Homework 3*
SNU 4190.210, Fall 2011
Kwangkeun Yi

due: 10/08(Mon), 24:00

The objectives of this homework :

- Learn to make sure about types while making programs.
- Implement inner parts of data.
- Implement inner parts of data in different ways.
- Learn to make programs with interface only, without being aware of implementation of inner parts.
- Learn to make programs with interface only, even when there are various ways to implement inner parts.

Exercise 1 "Verification of maze"

A maze quiz is placed occasionally in an appendix of a magazine. Imagine maze drawn in a paper. Let's think of a maze as following.

- Squares are jammed in a paper like a graph paper. Each square is considered as a room.
- Walls between adjacent rooms may be opened or closed.
- Entrance room and exit room is fixed.

Before a maze quiz is pubilshed in a magazine, editorial staffs would verify the existence of solution: whether there is a path from an entrance room to an exit room or not.

Define a function `maze-check` which does the verification. (This verificational function is simpler than a function which finds the solution of a maze quiz.)

$$\texttt{maze-check} : \mathit{maze} \times \mathit{room} \times \mathit{room} \rightarrow \mathit{bool}$$

`maze-check` receives a maze, an entrance room, and an exit room, and verifies whether there is a way which connects two rooms. Assume that the maze is finite and entrance and exit rooms are in the maze.

*translated by Youngseok Lee

While implementing the above function, you could implement it while not knowing how to implement a maze and a set.

$$\texttt{can-enter} : \mathit{room} \times \mathit{maze} \to \mathit{room\ list}$$
$$\texttt{same-room?} : \mathit{room} \times \mathit{room} \to \mathit{bool}$$

$$\texttt{empty set} : \mathit{room\ set}$$
$$\texttt{add-element} : \mathit{room} \times \mathit{room\ set} \to \mathit{room\ set}$$
$$\texttt{is-member?} : \mathit{room} \times \mathit{room\ set} \to \mathit{bool}$$
$$\texttt{is-subset?} : \mathit{room\ set} \times \mathit{room\ set} \to \mathit{bool}$$

`can-enter` outputs a list of all the adjacent rooms which can be reached from a given room. `same-room?` determines whether two given rooms are same. In this homework, you don't need to implement above six functions. □

Exercise 2 "Creating a maze"

Let's think of a graph paper that has $N \times M$ regular hexagonal rooms. A maze is defined as following.

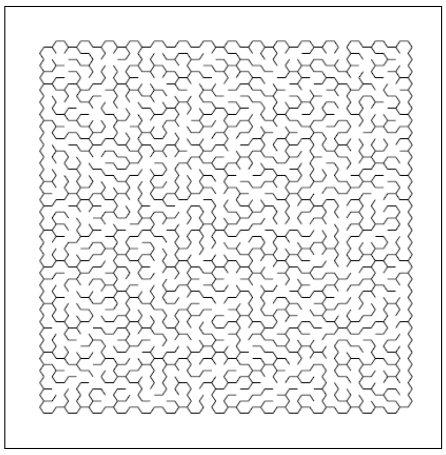
- Entrance and exit are at a room at the top row and the bottom row each.
- There is a unique path from entrance to exit.

Define a function `mazeGen` that creates a maze.

$$\texttt{mazeGen} : \mathit{int} \times \mathit{int} \to \mathit{maze}$$

That is, (`mazeGen` n m) takes positive numbers n and m and produces a maze in a $N \times M$ hexagonal graph paper.

For example, the following diagram is one of the hexagonal maze:



Define the above function using the following functions.

`init-maze` : $int \times int \rightarrow maze$
`open-n` : $int \times int \times maze \rightarrow maze$
`open-w` : $int \times int \times maze \rightarrow maze$
`open-se` : $int \times int \times maze \rightarrow maze$
`open-sw` : $int \times int \times maze \rightarrow maze$
`open-ne` : $int \times int \times maze \rightarrow maze$
`open-nw` : $int \times int \times maze \rightarrow maze$
`maze-pp` : $maze \rightarrow void$

`(init-maze` n m`)` is a maze with six walls of every room closed. All the $n \times m$ rooms are distinguished by its coordinate, and the coordinate would be from (0,0) to ($n-1$,$m-1$). `(open-`d n m M`)` produces a maze that has room (n,m) with its d-direction wall opened in maze M. `(maze-pp` M`)` draws the maze M beautifully.

For reference, when making a maze, the objective is to make it as difficult as possible. To make it simple, do as following. At first, let all the walls of rooms be opened from entrance to exit, then let other walls of rest of the rooms be appropriately opened to make it difficult.

However, the above method makes a relatively easy maze. The solution path can be seen easily, and a path which is not the solution can be relatively short. The better idea is to open walls of random rooms until the entrance is connected with the exit.□

Exercise 3 "Wallpaper design"

Structure of wallpaper pattern is usually a repetition of same patterns. A basic pattern is a square of black or white. The method of designing a pattern is to make a 4 times bigger square pattern by joining 4 small basic patterns together, then again, join 4 of this bigger pattern to make a greater pattern which is 4 times bigger, and so on. When you think it's done, put these designed squares in a paper repeatedly.

Define following functions which conceal the inner parts of these pattern data.

`black` : *pattern*
`white` : *pattern*
`glue` : $pattern \times pattern \times pattern \times pattern \rightarrow pattern$
`rotate` : $pattern \rightarrow pattern$
`neighbor` : $location \times pattern \rightarrow int$
`pprint` : $pattern \rightarrow void$

Each function works as following:

- `black`: A black pattern of basic size.
- `white`: A white pattern of basic size.
- `glue`: Receives 4 square patterns of same size in the order of NW, NE, SE, SW, and produces a square whose size equals to four times the size of given patterns, and links four given patterns in a given location.
- `rotate`: Receives a square pattern, and outputs a pattern which is rotated 90 degrees clockwise.

- `neighbor`: The number of black squares among maximum of 8 squares which surrounds the basic square in given location. The location of a basic square would be a list of numbers which is the number of the sector where the given square is located in, while dividing the original square into quarter of a size repeatedly. The area of NW would be 0, NE would be 1, SE would be 2, and SE would be 3. If the given pattern is a basic square, the location of the square will be an empty list. For example, a square which has location (3 3) would be a most bottom left basic square in a square which contains 16 basic squares. The number of basic squares would be 4^i, and the length of the list which represents the location of a basic square would be always i. `neighber` is defined only if these condition are satisfied.
- `pprint`: Draws a square pattern in a screen.

For instance, you can make and print the following wallpaper (Which pattern would be printed?)

```
(define B black)
(define W white)
(define Basic (glue B B B W))
(define (turn pattern i)
  (if (<= i 0) pattern (turn (rotate pattern) (- i 1))))
(define Compound (glue Basic (turn Basic 1) (turn Basic 2) (turn Basic 3)))
(pprint Compound)
```

There are two ways to implement patterns when you design a program like the one above:

- Represent by a list of rows of basic squares in a square pattern. For example, `Basic` would be expressed by `((B B) (W B))`, and `Compound` would be expressed by `((B B W B) (W B B B) (B B B W) (B W B B))` for the above example.
- Represent it as a tree structure with a leaf containing a basic square and every node branching out in 4 ways.

Implement these two ways. Define the above six functions with these two methods of implementation. You must use both methods appropriately to suit the representation of given data.

Functions of interface in case of an array implementation:

`glue-array-from-tree` : $pattern \times pattern \times pattern \times pattern \rightarrow pattern$
`glue-array-from-array` : $pattern \times pattern \times pattern \times pattern \rightarrow pattern$
`rotate-array` : $pattern \rightarrow pattern$
`neighbor-array` : $location \times pattern \rightarrow int$
`pprint-array` : $pattern \rightarrow void$
`is-array?` : $pattern \rightarrow bool$

Functions of interface in case of a tree implementation:

`glue-tree-from-tree` : $pattern \times pattern \times pattern \times pattern \to pattern$
`glue-tree-from-array` : $pattern \times pattern \times pattern \times pattern \to pattern$
`rotate-tree` : $pattern \to pattern$
`neighbor-tree` : $location \times pattern \to int$
`pprint-tree` : $pattern \to void$
`is-tree?` : $pattern \to bool$

□

Exercise 4 "Judge of Miss Wallpaper"

Define the following function in addition to functions which handle a pattern of wallpapers.

`equal` : $pattern \times pattern \to bool$
`size` : $pattern \to int$

`equal` determines whether two given patterns are same, and `size` output i when the number of basic squares are 4^i. Two patterns received by `equal` can be represented in different ways.

Define a function `beautiful` using the exposed functions.

`beautiful` : $pattern \to bool$

A function `beautiful` returns true either if the given pattern is symmetric about the center point or if the number of black basic squares adjacent to every basic square is greater than 1 and less than 6. □