

第七章 源程序的中间形式

- 波兰表示
- N-元表示
- 抽象机代码

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

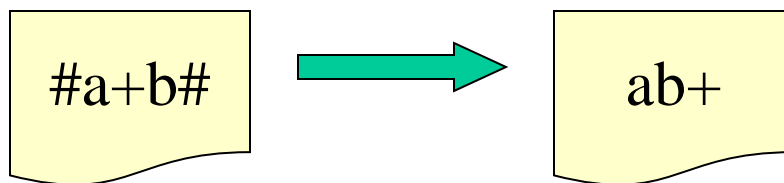
波兰表示

算术表达式: $F * 3.1416 * R * (H + R)$

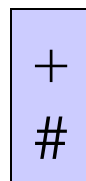
转换成波兰表示: $F3.1416 * R * HR + *$

赋值语句: $A := F * 3.1416 * R * (H + R)$

波兰表示: $AF3.1416 * R * HR + * :=$



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

算术表达式:

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

操作符栈

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 $\langle \text{expr} \rangle$ then $\langle \text{stmt}_1 \rangle$ else $\langle \text{stmt}_2 \rangle$



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

BZ: 二目操作符

若 $\langle \text{expr} \rangle$ 的计算结果为 0 (false),
则产生一个到 $\langle \text{label}_1 \rangle$ 的转移

BR: 一目操作符

产生一个到 $\langle \text{label}_2 \rangle$ 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，
使用波兰表示优化不是十分方便。

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： $w * x + (y + z)$

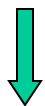


- (1) *, w, x
- (2) +, y, z
- (3) +, (1), (2)

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 ≤ 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:

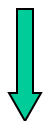
操作	三元式
1. (1)	(1) $*$, C, D
2. (2)	(2) $/$, (1), E
3. (3)	(3) $+$, B, (2)
4. (4)	(4) $:=$, A, (3)
5. (1)	(5) $:=$, F, (1)
6. (5)	

四元式表示

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

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

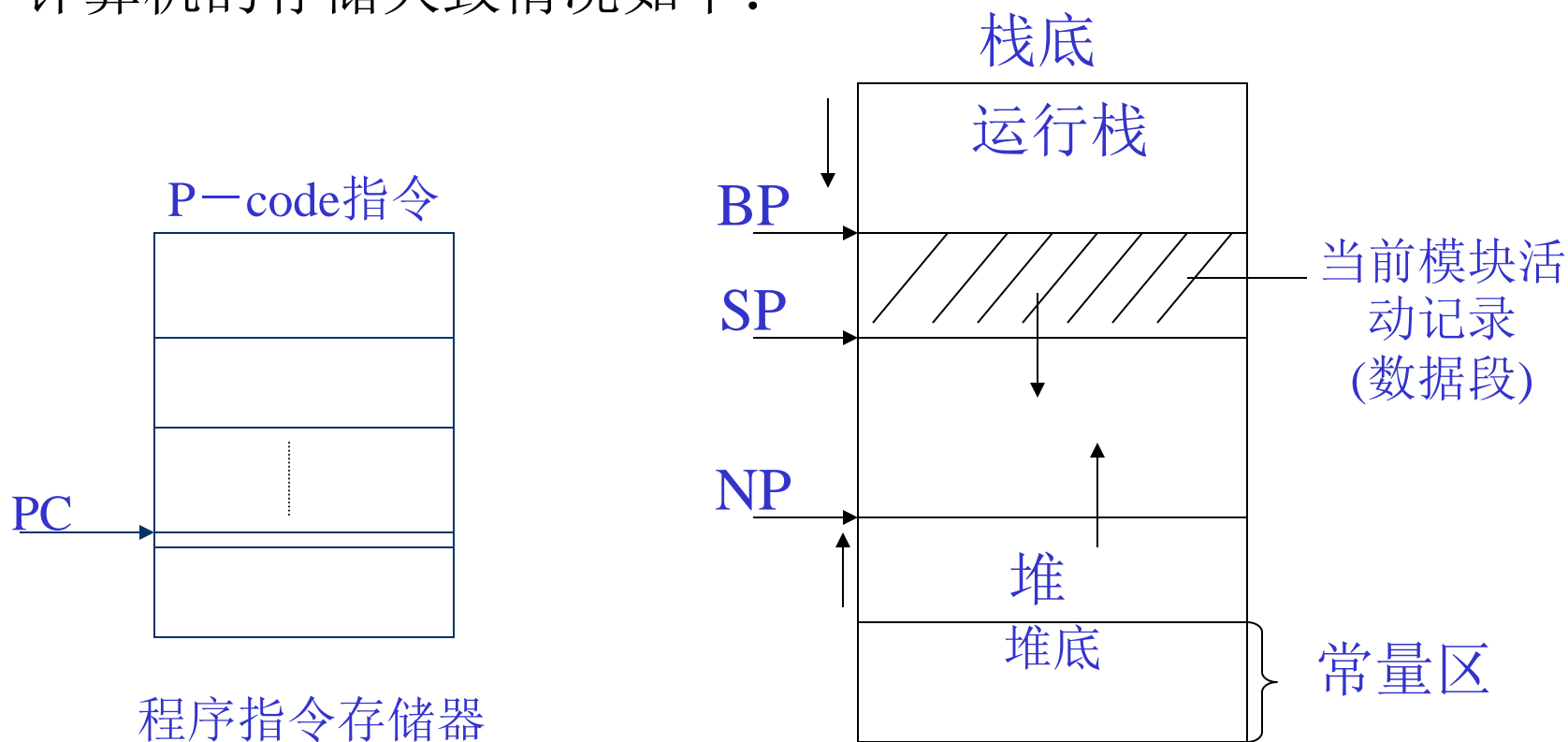
保存程序指令的存储器

堆栈式数据及操作存储

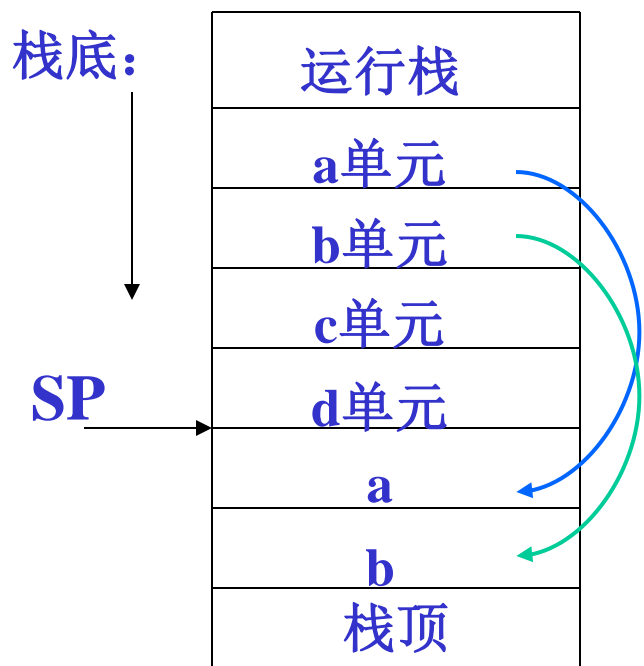
寄存器有：

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

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。

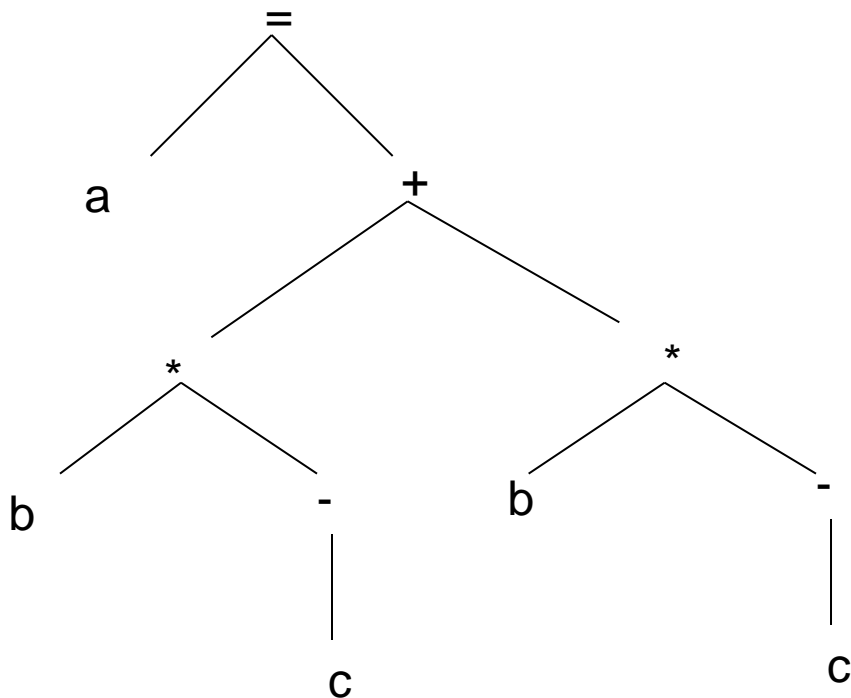
显然，生成抽象机P-code的编译程序是很容易移植的。

作业： P144 1,2,3,4

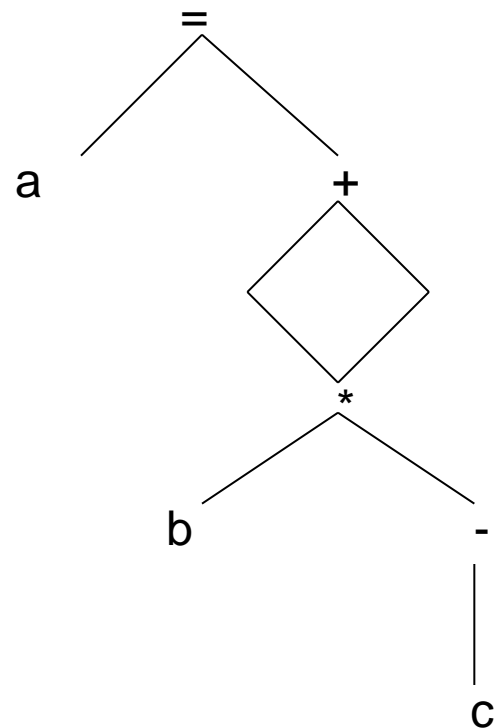
中间代码的图表示

- 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

- 赋值语句: $a := b * (-c) + b * (-c)$



语法树

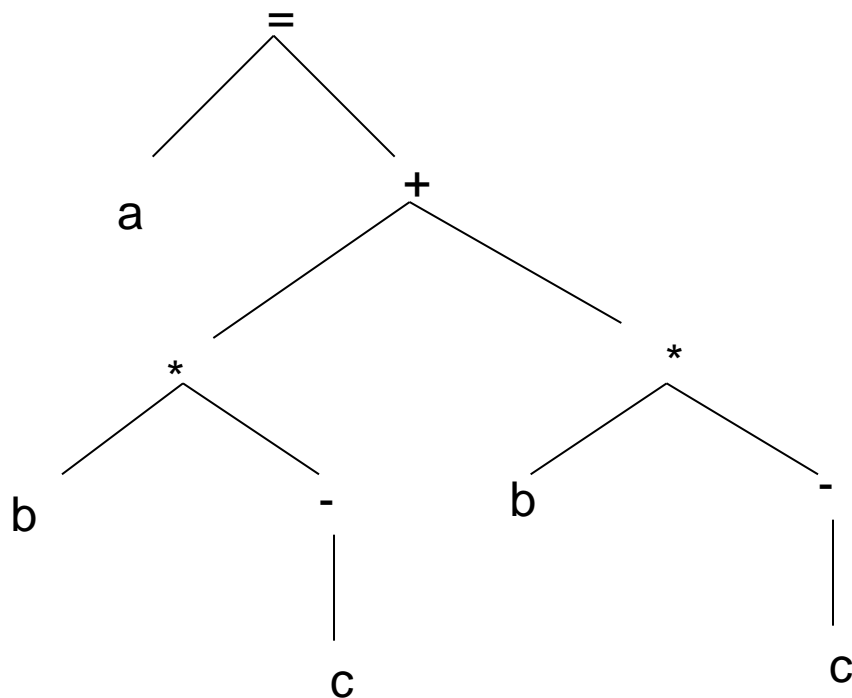


DAG图

中间代码：三地址码

- 适合目标代码生成和优化的一种表达形式
- 三地址码是语法树或者DAG图的线性表示
- 树的中间结点由临时变量表示

三地址码与语法树的对应关系



语法树

$t1 := -c$

$t2 := b * t1$

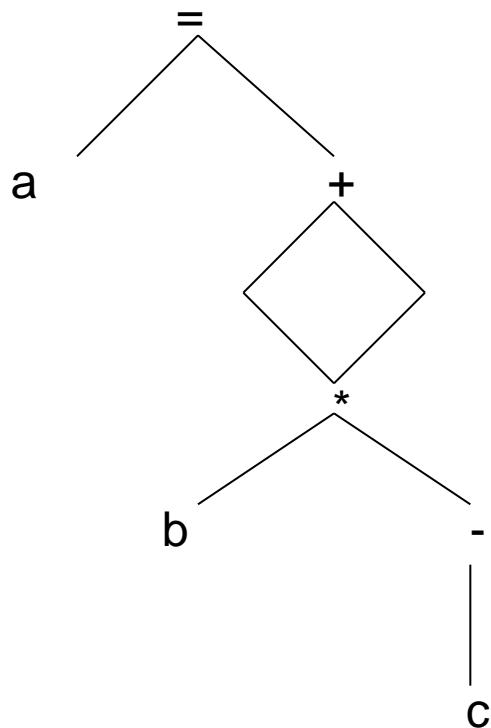
$t3 := -c$

$t4 := b * t3$

$t5 := t2 + t4$

$a := t5$

三地址码与DAG图的对应关系



DAG图

$t1 := -c$

$t2 := b * t1$

$t3 := -c$

$t4 := b * t3$

$t5 := t2 + t2 (t4)$

$a := t5$