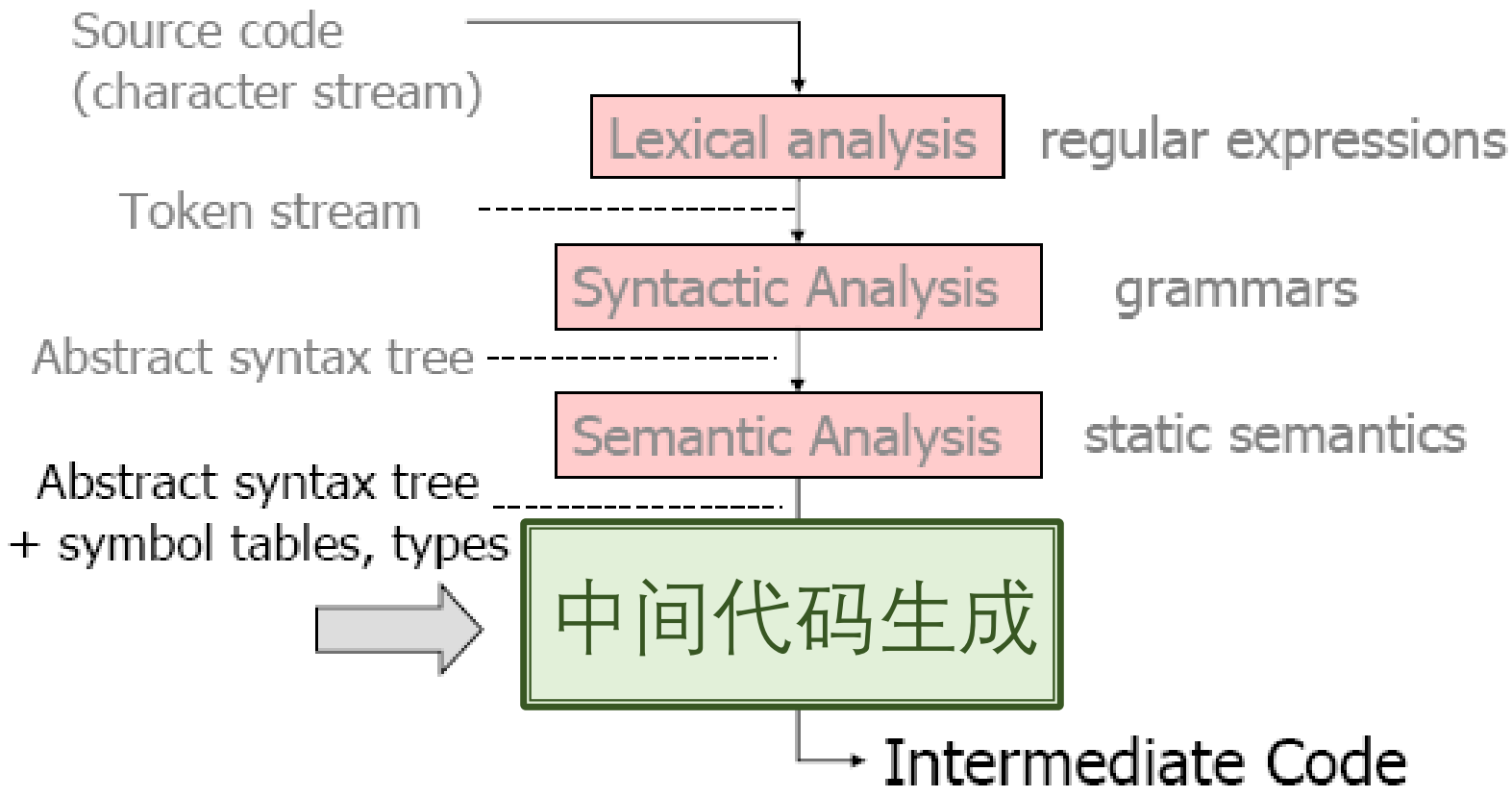
编译原理

--语义分析和中间代码生成

刘爽

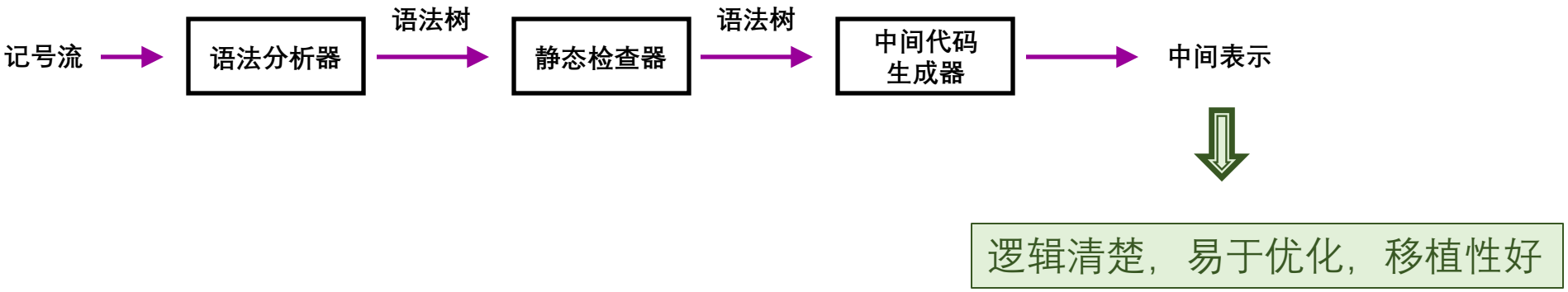
中国人民大学信息学院

所处位置


Source code
(character stream)
Lexical analysis
regular expressions
Token stream
Syntactic Analysis
grammars
Abstract syntax tree
Semantic Analysis
static semantics
Abstract syntax tree
+ symbol tables, types
中间代码生成
Intermediate Code

语义分析的位置和作用

- 紧跟在词法分析和语法分析之后，编译程序要做的工作就是进行静态语义检查和翻译。
- 编译器必须要检查源程序是否符合源语言规定的语法和语义要求。这种检查称为静态检查，检查并报告程序中某些类型的错误。



静态语义检查

- 静态语义检查通常包括:
  - 类型检查：如果操作符作用于不相容的操作数，使用不同数目的参数或者错误类型的参数对函数进行调用等，编译器应该报错
  - 控制流检查：引起控制流从某个结构中跳转出来的语句必须能够决定控制流转向的合法目标地址，例如必须保证break或者continue语句包含在while (或 for)语句中
  - 唯一性检查：有时，有的对象只能被定义一次。比如，同一case语句的标号不能相同，枚举类型的元素不能重复。
  - 与名字相关的检查：有时候要求同一名字在特定位置出现两次或多次（如，标识结构的开始和结尾）
  - 名字的作用域分析等

Outline

- 中间语言
- 一些语法成分的翻译
  - 声明语句
  - 表达式
  - 赋值语句
  - 控制语句
  - 过程调用

源语言的中间表示方法

- 抽象语法树
- 后缀式
- 三地址代码（包括三元式、四元式、间接三元式）
- 无循环有向图DAG (Directed Acyclic Graph)图表示

后缀式

这种表示法不用使用括号

- 后缀式表示又称逆波兰表示法。
- 这种表示法是：把运算量（操作数）写在前面，把算符写在后面（后缀）。
- 一个表达式的后缀形式可以如下定义：
  - 如果E是一个变量或常量，则E的后缀式是E自身
  - 如果E是E_1op E_2形式的表达式，这里op是任何二元操作符，则E的后缀式为$E_1'E_2'$op。这里E_1'和E_2'分别是E_1和E_2的后缀式。
  - 如果E是(E_1)形式的表达式，则E_1的后缀式就是E的后缀式
- 只要知道每个算符的目数，对于后缀式，无论从哪一端进行扫描，都能对它正确的进行唯一分解

后缀式

- 表达式翻译为后缀式的语义规则描述：

产生式	语义规则
$E \rightarrow E_1\ op\ E_2$	$E.code := E_1.code \parallel E_2.code \parallel op$
$E \rightarrow (E_1)$	$E.code := E_1.code$
$E \rightarrow id$	$E.code := id$

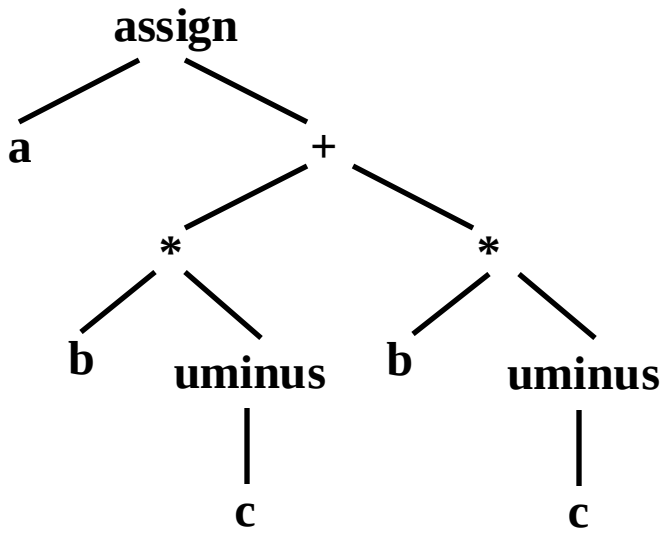
后缀式ab + c*的计值过程
1. 把a压入栈
2. 把b压入栈
3. 弹出栈顶两项，相加之和压入栈
4. 把c压入栈
5. 弹出栈顶两项，相乘之积压入栈

- 其中E.code表示E的后缀式，op表示任何二元操作符，“||”表示后缀形式的连接

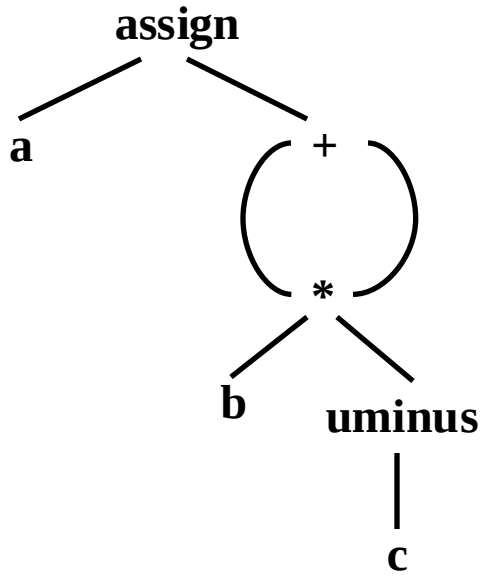
推广到表达式外的范围：例如 a:=b*c+b*d 后缀形式：abc*bd*+:=

图表示法

- 图表示法主要包括DAG(Directed Acyclic Graph)与抽象语法树
- 语法树描述了源程序的自然层次结构。DAG以更紧凑的形式给出了相同的信息。两者不同的是:
  - 在一个DAG中代表公共子表达式的结点具有多个父结点
  - 在一颗抽象语法树中公共子表达式被表示为重复的子树



a:= b*-c + b*-c



a bc uminus * bc uminus * + assign

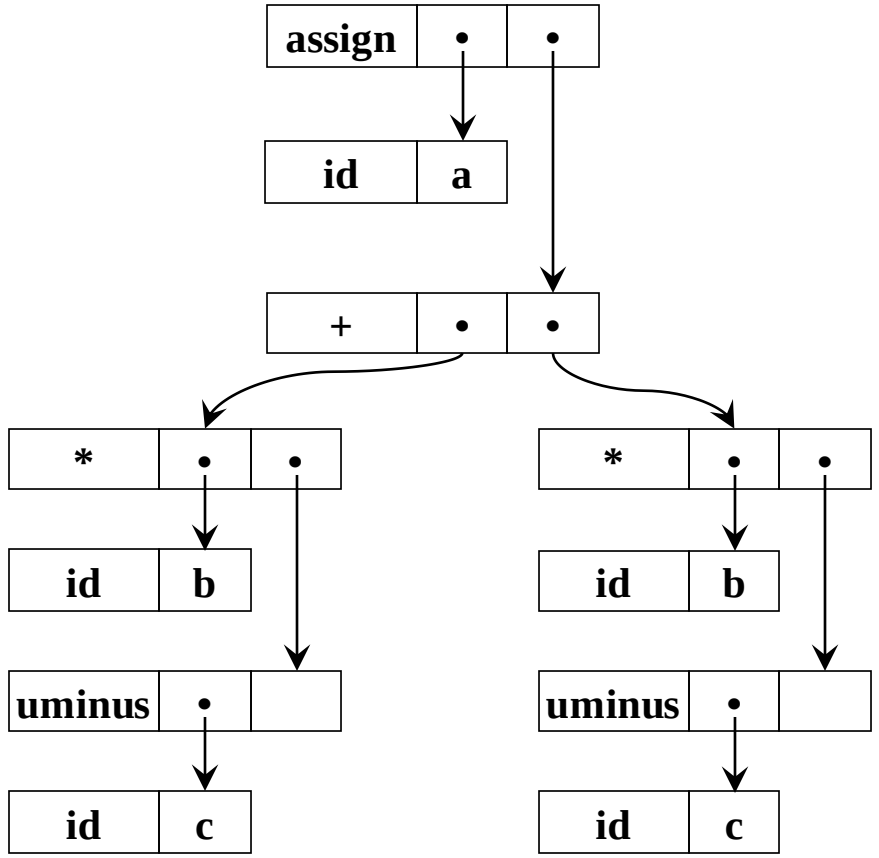
抽象语法树

- 产生赋值语句抽象语法树的属性文法

产生式	语义规则
S→id := E	S.nptr := mknode('assign', mkleaf(id, id.place), E.nptr)
$E \rightarrow E_1 + E_2$	E.nptr := mknode('+' , E_1.nptr, E_2.nptr)
$E \rightarrow E_1 * E_2$	E.nptr := mknode('*' , E_1.nptr, E_2.nptr)
$E \rightarrow - E_1$	E.nptr := mknode('uminus', E_1.nptr)
$E \rightarrow (E_1)$	E.nptr := E_1.nptr
E→id	E.nptr := mkleaf(id , id.place)

每个结点用一个记录来表示，该记录包括一个运算符号域和若干个指向子结点的指针域

抽象语法树的表示形式



0	id	b	
1	id	c	
2	uminus	1	
3	*	0	2
4	id	b	
5	id	c	
6	uminus	5	
7	*	4	6
8	+	3	7
9	id	a	
10	assign	9	8
11	…		

如果函数mknode(op, child)和mknode(op, left, right)尽可能返回一个指向一个存在的结点的指针以代替建立新的结点，那么就会生成DAG图

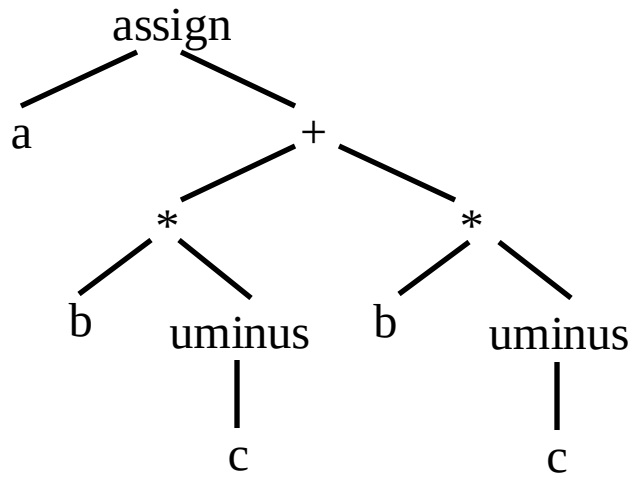
三地址代码

- 三地址代码是下列一般形式的语句序列
  - x := y op z
  - 其中，x、y和z是名字，常量或编译器生成的临时变量
  - op代表任何操作符（定点运算符、浮点运算符、逻辑运算符等）
- 像x+y*z这样的表达式要翻译为如下；
  - T_1 := y * z
  - T_2 := x + T_1
  - 其中T_1，T_2为编译时产生的临时变量。

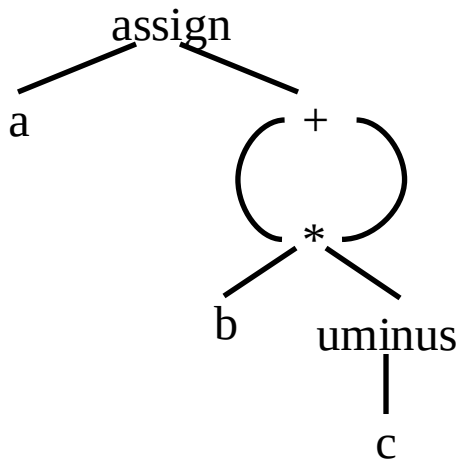
三地址代码

- 这种对复杂算术表达式和嵌套控制流语句的拆解使得三地址码适用于目标代码生成及优化。
- 由程序计算出来的中间值的名字的使用使得三地址码容易被重排列——而不像后缀表达式那样
- 三地址码可以看成是语法树或DAG的线性表示。
- 三地址码的得名原因是每条语句通常包含三个地址，两个是操作数地址，一个是结果地址。
- 在实际的实现中，程序员定义的名字被一个指向该名字的符号表表项的指针所代替。

三地址码例子



t_1 := -c
t_2 := b * t_1
t_3 := -c
t_4 := b * t_3
t_5 := t_2 + t_4
a := t_5



t_1 := -c
t_2 := b * t_1
t_3 := t_2 + t_2
a := t_3

三地址语句的类型

- 三地址语句类似于汇编语言代码。语句可以带有符号标号，而且存在各种控制流语句。
- 符号标号代表存放中间代码的数组中三地址代码语句的下标。三地址语句的种类：
  - 形如x:= y op z的赋值语句，其中op为二元算术算符或逻辑算符
  - 形如x:= op y的赋值语句，其中op为一元算符
  - 形如x:= y的复制语句，将y的值赋给x
  - 形如goto L的无条件跳转语句，即下一条将被执行的语句是带有标号L的三地址语句
  - 形如if x relop y goto L或 if a goto L的条件跳转语句
    - 第一种形式使用关系运算符号relop（<，>等）
    - 第二种a为布尔变量或常量
  - 用于过程调用的语句param x和call p, n，以及返回语句return y。源程序中的过程调用p(x1,x2,…,xn)：
  - 形如x := y[i]及x[i] := y的索引赋值
  - 形如x := &y, x := *y和*x := y的地址和指针赋值

```
param x1
……
param xn
call p, n
n表示实参个数; return y
中y为过程返回的一个值
```

三地址语句的类型

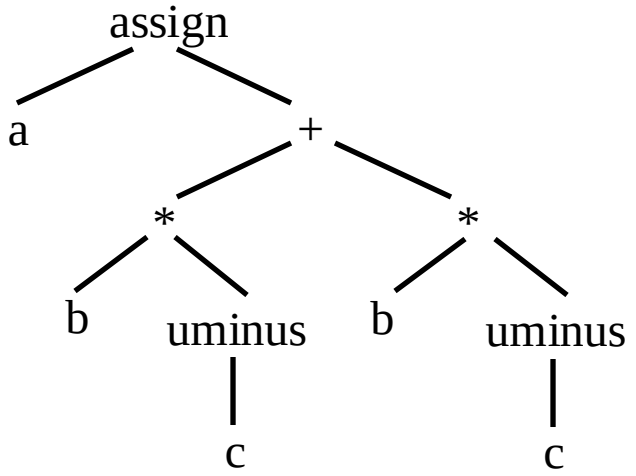
- 设计中间代码形式时，运算符的选择是非常重要的。
- 算符种类应足以用来实现源语言中的运算。
- 一个小型算符集合较易于在新的目标机器上实现。
- 局限的指令集合会使某些源语言运算表示成中间形式时代码加长，需要在目标代码生成时做较多的优化工作。

生成三地址码的S-属性文法

- S具有综合属性S.code，代表赋值语句S的三地址码
- 非终结符E有如下性质：
  - E.place表示存放E值的名字
  - E.code表示对E求值的三地址语句序列
- 函数newtemp的功能是每次调用它时，将返回一个不同临时变量的名字。如T_1,T_2,….
- 用gen(x ‘:=’ y ‘+’ z)表示生成三地址语句x:=y+z。
- 在实际实现中，三地址码可能被送到输出文件中，而不是生成code属性。

生成三地址码的S-属性文法

产生式	语义规则
S→id := E	S.code := E.code || gen(id.place ':= ' E.place)
$E \rightarrow E_1 + E_2$	E.place := newtemp;
	E.code := E_1.code || E_2.code || gen(E.place ':=' E_1.place '+' E_2.place)
$E \rightarrow E_1 * E_2$	E.place := newtemp;
	E.code := E_1.code || E_2.code || gen(E.place ':=' E_1.place '*' E_2.place)
$E \rightarrow - E_1$	E.place := newtemp;
	E.code := E_1.code || gen(E.place ':=' 'uminus' E_1.place)
$E \rightarrow (E_1)$	E.place := E_1.place E.code := E_1.code
E→id	E.place := id.place E.code := ''



t_1 := -c
t_2 := b * t_1
t_3 := -c
t_4 := b * t_3
t_5 := t_2 + t_4
a := t_5

三地址语句的实现

- 三地址语句是中间代码的一种抽象形式。
- 这些语句可以以带有操作符和操作数域的记录来实现。
- 四元式、三元式及间接三元式是三种类型的三地址语句表示。

四元式

- 一个四元式是带有四个域的记录结构，这四个域分别称为op, arg1, arg2及result（均是指向有关名字的符号表的入口指针）
  - 域op包含一个代表运算符的内部码
  - 三地址语句x:=y op z通过将y放入arg1，z放入arg2，并且将x放入result，:=为算符
  - 像x:=y或x:=-y这样的一元操作符语句不使用arg2
  - 像param这样的运算符仅使用arg1域
  - 条件和无条件语句将目标标号存入result域
  - 临时变量也要填入符号表中

	op	arg1	arg2	result
(0)	uminus	c		T_1
(1)	*	b	T_1	T_2
(2)	uminus	c		T_3
(3)	*	b	T_3	T_4
(4)	+	T_2	T_4	T_5
(5)	assign	T_5		a

三元式

- 为了避免把临时变量填入符号表，我们可以通过计算这个临时变量的语句的位置来引用这个临时变量。
- 这样三地址代码的记录只需要三个域op, arg1和arg2
- 对于一目运算符op, arg1和arg2只需用其一。我们可以随意选用一个

a := b * -c + b * -c

	op	arg1	arg2	result
(0)	uminus	c		T_1
(1)	*	b	T_1	T_2
(2)	uminus	c		T_3
(3)	*	b	T_3	T_4
(4)	+	T_2	T_4	T_5
(5)	assign	T_5		a

	op	arg1	arg2
(0)	uminus	c	
(1)	*	b	0
(2)	uminus	c	
(3)	*	b	2
(4)	+	1	3
(5)	assign	a	4

间接三元式

- 为了便于代码优化处理，有时不直接使用三元式表，而是另设一张指示器（称为间接码表），它将运算的先后顺序列出有关三元式在三元表中的位置。
- 换句话说，我们用一张间接码表辅以三元式表的办法来表示中间代码。这种表示方法称为间接三元式。

三元式序列所需存储空间少（不需临时变量），但不便于优化处理
解决方法：建立两个表：
三元式表：存放各三元式本身
间接码表：按运算先后顺序列出三元式在三元式表中的位置

间接三元式举例

- X:=(A+B)*C
  Y:=D^(A+B)

	op	arg1	arg2
(1)	+	A	B
(2)	*	(1)	C
(3)	:=	X	(2)
(4)	^	D	(1)
(5)	:=	Y	(4)

当在代码优化过程中需要调整运算顺序时，只需重新安排间接码表，无需改动三元式表

对于间接三元式表示，语义规则中应增添产生间接码表的动作，并且在向三元式表填进一个三元式之前，必须先查看一下此式是否已在其中，就无须填入。

间接代码表

(1) (2) (3) (1) (4) (5)

表示方法比较：间址的使用

- 三元式与四元式的差异
  - 使用四元式表示，定义或使用临时变量的三地址语句可通过符号表直接访问该临时变量的地址
  - 使用四元式的一个更重要的好处体现在优化编译器中。在三元式中，如果要移动一条临时值的语句需要改变arg1和arg2中对该语句的引用。间接三元式没有上述问题
- 间接三元式看上去和四元式非常相似，他们都需要大约相同的存储空间，并且对代码重新排序的效率相同

三元式
优点：节省空间
缺点：不利于优化
四元式与间接三元式
缺点：占用存储空间相对较大
优点：代码调整时改动少

练习

- 求下列各式的逆波兰表示

  -a-(b*c/(c-d) + (-b)*a)

  -A+B*C^(D/E)/F

- 写出A+B*(C-D)-E/F^G的三元式表示和四元式表示

练习

- -a-(b*c/(c-d) + (-b)*a)的逆波兰表示：

  a *uminus* bc*cd-/b *uminus* a*+-

- -A+B*C^(D/E)/F的逆波兰表示:

  A-BCDE/^*F/+

- A+B*(C-D)-E/F^G

三元式表示

(1) (－, C, D)
(2) (*, B, (1))
(3) (+, A, (2))
(4) (^, F, G)
(5) (/, E, (4))
(6) (-, (3), (5))

四元式表示

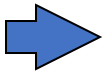
(1) (－, C, D , T1)
(2) (*, B, T1, T2)
(3) (+, A, T2 , T3)
(4) (^, F, G , T4)
(5) (/, E, T4, T5)
(6) (-, T3, T5 , T6)

栈式抽象机指令代码

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		
·····			

例：

a := b+c;



```
LDA (a)
LOD b
LOD c
ADD
STN
```

栈式抽象机指令代码

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

Outline

- 中间语言
- 一些语法成分的翻译
  - 声明语句
  - 表达式
  - 赋值语句
  - 控制语句
  - 过程调用

声明语句引起的翻译动作

- 当过程或程序块内部的声明语句被考察的时候，我们需要为局部于该过程的名字分配存储空间。
  - 对每个局部名字，都将在符号表中创建一个表项
  - 填写类型、相对存储地址(基址 + 偏移量)等相关信息
    - 相对地址指相对于静态数据区基址或活动记录基址的偏移量
- 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

单个过程中的声明语句

- 变量和过程设计
  - 设置全局变量：offset，跟踪下一个可用的相对地址
  - 过程enter(name, type, offset)为名字建立符号表表项
    - 综合属性type和width说明非终结符T的类型及宽度(或该类型的对象所占用的内存数)

P→ D	**{offset:=0}**
D→D;D	
D→id:T	**{enter(id.name, T.type, offset);** **offset:=offset + T.width}**
T→integer	**{T.type := integer;** **T.width := 4}**
T→real	**{T.type := real;** **T.width := 8}**
T→array[num]of T_1	**{T.type := array(num.val, T_1.type);** **T.width :=num.val * T_1.width}**
T→↑T_1	**{T.type := pointer(T_1.type);** **T.width := 4}**

声明语句-例子

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

对应的属性翻译文法为：

<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 $_{\uparrow t}$

声明语句-例子

动作程序

<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 $_{\uparrow t}$

- @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。假设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 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）

常量声明-例子

- **常量声明：**常量标识符通常被看作是全局名。
- **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>
```

- **常量声明语句的属性翻译文法为：**

```
<const del> →constant <type> ↑t <entity> ↑n :=<const expr>↑c, s @insert↓ t, n, c, s ;
<type> ↑t → real ↑t |integer ↑t |string ↑t

<const expr>↑c, s →<integer const >↑c, s |<real const>↑c, s
                   |<string const>↑c, s
```

常量声明-例子

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

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

翻译处理过程为： 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 填入符号表中

简单变量声明-例子

- 简单变量声明语句的输入文法为：

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

- 简单变量声明语句的属性翻译文法为：

<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** $_{\uparrow t,\, i}$

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-例子

<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

```
void svardef(int t，int i，char* n)
{
  int j = tableinsert(n，t，i)；/*给定一简单变量n的类型t，所需的数据空间大小i。tableinsert将n和属性i及t一起填入符号表。j用作临时变量，记录函数返回值。tableinsert返回值为0，表示变量重复定义；-1表示符号表溢出。 */
if(j==0)
    errmsg(duplident，statmtno)；
else if (j==-1) {
    errmsg(tblovflow，statno)；
    abort；/* 符号表溢出，终止编译程序 */
     }
}
```

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

@svardef 动作符号是把n，i 和t 填入符号表中。

@allocsv的动作是分配存储

```
void allocsv(int i)     /* 存储分配 */
{
/*设简单变量所要求的数据空间大小为i，由此可更新指针codeptr的内容*/
    codeptr = codeptr + i；
}
```

数组变量声明

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/内情向量）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

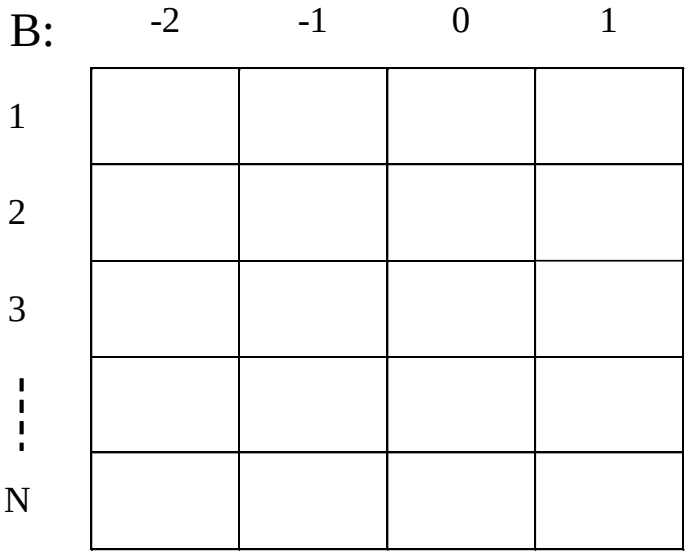
- **动态数组**：大小只有在运行时才能最后确定。
- 在编译时仅为该模板分配一个空间，而模板本身的内容将在运行时才能填入。

数组变量声明

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)

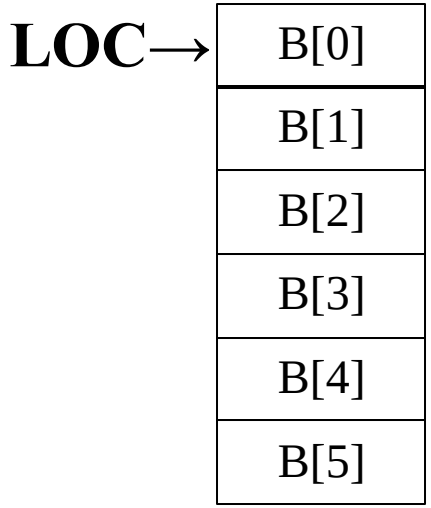
LOC是数组首地址

(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

数组变量声明-一维数组

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

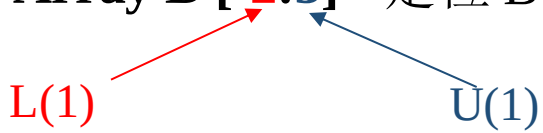


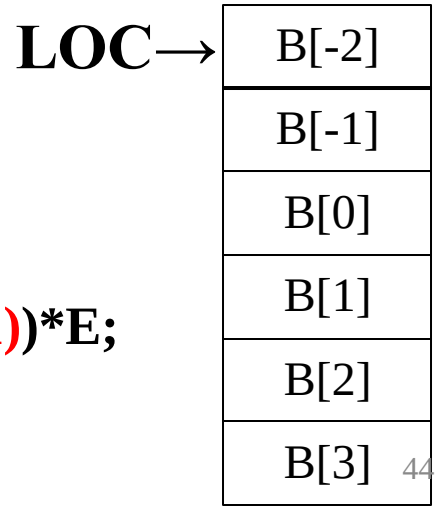
- **先考虑 1 维： Array B[5]**

  如何通过LOC, 定位 B[i]的地址？

  ADDR = LOC + i*E;

- **B的定义带有上下界：Array B [-2:3]** 定位 B[i]的地址？

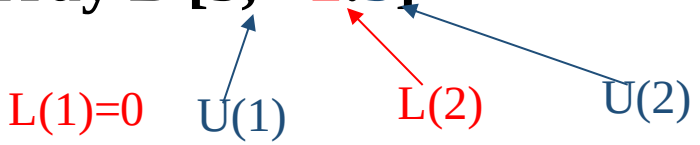




ADDR = LOC + (x - L(1))*E;

数组变量声明-二维数组

二维数组B的定义带有上下界：Array B [3, -2:3]



LOC→

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

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

ADDR = LOC

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

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

其中P(i) = 1 （if i=2）

= U(i)-L(i)+1 （if i=1）

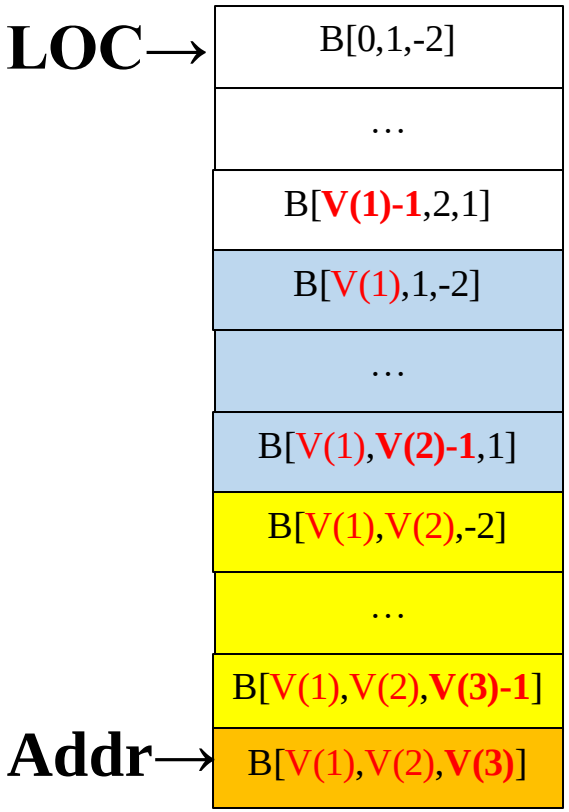
数组变量声明-三维数组

三维数组B的定义带有上下界：**Array B [2, 1:2, -2:1]**

L(1)=0

U(1) L(2) U(2) L(3) U(3)

定位 B[V(1), V(2), V(3)]的地址？



ADDR = LOC

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

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

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

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

其中P(i) = 1 （if i=n）

= (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] （if 1<=i<n）

ADDR = LOC

+ (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

数组变量声明-总结

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)

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：

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

其中

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

注：E为数组元素大小（字节数）

数组变量声明-优化

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

其中

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

可变部分　　不变部分（可提前计算）

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

可变部分　　不变部分（可提前计算）

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

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

array B (3, -2: 1)　char ;

则 P(2) = 1

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

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

= -[1×4 + (-2)×1]×E
= -2×E

数组元素B(2 , 1) 的地址为：

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

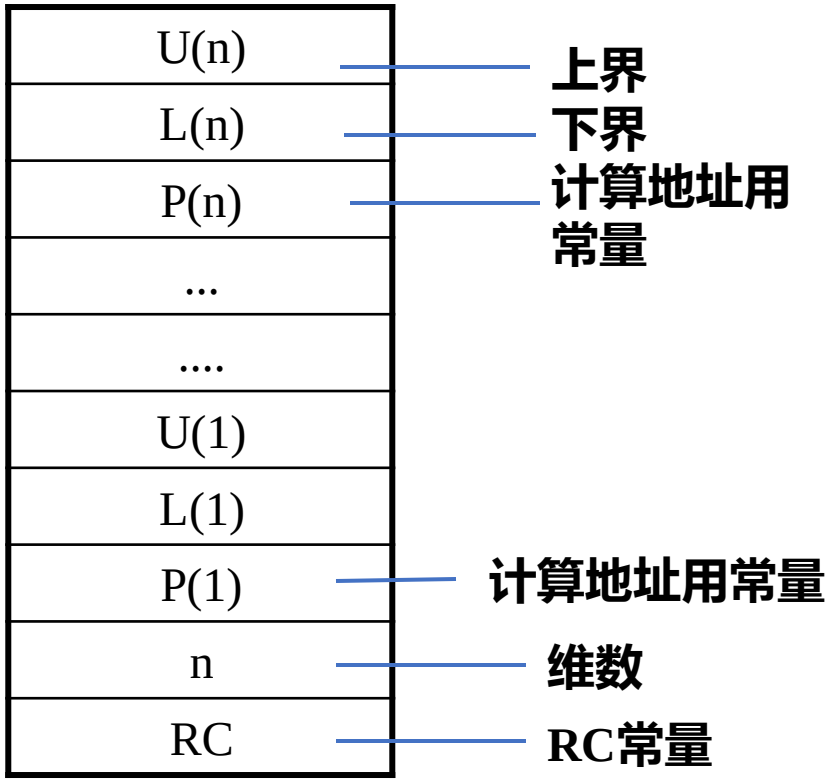
= LOC – 2E+(2×4+1×1)×E

= LOC + 7×E

数组变量声明-信息向量

- **数组信息向量表（模板）**

一般形式：



功能：1、用于计算下标变量地址
2、检查下标是否越界

注：1、数组模板所需的空间大小取决于数组的维数，即3n+2，所以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

数组变量声明-信息向量例子

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

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

数组变量声明-例子

`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
```

声明语句的属性翻译文法为：

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

$$\text{<sublist>}_{\uparrow j} \rightarrow \text{<subscript>}\ @\text{dimen\#}_{\uparrow j} \mid \text{<subscript>, <sublist>}_{\uparrow j}\ @\text{dimen\#}_{\uparrow j}$$

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

数组变量声明-例子

<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。即执行动作： p:= p+2; j := 0

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

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)

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

Outline

- 6.1 中间语言
- 6.2 一些语法成分的翻译
  - 6.2.1 说明语句
  - 6.2.2 表达式
  - 6.2.3 赋值语句
  - 6.2.4 控制语句
  - 6.2.5 过程调用

表达式

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。

表达式-例子

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 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;
```

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

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

表达式-例子

对于输入表达式 x + y * 3：

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>)

<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。

表达式-类型检查和转换

- **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
- **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

表达式-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 doulbe
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!

表达式-例子

<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

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。

语义动作

次栈顶

栈顶

```
char* add(char *t, char *s)    /*输入参数(形参)为两个操作数的类型*/
{
if((strcmp(t, "real")==0)&& (strcmp(s, "int")==0))  /*如果t为实数和s为整数*/
{
  emit(CONVER, $top-1);      /*生成整型转为实型的指令*/
  emit("ADD");
  return "real";
            }
if((strcmp(t, "int")==0)&& (strcmp(s,"real")==0))
{
  emit(CONVER, $top);
  emit("ADD");
  return "real";
            }
emit("ADD");
return t;
}
```

越界检查，计算地址公式中的可变部分

```
int offset(int k, char* t);
    {/*形参k为数组下标，形参t为操作数类型*/
      k = k+1;
      if(t!= "int")
errmsg("作为数组下标应是整型表达式", statno);
else emitl("OFFSET", k);
return k;
    }
```

```
void checkdim(int k, int j)
 /*形参k为数组下标，形参j为数组变量在符号表的位置*/
    {
      if (k!= symbtbl[j]->dim)
            errmsg("维数与声明不匹配", statno);
      else
{
          emit(ARREF);
          emit(DEREF);
}
    }
```

生成数组元素地址

加载数组元素内容

```
int template(int j)
{    /*形参j为数组变量在符号表的位置*/
      emitl("TEMPLATE", symbtbl[j]->objaddr);
      k=0;  /*下标计数器初始化*/
      return k;
    }
```

模板入口地址

表达式-例子

★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

布尔表达式的作用

- 用作计算逻辑值
- 用作控制语句（如if语句，while语句等）
- 布尔表达式和关系表达式
  - 布尔表达式是用布尔运算符作用到布尔变量或关系表达式上而组成的。（not、and、or）
  - 关系表达式是由关系运算符连接的算术表达式

文法：E->E∧E | E∨E | ¬ E | (E) | i | i rop i | true | false
其中：
逻辑运算符：∧，∨，¬ (and, or, not)
布尔运算量：逻辑值，布尔变量，关系表达式
关系式：E1 rop E2 （rop为“>, >=, =, !=, <, <=” 其一）

布尔表达式值的计算

- 如同计算算术表达式一样，一步不差的从表达式各部分的值计算出整个表达式的值
- 采用优化措施计算布尔表达式的值
  - 计算A or B时，如果A为真，则不用计算B，结果为真
  - 计算A and B时，如果A为假，不用计算B，结果为假
  - 一般的计算公式如下：
    - A or B：if A then true else B
    - A and B：if A then B else false
    - not A：if A then false else true

把布尔表达式翻译成地址代码的翻译模式

$E \rightarrow E_1$ or E_2	{E.place := newtemp; emit(E.place ':=' E_1.place 'or' E_2.place}
$E \rightarrow E_1$ and E_2	{E.place := newtemp; emit(E.place ':=' E_1.place 'and' E_2.place}
E→not E_1	{E.place := newtemp; emit(E.place ':=' 'not' E_1.place}
E→ (E_1)	{E.place := E_1.place}

E→id1 relop id2	{E.place := newtemp;
	emit('if' id1.place relop.op id2.place 'goto' nextstat+3);
	emit(E.place ':=' '0');
	emit('goto' nextstat+2);
	emit(E.place ':=' '1')}

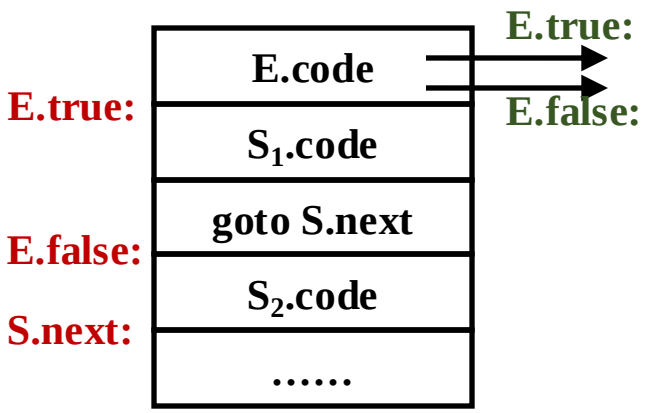
E→id	{E.place := id.place}

a<b or c<d and e<f

100: if a<b goto 103
101: T1 := 0
102: goto 104
103: T1:= 1
104: if c<d goto 107
105: T2:=0
106: goto 108
107: T2:=1
108: if e<f goto 111
109: T3:=0
110: goto 112
111: T3:=1
112: T4:=T2 and T3
113: T5:=T1 or T4

作为条件控制的布尔式翻译

- 出现在条件语句中的布尔表达式，作用仅在于控制对执行语句的选择。if E then S_1 else S_2



if a>c or b<d then S1 else S2

```
        if a>c goto L2
        goto L1
L1:     if b<d goto L2
        goto L3
L2:     (S1的三地址码序列)
        goto Lnext
L3:     (S2的三地址码序列)
Lnext:
```

产生布尔表达式三地址代码的语义规则

- 函数newlabel产生新的符号标号
- if E then S_1 else S_2。假定E形如a<b，则将生成如下E的代码：
  - if a<b goto E.true

    goto E.false
- 对于上述翻译的讨论：
  - 如果E为E_1 or E_2，如果E_1为真，则立即可知E为真，于是E_1.true与E.true是相同的；若E_1为假，则必须对E_2求值，因此我们置E_1.false为E_2的代码的第一条指令的标号，而E_2的真假出口可以分别与E的真假出口相同。

产生式	语义规则
$E \rightarrow E_1$ or E_2	E_1.true:= E.true; E_1.false:= newlabel; E_2.true:= E.true; E_2.false:= E.false; E.code:= E_1.code || gen(E_1.false ':') || E_2.code
$E \rightarrow E_1$ and E_2	E_1.true:= newlabel; E_1.false:= E.false; E_2.true:= E.true; E_2.false:= E.false; E.code:= E_1.code || gen(E_1.true ':') || E_2.code
$E \rightarrow$ not E_1	E_1.true:= E.false; E_1.false:= E.true; E.code:= E_1.code
$E \rightarrow (E_1)$	E_1.true:= E.true; E_1.false:= E.false; E.code:= E_1.code
$E \rightarrow id_1$ relop id_2	E.code :=gen('if' id_1.place relop.op id_2.place 'goto' E.true) || gen('goto' E.false)
E→true	E.code:= gen('goto' E.true)
E→false	E.code:= gen('goto' E.false)

Outline

- 6.1 中间语言
- 6.2 一些语法成分的翻译
  - 6.2.1 声明语句
  - 6.2.2 表达式
  - 6.2.3 赋值语句
  - 6.2.4 控制语句
  - 6.2.5 过程调用

赋值语句

- 赋值语句中表达式的类型
  - 整型
  - 实型
  - 数组

- 将赋值语句转换为三地址码要做的工作
  - 在符号表中查找名字
  - 存取其中的元素

符号表中的名字

S→id := E **{p:=lookup(id.name); //查找符号表，看变量id是否说明**
if p ≠ nil then
emit(p ':=' E.place) //生成三地址语句
else error}

E→E_1 + E_2 **{E.place := newtemp; //结果指向临时变量**
emit(E.place ':=' E_1.place '+' E_2.place}

E→ E_1 * E_2 **{E.place := newtemp;**
emit(E.place ':=' E_1.place '*' E_2.place}

E→ -E_1 **{E.place := newtemp;**
emit(E.place ':=' 'uminus' E_1.place}

E→ (E_1) **{emit(E.place := E_1.place}**

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

- lookup 操作支持最近嵌套原则。
- 其上下文是由前面定义的过程，当作用于name时，需要先检查name是否已经在符号表中定义过了。

符号表中的名字

S→id := E **{p:=lookup(id.name); //查找符号表，看变量id是否说明**
if p ≠ nil then
emit(p ':=' E.place) //生成三地址语句
else error}

$E \rightarrow E_1 + E_2$ **{E.place := newtemp; //结果指向临时变量**
emit(E.place ':=' E_1.place '+' E_2.place}

$E \rightarrow E_1 * E_2$ **{E.place := newtemp;**
emit(E.place ':=' E_1.place '*' E_2.place}

$E \rightarrow -E_1$ **{E.place := newtemp;**
emit(E.place ':=' 'uminus' E_1.place}

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

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

例：a:=-b*c+d 按该方案的语法制导翻译过程。 按归约次序依次为

1）E→b，id.name为b，设E.place为b;
2）E→-E1,设newtemp为t1，emit生成三地址语句t1:= uminus b;
3) E→c，id.name为c，设E.place为c;
4) E→E1*E2,设newtemp为t2，此时E1.place为t1， E2.place为c，emit生成三地址语句t2:= t1*c;
5) E→d，id.name为d，设E.place为d;
6) E→E1+E2,设newtemp为t3，此时E1.place为t2， E2.place为d，emit生成三地址语句t3:= t2+d;
7) S→a:=E, id.name为a，此时E.place为t3，emit生成三地址语句a:= t3;

赋值语句中的类型转换

- 变量和常量有许多不同的类型，因而编译器或者拒绝某种混合类型的操作，或者生成适当的强制（类型转换）指令。
- 假设有两种类型：
  - $E \rightarrow E_1 + E_2$ {E.type :=
    if E1.type = integer and
    E2.type= integer then integer
    else real}

赋值语句中的类型转换

(1) S→L := E
(2) E→E + E
(3) E→(E)
(4) E→L
(5) L→Elist]
(6) L→id
(7) Elist→Elist,E
(8) Elist→ id[E

引入类型转换功能的三地址语句t:=inttoreal x，将整型变量x转换为实型变量t

```
E.Place := newtemp;
if E1.type = integer and E2.type = integer then begin
    emit(E.place ':=' E1.place 'int+' E2.place);
    E.type := integer
end
else if E1.type = real and E2.type = real then begin
    emit(E.place ':=' E1.place 'real+' E2.place);
    E.type := real
end
else if E1.type = integer and E2.type = real then begin
    u:=newtemp;
    emit(u ':=' 'inttoreal' E1.place;
    emit(E.place ':=' u 'real+' E2.place);
    E.type := real
end
else if E1.type = real and E2.type = integer then begin
    u:=newtemp;
    emit(u ':=' 'inttoreal' E2.place;
    emit(E.place ':=' E1.place 'real+' u);
    E.type := real
end
else E.type := type_error;
```

赋值语句-例子

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

X:= Y+X;

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

@setL是设置变量为“左值”（被赋变量），即将属性L置true

```
procedure setL;
    return (true );
end;
```

指示取变量地址

@resetL是设置变量为非被赋变量，即把属性L置成false

```
procedure resetL;
    return (false );
end;
```

指示取变量之值

@storin进行类型转换，并生成STN指令

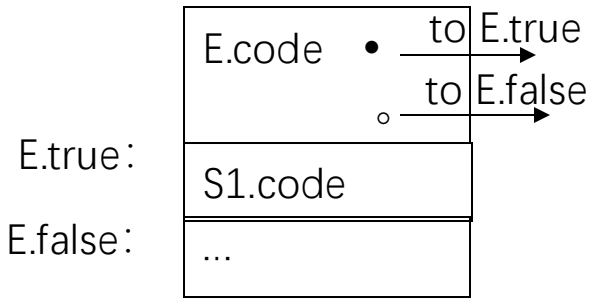
```
procedure storin(t,s);
  string t, s;
  if t ≠ s
  then /*生成进行类型转换的指令*/
  emit('STN');
end;
```

Outline

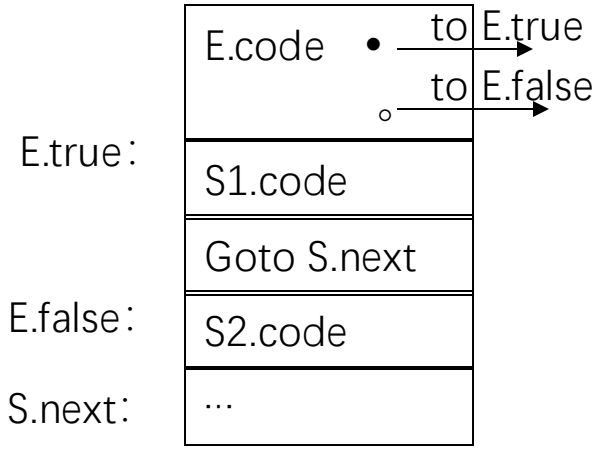
- 6.1 中间语言
- 6.2 一些语法成分的翻译
  - 6.2.1 说明语句
  - 6.2.2 赋值语句
  - 6.2.3 布尔表达式
  - 6.2.4 控制语句
  - 6.2.5 过程调用

6.2.4 控制流语句的翻译

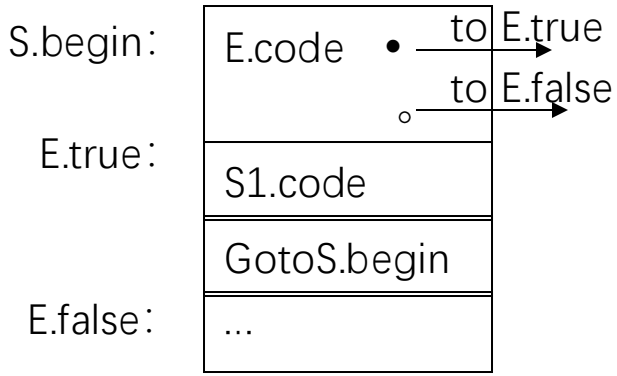
- 常见控制结构：序列、条件分支、循环
- 控制语句在源程序中的可以并列也可嵌套
- 控制转移



If-then代码结构



If-then-else代码结构



while-do代码结构

循环语句-If

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$_{\downarrow z}$

循环语句-If

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@labprod是把从继承属性y得到的标号设置到目标程序中

```
void labprod(char* y)
/*形参y为继承属性中的标号*/
{
    setlab(y);
}
```

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

```
char* brf()  /*无继承属性。*/
{
    char* labx;
    labx=genlab();  /* 生成标号 */
    emit("BRF", labx);
    return labx;
}
```

动作程序@br是是生成JMP指令，并将转移标号返回给属性z

```
char* br()   /*无继承属性*/
{
    char* labz;
    labz=genlab();      /* 生成标号
*/
    emit("BR", labz);
    return labz;
}
```

循环语句-例子

- **For循环的属性文法为：**

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 语句例子：

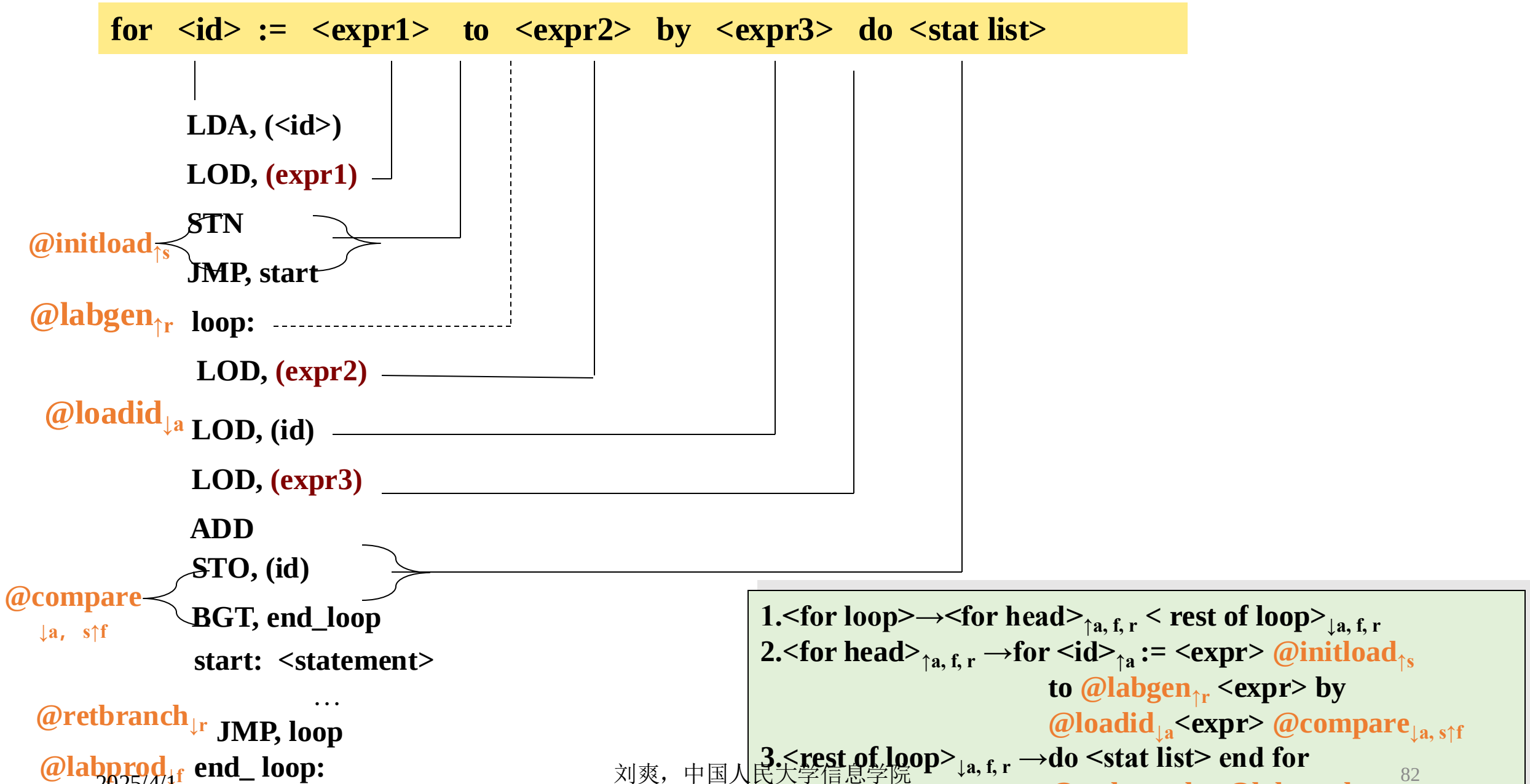
```
for i:= 1 to n by z do
    <statement>
    ...
end for;
```

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

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



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$_{\downarrow f}$

Outline

- 6.1 中间语言
- 6.2 一些语法成分的翻译
  - 6.2.1 说明语句
  - 6.2.2 赋值语句
  - 6.2.3 布尔表达式
  - 6.2.4 控制语句
  - 6.2.5 过程调用

6.2.5 过程调用

- 实质：把程序控制转移到子程序(过程段)
- 语义动作：
  - 传递参数信息（实参的地址）给被调用的子程序
  - 告诉子程序在它工作完毕后返回到什么地方。
- 例如：过程调用CALL S (A+B，Z)

  将被翻译成：

```
T:=A+B
par T          //第一个实参地址
par Z          //第二个实参地址
call S         //转子指令
```

过程调用

- **翻译模式**

（1）S→call id（Elist）

{For　队列QUEUE 的每一项p　Do

emit('par',p);

emit('call',id.place)}

（2） Elist → Elist , E

{把E.PLACE 排在QUEUE 的末端 }

（3） Elist →E

{建立一个QUEUE， 只包含一项E.PLACE}

类型检查

- 验证对数据的操作是否遵守源语言的类型规则
- 步骤
  - 在符号表中记录标识符的类型信息。
  - 在语义动作中加入对类型规则的检查。

实例

S → id:=E
{ if id. type= E. type
then S. type:= void;
else S. type:= type_ error; }

S → if E then S1
{if E. type= Boolean
then S. type:= S1. type;
else S. type := type_ error; }

S → while E do S1
{if E. type= boolean
then S. type:= S1. type;
else S. type:= type_ error; }

过程调用和返回

参数传递的基本形式	传值（call by value)	传地址（call by reference)
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	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
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	通过对形参的间接访问来访问相应的实参
特点:	数据传递是单向的	结果随时送回调用段

过程调用和返回-例子

例：有过程调用：
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

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>:….

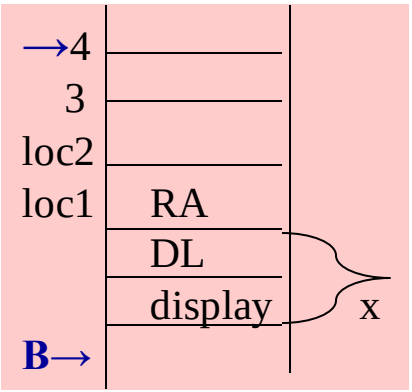
传值调用

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:symbal, int: cur, string: repl);
```



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。

x: display+DL

```
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）

- **过程调用的属性文法为：**

<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>$_{\downarrow i,z}$

```
int lookupproc(char* n)  /*形参n为被调用过程的名字*/
{
        int i=0;
        i=lookup(n);  //查找符号表，返回符号表位置
        if(i<1){
                error("过程"，n，"未定义"，statno);
                errorrecorvery(panic);
                i=0;
                z=0; //z表示过程定义的形参的个数，全局变量
                return i;
        }else{
                z = symbtbl[i]->dim; //symbtbl为符号表
                return i;
        }}
```

<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>$_{\downarrow i,z}$

```
int chktype(char* t，int i，int m，int z)
{
      if(z< 1){
            error(“变元数大于过程的形参数”，symbtbl[i]->name，statno)；
            return z；
      }
      m=m+1；/*实参计数*/
      if(strcmp(t，symbtbl[i+m]->type)!=0){
            error(“变元和形参的类型不匹配”，symbtbl[i+m]->type，statno)；
      }
      z=z-1；/*减去已匹配的形参数*/
      return z；/*剩下待匹配的形参数*/
}
```

```
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：

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

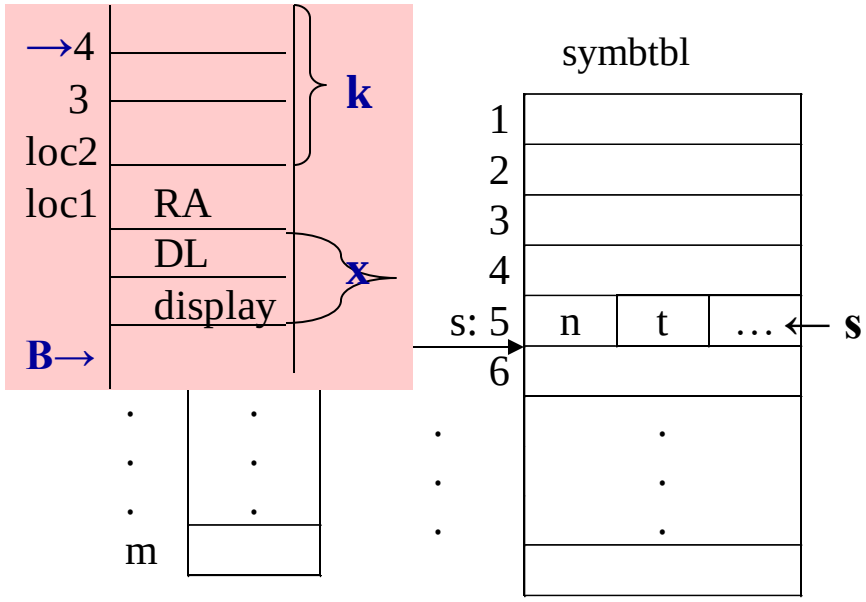
@tblinsert 是把过程名和它的形参名填入符号表中：

```
void tblinsert(char* t，char *n)
{
   int hloc;
   if(lookup(n)>0) /*名字已在符号表*/
            error(“名字定义重复”，statno);
   else/*将新名字填入符号表*/
   {
            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；/*增加全局符号表下标s*/
   }
}
```

最后得到的形参个数

```
void emitstores(int k)
{
    int i=0;
    emitl("ALLOCATE", k+x+…);
    emitl("STO ", ll, x+1); / *保存返回地址* /
    for(i=k+x+1; i>=x+2; i--)  / *存贮参数值* /
    {
        emitl("STO ", ll, i)
     }
}
```



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 */
```

过程调用和返回-例子

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）