Objective:

- Stack Applications
  - Backtracking: Traversing in Maze
  - Expression Evaluation

Rat in a Maze

In detail, it is discussed in your textbook-A: section-3.3.
For now, I will discuss here about the basic problem and how to approach it.

Green Border Cell = Source
Red Border Cell = Destination
Blue Border Cell = Current Position
Green Highlighted text = active path.
Red Highlighted text = backtrack

Any possible path to go from green/source cell to red/destination cell?
From any given cell: Possible moves top/bottom/left/right.
One of the possible path finding is described below. Observe/examine the tables/matrices from left to right and top to bottom.

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

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

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

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

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

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

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

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

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

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

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

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

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

Dead End: can't move further. How do we

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

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

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

know that its dead end? It is because of coloring the visited cells.
Go back where: seems an easy case in this case: go back to immediately previous visited location.

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

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

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

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

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

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

This scenario and the next three tables/moves reveals the importance of coloring the visited cells and also put us in need to maintain/store the visited cells in stack so we may fetch them in reverse order.

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

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

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

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

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

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

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

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

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

Ponder yourself: Will there always be a path? NO! So, how do we know that path doesn't exist.
Homework: Write code to find whether the path is available in the given Boolean matrix for given source and destination. If path available then display the path as well.

Expression Evaluation

I have written algorithms (actually code skeleton ☺) below related to expression validation and evaluation. Which are self-explanatory. Read them, we shall talk about them in the session on 25-Mar-2020.

Homogeneous Brackets Validity

Purpose: Validate the valid combination of parenthesis.
Example: Valid: ((()))(), not Valid: ())()(()

```
Input = Infix Expression
Output = 1 means valid 0 means invalid expression
Create a stack that can hold opening brackets in it
While (Scan the infixstring from left to left till the end)
{
        get next_character from infixstring
        if ( next_character is opening bracket '(' )
        {
                push next_character into stack
        }
        else if ( next_character is closing bracket ')' )
        {
                if (stack is empty)
                        return 0
                pop opening bracket from stack
        }
}
if (stack is empty)
       return 1
else
       return 0
```

Heterogeneous Brackets Validity

Purpose: Validate the valid combination of heterogeneous brackets.
Example: Valid: ({()})[], not Valid: {[(])()()}

```
Input = Infix Expression
Output = 1 means valid 0 means invalid expression
Create a stack that can hold opening brackets in it
While (Scan the infixstring from left to left till the end)
{
        get next_character from infixstring
        if ( next_character is opening bracket '(' OR '{' OR '[')
        {
                push next_character into stack
        }
        if ( next_character is closing bracket ')' OR '}' OR ']')
        {
                if (stack is empty)
                        return 0
                stacktop = pop from stack
                if (! (stacktop is exactly opposite of next_character ) )
                {
                        return 0;
                }
        }

        if (stack is empty)
                return 1
        else
                return 0
}
```

Three ways to write expression:

Infix, Postfix and Prefix notations are three different but equivalent ways of writing expressions. It is easiest to demonstrate the differences by looking at examples of operators that take two operands.

Infix notation: X + Y
Operators are written in-between their operands. This is the usual way we write expressions.

Postfix notation: (also known as "Reverse Polish notation"): X Y +
Operators are written after their operands.

Prefix notation: (also known as "Polish notation"): + X Y
Operators are written before their operands.

Infix To Postfix Conversion (Reverse Polish Notation (RPN))

***Reverse Polish notation** (**RPN**) is a mathematical notation in which every operator follows all of its operands, in contrast to Polish notation, which puts the operator in the prefix position. It is also known as **postfix notation** and is parenthesis-free as long as operator arities are fixed. The description "Polish" refers to the nationality of logician Jan Łukasiewicz, who invented (prefix) Polish notation in the 1920s.*

The reverse Polish scheme was proposed in 1954 by Burks, Warren, and Wright[1] and was independently reinvented by F. L. Bauer and E. W. Dijkstra in the early 1960s to reduce computer memory access and utilize the stack to evaluate expressions. The algorithms and notation for this scheme were extended by Australian philosopher and computer scientist Charles Hamblin in the mid-1950s.[2][3]

During the 1970s and 1980s, RPN was known to many calculator users, as it was used in some handheld calculators of the time designed for advanced users: for example, the HP-10C series and Sinclair Scientific calculators. [see http://en.wikipedia.org/wiki/Reverse_Polish_notation for detail]

Purpose: Solving an expression without taking care of precedence rules in a single pass.
Assumption: each operand and operator is single character symbol.
Example:

Infix Expression String: a + b	Output Expression String: ab+
Infix Expression String: a / b + c	Output Expression String: ab/c+
Infix Expression String: a / (b + c)	Output Expression String: abc+/

Infix to Postfix Conversion manually/by hand:
See the example below: ^ means power operator

$$A + D - Z * (C - E) / B * Y \text{ ^ } Z$$

Let's convert the above expression into postfix. This conversion is just like the way we actually solve an infix expression.
Step by Step conversion:
We shall start with solving $(C - E)$ as per precedence rules.
$(C - E)$ will be equal to $-CE$ according postfix expression operator placement.
We now consider $-CE$ as single operand and place it in infix expression.

$$A + D - Z * -CE / B * Y \text{ ^ } Z$$

We may now proceed to convert $A + D$ OR $Z * -CE$ OR $Y \text{ ^ } Z$
Why: because it will not affect the end result of expression although as per precedence, we should go for $Y \text{ ^ } Z$

$$A\,D + - Z * -CE / B * Y \text{ ^ } Z$$
$$A\,D + - Z - CE * / B * Y \text{ ^ } Z$$
$$A\,D + - Z - CE * B / * Y \text{ ^ } Z$$
$$A\,D + - Z - CE * B / * YZ\text{^}$$
$$A\,D + - Z - CE * B / YZ\text{^} *$$
$$A\,D + Z - CE * B / YZ\text{^} * -$$

Algorithm

```
Input = infix string
Output = postfix string
Create a stack that can store operators in it
While (Scan the infix_string from left to right till the end)
{
        get next_character from infix_string
        if (next_character is operand)
        {
                Append next_character to postfix_string
        }
        else if (next_character is operator)
        {
                while (stack is not empty AND precedence(stacktop) > precedence(next_character))
                {
                        pop the operator from stack and append it to postfix_string
                }
                if ( next_character is not ')' )
                {
                        push next_character to stack
                }
                else if ( next_character is ')' )
                {
                        pop from stack //it will pop '(' bracket
                }
        }
}
while(stack is not empty)
{
        pop the operator from stack and append it to postfix_string
}
```

Some Rules about Precedence and about push and pop of operators in stack

- Closing bracket can never be pushed in stack.
- If operand then append in postfix string.
- If operator then push in stack.
- A low precedence operator can never on top of high precedence operator in the stack.
  E.g. if in stack, the operator are in this order from bottom to top
  + / ....this is right but
  / + ...this is wrong and
  / / ....this is also wrong but what about this
  + / (+ ....this is also right
  It means that this rules implements in the stack from the start of an '(' till the next '(' occurs.
- If operator is opening bracket then push it in stack but If operator is closing bracket ')' then pop the stack until '(' not found in the stack.
  After popping all the operators until '('also pop that '('.

Step by Step conversion of the $A + D - Z * (C + E * U - (X * S + T))/B * Y \wedge Z$ As per above algorithm

Infix/Input Char	Operator Stack	Output/Postfix String	Comments
A		A	
$+$	$+$	A	Should we solve +? Can't decide until we see next operator.
D		AD	
$-$	$-$	$AD+$	Should we solve -? Can't decide until we see next operator. But now we may decide about the previous operator + on stack top. Since + and – have same precedence but associativity left to right, so, we pop + and append to postfix string.
Z	$-$	$AD+Z$	
$*$	$-*$	$AD+Z$	
$($	$-*($	$AD+Z$	
C	$-*($	$AD+ZC$	
$+$	$-*(+$	$AD+ZC$	
E	$-*(+$	$AD+ZCE$	
$*$	$-*(+*$	$AD+ZCE$	
U	$-*(+*$	$AD+ZCEU$	
$-$	$-*(-$	$AD+ZCEU*+$	Keep popping from the stack and append to output string until you find an operator of higher precedence
$($	$-*(-($	$AD+ZCEU*+$	
X	$-*(-($	$AD+ZCEU*+X$	
$*$	$-*(-(*$	$AD+ZCEU*+X$	
S	$-*(-(*$	$AD+ZCEU*+XS$	
$+$	$-*(-(+$	$AD+ZCEU*+XS*$	
T	$-*(-(+$	$AD+ZCEU*+XS*T$	
$)$	$-*(-$	$AD+ZCEU*+XS*T+$	
$)$	$-*$	$AD+ZCEU*+XS*T+-$	
$/$	$-/$	$AD+ZCEU*+XS*T+-*$	
B	$-/$	$AD+ZCEU*+XS*T+-*B$	
$*$	$-*$	$AD+ZCEU*+XS*T+-*B/$	
Y	$-*$	$AD+ZCEU*+XS*T+-*B/Y$	
$\wedge$	$-*\wedge$	$AD+ZCEU*+XS*T+-*B/Y$	
Z	$-*\wedge$	$AD+ZCEU*+XS*T+-*B/YZ$	
		$AD+ZCEU*+XS*T+-*B/YZ\wedge*-$	Input string finished so empty the stack and append the operators to output string

Postfix Expression Evaluation

```
Input = postfix string
Output = result of postfix expression
Create a stack that can store operands in it
While (Scan the postfixstring from left to right till the end)
{
        get next_character from postfix_string
        if (next_character is operand)
        {
                push next_charcter into stack.
        }
        else if (next_character is operator)
        {
                operand2 = pop operand from stack
                operand1 = pop operand from stack
                Perform operation on the operand1 and operand2 depending on the operator.
                Push the result of this operation in stack.
        }
}
Pop the result of "Postfix Expression Evaluation" from stack.
```

Step by Step evaluation of the $82+2/$ As per above algorithm: assume each operand is single digit.

Postfix Express Char	Operand Stack
8	8
2	8,2
+	10
2	10,2
/	5

What was the original infix expression: $(8+2)/2 \ = \ 82+2/$ But we don't care about precedence while solving postfix expression.

Step by Step evaluation of the $(A+B)/D-E \ = \ AB+D/E-$ As per above algorithm

Postfix Expression Scan	Operand Stack	Comments
A	A	
B	B	
$+$	$AB+$	Consider $AB+$ one operand: taking the values of A amd B we solved $A+B$ and pushed the result onto stack.
D	$AB+,D$	Two operands in stack i.e. $AB+$ and D
$/$	$AB+D/$	
E	$AB+D/,E$	
$-$	$AB+D/E-$	

Infix To Prefix Conversion (Polish Notation (PN))

Example:

Infix Expression String: a + b	Output Expression String: +ab
Infix Expression String: a / b + c	Output Expression String: +/abc
Infix Expression String: a / (b + c)	Output Expression String: /a+bc

I am skipping its conversion algorithm for the online session

Advantage of Evaluating Expressions, which are in Postfix and Prefix form

- No nesting of brackets
- Both prefix and postfix form expressions can be evaluating in one pass (reading the expression only once) rather than multiple passes in Infix Expression.

→ For interest: you may also look at shunting yard algorithm (a method for parsing mathematical expressions specified in infix notation) invented by Edsger Dijkstra
http://en.wikipedia.org/wiki/Shunting-yard_algorithm

Homework: Implement the algorithm for Infix to postfix and postfix expression evaluation. Actually, if you observe closely, you will find that it is not needed to first convert and then solve postfix, you can do the evaluation part while converting infix to postfix.