3.2.1 数制转换

十进制数N和其它d进制数的转换的算法基于原理：

N = (N div d)×d + N mod d

其中，div 相除取整，mod 相除取余。

例如：（1348)$_{10}$ = (2504)$_{8}$ ，其运算过程如下：

N	N div 8	N mod 8
1348	168	4
168	21	0
21	2	5
2	0	2

计算顺序 ↓

↑ 输出顺序

由于上述计算过程是从低位到高位顺序产生八进制数的各个数位，而打印输出，一般来说应从高位到低位进行，恰好和计算过程相反。因此，若将计算过程中得到的八进制数的各位顺序进栈，则按出栈序列打印输出的即为与输入对应的八进制数。

具体方法见算法3.1

```
void conversion () {
    //对于输入的任意一个非负十进制整数，
    //打印输出与其等值的八进制数
   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”, e )；
   }
} // conversion
```

算法 3.1

3.2.2 括号匹配的检验

假设在表达式中，（ [ ] ( )) 或 [([ ] [ ])] ，等为正确的格式，[( ]) 或 ([( )) 或 (( )]) 均为不正确的格式。

则 检验括号是否匹配的方法可用“期待的急迫程度”这个概念来描述。

例如：考虑下列括号序列：

[([ ] [ ])]

1 2 3 4 5 6 7 8

分析可能出现的**不匹配**的情况:

- 到来的右括弧并非是所“期待”的；
- 到来的是“不速之客”；
- 直到结束，也没有到来所“期待”的括弧。

算法的设计思想：

1）凡出现**左括弧**，则**进栈**；

2）凡出现**右括弧**，首先检查栈是否空，若**栈空**，则表明该**“右括弧”多余**，否则和栈顶元素比较，若**相匹配**，则**“左括弧出栈”**，否则表明**不匹配**。

3）表达式检验结束时，若栈空，则表明表达式中**匹配正确**，否则表明**“左括弧”有余**。

```
Status matching(string& exp) {
  int state = 1;
  while (i<=Length(exp) && state) {
    switch of exp[i] {
      case 左括弧：{Push(S,exp[i])； i++； break；}
      case”)”:{     //右括号情况
        if(NOT StackEmpty(S)&&GetTop(S)=“(“
          {Pop(S,e)； i++；} //匹配成功
        else {state = 0；}
        break；  }  … …//其它类型括号类似处理
  }
  if (StackEmpty(S)&&state) return OK； ……
```

3.2.3 行编辑程序问题

问题：一个简单的行编辑程序的功能是：接受用户从终端输入的程序或数据，并存入用户的数据区。

如何实现？“每接受一个字符即存入存储器” ？

这样做并不恰当！

应该是，在用户输入一行的过程中，允许用户输入出差错，并在发现有误时可以及时更正。

合理的作法是：

设立一个输入缓冲区，用以接受用户输入的一行字符，然后逐行存入用户数据区，并假设“#”为退格符，“@”为退行符。

假设从终端接受了这样两行字符：

```
whli##ilr#e ( s#*s)
outcha@putchar(*s=#++);
```

则实际有效的是下列两行：

```
while (*s)
putchar(*s++);
```

```
Void LineEdit ()  {  //利用字符栈S，从终端接受
//一行并传送至调用过程的数据区。
InitStack (S);       //构造空栈S
ch = getchar ();    //从终端接收第一个字符
 while (ch != EOF) { //EOF为全文结束符
    while (ch != EOF && ch != '\n') {
      switch (ch) {
       case  '#'  : Pop(S, c); break;     //仅当栈非空时退栈
       case '@': ClearStack(S); break;   // 重置S为空栈
       default : Push(S, ch);  break;
                          //有效字符进栈，未考虑栈满情形
      }
      ch = getchar();  // 从终端接收下一个字符 }
```

```
    将从栈底到栈顶的字符传送至调用过程的
    数据区；
    ClearStack(S);      // 重置S为空栈
    if (ch != EOF)  ch = getchar();
  }
  DestroyStack (S);
}//LineEdit
```

算法 3.2

3.2.4 迷宫求解

求迷宫中从入口到出口的所有路径是一个经典的程序设计问题，通常用的是“**穷举求解**”的方法。即从入口出发，顺某一方向向前探索，若能走通，则继续往前走；否则沿原路退回，换一个方向再继续探索，直至所有可能的通路都探索到为止。

为了保证在任何位置上都能沿原路退回，显然需要用一个后进先出的结构来保存从入口到当前位置的路径。因此，求迷宫通路的算法需要用到栈。

首先，在计算机中可以用如图3.4 所示的方块图表示迷宫。图中每个方块或为通道（以空白方块表示），或为墙（以带阴影线方块表示），所求路径必须是简单路径，即在求得的路径上不能重复出现同一通道块。

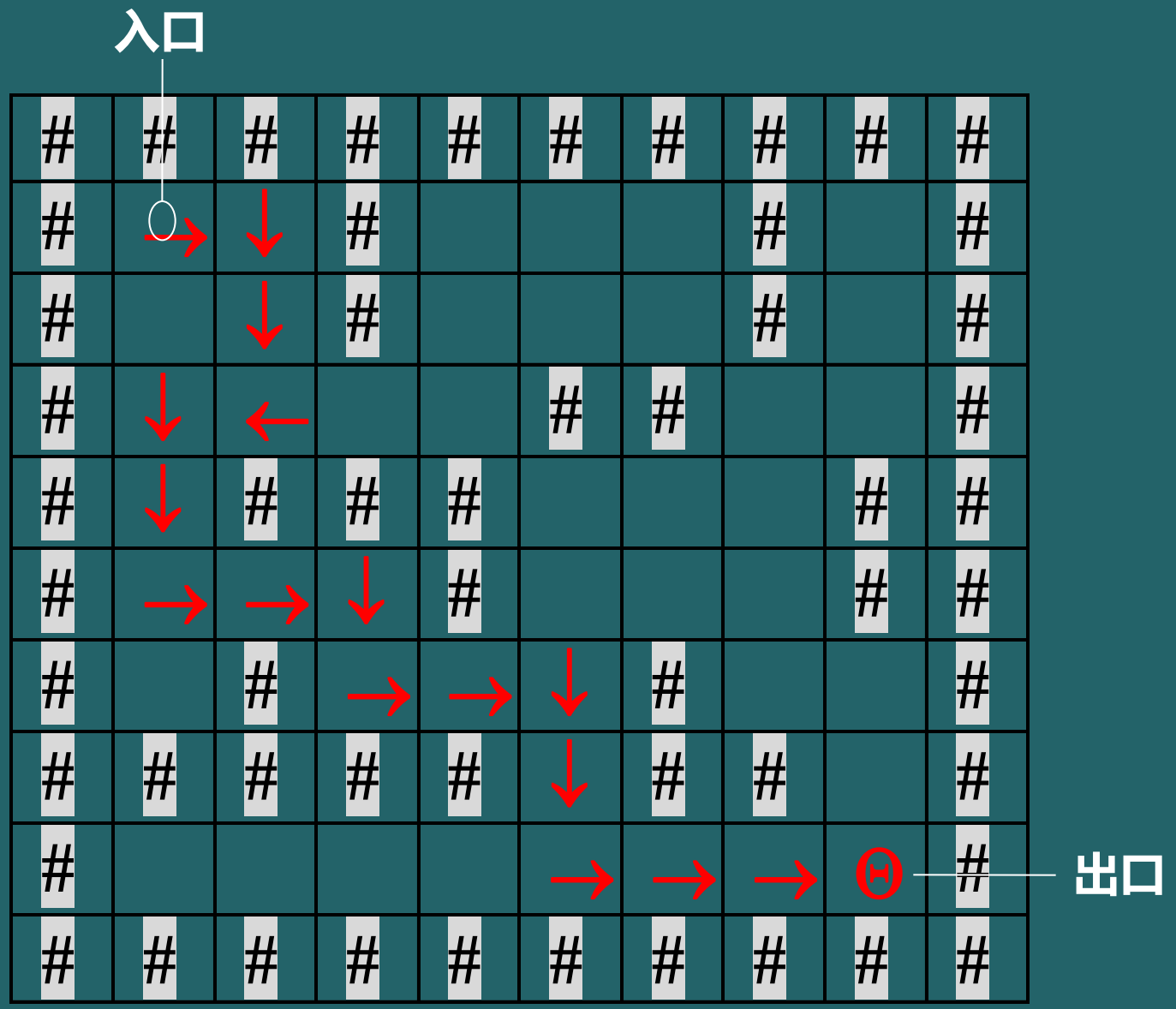

图3.4 迷宫

假设“当前位置”指的是“在搜索过程中某一时刻所在图中某个方块位置”，则求迷宫中一条路径算法的基本思想是：

- 若当前位置“可通”，则纳入“当前路径”，并继续朝“下一位置”探索，即切换“下一位置”为“当前位置”，如此重复直至到达出口；
- 若当前位置“不可通”，则顺着“来向”退回到“前一通道快”，换方向继续探索；
- 若该通道快的四周“均无通路”，则应从“当前路径”上删除该通道快。

假设以栈 S 记录“当前路径”，则栈顶中存放的是“当前路径上最后一个通道快”。由此，“纳入路径”的操作即为“当前位置入栈”；“从当前路径上删除前一通道快”的操作即为“出栈”。

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；//纳入路径

若该位置是出口位置，则结束；//求得路径存放在栈中

否则切换当前位置的东邻方块为新的当前位置；

}

否则，

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；// 从路径中删去该通道块

若栈不空，**则**重新测试新的栈顶位置，

直至找到一个可通的相邻块或出栈至栈空；

}

} **while(栈不空)**

若栈空，**则**表明迷宫没有通路。

```
typedef struct{
   int ord;                    //通道块在路经上的“序号”
   PosType seat;      //通道块在迷宫中的“坐标位置”
   int di;          //从此通道块走向下一通道块的“方向”
}SElemType;                              //栈的元素类型
Status MazePath (MazeType maze, PosType start, PosType
    end){
     //若迷宫maze中存在从入口start到出口end的通道，则求
     //得一条存放在栈中（从栈底到栈顶），并返回
     // TRUE；否则返回FALSE
      InitStack (S); curpos=start;
                              //设定“当前位置”为“入口位置”
        curstep=1;                          //探索第一步
```

```
do{
 if (Pass (curpos))  {
      //当前位置可以通过，即是未曾走到过的通道块
     FootPrint (curpos);                  //留下足迹
     e = (curstep, curpos, 1);
     Push (S,e);                          //加入路经
     if(curpos==end)   return (TRUE);
                                  //到达出口（终点）
     curpos = NextPos (curpos, 1);
                          //下一位置是当前位置的东邻
     curstep++;                           //探索下一步
}//if
else{                              //当前位置不能通过
```

```
        if (!StackEmpty(S)) {
          Pop (S, e);
          while (e.di=4&& !StackEmpty(S)) {
             MarkPrint (e.seat); Pop (S, e);
                          //留下不能通过的标记，并退回一步
          }//while
          if (e.di<4){
            e.di++; Push (S, e);          //换下一个方向探索
            curpos = NextPos (e.seat e.di);
                        //设定当前位置是该新方向上的相邻快
           }//if
        }//if
      }//else
    }while (!StackEmpty(S));
  return (FALSE);
}//MazePath
```

算法 3.3

3.2.5 表达式求值

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

在表达式求值中，一种简单直观、广泛使用的算法，通常称为“算符优先法”。

例如：对表达式 4+2*3-10/5 求值，按“算符优先法”对其求解过程如下：

4+2*3-10/5=4+6-10/5=10-10/5=10-2=8

任河一个表达式都是由操作数（operand）、运算符（operator）和界限符（delimiter）组成的，我们称之为单词。

把运算符和界限符统称为算符，它们构成的集合命名为OP。根据优先运算规则，在运算的每一步中，任意两个相继出现的算符θ_1和θ_2之间的优先关系至多是下面三种关系之一：

$\theta_1 < \theta_2$　θ_1的优先权低于θ_2

$\theta_1 = \theta_2$　θ_1的优先权等于θ_2

$\theta_1 > \theta_2$　θ_1的优先权高于θ_2

各种基本算符间的优先关系见教材p53，表3.1

算符优先算法求表达式值过程为：

1) 设立操作符**栈OPTR**，操作数或运算结果栈**OPND**；

2) 设表达式的结束符为“#”，予设运算符栈的栈底为“#”，操作数栈为空栈；

3) 依次读入表达式中每个字符，若是操作数则进OPND栈，若是运算符则和OPTR栈的栈顶运算符比较优先权后作相应操作，直至整个表达式求值完毕（即OPTR栈的栈顶元素和当前读入的字符均为‘#’）。

```
OperandType EvaluateExpression() {
  //算术表达式求值的算符优先算法。设OPTR和OPND
  //分别为运算符栈和运算数栈，OP为运算符集合。
  InitStack (OPTR); Push(OPTR,’#’);
  initStack (OPND); c=getchar();
  while (c!= ‘#’ ||GetTop(OPTR)!= ‘#’) {
    if (!In (c, OP)) { Push (OPND, c); c= getchar();}
                                        //不是运算符则进栈
    else
      switch (Precede( GetTop (OPTR), c)){
        case’ <’ :                      //栈顶元素优先权低
            Push (OPTR, c); c= getchar();
            break;
```

```
            case‘ =’ :  //脱括号并接受下一字符
                        Pop (OPTR, c); c= getchar();
                        break;
                case‘ >’ :          //退栈并将运算结果入栈
                        Pop (OPTR, theta);
                        Pop (OPND, b); Pop (OPND, a);
                        Push (OPND, Operate (a, theta, b));
                        break;
                }//switch
        }//while
        return GetTop(OPND);
}//EvaluateExpression
```

算法 3.4