编译技术

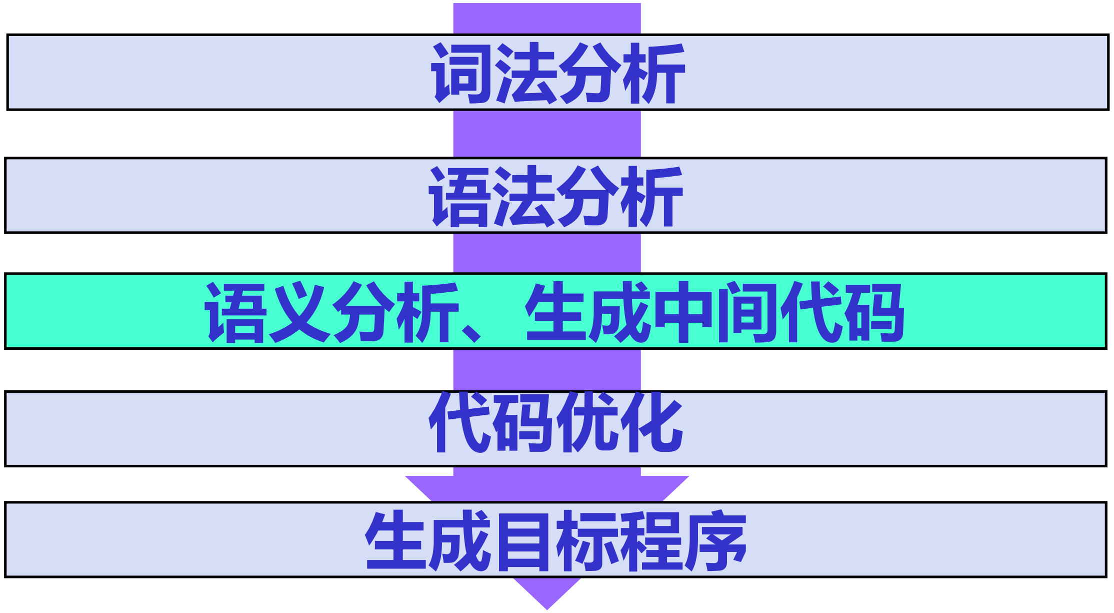
胡春明
hucm@buaa.edu.cn

2019.9-2019.12

编译过程

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：



词法分析

语法分析

语义分析、生成中间代码

代码优化

生成目标程序

北京航空航天大学计算机学院

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“需求分析”

文法

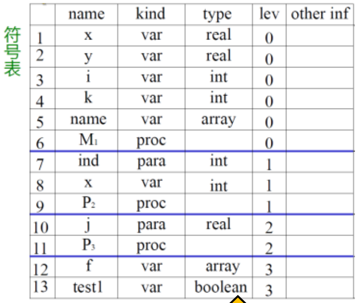
E::= E+T | T
T ::= T*F | F
F ::= (E) | i

```
int a;
int b;
int c;
int d;

a = b + c + d;
b = a * a + b * b;
```

源程序

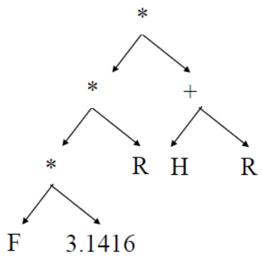
符号表
（编译时数据结构）



符号表

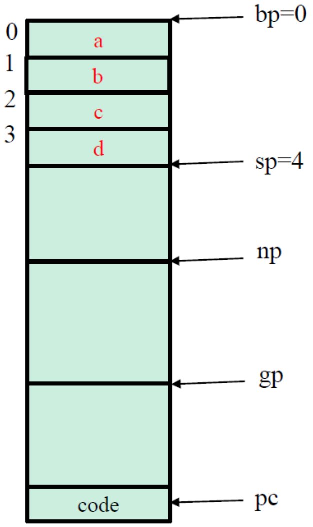
	name	kind	type	lev	other inf
1	x	var	real	0	
2	y	var	real	0	
3	i	var	int	0	
4	k	var	int	0	
5	name	var	array	0	
6	M_1	proc		0	
7	ind	para	int	1	
8	x	var	int	1	
9	P_2	proc		1	
10	j	para	real	2	
11	P_3	proc		2	
12	f	var	array	3	
13	test1	var	boolean	3	

Parsing
语法制导翻译（SDT）



抽象语法树

编译器
对中间语言抽象机及运行时存储组织的设想



LDA 1
LDA 2
ADI
LDA 3
ADI
STO 0
LDA 0
LDA 0
MULT
LDA 1
LDA 1
MULT
ADI
STO 1

中间代码

确保执行时符合编译器对运行组织的构想

第十章 语义分析和代码生成

• 声明的处理	符号表操作
• 表达式的处理	变量的引用/运行时语义检查
• 赋值语句的处理	写内存指令
• 控制语句的处理	控制流处理（标号/比较/跳转）
• 过程调用和返回	参数传递、运行栈操作

表达式及简单变量引用

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| +<term>@add<terms>

| - <term>@sub<terms>

4.<term>→<factor><factors>

5.<factors>→ ε|*<factor>@mul<factors>

| /<factor>@div<factors>

6.<factor>→<variable>$_{\uparrow n}$@lookup$_{\downarrow n \uparrow j}$

@push$_{\downarrow j}$| <integer>$_{\uparrow i}$@pushi$_{\downarrow i}$

| (<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;
    emitl(‘LDC’, i) ;
end;
```

对于输入表达式 **x + y * 3**：

1.<expression>→<expr>
2.<expr>→<term><terms>
3.<terms>→ε| +<term>@add<terms>
| - <term>@sub<terms>
4.<term>→<factor><factors>
5.<factors>→ ε|*<factor>@mul<factors>
| /<factor>@div<factors>
6.<factor>→<variable>$_{\uparrow n}$@lookup$_{\downarrow n\uparrow j}$
@push$_{\downarrow j}$| <integer>$_{\uparrow i}$@pushi$_{\downarrow i}$
| (<expr>)

<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

上面忽略了操作数的类型，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

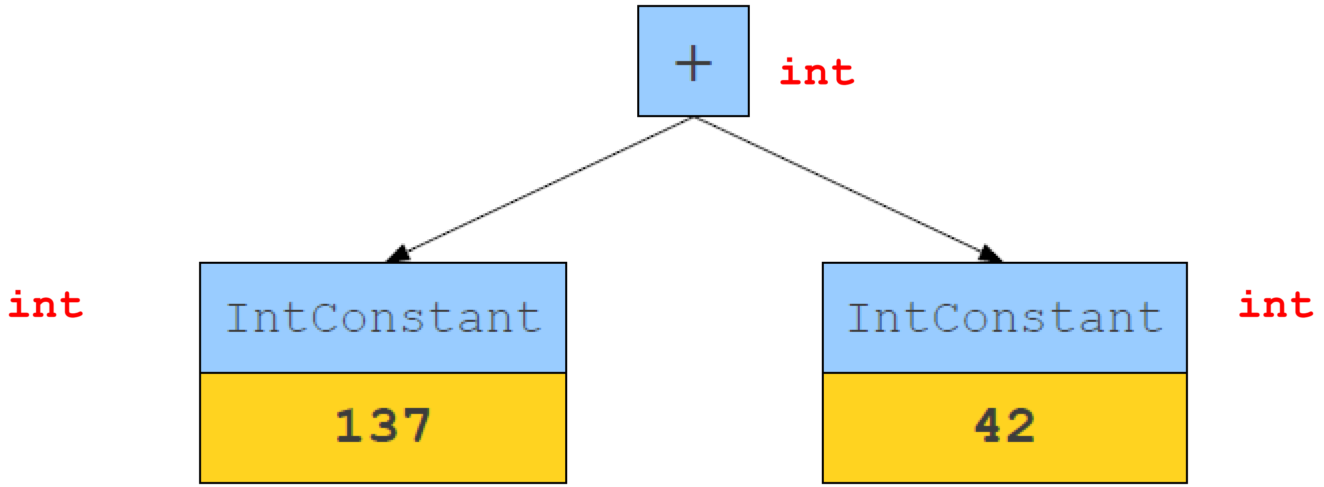
占用运行时间，影响效率（less efficient）

不做任何类型检查

主要靠对程序员的信任

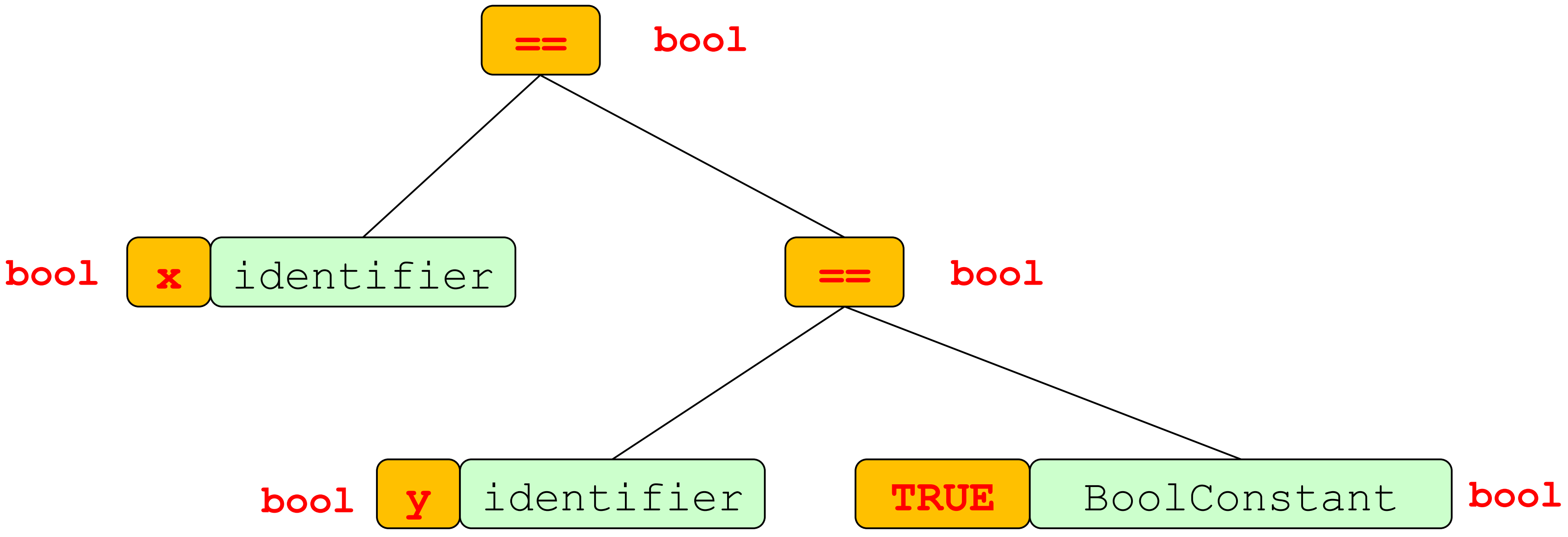
类型检查：核心是逻辑推断

根据已知的类型，对表达式的类型进行推断



类型检查：核心是逻辑推断

根据已知的类型，对表达式的类型进行推断



类型检查：核心是逻辑推断

true/false	bool
<int constant>	int
<string constant>	string
<double constant>	double

类型检查：核心是逻辑推断

true/false	bool
<int constant>	int
<string constant>	string
<double constant>	double
int + int	int
double + double	double
T + T	T （T是基本类型）
T == T	Bool （T是基本类型）
T a[i]	T （T是基本类型）

具有偏序关系的类型

如果类型A可转化为B，则 A ≤ B

在C语言中，可根据 字节长度 确定转化关系

```
int <= long long
int <= float
float <= double

bool <= int
int <= bool
```

类型检查：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

语言结构中隐含的类型

在条件、循环中的条件判断：　　Bool类型

```
if (expr) {
    // do something
}
```

函数的形参类型及返回值类型

```
int fib(int x) {
    if(x<=2) return 1;
    return fib(x-1)+fib(x-2);
}
```

语言结构中隐含的类型

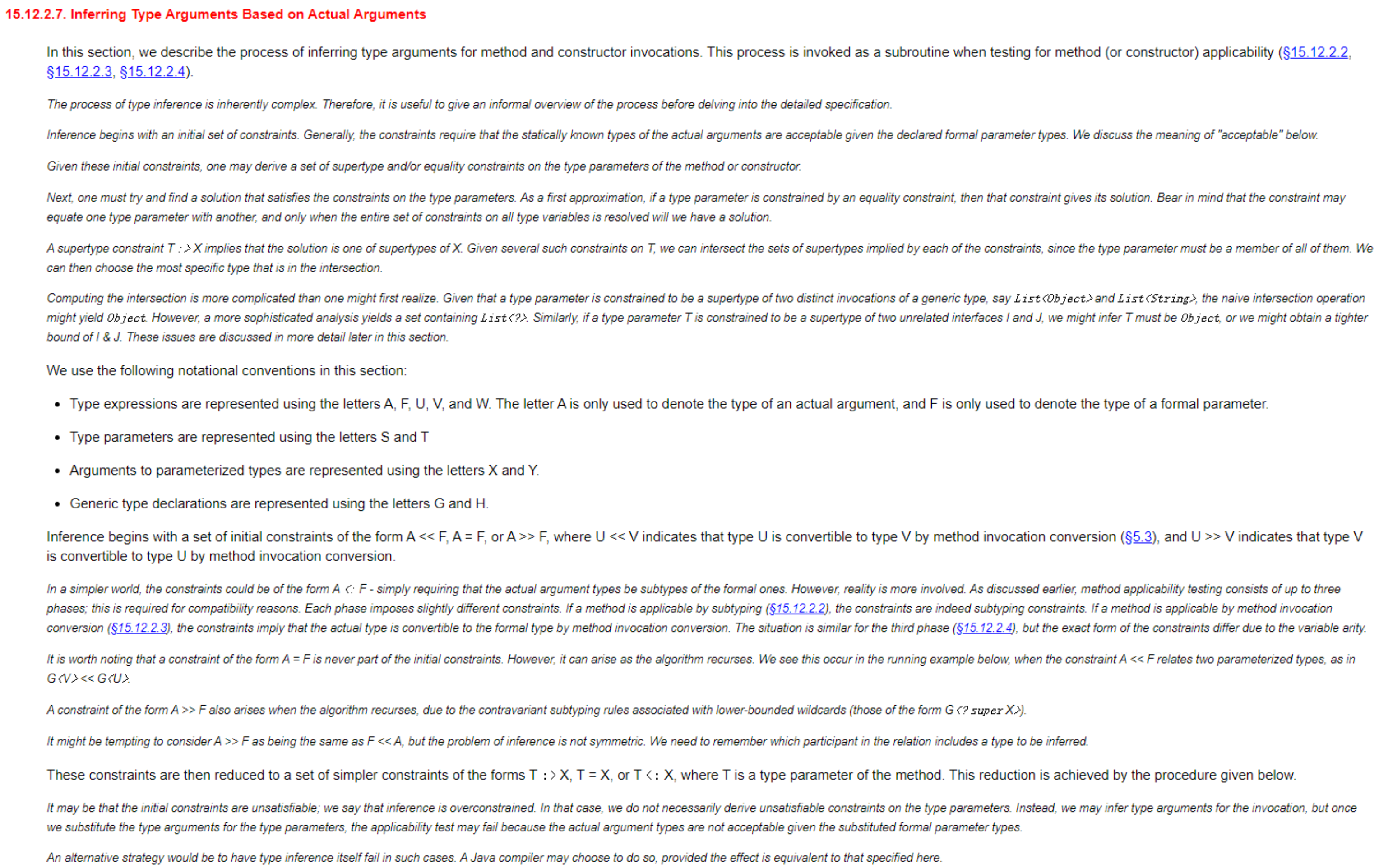
例子：

```
int x, y, z;
if(((x==y) >5 && x+y<z) || x==z) {
     // do something
}
```

类型错误：

```
(x==y) >5
(x==y) >5 && x+y<z
(((x==y) >5 && x+y<z) || x==z
```

Java语言规范中的类型转换



15.12.2.7. Inferring Type Arguments Based on Actual Arguments

In this section, we describe the process of inferring type arguments for method and constructor invocations. This process is invoked as a subroutine when testing for method (or constructor) applicability (§15.12.2.2, §15.12.2.3, §15.12.2.4).

The process of type inference is inherently complex. Therefore, it is useful to give an informal overview of the process before delving into the detailed specification.

Inference begins with an initial set of constraints. Generally, the constraints require that the statically known types of the actual arguments are acceptable given the declared formal parameter types. We discuss the meaning of "acceptable" below.

Given these initial constraints, one may derive a set of supertype and/or equality constraints on the type parameters of the method or constructor.

Next, one must try and find a solution that satisfies the constraints on the type parameters. As a first approximation, if a type parameter is constrained by an equality constraint, then that constraint gives its solution. Bear in mind that the constraint may equate one type parameter with another, and only when the entire set of constraints on all type variables is resolved will we have a solution.

A supertype constraint T :> X implies that the solution is one of supertypes of X. Given several such constraints on T, we can intersect the sets of supertypes implied by each of the constraints, since the type parameter must be a member of all of them. We can then choose the most specific type that is in the intersection.

Computing the intersection is more complicated than one might first realize. Given that a type parameter is constrained to be a supertype of two distinct invocations of a generic type, say `List<Object>` and `List<String>`, the naive intersection operation might yield `Object`. However, a more sophisticated analysis yields a set containing `List<?>`. Similarly, if a type parameter T is constrained to be a supertype of two unrelated interfaces I and J, we might infer T must be `Object`, or we might obtain a tighter bound of I & J. These issues are discussed in more detail later in this section.

We use the following notational conventions in this section:

- Type expressions are represented using the letters A, F, U, V, and W. The letter A is only used to denote the type of an actual argument, and F is only used to denote the type of a formal parameter.
- Type parameters are represented using the letters S and T
- Arguments to parameterized types are represented using the letters X and Y.
- Generic type declarations are represented using the letters G and H.

Inference begins with a set of initial constraints of the form A << F, A = F, or A >> F, where U << V indicates that type U is convertible to type V by method invocation conversion (§5.3), and U >> V indicates that type V is convertible to type U by method invocation conversion.

In a simpler world, the constraints could be of the form A <: F - simply requiring that the actual argument types be subtypes of the formal ones. However, reality is more involved. As discussed earlier, method applicability testing consists of up to three phases; this is required for compatibility reasons. Each phase imposes slightly different constraints. If a method is applicable by subtyping (§15.12.2.2), the constraints are indeed subtyping constraints. If a method is applicable by method invocation conversion (§15.12.2.3), the constraints imply that the actual type is convertible to the formal type by method invocation conversion. The situation is similar for the third phase (§15.12.2.4), but the exact form of the constraints differ due to the variable arity.

It is worth noting that a constraint of the form A = F is never part of the initial constraints. However, it can arise as the algorithm recurses. We see this occur in the running example below, when the constraint A << F relates two parameterized types, as in G<V> << G<U>.

A constraint of the form A >> F also arises when the algorithm recurses, due to the contravariant subtyping rules associated with lower-bounded wildcards (those of the form G<? `super` X>).

It might be tempting to consider A >> F as being the same as F << A, but the problem of inference is not symmetric. We need to remember which participant in the relation includes a type to be inferred.

These constraints are then reduced to a set of simpler constraints of the forms T :> X, T = X, or T <: X, where T is a type parameter of the method. This reduction is achieved by the procedure given below.

It may be that the initial constraints are unsatisfiable; we say that inference is overconstrained. In that case, we do not necessarily derive unsatisfiable constraints on the type parameters. Instead, we may infer type arguments for the invocation, but once we substitute the type arguments for the type parameters, the applicability test may fail because the actual argument types are not acceptable given the substituted formal parameter types.

An alternative strategy would be to have type inference itself fail in such cases. A Java compiler may choose to do so, provided the effect is equivalent to that specified here.

Source: https://docs.oracle.com/javase/specs/jls/se7/html/jls-15.html#jls-15.12.2.7

表达式：类型检查+数组变量引用

数组元素的地址计算

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

其中

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

可变部分　　　　不变部分（可提前计算）

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

还能如何进一步加速计算?

$$ADDR = \overset{\text{可变部分}}{\boxed{LOC + \sum_{i=1}^{n} V(i)\times P(i)\times E}} - \overset{\text{不变部分（可提前计算）}}{\boxed{\sum_{i=1}^{n} L(i)\times P(i)\times E}}$$

计算第1维时： $V(1)*P(1)*E=V(1)*(U(2)-L(2)+1)*\underline{(U(3)-L(3)+1)\ldots}$

计算第2维时： $V(2)*P(2)*E=V(2)*\underline{(U(3)-L(3)+1)*\ldots}$

还能如何进一步加速计算?

$$\mathrm{ADDR} = \overset{\text{可变部分}}{\boxed{\mathrm{LOC} + \sum_{i=1}^{n} \mathrm{V(i)}\times \mathrm{P(i)}\times \mathrm{E}}} - \overset{\text{不变部分（可提前计算）}}{\boxed{\sum_{i=1}^{n} \mathrm{L(i)}\times \mathrm{P(i)}\times \mathrm{E}}}$$

计算第1维时： $V(1)*P(1)*E=V(1)*(U(2)-L(2)+1)*\underline{(U(3)-L(3)+1)\ldots}$

计算第2维时： $V(2)*P(2)*E=V(2)*\underline{(U(3)-L(3)+1)*\ldots}$

$$\mathbf{VP(i)}= \sum_{k=1}^{n} V(k)\times P(k)$$

可递归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

还能如何进一步加速计算?

$$ADDR = \overset{\text{可变部分}}{\boxed{LOC + \sum_{i=1}^{n} V(i)\times P(i)\times E}} - \overset{\text{不变部分（可提前计算）}}{\boxed{\sum_{i=1}^{n} L(i)\times P(i)\times E}}$$

$$ADR = LOC + (\sum_{i=1}^{n} VP(i) \ + RC) * E$$

$$VP(i) = \sum_{k=1}^{n} V(k)\times P(k) = VP(i-1) + V(i) * P(i)$$

相关指令 (p.185)

TEMPLATE D	将数组模板地址addr加载到栈顶
OFFSET K	检查操作数栈的栈顶（第k个下标）是否越界 计算地址公式中的可变部分 VP(i)
ARREF	栈顶的VP(n)与次栈顶的LoC，计算数组元素地址并压入栈顶
DEREF	用数组元素的值代替操作数栈栈顶的数组元素地址

问题：分析扫描代码 B (2,1)，如何将 B(2,1)的值放入数据栈栈顶?

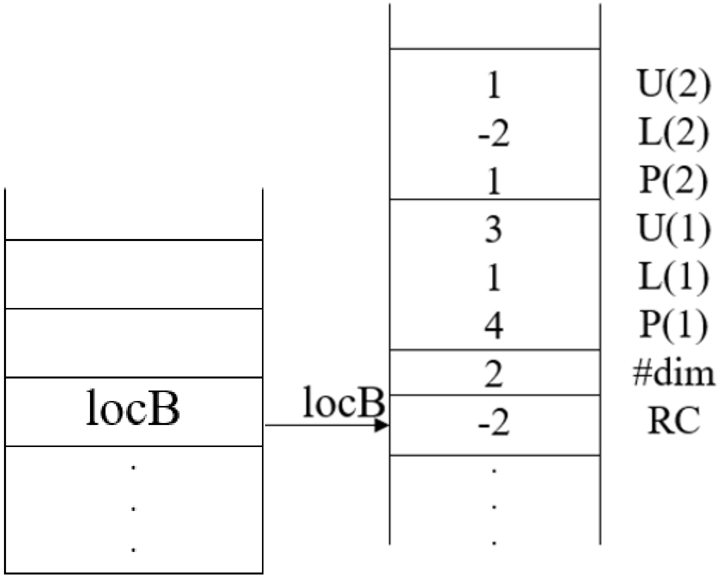
$$VP(i)=\sum_{k=1}^{n} V(k)\times P(k) = VP(i-1) + V(i) *P(i)$$

B (2,1)

$$ADR = LOC + (\sum_{i=1}^{n} VP(i) + RC) * E$$

TMP locB 将数组模板地址addr加载到栈顶

(a-)



栈底

B的模板

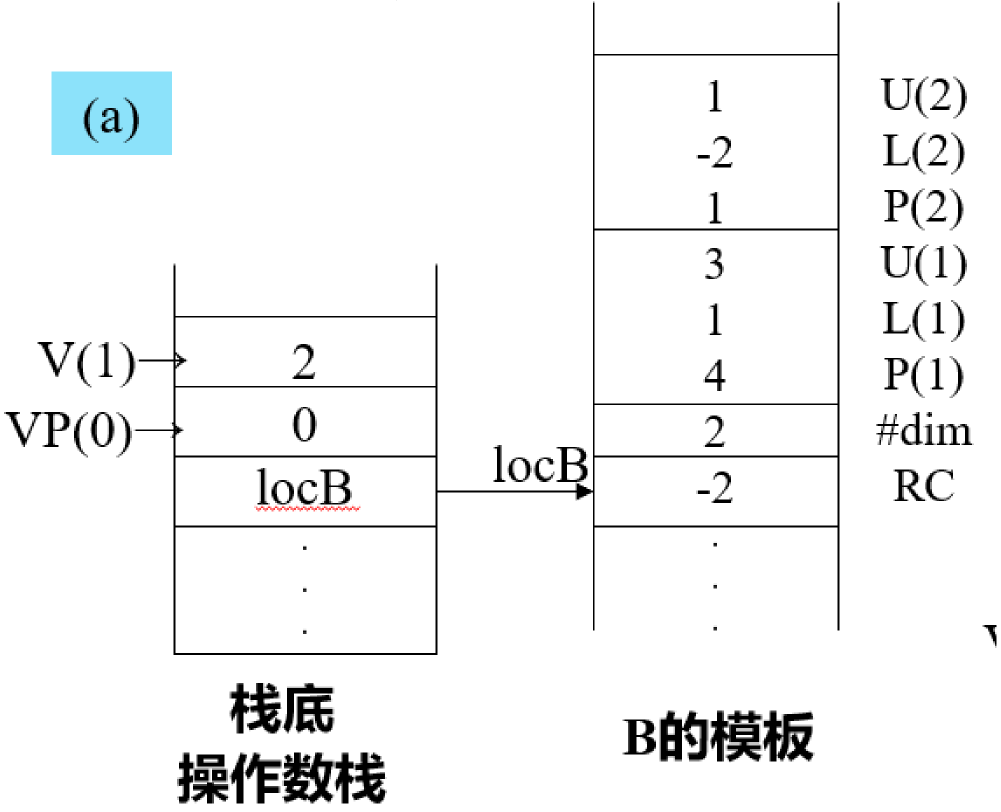
VP(i)= $\sum_{k=1}^{n} V(k)\times P(k)$ = VP(i-1) + V(i) *P(i)

B (2,1)

$$ADR = LOC + (\sum_{i=1}^{n} VP(i) + RC) * E$$

TMP locB　　将数组模板地址addr加载到栈顶

LDI 2　　将第k=1维的下标2压入栈顶



$$VP(i)= \sum_{k=1}^{n} V(k)\times P(k) = VP(i-1) + V(i) *P(i)$$

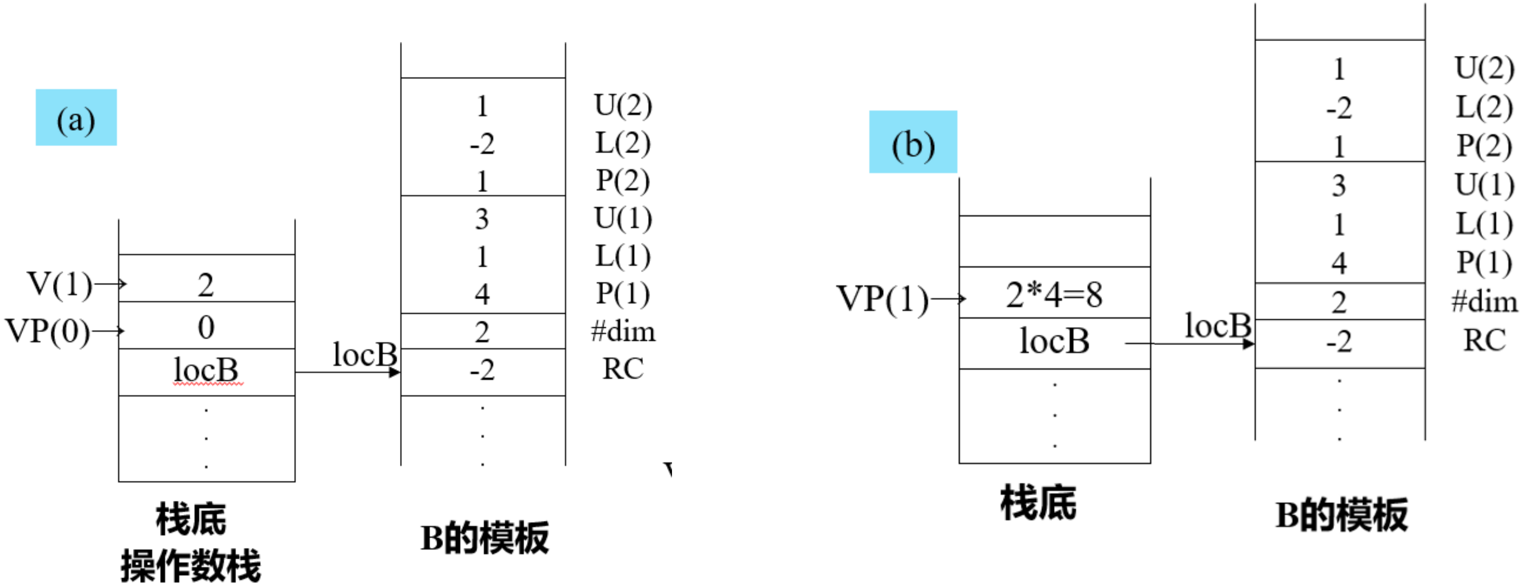
B (2,1)

$$ADR = LOC + (\sum_{i=1}^{n} VP(i) \ + RC) * E$$

TMP locB 将数组模板地址addr加载到栈顶

LDI 2 将第k=1维的下标2压入栈顶

OFS 1 弹栈，得到第k=1维下标2，与L(i),U(i)做越界检查，计算VP(i)，替换栈顶的VP(0)



$$VP(i)= \sum_{k=1}^{n} V(k)\times P(k) = VP(i-1) + V(i) *P(i)$$

B (2,1)

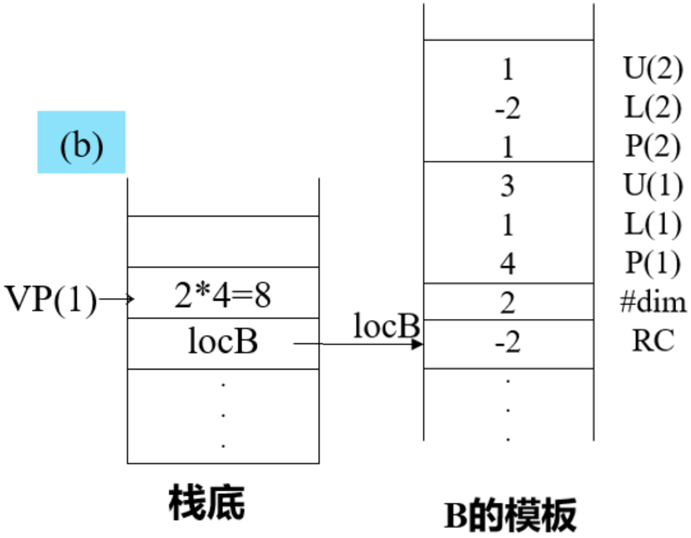
$$ADR = LOC + (\sum_{i=1}^{n} VP(i) \ + RC) * E$$

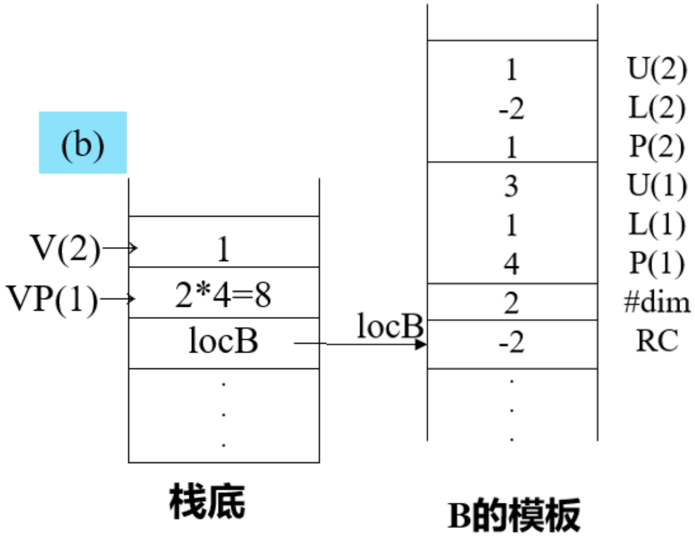
TMP locB 将数组模板地址addr加载到栈顶

LDI 2 将第k=1维的下标2压入栈顶

OFS 1 弹栈，得到第k=1维下标2，与L(i),U(i)做越界检查，计算VP(i)，替换栈顶的VP(0)

LDI 1 将第k=2维的下标1压入栈顶





$VP(i)=\sum_{k=1}^{n} V(k)\times P(k)$ = VP(i-1) + V(i) *P(i)

B (2,1)

$$ADR = LOC + (\sum_{i=1}^{n} VP(i) \ + RC) * E$$

TMP locB　将数组模板地址addr加载到栈顶

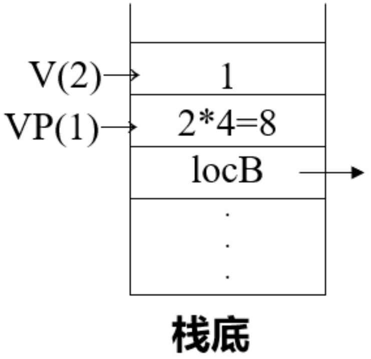
LDI 2　将第k=1维的下标2压入栈顶

OFS 1　弹栈，得到第k=1维下标2，与L(i),U(i)做越界检查，计算VP(i)，替换栈顶的VP(0)

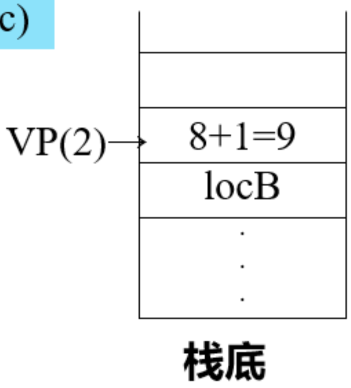
LDI 1　将第k=2维的下标1压入栈顶

OFS 2　弹栈，得到第k=2维下标1，与L(2),U(2)做越界检查，计算VP(2)，替换栈顶的VP(1)

(b)



(c)



$$VP(i)=\sum_{k=1}^{n} V(k)\times P(k) = VP(i-1) + V(i) *P(i)$$

B (2,1)

$$ADR = LOC + (\sum_{i=1}^{n} VP(i) \ + RC) * E$$

TMP locB　将数组模板地址addr加载到栈顶

LDI 2　将第k=1维的下标2压入栈顶

OFS 1　弹栈，得到第k=1维下标2，与L(i),U(i)做越界检查，计算VP(i)，替换栈顶的VP(0)

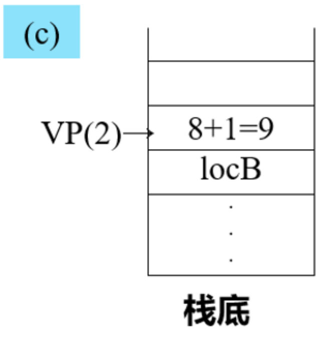
LDI 1　将第k=2维的下标1压入栈顶

OFS 2　弹栈，得到第k=2维下标1，与L(2),U(2)做越界检查，计算VP(2)，替换栈顶的VP(1)

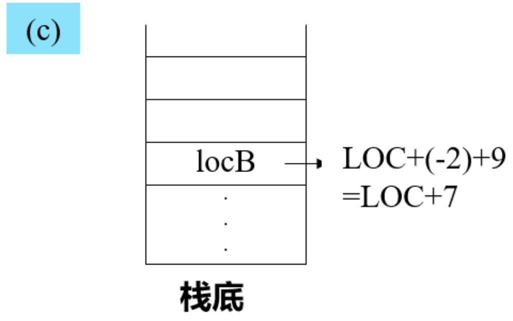
ARREF　弹栈得到VP(2)，与locB+RC相加得到ADDR替换栈顶

DEREF　弹栈得到B(2,1)地址，将值取出压入栈顶

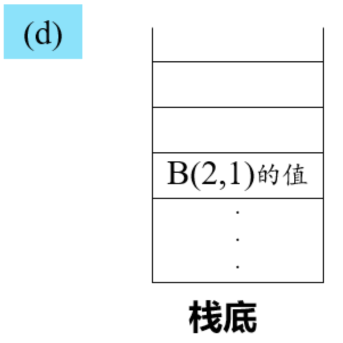
(c)



(c)

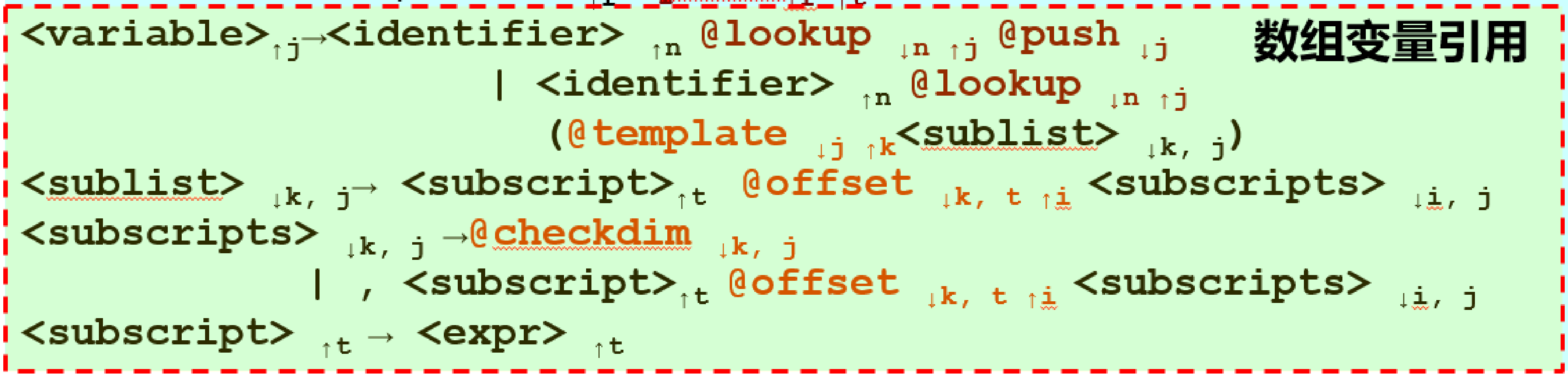


(d)



```
<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
```

```
<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     数组变量引用
                   | <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
```



```
<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
```

数组变量引用

★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。

@template

根据符号表表项 J, 输出TMP指令,在运行时加载数组模板

维数计数器 k 初始化 (k=0), 用于在后面判断维数是否越界

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

数组变量引用

<variable>$_{\uparrow j}$→<identifier> $_{\uparrow n}$ @lookup $_{\downarrow n}$ $_{\uparrow j}$ @push $_{\downarrow j}$

| <identifier> $_{\uparrow n}$ @lookup $_{\downarrow n}$ $_{\uparrow j}$

(@template $_{\downarrow j}$ $_{\uparrow k}$<sublist> $_{\downarrow k,\ j}$)

<sublist> $_{\downarrow k,\ j}$→ <subscript>$_{\uparrow t}$ @offset $_{\downarrow k,\ t}$ $_{\uparrow i}$ <subscripts> $_{\downarrow i,\ j}$

<subscripts> $_{\downarrow k,\ j}$ →@checkdim $_{\downarrow k,\ j}$

| , <subscript>$_{\uparrow t}$ @offset $_{\downarrow k,\ t}$ $_{\uparrow i}$ <subscripts> $_{\downarrow i,\ j}$

<subscript> $_{\uparrow t}$ → <expr> $_{\uparrow t}$

@offset

根据当前维度 k, 当前表项 j

维数计数器 k++

完成越界检查，并计算数组地址中的可变部分 OFS k

```
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);
end;
```

数组变量引用

```
<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
```

@checkdim

根据当前维度 k, 当前表项 j

检查维数 k与符号表是否一致

产生ARREF指令

产生DEREF指令

```
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( ‘ARREF’);
               emit( ‘DEREF’);
              end;
end;
```

逻辑运算与逻辑表达式

$\sim (x = y \,\&\, y \neq z \mid z < x)$

```
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
```

赋值语句

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
```

$\langle assignstat \rangle \rightarrow @setL_{\uparrow L} \langle variable \rangle_{\downarrow L \uparrow t} :=$
$@resetL_{\uparrow L} \langle expr \rangle_{\uparrow s} @storin_{\downarrow t,s}$;

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;
```

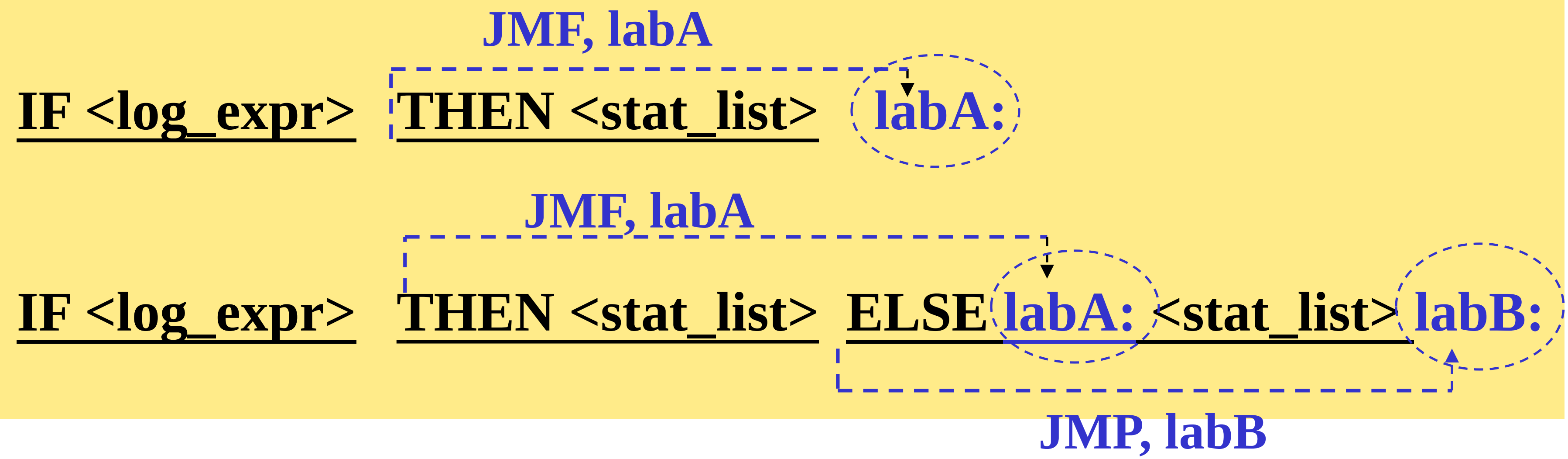
指示取变量之值

```
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;
```

控制流

10.6 控制语句的处理

if-then-else 语句



JMF, labA

IF <log_expr> THEN <stat_list> labA:

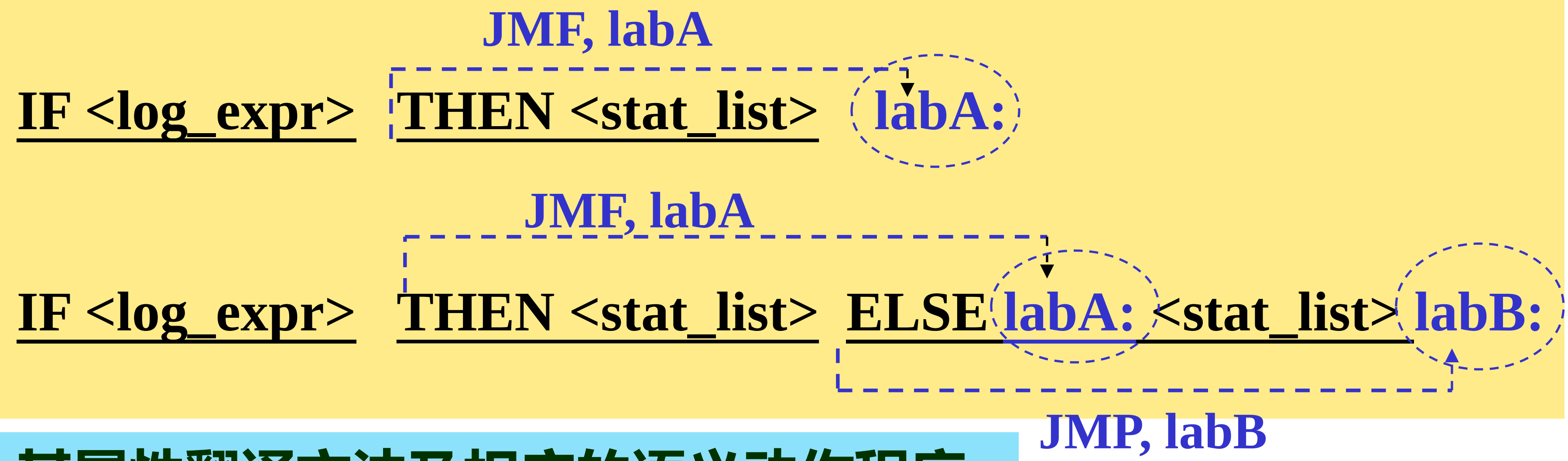
JMF, labA

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

JMP, labB

10.6 控制语句的处理

if-then-else 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

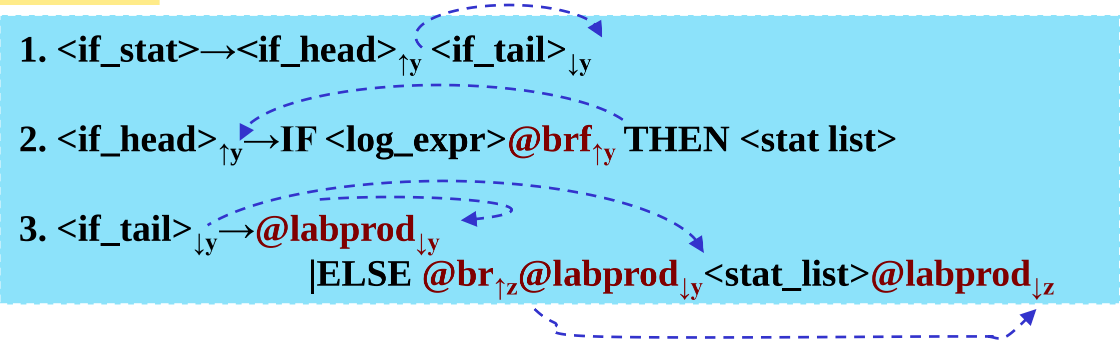
```
procedure  brf;
     string labx;
     labx := genlab;
       /*产生一标号赋给labx*/
     emitl(‘JMF’, labx);
     return (labx);
end;
```

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;
     emitl( ‘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$_{\downarrow z}$

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$_{\uparrow s}$ 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$_{\downarrow r}$ @labemit$_{\downarrow f}$

```
procedure labgen
        string r;
        r := genlab;
        setlab(r);
        return ( r );
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 loadid( a )
        address a;
        emitl(‘LOD’, a);
end;
```

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

while 循环语句

while 语句例子:

```
while (i<n) {
    j--;
    i++;
}
```

原始文法

ATG文法

过程调用与返回

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

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

传值：

```
I:     2
x:     6
A[1]:  1
```

```
A[1] = 1
A[2] = 2
```

传地址：

```
I:     2
x:     A[1]
A[1]:  6
```

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

传名：

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

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

10.7.2 过程调用处理

process_symb(symb, cursor,replacestr);

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

传值调用

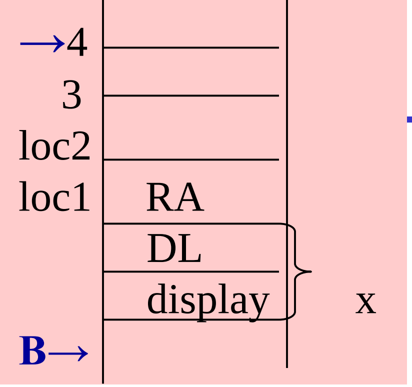
调用该过程生成的目标代码为：

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

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。

```
procedure  process_symb
        (string:symbal, int: cur, string:
```



→4
3
loc2
loc1
RA
DL
display
x
B→

x: display+DL

过程体头部
应生成指令，
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:

<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;
```

过程调用的ATG文法:

<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> → @chklength

@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

```
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;
```

形参个数计数，j初值为0

过程说明（定义）的ATG文法如下：

```
<proc defn>→<proc defn head>  @initcnt↑j
                         <parameters>↓j↑k  @emitstores↓k
<proc defn head>→procedure↑t <id>↑n @tblinsert↓t, n
<parameters>↓j↑k→@echo↓j↑k | (<parm list>↓j↑k)
<parm list>↓j↑l→<type>↑t : <id>↑n @tblinsert↓t, n

@upcnt↓j↑k <parms>↓k↑l
<parms>↓j↑l→@e      ↓j↑l | , <type>↑t : <id>↑n @tblinsert↓t, n

@upcnt↓j    <parms>↓k↑l
```

形参名填表

K := j ++

K := j

l := j

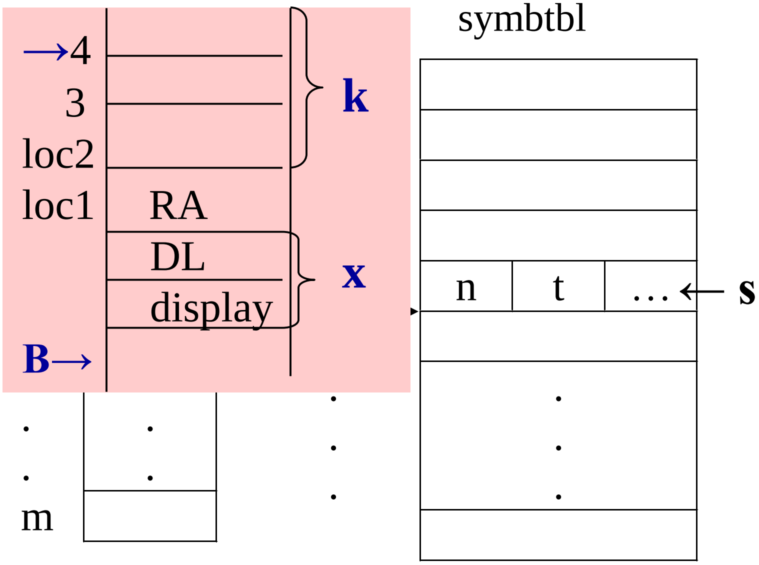
@tblinsert 是把过程名和它的形参名填入符号表中：

```
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;
  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;
```



```
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 */
```

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。

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1：写出类C语言的 struct 语句的属性翻译文法及其处理动作程序。

教材，Pascal-S的RECORD

作业2：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。

谢谢！