Homework 5
SNU 4910.210 Fall 2014

Chung-Kil Hur

due: 11/09 (Sun) 24:00

Exercise 1 "Tile Design"

We have the following interface for designing tiles. Implement each function.

`black` : *tile*
`white` : *tile*
`glue` : $tile * tile * tile * tile \to tile$
`rotate` : $tile \to tile$
`neighbor` : $location * tile \to int$
`pprint` : $tile \to void$

The informal specifications of the functions are as follows.

- `black`: a black basic tile of size 1×1.
- `white`: a white basic tile of size 1×1.
- `glue`: takes four tiles t_1, t_2, t_3, t_4 of size $n \times n$ and makes a tile of size $2n \times 2n$ containing t_1, t_2, t_3, t_4 in the NW, NE, SE, SW corner, respectively.
- `rotate`: takes a tile and rotate it 90 degrees clockwise.
- `neighbor`: takes a location l and a (possibly composite) tile t and returns the number of black basic tiles around the basic tile located at l inside the tile t. The location of a basic tile inside a composite tile is recursively identified by a list of numbers in $\{0, 1, 2, 3\}$, where $0, 1, 2, 3$ represent the NW, NE, SE, SW corner, respectively. For example, the location `'(3 3)` points to the basic block located at the far south west corner of a composite

block of size 4×4. That is, a location inside a tile of size $2^i \times 2^i$ always has length i.

- `pprint`: pretty-prints the given tile on the screen.

For example, we can make a tile and print it on the screen as follows.

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

There are several ways to implement the interface for tile design.

- You can represent a tile as a list of rows, which are in turn lists of basic blocks. For example, the tile `Basic` above is represented as `((B B) (W B))` and `Compound` as `((B B W B) (W B B B) (B B B W) (B W B B))`.
- You can represent a tile as a tree where each internal node has four branches and each leaf has a basic tile.
- etc.

Implement the interface for tile design using both representations.
For the list representation, write the following functions:

`glue-array-from-tree` :	*tile* ∗ *tile* ∗ *tile* ∗ *tile* → *tile*
`glue-array-from-array` :	*tile* ∗ *tile* ∗ *tile* ∗ *tile* → *tile*
`rotate-array` :	*tile* → *tile*
`neighbor-array` :	*location* ∗ *tile* → *int*
`pprint-array` :	*tile* → *void*
`is-array?` :	*tile* → *bool*

For the tree representation, write the following functions:

`glue-tree-from-tree` :	*tile* ∗ *tile* ∗ *tile* ∗ *tile* → *tile*
`glue-tree-from-array` :	*tile* ∗ *tile* ∗ *tile* ∗ *tile* → *tile*
`rotate-tree` :	*tile* → *tile*
`neighbor-tree` :	*location* ∗ *tile* → *int*
`pprint-tree` :	*tile* → *void*
`is-tree?` :	*tile* → *bool*

Write the conversion functions between the two representations:

$$\begin{aligned}\texttt{array-to-tree} &: \quad \textit{tile} \to \textit{tile}\\ \texttt{tree-to-array} &: \quad \textit{tile} \to \textit{tile}\end{aligned}$$

When you implement the original external interface (two constants and four functions), properly use the above internal functions for the two representations. □

Exercise 2 "Beautiful Tiles"

Add the following two functions to the interface for tile design and implement them.

$$\begin{aligned}&\texttt{equal} : \textit{tile} * \textit{tile} \to \textit{bool}\\ &\texttt{size} : \textit{tile} \to \textit{int}\end{aligned}$$

`equal` check whether two tiles are the same. `size` returns i, given a tile of size $2^i \times 2^i$. Note that `equal` may take two tiles with different representations.

Only using the extended interface for tile design, write the function `beautiful` that checks whether a given tile is beautiful or not.

$$\texttt{beautiful} : \textit{tile} \to \textit{bool}$$

A tile is beautiful if it is symmetric with respect to its center, or every basic tile has at least two and at most five black neighbors. □

Exercise 3 "Turing machine"

We implement the Turing machine. See the following wiki page for the definition of Turing machine.

`http://en.wikipedia.org/wiki/Turing_machine`

For this, implement and use the following interfaces.

- The interface for making and using tapes:

$$\begin{aligned}&\texttt{init-tape} : \textit{symbol list} \to \textit{tape}\\ &\texttt{read-tape} : \textit{tape} \to \textit{symbol}\\ &\texttt{write-tape} : \textit{tape} * \textit{symbol} \to \textit{tape}\\ &\texttt{move-tape-left} : \textit{tape} \to \textit{tape}\\ &\texttt{move-tape-right} : \textit{tape} \to \textit{tape}\\ &\texttt{print-tape} : \textit{tape} \to \textit{void}\end{aligned}$$

symbol is the set of strings and "-"is considered as the blank symbol. In order to move the head left or right, you should move the tape in the other direction. `print-tape` prints the current symbol under the head.

- The interface for making and using execution rules:

  `empty-ruletable` : *ruletable*
  `add-rule` : $rule * ruletable \to ruletable$
  `make-rule` : $state * symbol * symbol * move * state \to rule$
  `match-rule` : $state * symbol * ruletable \to symbol \times move \times state$

  state is the set of strings and *move* is `'left`, `'right`, or `'stay`. `make-rule` makes a rule consisting of the current state, the symbol under the head, the symbol to write, the direction to move and the next state. `match-rule` maps the current state and the symbol under the head, by looking up the rule table, to the symbol to write, the direction to move and the next state.

- The interface for making and using Turing machines:

  `make-tm` : $symbol\ list * state * ruletable \to tm$
  `step-tm` : $tm \to tm$
  `run-tm` : $tm \to tm$
  `print-tm` : $tm * int \to void$

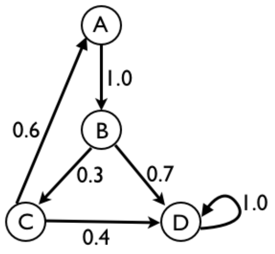
  `make-tm` initialize the tape with the given list of symbols, locate the head to point to the first symbol, set the current state to be the given state, and equip the Turing machine with the given ruletable. `step-tm` executes the given Turing machine one step and returns the resulting Turing machine. `run-tm` executes the given Turing machine until it terminates and returns the resulting Turing machine. Note that `run-tm` runs forever if the given Turing machine does not terminate. `print-tm` takes a Turing machine M and an integer n and prints out the $2n + 1$ symbols around the current head (*i.e.*, the n symbols before the head, the symbol under the head, and the n symbols after the head). When printing, use "." as the delimiter between tape symbols. □

Challenge 1 "Where is she?"

When she goes to the department store, she turns off her mobile phone. So, in order to find her easily, we are going to write a program that calculates the probability for her location. The inputs are (*i*) a graph that models her transitions between the shops and (*ii*) an integer for the time taken after she went to the department store.

The nodes in a graph represent the shops and its edges show the probability of her transitions between shops in every 10 minutes. Note that for each shop, the summation of the probabilities of its out-going edges must be 1. We assume that she randomly chooses the first shop to go (*i.e.*, the probability of each shop to be her first shop is the same).

For example, suppose the following graph models her transitions.



Then, the probabilities of her being in the shops `A`, `B`, `C`, and `D` after 10 minutes are 15%, 25%, 7.5% and 52.5%, respectively. After 20 minutes, 4.5%, 15%, 7.5% and 73%, respectively.

Define the function `catchYou` that calculates such probabilities.

$$\texttt{catchYou} : \mathit{graph} * \mathit{int} \rightarrow (\mathit{store} \times \mathit{real})\mathit{list}$$

We assume that there are exactly five shops (`A, B, C, D, E`) in the department store. The inputs are a graph for her transitions and an integer for how many 10 minutes she spent.

For example, the above graph is represented as follows:

```
(define model '((A B 1.0) (B C 0.3) (B D 0.7) (C A 0.6) (C D 0.4) (D
D 1.0)))
```

The output is a list of pairs of a shop name and the probability of her being in the shop. For instance,

```
(catchYou model 2)
```

outputs the following list:

```
((A . 4.5) (B . 15) (C . 7.5) (D . 73))
```

□