Homework 2
SNU 4910.210 Spring 2006

due: 4/17 24:00

Exercise 1 "미로 검증"

잡지에 가끔 미로 퀴즈가 부록으로 있다. 종이에 그려진 미로를 상상해보자. 그 미로를 다음과 같이 바라보자:

- 종이에 정4각형들이 빼곡히 채워져 있다(모눈종이). 각 정4각형은 하나의 방이다.
- 각 방들은 이웃한 방들과 사이의 벽들이 몇 개 터져 있기도 하고 막혀있기도 하다.
- 시작 방과 끝 방이 정해져 있다.

미로퀴즈를 잡지에 출판하기에 앞서, 편집진은 과연 미로퀴즈의 답이 있는 지 확인하는 과정을 거친다. 시작 방에서 끝 방으로 이어지는 길이 있는지.

그러한 검증을 하는 `maze-check` 함수를 정의해 보자. (이러한 검증함수는 실제로 미로를 찾아내 주는 함수보다 간단하다):

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

`maze-check`은 미로와 시작 방과 끝 방을 주면 그 두 방을 연결하는 길이 있는지를 확인해 준다. 이때 미로는 유한하고 시작 방과 끝 방은 항상 그 미로안에 있는 방이라고 가정한다.

위의 함수를 구현 할 때는 미로가 어떻게 구현되었는지, 집합은 어떻게 구

현되었는 지 알지 못하는 상태에서 다음의 함수들을 써서 구현할 수 있다:

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

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

`can-enter`는 미로의 주어진 방에서 갈 수 있는 이웃한 방들의 리스트를 준다. `same-room?`은 두 방이 같은 방인지를 판별해 준다. 위의 여섯 함수들은 이번 숙제에서는 구현하지 않는다. □.

Exercise 2 "종이 벽지 디자인"

벽지 무늬의 전체구조는 대게가 같은 무늬들의 반복이다. 기본 무늬는 검거나 흰 정사각형이다. 무늬를 디자인하는 작업은 기본 정사각형들 4개를 연결해서 4배 큰 정사각형 무늬를 만들고, 이것들 4개를 다시 연결해서 4배 더 큰 정사각형을 만들고, 등등. 되었다 싶으면 디자인된 정 사각형들을 반복해서 종이에 짜넣는 방법을 취한다.

이러한 무늬 데이타의 속 내용을 감추는 다음의 함수들을 정의하라.

$$\texttt{black} : \text{무늬}$$
$$\texttt{white} : \text{무늬}$$
$$\texttt{glue} : \text{무늬} \times \text{무늬} \times \text{무늬} \times \text{무늬} \rightarrow \text{무늬}$$
$$\texttt{rotate} : \text{무늬} \rightarrow \text{무늬}$$
$$\texttt{neighbor} : \text{위치} \times \text{무늬} \rightarrow \text{int}$$
$$\texttt{pprint} : \text{무늬} \rightarrow \text{void}$$

각 함수들이 하는 일은 다음과 같다:

- `black`: 기본크기의 검은 정사각형 무늬.
- `white`: 기본크기의 흰 정사각형 무늬.
- `glue`: 같은 크키의 정사각형 무늬 4개를 NW, NE, SE, SW방향의 순서로 받아서 그 위치에 놓고 연결한 4배 크기의 정사각형 무늬를 만든다.
- `rotate`: 정사각형 무늬를 받아서 90도 시계방향으로 돌려진 무늬를 만든다.

- `neighbor`: 주어진 위치의 기본 정사각형의 주변에 있는 최대 8개의 정사각형중 검은 정사각형의 갯수. 기본 정사각형의 위치는 전체 정사각형에서 부터 시작해서 계속 4등분 해 가면서 그 정사각형이 포함된 구역의 번호들의 리스트이다. NW 구역은 0, NE 구역은 1, SE 구역은 2, SW 구역은 3이다. 무늬가 기본 정사각형 하나일 때는 그 정사각형의 위치는 빈 리스트가 된다. 예를 들어, 위치가 (3 3) 인 정사각형은 16개의 기본 정사각형으로 구성된 정사각형 판에서 가장 왼쪽 아래의 기본 정사각형을 말한다. 한 무늬가 가지는 기본 정사각형의 갯수는 4^i개 이고, 기본 정사각형의 위치를 표현하는 리스트의 길이는 항상 i가 된다. 이 조건을 만족할 때에만 `neighbor`가 정의된다.

- `pprint`: 정사각형 무늬를 화면에 그려준다.

예를 들어서 다음과 같이 벽지무늬들을 만들어서 프린트할 수 있겠다 (어떤 무늬가 프린트될까?)

```
(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))))
(pprint (glue Basic (turn Basic 1) (turn Basic 2) (turn Basic 3)))
```

위와 같은 프로그램을 고안하는 데, 무늬를 구현하는 방법이 두가지가 있다:

- 정사각형을 기본 정사각형들의 배열로(리스트의 리스트?) 표현하는 방법. 예를 들어, 위의 예에서 `Basic`은 `((B B) (W B))`로, 프린트 된 무늬는 `((B B W B) (W B B B) (B B B W) (B W B B))`로 표현되겠다.

- 잎새에 기본 정사각형이 매달린, 모든 가지가 4갈래로 갈라지는 트리 구조로 표현하는 방법.

이 두가지 구현 방안을 속내용을 감추면서(data abstraction) 다음의 함수들을 구현하고, 위의 여섯가지 함수를 정의할 때 데이타의 표현방식에 맞는 적절한 함수들을 사용하도록 정의한다.

배열로 구현하는 경우, 드러나는 함수들:

$$\begin{aligned}
\texttt{glue-a-from-t}: &\quad \text{무늬} \times \text{무늬} \times \text{무늬} \times \text{무늬} \to \text{무늬} \\
\texttt{glue-a-from-a}: &\quad \text{무늬} \times \text{무늬} \times \text{무늬} \times \text{무늬} \to \text{무늬} \\
\texttt{rotate-a}: &\quad \text{무늬} \to \text{무늬} \\
\texttt{neighbor-a}: &\quad \text{위치} \times \text{무늬} \to \text{int} \\
\texttt{pprint-a}: &\quad \text{무늬} \to \text{void} \\
\texttt{is-a?}: &\quad \text{무늬} \to \text{bool}
\end{aligned}$$

트리로 구현하는 경우, 드러나는 함수들:

$$\begin{aligned}
\texttt{glue-t-from-t}: &\quad \text{무늬} \times \text{무늬} \times \text{무늬} \times \text{무늬} \to \text{무늬} \\
\texttt{glue-t-from-a}: &\quad \text{무늬} \times \text{무늬} \times \text{무늬} \times \text{무늬} \to \text{무늬} \\
\texttt{rotate-t}: &\quad \text{무늬} \to \text{무늬} \\
\texttt{neighbor-t}: &\quad \text{위치} \times \text{무늬} \to \text{int} \\
\texttt{pprint-t}: &\quad \text{무늬} \to \text{void} \\
\texttt{is-t?}: &\quad \text{무늬} \to \text{bool}
\end{aligned}$$

Exercise 3 "벽지 아가씨 심사위원"

벽지 무늬를 다루는 함수들에 다음의 함수를 추가로 정의하고:

$$\begin{aligned}
&\texttt{equal} : \text{무늬} \times \text{무늬} \to \text{bool} \\
&\texttt{size} : \text{무늬} \to \text{int}
\end{aligned}$$

`equal`은 두 무늬가 같은지를 판별하고, `size`는 기본 정사각형의 갯수가 4^i일 때 i를 내놓는다. `equal`이 받아들이는 두개의 무늬들은 다르게 표현된 것들일 수 있다.

그렇게 드러난 함수들을 이용해서 함수 `beautiful`을 정의하라.

$$\texttt{beautiful} : \text{무늬} \to \text{bool}$$

함수 `beautiful`은 벽지 무늬가 중앙점을 기준으로 대칭이거나, 대칭이지 않다면 모든 정사각형의 이웃한 검은 정사각형들의 갯수가 1개보다 많고 6개보다 작을 때이다. □

Exercise 4 "어울리지 않아"

스트링은 0과 9사이의 정수들의 리스트이다: 예) 0000, 1102201, 998011199 등. 빈 스트링은 없다. "스트링 s가 코드 c와 어울린다"는 것은 코드 c가 표현하는 스트링 집합에 s가 포함된다는 뜻이다. 코드 c 는 다음과 같이 정의되고:

$$c \quad \to \quad 0 \mid 1 \mid \cdots \mid 9 \mid c \cdot c \mid c|c \mid c\texttt{+}$$

코드 c가 뜻하는 스트링의 집합 $[\![c]\!]$는 다음과 같이 정의된다:

$$\begin{aligned}
[\![0]\!] &= \{0\} \\
&\vdots \\
[\![9]\!] &= \{9\} \\
[\![c_1 \cdot c_2]\!] &= \{s_1 s_2 \mid s_1 \in [\![c_1]\!], s_2 \in [\![c_2]\!]\} \\
[\![c_1 \,|\, c_2]\!] &= [\![c_1]\!] \cup [\![c_2]\!] \\
[\![c\texttt{+}]\!] &= [\![c]\!] \cup [\![c \cdot c]\!] \cup [\![c \cdot c \cdot c]\!] \cup \cdots
\end{aligned}$$

코드 데이타의 속내용을 감추는 다음의 함수들을 정의하고:

`atom` : int → code	`is-atom?` : code → bool
`dot` : code × code → code	`is-dot?` : code → bool
`bar` : code × code → code	`is-bar?` : code → bool
`plus` : code → code	`is-plus?` : code → bool
`de-atom` : code → int	`de-plus` : code → code
`de-dot-0` : code → code	`de-bar-0` : code → code
`de-dot-1` : code → code	`de-bar-1` : code → code

하는 일은:

$$\begin{aligned}
\texttt{(de-atom (atom } n\texttt{))} &= n \\
\texttt{(de-dot-}i\texttt{ (dot } c_0\ c_1\texttt{))} &= c_i \\
\texttt{(de-bar-}i\texttt{ (bar } c_0\ c_1\texttt{))} &= c_i \\
\texttt{(de-plus (plus } c\texttt{))} &= c
\end{aligned}$$

위의 함수들을 이용해서, 주어진 스트링 s가 코드 c와 어울리는 지를 결정하는 함수

`match` : 스트링 × 코드 → bool

를 정의하라. 예를 들어, `(match` 1001 1·0+·1`)`는 참을, `(match` 1001 (10)+·1`)`은 거짓을 낸다. 스트링은 정수들의 리스트로 구현한다: 예) 1001은 `'(1 0 0 1)`로. □

Exercise 5 "사전작업"

사전구조는 서로 연관된 키와 값들의 집합이다. 전산학에서 아주 널리 사용되며 대부분의 고급언어에서 라이브러리의 형태로 지원한다. 다음 문제를 풀기위한 사전작업으로 사전구조를 만들어 보자.

저장된 값들을 가지고 다니는 사전은 자유롭게 정의한다.

$$Dict_{Key,Value} \in 2^{Key \mapsto Value}$$

아래와 같은 함수들과 아무것도 들어있지 않은 빈 사전 `emptyDict`를 정의하라.

$$\texttt{dictLookup} : Dict \times Key \rightarrow Value$$
$$\texttt{dictInsert} : Dict \times Key \times Value \rightarrow Dict$$
$$\texttt{dictRemove} : Dict \times Key \rightarrow Dict$$
$$\texttt{dictMap} : Func \times Dict \rightarrow Dict$$
$$\texttt{dictFilter} : Pred \times Dict \rightarrow Dict$$

`dictMap`은 함수와 사전을 받아 사전이 가진 모든 값들(키가 아닌)에 인자로 받은 함수를 적용하고 그 결과를 돌려준다. `dictFilter`는 참, 거짓을 알려주는 함수와 사전을 받아 사전이 가진 모든 값들(키가 아닌)에 적용하여 참이 되는 값만 남겨 돌려주는 일을 한다. 그 밖의 함수 의미는 각각의 이름으로 알 수 있을 것이다. 다음은 간단한 사용 예이다.

```
(define dict1 (dictInsert emptyDict (cons 1 2) "first"))
(define dict2 (dictInsert dict1 (cons 3 4) "second"))
(dictLookup dict2 (cons 1 2))
"first"
```

□

Exercise 6 "보드게임매니아"

여러분은 보드게임을 좋아한다. 훌륭한 프로그래머인 여러분은 보드게임을 기계가 풀 수 있도록 만들고 싶다. 이번에 가지고 놀 게임은 Peg Solitaire다. 게임의 규칙은 다음과 같다. 여러분은 여러개의 구멍이 뚫린 게임판을 받게 된다. 그리고 몇몇 구멍들에는 막대가 들어있다. 이 막대들을 한 개만 남기고 모두 제거하는 것이 게임의 목표이다. 막대들은 다른 자리로 옮길 수 있는데 하나의 막대가 다른 막대를 뛰어 넘게되면 사이에 끼인 막대는 사라진다. 다음 세 조건을 만족하면 막대를 옮길 수 있다.

1. 막대는 위, 아래, 왼쪽, 오른쪽으로 두 칸 떨어진 곳에 옮긴다.
2. 옮기게 될 목표지점은 항상 빈 구멍이어야 한다.
3. 원래 있던 자리와 새롭게 옮기게 될 자리 사이에는 다른 어떤 막대가 끼어 있어야 한다.

앞서 정의한 사전구조로 게임판을 나타낸다. 키는 구멍의 좌표가 되고 값은 그 구멍에 막대가 있는지 없는지를 표시한다. 프로그램의 입력은 아래와 같다.

```
[(cons (cons -1 3) "Hole"), (cons (cons 0 3) "Hole"), ...,
(cons (cons -1 1) "Peg")]
```

이 문제를 풀기 위하여 다음 함수들을 정의한다.

$$Board = Dict_{Pos,Stat} \times Pos\ list$$

$$\texttt{loadBoard} : Input \times Board \rightarrow Board$$

Board는 게임판의 세팅을 타나내는 사전과 막대들의 위치만 따로 모은 리스트의 튜플이다. `loadBoard`는 입력과 빈 Board를 받아 세팅이 끝난 Board를 만들어 돌려주는 함수이다.

$$Move = Pos \times Dir$$

$$Dir = \text{“}Up\text{”}|\text{“}Down\text{”}|\text{“}Left\text{”}|\text{“}Right\text{”}$$

$$\texttt{findMvs} : Board \rightarrow Move\ list$$

`findMvs`는 Board를 받아 현재 상황에서 한 스텝안에 일어날 수 있는 모든 움직임을 모은 리스트를 돌려주는 함수이다.

$$\texttt{makeMove} : Board \times Move \rightarrow Board$$

`makeMove`함수는 Board를 받아 Move대로 움직인 후의 Board를 만들어 준다.

$$\texttt{solver} : Integer \times Board \times Move\ list \rightarrow Move\ list\ list$$

`solver`는 Board의 상황에서 첫번째 인자의 수 만큼 움직여 단 한 개를 제외한 모든 막대를 없앨 수 있는 움직임이 존재하는지를 알려준다. 이때 첫 번째 움직임은 세번째 리스트에 있는 Move들 중 하나여야 한다.

$$\texttt{solve} : Integer \times Board \rightarrow Move\ list\ list$$

`solve`는 정수 n과 Board를 받아 주어진 Board의 상황에서 n번의 움직임 안에 구할 수 있는 해가 존재하는지 계산한다. 위에서 정의한 solver 함수를 사용할 것.

지금까지 설명한 `loadBoard`, `findMvs`, `makeMove`, `solver`, `solve` 함수를 만들어라. 단, 게임판의 내용을 담고 있는 사전구조에 접근할 때는 항상 이전 문제에서 구현한 함수들만 사용 하도록 한다. □