HW3(deadline:2014/10/26)

4. Using manual transformation, change the following postfix or prefix expressions to infix:

   a. A B * C – D +
   b. A B C + * D –
   c. + – * A B C D
   d. – * A + B C D

6. If the values of A, B, C, and D are 2, 3, 4, and 5, respectively, manually calculate the value of the following prefix expressions:

   a. + – * A B C D
   b. – * A + B C D

8. Determine the value of the following postfix expressions when the variables have the following values: A is 2, B is 3, C is 4, and D is 5.

   a. A B C D * – +
   b. D C * B A + –

16. One of the applications of a stack is to backtrack—that is, to retrace its steps. As an example, imagine we want to read a list of items, and each time we read a negative number we must backtrack and print the five numbers that come before the negative number and then discard the negative number.

    Use a stack to solve this problem. Read the numbers and push them into the stack (without printing them) until a negative number is read. At this time, stop reading and pop five items from the stack and print them. If there are fewer than five items in the stack, print an error message and stop the program.

    After printing the five items, resume reading data and placing them in the stack. When the end of the file is detected, print a message and the items remaining in the stack.

    Test your program with the following data:

    ```
    1 2 3 4 5 -1 1 2 3 4 5 6 7 8 9 10 -2 11 12 -3 1 2 3 4 5
    ```

24. Write a program that simulates a mouse in a maze. The program must print the path taken by the mouse from the starting point to the final point, including all spots that have been visited and backtracked. Thus, if a spot is visited two times, it must be printed two times; if it is visited three times, it must be printed three times.

    The maze is shown in Figure 3-25. The entrance spot, where the mouse starts its journey, is chosen by the user who runs the program. It can be changed each time.

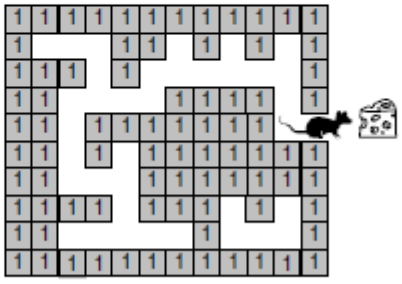

FIGURE 3-25 Mouse Maze for Project 24

A two-dimensional array can be used as a supporting data structure to store the maze. Each element of the array can be black or white. A black element is a square that the mouse cannot enter. A white element is a square that can be used by the mouse. In the array a black element can be represented by a 1 and a white element by a 0.

When the mouse is traversing the maze, it visits the elements one by one. In other words, the mouse does not consider the maze as an array of elements; at each moment of its journey, it is only in one element. Let's call this element the `currentSpot`. It can be represented by a structure of two integer fields. The first field is the row and the second is the column coordinate of the spot in the maze. For example, the exit in Figure 3-25 is at (5,12)—that is, row 5, column 12.

The program begins by creating the maze. It then initializes the exit spot and prompts the user for the coordinates of the entrance spot. The program must be robust. If the user enters coordinates of a black spot, the program must request new coordinates until a white spot is entered. The mouse starts from the entrance spot and tries to reach the exit spot and its reward. Note, however, that some start positions do not lead to the exit.

As the mouse progresses through its journey, print its path. As it enters a spot, the program determines the class of that spot. The class of a spot can be one of the following:

a. Continuing—A spot is a continuing spot if one and only one of the neighbors (excluding the last spot) is a white spot. In other words, the mouse has only one choice.
b. Intersection—A spot is an intersection spot if two or more of the neighbors (excluding the last spot) is a white spot. In other words, the mouse has two or more choices.
c. Dead end—A spot is a dead-end spot if none of the neighbors (excluding the last spot) is a white spot. In other words, the mouse has no spot to choose. It must backtrack.
d. Exit—A spot is an exit spot if the mouse can get out of the maze. When the mouse finds an exit, it is free and receives a piece of cheese for a reward.

To solve this problem, you need two stacks. The first stack, the visited stack, contains the path the mouse is following. Whenever the mouse arrives at a spot, it first checks to see whether it is an exit. If not, its location is placed in the stack. This stack is used if the mouse hits a dead end and must backtrack. Whenever the mouse backtracks to the last decision point, also print the backtrack path.

When the mouse enters an intersection, the alternatives are placed in a second stack. This decision point is also marked by a special decision token that is placed in the visited stack. The decision token has coordinates of (–1,–1). To select a path, an alternative is then popped from the alternatives stack and the mouse continues on its path.

While backtracking, if the mouse hits a decision token, the token is discarded and the next alternative is selected from the alternatives stack. At this point print an asterisk (*) next to the location to show that the next alternative path is being selected.

If the mouse arrives at a dead end and both stacks are empty, the mouse is locked in a portion of the maze with no exit. In this case, print a trapped message and terminate the search for an exit.

After each trial, regardless of the outcome, the user should be given the opportunity to stop or continue.