大学之道，在明明德，
在亲民，在止于至善。
知止而后有定，定而后
能静，静而后能安，安
而后能虑，虑而后能得。
物有本末，事有始终。
知所先后，则近道矣。

3.3 栈的应用举例

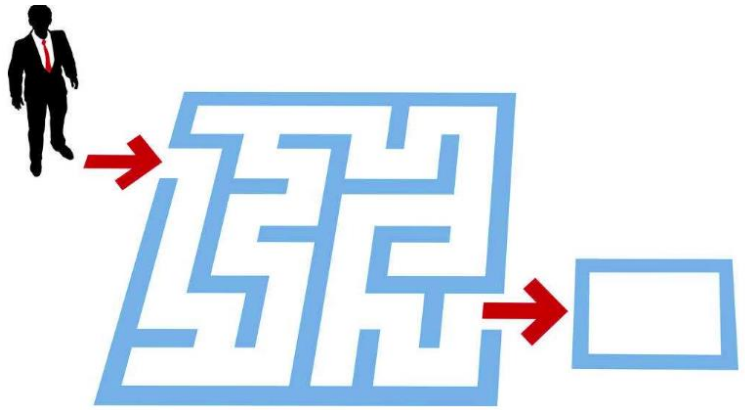
- 数制转换
- 括号匹配的检验
- 迷宫求解
- 表达式求值

3.3.3 迷宫求解

1. 问题介绍

有一个迷宫地图，有一些可达的位置，也有一些不可达的位置（障碍、墙壁、边界）。从一个位置到下一个位置只能通过上下左右四个方向走一步来实现。那么从起点出发，如何找到一条到达终点的通路？

两个问题：迷宫表示；求解算法



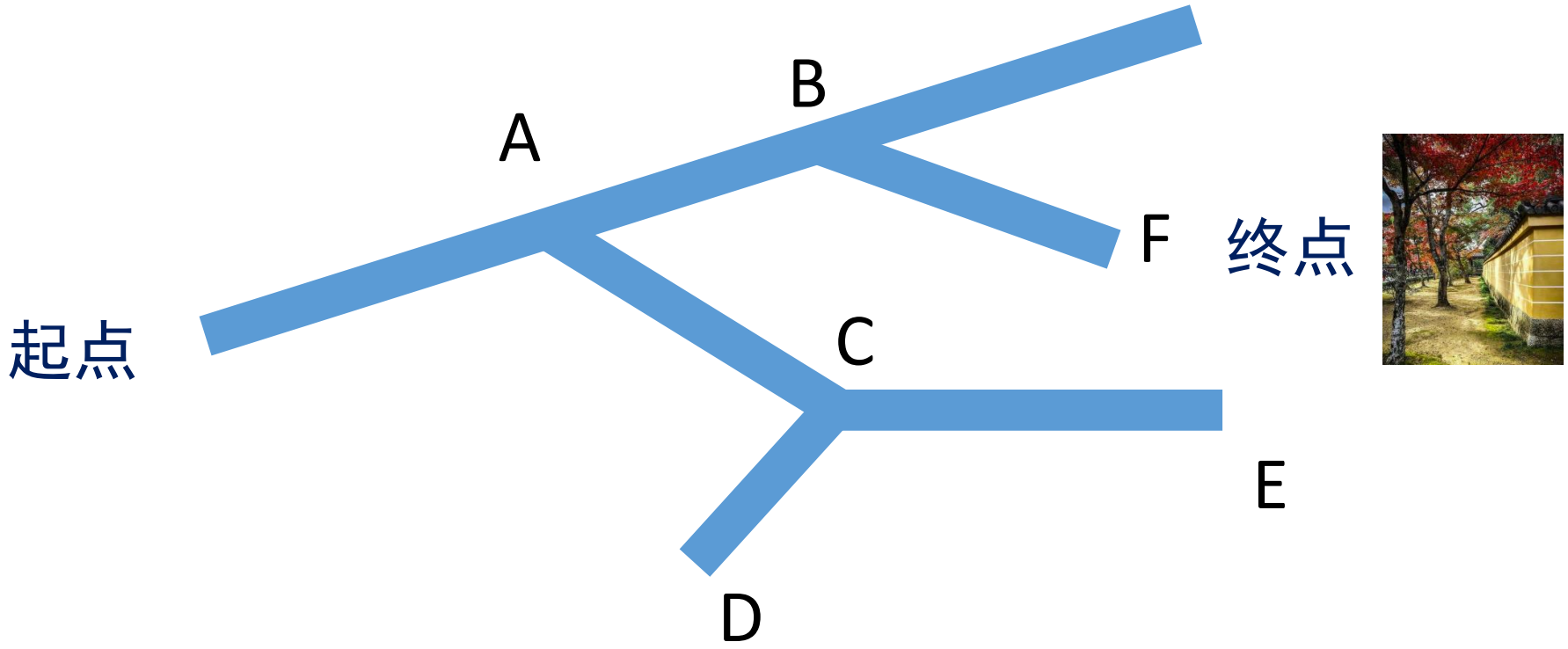
迷宫地图表示

迷宫地图包含：（1）各点位置坐标（2）各位置的状态，即该处是墙还是路。可由一个二维数组来表示：横纵坐标表示迷宫各处的位置坐标，数组元素表示各位置处的状态信息）可以用0表示墙，用1表示可通行。

0	1	0	0	0	0
0	1	1	1	0	0
0	1	0	0	0	0
0	1	1	0	0	0
0	1	0	0	0	0
0	1	0	0	0	0

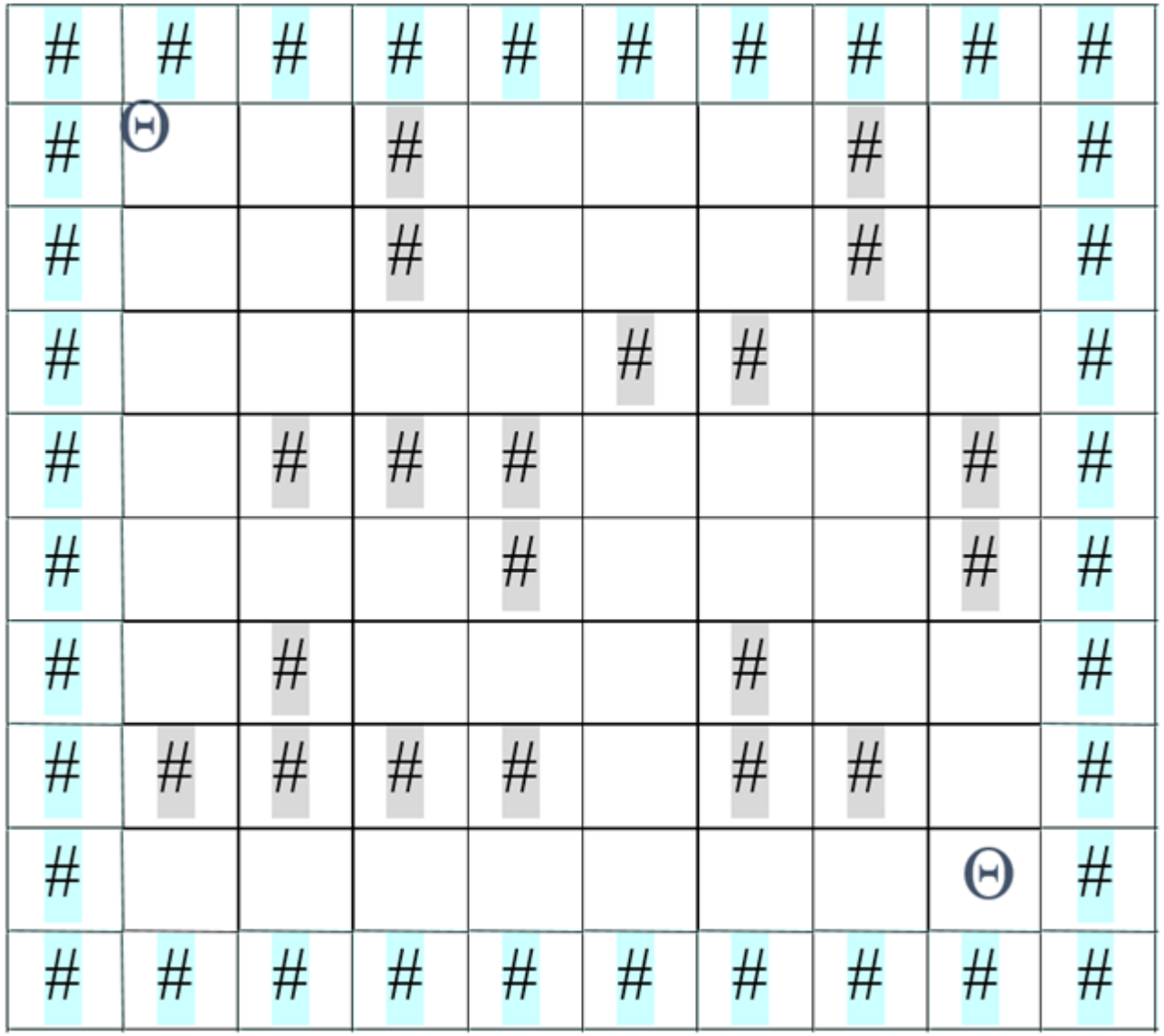
算法思路

设想一下，在实际生活中，要去一个目的地。路线图如下：



回溯法

从当前位置出发，下一步有四种选择，先选择一个方向，如果该方向可通行，则当前位置切换为下一个位置。如果不通，则切换方向走，如果所有方向都走不了，回到上一步的位置去，继续试探其他方向，直到找到路或者没有路。



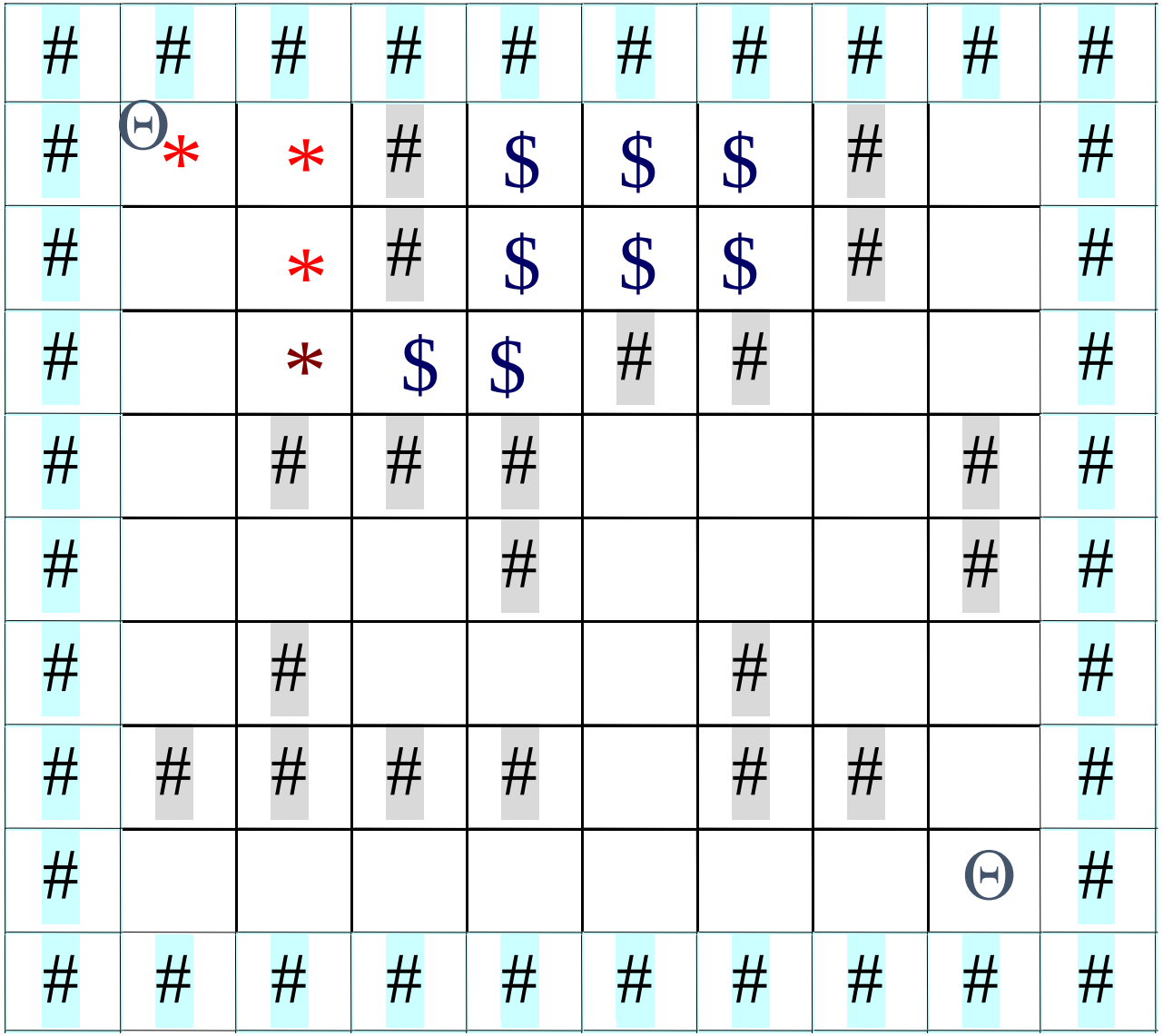
求迷宫路径算法的基本思想

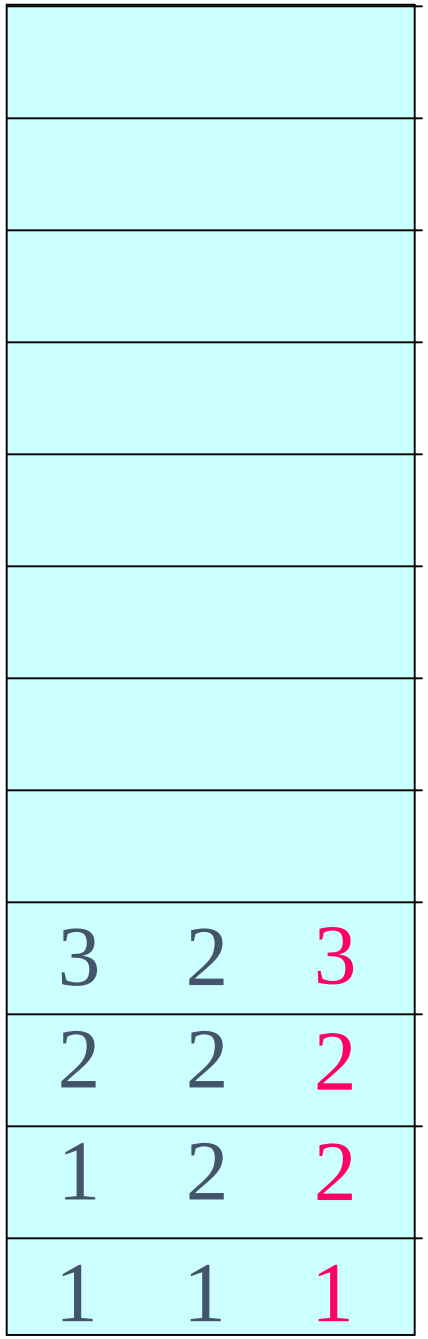
若当前位置“可通”，则纳入路径，继续(向东)前进；

若当前位置“不可通”，则后退，换方向继续探索；

若四周“均无通路”，则将当前位置从路径中删除出去。

思考：试探过程中逐渐寻到路径，那如何保存路径？


#
Θ
*
$


3 2 3
2 2 2
1 2 2
1 1 1

求迷宫中一条从入口到出口的路径的算法

设定当前位置的初值为入口位置；

do {

若当前位置可通，

则{将当前位置插入栈顶；

若该位置是出口位置，则算法结束；

否则切换当前位置的东邻方块为新的当前位置；}

否则 {……}

} while（栈不空）；

若栈不空且栈顶位置尚有其他方向未被探索，则设定新的当前位置为：沿顺时针方向旋转找到的栈顶位置的下一相邻块；

若栈不空但栈顶位置的四周均不可通，

则｛

删去栈顶位置；//回溯

若栈不空，则重新测试新的栈顶位置，测试栈顶的下一个方向；

｝

若栈空，则表明迷宫没有通路。

3. 3. 4表达式求值

问题：表达式的求值问题是一个比较常见的问题之一，通常在编写程序时，输入表达式编译器自动去处理。那么编译器对表达式进行求值的具体原理和过程是什么？

如：1+2； 1 * 2 + (4 - 6/2) * 2

中缀表达式形式

表达式：exp = a * b + (c - d/e) * f

限于二元运算符的表达式定义：

表达式 = （操作数）+（运算符）+（操作数）

操作数 = 简单变量 | 表达式

简单变量 = 标识符 | 无符号整数

表达式的三种表示方法：

设　Exp = S1　OP　S2

OP　S1　S2　　为前缀表示法

S1　OP　S2　　为中缀表示法

S1　S2　OP　　为后缀表示法

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

前缀式:　　+ × a b × – c / d e f

中缀式:　　a × b + c – d / e × f

后缀式:　　a b × c d e / – f × +

例：exp = a * b + (c - d/e) * f

前缀表达式

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

+ * ab * {(c - d/e) } f

+ * ab * - c{ d/e } f

+ * ab * - c/ def

后缀表达式

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

a b * { c - d/e} f * +

a b * c { d/e } - f * +

a b * c d e / - f * +

练习: 7-2+3*(2-1)+(6/2-1),转换为后缀式和前缀式

例如：　**Exp** = a × b + (c – d / e) × f

前缀式: + × a b × – c / d e f

中缀式:　a × b + c – d / e × f

后缀式: a b × c d e / – f × +

结论：

1）操作数之间的相对次序不变

2）运算符的相对次序不同

3）中缀式丢失了括弧信息，致使运算的次序不确定；

4) 前缀式的运算规则为：连续出现的两个操作数和在它们之前且紧靠它们的运算符构成一个最小表达式；

5) 后缀式的运算规则为：运算符在式中出现的顺序恰为 表达式的运算顺序；每个运算符和在它之前出现且紧靠它的两个操作数构成一个最小表达式

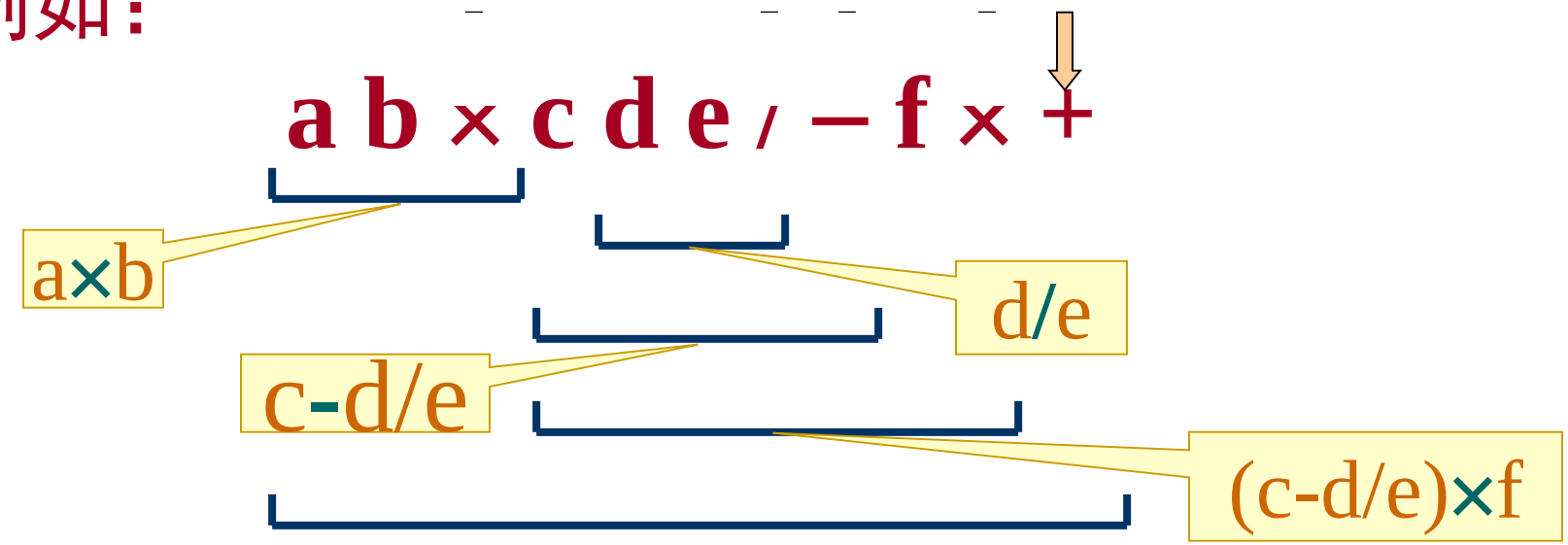
下面给出两种求表达式值的方法

- 分成两步进行求值
  - ① 原表达式变为后缀式
  - ② 后缀式求值
- 直接从原表达式求值

如何从后缀式求值

先找运算符，再找参与运算的操作数。思考：操作数保存在哪里？

例如：



算法描述

```
void value(char suffix[ ],int *c)
{ char *p,ch;
   InitStack(S); p=suffix; ch=*p;
   while(ch!='#'){
      if(!IN(ch, OP)) push(S,ch);
      else {pop(S,a);pop(S,b);push(S,operate(a,ch,b))}
   }
    pop(S,*c);
}
```

如何从原表达式求得后缀式

分析 “原表达式” 和 “后缀式”中的操作数和运算符:

原表达式: $a \times b + (c - d / e) \times f$

后缀式: **a b × c d e / – f × +**

需要注意：1）操作数相对次序不变；2）原表达式中每个运算符的运算次序要由**它之后的一个运算符**来定；3）在后缀式中，优先数高的运算符领先于优先数低的运算符。

思考：转换过程中需要保存运算符，定义什么类型的变量？

a × b + (c − d / e) × f

需要一个运算符栈。

从原表达式求得后缀式的规律

1） 设立暂存运算符的栈；

2） 设表达式的结束符为“#”， 予设运算符栈的栈底为“#”

3） 若当前字符是操作数，则直接发送给后缀式；

4） 若当前运算符的优先数高于栈顶运算符，则进栈；若当前运算符是左圆括号，则进栈；

5） 若当前运算符是右圆括号， 退出栈顶运算符发送给后缀式，直至退出的栈顶运算符是左圆括号为止；若当前运算符优先数小于等于栈顶运算符，则退出栈顶运算符发送给后缀式

```
void transform(char suffix[ ], char exp[ ] ) {
  InitStack(S);  Push(S, '#');
  p = exp;  ch = *p;
  while (!StackEmpty(S)) {
      if (!IN(ch, OP)) Pass( Suffix, ch);
      else {... ...}
      if ( ch!= '#' ) { p++;  ch = *p; }
      else { Pop(S, ch);  Pass(Suffix, ch); }
  } // while
} // CrtExptree
```

```
switch (ch) {
   case '(' :  Push(S, ch); break;
   case ')' :  Pop(S, c);
               while (c!= '(' )
                  { Pass( Suffix, c);  Pop(S, c) }
               break;
    defult  :
      while(Gettop(S, c)&&precede(c)>precede(ch))
        { Pass( Suffix, c);  Pop(S, c); }
      if ( ch!= '#' )  Push( S, ch);
      break;
} // switch
```

直接从原表达式求值

运算规则

使用两个工作栈：

运算符栈：OPTR　　操作数栈：OPND

算法基本思想：

（1）置操作数栈OPND及操作符栈OPTR为空栈，“#”置为OPTR的栈底元素。

（2）依次读入表达式中每个字符，若是操作数则进OPND，若是运算符，则进行如下判断：

若是“（”，进运算符栈；若是“）”，当运算符栈顶是“（”，则弹出栈顶元素，继续扫描下一符号。否则当前读入符号暂不处理，从操作数栈弹出两个操作数，从运算符栈弹出一个运算符，生成运算指令，结果送入操作数栈，继续处理当前读入符号。

若读入的运算符的优先级大于运算符栈顶的优先级，则进运算符栈，继续扫描下一符号；否则从操作数栈顶弹出两个操作数，从运算符栈弹出一个运算符，生成运算指令，把结果送入操作数栈。继续处理刚才读入的符号。

若读入的是“#”，且运算符栈顶的符号也是“#”时，则表达式处理结束。从操作数栈弹出表达式结果。

```
OperandType  EvaluateExpression() {
  InitStack(OPTR); Push(OPTR,'#');
  InitStack(OPND);   c=getchar();
 While(c!='#'||GetTop(OPTR)!='#') {
         if(!In(c,OP)) {Push(OPND,c);c=getchar();}
      else{...}
     }//while
c=Gettop(OPND);
DestroyStack(OPTR);
DestroyStack(OPND);
return c; }
```

```
if(Precede(GetTop(OPTR)) < Precede(c)  || c == '('
      {Push(OPTR,c); c=getchar();}
else if ((Precede(GetTop(OPTR)) == Precede(c))
      {x=Pop(OPTR); c=getchar(); }
else if ((Precede(GetTop(OPTR)) > Precede(c))
     {theta = Pop(OPTR);
      b= Pop(OPND); a = Pop(OPND);
      Push(OPND,Operate(a,theta,b)); }
```

例：3*（7-2）

步骤	OPTR 栈	OPND栈	输入字符	主要操作
1	#		3*(7-2)#	Push(OPND,’3’)
2	#	3	*(7-2)#	Push(OPTR,’*’)
3	#*	3	(7-2)#	Push(OPTR,’(’)
4	#*(	3	7-2)#	Push(OPTR,’7’)
5	#*(	37	-2)#	Push(OPTR,’-’)
6	#*(-	37	2)#	Push(OPTR,’2’)
7	#*(-	372	)#	Operate(‘7’,’-’,’2’)
8	#*(	35	)#	Pop(OPTR)
9	#*	35	#	operate(‘3’,’*’,’5’)
10	#	15	#	return (GetTop(opnd))

作业

1. 表达式求值

   基本要求：一位数字

   扩展：任意位数字。

2. 迷宫求解（扩展选作）