

# 编译技术



胡春明  
[hucm@buaa.edu.cn](mailto:hucm@buaa.edu.cn)

2019.9-2019.12



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：



## 第七章 源程序的中间形式

- 波兰表示
- N - 元表示
- 抽象机代码

## 波兰表示

## 7.1 波兰表示

一般编译程序都生成中间代码，然后再生成目标代码，主要优点是可移植(与具体目标程序无关)，且易于目标代码优化。有多种中间代码形式：

波兰表示      N - 元组表示      抽象机代码

### 波兰表示

由波兰逻辑学家 J.Lukasiewicz 提出

- 前缀表示：<操作符> <操作数序列>
- 后缀表示：<操作数序列> <操作符>

前缀表达： (波兰表达)      + 3 5

后缀表达： (逆波兰表达)      3 5 +

## 7.1 波兰表示

一般编译程序都生成中间代码，然后再生成目标代码，主要优点是可移植(与具体目标程序无关)，且易于目标代码优化。有多种中间代码形式：

波兰表示      N - 元组表示      抽象机代码

### 波兰表示

算术表达式:       $F * 3.1416 * R * (H + R)$

转换成波兰表示:       $F3.1416 * R * HR + *$

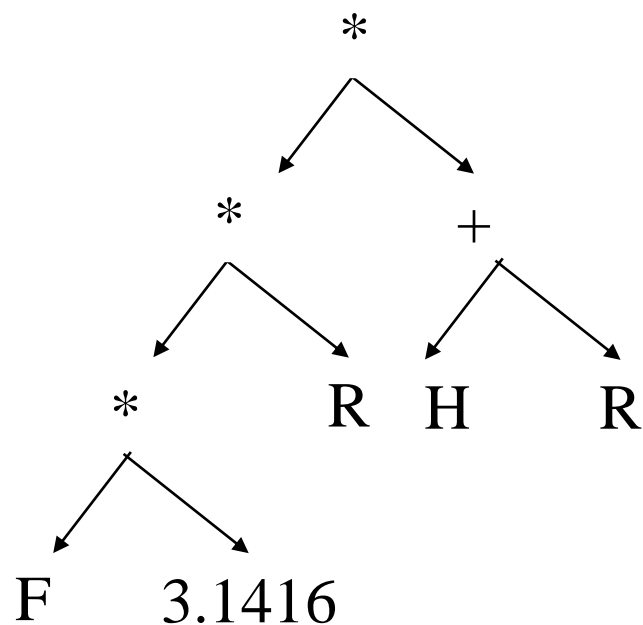
赋值语句:       $A := F * 3.1416 * R * (H + R)$

波兰表示:       $AF3.1416 * R * HR + * :=$

## 7.1 波兰表示

### 波兰表示 从语法树的角度看 “波兰表示”

算术表达式:  $F * 3.1416 * R * (H + R)$



前序遍历

中序遍历

后序遍历

$E ::= E + T \mid T$   
 $T ::= T * F \mid F$   
 $F ::= (E) \mid i$

## 7.1 波兰表示

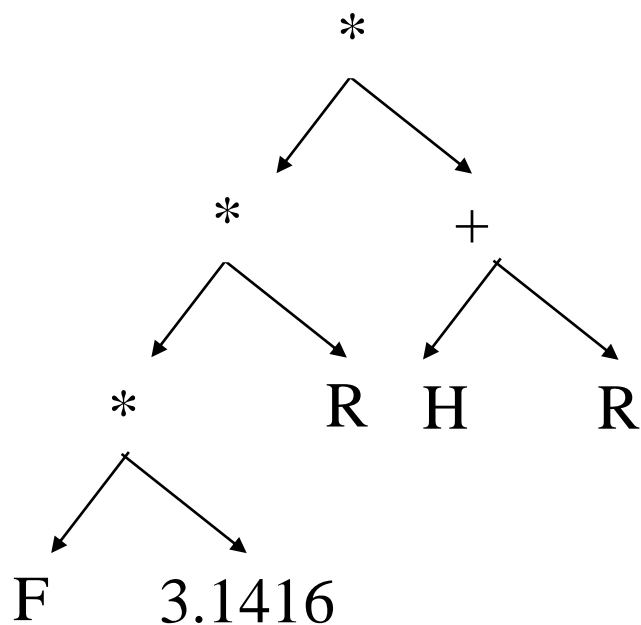
### 波兰表示

从语法树的角度看 “波兰表示”

$E ::= E + T \mid T$   
 $T ::= T * F \mid F$   
 $F ::= (E) \mid i$

算术表达式:

$F * 3.1416 * R * (H + R)$



逆波兰表示:  $F3.1416 * R * H R + *$

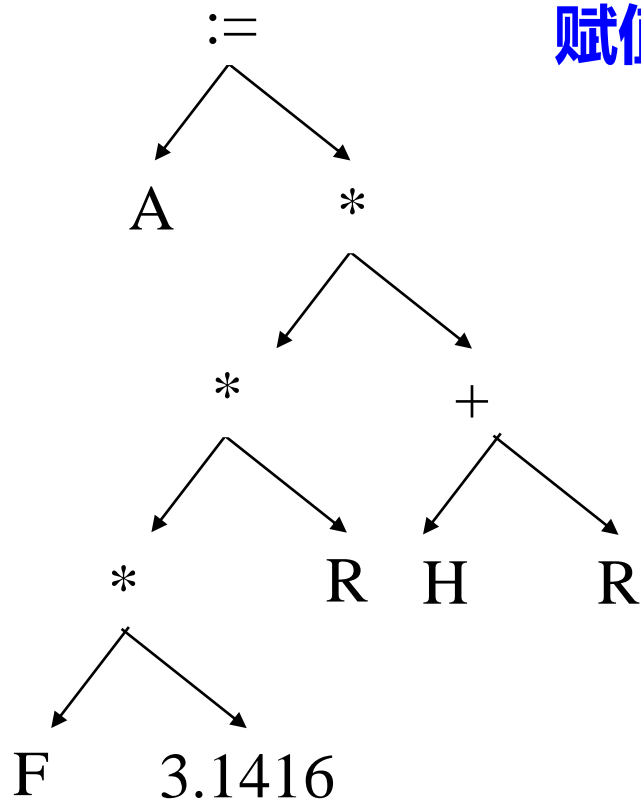


## 7.1 波兰表示

### 波兰表示

从语法树的角度看“波兰表示”

赋值语句:  $A := F * 3.1416 * R * (H + R)$



前序遍历

中序遍历

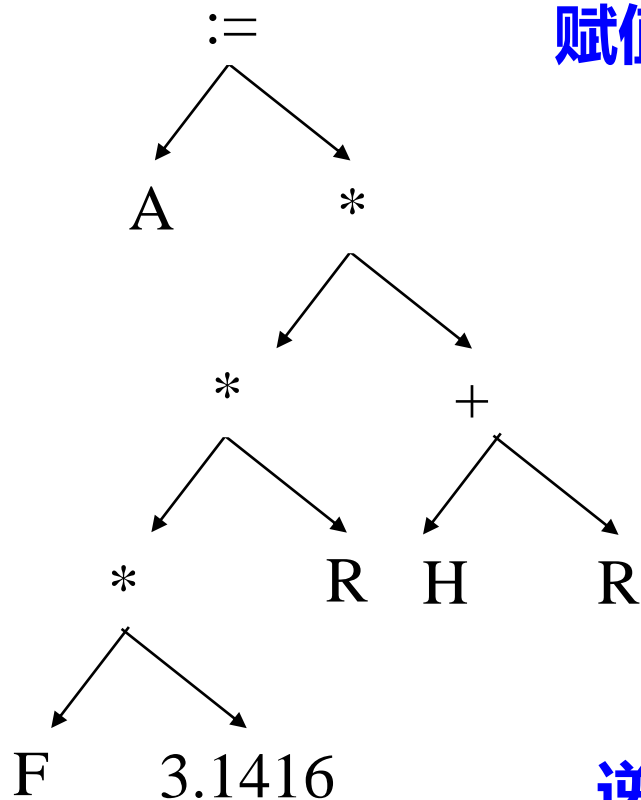
后序遍历

## 7.1 波兰表示

### 波兰表示

从语法树的角度看“波兰表示”

赋值语句:  $A := F * 3.1416 * R * (H + R)$



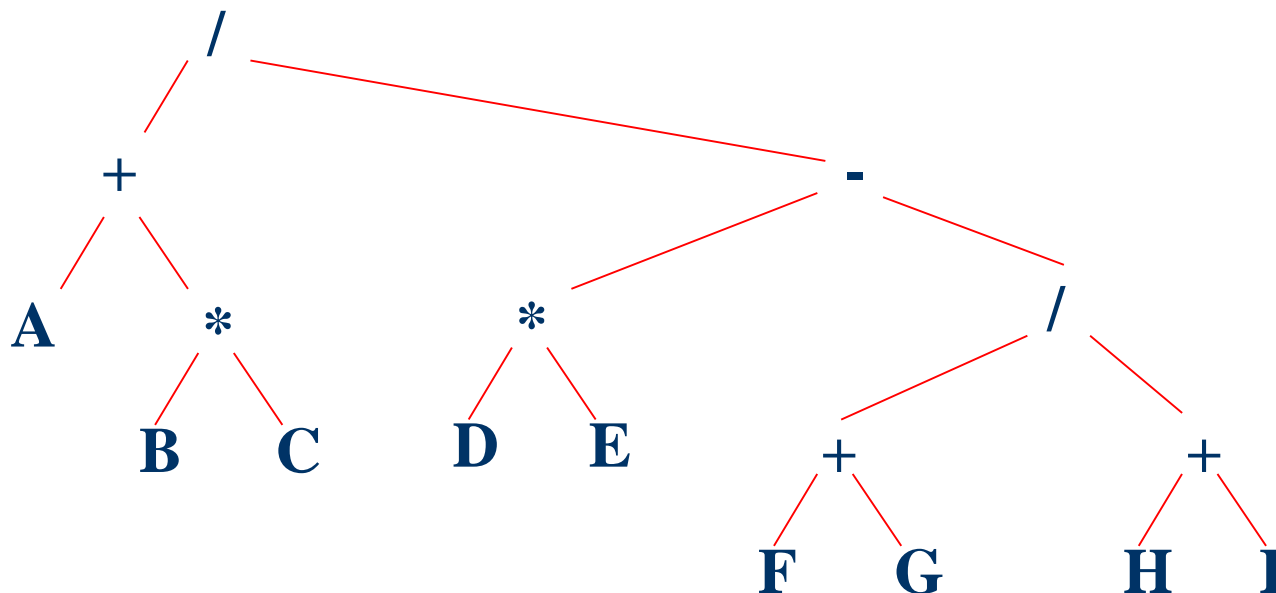
前序遍历

中序遍历

后序遍历

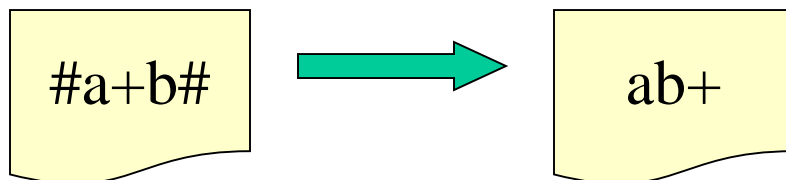
逆波兰表示:  $AF3.1416 * R * HR + * :=$

例：  $(A+B * C) / (D * E - (F+G) / (H+I))$

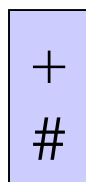


前序遍历（根左右）：  $/+A*BC-*DE/+FG+HI$  （波兰表达）  
 后序遍历（左右根）：  $ABC*+DE*FG+HI+/-/$  （逆波兰表达）  
 中序遍历（左根右）：  $A+B*C/D*E-F+G/H+I$

## 构造一个算法，生成“波兰表示”



操作符栈



#优先级最低

**算法：**

设一个操作符栈；当读到操作数时，立即输出该操作数，当扫描到操作符时，与栈顶操作符比较优先级，若栈顶操作符优先级高于栈外，则输出该栈顶操作符，反之，则栈外操作符入栈。

## 转换算法

波兰表示

操作符栈

算术表达式:

$F * 3.1416 * R * (H + R)$

输入

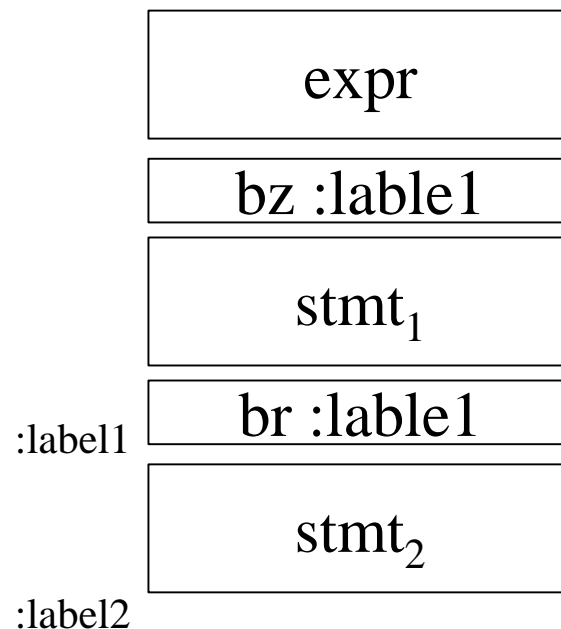
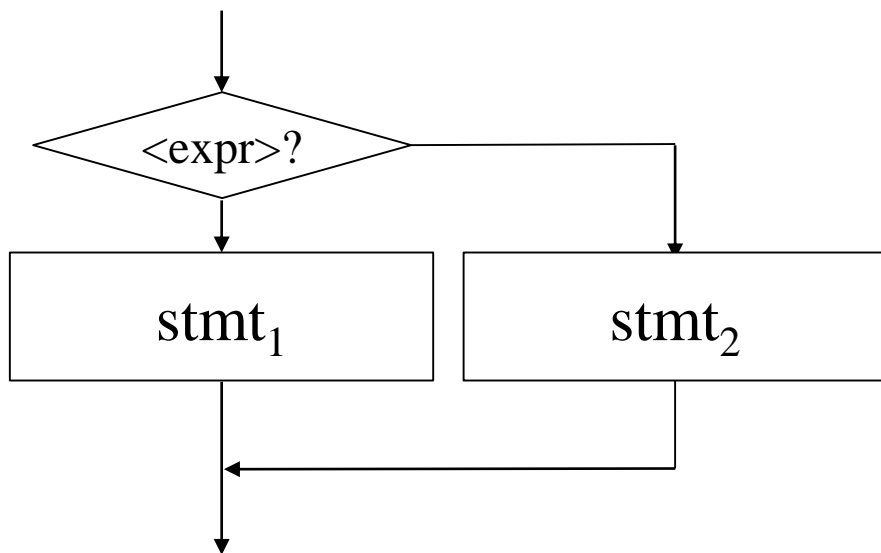
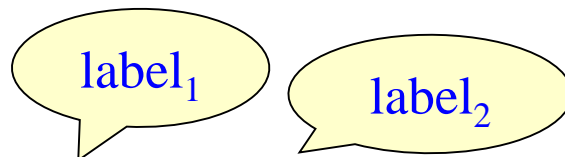
输出

		F * 3.1416 * R * (H + R)	
		* 3.1416 * R * (H + R)	F
*		3.1416 * R * (H + R)	F
*	.	* R * (H + R)	F 3.1416
*	>	R * (H + R)	F 3.1416 *
*	.	* (H + R)	F 3.1416 * R
*	>	(H + R)	F 3.1416 * R *
*	<.	H + R)	F 3.1416 * R *
*(		+ R)	F 3.1416 * R * H
*(	<.	R)	F 3.1416 * R * H
*( +	.	)	F 3.1416 * R * HR
*( +		)	F 3.1416 * R * HR +
*(	>	)	F 3.1416 * R * HR + *

波兰表示:  $F3.1416 * R * HR + *$

## if 语句的波兰表示

**if 语句** : if <expr> then <stmt<sub>1</sub>> else <stmt<sub>2</sub>>



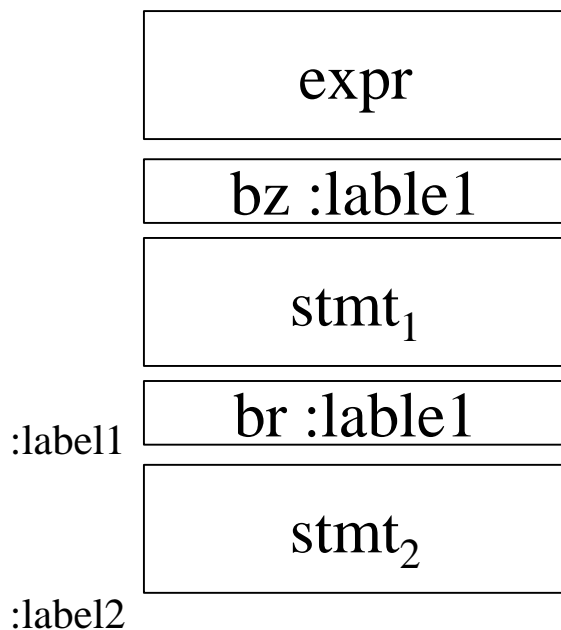
## if 语句的波兰表示

**if 语句** : if <expr> then <stmt<sub>1</sub>> else <stmt<sub>2</sub>>

label<sub>1</sub>

label<sub>2</sub>

**波兰表示为** : <expr><label<sub>1</sub>>BZ<stmt<sub>1</sub>><label<sub>2</sub>>BR<stmt<sub>2</sub>>



**BZ: 二目操作符**

若<expr>的计算结果为0 (false),  
则产生一个到<label<sub>1</sub>>的转移

**BR: 一目操作符**

产生一个到<label<sub>2</sub>>的转移

波兰表示为 :  $\langle \text{expr} \rangle \langle \text{label}_1 \rangle \text{BZ} \langle \text{stmt}_1 \rangle \langle \text{label}_2 \rangle \text{BR} \langle \text{stmt}_2 \rangle$

由if语句的波兰表示可生成如下的目标程序框架:

```
    <expr>  
    BZ label1  
    <stmt1>  
    BR label2  
label1: <stmt2>  
label2:
```

其他语言结构也很容易将其翻译成波兰表示,  
使用波兰表示的问题: 优化不是十分方便。



其他语言结构也很容易将其翻译成波兰表示，  
**使用波兰表示的问题：优化不是十分方便。**

波兰表达式 隐含了 “栈操作”  
能否将波兰表达式拆成一组 “原子操作” ？

## N元表示

## 7.2 N - 元表示

在该表示中，每条指令由n个域组成，通常第一个域表示操作符，其余为操作数。

常用的n元表示是：      三元式      四元式

**三元式**

操作符	左操作数	右操作数
-----	------	------

表达式的三元式：

$w * x + (y + z)$



(1)  $*$ ,  $w$ ,  $x$

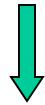
(2)  $+$ ,  $y$ ,  $z$

(3)  $+$ , (1), (2)

**第三个三元式中的操作数(1)**  
**(2)表示第(1)和第(2)条三元式的计算结果。**

## 条件语句的三元式:

```
if x > y then
    z := x;
else z := y+1;
```



- (1) -, x, y
- (2) BMZ, (1), (5)
- (3) :=, z, x
- (4) BR, , (7)
- (5) +, y, 1
- (6) :=, z, (5)
- (7)

:

其中:

**BMZ:** 是二元操作符,测试第二个域的值,若 $\leq 0$ ,则按第3个域的地址转移,若 $> 0$ ,则顺序执行。

**BR:** 一元操作符,按第3个域作无条件转移。

**使用三元式不便于代码优化，因为优化要删除一些三元式，或对某些三元式的位置要进行变更，由于三元式的结果(表示为编号)，可以是某个三元式的操作数，随着三元式位置的变更也将作相应的修改，很费事。**

**间接三元式：**

**为了便于在三元式上作优化处理，可使用间接三元式**

**三元式的执行次序用另一张表表示,这样在优化时，三元式可以不变，而仅仅改变其执行顺序表。**

例:  $A := B + C * D / E$   
 $F := C * D$

用间接三元式表示为:

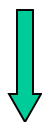
操作	三元式
1. (1)	(1) $*$ , C, D
2. (2)	(2) $/$ , (1), E
3. (3)	(3) $+$ , B, (2)
4. (4)	(4) $:=$ , A, (3)
5. (1)	(5) $:=$ , F, (1)
6. (5)	

## 四元式表示 (TAC)

操作符	操作数1	操作数2	结果
-----	------	------	----

**结果：**通常是由编译引入的临时变量，可由编译程序分配一个寄存器或主存单元。

**例：**  $(A + B) * (C + D) - E$



$+$ , A, B, T1  
 $+$ , C, D, T2  
 $*$ , T1, T2, T3  
 $-$ , T3, E, T4

式中T1, T2, T3, T4  
为临时变量，由四  
元式优化比较方便

## 四元式表示 (TAC)

操作符	操作数1	操作数2	结果
-----	------	------	----

```

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

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

```

_t0 = b + c;
a = _t0 + d;
_t1 = a * a;
_t2 = b * b;
b = _t1 + _t2;
    
```



## 四元式表示 (TAC)

操作符	操作数1	操作数2	结果
-----	------	------	----

```

int x;
int y;
int z;

if (x < y)
    z = x;
else
    z = y;

z = z * z;
    
```

```

    t0 = x < y;
    IfZ _t0 Goto _L0;
    z = x;
    Goto _L1;
_L0:
    z = y;
_L1:
    z = z * z;
    
```

## 四元式表示 (TAC)

操作符	操作数1	操作数2	结果
-----	------	------	----

```
int x;
int y;
```

```
while (x < y) {
    x = x * 2;
}
```

```
y = x;
```

```
_L0:
    t0 = x < y;
    IfZ _t0 Goto _L1;
    x = x * 2;
    Goto _L0;
_L1:
    y = x;
```

## 四元式表示 (TAC)

操作符	操作数1	操作数2	结果
-----	------	------	----

```
void main() {
    int x, y;
    int m2 = x * x + y * y;

    while (m2 > 5) {
        m2 = m2 - x;
    }
}
```

```
main:
    BeginFunc 24;
    _t0 = x * x;
    _t1 = y * y;
    m2 = _t0 + _t1;
_L0:
    _t2 = 5 < m2;
    IfZ _t2 Goto _L1;
    m2 = m2 - x;
    Goto _L0;
_L1:
    EndFunc;
```

## 四元式表示 (TAC)

操作符	操作数1	操作数2	结果
-----	------	------	----

```

void SimpleFn(int z) {
    int x, y;
    x = x * y * z;
}

void main() {
    SimpleFunction(137);
}

```

```

_SimpleFn:
    BeginFunc 16;
    _t0 = x * y;
    _t1 = _t0 * z;
    x = _t1;
    EndFunc;

main:
    BeginFunc 4;
    _t0 = 137;
    PushParam _t0;
    LCall _SimpleFn;
    PopParams 4;
    EndFunc;

```

## 抽象语法树

## 7.3 抽象语法树

### 中间代码的图结构表示

#### 抽象语法树:

用树型图的方式表示中间代码

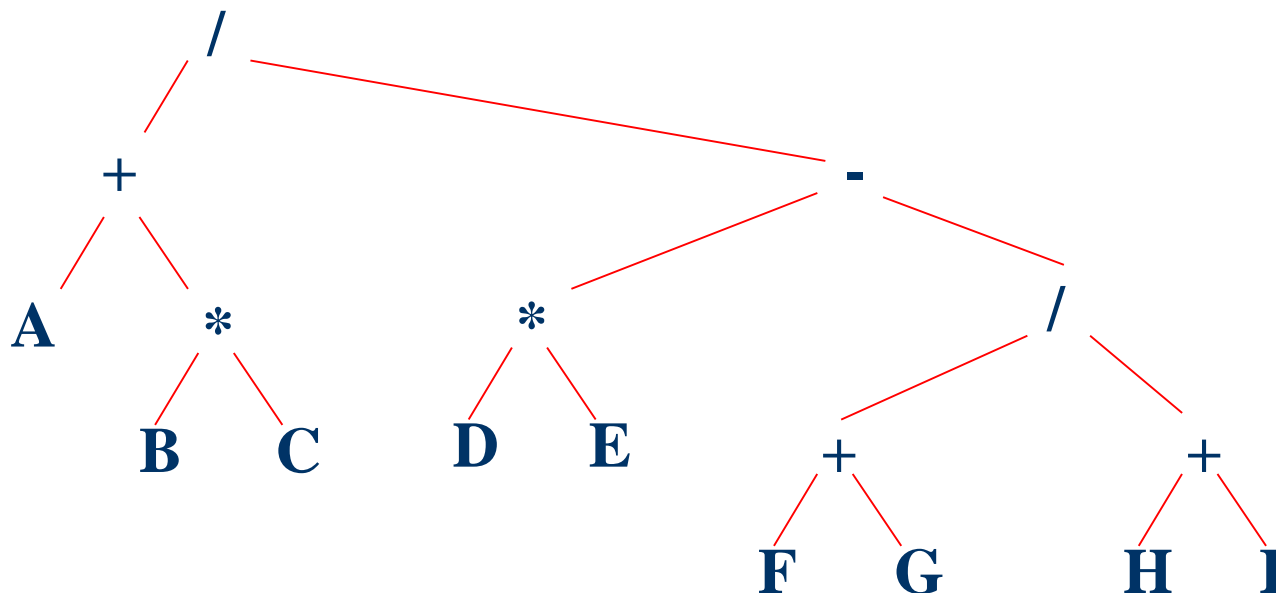
**操作数**出现在叶节点上, **操作符**出现在中间结点

#### DAG图:

Directed Acyclic Graphs 有向无环图

语法树的一种归约表达方式

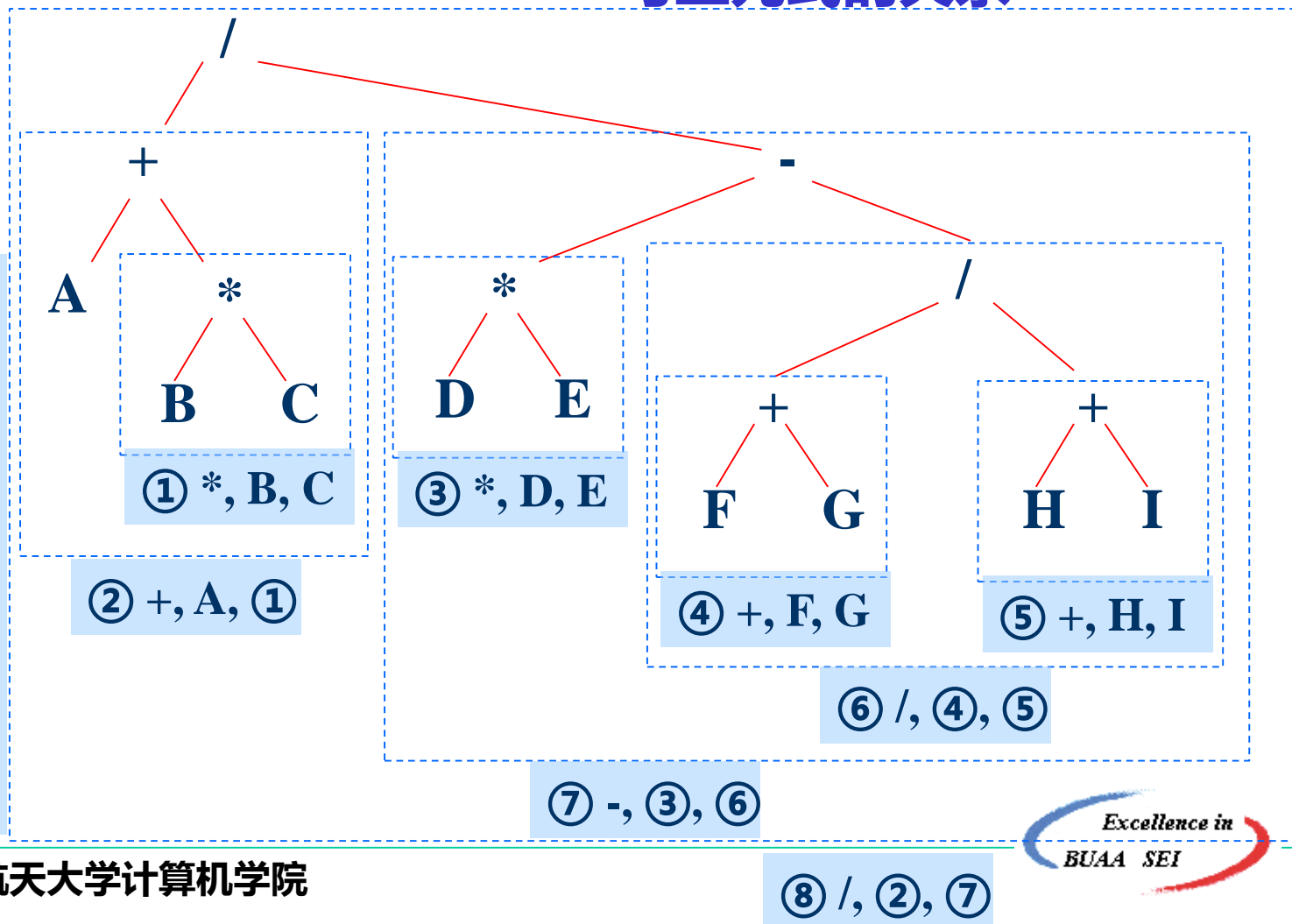
例：  $(A+B * C) / (D * E - (F+G) / (H+I))$



前序遍历（根左右）：  $/+A*BC-*DE/+FG+HI$  （波兰表达）  
 后序遍历（左右根）：  $ABC*+DE*FG+HI+/-/$  （逆波兰表达）  
 中序遍历（左根右）：  $A+B*C/D*E-F+G/H+I$

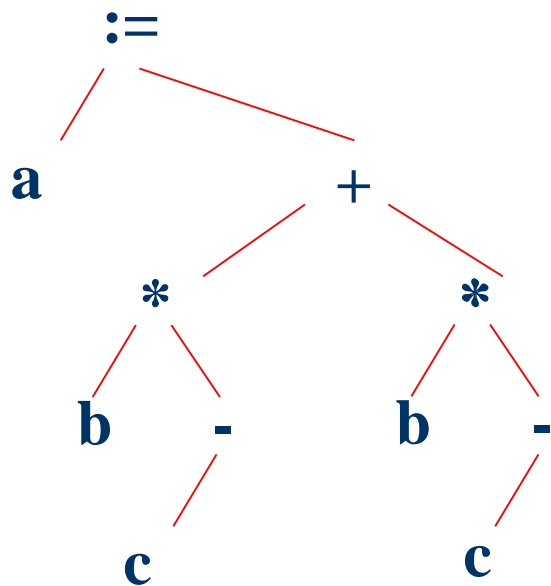
例:  $(A+B * C) / (D * E - (F+G) / (H+I))$

## 与三元式的关系

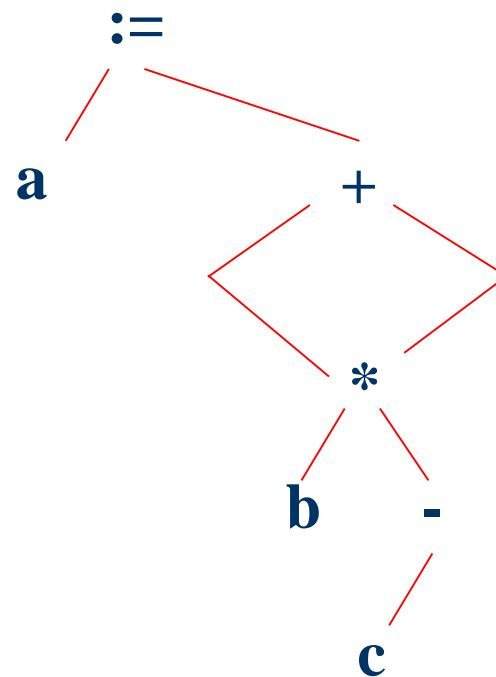




例：赋值语句：  $a := b * (-c) + b * (-c)$

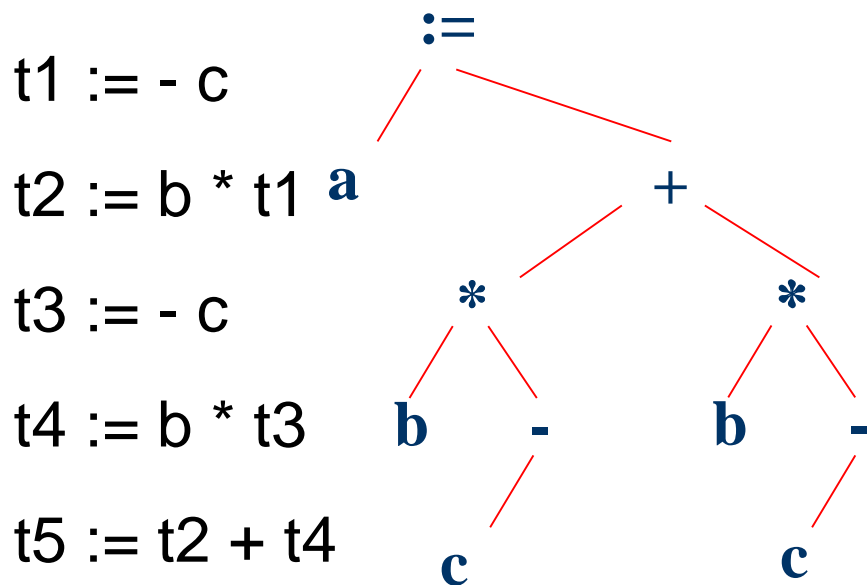


抽象语法树  
(其中有重复部分)

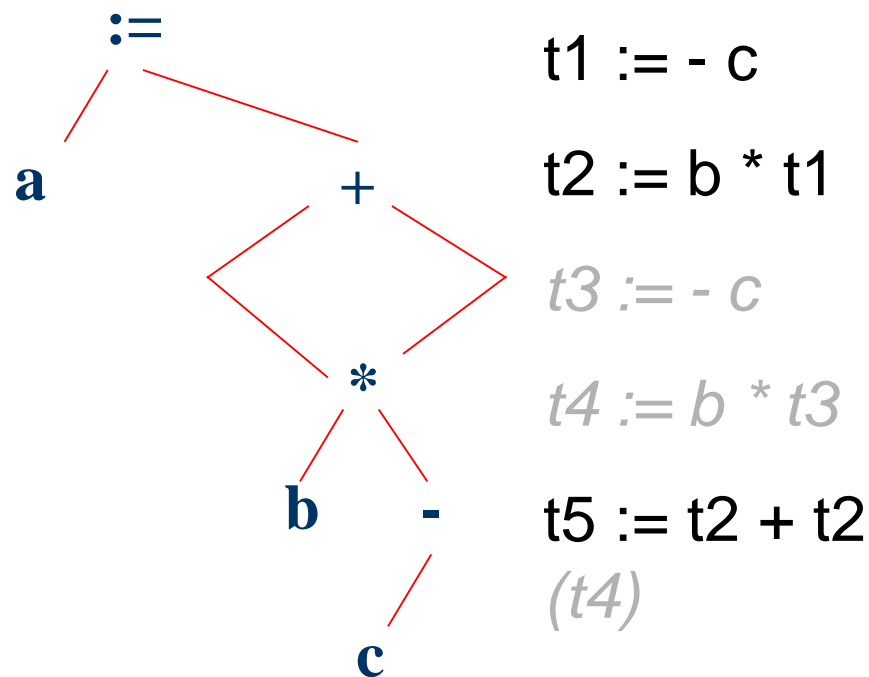


有向无环图 (DAG)

例：赋值语句：  $a := b * (-c) + b * (-c)$  对应的TAC



抽象语法树  
(其中有重复部分)



有向无环图 (DAG)

## 例子：P-Code的抽象机

## 7.3 抽象机代码

许多pascal编译系统生成的中间代码是一种称为P - code的抽象代码，P - code的“P”即“Pseudo”

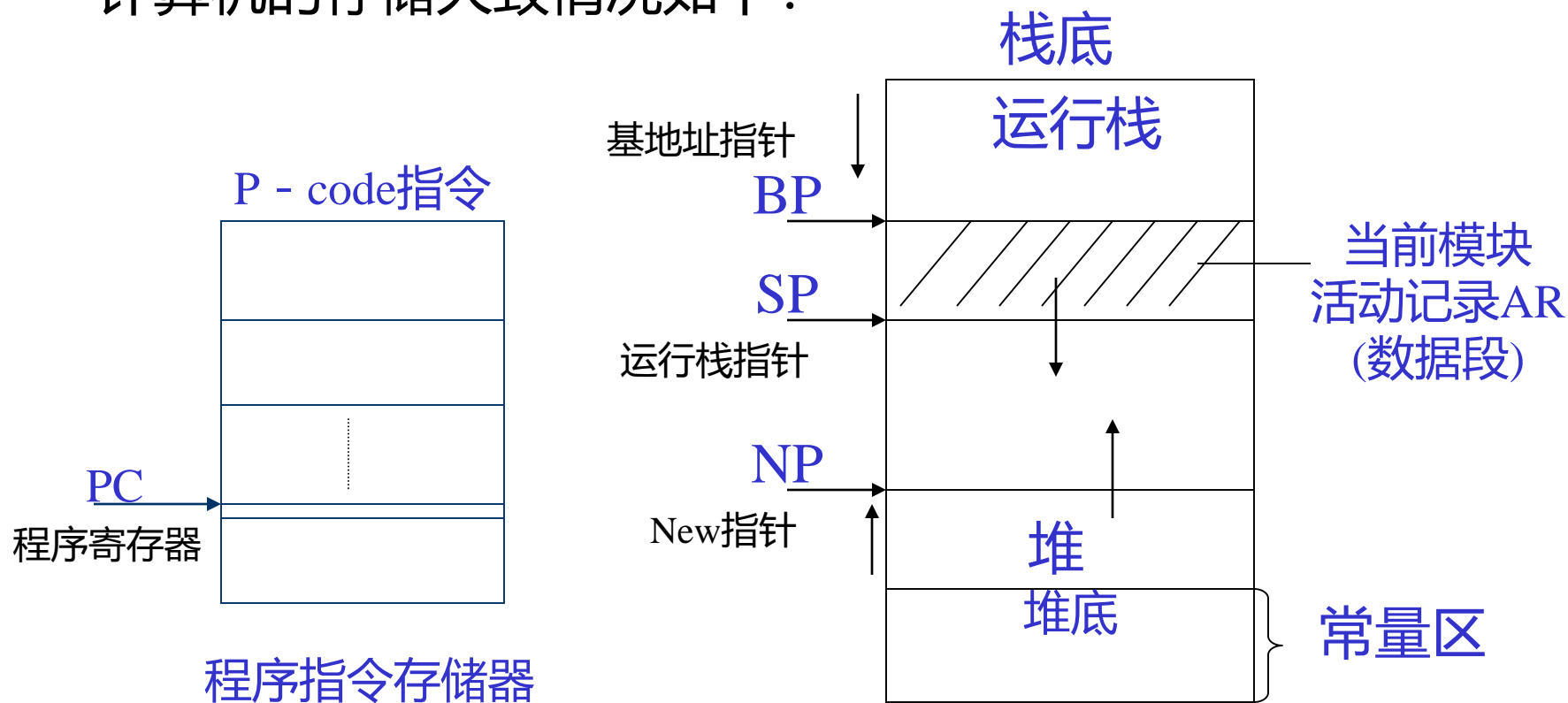
抽象机：

寄存器  
保存程序指令的存储器  
堆栈式数据及操作存储

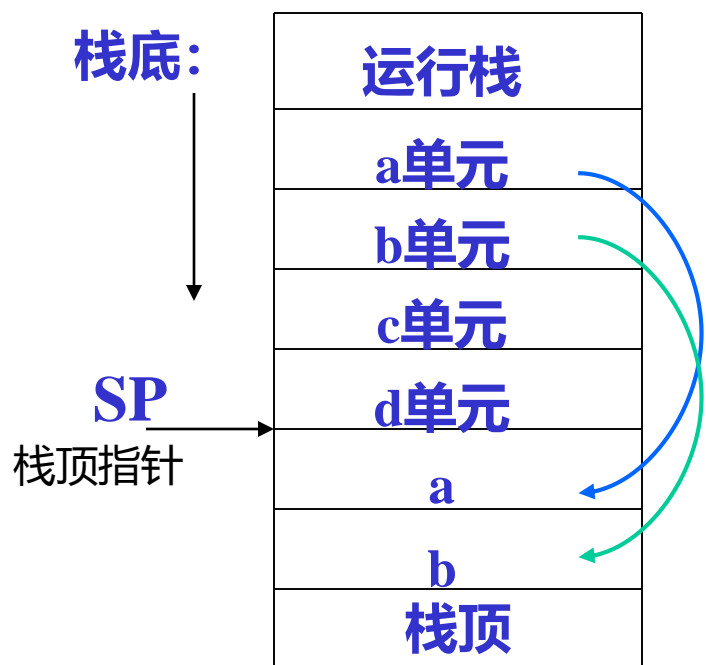
## 寄存器有：

1. PC - 程序计数器
2. NP - New指针，指向“堆”的顶部。“堆”用来存放由New生成的动态数据。
3. SP - 运行栈指针，存放所有可按源程序的数据声明直接寻址的数据。
4. BP - 基地址指针，即指向当前活动记录的起始位置指针。
5. 其他，（如MP - 栈标志指针，EP - 极限栈指针等）

计算机的存储大致情况如下：



运行P - code的抽象机没有专门的运算器或累加器，所有的运算(操作)都在运行栈的栈顶进行，如要进行 $d:=(a+b)*c$ 的运算，生成P - code序列为：



取a	LOD a
取b	LOD b
+	ADD
取c	LOD c
*	MUL
送d	STO d

P - code实际上是波兰表示形式的中间代码

## P-Code指令集 (p184, 10.2)

40 ABI 取整数绝对值

41 ABR 取实数绝对值

28 ADI 整数加

29 ADR 实数加

53 DVI 整数除

54 DVR 实数除

42 NOT 布尔“非”

43 AND 布尔“与”

26 CHK Q 检查越界

15 CSP Q 调用标准过程

12 CUP P, Q 调用用户过程

19 GEQ P, Q 大于等于

20 GRT P, Q 大于

48 INN 判定集合成员

48 INT 取交集

4 LDA P, Q 加载地址

7 LDC P, Q 加载常量

23 UJP Q 无条件跳转

24 FJP Q 为假时条件跳转



## P-Code指令集

40 ABI 取整数绝对值

41 ABR 取实数绝对值

28 ADI 整数加

29 ADR 实数加

53 DVI 整数除

54 DVR 实数除

42 NOT 布尔“非”

43 AND 布尔“与”

26 CHK Q 检查越界

15 CSP Q 调用标准过程

12 CUP P, Q 调用用户过程

19 GEQ P, Q 大于等于

20 GRT P, Q 大于

48 INN 判定集合成员

48 INT 取交集

4 LDA P, Q 加载地址

7 LDC P, Q 加载常量

23 UJP Q 无条件跳转

24 FJP O 为假时条件跳转

```
int a;
```

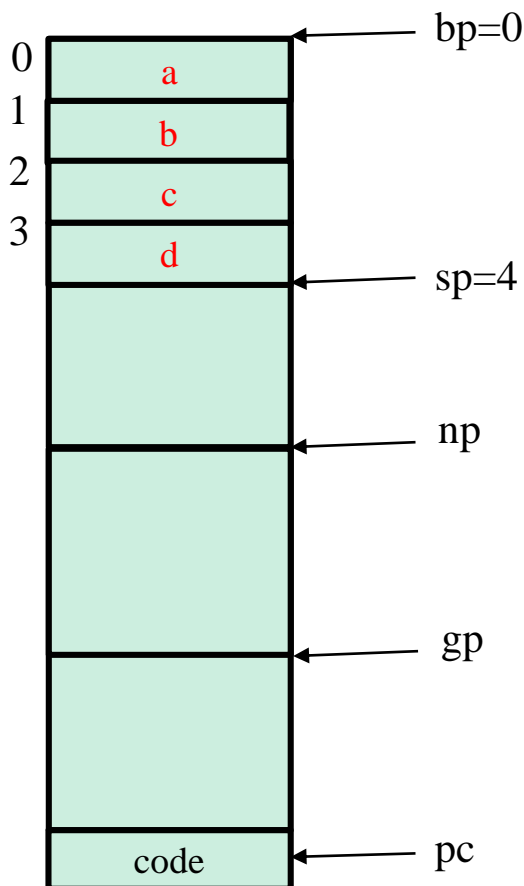
```
int b;
```

```
int c;
```

```
int d;
```

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

```
b = a * a + b * b;
```



LDA 1

LDA 2

ADI

LDA 3

ADI

STO 0

LDA 0

LDA 0

MULT

LDA 1

LDA 1

MULT

ADI

STO 1

## P-Code指令集

40 ABI 取整数绝对值

41 ABR 取实数绝对值

28 ADI 整数加

29 ADR 实数加

53 DVI 整数除

54 DVR 实数除

42 NOT 布尔“非”

43 AND 布尔“与”

26 CHK Q 检查越界

15 CSP Q 调用标准过程

12 CUP P, Q 调用用户过程

19 GEQ P, Q 大于等于

20 GRT P, Q 大于

48 INN 判定集合成员

48 INT 取交集

4 LDA P, Q 加载地址

7 LDC P, Q 加载常量

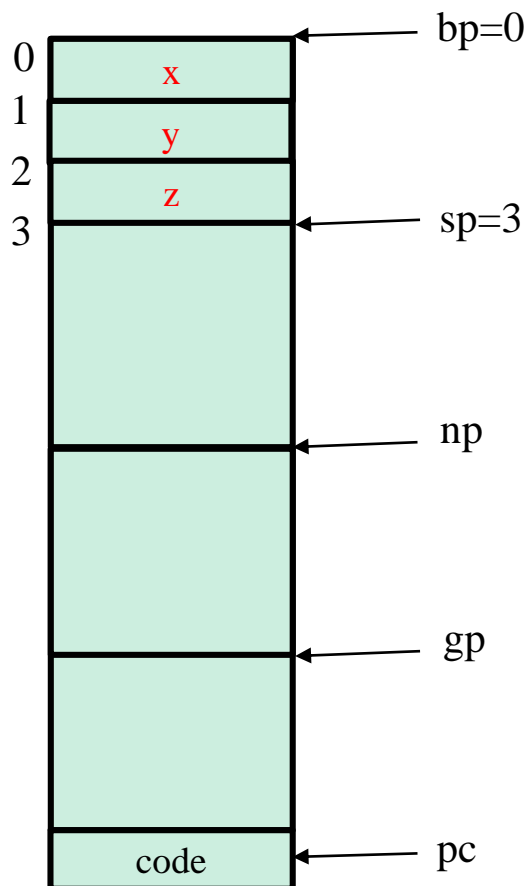
23 UJP Q 无条件跳转

24 FJP Q 为假时条件跳转

```
int x;
int y;
int z;
```

```
if (x < y)
    z = x;
else
    z = y;
```

```
z = z * z;
```



LDA 0

LDA 1

LES

BRF :lable1

LDA 0

STO 2

BR :label2

label1: LDA 1

STO 2

label2: LDA 2

LDA 2

MULT

STO 2

## P-Code指令集

40 ABI 取整数绝对值

41 ABR 取实数绝对值

28 ADI 整数加

29 ADR 实数加

53 DVI 整数除

54 DVR 实数除

42 NOT 布尔“非”

43 AND 布尔“与”

26 CHK Q 检查越界

15 CSP Q 调用标准过程

12 CUP P, Q 调用用户过程

19 GEQ P, Q 大于等于

20 GRT P, Q 大于

48 INN 判定集合成员

48 INT 取交集

4 LDA P, Q 加载地址

7 LDC P, Q 加载常量

23 UJP Q 无条件跳转

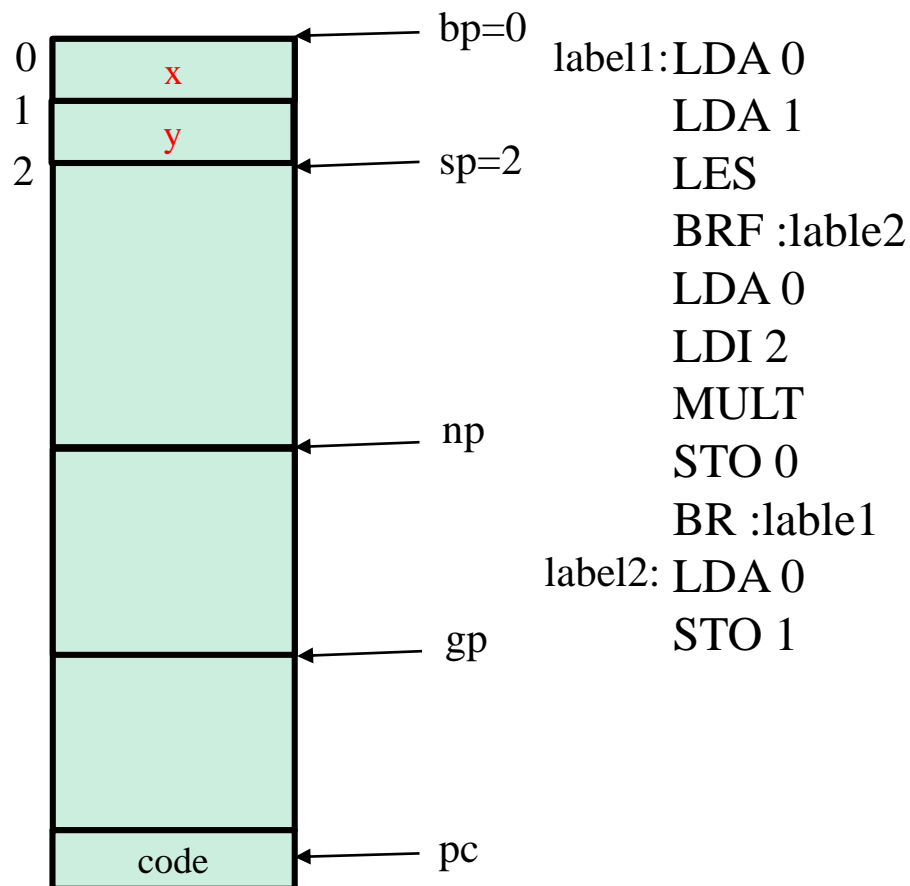
24 FJP Q 为假时条件跳转

```
int x;
```

```
int y;
```

```
while (x < y) {
    x = x * 2;
}
```

```
y = x;
```



## P-Code指令集

40 ABI 取整数绝对值

41 ABR 取实数绝对值

28 ADI 整数加

29 ADR 实数加

53 DVI 整数除

54 DVR 实数除

42 NOT 布尔“非”

43 AND 布尔“与”

26 CHK Q 检查越界

15 CSP Q 调用标准过程

12 CUP P, Q 调用用户过程

19 GEQ P, Q 大于等于

20 GRT P, Q 大于

48 INN 判定集合成员

48 INT 取交集

4 LDA P, Q 加载地址

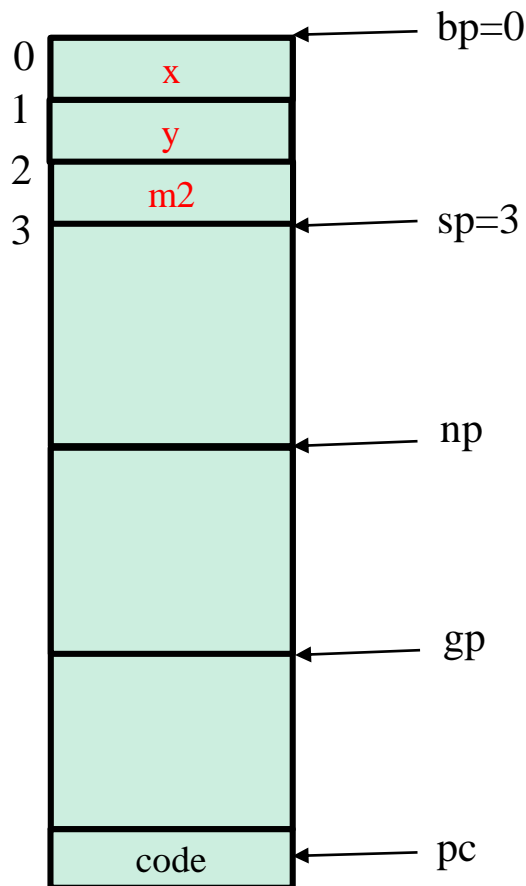
7 LDC P, Q 加载常量

23 UJP Q 无条件跳转

24 FJP Q 为假时条件跳转

```
void main() {
    int x, y;
    int m2 = x * x + y * y;

    while (m2 > 5) {
        m2 = m2 - x;
    }
}
```



```
LDA 0
LDA 0
MULT
LDA 1
LDA 1
MULT
ADI
STO 2
label1: LDA 2
LDI 5
GT
BRF :label2
LDA 2
LDA 0
SUB
STO 2
BR :label1
label2: ...
```

## P-Code指令集

40 ABI 取整数绝对值

41 ABR 取实数绝对值

28 ADI 整数加

29 ADR 实数加

53 DVI 整数除

54 DVR 实数除

42 NOT 布尔“非”

43 AND 布尔“与”

26 CHK Q 检查越界

15 CSP Q 调用标准过程

12 CUP P, Q 调用用户过程

19 GEQ P, Q 大于等于

20 GRT P, Q 大于

48 INN 判定集合成员

48 INT 取交集

4 LDA P, Q 加载地址

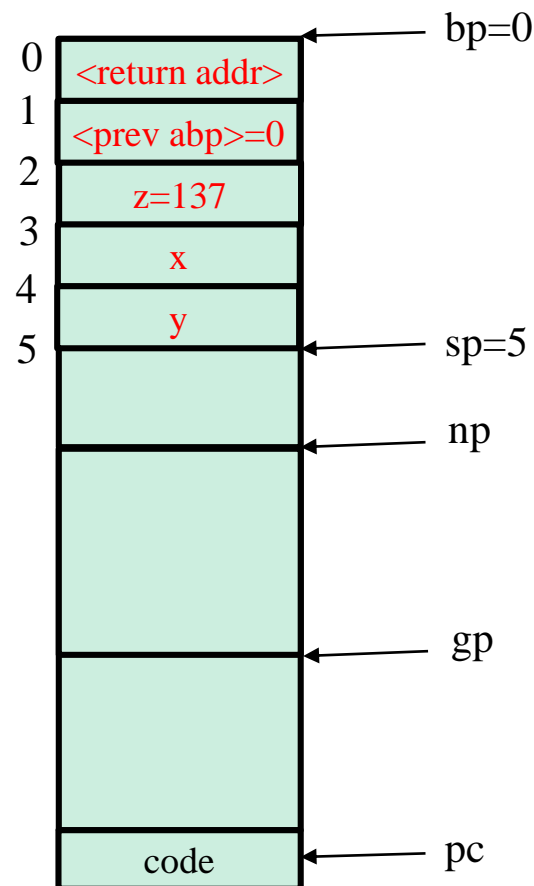
7 LDC P, Q 加载常量

23 UJP Q 无条件跳转

24 FJP Q 为假时条件跳转

```
void SimpleFn(int z) {
    int x, y;
    x = x * y * z;
}
```

```
void main() {
    SimpleFunction(137);
}
```



label1: ALLOCATE 3

LDA 3

LDA 4

MULT

LDA 2

MULT

STO 3

LDA 1

STO bp

BR :label2

...

LDI :label2

LDI 0

LDI 137

BR :label1

label2: ...

编译程序生成P - code指令程序后，我们可以用一个解释执行程序（interpreter）来解释执行P - code，当然也可以把P - code再变成某一机器的目标代码。

显然，生成抽象机P - code的编译程序是很容易移植的。

## 错误处理

## 第八章 错误处理

- 概述
- 错误分类
- 错误的诊察和报告
- 错误处理技术



## 8.1 概述

### 1. 必备功能之一

正确的源程序：通过编译生成目标代码

错误的源程序：通过编译发现并指出错误

### 2. 错误处理能力

- (1) 诊察错误的能力
- (2) 报错及时准确
- (3) 一次编译找出错误的多少
- (4) 错误的改正能力
- (5) 遏止重复的错误信息的能力


## 8.2 错误分类

从编译角度，将错误分为两类：**语法错误**和**语义错误**

**语法错误**：源程序在语法上不合乎文法

如：

A[ I, J := B +\* C



**语义错误**主要包括：**程序不符合语义规则**或  
**超越具体计算机系统的限制**

## 语义规则：

- 标识符先说明后引用
- 标识符引用要符合作用域规定
- 过程调用时实参与形参要一致
- 参与运算的操作数类型一致
- 下标变量下标不能越界

## 超越系统限制：

- 数据溢出错误
- 符号表、静态存储分配数据区溢出
- 动态存储分配数据区溢出

## 8.3 错误的诊察和报告

### 错误诊察:

1. 违反语法和语义规则以及超过编译系统限制的错误。

**编译程序:** 语法和语义分析时  
(语义分析要借助符号表)

2. 下标越界, 计算结果溢出以及动态存储数据区溢出。

**目标程序:** 目标程序运行时

对此, 编译程序要生成相应的目标程序作检查  
和进行处理

## 错误报告:

### 1. 出错位置: 即源程序中出现错误的位置

实现: 行号计数器      `line_no`

单词序号计数器   `char_no`

一旦诊察出错误, 当时的计数器内容就是出错位置

### 2. 出错性质:

可直接显示文字信息

可给出错误编码

## 3. 报告错误的两种方式:

### (1) 分析完以后再报告(显示或者打印)

编译程序可设一个保存错误信息的数据区(可用记录型数组), 将语法语义分析所诊察到的错误送数据区保存, 待源程序分析完以后, 显示或打印错误信息。

例:  $A[x, y := B + *C$



源程序行号

错误序号

错误性质

X X

6

缺少 "]"

X X

10

表达式语法错误

## (2) 边分析边报告

可以在分析一行源程序时若发现有错，立即输出该行源程序，并在其下输出错误信息。

Line - no    A[x , y                    := B+ \*C

↑  
缺 “[ ]”

↑  
n

↑  
表达式语法错

↑  
m

错误编号

错误编号

有时候报错不一定十分准确  
(位置和性质)，需进一步分析

例

begin

.....

i := 1 step 1    until n do

.....

end

## 8.4 错误处理技术

发现错误后，在报告错误的同时还要对错误进行处理，以方便编译能进行下去。目前有两种处理办法：

1. **错误改正：**指编译诊察出错误以后，根据文法进行错误改正。

如：  $A[i, j] := B + *C$

要正确地改正错误  
是很困难的

但不是总能做到,如  $A := B - C * D + E$

2. **错误局部化处理：**指当编译程序发现错误后，尽可能把错误的影响限制在一个局部的范围，避免错误扩散和影响程序其他部分的分析。



## (1) 一般原则

**当诊察到错误以后，就暂停对后面符号的分析，跳过错误所在的语法成分然后继续往下分析。**

**词法分析：发现不合法字符，显示错误，并跳过该标识符(单词)继续往下分析。**

**语法语义分析：跳过所在的语法成分(短语或语句)，一般是跳到语句右界符，然后从新语句继续往下分析。**

## (2) 错误局部化处理的实现（递归下降分析法）

**CX: 全局变量，存放错误信息。**

- 用递归下降分析时，如果发现错误，便将有关错误信息（字符串或者编号）送CX，然后转出错误处理程序；

- 出错程序先打印或显示出错位置以及出错信息，然后跳出一段源程序，直到跳到语句的右界符（如：end）或正在分析的语法成分的合法后继符号为止，然后再往下分析。

有如下分析程序:

```
procedure if_stmt;
begin
  nextsym;                                /*读下个单词符号*/
  B;                                       /*调用布尔表达式处理程序*/
  if not class='then' then
  begin
    cx := '缺then'                        /*错误性质送cx*/
    error;                                /*调用出错处理程序*/
  end;
else
  begin
    nextsym;
    statement
  end;
  if class='else' then
  begin
    nextsym;
    statement;
  end
end if_stmt;
```

## 局部化处理的出错程序为:

```
procedure error;  
  begin  
    write(源程序行号, 序号, cx)  
    repeat  
      nextsym;  
    until class = ';' or class = 'end' or class = 'else'  
  end error;
```



real x, 3a, a, bcd, 2fg;

## (3) 提高错误局部化程度的方法

设  $S_1$ : 合法后继符号集 (某语法成分的后继符号)

$S_2$ : 停止符号集 (跳读必须停止的符号集)

进入某语法成分的分析程序时:

$S_1 :=$  合法后继符号

$S_2 :=$  停止符号

当发现错误时:  $\text{error}(S_1, S_2)$

```
procedure error(S1,S2)
begin
    write(line_no, char_no, cx);
    repeat
        nextsym
    until(class in S1 or class in S2 );
end
```

if<B> then <stmt>[else< stmt >]

若<B>有错,则可跳到then,

若<stmt>有错,则可跳到else。

## 3.目标程序运行时错误检测与处理.

下标变量下标值越界

计算结果溢出

动态存储分配数据区溢出

- 在编译时生成检测该类错误的代码。

对于这类错误,要正确的报告出错误位置很难,因为目标程序与源程序之间难以建立位置上的对应关系

一般处理办法:

当目标程序运行检测到这类错误时,就调用异常处理程序,打印错误信息和运行现场(寄存器和存储器中的值)等, 然后停止程序运行。

## 小结：本节+上一节：为代码生成做“需求分析”

- 数据结构在内存里是如何表示的
  - 函数在内存中是如何表示的
  - 他们在内存中如何放置，如何管理
  - 中间代码如何表示
- 让编译器聪明一些：错误处理！

**作业： p144 1,2,3,4**



## 编译实验：错误处理

## 目标和要求:

- 针对**常见的错误分类**编写错误处理程序，考核学生对错误处理方法的掌握情况，培养学生编写错误处理程序的能力
- 学生需分析编译过程的每个阶段可能出现的错误，学完“第8章 错误处理”后，对各阶段的错误进行**错误局部化处理**，并进行补充和完善，按要求输出错误信息
- 作业提交至教学平台，用5个测试程序进行测试，并进行相似性检查。根据输出结果与预期结果一致部分所占的比例给分。

## 评测:

- 自动评测
- 错误类别码按下表中的定义输出，**行号从1开始计数**
- 约定1：输入的被编译源文件统一命名为testfile.txt
- 约定2：错误信息输出的结果文件统一命名为error.txt
- 约定3：结果文件中每行按如下方式组织

**错误所在的行号 错误的类别码**

(行号与类别码之间只有一个空格)

(类别码严格按照表格中的小写英文字母)

## 错误类型及编码：

错误类型	错误类别码
非法符号或不符合词法	a
名字重定义	b
未定义的名字	c
函数参数个数不匹配	d
函数参数类型不匹配	e
条件判断中出现不合法的类型	f
无返回值的函数存在不匹配的return语句	g
有返回值的函数缺少return语句或存在不匹配的return语句	h
数组元素的下标只能是整型表达式	i
不能改变常量的值	j
应为分号	k
应为右小括号')'	l
应为右中括号']'	m
do-while语句中缺少while	n
常量定义中=后面只能是整型或字符型常量	o

## 测试样例:

```
1  const int const1 = 1, const2 = -100;
2  const char const3 = '?';
3  int change1;
4  char change3;
5  int gets1(int var1,int var2){
6      change1 = var1 + var2;
7      return (change1);
8  }
9  void main(){
10     change1 = 10;
11     printf("Hello World");
12     printf(gets1(10, 20));
13 }
```

2 a  
6 k

## 注意事项:

- 本次作业的每个测试程序各包含1-3个错误，均来自前述表格；按报出错误占应报错误的比例得分
- 表中未包含的错误，需自行设计，本作业考核不涉及
- 完成本次作业时，不需要输出词法分析和语法分析作业要求输出的内容
- 本次考核之外，发现错误时最好直接输出描述信息，而不是仅给出错误类别码
- 输出到 error.txt 中的错误信息，每行末尾都需要加一个换行符 \n

谢谢!