明放符在变量的后面。如：Pascal，PL/1，Ada等，需要返填。

如PL/1声明语句:

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

';'可暂不考虑

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

该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

其属性翻译文法为:

<declaration>→DECLARE ‘(’ <entity list> ‘)’ @dcl_dy$\uparrow_x$

<type> $\uparrow_t$ @fix_up$\downarrow_{x,t}$

<entity list>→<entity name> $\uparrow_n$ @name_defn $\downarrow_n$

| <entity name> $\uparrow_n$, @entity_listfn $\downarrow_n$

<type> $\uparrow_t$→FIXED $\uparrow_t$ | FLOAT $\uparrow_t$ | CHAR $\uparrow_t$

<declaration>→DECLARE @dec_on$_{\uparrow x}$'('<entity list> ')' <type> $_{\uparrow t}$ @fix_up$_{\downarrow x,\,t}$
<entity list>→<entity name> $_{\uparrow n}$ @name_defn $_{\downarrow n}$
| <entity name> $_{\uparrow n}$, @name_defn $_{\downarrow n}$ <entity list>
<type> $_{\uparrow t}$→FIXED $_{\uparrow t}$ | FLOAT $_{\uparrow t}$ | CHAR $_{\uparrow t}$

动作程序

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

@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** 。

@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。

@fix_up$_{\downarrow x,\,t}$ 是将类型信息 **t** 填入从 **x** 位置开始到相应的数据存储区分配地址的符号表中。（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。

<declaration>→DECLARE @dec_on$_{\uparrow x}$‘(’<entity list> ‘)’ <type> $_{\uparrow t}$ @fix_up$_{\downarrow x,\, t}$

<entity list>→<entity name> $_{\uparrow n}$ @name_defn $_{\downarrow n}$ |

<entity name> $_{\uparrow n}$, @name_defn $_{\downarrow n}$ <entity list>

<type> $_{\uparrow t}$→FIXED $_{\uparrow t}$ | FLOAT $_{\uparrow t}$ | CHAR $_{\uparrow t}$

DECLARE(A, B, C) FLOAT;

<declaration> ⇒ DECLARE @dec_on$_{\uparrow x}$‘(’<entity list> ‘)’ <type>$_{\uparrow t}$ @fix_up$_{\downarrow x,\, t}$

⇒DECLARE @dec_on$_{\uparrow x}$‘(’ <entity name>$_{\uparrow n}$, @name_defn$_{\downarrow n}$ <entity list> ‘)’ <type>$_{\uparrow t}$ @fix_up$_{\downarrow x,\, t}$

⇒DECLARE @dec_on$_{\uparrow x}$‘(’ A , @name_defn$_{\downarrow A}$ <entity list> ‘)’ <type>$_{\uparrow t}$ @fix_up$_{\downarrow x,\, t}$

⇒DECLARE @dec_on$_{\uparrow x}$‘(’ A , @name_defn$_{\downarrow A}$ <entity name>$_{\uparrow n}$, @name_defn$_{\downarrow n}$ <entity list>‘)’ <type>$_{\uparrow t}$ @fix_up$_{\downarrow x,\, t}$

⇒DECLARE @dec_on$_{\uparrow x}$‘(’ A , @name_defn$_{\downarrow A}$ B , @name_defn$_{\downarrow B}$ <entity list>‘)’ <type>$_{\uparrow t}$ @fix_up$_{\downarrow x,\, t}$

⇒DECLARE @dec_on$_{\uparrow x}$‘(’ A , @name_defn$_{\downarrow A}$ B , @name_defn$_{\downarrow B}$ <entity name>$_{\uparrow n}$ @name_defn$_{\downarrow n}$‘)’ <type>$_{\uparrow t}$ @fix_up$_{\downarrow x,\, t}$

⇒DECLARE @dec_on$_{\uparrow x}$‘(’ A , @name_defn$_{\downarrow A}$ B , @name_defn$_{\downarrow B}$ C @name_defn$_{\downarrow C}$‘)’ <type>$_{\uparrow t}$ @fix_up$_{\downarrow x,\, t}$

⇒DECLARE @dec_on$_{\uparrow x}$‘(’ A , @name_defn$_{\downarrow A}$ B , @name_defn$_{\downarrow B}$ C @name_defn$_{\downarrow C}$‘)’ FLOAT @fix_up$_{\downarrow x,\, \mathrm{FLOAT}}$

10.3.1 常量类型声明处理

常量标识符通常被看作是全局名。

常量声明的ATG如下:

<const del> →constant <type>$_{\uparrow t}$ <entity>$_{\uparrow n}$:= <const expr>$_{\uparrow c, s}$
@insert$_{\downarrow t,\ n,\ c,\ s}$;

<type>$_{\uparrow t}$ → real$_{\uparrow t}$ | integer$_{\uparrow t}$ | string$_{\uparrow t}$

<const expr>$_{\uparrow c, s}$ →<integer const >$_{\uparrow c, s}$ | <real const>$_{\uparrow c, s}$
| <string const>$_{\uparrow c, s}$

由该文法产生的一个声明实例为:

constant integer SYMBSIZE := 1024;

由该文法产生的一个声明实例为:

constant integer SYMBSIZE := 1024;

翻译处理过程为:

<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（SYMBSIZE)，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 填入全局符号表中。

10.3.2 简单变量声明处理

ATG文法:

<svar del> → <type> $_{\uparrow t, i}$ <entity> $_{\uparrow n}$ @svardef $_{\downarrow t, i, n}$ @allocsv$_{\downarrow i}$;

<type>$_{\uparrow t, i}$ → real$_{\uparrow t, i}$ |integer$_{\uparrow t, i}$ |character$_{\uparrow t}$ (<number>$_{\uparrow i}$) | logical$_{\uparrow t, i}$

其中

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 ;
```

<svar del> → <type> ↑t , i <entity> ↑n @svardef ↓ t, i, n @allocsv↓i

<type> ↑t, i → real ↑t, i |integer ↑t, i |character ↑t (<number>↑i) | logical ↑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 ( duplicate , statementno);
    else  if   j = -1                   //符号表已满
         then errmsg( tblovflow, statementno);
         abort;                         //符号表溢出，编译失败，终止。
end svardef;
```

@allocsv↓i 为简单变量分配数据空间

```
procedure allocsv( i );
    codeptr := codeptr + i ;    //codeptr 为地址区指针
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），往往需要采用动态申请存储空间的办法把可变长实体存储在堆中。

我们可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，并一起填入符号表内。

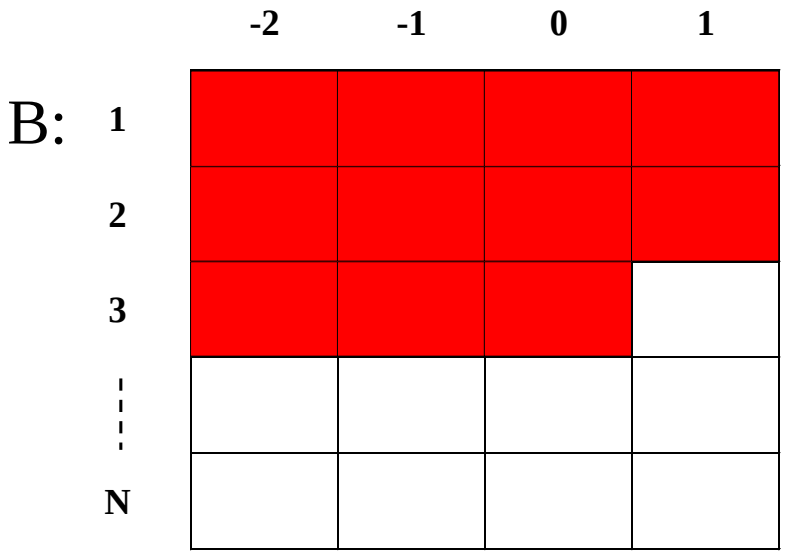
即可变长实体存储在堆中，而其指针存放在符号表中。

10.3.3 数组变量声明的处理

对于静态数组，即数组的大小在编译时是已知的，编译程序在处理数组声明时，可建立一个数组模板（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。

对于动态数组，其大小只有在运行时才能最后确定。我们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 ;



实际数组B各元素的存储次序为:

LOC→

B(1,-2)
B(1,-1)
B(1,0)
B(1,1)
B(2,-2)
B(2,-1)
.
.
.
B(N,1)

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

a）n维数组的地址计算公式

设数组的维数为n， 各维的下界和上界为L(i)和U(i)

例如， 上例二维数组B

L(1) = 1 (隐含值）, U(1) = N

L(2) = -2 , U(2) = 1

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

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:

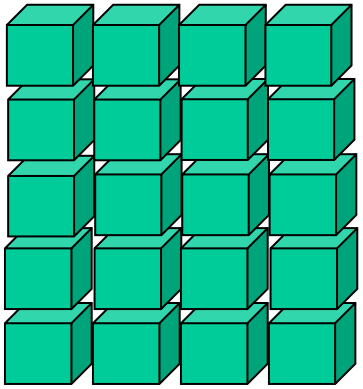
$$ADR = LOC + \sum_{i=1}^{n}[V(i) - L(i)] \times P(i) \times E$$

注：E为数组元素大小（字节数）

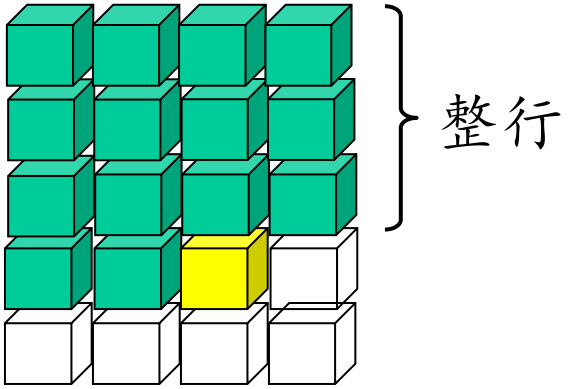
其中

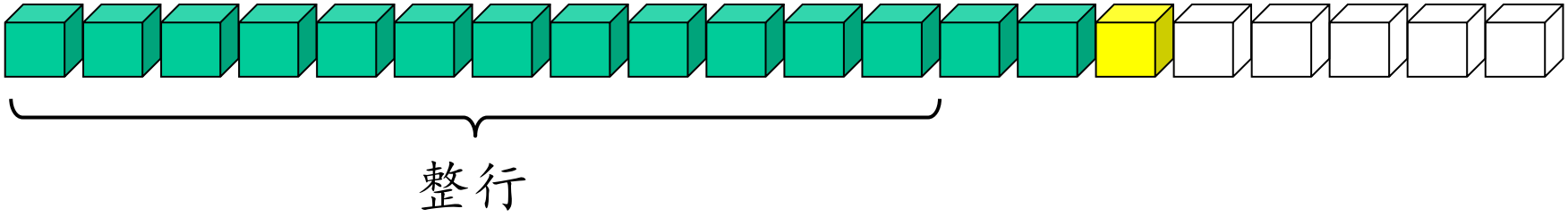
$$P(i) = \begin{cases} 1 & \text{当 } i = n \text{ 时} \\ \prod_{j=i+1}^{n}[U(j) - L(j) + 1] & \text{当 } 1 \le i < n \text{ 时} \end{cases}$$

二维变一维

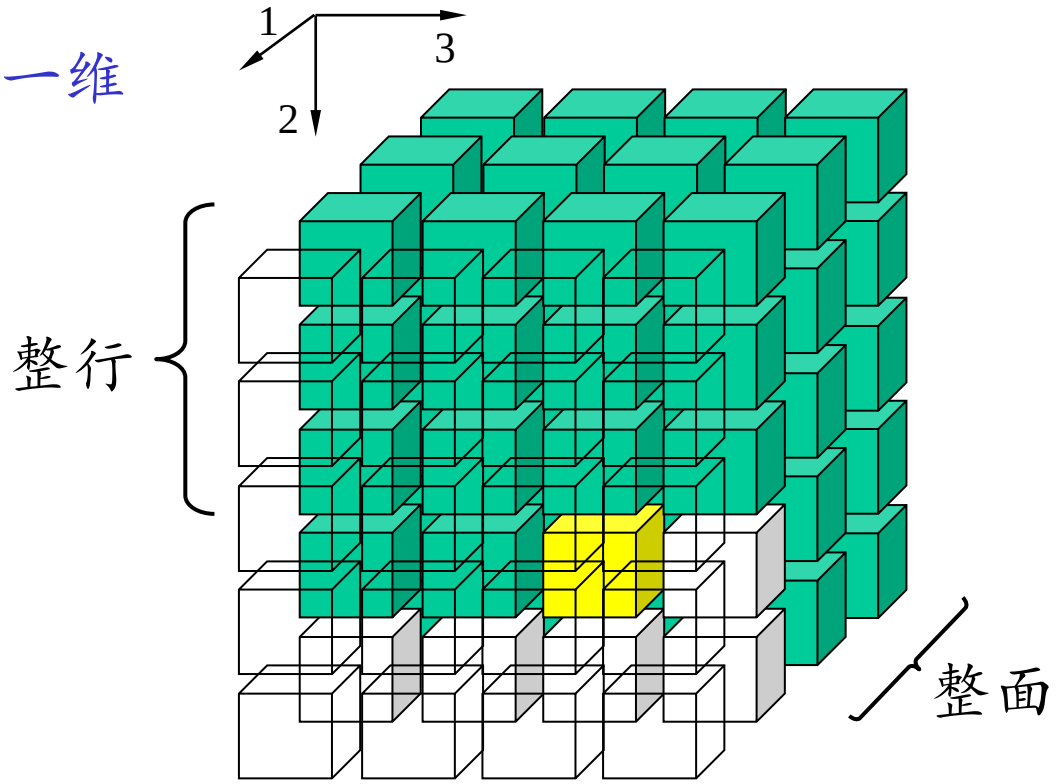


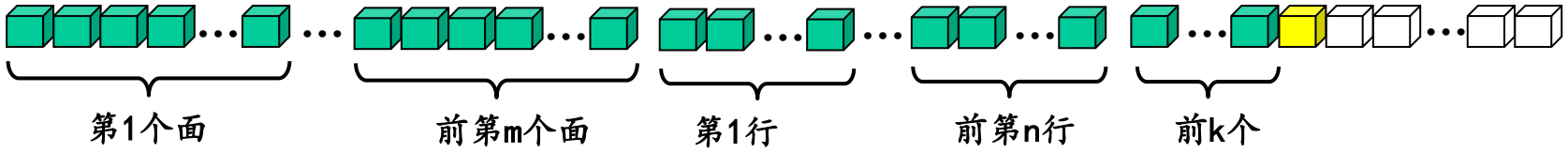
二维变一维
的地址计算

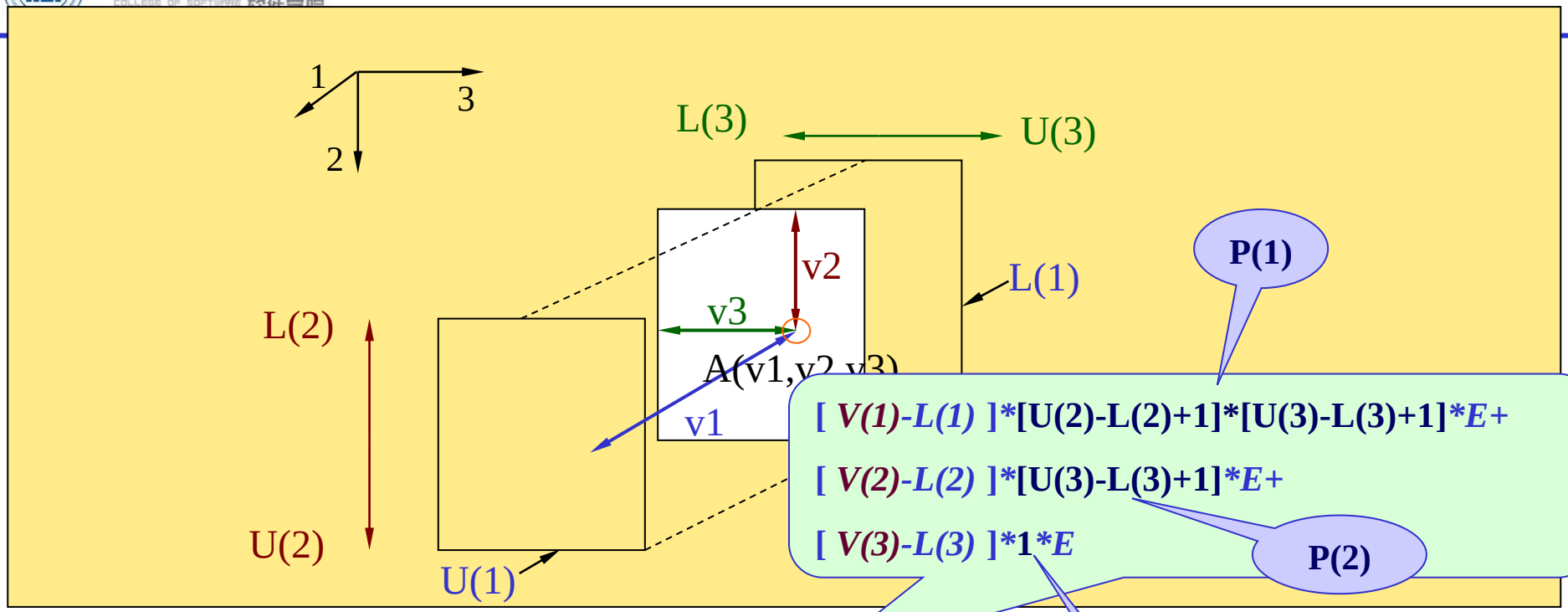




三维变一维







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

注：E为数组元素大小（字节数）

其中

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

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

其中

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

若令 (不变部分)

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

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

RC为数组元素地址计算公式中的不变部分。因为只要知道数组的维数和每一维的上下界值，便可求得RC值。

以前面所举的二维数组B为例，若N = 3

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

则 P(1) = [U(2) - L(2) +1]
= 1- (-2)+1
= 4

P(2) = 1

$$RC = -\sum_{i=1}^{2} L(i)P(i) * E$$

= -[1 × 4 + (-2) × 1] × E
= -2E

因此， 若有数组元素B(2 , 1), 则它的地址为:

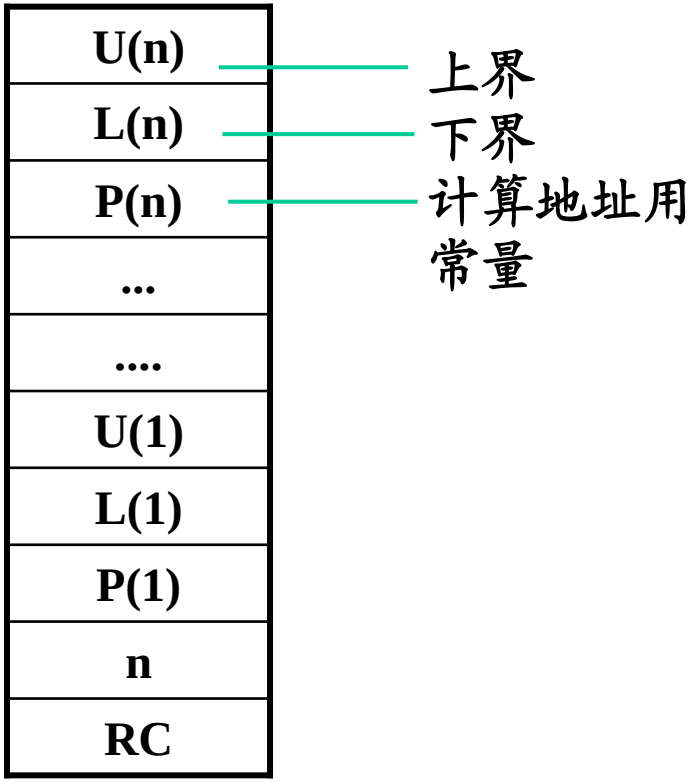
$$ADR = LOC - 2E + \sum_{i=1}^{2} V(i) \times P(i) \times E = LOC - 2E + (2 \times 4 + 1 \times 1) \times E$$
$$= LOC + 7 \times E$$

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构造。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

P(1) = [U(2) - L(2) +1]
= 1- (-2)+1
= 4

$$RC = -\sum_{i=1}^{2} L(i)P(i)$$

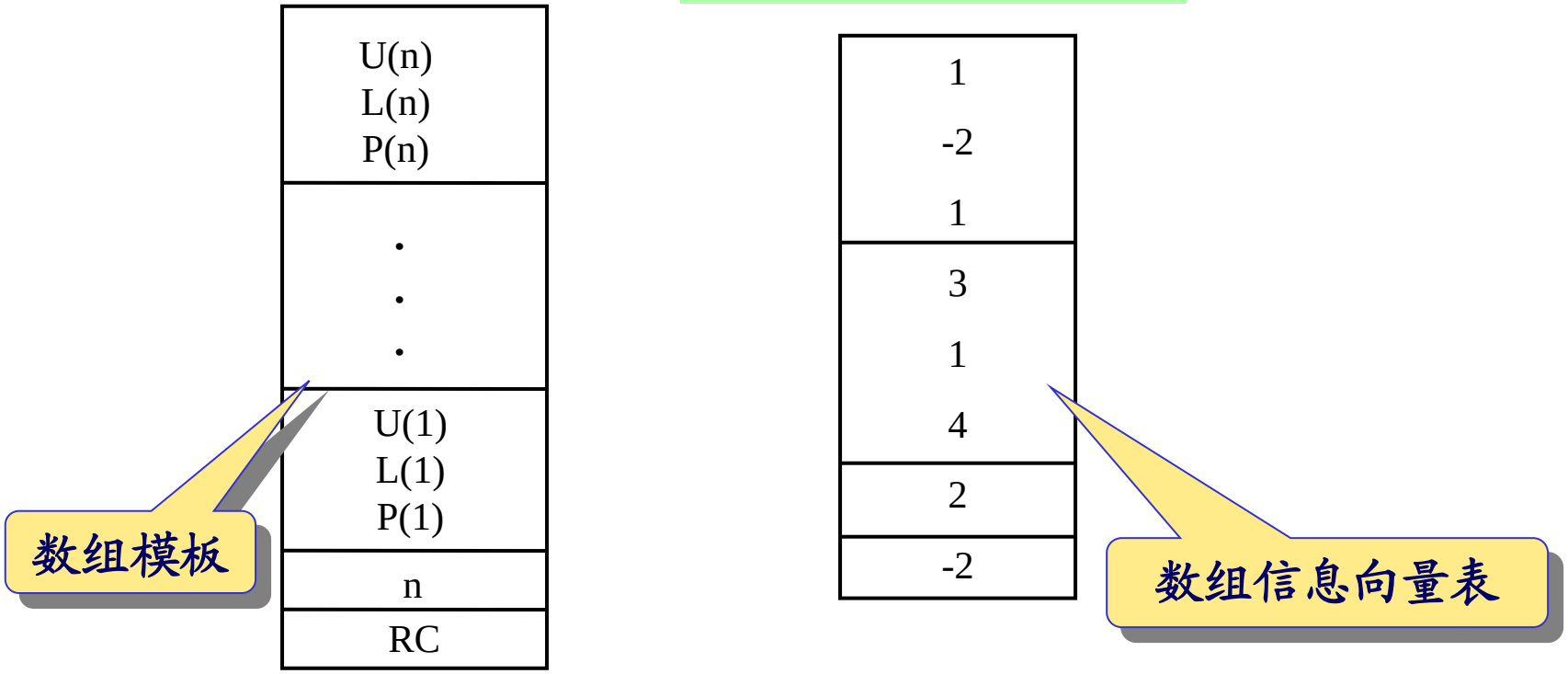
= -[1 × 4 + (-2) × 1]
= -2

数组信息向量表

1	U(2)--上界
-2	L(2)--下界
1	P(2)--计算地址常量
3	U(1)--上界
1	L(1)--下界
4	P(1)--计算地址常量
2	n---维数
-2	RC

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：

array B (3, -2: 1) char ;



数组声明的输入文法:

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

<array del> →array <entity> (<sublist>) <type>

<sublist> →<subscript> | <subscript>, <sublist>

<subscript> →<integer expr> | <integer expr> : <integer expr>

数组声明的ATG文法:

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

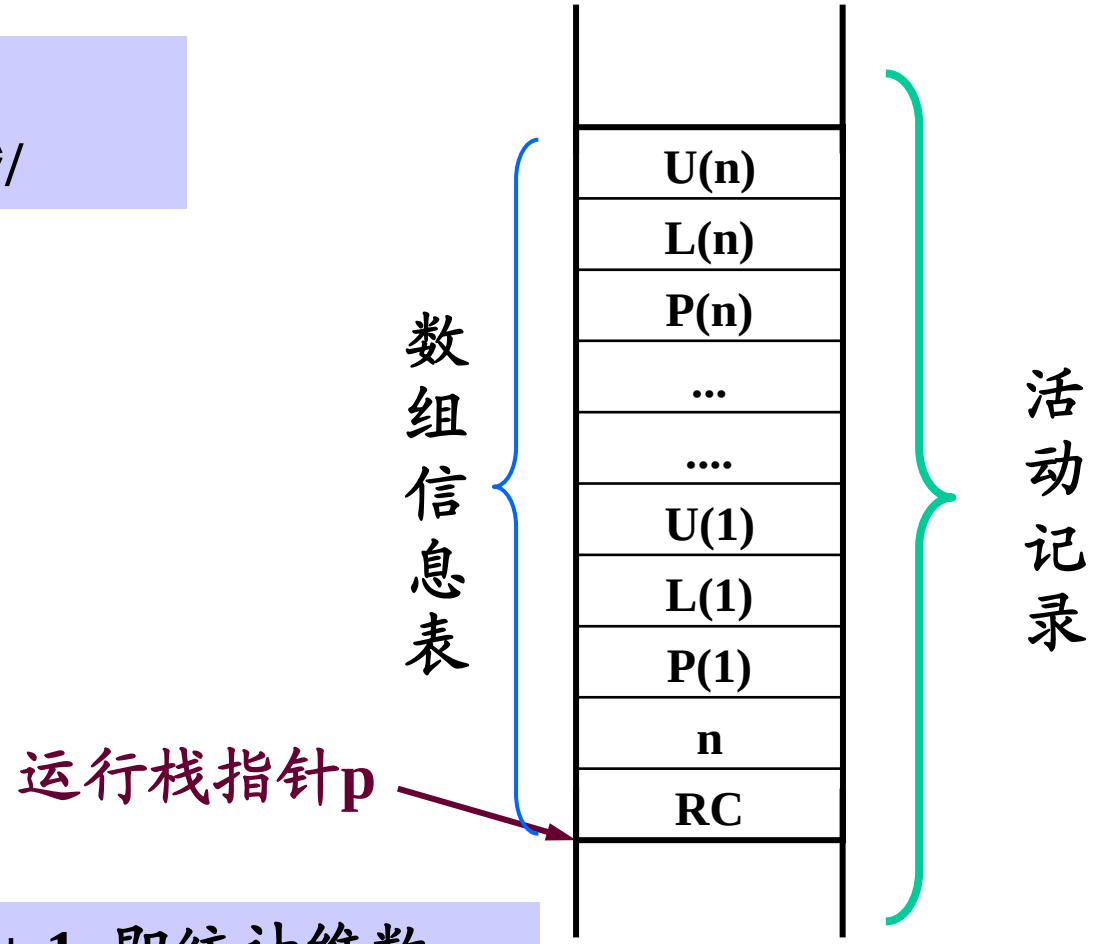
<array del> →array $_{\uparrow k}$ @init$_{\uparrow j}$ <entity>$_{\uparrow n}$ (<sublist>$_{\uparrow j}$)
<type>$_{\uparrow t}$ @symbinsert$_{\downarrow j, n, t, k}$

<sublist> $_{\uparrow j}$ →<subscript> @dimen#$_{\uparrow j}$
| <subscript>, <sublist>$_{\uparrow j}$ @dimen#$_{\uparrow j}$

<subscript> →<integer expr>$_{\uparrow u}$ @bannds$_{\downarrow u}$
| <integer expr>$_{\uparrow l}$: @lowerbnd$_{\downarrow l}$
<integer expr>$_{\uparrow u}$ @upperbnd$_{\downarrow u, l}$

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。

```
p:= p+2;
j := 0;     /*维数计数器*/
```



2) @dimen#$_{\uparrow j}$: j := j + 1, 即统计维数

<array del> →array $_{\uparrow k}$ @init$_{\uparrow j}$ <entity>$_{\uparrow n}$ (<sublist>$_{\uparrow j}$)
<type>$_{\uparrow t}$ @symbinsert$_{\downarrow j, n, t, k}$

<subscript> →<integer expr>↑u @bannds↓u
| <integer expr>↑l : @lowerbnd↓l
<integer expr>↑u @upperbnd↓u, l

3) @bannds将省略下界表达式情况的u => 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] × P(i+1)

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

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 => L(i)
@upperbnd 把 u => 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$_{\downarrow l}$

生成将 l => L(i) 的代码

@upperbnd$_{\downarrow u}$

生成 把 u => 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$_{\downarrow j, n, t, k}$

a) 把j, n, 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。）

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：

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>$_{\uparrow n}$@lookup$_{\downarrow n\uparrow j}$ @push$_{\downarrow j}$
11. | <integer>$_{\uparrow i}$@pushi$_{\downarrow i}$
12. | (<expr>)

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;
   emit('LDC', i) ;
end;
```

对于输入表达式 x + y * 3:

<expression>
=> <expr>
=> <term><terms>
=> <factor><factors><terms>
=> <variable><factors><terms>
=> <variable> ε <terms>
=> <variable> <terms>
=> <variable> + <term><terms>
=> <variable> + <term> ε
=> <variable> + <term>
=> <variable> + <factor><factors>
=> <variable> + <variable><factors>
=> <variable> + <variable>*<factor><factors>
=> <variable> + <variable>*<factor> ε
=> <variable> + <variable>*<factor>
=> <variable> + <variable>*<integer>

1.<expression>→<expr>
2.<expr>→<term><terms>
3.<terms>→ε
4. | +<term><terms>
5. | - <term><terms>
6.<term>→<factor><factors>
7.<factors>→ε
8. | *<factor><factors>
9. | /<factor><factors>
10.<factor>→<variable>
11. | <integer>
12. | (<expr>)

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>$_{\uparrow n}$@lookup$_{\downarrow n\uparrow j}$ @push$_{\downarrow j}$
11. | <integer>$_{\uparrow i}$@pushi$_{\downarrow i}$
12. | (<expr>)

对于输入表达式 x + y * 3:

<expression>
=> <expr>
=> <term><terms>
=> <factor><factors><terms>
=> <variable>$_{\uparrow n}$@lp$_{\downarrow n\uparrow j}$@ph$_{\downarrow j}$<factors><terms>
=> <variable>$_{\uparrow n}$@lp$_{\downarrow n\uparrow j}$@ph$_{\downarrow j}$<terms>
=> <variable>$_{\uparrow n}$@lp$_{\downarrow n\uparrow j}$@ph$_{\downarrow j}$ +<term>@add<terms>
=> <variable>$_{\uparrow n}$@lp$_{\downarrow n\uparrow j}$@ph$_{\downarrow j}$ +<factor><factors>@add<terms>
=> <variable>$_{\uparrow n}$@lp$_{\downarrow n\uparrow j}$@ph$_{\downarrow j}$ +<variable>$_{\uparrow n}$@lp$_{\downarrow n\uparrow j}$@ph$_{\downarrow j}$<factors>@add<terms>
=> <variable>$_{\uparrow n}$@lp$_{\downarrow n\uparrow j}$@ph$_{\downarrow j}$ +<variable>$_{\uparrow n}$@lp$_{\downarrow n\uparrow j}$@ph$_{\downarrow j}$*<factor>@mul<factors>@add<terms>
=> <variable>$_{\uparrow n}$@lp$_{\downarrow n\uparrow j}$@ph$_{\downarrow j}$ +<variable>$_{\uparrow n}$@lp$_{\downarrow n\uparrow j}$@ph$_{\downarrow j}$*<integer>$_{\uparrow i}$@phi$_{\downarrow i}$@mul<factors>
@add<terms>
=> <variable>$_{\uparrow n}$@lp$_{\downarrow n\uparrow j}$@ph$_{\downarrow j}$ +<variable>$_{\uparrow n}$@lp$_{\downarrow n\uparrow j}$@ph$_{\downarrow j}$*<integer>$_{\uparrow i}$@phi$_{\downarrow i}$@mul@add<terms>
=> <variable>$_{\uparrow n}$@lp$_{\downarrow n\uparrow j}$@ph$_{\downarrow j}$ +<variable>$_{\uparrow n}$@lp$_{\downarrow n\uparrow j}$@ph$_{\downarrow j}$*<integer>$_{\uparrow i}$@phi$_{\downarrow i}$@mul@add

LOD, < ll, on>$_x$ LOD, < ll, on>$_y$ LDC, 3 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>$_{\uparrow n}$@lookup$_{\downarrow n\uparrow j}$ @push$_{\downarrow j}$
11. | <integer>$_{\uparrow i}$@pushi$_{\downarrow i}$
12. | (<expr>)

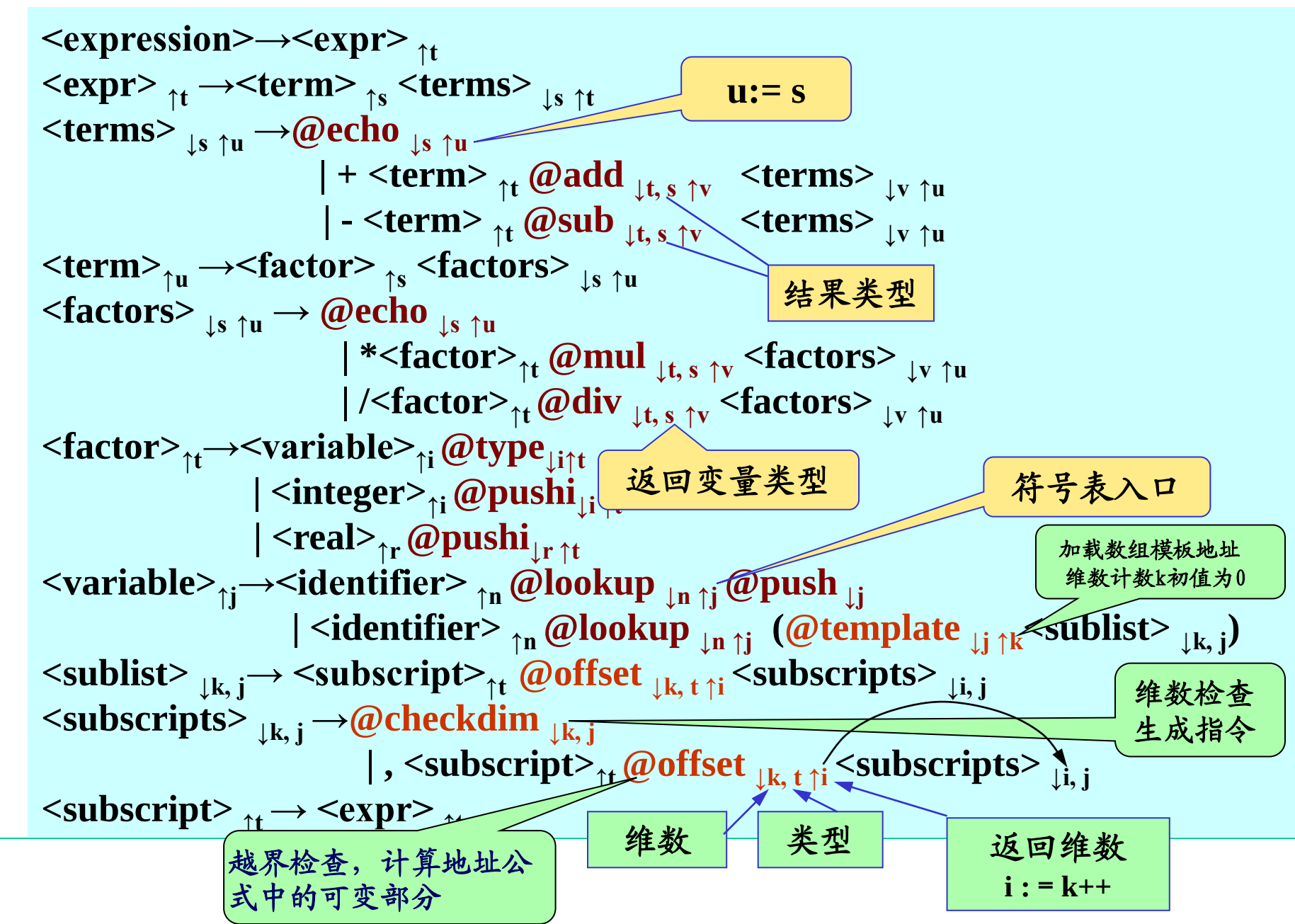
对于输入表达式 x + y * 3:

<expression>
=> <expr>
=> <term><terms>
=> <factor><factors><terms>
=> <variable>$_{\uparrow n}$@lp$_{\downarrow n\uparrow j}$@ph$_{\downarrow j}$<factors><terms>
=> <variable>$_{\uparrow n}$@lp$_{\downarrow n\uparrow j}$@ph$_{\downarrow j}$<terms>
=> <variable>$_{\uparrow n}$@lp$_{\downarrow n\uparrow j}$@ph$_{\downarrow j}$ +<term>@add<terms>
=> <variable>$_{\uparrow n}$@lp$_{\downarrow n\uparrow j}$@ph$_{\downarrow j}$ +<factor><factors>@add<terms>
=> <variable>$_{\uparrow n}$@lp$_{\downarrow n\uparrow j}$@ph$_{\downarrow j}$ +<variable>$_{\uparrow n}$@lp$_{\downarrow n\uparrow j}$@ph$_{\downarrow j}$<factors>@add<terms>
=> <variable>$_{\uparrow n}$@lp$_{\downarrow n\uparrow j}$@ph$_{\downarrow j}$ +<variable>$_{\uparrow n}$@lp$_{\downarrow n\uparrow j}$@ph$_{\downarrow j}$*<factor>@mul<factors>@add<terms>
=> <variable>$_{\uparrow n}$@lp$_{\downarrow n\uparrow j}$@ph$_{\downarrow j}$ +<variable>$_{\uparrow n}$@lp$_{\downarrow n\uparrow j}$@ph$_{\downarrow j}$*<integer>$_{\uparrow i}$@phi$_{\downarrow i}$@mul<factors>
@add<terms>
=> <variable>$_{\uparrow n}$@lp$_{\downarrow n\uparrow j}$@ph$_{\downarrow j}$ +<variable>$_{\uparrow n}$@lp$_{\downarrow n\uparrow j}$@ph$_{\downarrow j}$*<integer>$_{\uparrow i}$@phi$_{\downarrow i}$@mul@add<terms>
=> <variable>$_{\uparrow n}$@lp$_{\downarrow n\uparrow j}$@ph$_{\downarrow j}$ +<variable>$_{\uparrow n}$@lp$_{\downarrow n\uparrow j}$@ph$_{\downarrow j}$*<integer>$_{\uparrow i}$@phi$_{\downarrow i}$@mul@add

LOD, < ll, on>$_x$ LOD, < ll, on>$_y$ 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。


<expression>→<expr> ↑t
<expr> ↑t →<term> ↑s <terms> ↓s ↑t
<terms> ↓s ↑u →@echo ↓s ↑u
| + <term> ↑t @add ↓t, s ↑v <terms> ↓v ↑u
| - <term> ↑t @sub ↓t, s ↑v <terms> ↓v ↑u
<term> ↑u →<factor> ↑s <factors> ↓s ↑u
<factors> ↓s ↑u → @echo ↓s ↑u
| *<factor> ↑t @mul ↓t, s ↑v <factors> ↓v ↑u
| /<factor> ↑t @div ↓t, s ↑v <factors> ↓v ↑u
<factor> ↑t→<variable> ↑i @type ↓i↑t
| <integer> ↑i @pushi ↓i ↑t
| <real> ↑r @pushi ↓r ↑t
<variable> ↑j→<identifier> ↑n @lookup ↓n ↑j @push ↓j
| <identifier> ↑n @lookup ↓n ↑j (@template ↓j ↑k <sublist> ↓k, j)
<sublist> ↓k, j→ <subscript> ↑t @offset ↓k, t ↑i <subscripts> ↓i, j
<subscripts> ↓k, j →@checkdim ↓k, j
| , <subscript> ↑t @offset ↓k, t ↑i <subscripts> ↓i, j
<subscript> ↑t → <expr> ↑t
u:= s
结果类型
返回变量类型
符号表入口
加载数组模板地址
维数计数k初值为0
维数检查
生成指令
越界检查，计算地址公式中的可变部分
维数
类型
返回维数
i : = k++

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

```
procedure offset( k, t );
     integer k;  string t;
     k := k+1;
     if t ≠ ‘ integer’
      then errmsg(‘数组下标应为整
                  型表达式’, statno);
     else  emit( ‘OFS’, k );
     return (k);
end;
```

越界检查，计算地址公式中的可变部分

```
procedure checkdim( k, j);
     integer k, j;
     if k ≠ symbtbl (j).dim
     then errmsg(‘数组维数与
          声明不匹配’, statno);
     else  begin
             emit( ‘ARR’);
             emit( ‘DER’);
           end;
end;
```

生成数组元素地址

加载数组元素内容

```
procedure template(j);
    integer j;
    emit( ‘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:

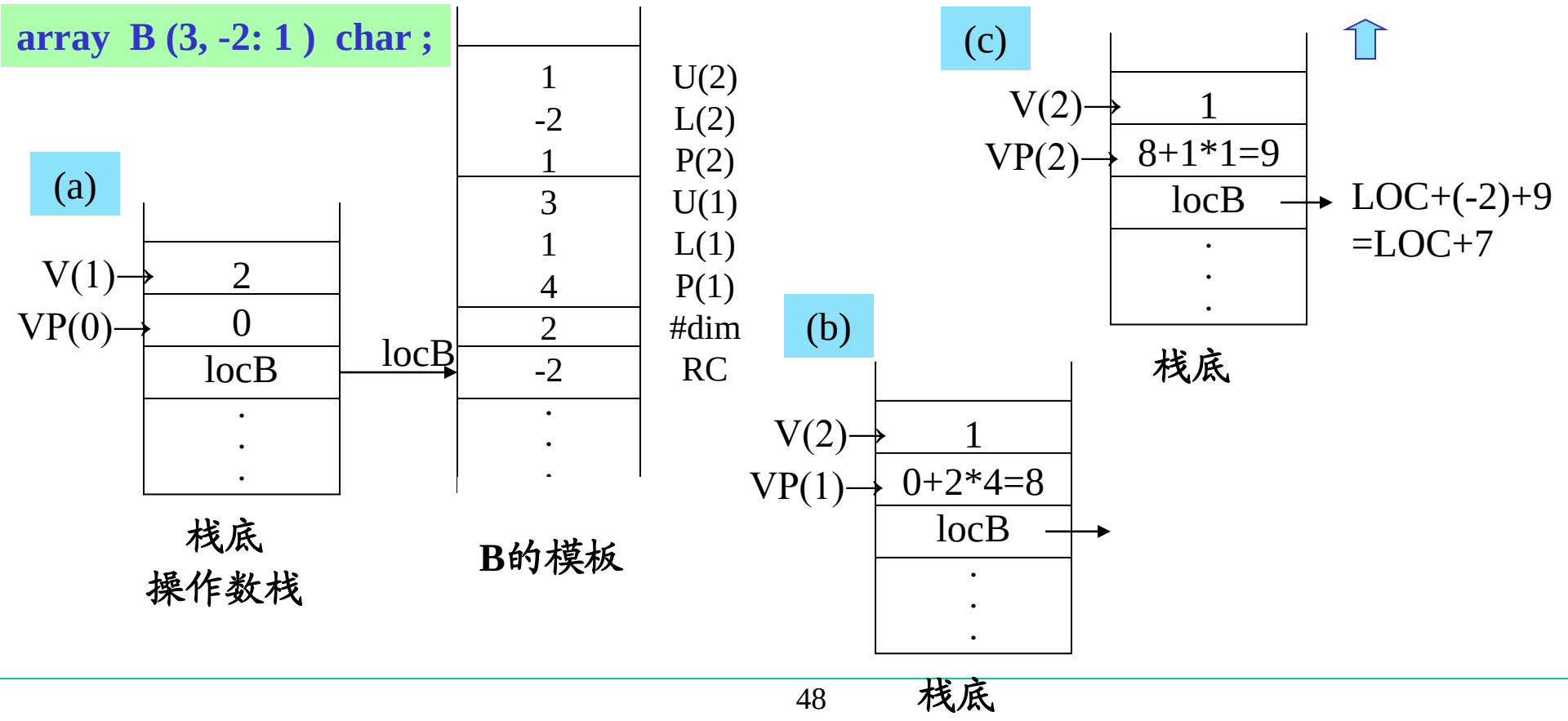
VP(0) = 0;

VP(k) = VP(k-1) + V(k) *P(k)　　1≤k≤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

array B (3, -2: 1) char ;



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 ~ (x =y & y≠z | 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
```

<assignstat > → @setL ↑L <variable>↓L↑t := @resetL↑L <expr> ↑s @storin↓t, s ;

置“左值”特征L为真

被赋变量类型

类型转换，生成STN指令

置“左值”特征L为假

表达式类型

@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

JMF, labA

IF <log_expr> THEN <stat_list> labA:

JMF, labA

IF <log_expr> THEN <stat_list> ELSE labA: <stat_list> labB:

JMP, labB

其属性翻译文法及相应的语义动作程序：

1. <if_stat>→<if_head>$_{\uparrow y}$ <if_tail>$_{\downarrow y}$

2. <if_head>$_{\uparrow y}$→IF <log_expr>@brf$_{\uparrow y}$ THEN <stat list>

3. <if_tail>$_{\downarrow y}$→@labprod$_{\downarrow y}$
   |ELSE @br$_{\uparrow z}$@labprod$_{\downarrow y}$<stat_list>@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*/
   emit('JMF', labx);
   return (labx);
end;
```

动作程序@br是是生成JMP指令，并将转移标号返回给属性z

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

1. <if_stat>→<if_head>$_{\uparrow y}$ <if_tail>$_{\downarrow y}$

2. <if_head>$_{\uparrow y}$→IF <log_expr>@brf$_{\uparrow y}$ THEN <stat list>

3. <if_tail>$_{\downarrow y}$→@labprod$_{\downarrow y}$
   |ELSE @br$_{\uparrow z}$@labprod$_{\downarrow y}$<stat_list>@labprod$_{\downarrow z}$

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.<for loop>→<for head>$_{\uparrow a, f, r}$ < rest of loop>$_{\downarrow a, f, r}$

2.<for head>$_{\uparrow a, f, r}$ →for <id>$_{\uparrow a}$:= <expr> @initload$_{\uparrow s}$
to @labgen$_{\uparrow r}$ <expr> by
@loadid$_{\downarrow a}$<expr> @compare$_{\downarrow a, s \uparrow f}$

3.<rest of loop>$_{\downarrow a, f, r}$ →do <stat list> end for
@retbranch$_{\downarrow r}$ @labemit$_{\downarrow f}$

@initload 只生成给循环变量赋初值的指令。

for <id> := <expr1> to <expr2> by <expr3> do <stat list>

LDA, (<id>)

LOD, (expr1)

@initload$_{\uparrow s}$ STN

JMP, start

@labgen$_{\uparrow r}$ loop:

LOD, (expr2)

@loadid$_{\downarrow a}$ LOD, (id)

LOD, (expr3)

@compare$_{\downarrow a,\ s\uparrow f}$ ADD

STO, (id)

BGT, end_loop

start: <statement>

…

@retbranch$_{\downarrow r}$ JMP, loop

@labprod$_{\downarrow f}$ end_ loop:

1.<for loop>→<for head>$_{\uparrow a,\ f,\ r}$ < rest of loop>$_{\downarrow a,\ f,\ r}$

2.<for head>$_{\uparrow a,\ f,\ r}$ →for <id>$_{\uparrow a}$:= <expr> @initload$_{\uparrow s}$ to @labgen$_{\uparrow r}$ <expr> by @loadid$_{\downarrow a}$<expr> @compare$_{\downarrow a,\ s\uparrow f}$

3.<rest of loop>$_{\downarrow a,\ f,\ r}$ →do <stat list> end for @retbranch$_{\downarrow r}$ @labprod$_{\downarrow f}$

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

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

```
procedure compare( a, s);
   address a;    string f, s;
   emit( 'ADD');
   emit( 'STO', a );
   f := genlab;
   emit( '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 => 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 => 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

如：ALGOL 的换名形参。

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：

I ： 2

A[1]: 6

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

传名：

I ： 2

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

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

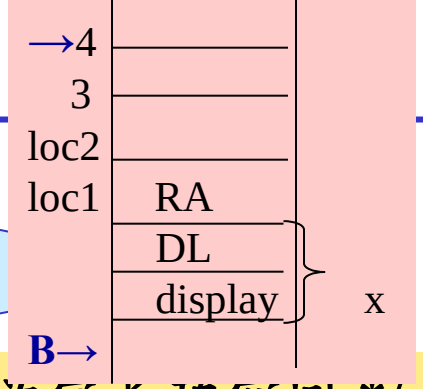
若实参并非上例中所示变量，而是表达式，则应生成相应计算实参表达式值的指令序列。

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。

设过程说明的首部有如下形式:

procedure process_symb(string:

x: display+DL



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）

过程调用的ATG文法:

实参个数计数，m初值为0

过程名符号表入口位置

形参数目

<proc call>→ <call head>$_{\uparrow i,z}$ @initm$_{\uparrow m}$ <args> $_{\downarrow i,z}$ @genjsr$_{\downarrow i}$

<call head>$_{\uparrow i,z}$ → <id>$_{\uparrow n}$ @lookupproc$_{\downarrow n \uparrow i,z}$

<args>$_{\downarrow i,z}$ → @chklength$_{\downarrow i,z}$ | (<arg list>$_{\downarrow i,z}$)

<arg list>$_{\downarrow i,z}$ → <expr>$_{\uparrow t}$ @chktype$_{\downarrow t,i,m,z \uparrow z}$ <exprs>$_{\downarrow i,z}$

<exprs>$_{\downarrow i,z}$ → @chklength$_{\downarrow i,z}$

| , <expr>$_{\uparrow t}$ @chktype$_{\downarrow t,i,m,z \uparrow z}$ <exprs>$_{\downarrow i,z}$

```
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;
```

<proc call>→ <call head>$_{\uparrow i,z}$ @initm$_{\uparrow m}$ <args>$_{\downarrow i,z}$ @genjsr$_{\downarrow i}$
<call head>$_{\uparrow i,z}$ → <id>$_{\uparrow n}$ @lookupproc$_{\downarrow n \uparrow i,z}$
<args>$_{\downarrow i,z}$ → @chklength$_{\downarrow i,z}$ | (<arg list>$_{\downarrow i,z}$)
<arg list>$_{\downarrow i,z}$ → <expr>$_{\uparrow t}$ @chktype$_{\downarrow t,i,m,z\uparrow z}$ <exprs>$_{\downarrow i,z}$
<exprs>$_{\downarrow i,z}$ → @chklength$_{\downarrow i,z}$
| , <expr>$_{\uparrow t}$ @chktype$_{\downarrow t,i,m,z\uparrow z}$ <exprs>$_{\downarrow i,z}$

m ++, z --

```
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;
```

LOD, (addr of symb)
LOD, (addr of cursor)
LOD, (addr of replacestr)
JSR, (addr of process_symb)
<retaddr>:….

@chklenth 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 symbtbl [i] .addr

过程说明（定义）的ATG文法如下：

形参个数计数，j初值为0

<proc defn>→<proc defn head> @initcnt$_{\uparrow j}$
<parameters>$_{\downarrow j\uparrow k}$ @emitstores$_{\downarrow k}$

<proc defn head>→procedure$_{\uparrow t}$ <id>$_{\uparrow n}$ @tblinsert$_{\downarrow t,\, n}$

形参名填表

<parameters>$_{\downarrow j\uparrow k}$→@echo$_{\downarrow j\uparrow k}$ | (<parm list>$_{\downarrow j\uparrow k}$)

K := j ++

<parm list>$_{\downarrow j\uparrow l}$→<type>$_{\uparrow t}$: <id>$_{\uparrow n}$ @tblinsert$_{\downarrow t,\, n}$
@upcnt$_{\downarrow j\uparrow k}$ <parms>$_{\downarrow k\uparrow l}$

K := j

<parms>$_{\downarrow j\uparrow l}$→@echo$_{\downarrow j\uparrow l}$ | , <type>$_{\uparrow t}$: <id>$_{\uparrow n}$ @tblinsert$_{\downarrow t,\, n}$
@upcnt$_{\downarrow j\uparrow k}$ <parms>$_{\downarrow k\uparrow l}$

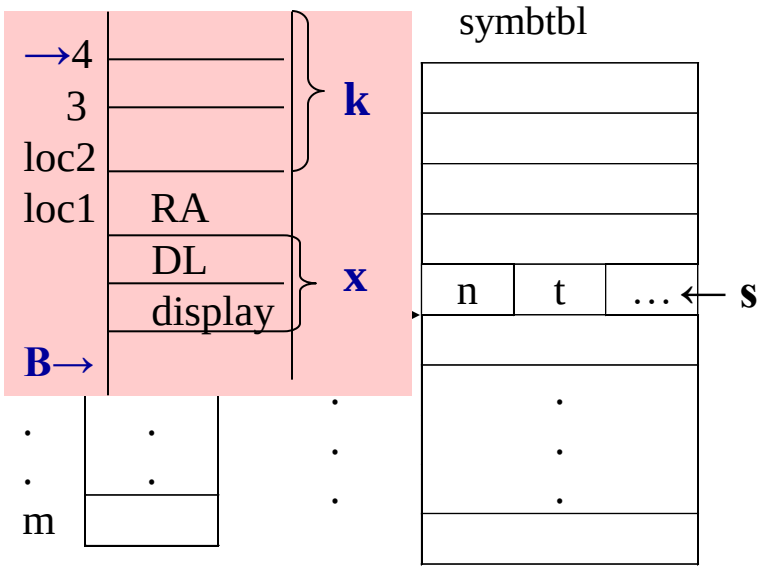
l := j

@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*/
  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;
```

最后得到的形参个数

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