Homework 6
SNU 4910.210 Fall 2008

Kwangkeun Yi

due: 12/04(Thu) 24:00

Exercise 1 "식계산기"

정수식들의 구조를 다음의 타입으로 정의했습니다:

```
type expr = NUM of int
          | PLUS of expr * expr
          | MINUS of expr * expr
          | MULT of expr * expr
          | DIVIDE of expr * expr
          | MAX of expr list
```

주어진 `expr`를 받아서 정수값을 만들어내는 함수 `eval`

```
eval: expr -> int
```

를 정의하세요. 이때, `MAX [1,3,2] = 3`, 즉 `MAX`는 정수식 리스트에서 가장 큰 정수를 찾아내는 정수식입니다. 빈 리스트의 경우는 0을 의미하는 정수식 입니다.

□

Exercise 2 "불확정성의 언어 미라쥐"

다음과 같은 언어 Mirage 를 생각합시다.

program	$\rightarrow$	c	
c	$\rightarrow$	`x` $:= e$	assignment
	$\mid$	c ; c	sequence
	$\mid$	c^*	non-deterministic 0 or more repetitions
	$\mid$	$c \cup c$	non-deterministic choice
	$\mid$	`x eq` e ? c	check
	$\mid$	`x neq` e ? c	check
e	$\rightarrow$	n	integer
	$\mid$	$e + e$	addition
	$\mid$	$e - e$	substraction
	$\mid$	`x`	

각 명령어는 기계상태를 변환시켜줍니다. 기계는 프로그램 변수의 정수값을 가지고 있습니다. 프로그램 변수는 오직 하나만 있습니다.

- "$x := e$"는 현재 기계상태에서 e를 계산해서 x의 값으로 기계상태를 변환
- "c_1 ; c_2"는 c_1 실행 후 결과 상태를 가지고 c_2를 실행
- "c^*"는 c를 0번 이상 임의의 횟수를 반복
- "$c_1 \cup c_2$"는 c_1이나 c_2중 하나를 선택해서 실행
- "x `eq` e ? c"는 현재상태에서 x의 값이 e와 같은 지 확인. 같으면 c를 실행, 틀리면 건너뜀
- "x `neq` e ? c"는 현재상태에서 x의 값이 e와 같은 지 확인. 다르면 c를 실행, 같으면 건너뜀
- 변수가 가질 수 있는 값은 -5 에서 +5 까지의 정수. 그 이외의 값을 변수가 가지게 되면 현재상태로 실행 끝

Mirage 프로그램의 싹쓸이(exhaustive) 실행기 `exeval`

$$\texttt{exeval} : \texttt{pgm} -> \texttt{state} -> \texttt{statelist}$$

을 작성합시다. "싹쓸이"란 모든 가능한 경우를 실행하는 것을 말합니다.

```
type pgm = cmd
and cmd = ASSIGN of exp
```

```
          | SEQUENCE of cmd * cmd
          | REPEAT of cmd
          | CHOICE of cmd * cmd
          | EQ of exp * cmd
          | NEQ of exp * cmd
and exp = NUM of int
          | ADD of exp * exp
          | SUB of exp * exp
          | VAR
type state = int
```

예를들어,

```
x := 1; ((x eq 1? x := x+1) U (x neq 1? x := x-1))*
```

를 실행하면 결과 상태들은 `[1, 2]`.

또다른 예로,

```
x := 1; (x := x+1)*
```

를 실행하면 결과 상태는 `[1,2,3,4,5]`.

□

Exercise 3 "Leftist Heaps: 왼쪽편에 쏠려있는 힙"

우선큐(priority queue, "유별난 큐")라는 구조의 핵심은, 원소를 넣고 빼는 것 보다는, 제일가는 원소를 알아보는 데에 유난히 특화되 있다는 것입니다. 힙(heap)이 대표적인 것이지요. 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
fun rank EMPTY = -1
  | rank NODE(r,_,_,_) = r
fun insert(x,h) = merge(h, NODE(0,x,EMPTY,EMPTY))
fun findMin EMPTY = raise EmptyHeap
  | findMin NODE(_,x,_,_) = x
fun deleteMin EMPTY = raise EmptyHeap
  | deleteMin 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:

```
fun 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 4 "파토"

게임(예, 고스톱)을 진행하다가 중간에 문제가 생기면 "이번 판은 무효"로 진행을 멈추게됩니다. 예외상황(exception)으로 다루어지는 경우입니다.

다음의 두 갈래 나무구조 tree를 생각하자:

```
type tree = Node of int * tree * tree
          | Leaf of int
          | Empty
```

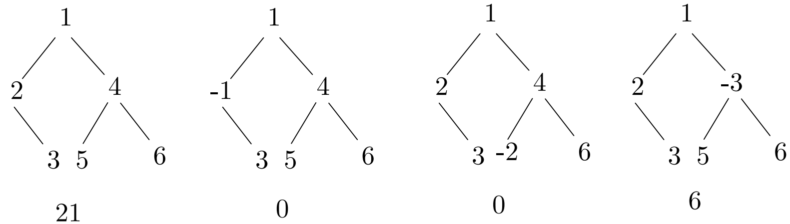
함수 sum을 생각하자:

$$\texttt{sum} : \texttt{tree} \to \texttt{int}$$

함수 sum은 입력 나무의 노드에 기록된 정수들을 모두 합하는 데, 다음과 같이 합해야 한다:

- `sum`(t) 계산 중에 왼쪽 하부트리(left sub-tree)의 뿌리에서 만나는 정수값이 음수이면 `sum`(t)의 최종 결과는 0.
- `sum`(t) 계산 중에 오른쪽 하부트리(right sub-tree)의 뿌리에서 만나는 정수값이 음수이면 그 하부트리의 `sum`값은 0으로 한다.

예를 들어 다음의 나무구조들에서 `sum`의 값은 각각 21, 0, 0, 6이다.



위이 조건을 만족하도록 `sum`의 두 버전을 정의하라. `sum-straight`는 예외상화을 사용하지 않고 정의하고, `sum-exception`은 예외상황을 이용해서 작성하라. □

Exercise 5 "이걸로"

nML에서는 예외상황을 일으킬 때 임의의 값을 예외상황에 싣고 예외상황처리기로 전달할 수 있다.

위의 `sum-exception`을 다시 정의해서, 다음의 조건이 만족하도록 하라:

- `sum-exception`(t) 계산 중에 왼쪽 하부트리(left sub-tree)의 뿌리에서 만나는 정수값이 음수이면 최종 결과는 0.
- `sum-exception`(t) 계산 중에 오른쪽 하부트리(right sub-tree)의 뿌리에서 만나는 정수값이 음수이면 그 하부트리의 `sum-exception`값은 그 음수값으로 한다.

위의 세 가지 나무구조들에서 위의 `sum-exception`을 돌리면 각각 21, 0, 0, 3을 계산한다. □

Exercise 6 "Queue = 2 Stacks"

큐는 반드시 하나의 리스트일 필요는 없습니다. 두개의 리스트로 큐를 효율적으로 규현할 수 있습니다. 큐에 넣고 빼는 작업이 거의 한 스텝에 이루어질 수 있지요.

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 갖다 놔야하겠습니다. 빈 큐는 $([], [])$ 이겠지요.

다음과 같은 Queue 타입의 모듈을 작성합니다:

```
signature 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:

```
structure IntListQ =
  struct
    type element = int list
    type queue = ...
    exception EMPTY_Q
    val emptyQ = ...
```

```
        val enQ = fn ...
        val deQ = fn ...
    end
```

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

이 모듈에 있는 함수들을 이용해서 큐를 만드는 과정의 예는:

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

□