3 栈和队列

董洪伟

http://hwdong.com

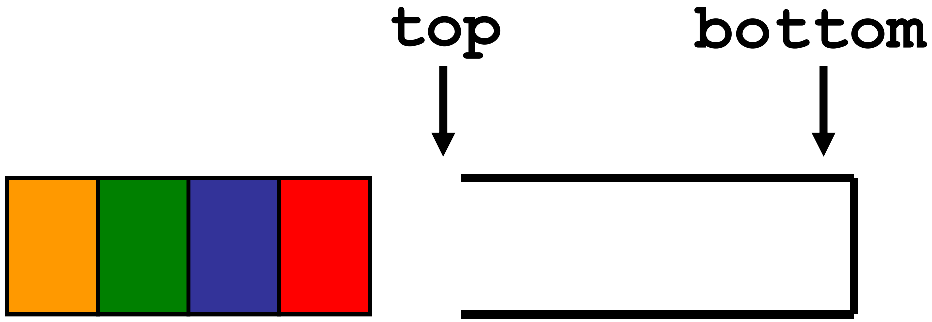
主要内容

- 栈的类型定义
- 栈的表示
  - 顺序表示
  - 链表表示
- 栈的应用
  - 括号匹配
  - 走迷宫
  - 表达式计算
- 栈和递归

- 队列的类型定义
- 队列的表示
  - 链表表示
  - 顺序表示：循环队列
- 队列的应用
  - 农夫过河问题

栈的类型定义

- 定义
  - 只允许在同一端删除、同一端插入的线性表
  - 允许插入、删除的一端叫做栈顶(**top**)，另一端叫做栈底(**bottom**)
- 特性
  - 先进后出(**FILO, First In Last Out**)



栈的类型定义

数据对象：具有线形关系的一组数据

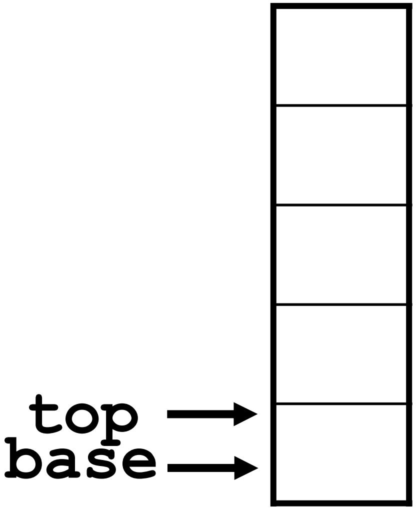
操作：

```
bool Push(Stack &S, ElemType e); //入栈
bool Pop(Stack &S, ElemType &e);  //出栈
bool Top(Stack S, ElemType &e); //取栈顶
bool IsEmpty(Stack S); //空吗？
bool Clear(Stack &S);    //清空
```

栈的表示：顺序表示

- 顺序表示：即用数组来实现
  - 用数组存放堆栈中的数据
  - 再施加**LIFO**的访问限制：插入、删除只能从一端进行

```
typedef struct{
    ElemType *base;
    ElemType *top;
    int         stacksize;
} SqStack;
```



栈的表示：顺序表示

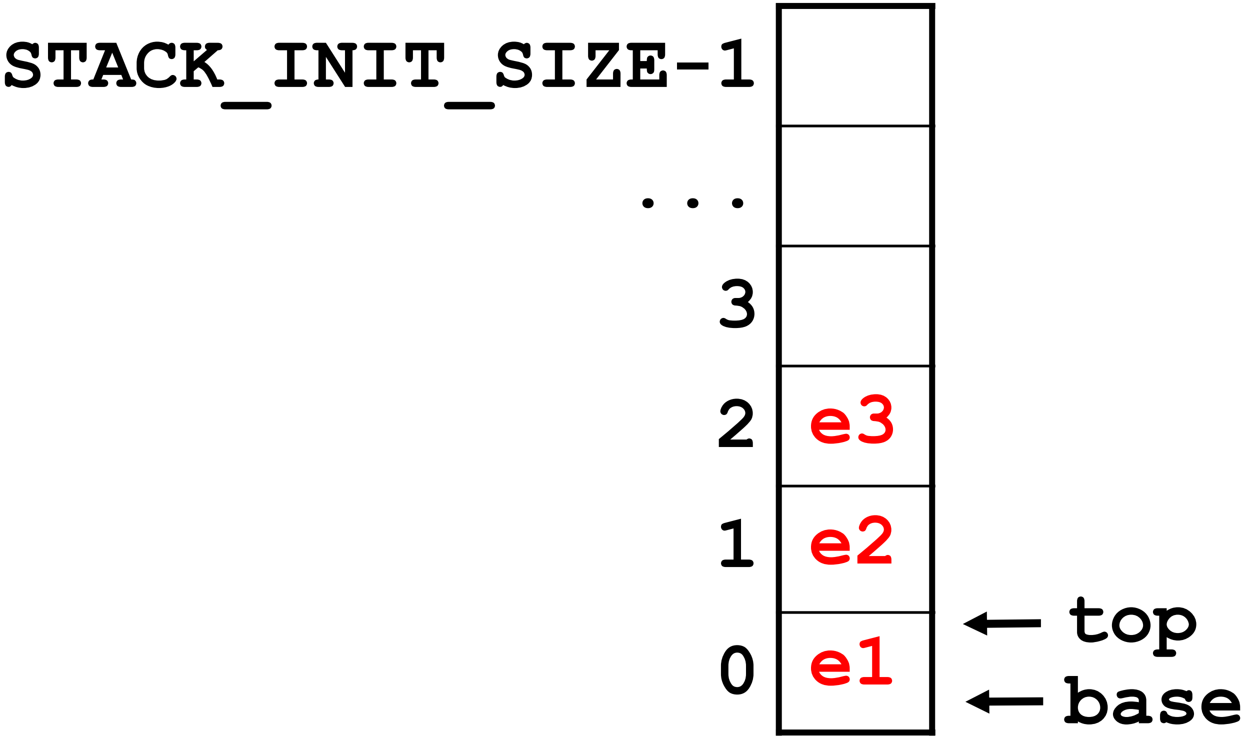
- 初始化

```
bool InitStack(SqStack &S){
   //分配空间
   S.base = (ElemType*) malloc
        (STACK_INIT_SIZE * sizeof(ElemType));
   if(!S.base)  return false;
   S.top       = S.base;             //设置指针
   S.stacksize = STACK_INIT_SIZE;   //设置大小
   return true;
}
```

- 插入

```
bool  Push(SqStack &S, ElemType e){
      //若空间不够，重新分配
      if(S.top - S.base >= S.stacksize)  {
         ElemType* p = (ElemType *)realloc
                 (S.base, (S.stacksize +
                          STACKINCREMENT)*
                          sizeof(ElemType));
         if(!p) return false;
         free(S.base);   S.base = p;
         S.top = S.base + S.stacksize;
         S.stacksize += STACKINCREMENT;
      }
      *S.top ++ = e; //插入数据
      return true;
}
```

```
int Push(SqStack &S, ElemType e){
     if(S.top - s.base >= S.stacksize)
     {    ...   }     //重新分配空间（略）
     *S.top ++ = e; return true;
}
```

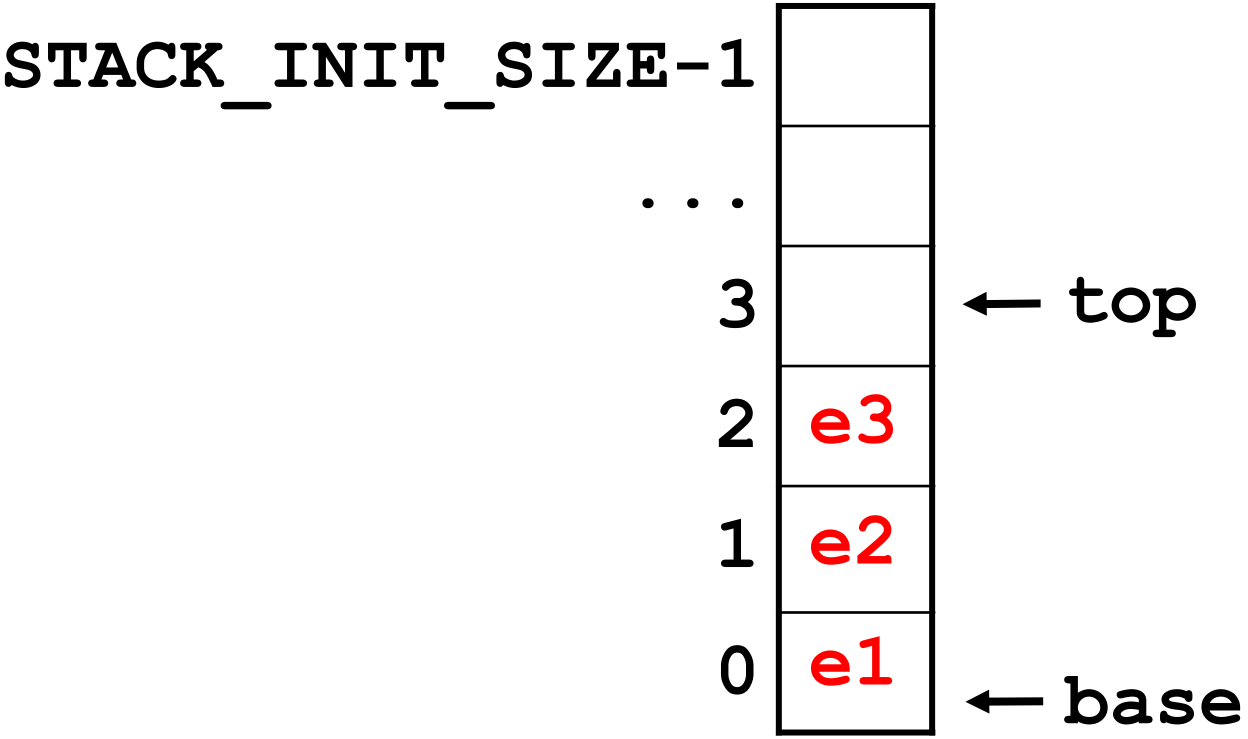


栈的表示：顺序表示

- 删除

```
bool Pop(SqStack &S, ElemType &e)
{
    if(S.top == S.base)  //空栈
        return false;
    e = * --S.top;        //出栈
    return true;
}
```

```
bool Pop(SqStack &S, ElemType &e){
     if(S.top == S.base)   //空栈
          return false;
     e = * --S.top;        //出栈
     return true;
}
```



栈的表示：链式表示

- 链式表示
  - 使用链表来实现
  - 栈不就是线性表 + **LIFO**限制么？
  - 参照线性表的链式表示

栈的应用

- 栈的应用
  - 颠倒元素顺序
    - 数制转换
  - 记录“历史信息”
    - 括号匹配的检验
    - 行编辑程序
    - 走迷宫
    - 表达式计算

栈的应用：数制转换

$$N = a_k d^k + a_{k-1} d^{k-1} + \ldots + a_1 d^1 + a_0$$

十进制数**N**转换为**d**进制数的转换,原理:

$$N = (N \text{ div } d) \times d + N \text{ mod } d$$

商　　　　　　余数

(其中：**div** 为整除运算，**mod** 为求余运算)

例如： $(1348)_{10} = (2504)_8$ ，其运算过程如下：

N	N div 8	N mod 8
1348	168	4 (个位)
168	21	0 (十位)
21	2	5 (百位)
2	0	2 (千位)

栈的应用：数制转换

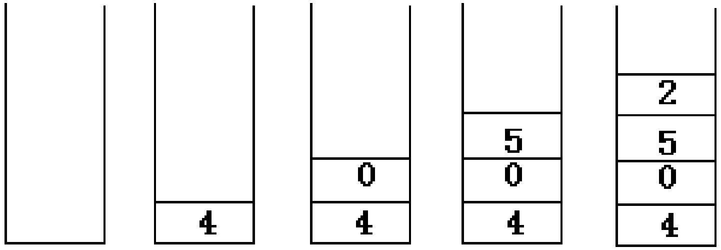
输入：任意一个非负十进制整数

输出：与其等值的八进制数。

由于输出每位数字(2、5、0、4)与得到每位数字（4、0、5、2）的次序正好相反,因此,可以用栈保持依次得到的各位(个、十、百、...)。

栈的应用：数制转换

```
void conversion () {
// 输入非负十进制整数，输出对应的八进制数
   SqStack  S; int N, int e;
   InitStack(S); // 构造空栈
   scanf ("%d",&N);
   while (N) {
       Push(S, N % 8);   N = N/8;
   }
   while (!StackEmpty(S)) {
       Pop(S,e);
       printf ( "%d\n", e );
   }
} // conversion
```



入栈过程

栈的应用：括号匹配检测

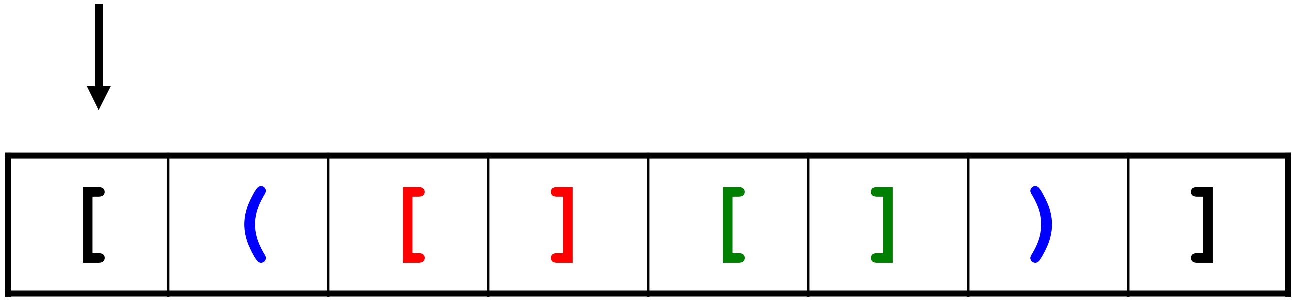
- 问题
  - 括号、引号等符号是成对出现的，必须相互匹配
  - 设计一个算法，自动检测输入的字符串中的括号是否匹配
  - 比如：
    - { } [([ ] [ ])] 匹配
    - [( ])，(( )] 都不匹配

栈的应用：括号匹配检测

- 括号的匹配规则
  - 从里向外开始
  - 左括号应当和最近的右括号匹配
  - [([ ] [ ])]

栈的应用：括号匹配检测

- 思考
  - 从左向右扫描字符串

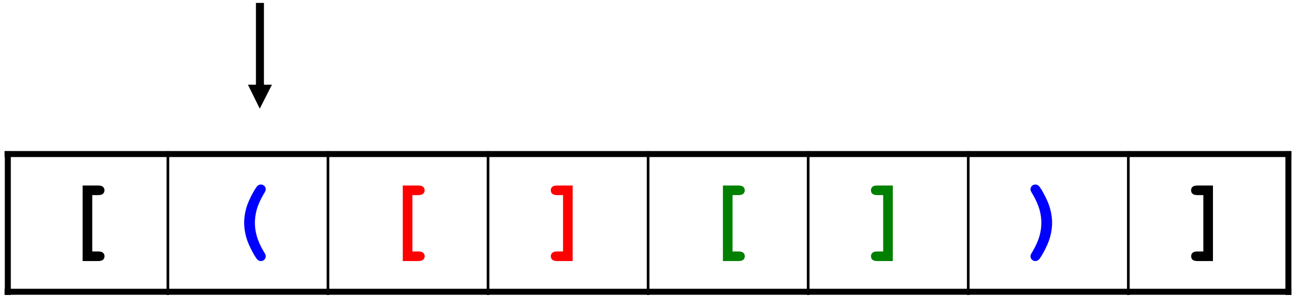


[	(	[	]	[	]	)	]

当前是[，期待一个]

栈的应用：括号匹配检测

- 思考
  - 从左向右扫描字符串

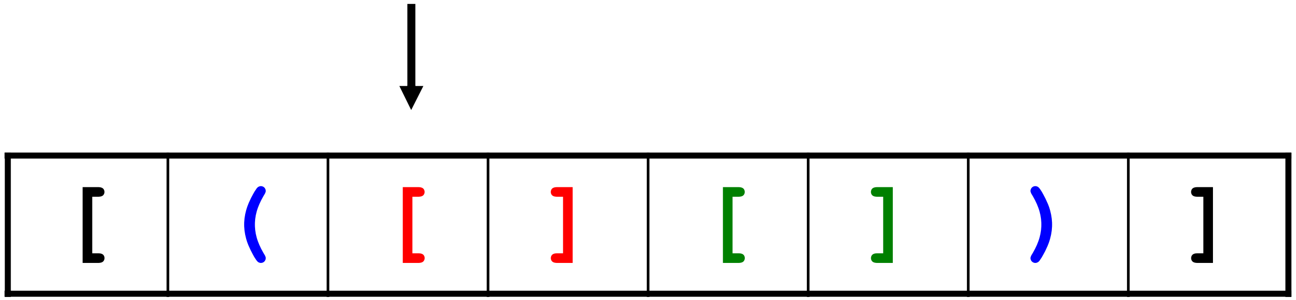


[	(	[	]	[	]	)	]

当前是(，和刚才的[不匹配，说明相匹配的符号还在右边，继续扫描

栈的应用：括号匹配检测

- 思考
  - 从左向右扫描字符串

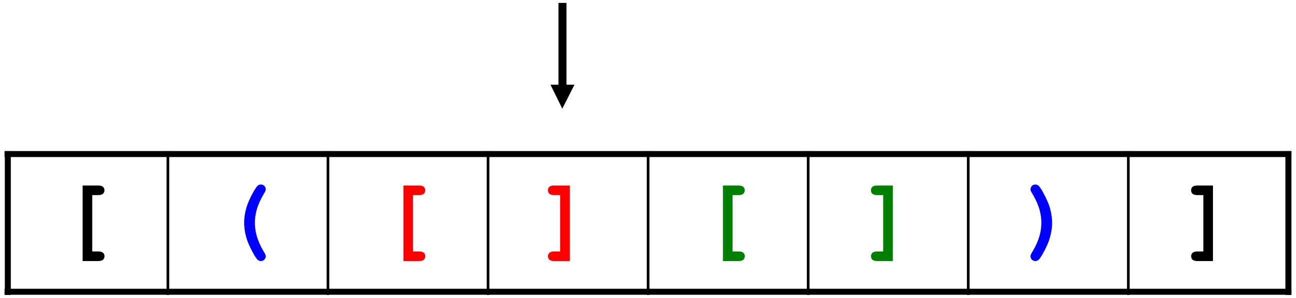


[	(	[	]	[	]	)	]

当前是 **[**，和刚才的 **(** 不匹配，说明相匹配的符号还在右边，继续扫描

栈的应用：括号匹配检测

- 思考
  - 从左向右扫描字符串

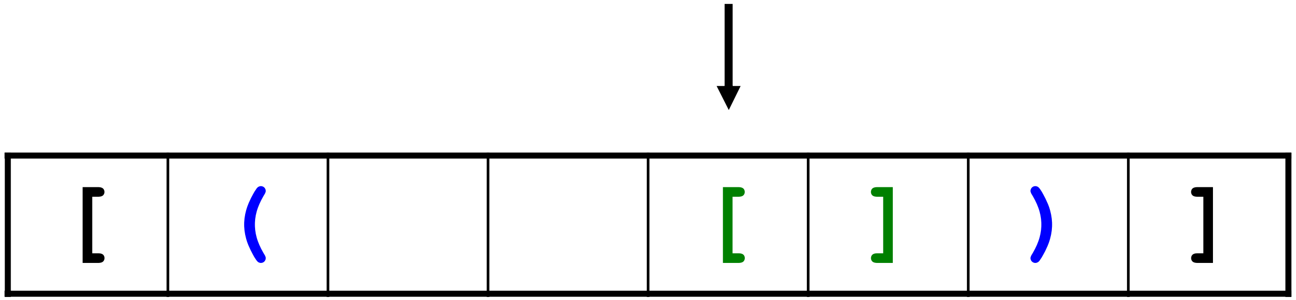


[	(	[	]	[	]	)	]

当前是]，和刚才的[正好一对，
可以从字符串中“删去”不考虑了

栈的应用：括号匹配检测

- 思考
  - 从左向右扫描字符串

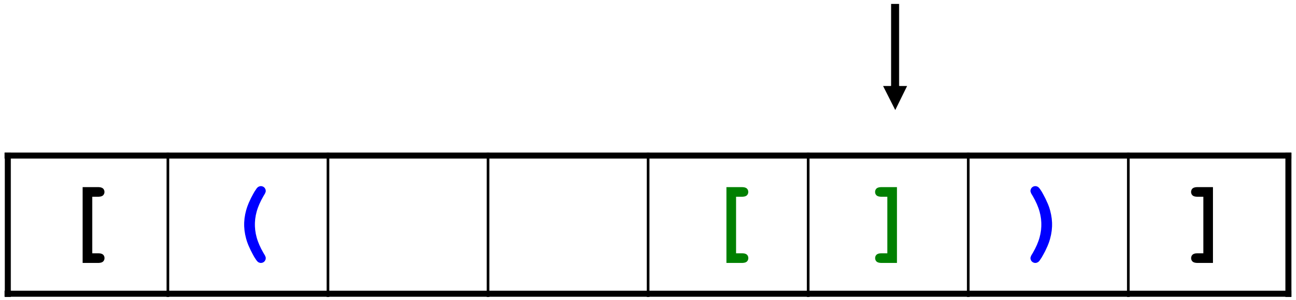


[	(			[	]	)	]

当前是 **[**，目前最近的一个是 **(**，
不匹配，继续扫描

栈的应用：括号匹配检测

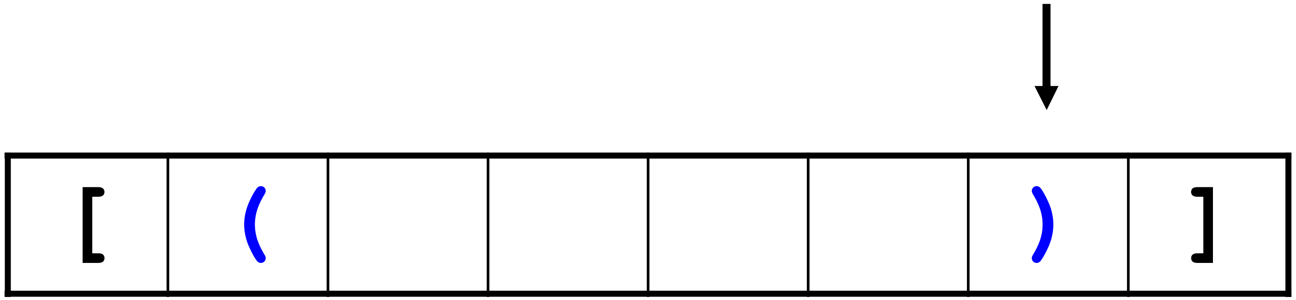
- 思考
  - 从左向右扫描字符串



当前是]，和刚才的[正好一对，
可以从字符串中“删去”不考虑了

栈的应用：括号匹配检测

- 思考
  - 从左向右扫描字符串

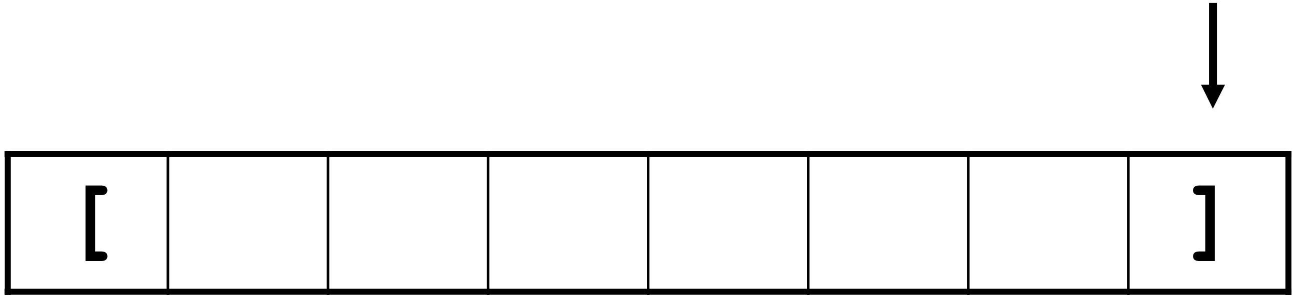


[	(					)	]

当前是)，目前最近的一个是(，正好一对，可以从字符串中“删去”不考虑了

栈的应用：括号匹配检测

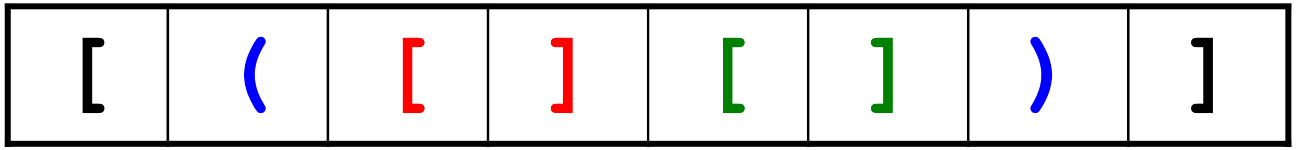
- 思考
  - 从左向右扫描字符串



当前是]，目前最近的一个是[，正好一对，可以从字符串中“删去”不考虑了，此时左右的括号都匹配成功

栈的应用：括号匹配检测

- 发现规律
  - 当扫描到当前字符的时候，需要知道已经扫描过的字符中，哪一个离它最近
  - 因此希望有一个工具，能够记录扫描的历史，这样可以方便的得到最近的上一次访问的字符



[	(	[	]	[	]	)	]

栈的应用：括号匹配检测

- 栈“记录历史”的特性
  - 人的记忆：
    - 越早发生的事情越难回忆
    - 越迟发生的事情越容易回忆
  - 栈的先进后出
    - 越早压入的元素越晚弹出
    - 越迟压入的元素越早弹出
  - 因此很自然的想到利用栈来模拟记忆

栈的应用：括号匹配检测

- 算法思想

```
准备一个栈，用于存放扫描遇到的左括号
从左向后扫描每一个字符{
    如果遇到的是左括号，则入栈
    如果遇到的是右括号，则
            把栈顶字符和当前字符比较
            若匹配，则弹出栈顶字符，继续向前扫描
            若不匹配，程序返回不匹配标志
    }
当所有字符都扫描完毕，栈应当为空
```

栈的应用：括号匹配检测

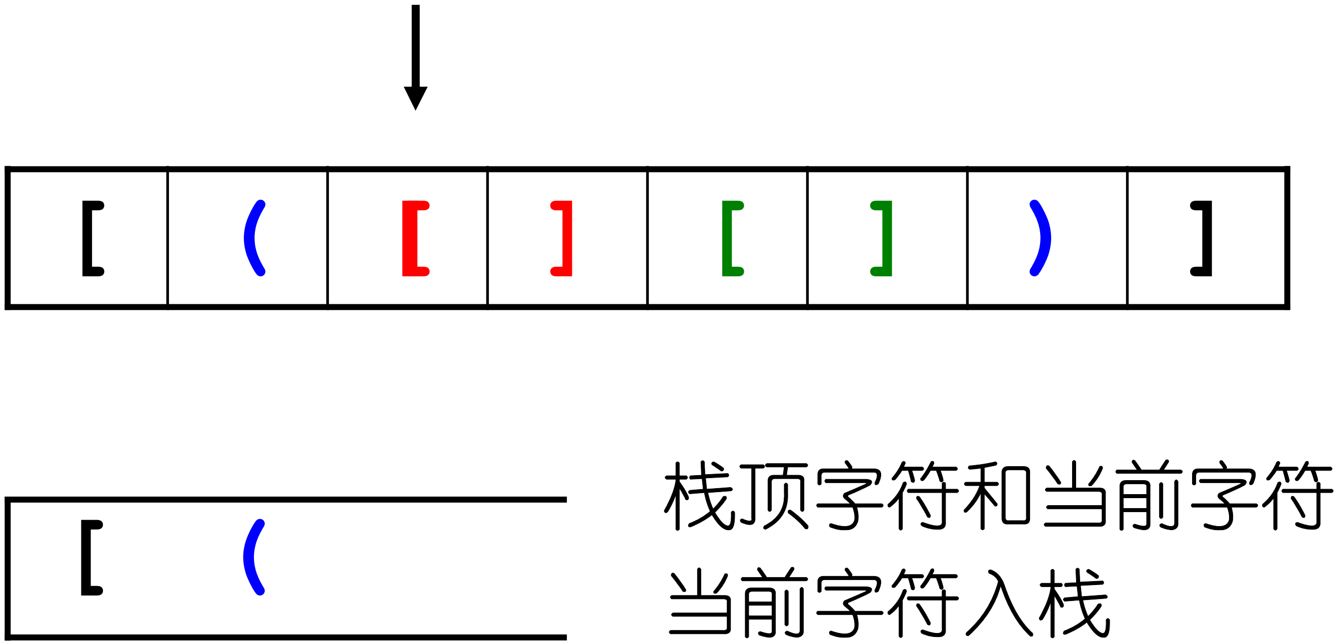
[	(	[	]	[	]	)	]

栈为空，
当前字符直接入栈

栈的应用：括号匹配检测

栈顶字符和当前字符不匹配，
当前字符入栈

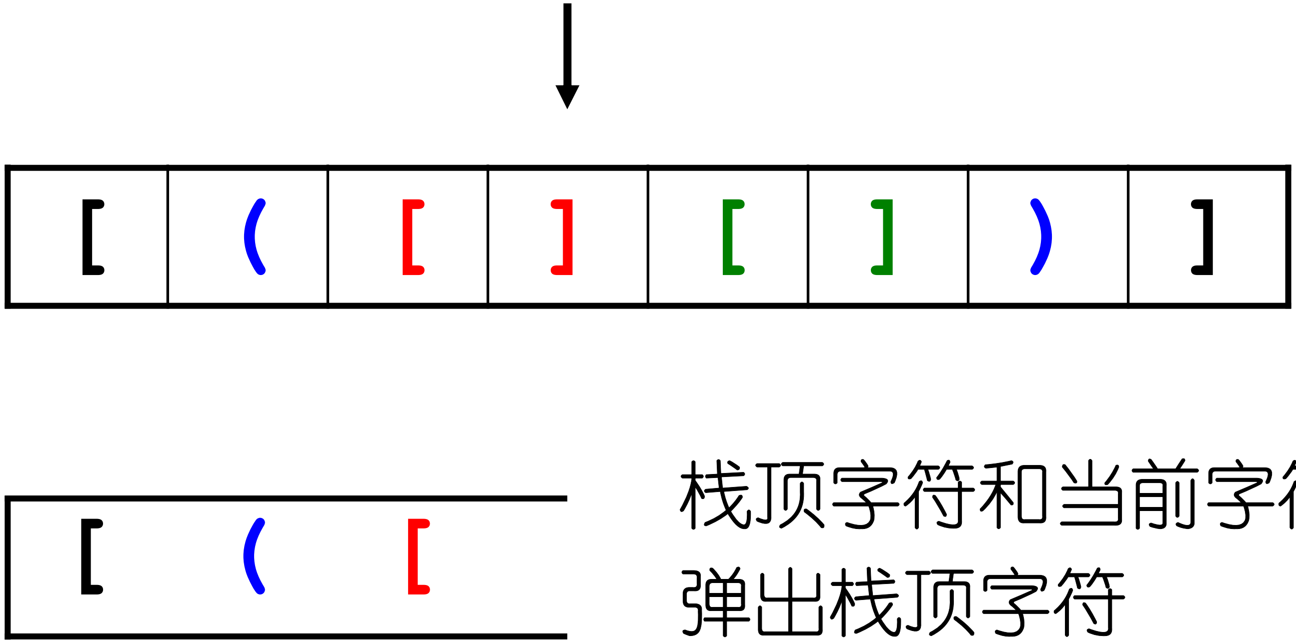
栈的应用：括号匹配检测



[	(	[	]	[	]	)	]

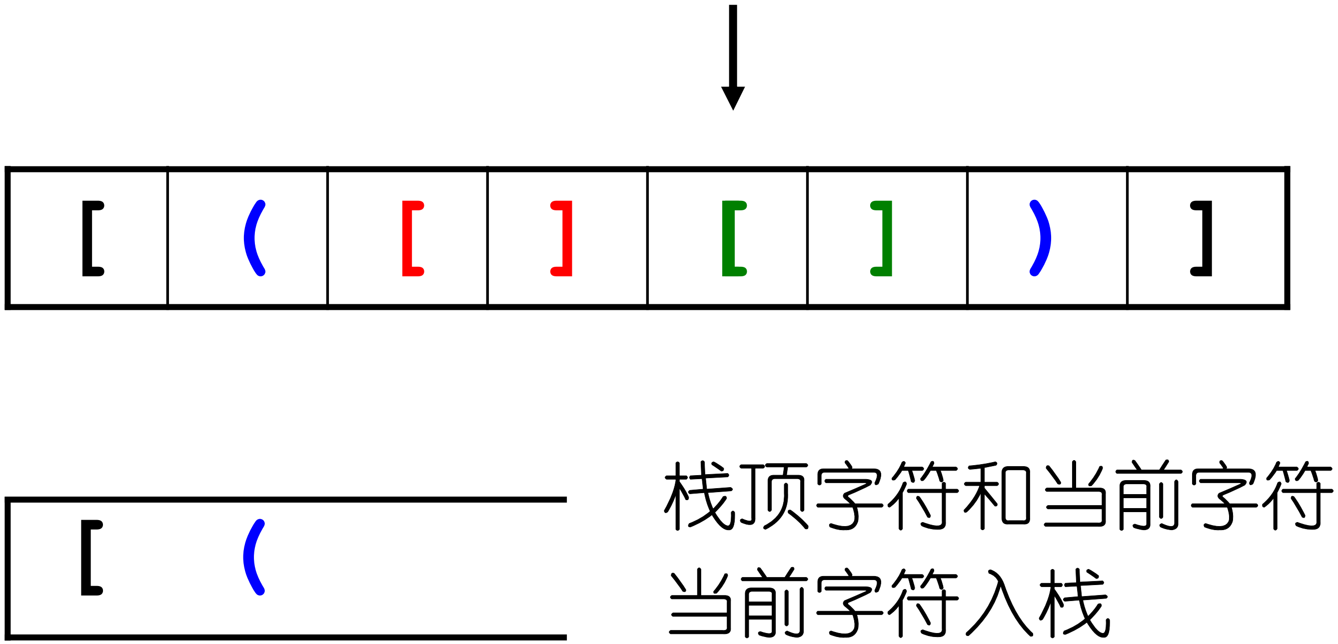
栈顶字符和当前字符不匹配，
当前字符入栈

栈的应用：括号匹配检测



栈顶字符和当前字符匹配，
弹出栈顶字符

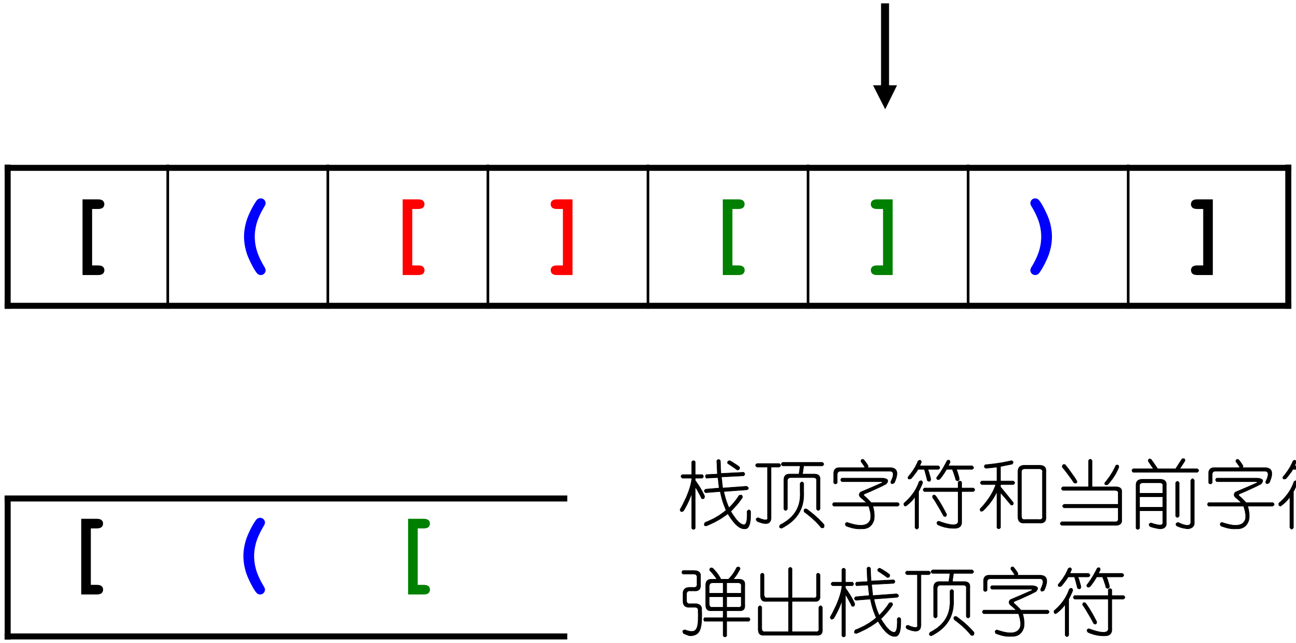
栈的应用：括号匹配检测



[	(	[	]	[	]	)	]

栈顶字符和当前字符不匹配，
当前字符入栈

栈的应用：括号匹配检测

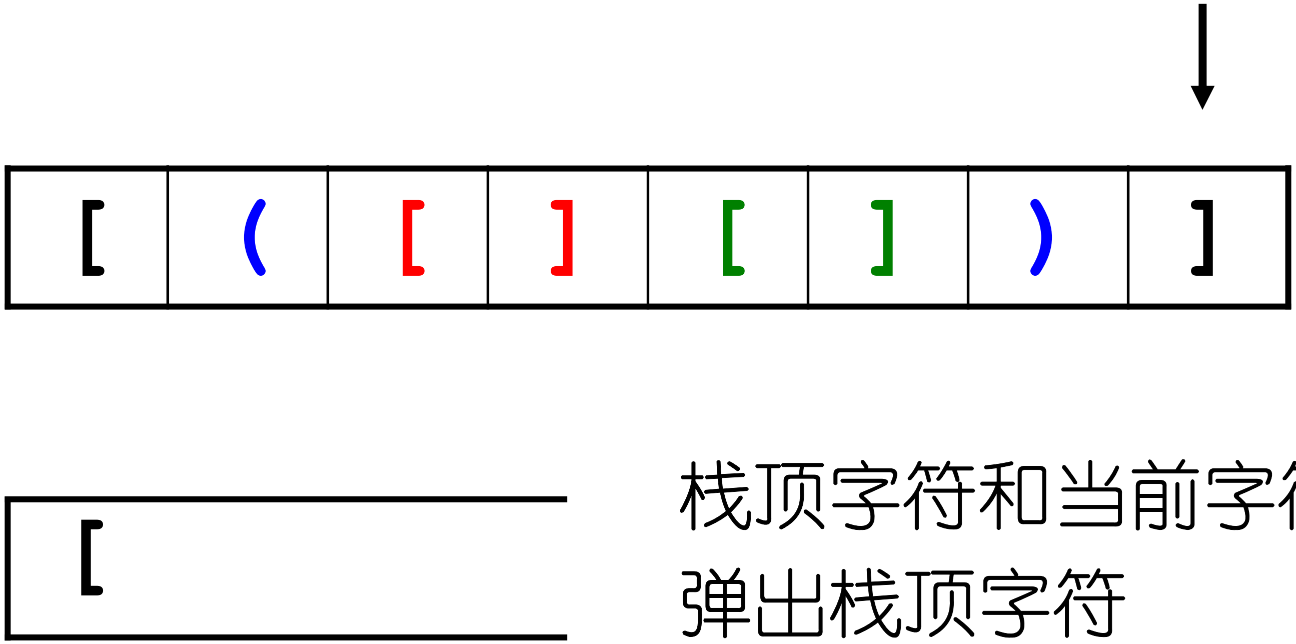


栈顶字符和当前字符匹配，
弹出栈顶字符

栈的应用：括号匹配检测

栈顶字符和当前字符匹配，
弹出栈顶字符

栈的应用：括号匹配检测



[	(	[	]	[	]	)	]

栈顶字符和当前字符匹配，
弹出栈顶字符

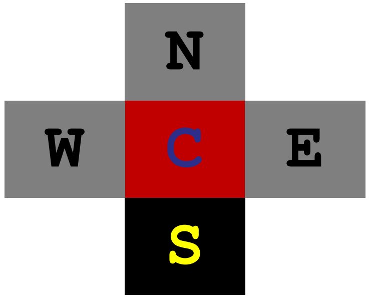
栈的应用：走迷宫

- 是一个探索过程
  - 从入口出发，当到达一个地点时，需要探索从该点按某个方向可到达的下一个地点，如可到达，则前进到新的地点；

C

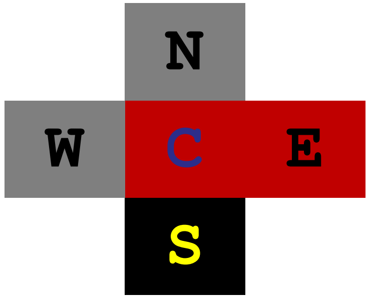
栈的应用：走迷宫

- 是一个探索过程
  - 从入口出发，当到达一个地点时，需要探索从该点按某个方向可到达的下一个地点，如可到达，则前进到新的地点；



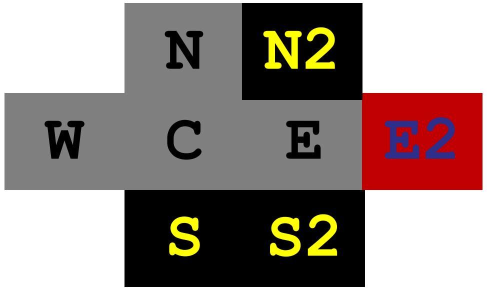
栈的应用：走迷宫

- 是一个探索过程
  - 从入口出发，当到达一个地点时，需要探索从该点按某个方向可到达的下一个地点，如可到达，则前进到新的地点；



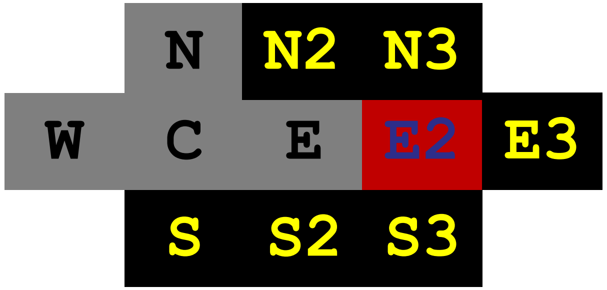
栈的应用：走迷宫

- 是一个探索过程
  - 从入口出发，当到达一个地点时，需要探索从该点按某个方向可到达的下一个地点，如可到达，则前进到新的地点；



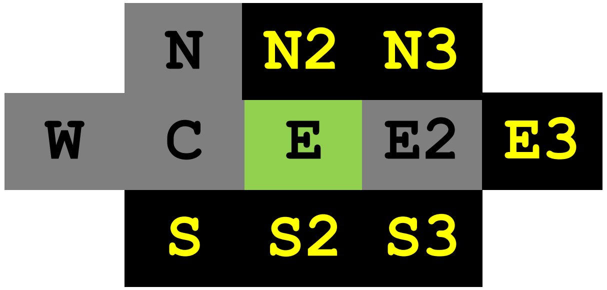
栈的应用：走迷宫

- 是一个探索过程
  - 从入口出发，当到达一个地点时，需要探索从该点按某个方向可到达的下一个地点，如可到达，则前进到新的地点；



栈的应用：走迷宫

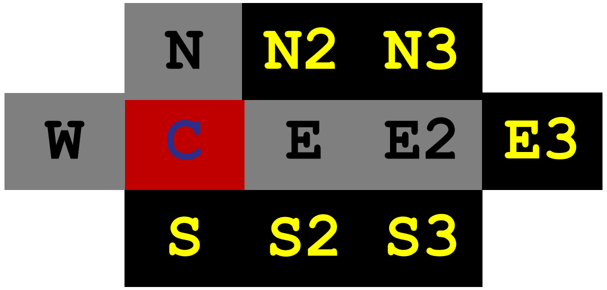
- 是一个探索过程
  - 从入口出发，当到达一个地点时，需要探索从该点按某个方向可到达的下一个地点，如可到达，则前进到新的地点；



  - 如当前地点不可通（四周堵死），则退回到路径上的前一个地点；

栈的应用：走迷宫

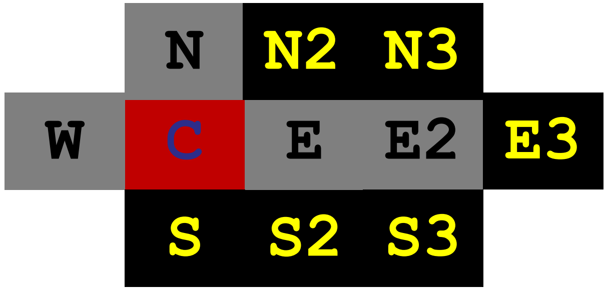
- 是一个探索过程
  - 从入口出发，当到达一个地点时，需要探索从该点按某个方向可到达的下一个地点，如可到达，则前进到新的地点；

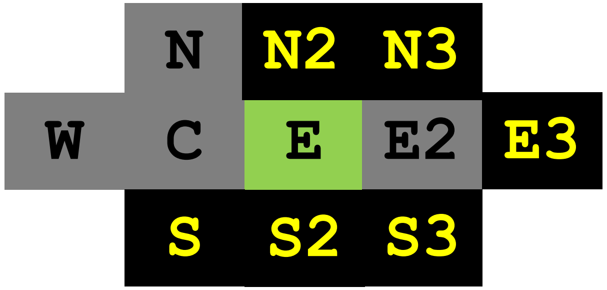


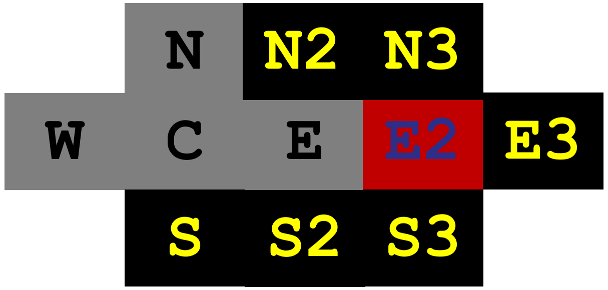
  - 如当前地点不可通（四周堵死），则退回到路径上的前一个地点；

栈的应用：走迷宫

- 是一个探索过程
  - 如当前地点不可通（四周堵死或走过），则退回到路径上的前一个地点；







  - 需要保存走过的路径，按后进先退的过程回退！

栈的应用：走迷宫

- 再次应用栈来记录历史
  - 为了保证在任何位置上都能沿原路退回，需要用一个“后进先出”的结构即栈来保存从入口到当前位置的路径
  - 在走出出口之后，栈中保存的正是一条从入口到出口的路径

```
初始化一个栈，入口位置入栈
while(栈不空){
    //栈顶位置为当前位置
    while(栈不空但栈顶位置四周均不通)
            弹出栈顶
    if(栈不空){
            前进到栈顶位置的下一个可通位置
    }
}
```

栈的应用：表达式求值

对含+、-、*、/、()的表达式进行求值，如

7+（4-2）*3-10/5

四则运算规则

（1） 先乘除、后加减

（2） 先左后右

（3） 先括号内后括号外

表达式的开头和结尾虚设#构成整个表达式的括号。上述表达式变为

#7+（4-2）*3-10/5#

运算符和界限符统称为算符，其集合记为OP.有：

+、-、*、/、（、）、#

表达式求值：算符优先关系

【例】　　#3*(2+4)-8/2#

【分析】

依次读入操作数和运算符，考虑运算符的优先级别，在读入运算符 θ_1时，如下一个运算符 θ_2 不比 θ_1优先，则可以用 θ_1计算;如下一个运算符 θ_2比 θ_1优先，则先保存 θ_1 ，待 θ_2运算完才能计算 θ_1 。依次类推。

例如 “-8+2”， -优先于+，可算- ，再算+

再如 “-8/2”，/优先于-，应该先算/，再算-

表达式求值：算符优先关系表

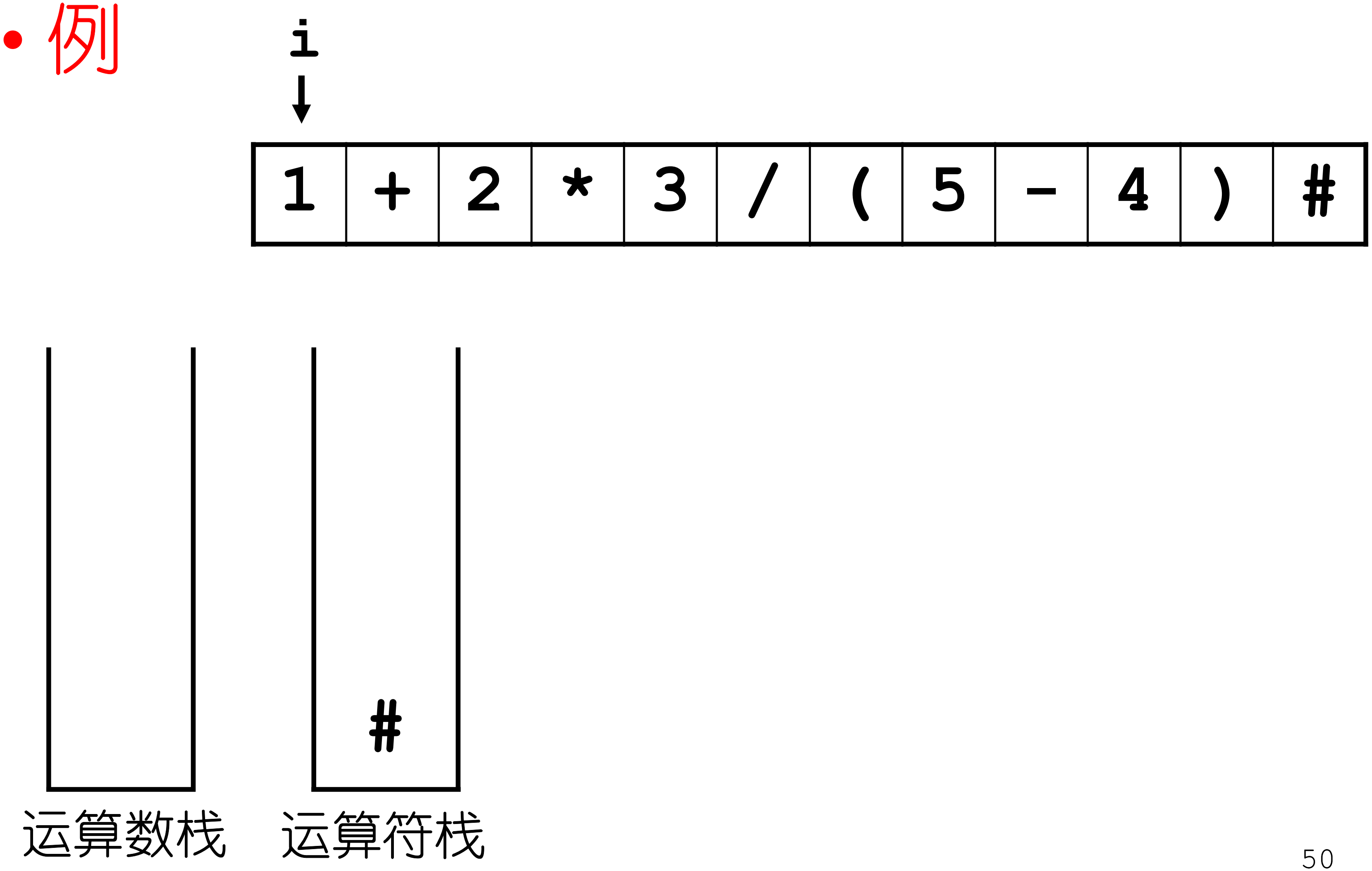
设表达式中算符θ_1出现在算符θ_2前，则两者优先关系：

θ_2 \ θ_1	+	-	*	/	(	)	#
+	>	>	<	<	<	>	>
-	>	>	<	<	<	>	>
*	>	>	>	>	<	>	>
/	>	>	>	>	<	>	>
(	<	<	<	<	<	=	无
)	>	>	>	>	无	>	>
#	<	<	<	<	<	无	=

优先级相等的只有：（、）；　#、#

表达式求值：#3*(2+4)-8/2#的计算

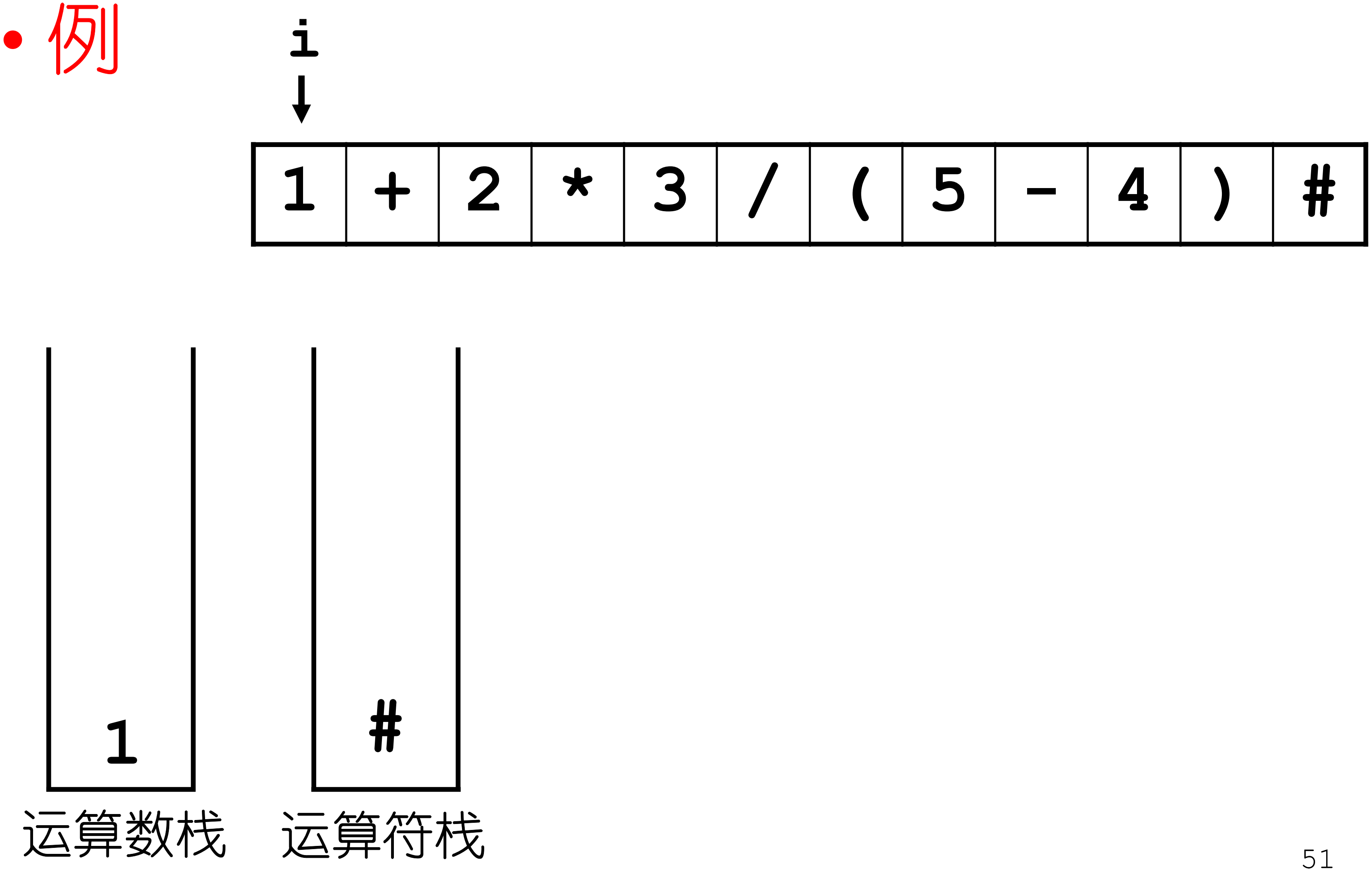
- 例



1	+	2	*	3	/	(	5	-	4	)	#

表达式求值：#3*(2+4)-8/2#的计算

- 例



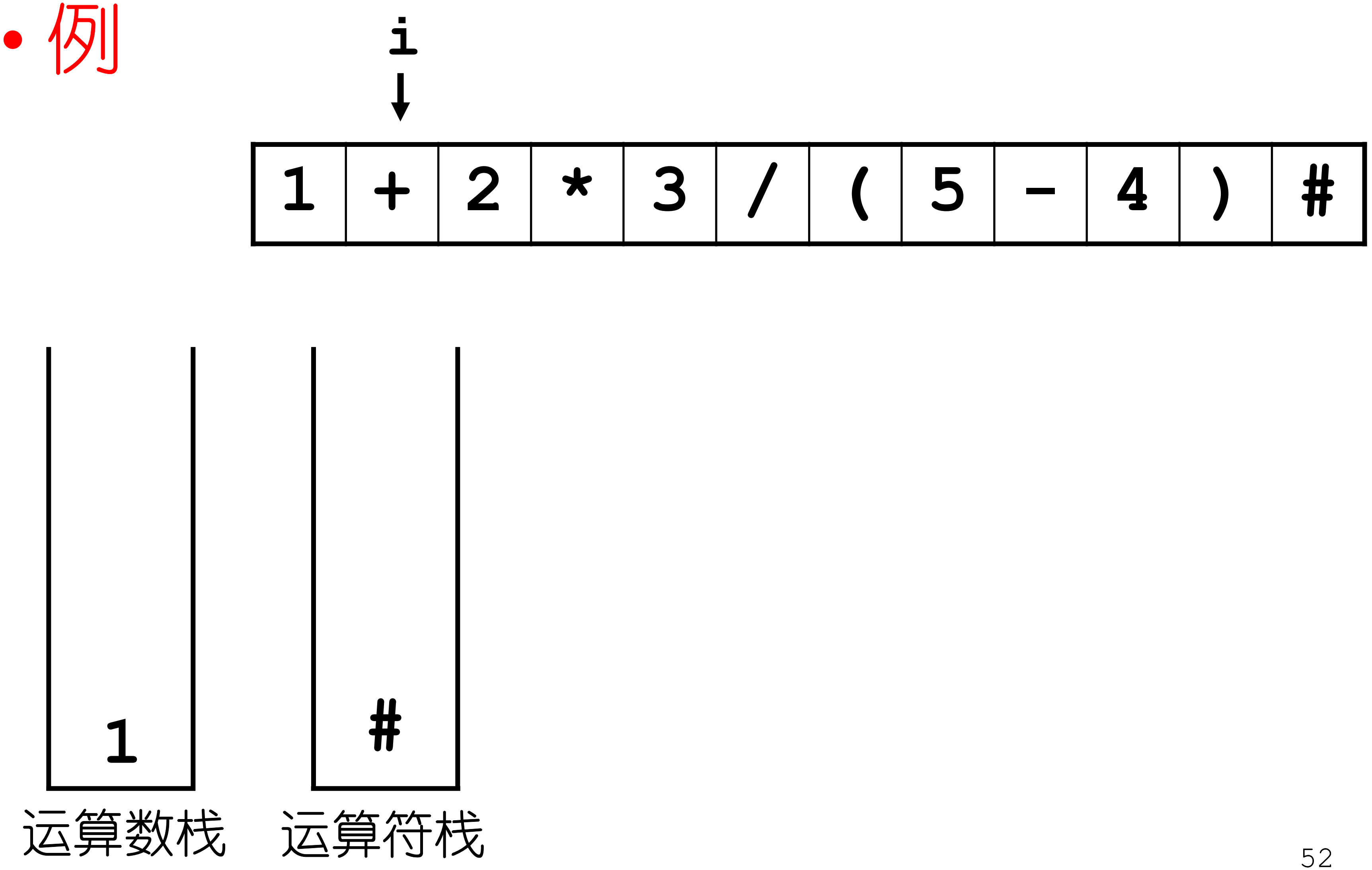
i

1	+	2	*	3	/	(	5	-	4	)	#

运算数栈　运算符栈

表达式求值：#3*(2+4)-8/2#的计算

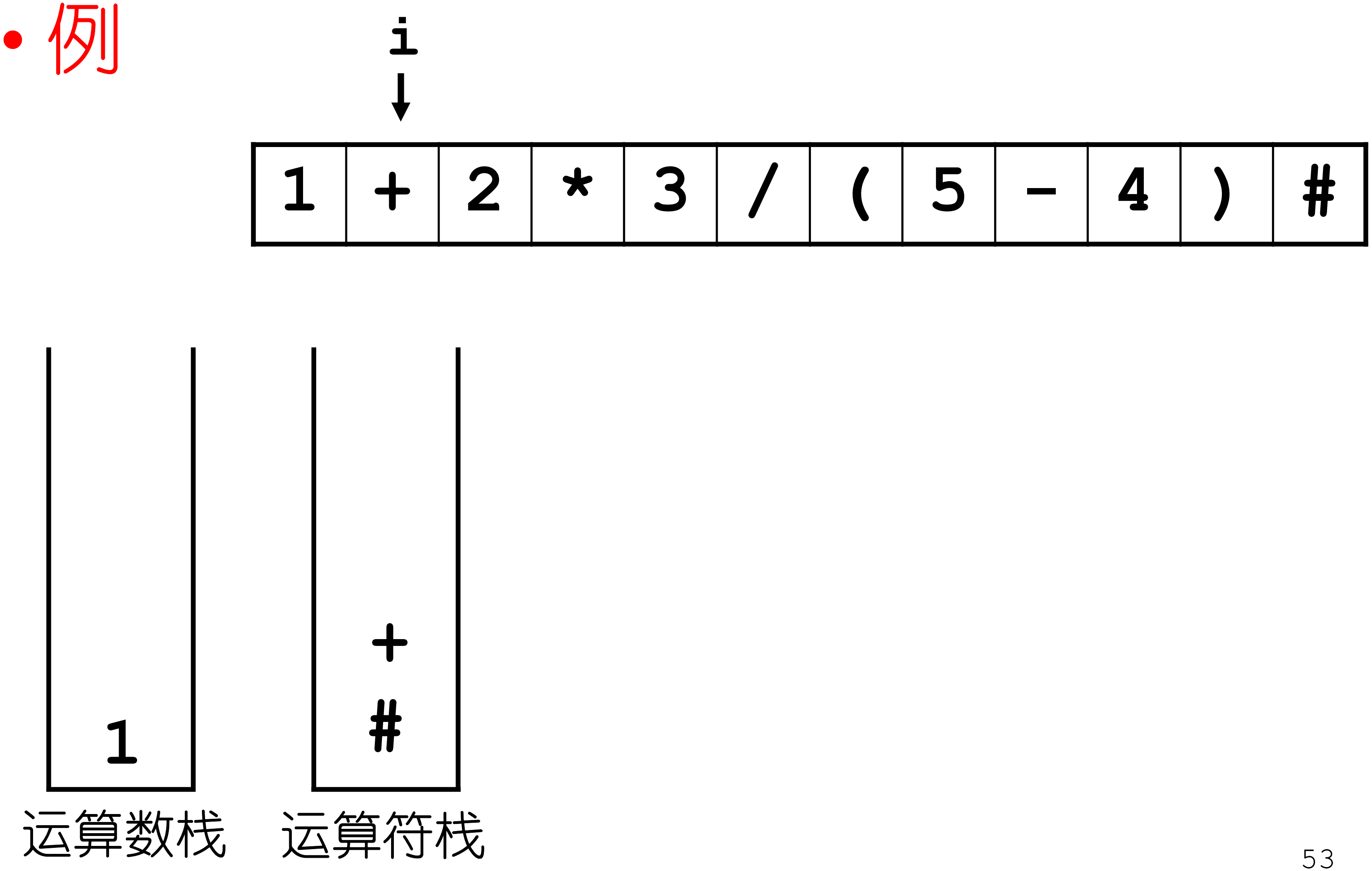
- 例



1	+	2	*	3	/	(	5	-	4	)	#

表达式求值：#3(2+4)-8/2#的计算*

- 例

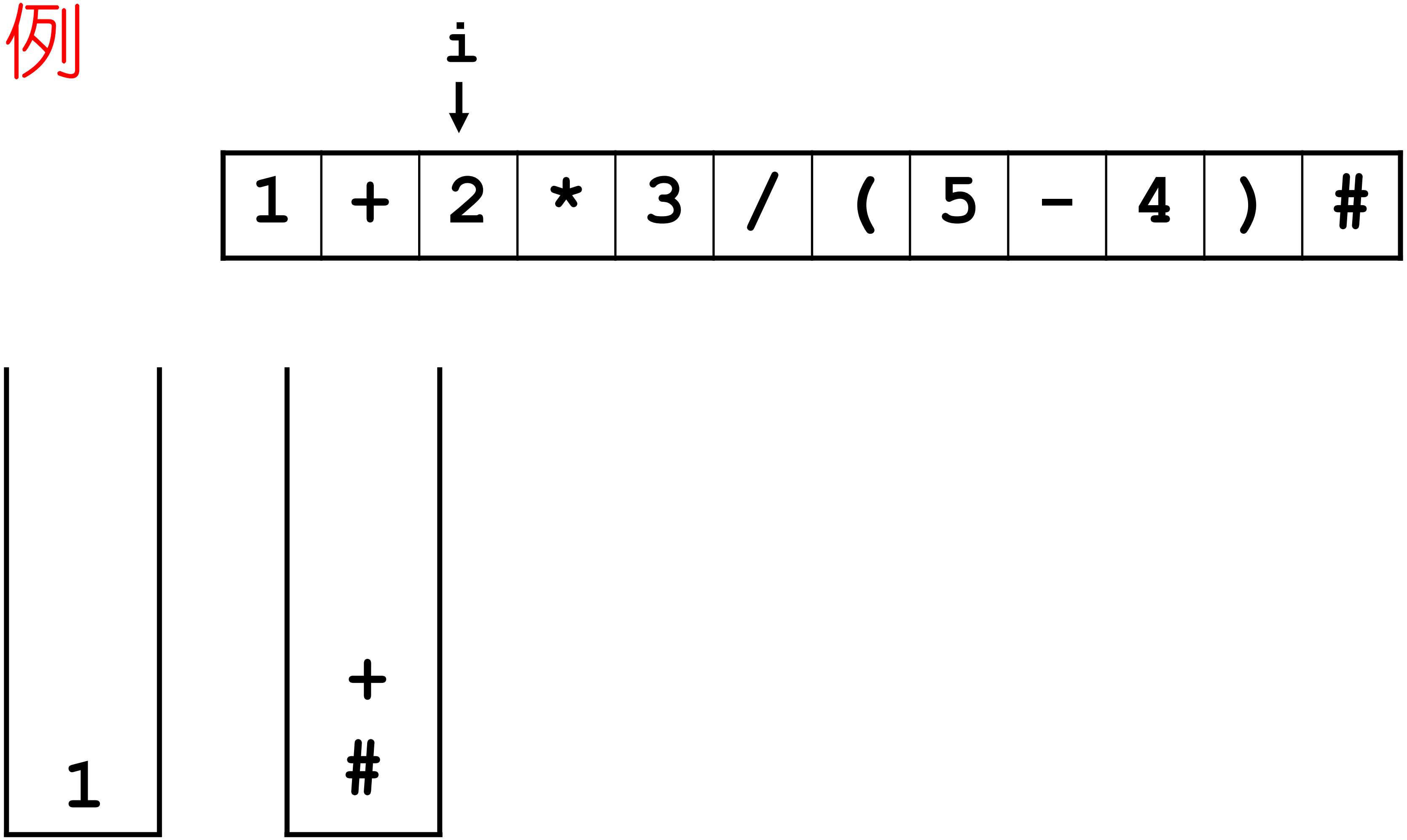


i

1	+	2	*	3	/	(	5	-	4	)	#

表达式求值：#3*(2+4)-8/2#的计算

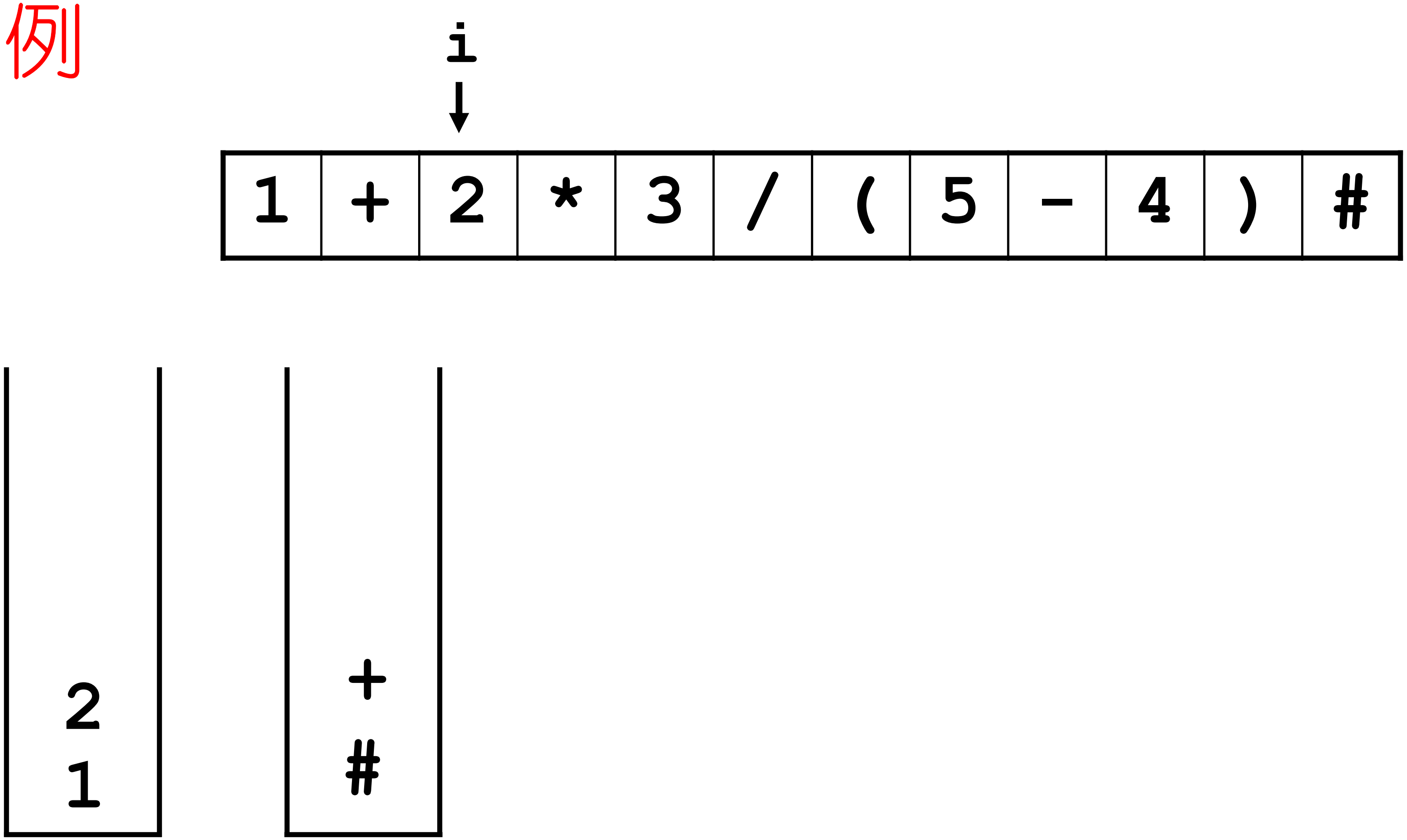
- 例



运算数栈 运算符栈

表达式求值：#3*(2+4)-8/2#的计算

- 例



i

1	+	2	*	3	/	(	5	-	4	)	#

2
1

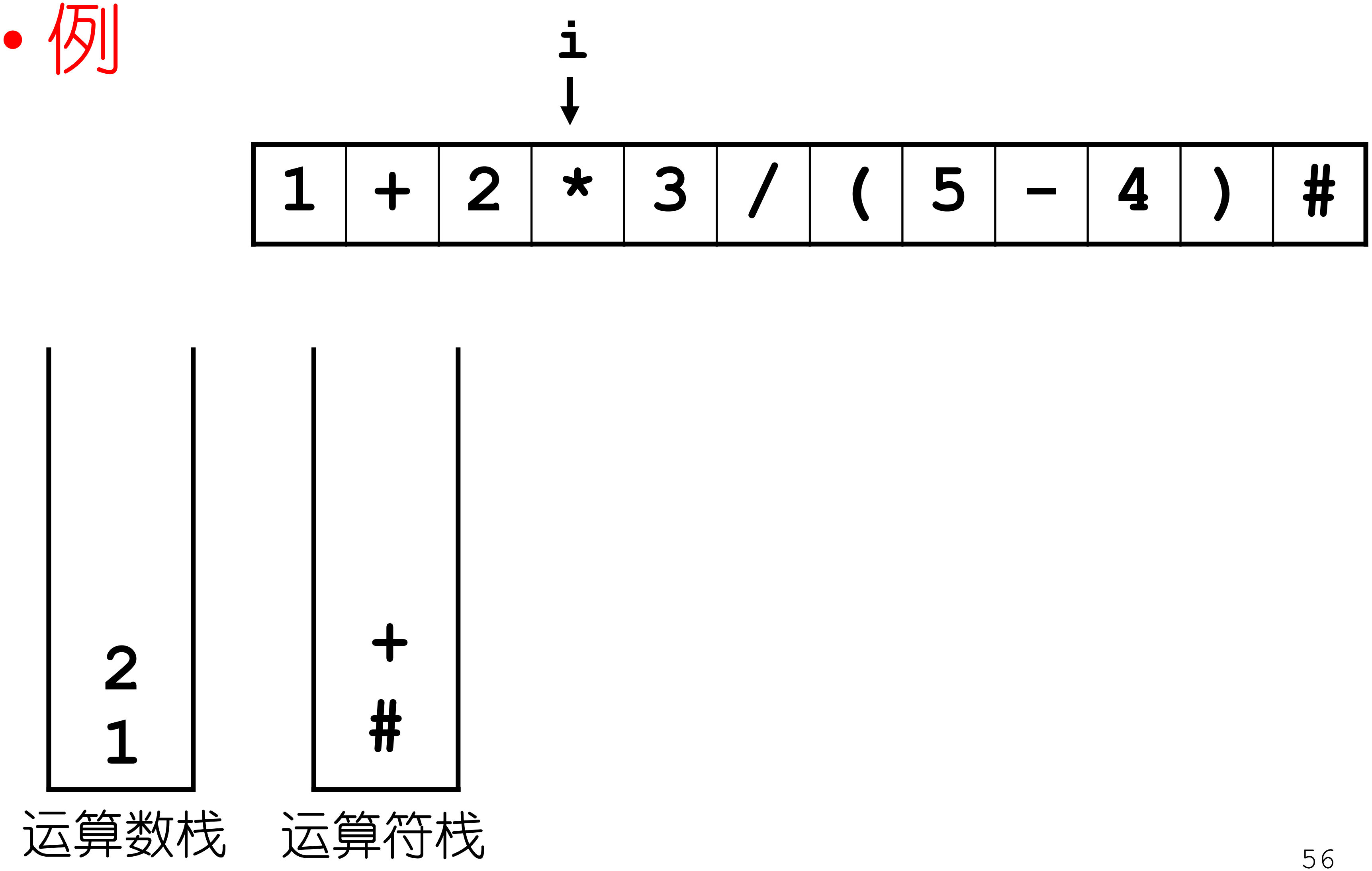
运算数栈

+
#

运算符栈

表达式求值：#3*(2+4)-8/2#的计算

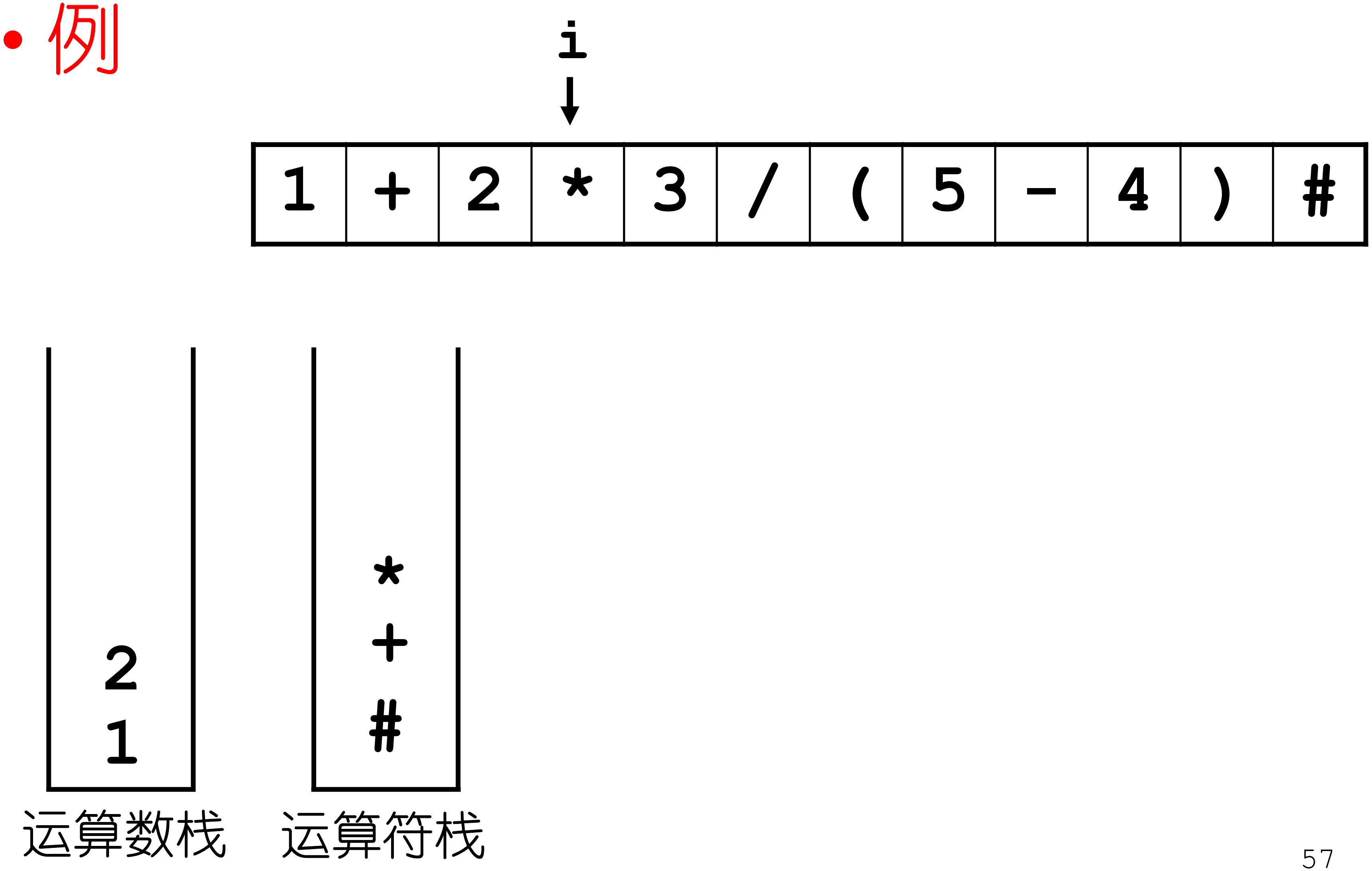
- 例



1	+	2	*	3	/	(	5	-	4	)	#

表达式求值：#3(2+4)-8/2#的计算*

- 例



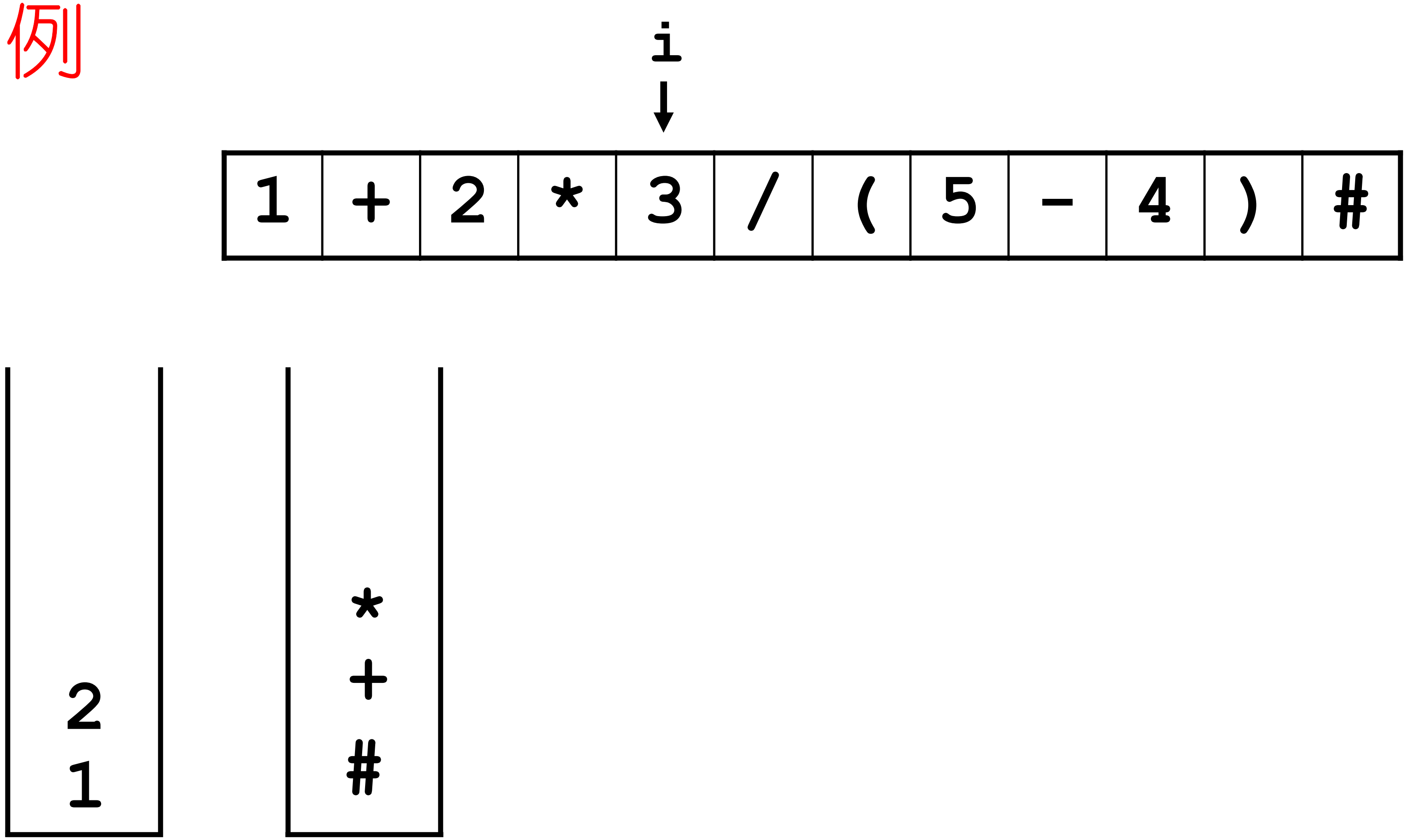
i

1	+	2	*	3	/	(	5	-	4	)	#

运算数栈　运算符栈

表达式求值：#3*(2+4)-8/2#的计算

- 例



i

1	+	2	*	3	/	(	5	-	4	)	#

2
1

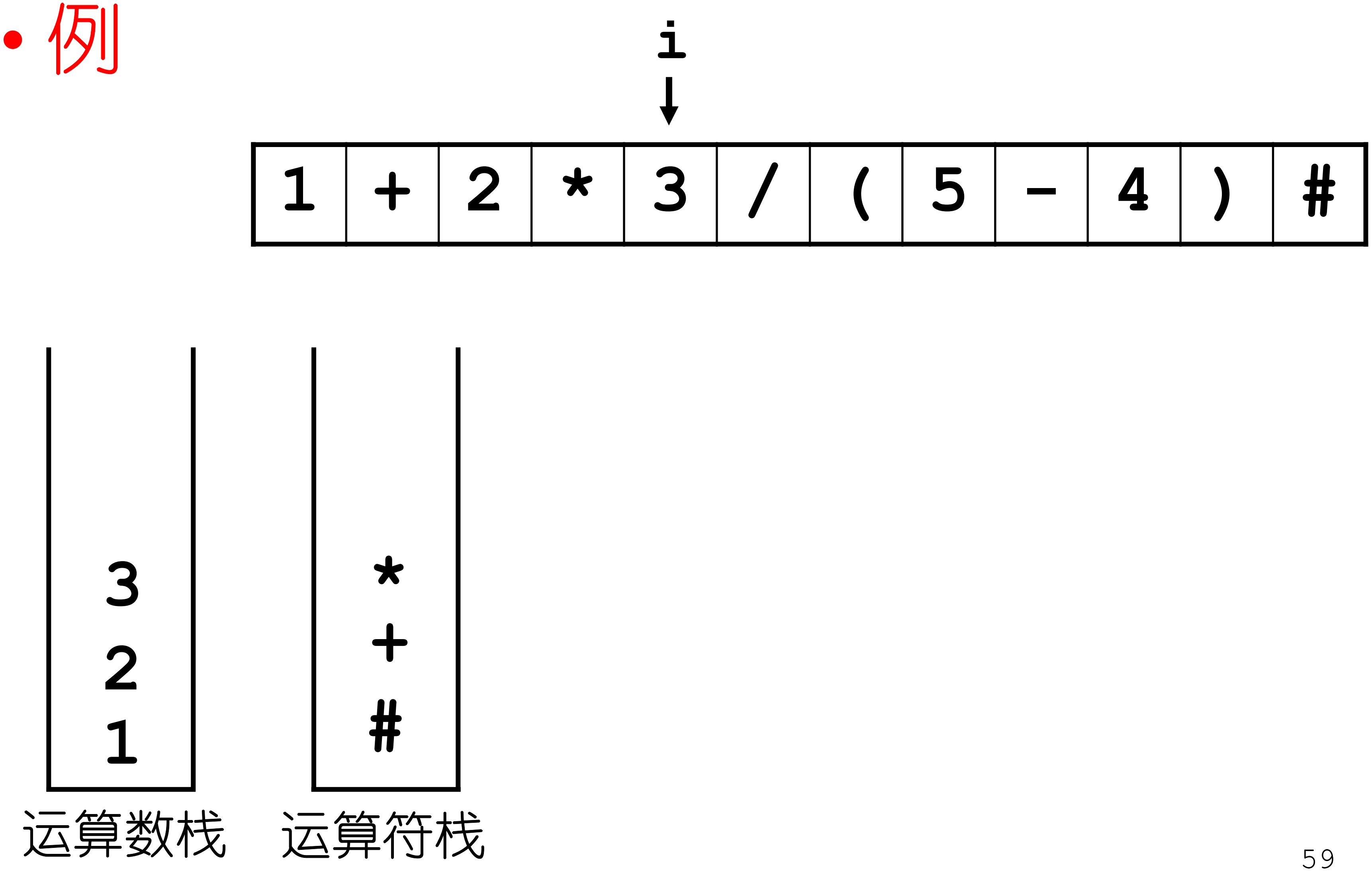
运算数栈

*
+
#

运算符栈

表达式求值：#3*(2+4)-8/2#的计算

- 例



i

1	+	2	*	3	/	(	5	-	4	)	#

3
2
1

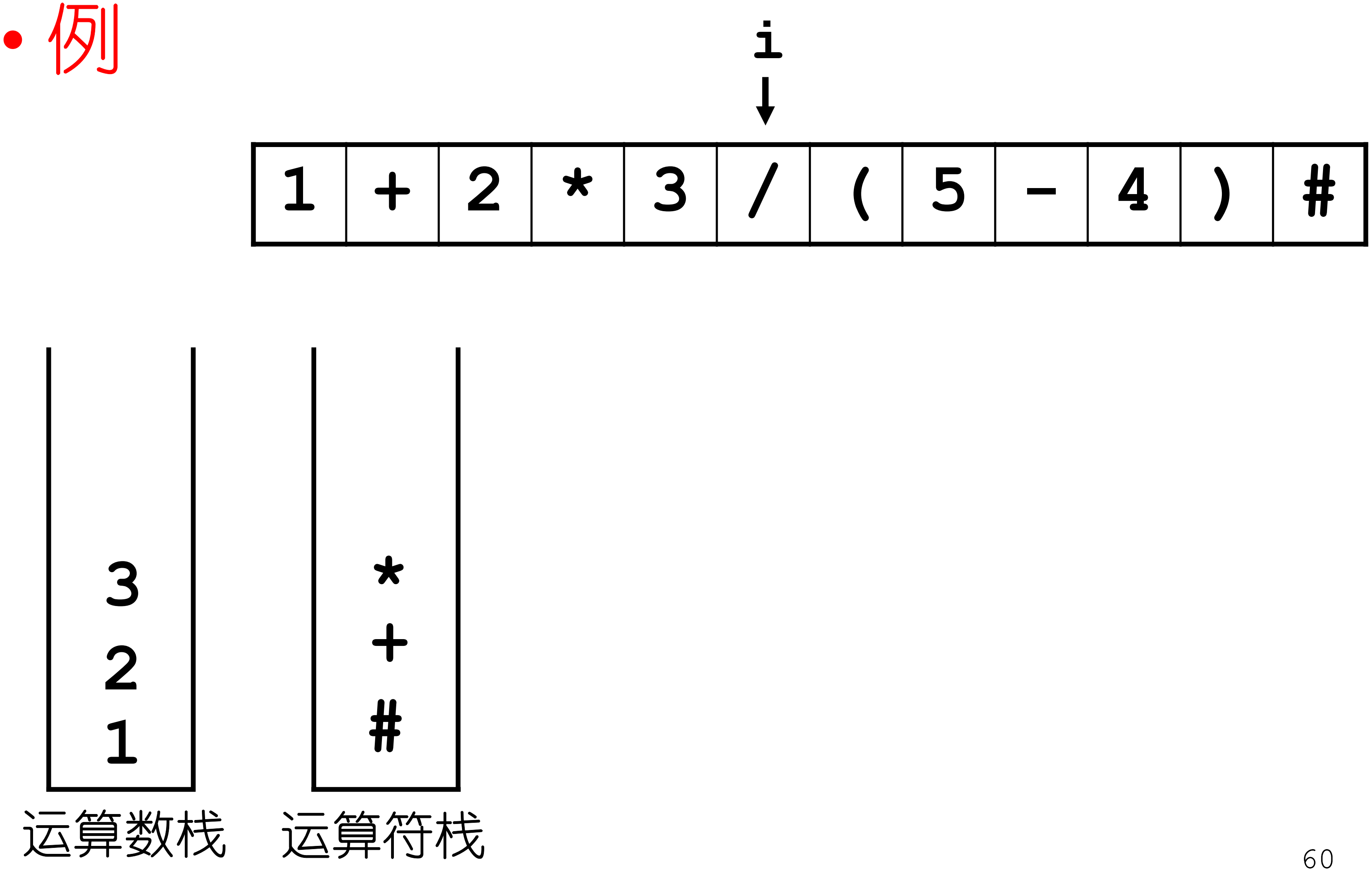
运算数栈

*
+
#

运算符栈

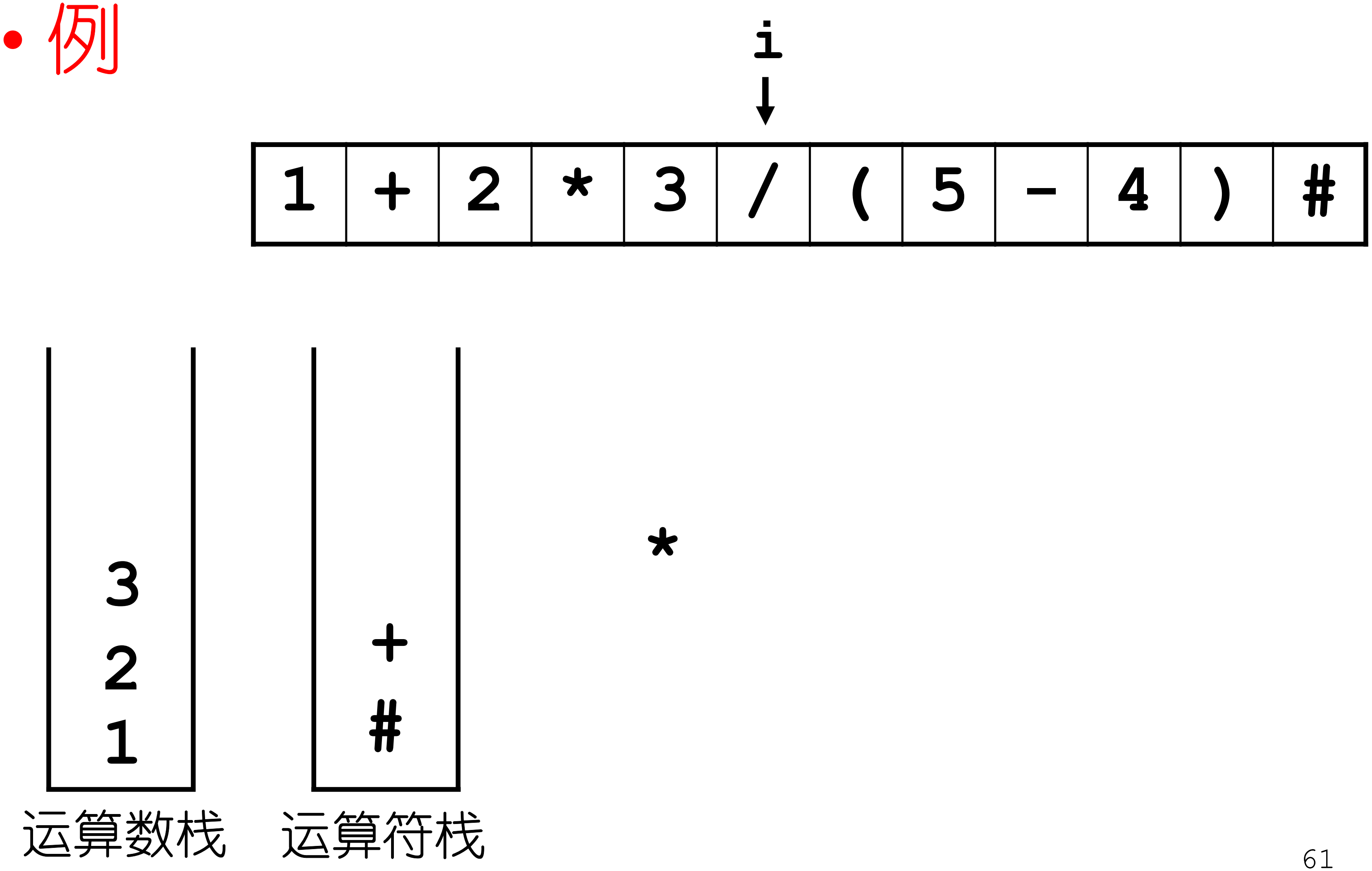
表达式求值：#3*(2+4)-8/2#的计算

- 例



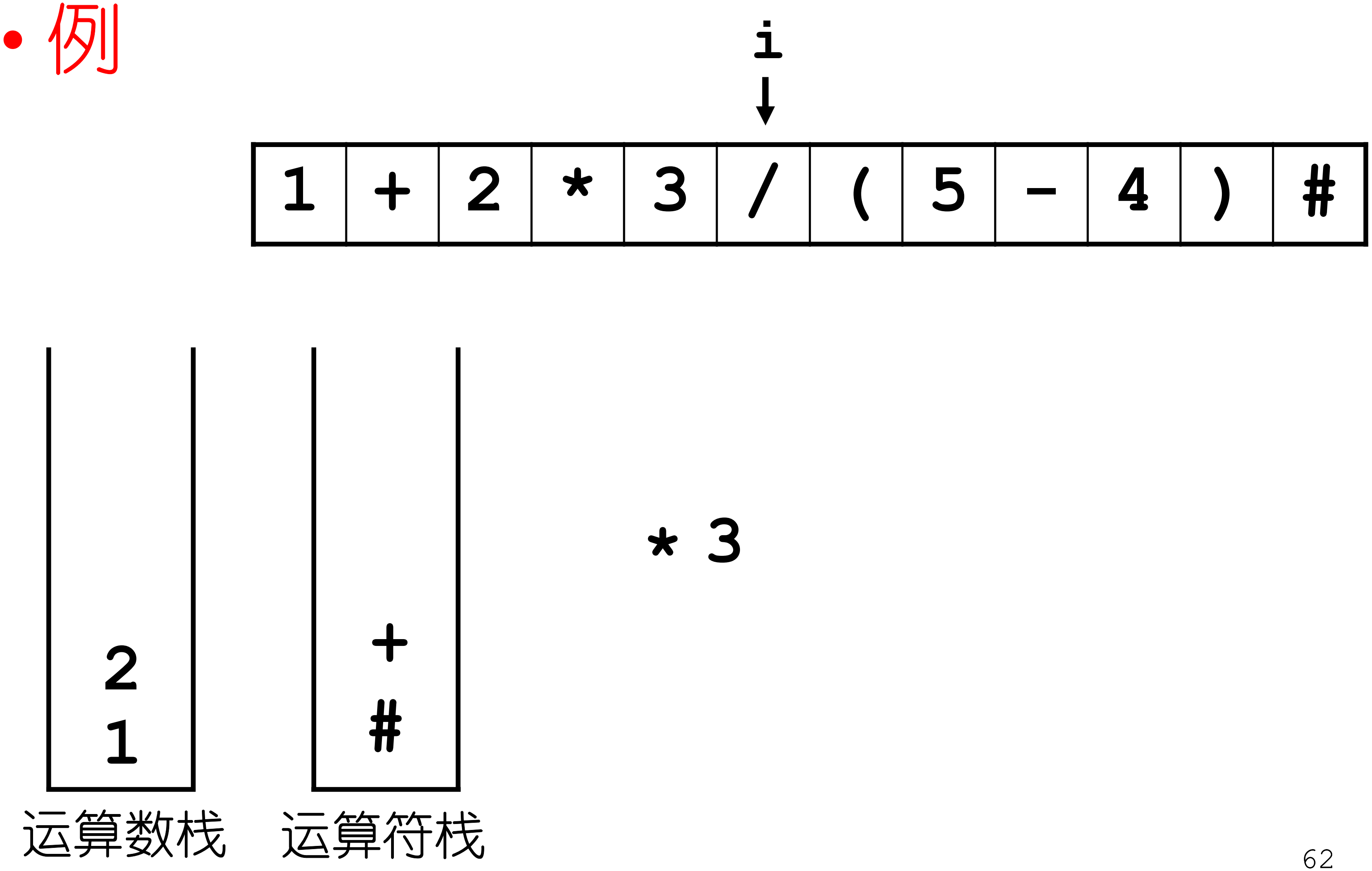
表达式求值：#3*(2+4)-8/2#的计算

- 例



表达式求值：#3*(2+4)-8/2#的计算

- 例



i

1	+	2	*	3	/	(	5	-	4	)	#

* 3

2

1

运算数栈

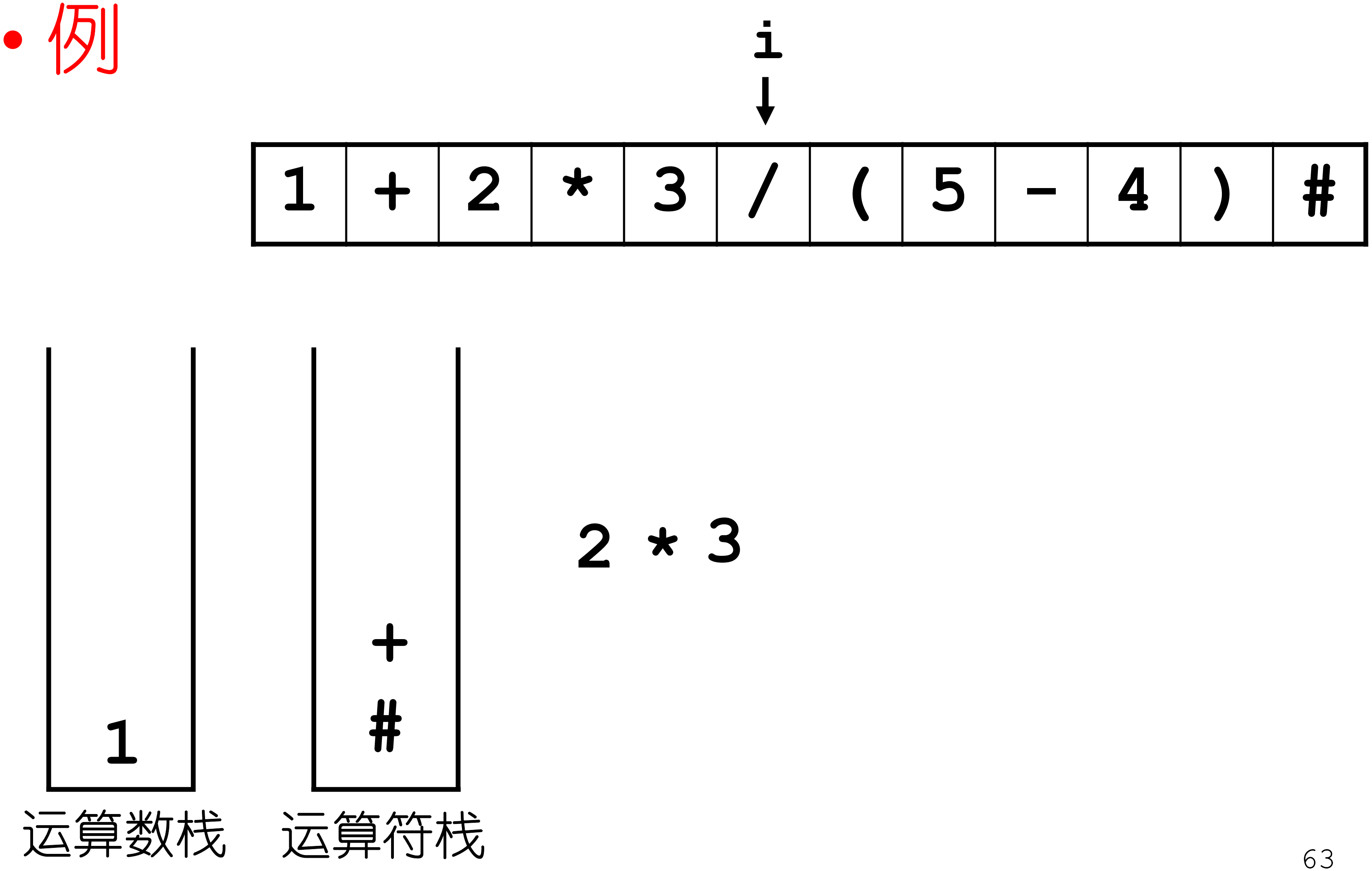
+

#

运算符栈

表达式求值：#3*(2+4)-8/2#的计算

- 例

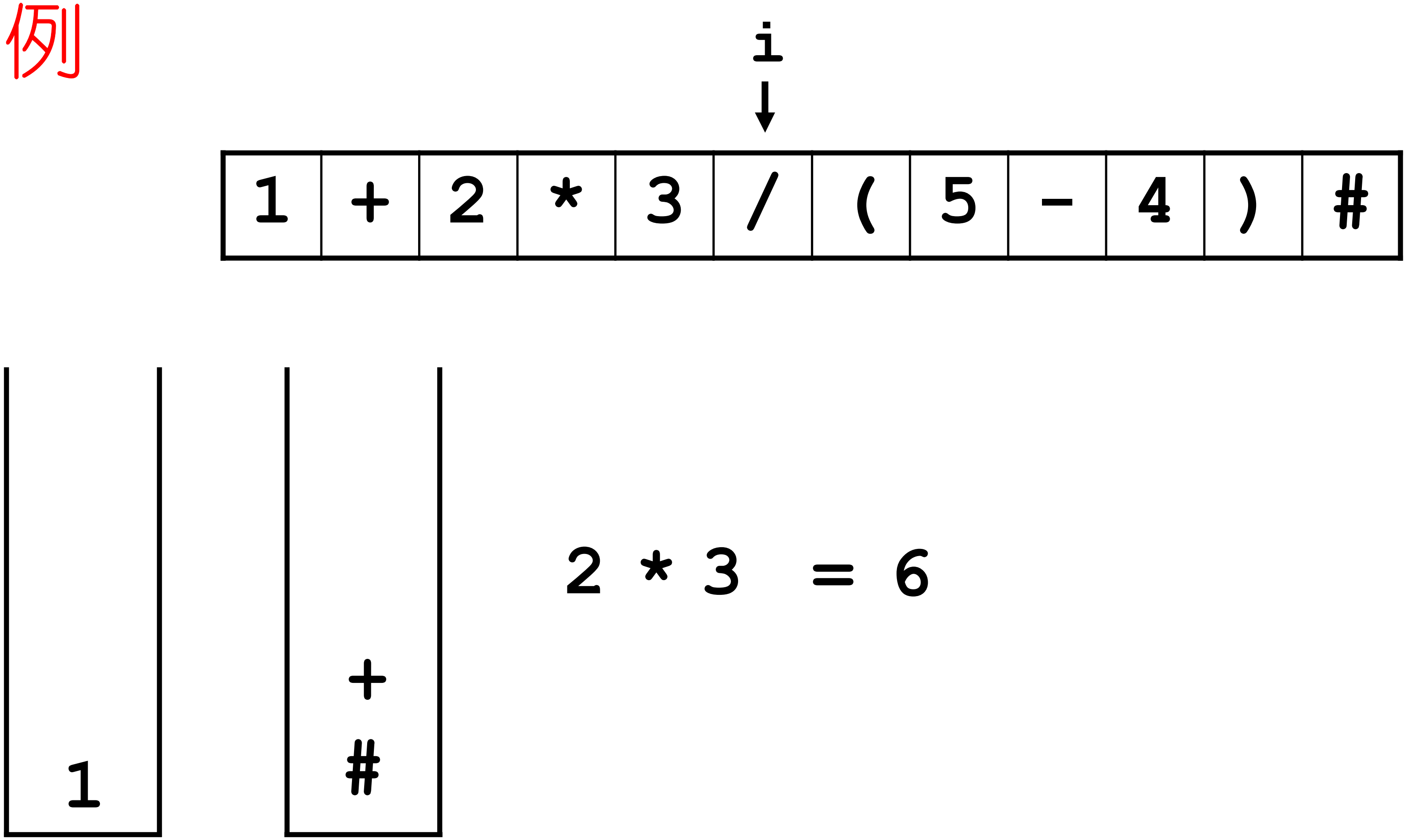


1	+	2	*	3	/	(	5	-	4	)	#

2 * 3

表达式求值：#3*(2+4)-8/2#的计算

- 例



i

1	+	2	*	3	/	(	5	-	4	)	#

1

+

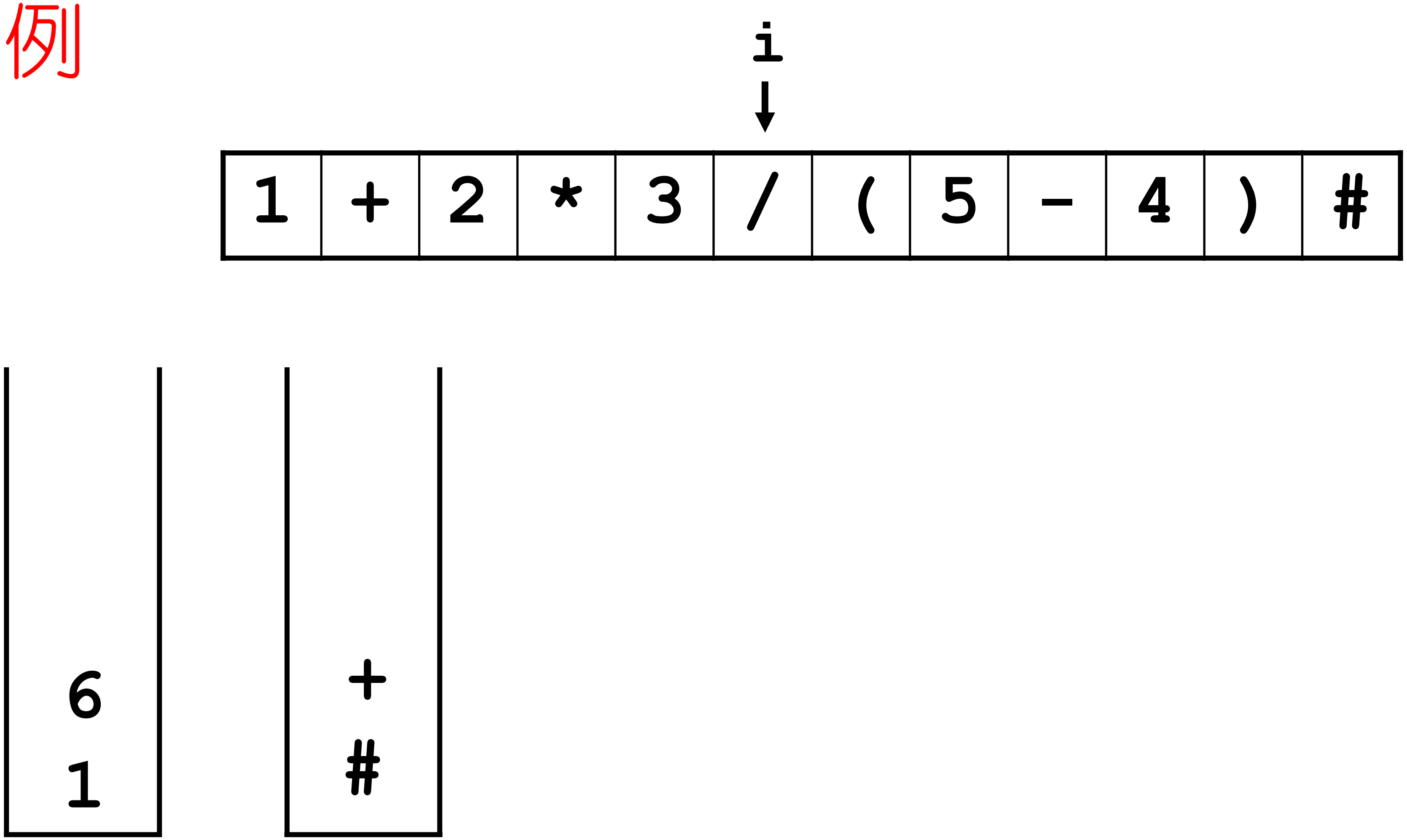
#

运算数栈 运算符栈

2 * 3 = 6

表达式求值：#3*(2+4)-8/2#的计算

- 例

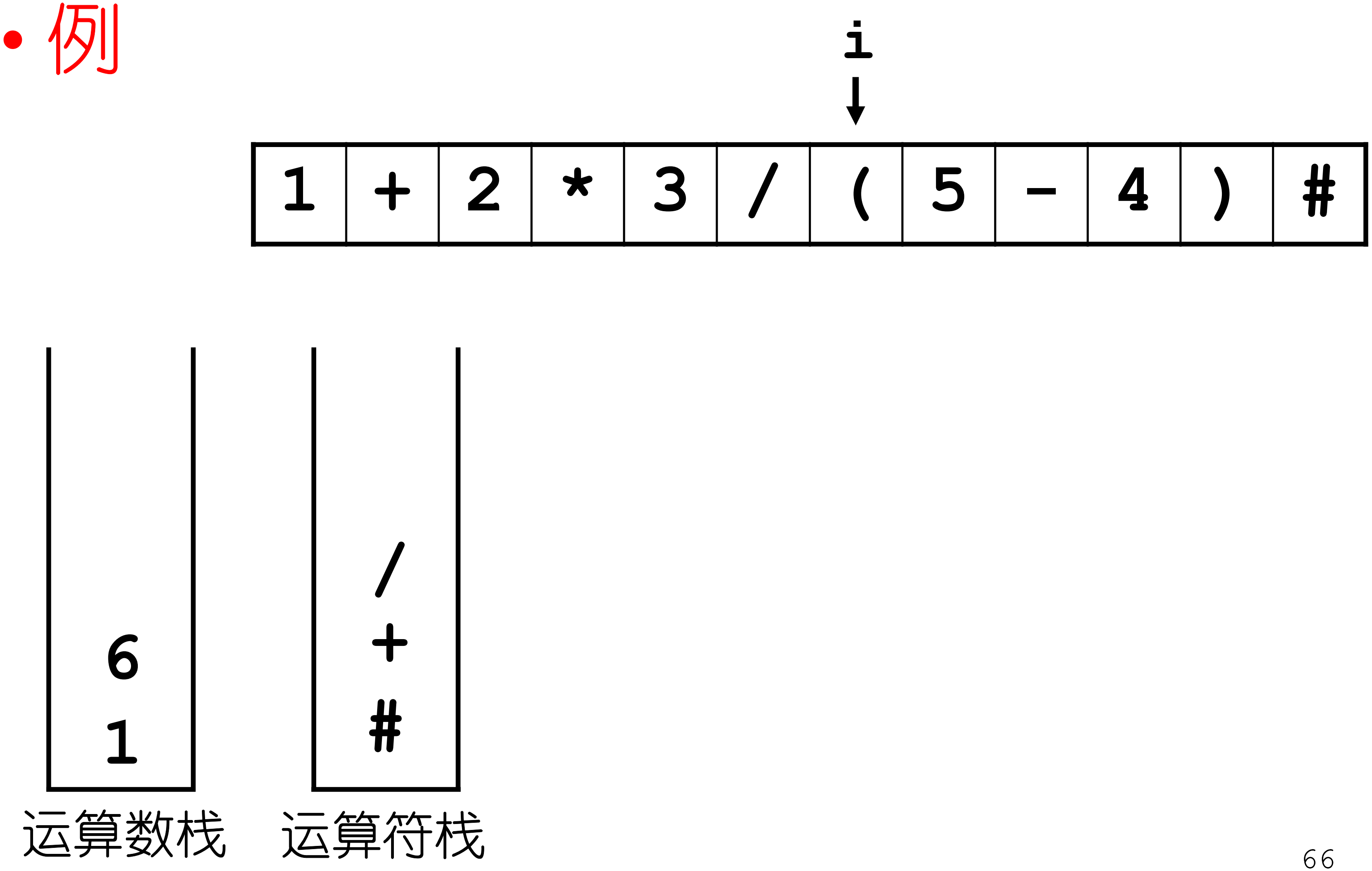


1	+	2	*	3	/	(	5	-	4	)	#

运算数栈　运算符栈

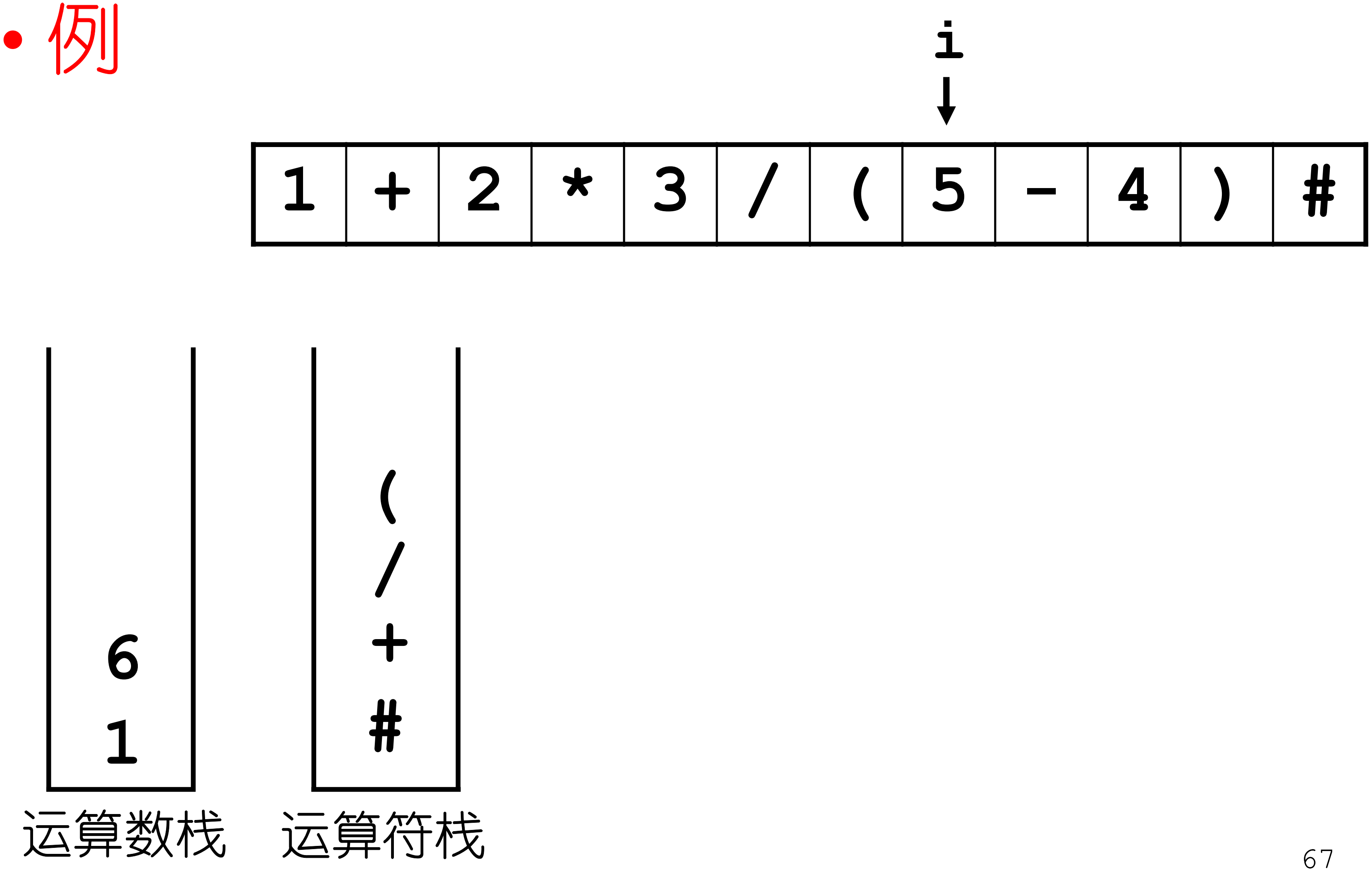
表达式求值：#3*(2+4)-8/2#的计算

- 例



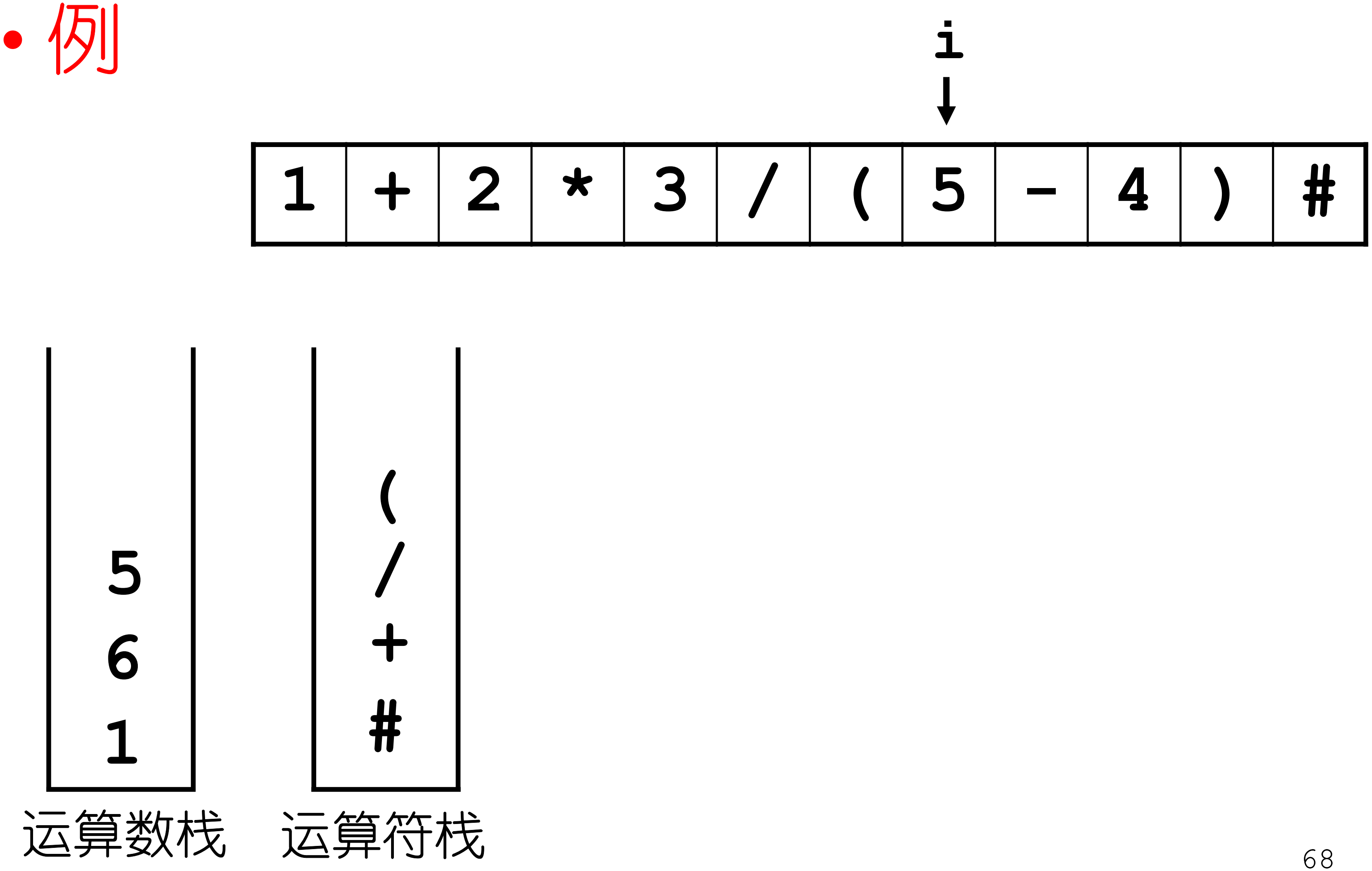
表达式求值：#3*(2+4)-8/2#的计算

- 例



表达式求值：#3*(2+4)-8/2#的计算

- 例



i

1	+	2	*	3	/	(	5	-	4	)	#

5
6
1

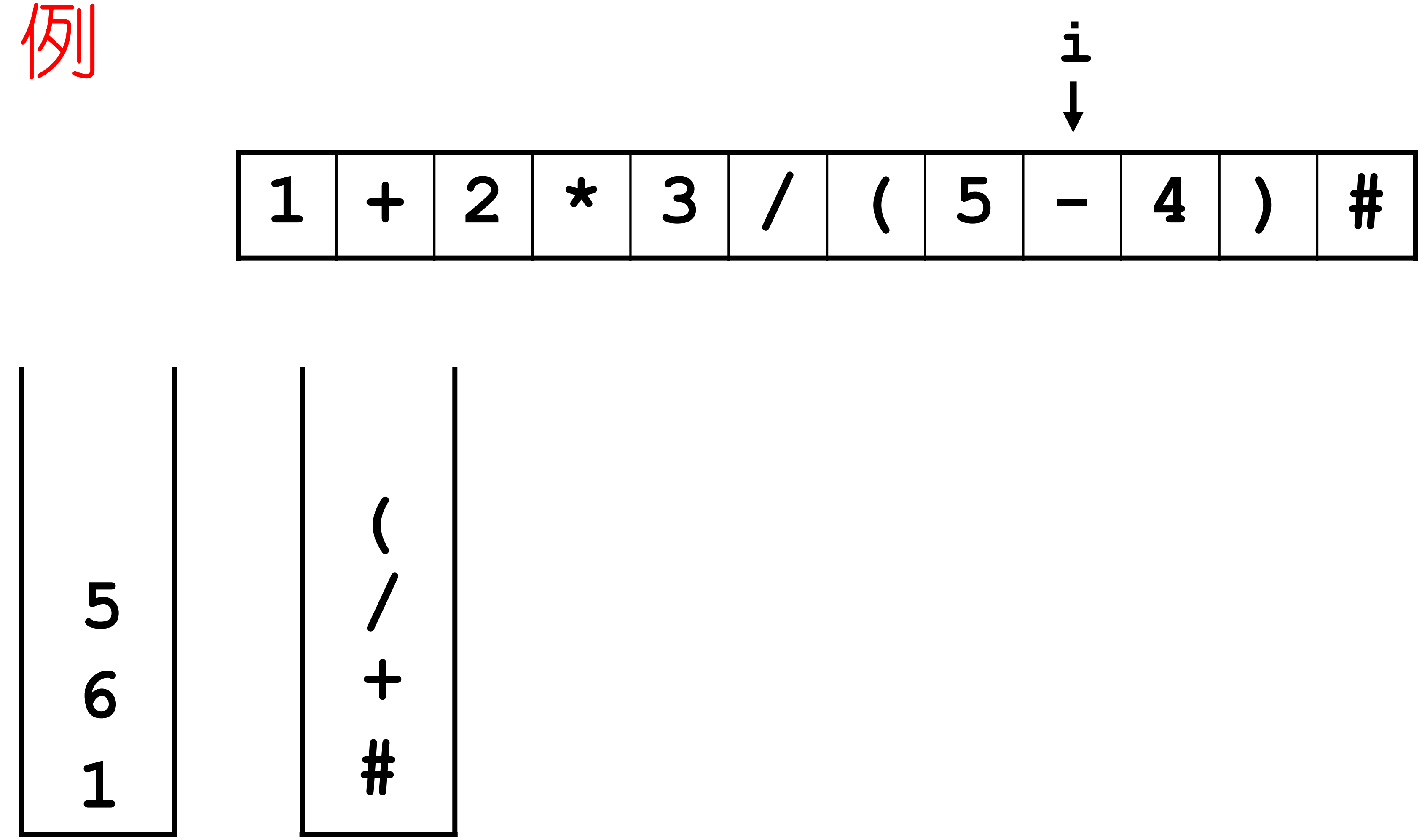
运算数栈

(
/
+
#

运算符栈

表达式求值：#3*(2+4)-8/2#的计算

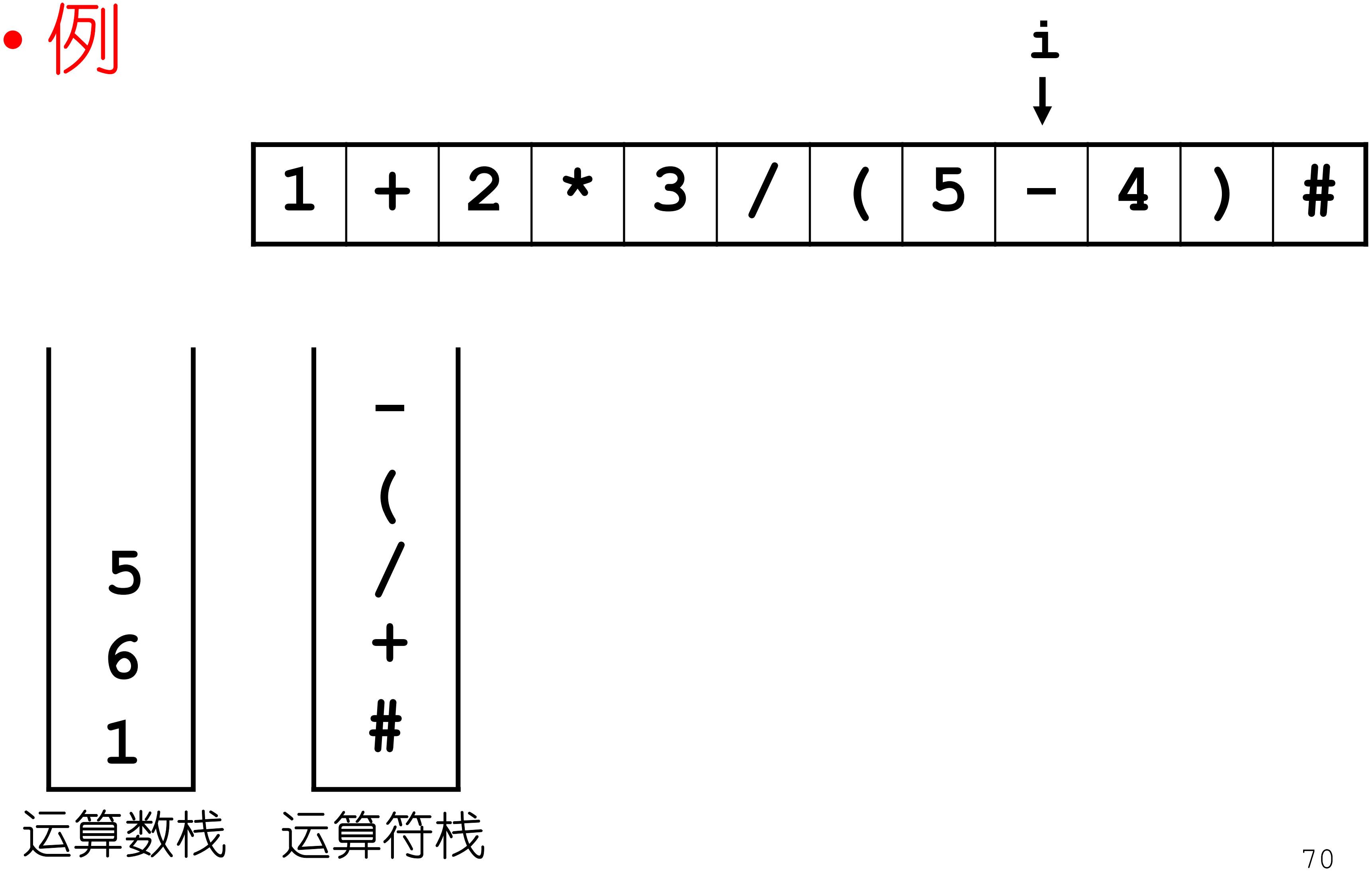
- 例



运算数栈　运算符栈

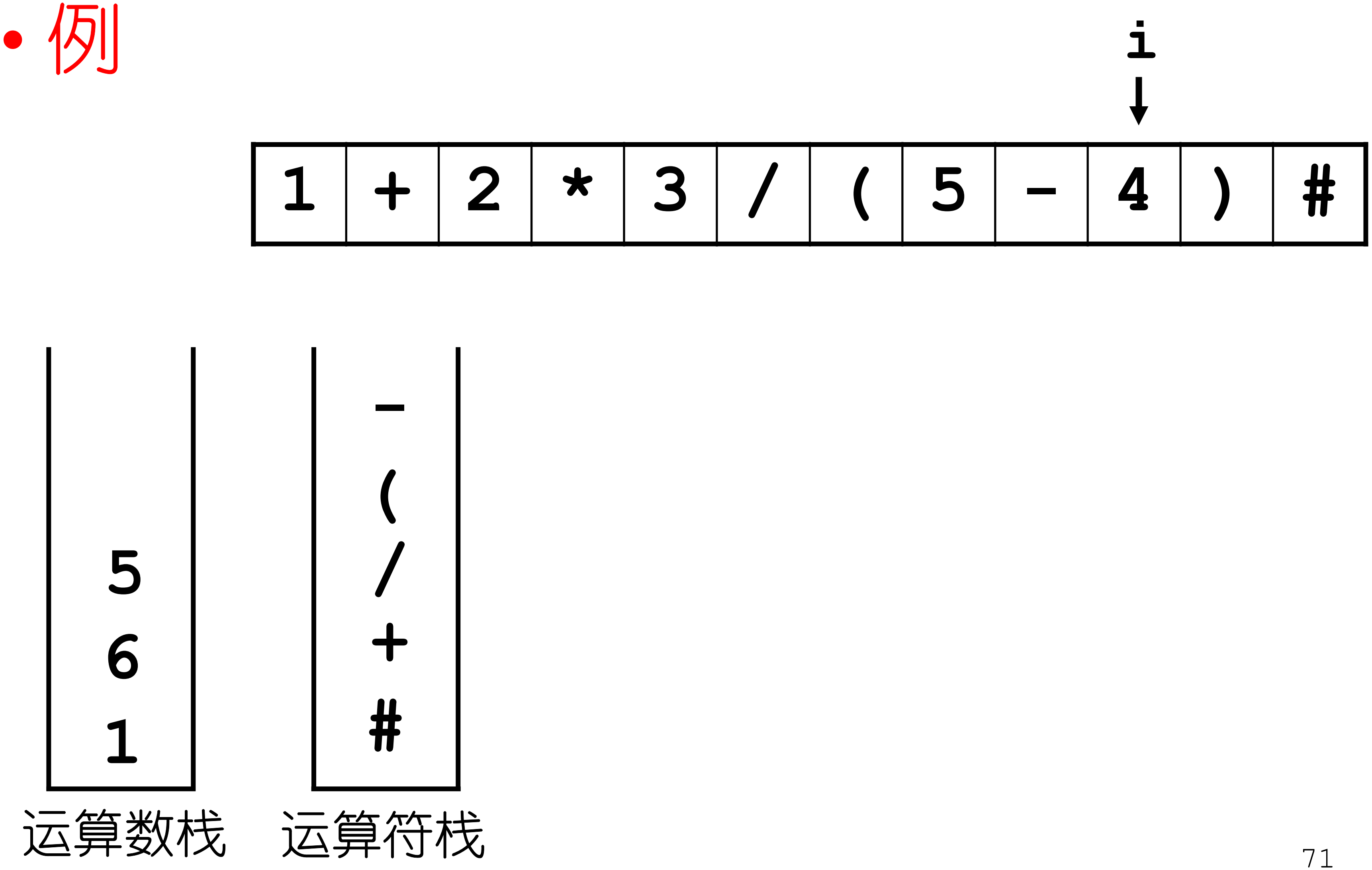
表达式求值：#3*(2+4)-8/2#的计算

- 例



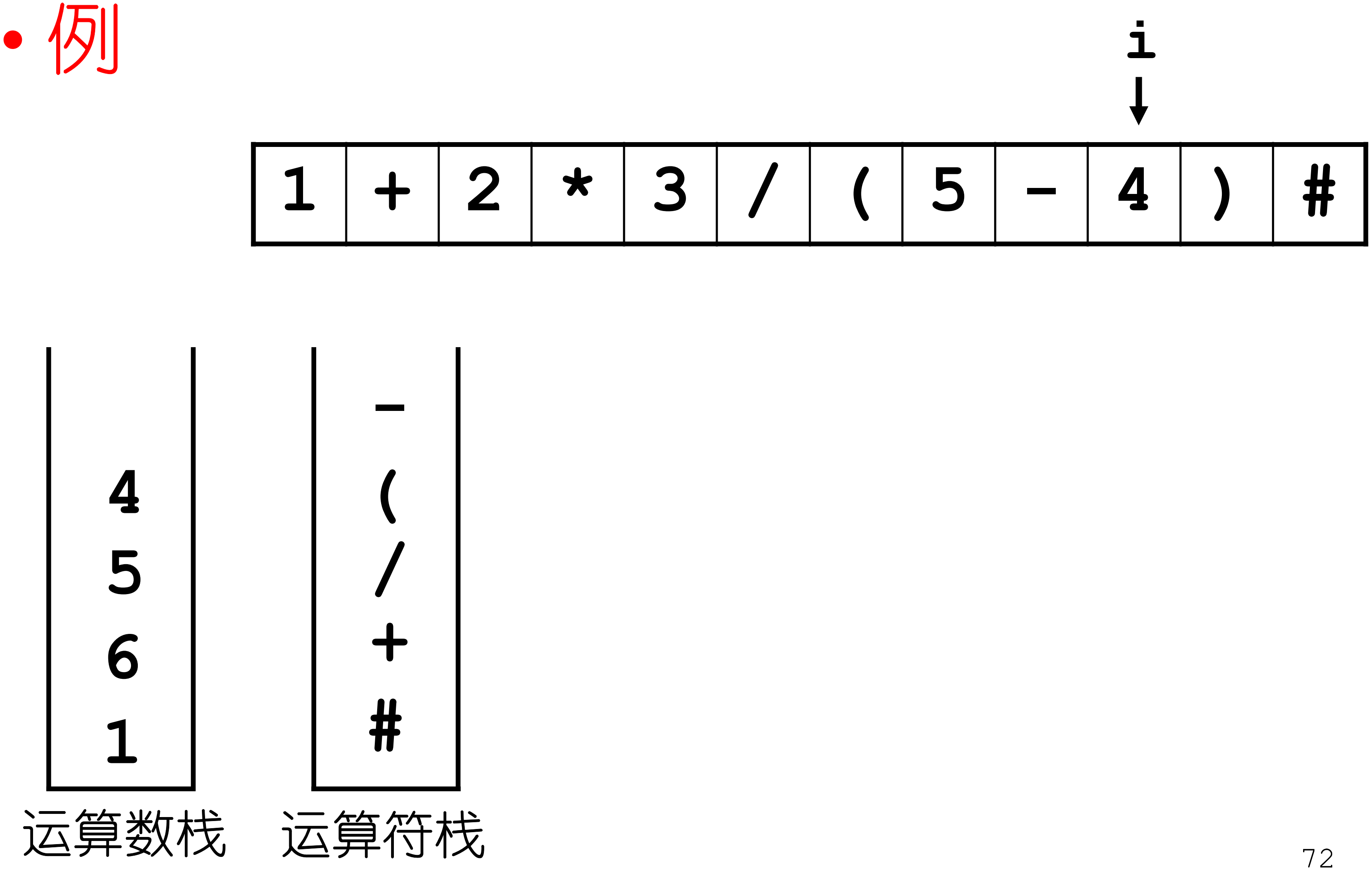
表达式求值：#3*(2+4)-8/2#的计算

- 例



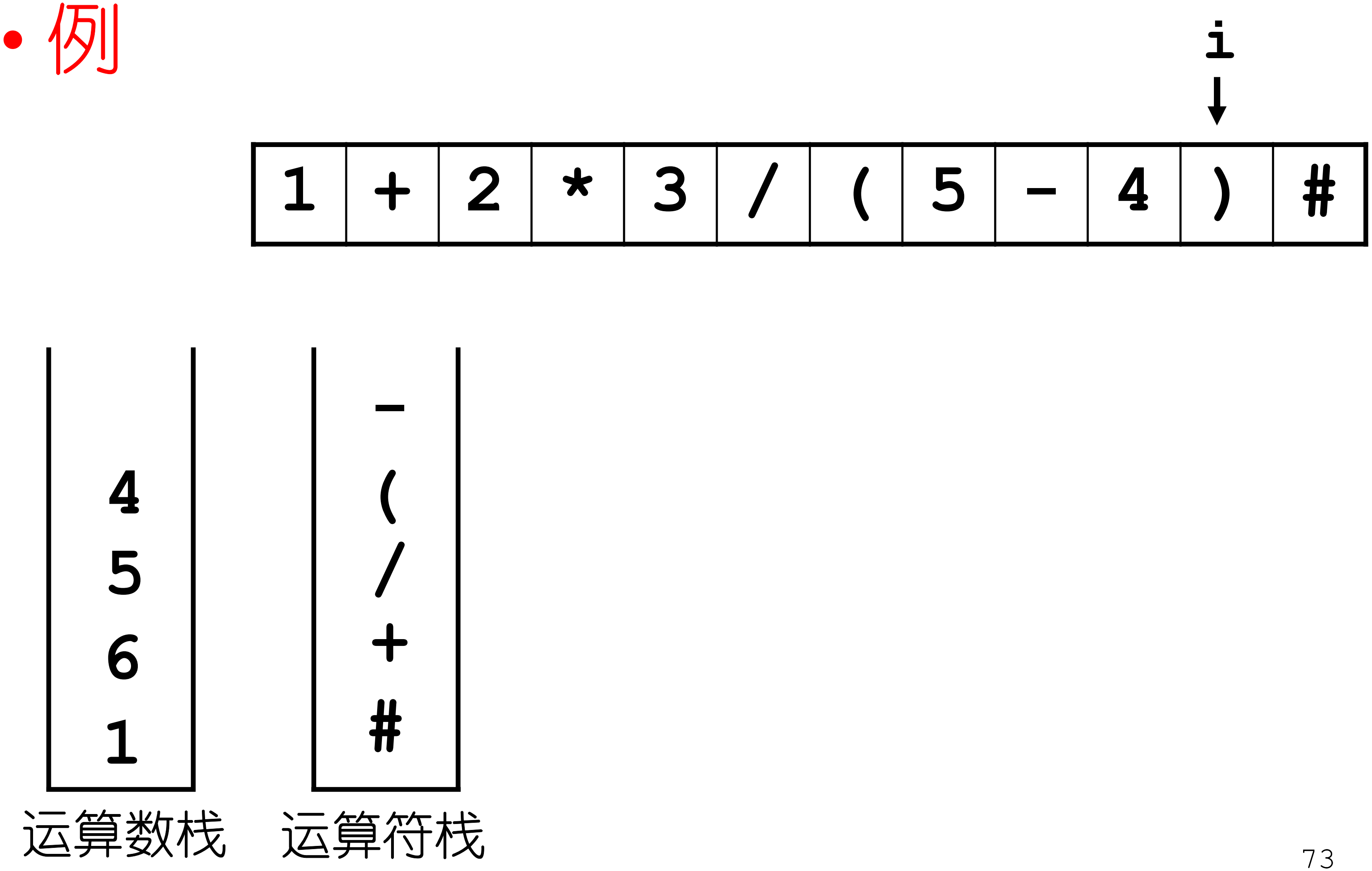
表达式求值：#3*(2+4)-8/2#的计算

- 例



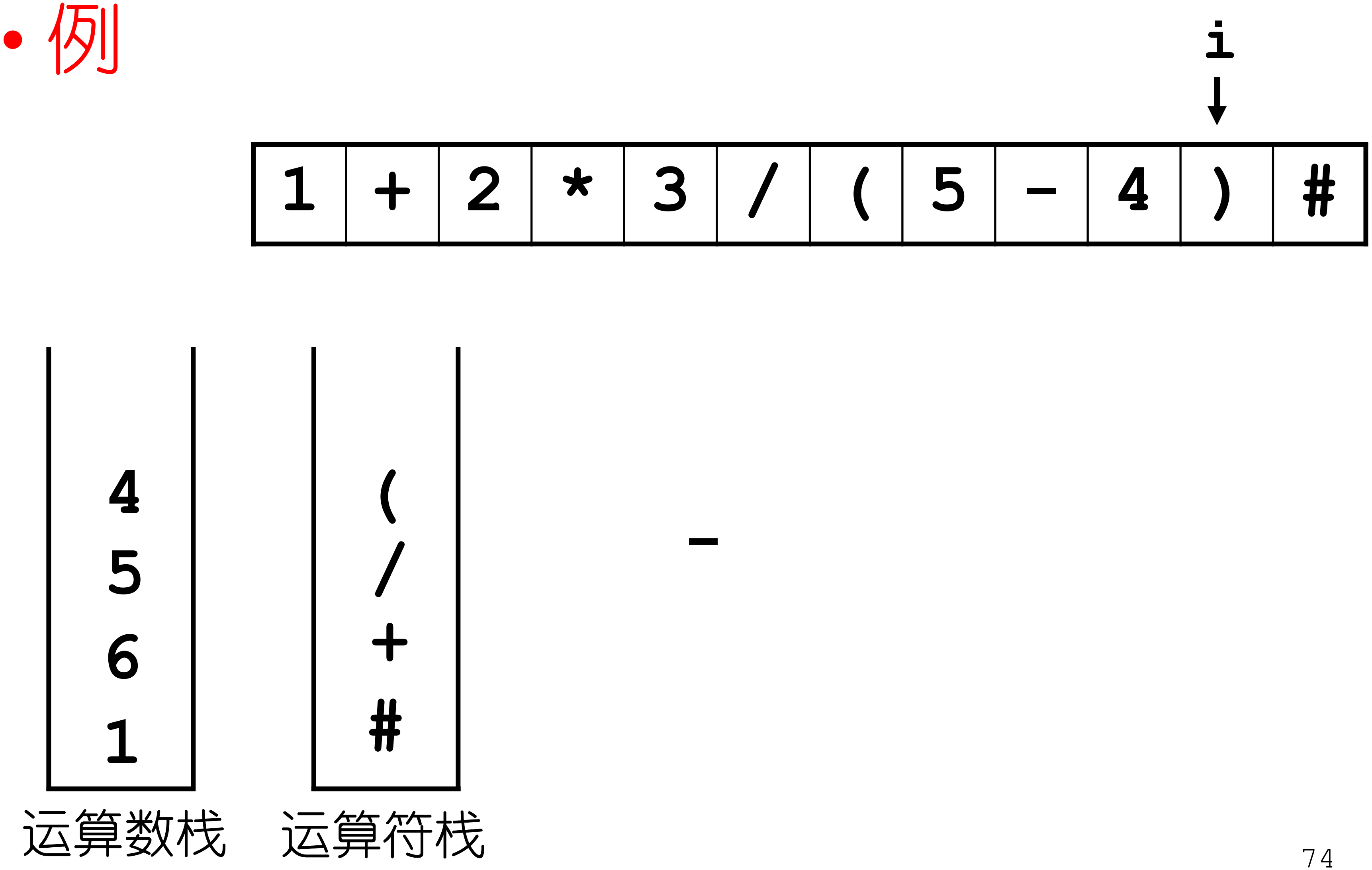
表达式求值：#3*(2+4)-8/2#的计算

- 例



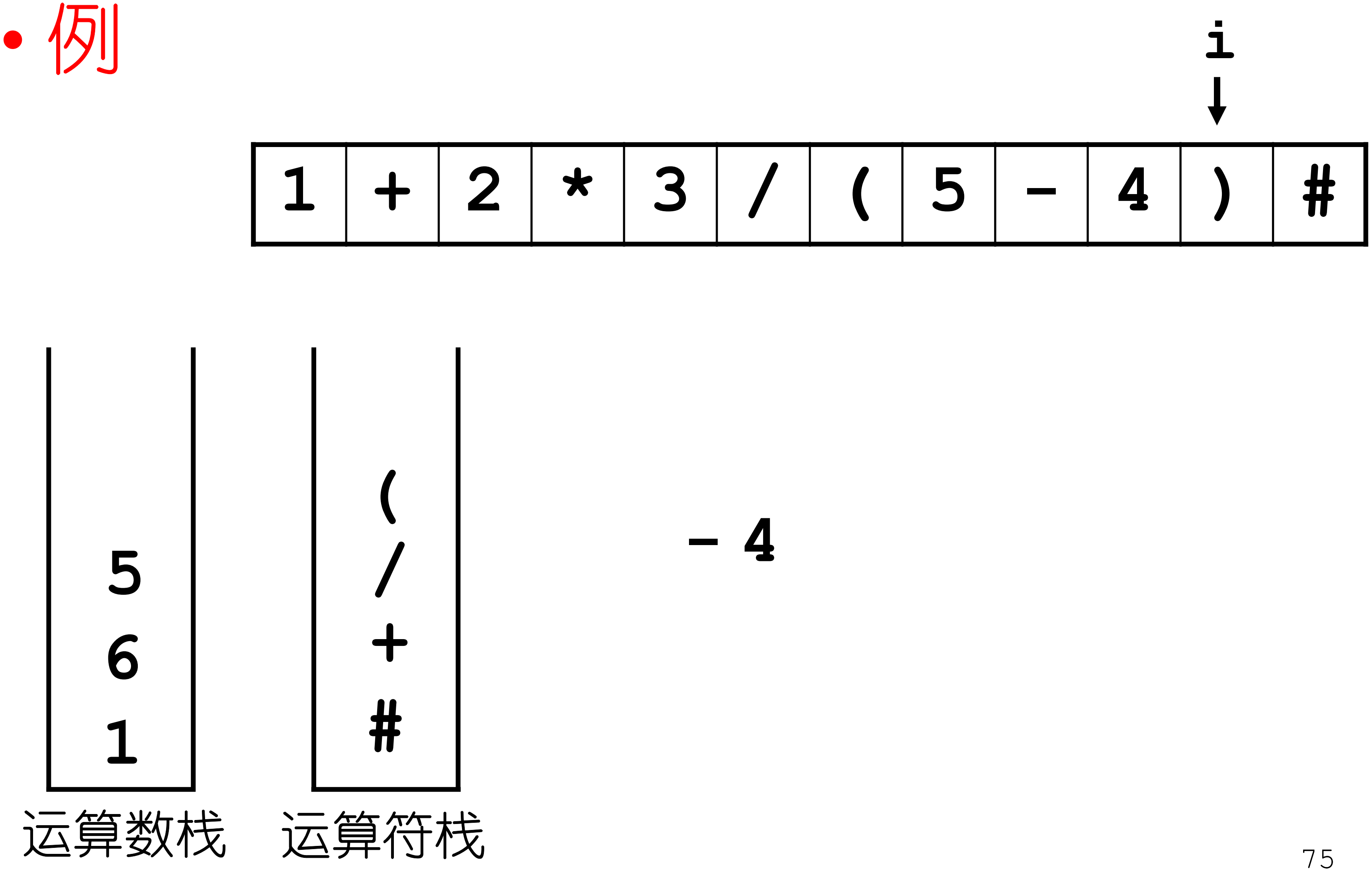
表达式求值：#3*(2+4)-8/2#的计算

- 例



表达式求值：#3*(2+4)-8/2#的计算

- 例



i

1	+	2	*	3	/	(	5	-	4	)	#

5
6
1

运算数栈

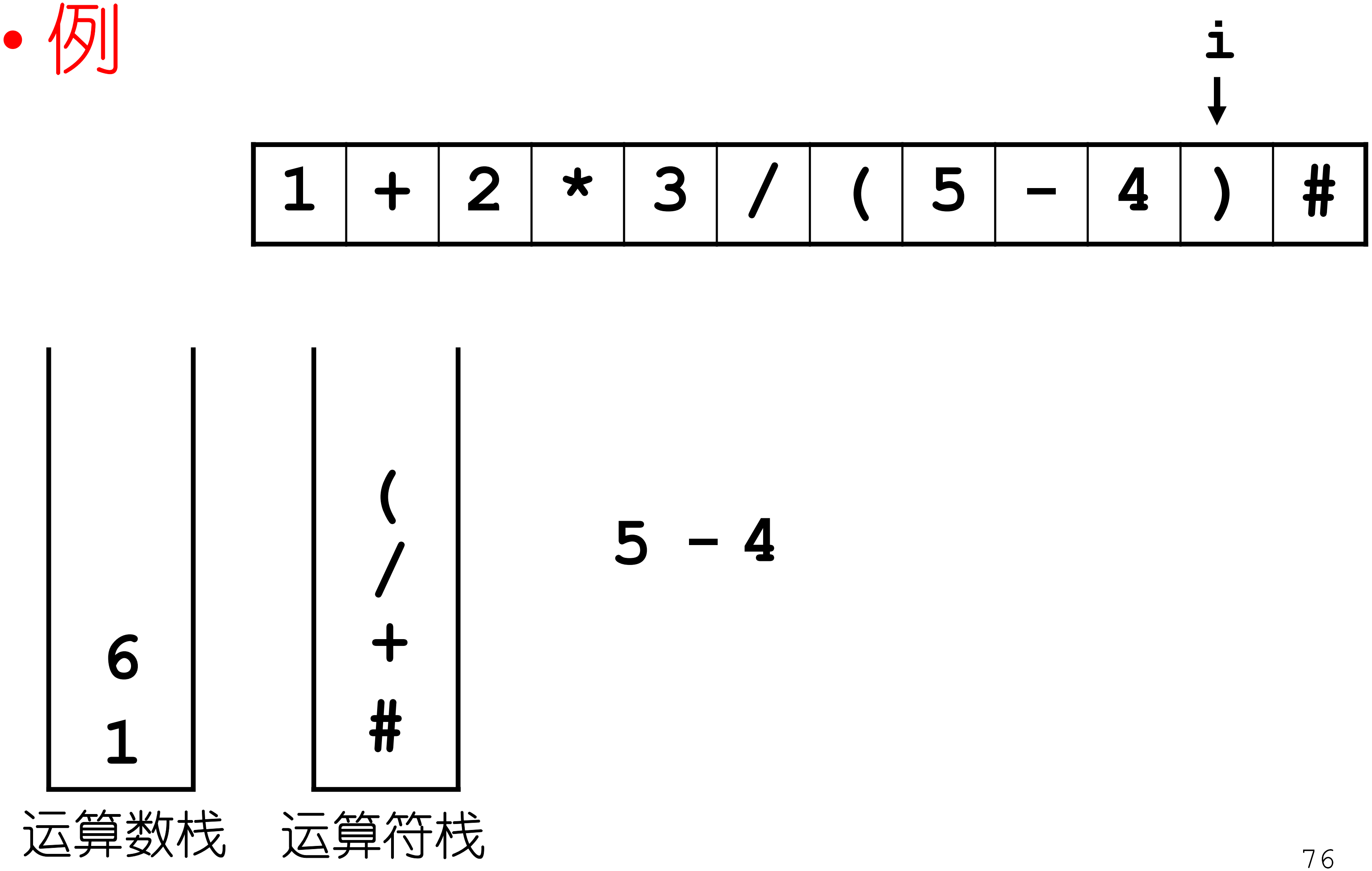
(
/
+
#

运算符栈

- 4

表达式求值：#3*(2+4)-8/2#的计算

- 例

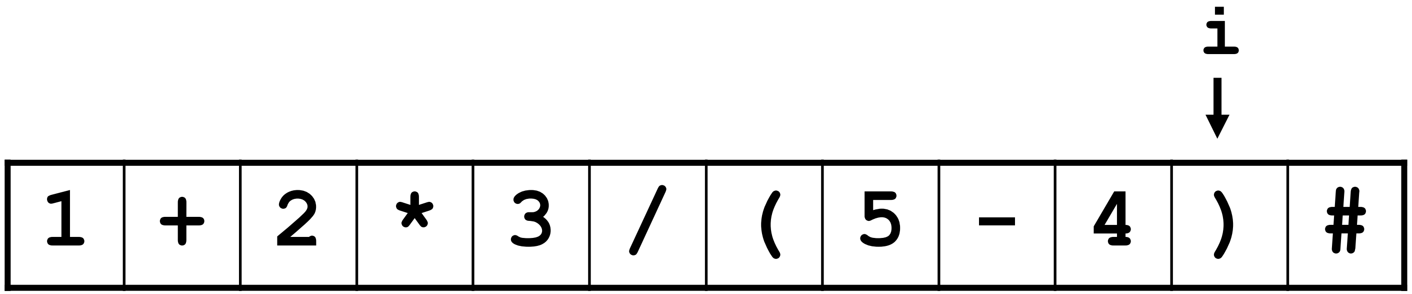


1	+	2	*	3	/	(	5	-	4	)	#

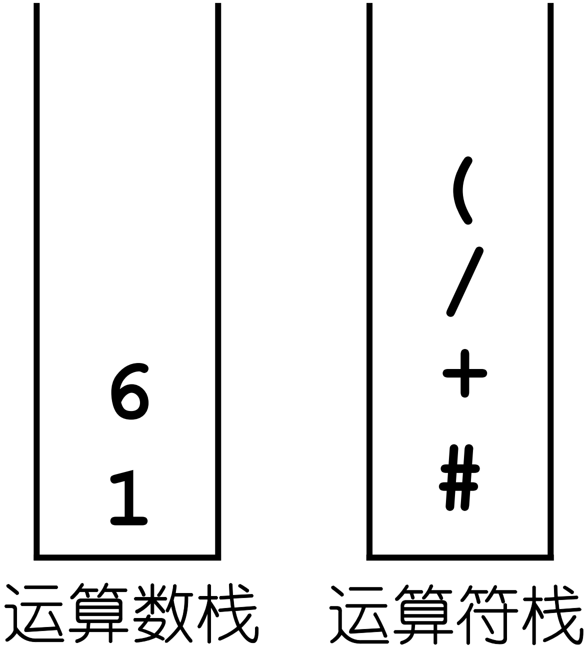
运算数栈　运算符栈

表达式求值：#3*(2+4)-8/2#的计算

- 例



1	+	2	*	3	/	(	5	-	4	)	#

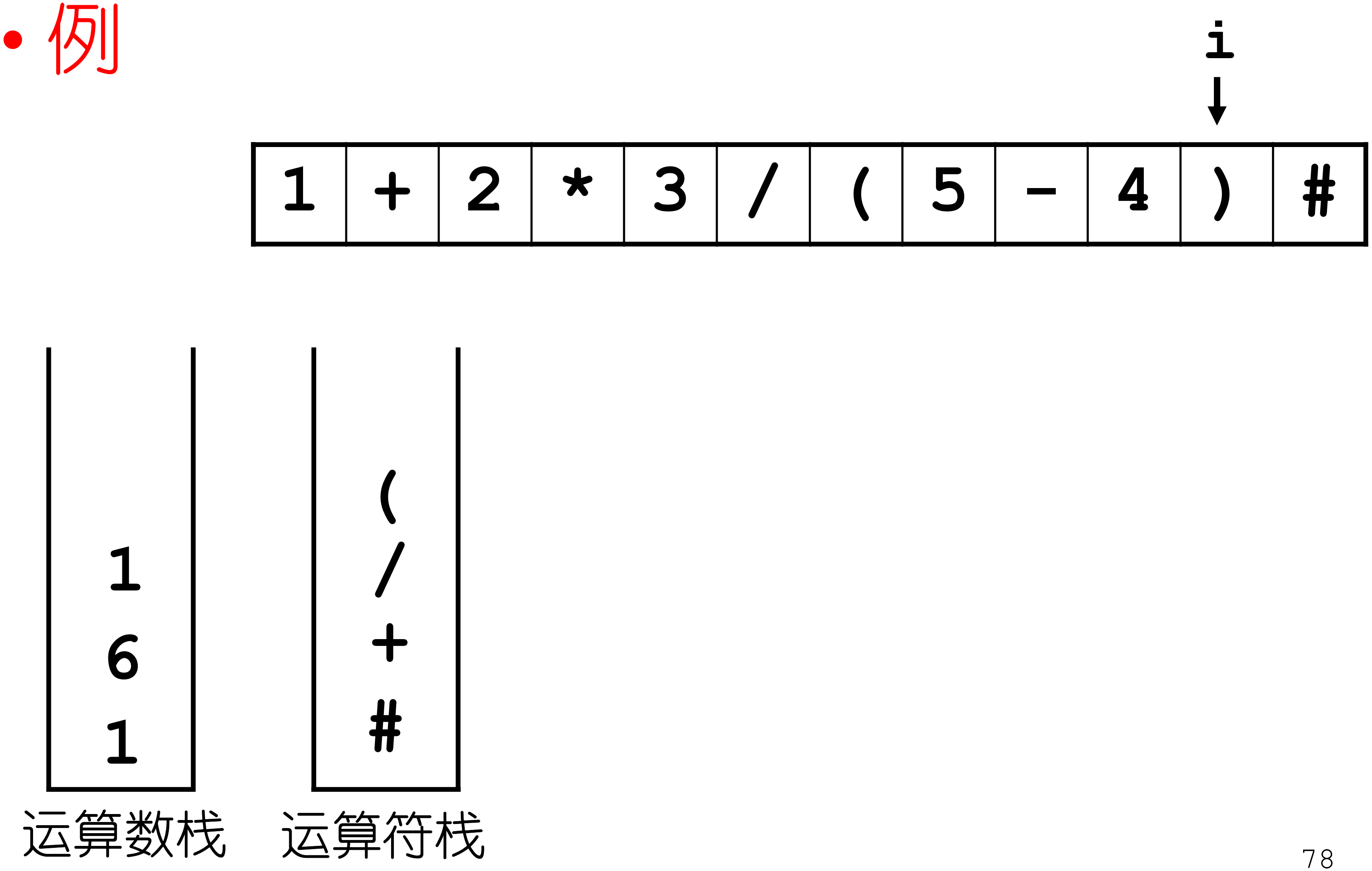


运算数栈　运算符栈

5 - 4 = 1

表达式求值：#3*(2+4)-8/2#的计算

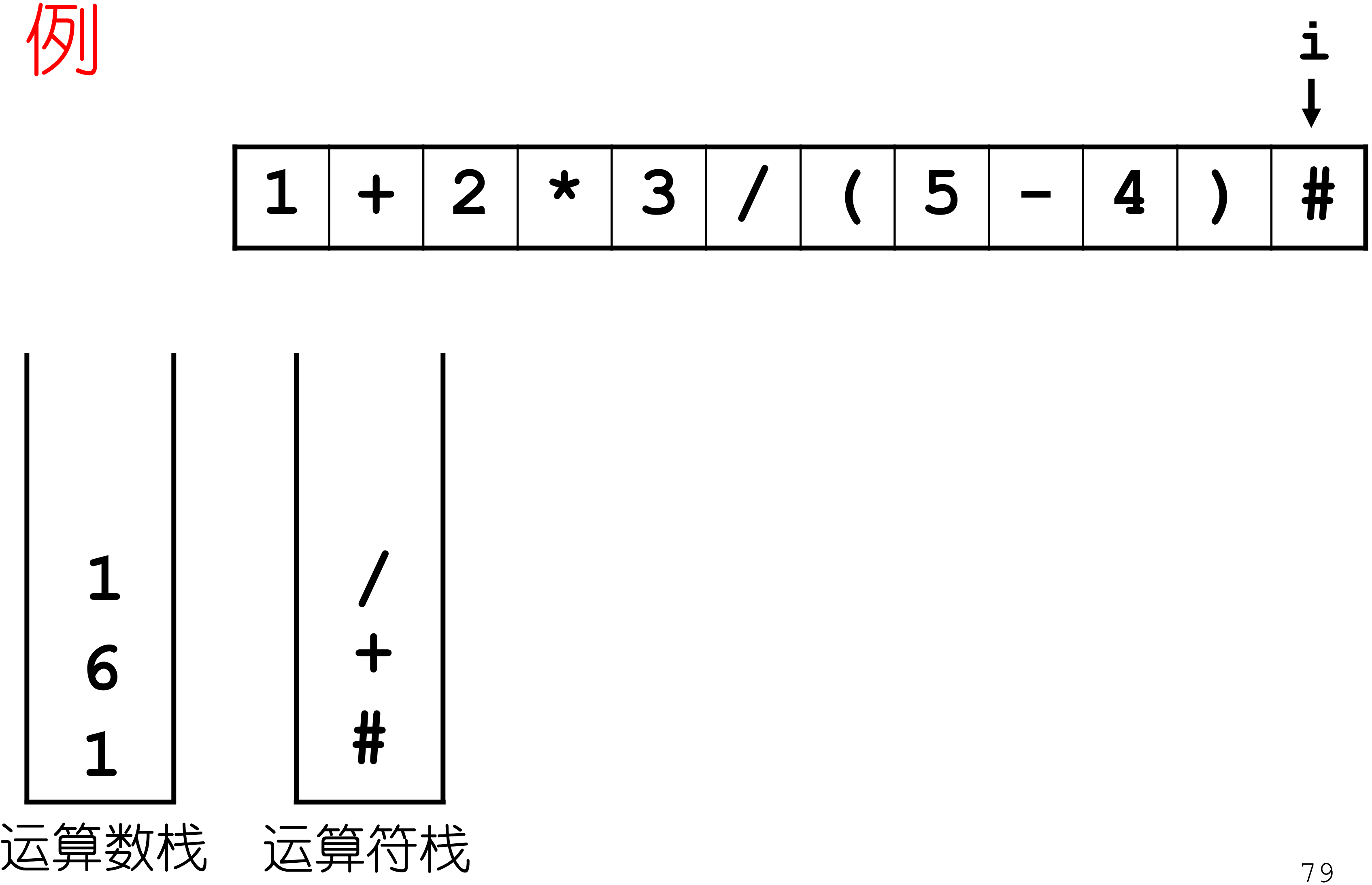
- 例



1	+	2	*	3	/	(	5	-	4	)	#

表达式求值：#3*(2+4)-8/2#的计算

- 例



i

1	+	2	*	3	/	(	5	-	4	)	#

1
6
1

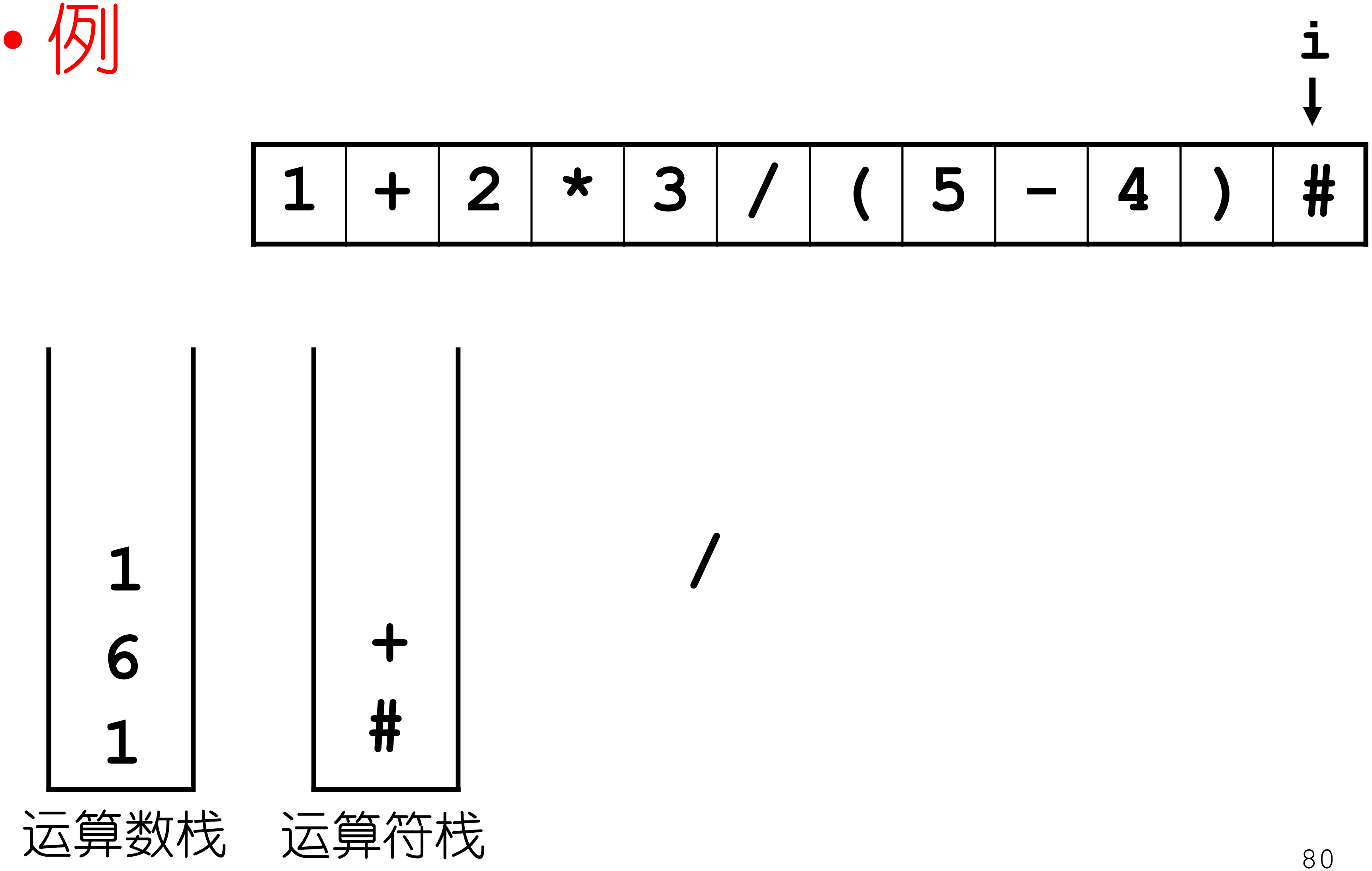
运算数栈

/
+
#

运算符栈

表达式求值：#3*(2+4)-8/2#的计算

- 例

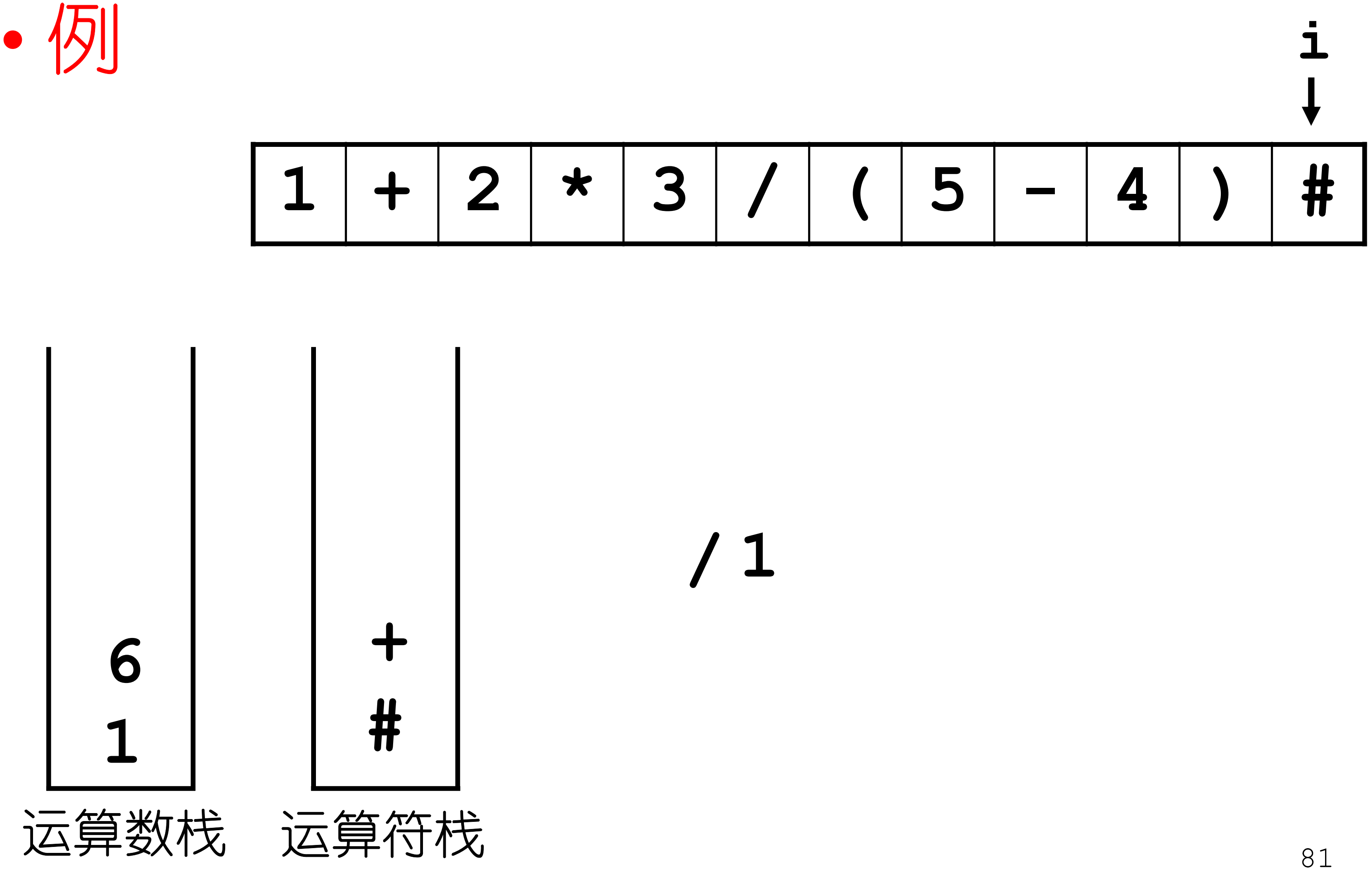


i

1	+	2	*	3	/	(	5	-	4	)	#

表达式求值：#3*(2+4)-8/2#的计算

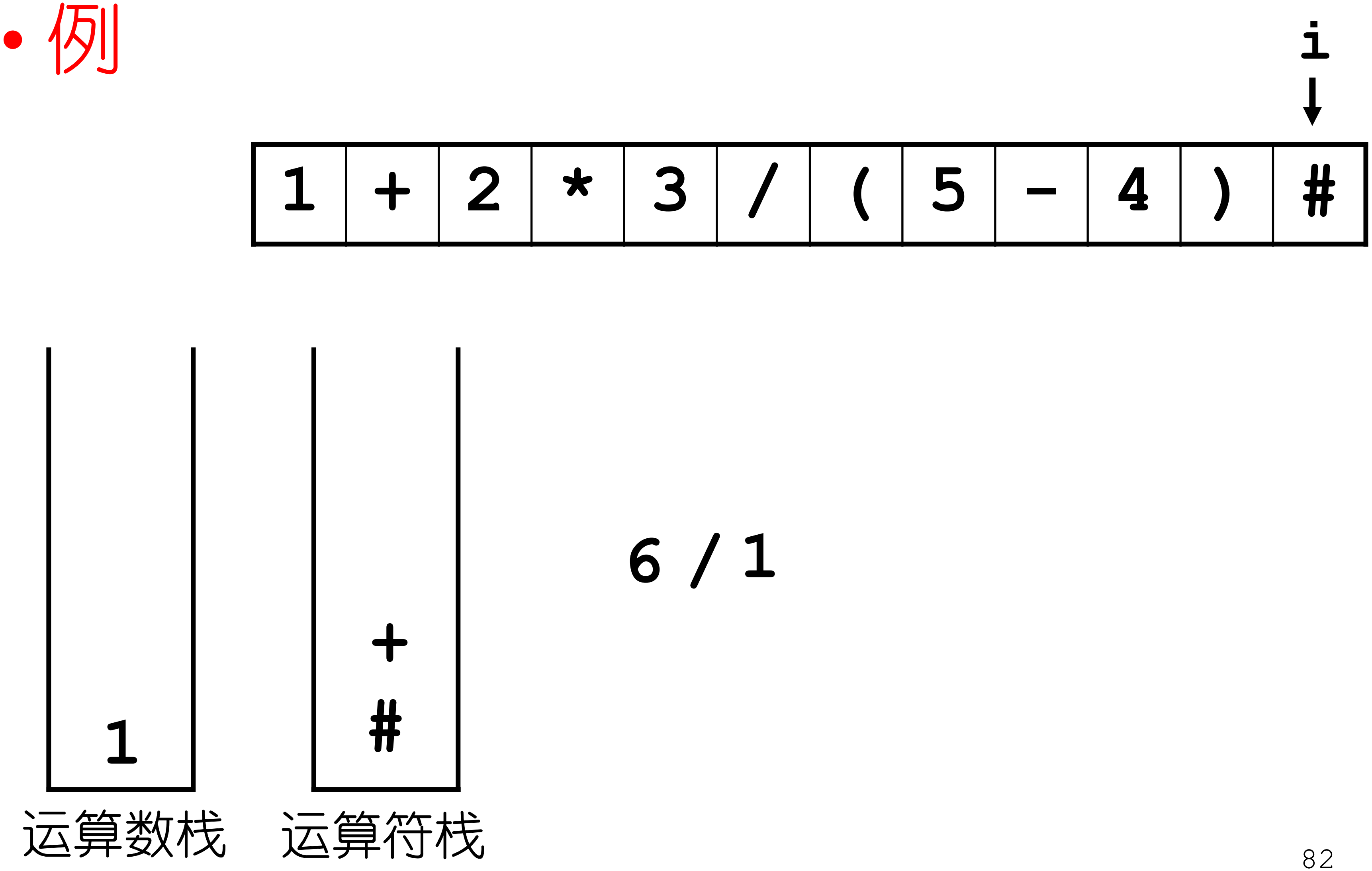
- 例



1	+	2	*	3	/	(	5	-	4	)	#

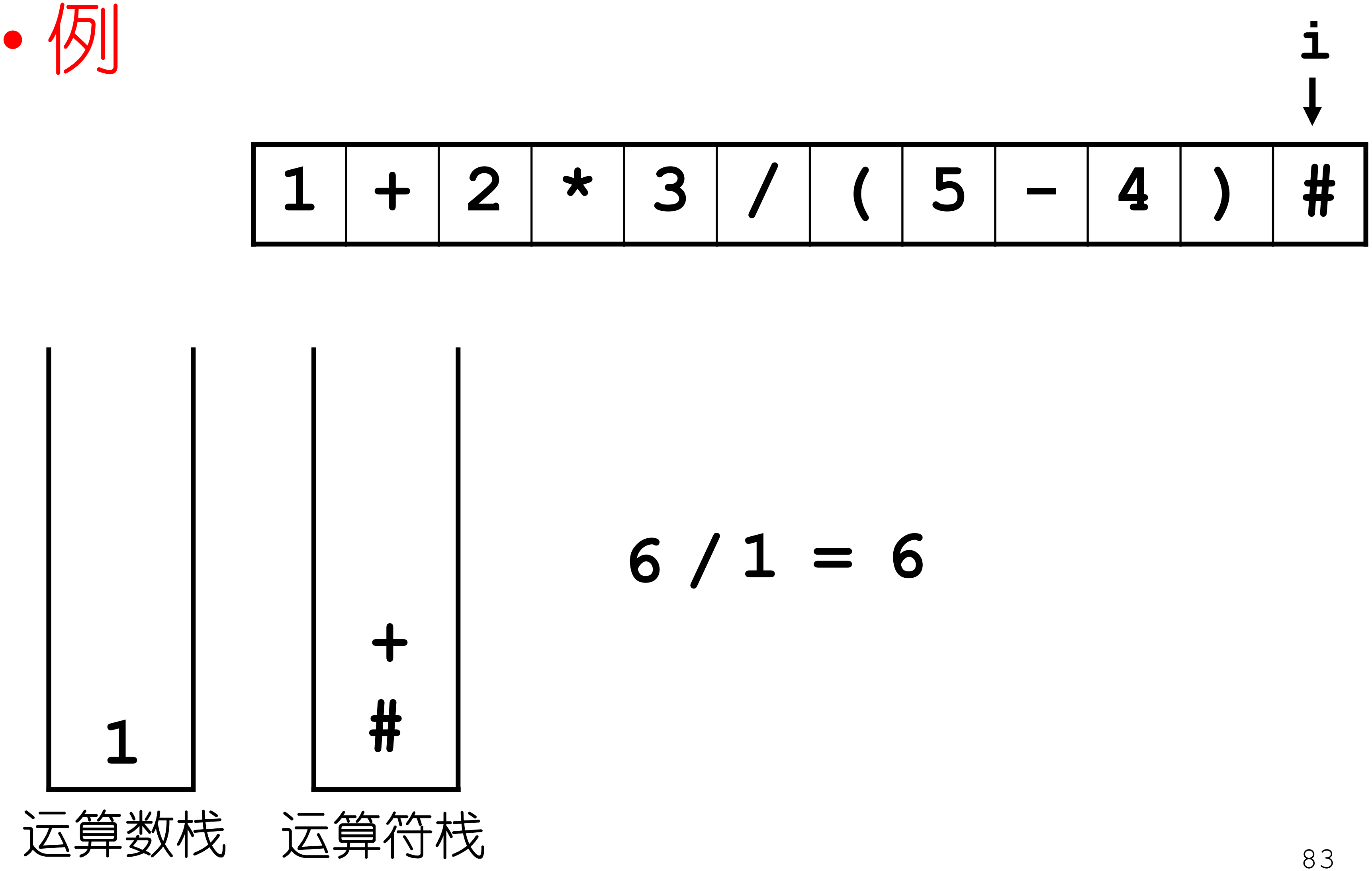
表达式求值：#3*(2+4)-8/2#的计算

- 例



表达式求值：#3*(2+4)-8/2#的计算

- 例

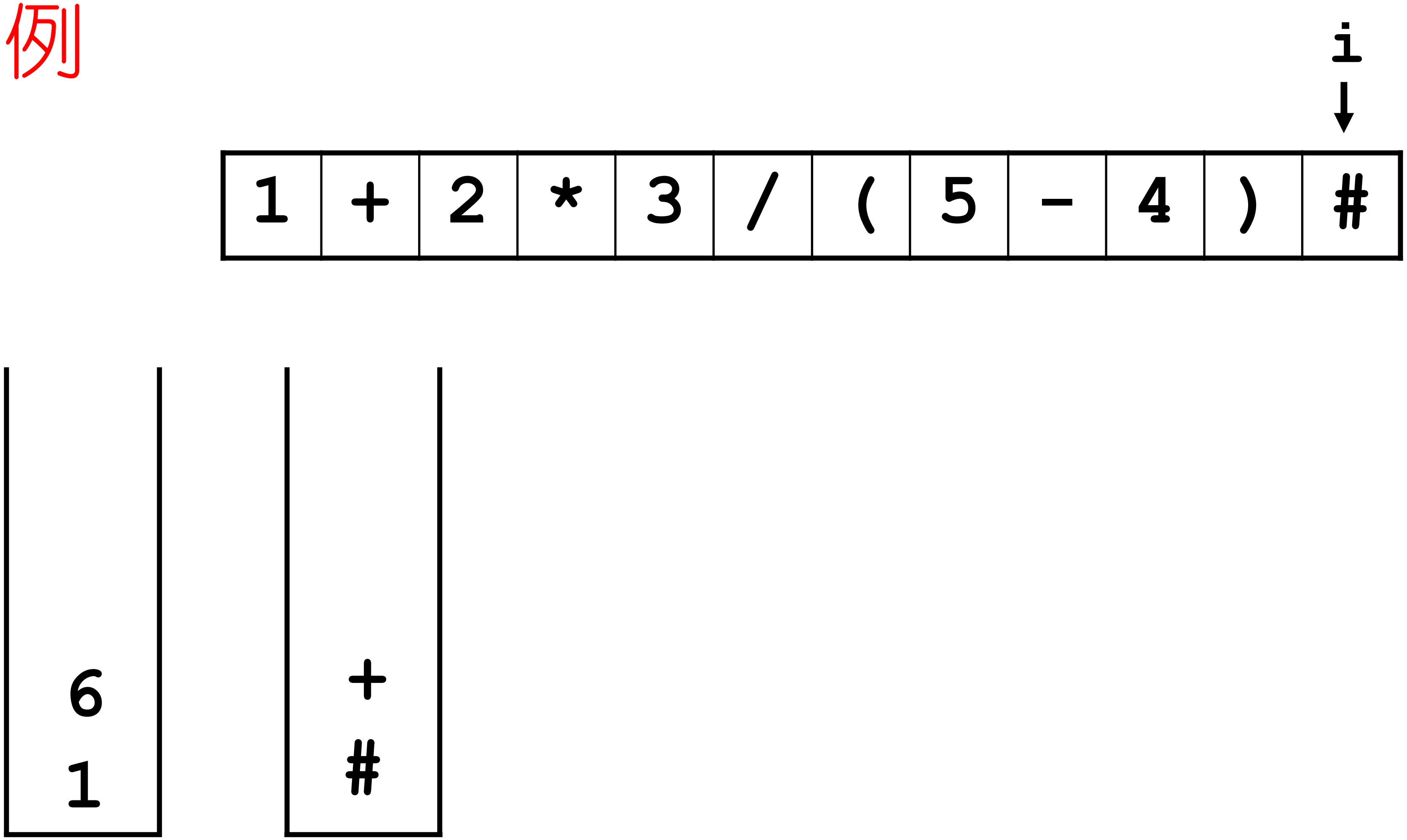


1	+	2	*	3	/	(	5	-	4	)	#

6 / 1 = 6

表达式求值：#3*(2+4)-8/2#的计算

- 例



i

1	+	2	*	3	/	(	5	-	4	)	#

6

1

运算数栈

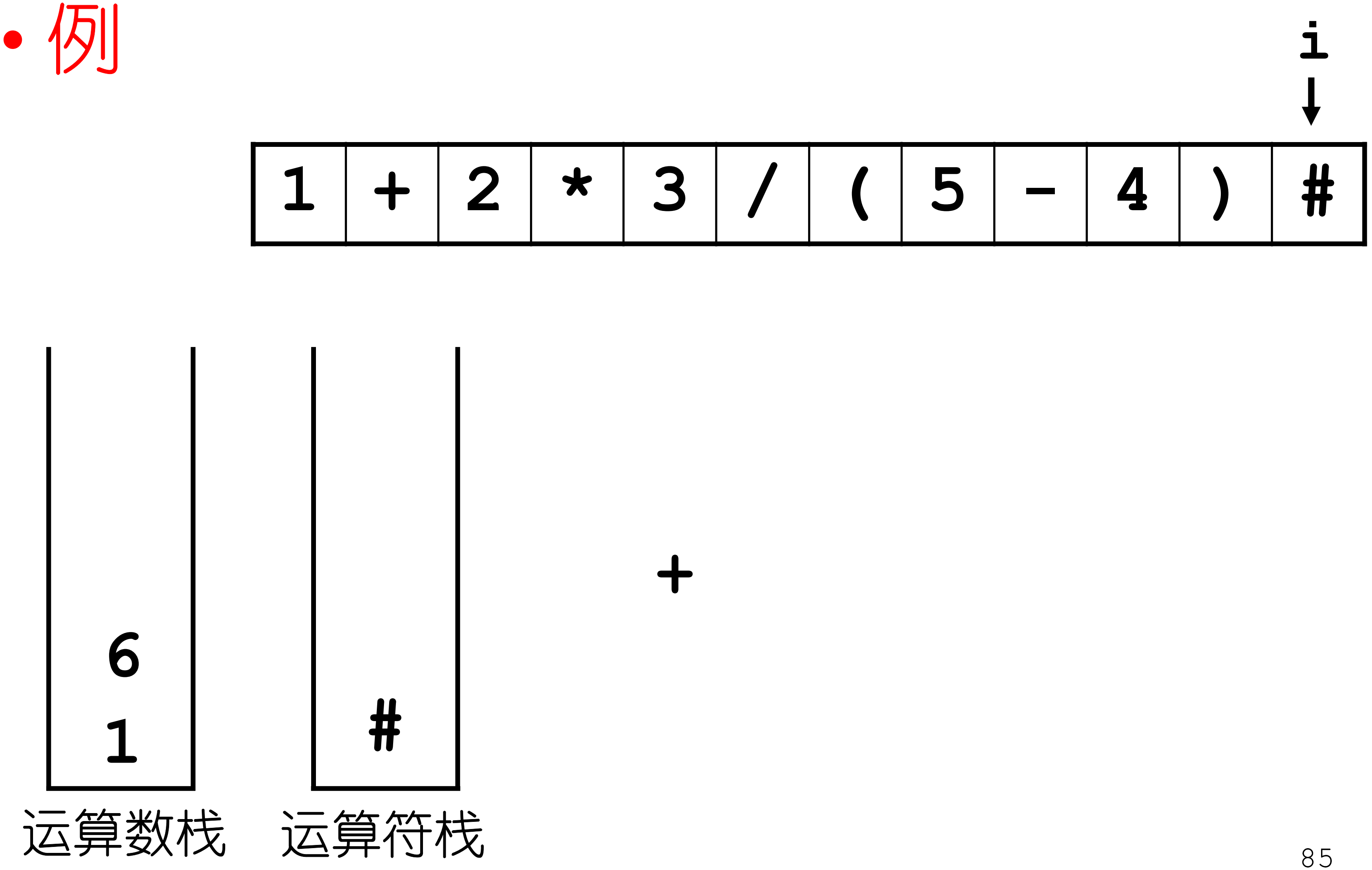
+

#

运算符栈

表达式求值：#3*(2+4)-8/2#的计算

- 例



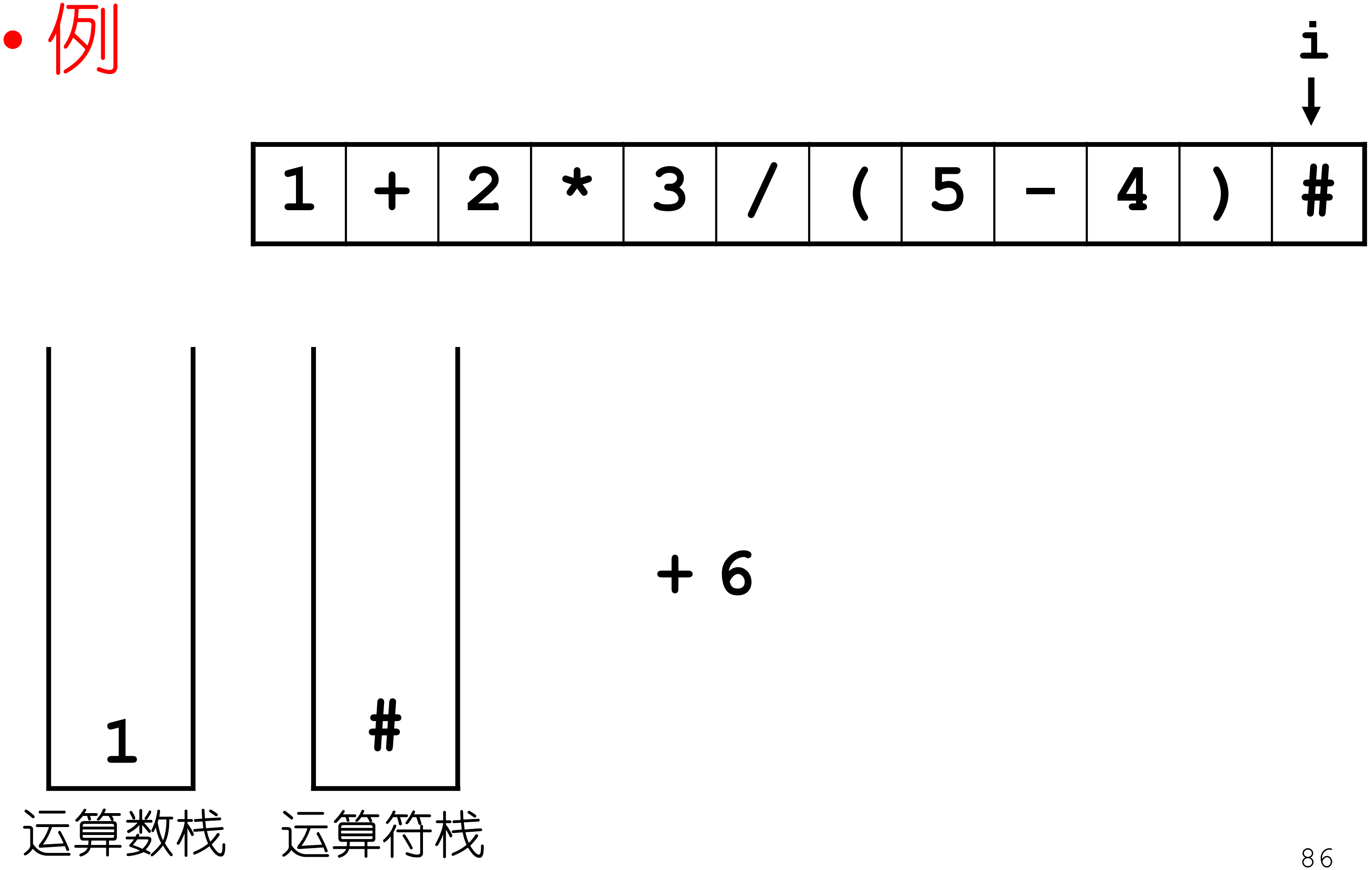
i

1	+	2	*	3	/	(	5	-	4	)	#

运算数栈　运算符栈

表达式求值：#3(2+4)-8/2#的计算*

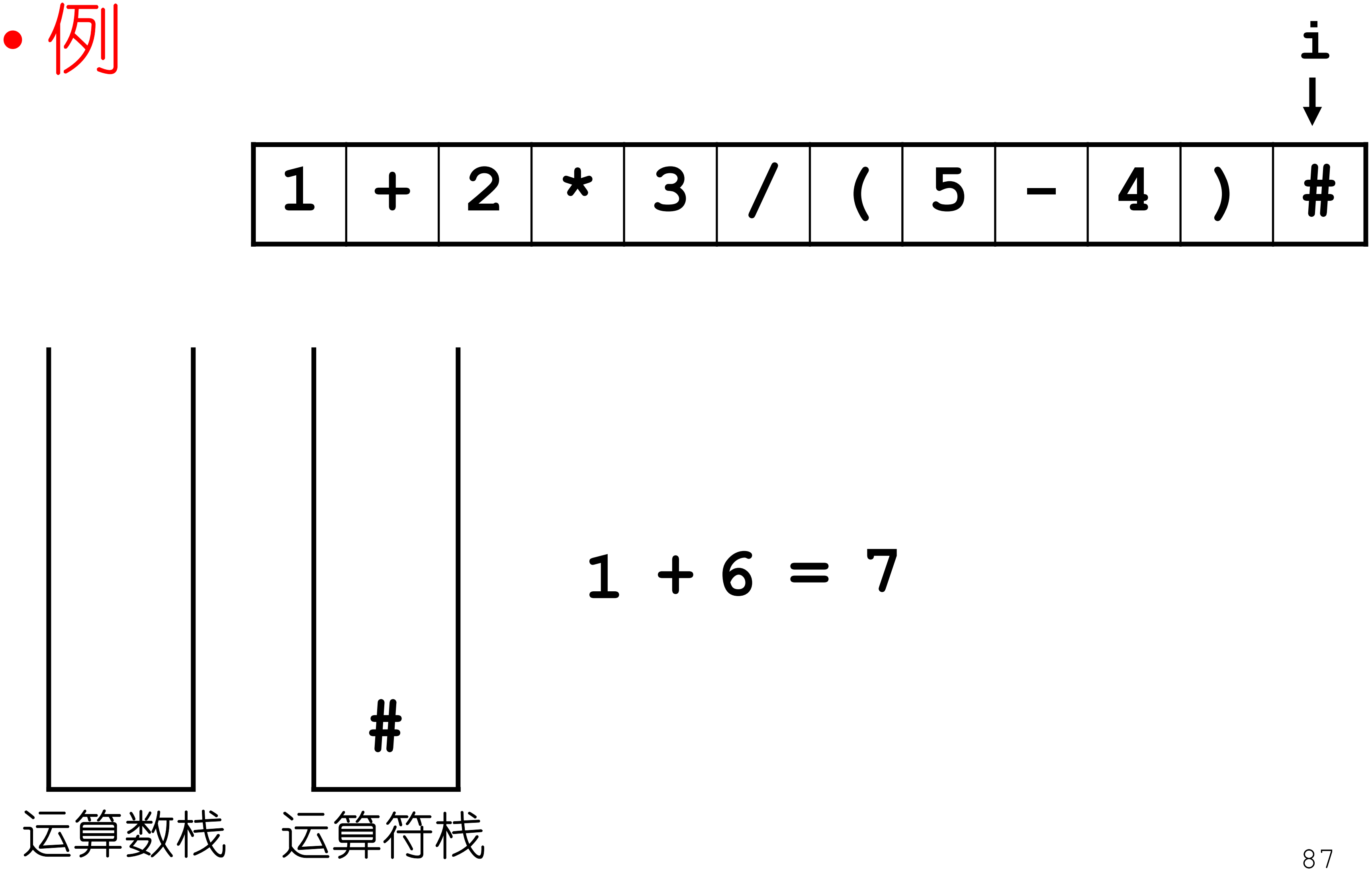
- 例



1	+	2	*	3	/	(	5	-	4	)	#

表达式求值：#3*(2+4)-8/2#的计算

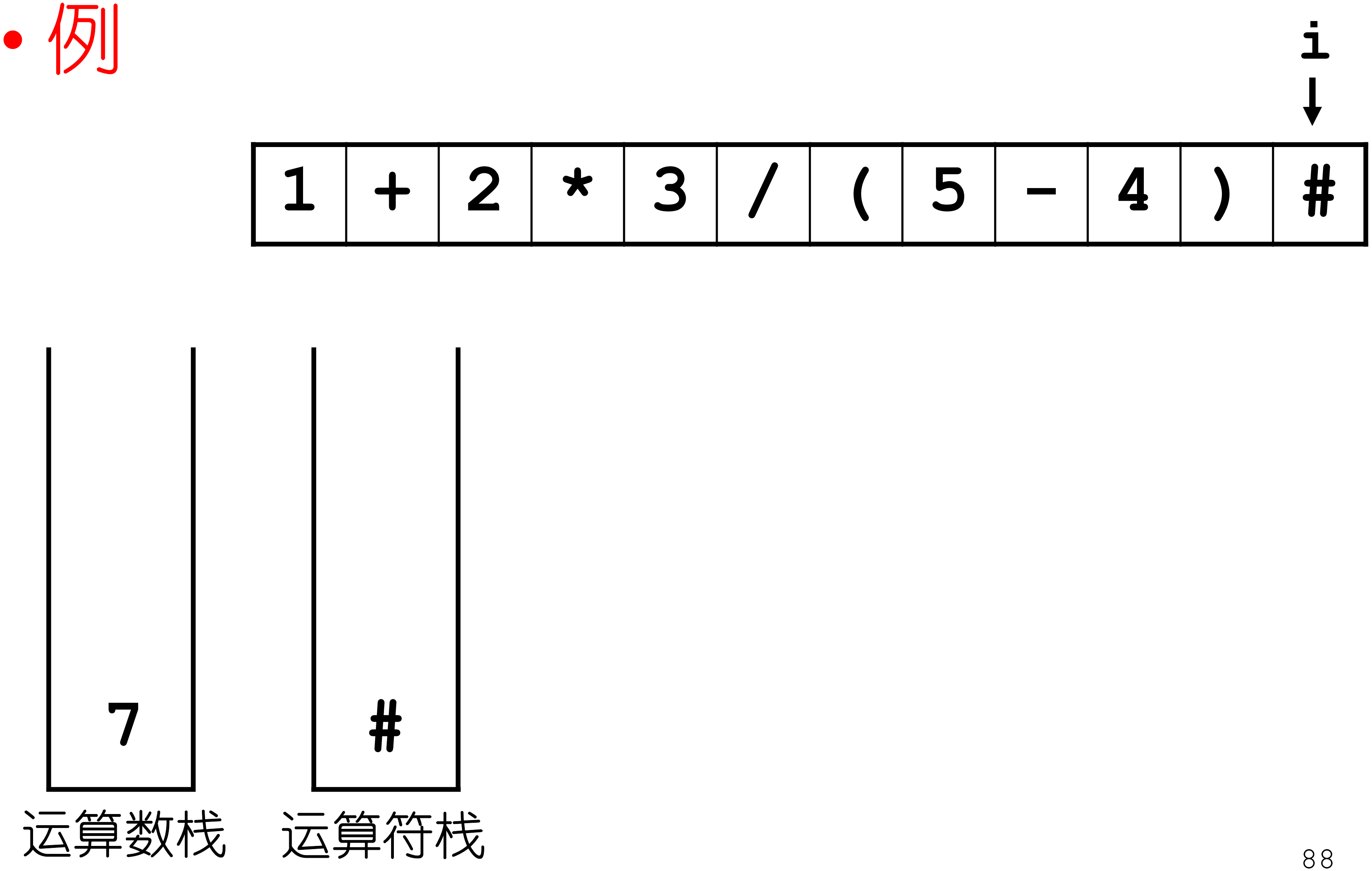
- 例



1 + 6 = 7

表达式求值：#3*(2+4)-8/2#的计算

- 例



i

1	+	2	*	3	/	(	5	-	4	)	#

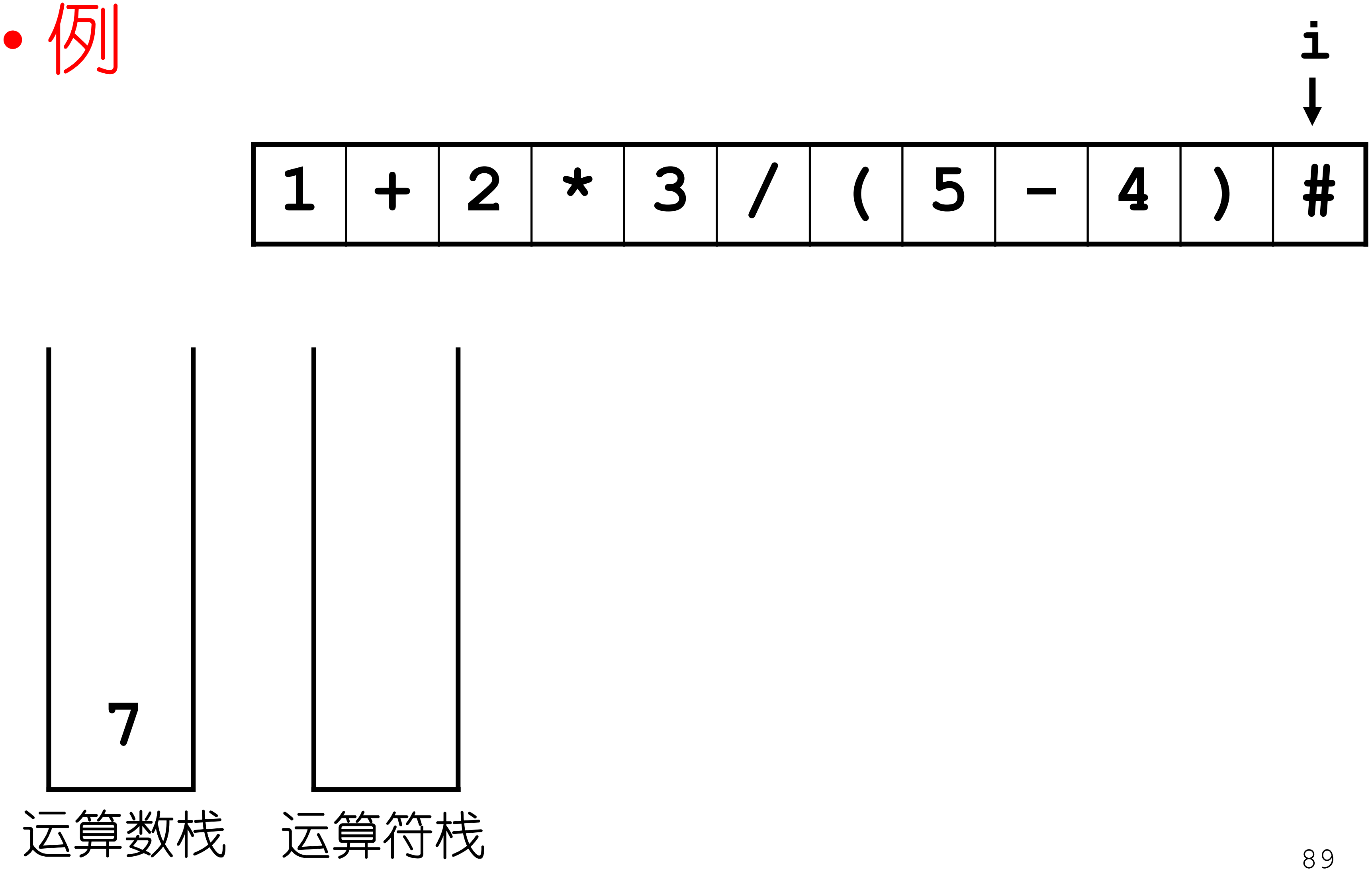
7

#

运算数栈　运算符栈

表达式求值：#3*(2+4)-8/2#的计算

- 例



i

1	+	2	*	3	/	(	5	-	4	)	#

7

运算数栈　运算符栈

表达式求值：算法过程

- 算法过程
  - 用两个栈分别存放运算符和运算数
  - 当前符号是运算数，直接入栈
  - 当前符号是运算符，且优先级更高，则入栈
  - 否则弹出栈顶元素运算，运算结果重新压入运算数栈

表达式求值：表达式的表示形式

- 表达式的表示
  - 中序表达式：运算符放在两个运算数中间
  - 前序表达式：运算符放在两个运算数之前
  - 后序表达式：运算符放在两个运算数之后
  - 例如：

中序	前序	后序
X+Y	**+XY**	**XY+**
X+Y*Z	**+X*YZ**	**XYZ*+**
(X+Y)*Z	***+XYZ**	**XY+Z***
a*(b/(c-d))+e	**+*a/b-cde**	**abcd-/*e+**

栈的应用：表达式的表示

- 前序表达式
  - 特点：
    - 运算符在运算数之前
    - 运算数顺序跟中序表达式相同
    - 运算符按照运算顺序的逆序排列

中序：$a \overset{3}{*} (b \overset{2}{/} (c \overset{1}{-} d)) \overset{4}{+} e$

前序：$+ * a / b - cde$

表达式求值：表达式的表示

- 后序表达式
  - 特点：
    - 运算符在运算数之后
    - 运算数顺序跟中序表达式相同
    - 运算符按照运算顺序排列

```
         3    2    1     4
中序： a * (b / (c - d)) + e

后序： abcd - / * e +
```

表达式求值：表达式的表示

- 手工计算方法：
  - 前序表达式
    - 取最后面的运算符
    - 取当前运算符后面的两个数作为运算数
    - 运算结果放到原来运算符的位置上
    - 循环，直到所有的运算符都运算完毕
  - 后序表达式？同学们自己总结

} 先找运算符
再找运算数

*2+34	234+*
*27	27*
14	14

表达式求值：表达式的表示

- **[练习]** 计算下列表达式的值：
  - `*/+1*2+345-76`

```
* / + 1 * 2 + 3 4 5 - 7 6
* / + 1 * 2 + 3 4 5 1
* / + 1 * 2 7 5 1
* / + 1 14 5 1
* / 15 5 1
* 3 1
```

表达式求值：表达式的表示

- 1234+*+5/76-*

1 2 3 4 + * + 5 / 7 6 - *

1 2 7 * + 5 / 7 6 - *

1 14 + 5 / 7 6 - *

15 5 / 7 6 - *

3 7 6 - *

3 1 *

3

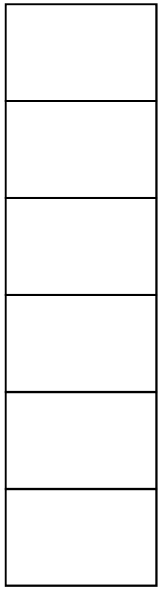
表达式求值：表达式的表示

- 后序表达式的计算机求解
  - 从左向右扫描每一个输入字符
  - 如果是运算数，入栈
  - 如果是运算符
    - 从栈中弹出所需的运算数
    - 运算
    - 结果压回栈
- 前序表达式与之类似

后缀表达式求值

= 25

3 5 * 6 8 4 / - 7 * + #



表达式求值：表达式的转换

- 中序表达式->后序表达式
  - 运算数顺序不变
  - 运算数后的运算符的顺序可以由中序表达式的计算过程确定。

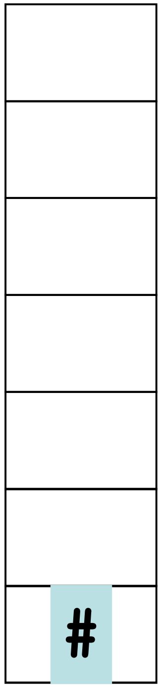
```
a*(b+c)

a*(bc+)

a*bc+

abc+*
```

栈的应用：表达式的转换

- 中序表达式的计算和中序转后序的区别：
  - 回顾中序表达式的计算算法：
    - 对于运算数：保持顺序不变
    - 对于运算符：
      - 优先于栈顶，则入栈
      - 栈顶运算符优先，计算该运算符
  - 因此类似的可以得出转换算法：
    - 对于运算数：直接输出
    - 对于运算符：
      - 优先于栈顶，则入栈
      - 栈顶运算符优先，输出该运算符

中缀表达式转后缀表达式

a * (b * (c + d / e) - f) #



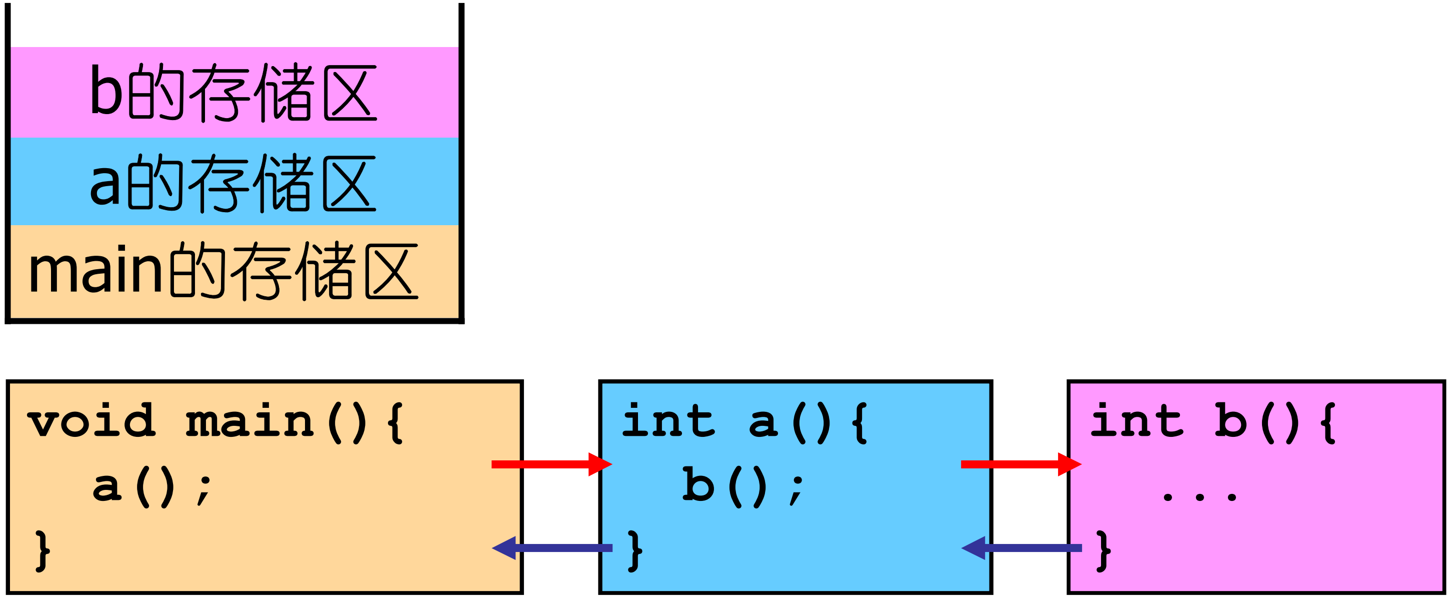
a * (b * (c + d / e) - f) #

栈和递归的实现

- 函数调用与运行栈
  - 当一个函数在运行期间调用另一个函数时，在运行该被调用函数之前，需先完成三件事：
    - 将所有的参数、返回地址等信息传递给被调用函数
    - 为被调用函数的局部变量分配存储区
    - 将控制转移到被调用函数的入口
  - 而从被调用函数返回之前，应该完成：
    - 保存被调函数的计算结果
    - 释放被调函数的数据区
    - 依照被调函数保存的返回地址将控制转移到调用函数

栈和递归的实现

- 运行栈
  - 当多个函数嵌套调用时，由于函数的运行规则是：后调用先返回
  - 因此函数存储的管理通常实行“栈式管理”



b的存储区

a的存储区

main的存储区

```
void main(){
   a();
}
```

```
int a(){
   b();
}
```

```
int b(){
   ...
}
```

栈和递归的实现

- 递归调用
  - 一个递归函数的运行过程类似于多个函数的嵌套调用
  - 差别仅仅在于“调用函数和被调用函数是同一个函数”
  - 运行栈中保存的都是同一个函数不同次调用时的信息

递归

- 当一个问题被分解为规模更小的相似的子问题，而子问题的解决方法和原问题是一样的。
- 可以将原问题转化成这些子问题，而子问题又可以被分解成更小的子问题，如此，最终会到达一个具有明显答案的子问题。
- 如：人口普查、排序、查找、遍历…
- 例1：求阶乘

$$\boxed{Fact(n) = \begin{cases} 1 & n = 1 \\ n \times Fact(n-1) & n > 1 \end{cases}}$$

递归

$$Fact(n) = \begin{cases} 1 & n = 1 \\ n \times Fact(n-1) & n > 1 \end{cases}$$

- 递归程序的两种情形：
  - (1) 递归情形
    - 整体问题的解决分成若干子问题
    - 子问题的解决方法和整体相同
    - 解决整体时假设子问题已经解决
  - (2) 基情形：子问题不需要再分解
    - 如果不留会怎么样？

递归

$$Fact(n) = \begin{cases} 1 & n = 1 \\ n \times Fact(n-1) & n > 1 \end{cases}$$

- 问题是递归定义的，自然可用递归程序解决

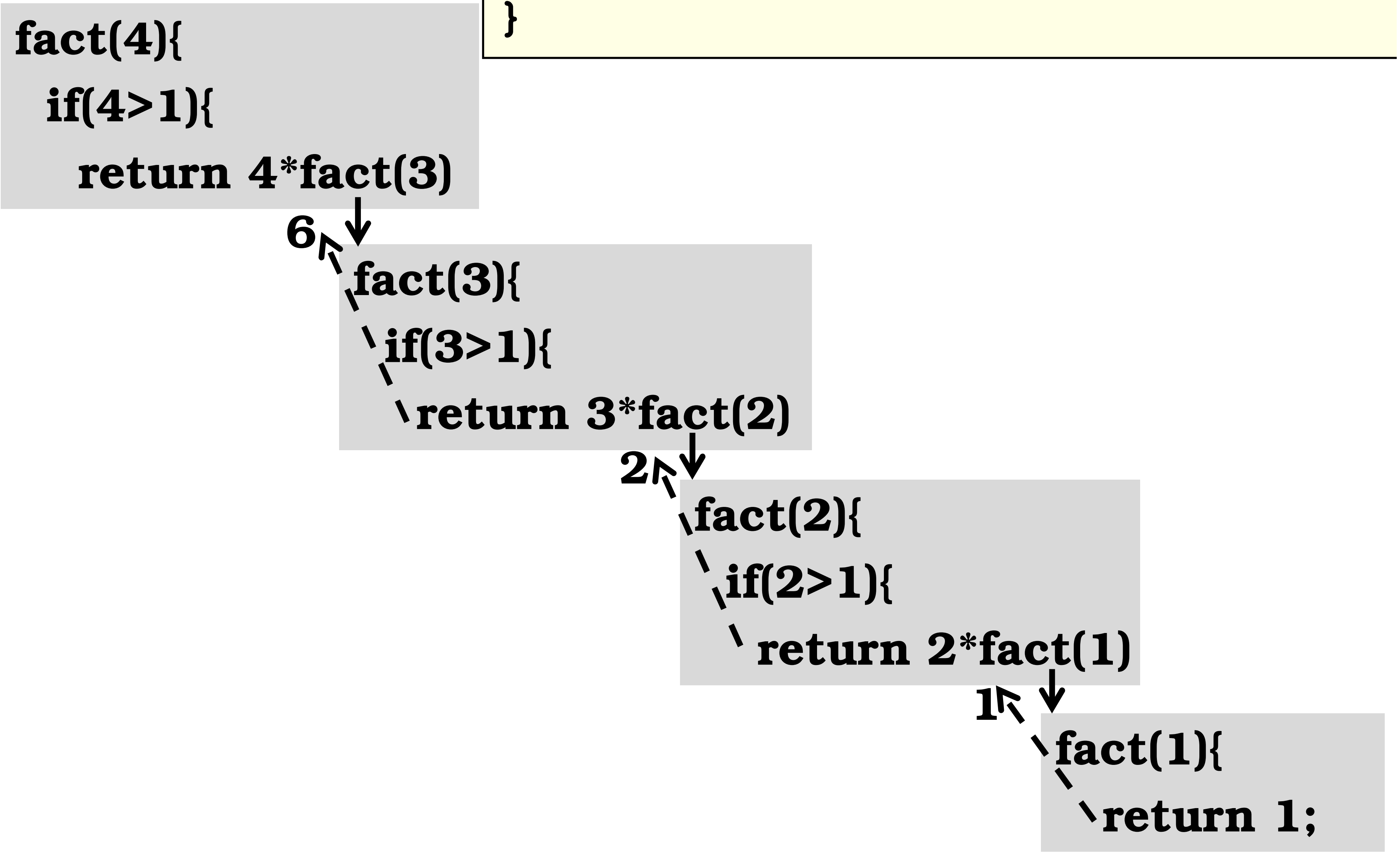
```
int fact(int n){
  if(n > 1) return n*fact(n-1);

  else return 1
}
```

递归情形

基情形

递归

```
int fact(int n){
  if(n > 1) return n*fact(n-1);
  else return 1;
}
```



```
fact(4){
  if(4>1){
    return 4*fact(3)
```

```
fact(3){
  if(3>1){
    return 3*fact(2)
```

```
fact(2){
  if(2>1){
    return 2*fact(1)
```

```
fact(1){
  return 1;
```

递归

- 例2：求最大公约数
  - 算法：辗转相除法

$$\boxed{GCD(M, N) = \begin{cases} M & N = 0 \\ GCD(N, M\%N) & N > 0 \end{cases}}$$

  - 比如：

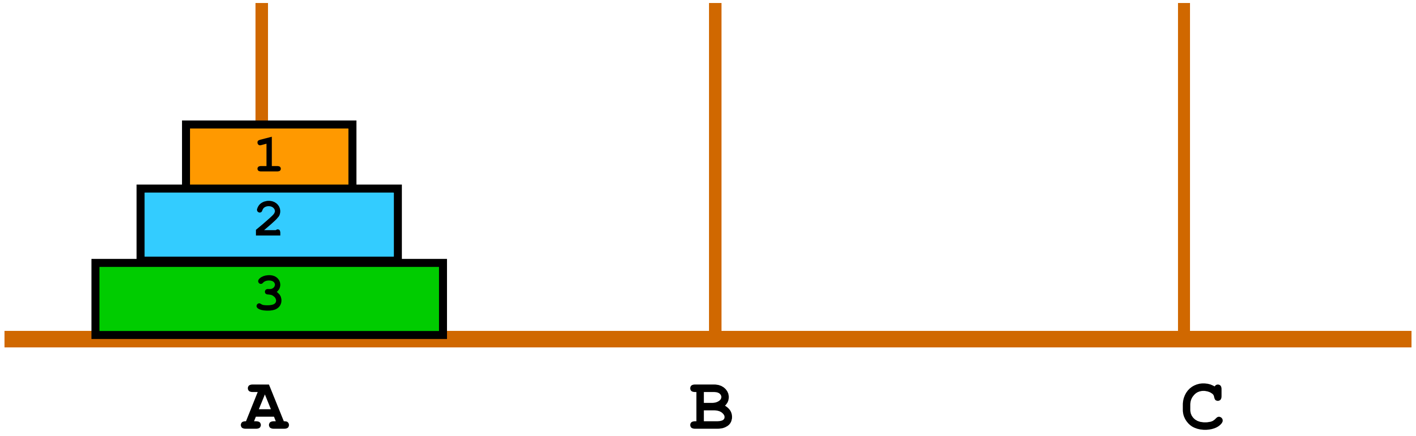
```
GCD(72,27)  =  GCD(27,18)
            =  GCD(18,9)
            =  GCD(9,0)
            =  9
```

递归

- 什么时候考虑用递归算法
  - 问题是递归定义的
    - 比如求阶乘、求**Fibonacci**级数等
  - 数据结构是递归定义的
    - 典型的比如二叉树
  - 解题思路包含有递归规律的
    - 有一些问题并不是直接用递归定义的
    - 但是仔细分析可以发现其中有递归的规律
    - 用递归程序可以很简单的解决

递归

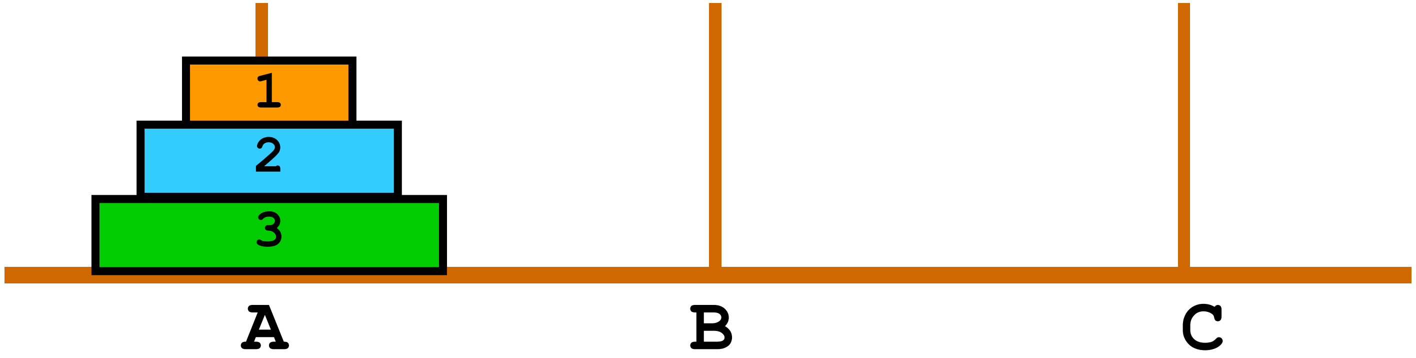
- 例3：汉诺塔



  - 每次只允许移动一个盘子
  - 必须保证小盘子在大盘子之上
  - 如何把所有的盘子从**A**移到**C**？

递归

- 这个问题本身看不出有递归的特点
- 但是解决方法却可以采用递归的策略：
  - 我们把**1**、**2**看成一个整体，假设能够把**1**、**2**移到**B**（至于怎么移动这个整体，以后再说）
  - 这时**3**上面没有盘子，可以直接把**3**移到**C**
  - 最后再把**1**、**2**移到**C**



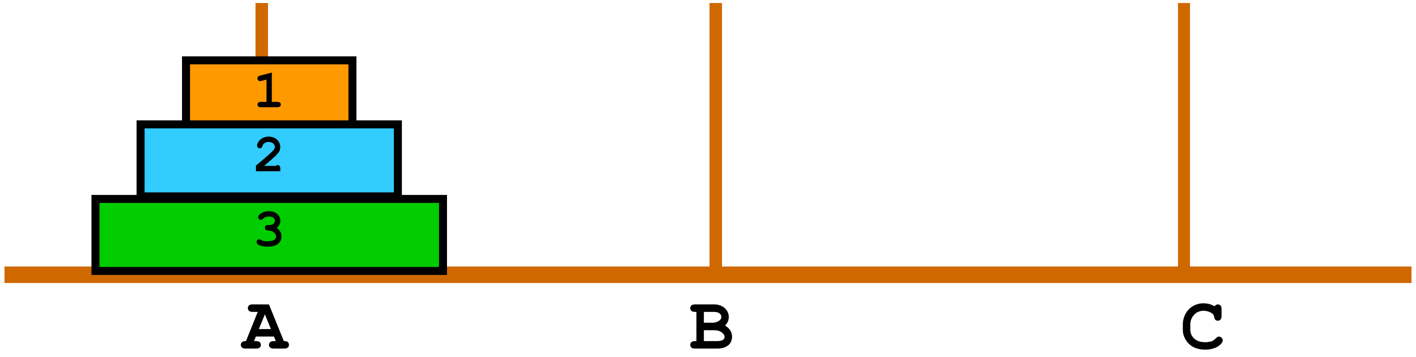
递归

这样移动**n**个盘子的问题就简化为：

(1)用**C**柱做过渡，将**A**柱上的**n-1**个盘子移到**B**上

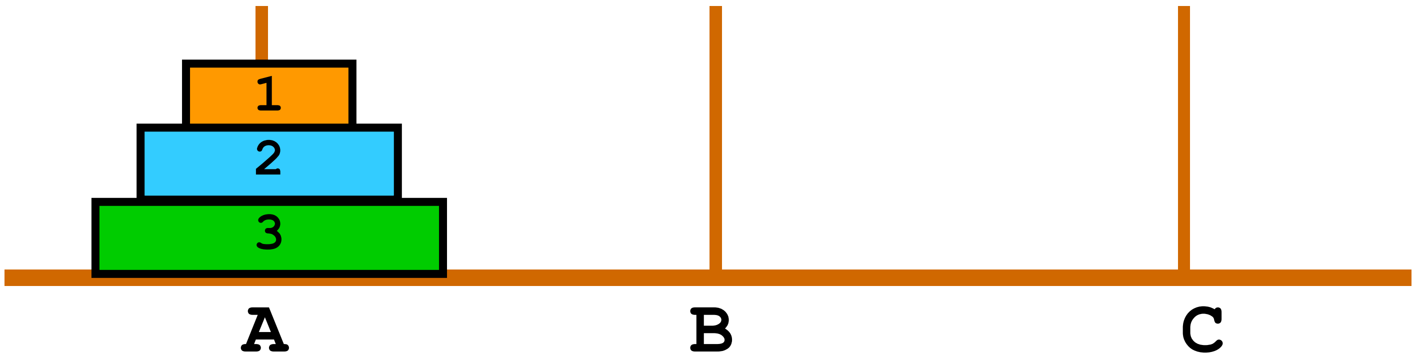
(2)把**A**柱上最下面的盘子直接移到**C**柱上

(3)用**A**柱做过渡，将**B**柱上的**n-1**个盘子移到**C**上



递归

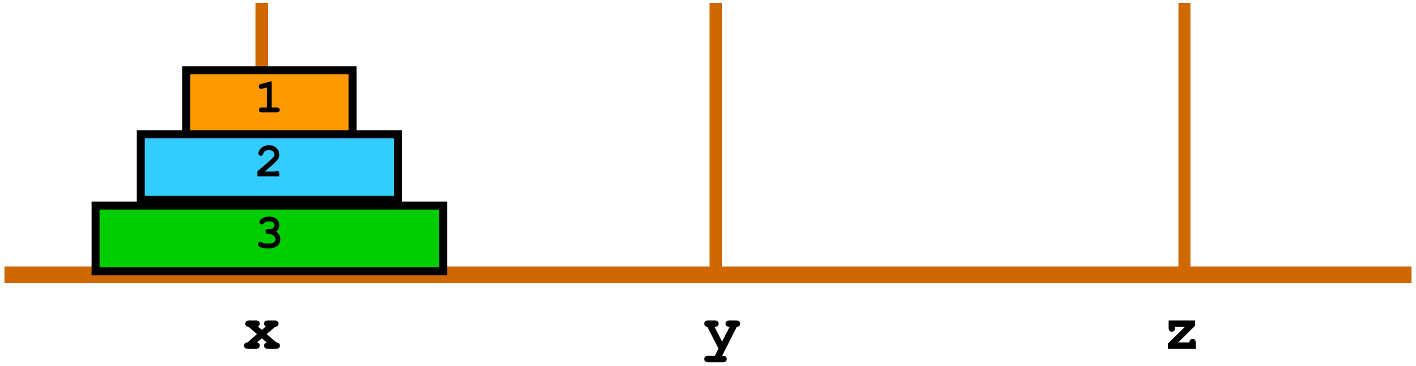
- 移动**1,2**个盘采用同样的方法
  - 先把上面的**1**个盘子移到一个过渡柱子上
  - 然后把最下面的**1**个盘子移到目标柱子上
  - 最后把上面的**1**个盘子移到目标柱子上



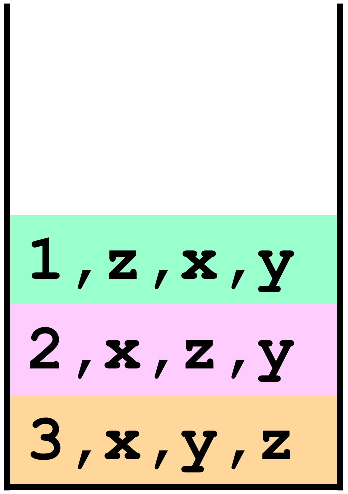
递归

- 递归算法

```
void hanoi
    (int n, char x, char y, char z){
    if (n==1)   move(x, 1, z);
    else {
          hanoi(n-1, x, z, y);
          move(x, n, z);
          hanoi(n-1, y, x, z);
     }
}
```



```
void hanoi
        (int n, char x, char y, char z){
    if (n==1)     move(x, 1, z);
    else {
        hanoi(n-1, x, z, y);
        move(x, n, z);
        hanoi(n-1, y, x, z);
    }
}
```



1,z,x,y

2,x,z,y

3,x,y,z

运行栈