Homework 2

SNU 4190.310, 2022 봄

이 광근

Due: 3/25(금), 24:00

Exercise 1 "반복기"

다음함수 `iter`를 정의하세요:

$$\texttt{iter}(n, f) = \underbrace{f \circ \cdots \circ f}_{n}$$

이 때, $n = 0$이면 아무일을 하지 않는(identity) 함수를 내 놓고, 양수이면 그 만큼 f를 반복해서 적용하는 함수를 내 놓습니다. 그래서,

```
iter(n, function x -> 2+x) 0
```

은 $2 \times n$을 내 놓게됩니다. □

Exercise 2 "대진표 스트링"

일반적으로 게임 대진표는 완전한 이진 나무구조(complete binary tree)입니다. 2022 얼드컵 팀들과 그 대진표를 다음과 같이 정의했습니다:

```
type team = Korea | France | Usa | Brazil | Japan | Nigeria | Cameroon
          | Poland | Portugal | Italy | Germany | Norway | Sweden | England
          | Argentina
```

```
type tourna = LEAF of team
            | NODE of tourna * tourna
```

tourna를 받아서 괄호를 이용한 1차원 스트링으로 변환해주는 함수 parenize를 작성하세요:

```
parenize: tourna -> string
```

예를들어,

```
parenize(NODE(NODE(LEAF Korea, LEAF Portugal), LEAF Brazil))
= "((Korea Portugal) Brazil)"
```

□

Exercise 3 "Mathemadiga"

고등학교때는 손으로하고, Maple이나 Mathematica에서는 자동으로 해주던 미분식 전개를 만들어봅시다.

```
diff: ae * string -> ae
```

diff는 식(algebraic expression)과 변수를 받아서 주어진 식을 변수로 미분한 결과 식을 돌려 줍니다. 예를들어, 식 $ax^2 + bx + c$을 x에 대해 미분시키면 $2ax + b$를 내 놓는 것입니다. 미분된것을 될 수 있으면 최소의 꼴로 줄이거나 등등의 작업을 하는 것은 자유입니다. 미분할 식은 다음의 ae 타입입니다:

```
type ae = CONST of int
        | VAR of string
        | POWER of string * int
        | TIMES of ae list
        | SUM of ae list
```

□

Exercise 4 "우선큐"

"우선큐"(priority queue)라는 구조는 원소들 사이의 순위가 "입장 순이 아니라

능력 순"이라는 것입니다. 원소들이 들어온 순서를 기준으로 우선순위가 매겨지지 (예, 스택이나 큐) 않고 원소마다 고유의 우선순위가 지정되는 것입니다.

우선큐는 대개 원소를 넣고 빼는것 보다는, 제일가는 원소를 알아보는 데에 유난히 특화됩니다. 힙(heap)이 대표적인 구현방법입니다. 그중에서도 왼쪽으로 쏠린 힙(leftist heap, 왼쏠힙)이라는 것을 구현해 봅시다.

- 왼쏠힙: 힙은 힙인데 모든 왼쪽 노드의 급수가 오른쪽 형제 노드의 급수보다 크거나 같다.
- 노드의 급수: 그 노드에서 오른쪽으로만 타고 내려가서 끝날 때 까지 내려선 횟수, 즉 오른편 척추의 길이.
- 힙: 이진 나무 구조로서 모든 갈래길 길목의 값이 갈라진 후의 모든 노드들의 값보다 작거나 같다.

왼쏠힙은 다음의 타입으로 정의됩니다:

```
type heap = EMPTY | NODE of rank * value * heap * heap
and rank = int
and value = int
```

넣고, 빼고, 하는 등의 함수는 다음으로 정의됩니다:

```
exception EmptyHeap
let rank h = match h with
  | EMPTY -> -1
  | NODE(r,_,_,_) -> r
let insert(x,h) = merge(h, NODE(0,x,EMPTY,EMPTY))
let findMin h = match h with
  | EMPTY -> raise EmptyHeap
  | NODE(_,x,_,_) -> x
let deleteMin h = match h with
  | EMPTY -> raise EmptyHeap
  | NODE(_,x,lh,rh) -> merge(lh,rh)
```

나머지 함수 `merge`

```
merge: heap * heap -> heap
```

를 정의하세요. 이 때, 왼쏠힙의 장점을 살려서 여러분이 정의한 merge는 $O(\log n)$으로 끝나도록 해야 합니다 (n은 힙의 노드 수). (참고사실: 왼쏠힙에서 오른쪽 척추에 붙어있는 노드수는 많아야 $\lfloor \log(n+1) \rfloor$입니다.) 정의할 때 다음의 함수를 이용하시기를:

```
let shake (x,lh,rh) = if (rank lh) >= (rank rh)
                      then NODE(rank rh + 1, x, lh, rh)
                      else NODE(rank lh + 1, x, rh, lh)
```

□

Exercise 5 "짚-짚-나무"

임의의 나무를 여러분 바지의 "지퍼"로 구현할 수 있답니다.

- 나무구조 타입은 아래와 같이 정의되겠지요:

```
type item = string
type tree = LEAF of item
          | NODE of tree list
```

- 아래의 zipper가 나무의 줄기를 타고 자유자재로 찢어놓기도 하고 붙여놓기도 합니다.

```
type zipper = TOP
            | HAND of tree list * zipper * tree list
```

  현재 나무줄기의 어느지점에 멈춰 있는 지퍼손잡이 HAND(l,z,r)에서, l은 왼편 형제 나무들(elder siblings)이고 r은 오른편 형제 나무들(younger siblings)이다.

- 나뭇가지에서의 현재 위치 location는 현재위치를 뿌리로하는 나무자체와 지퍼(zipper)로 표현되는 주변 나무들로 구성된다.

```
type location = LOC of tree * zipper
```

- 예를들어, "$a \times b + c \times d$" 가 다음과 같은 나무구조로 표현될 것이다. 모든 심볼은 항상 잎새에 매달리게 된다.

```
NODE [ NODE [LEAF a; LEAF *; LEAF b];
       LEAF +;
```

```
                NODE [LEAF c; LEAF *; LEAF d]
              ]
```

두번째 곱셈표에의 위치는 다음과 같다:

```
LOC (LEAF *,
     HAND([LEAF c],
          HAND([LEAF +; NODE [LEAF a; LEAF *; LEAF b]],
               TOP,
               []),
          [LEAF d]))
```

- 자, 주어진 위치에서 이제 자유자재로 나무를 탈 수 있습니다. 왼편으로 옮겨 가는 것은 다음과 같지요:

```
let goLeft loc = match loc with
    LOC(t, TOP) -> raise (NOMOVE "left of top")
  | LOC(t, HAND(l::left, up, right)) -> LOC(l, HAND(left, up, t::right))
  | LOC(t, HAND([],up,right)) -> raise NOMOVE "left of first"
```

- 다음의 나머지 함수들을 정의하세요:

```
goRight: location -> location
goUp: location -> location
goDown: location -> location
```

□

Exercise 6 "Queue = 2 Stacks"

큐는 반드시 하나의 리스트일 필요는 없습니다. 두개의 스택으로 큐를 효율적으로 규현할 수 있습니다. 큐에 넣고 빼는 작업이 거의 한 스텝에 이루어질 수 있습니다. (하나의 리스트위를 더듬는 두 개의 포인터를 다루었던 C의 구현과 장단점을 비교해 보세요.)

각각의 큐 연산들의 타입들은:

```
emptyQ: queue
enQ: queue * element -> queue
deQ: queue -> element * queue
```

큐를 $[a_1; \cdots; a_m; b_1; \cdots; b_n]$ 라고 합시다 (b_n이 머리). 이 큐를 두개의 리스트 L과 R로 표현할 수 있습니다:

$$L = [a_1; \cdots; a_m], \quad R = [b_n; \cdots; b_1].$$

한 원소 x를 삼키면 새로운 큐는 다음이 됩니다:

$$[x; a_1; \cdots; a_m], [b_n; \cdots; b_1].$$

원소를 하나 빼고나면 새로운 큐는 다음이 됩니다:

$$[a_1; \cdots; a_m], [b_{n-1}; \cdots; b_1].$$

뺄 때, 때때로 L 리스트를 뒤집어서 R로 같다 놔야하겠습니다. 빈 큐는 $([], [])$ 이겠지요.

다음과 같은 Queue 타입의 모듈을 작성합니다:

```
module type Queue =
  sig
    type element
    type queue
    exception EMPTY_Q
    val emptyQ: queue
    val enQ: queue * element -> queue
    val deQ: queue -> element * queue
  end
```

다양한 큐 모듈이 위의 Queue 타입을 만족시킬 수 있습니다. 예를들어:

```
module IntListQ =
   struct
     type element = int list
     type queue = ...
     exception EMPTY_Q
     let emptyQ = ...
     let enQ = ...
     let deQ = ...
   end
```

는 정수 리스트를 큐의 원소로 가지는 경우겠지요. 위의 모듈에서 함수 enQ와 deQ를 정의하기 바랍니다.

이 모듈에 있는 함수들을 이용해서 큐를 만드는 과정의 예는:

```
let myQ = IntListQ.emptyQ
let yourQ = IntListQ.enQ(myQ, [1])
let (x,restQ) = IntListQ.deQ yourQ
let hisQ = IntListQ.enQ(myQ, [2])
```

□