Homework 2

SNU 4190.310, Fall 2014

Kwangkeun Yi

due: 10/07, 24:00

Exercise 1 "CheckMetroMap"

Consider the following datatype `metro` :

```
type metro = STATION of name
           | AREA of name * metro
           | CONNECT of metro * metro
and name = string
```

Define a function `checkMetro` that checks if a given `metro` is well-formed.:

```
checkMetro: metro -> bool
```

A `metro` is well-formed if and only if every metro station's name (*id* in STATION(*id*)) in the `metro` only occurs in metro area (*m* in AREA(*id*, *m*)) that encompasses the STATION(*id*) with a same *id*.

For example, well-formed `metro` values are:

- `AREA("a", STATION "a")`
- `AREA("a", AREA("a", STATION "a"))`
- `AREA("a", AREA("b", CONNECT(STATION "a", STATION "b")))`

- `AREA("a", CONNECT(STATION "a", AREA("b", STATION "a")))`

Ill-formed ones are:

- `AREA("a", STATION "b")`
- `AREA("a", CONNECT(STATION "a", AREA("b", STATION "c")))`
- `AREA("a", AREA("b", CONNECT(STATION "a", STATION "c")))`

□

Exercise 2 "Leftist heap"

"Priority queue" means the order between its elements is not an order of entrance but the order of ability. Priority is not ranked based on the ranking order of the elemtnts are entered, each element is assigned a unique priority.

Therefore priority queue must be specialized to find out the best element among the elements in the heap. Heap is a typical example. Let's implement particular heap, leftist heap from them.

- Leftist heap: The priority of every left node is greater than or equal to the right sibling node's priority.
- Priority of a node: The count of going down the right child to countinue to a leaf. ie. the length of right spine.
- Heap: This is binary tree structure. The value of every node having forked road is less than or equal to the values of all the nodes after the split.

Leftist heap is defined in this way:

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

inserting, deleting functions are defined as follows:

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

```
      let deleteMin = function EMPTY -> raise EmptyHeap
                             | NODE(_,x,lh,rh) -> merge(lh,rh)
```

Implement the other function `merge`

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

The complexity of computation of the `merge` you made should be $O(\log n)$ taking advantage of leftist heap. (n is the number of nodes of a heap) (Note: The number of nodes attached in right spine in a leftist heap is at most $\lfloor \log(n+1) \rfloor$) Use the following function when you make the `merge`:

```
    let shake = function (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 3 "The zipper"

All trees can be implemented by your clothing's "zipper".

- We can represent trees in OCaml using values of the following datatype:

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

- Following `zipper` makes a tree torn or joined.

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

  A `HAND(l,z,r)` contains its list `l` of elder siblings (starting with the youngest), its father zipper `z`, and its list `r` of younger siblings (starting with the eldest)

- A `location` in the `tree` addresses a subtree, together with the `zipper`

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

- Assume we consider the parse tree of arithmetic expressions, with string items. The expression "$a \times b + c \times d$" parses as the tree.

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

The location of the second multiplication sign in the tree is as follows:

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

- Now we can navigate. For example, when you want to move left from your location, it could be done as follows.

```
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)"
```

- Define other navigating functions:

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

`goDown` go down the left-most child node.

□

Exercise 4 (10pts) "Galculator"

Make following calculator,

```
galculator: exp -> float
```

```
exception FreeVariable
      type exp = X
               | INT of int
               | REAL of float
               | ADD of exp * exp
               | SUB of exp * exp
               | MUL of exp * exp
               | DIV of exp * exp
               | SIGMA of exp * exp * exp
               | INTEGRAL of exp * exp * exp
```

The following are examples of representing expressions as a `exp`:

$\sum_{x=1}^{10}(x * x - 1)$	`SIGMA(INT 1, INT 10, SUB(MUL(X, X), INT 1))`
$\int_{x=1.0}^{10.0}(x * x - 1)\mathrm{d}x$	`INTEGRAL(REAL 1.0, REAL 10.0, SUB(MUL(X, X), INT 1))`

Use the rectangle method with the top-left corner approximation to evaluate an integral value. (See `http://en.wikipedia.org/wiki/Rectangle_method`) Keep the length of subintervals(dx) as 0.1. Raise exception Error whenever an expression is invalid.

- Evaluating `SIGMA(a,b,f)`, return 0 when $a > b$.
- Evaluating `INTEGRAL(a,b,f)`, evaluate `-INTEGRAL(b,a,f)` when $a > b$.
- Cast float values of `a,b` to integer using `int_of_float` evaluating `SIGMA`
- Use `float_of_int` for type casting.
- Do not worry about that some abmbiguous situations occur because of type casting, if those the situations are not mentioned.
- Evaluating `INTEGRAL`, the height of rectangle should be the function value of left-position.(`[a,b] -> f(a)`)
- Variables are the nearest bound. (Raise exception `FreeVariable` evaluating an no bound variable.)

□

Exercise 5 (20pts) "Calculator interpreter"

Implement module Zexpr following next signature.

```
signature ZEXPR =
sig
  exception Error of string
  type id = string
  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
            | VAR of id
            | LET of id * expr * expr
  type environment
  type value
  val emptyEnv: environment
  val eval: environment * expr -> value
  val int_of_value : value -> int
end
module Zexpr:ZEXPR = (*remove ‘‘:ZEXPR’’ when you submit*)
struct
  (*Implement here*)
end
```

Let E be an expression of the type `ZEXPR.expr`,

`Zexpr.eval (Zexpr.emptyEnv,` E`)`

is evaluating E and printing final value in the successful execution. `VAR ‘‘x’’` means the value named `x`.

Expression defining name and limiting the effective range, scope is `LET(‘‘x’’,` E_1`,` E_2`)`. In this case, we name the evaluated value of E_1 as `x`, and the scope is limited to E_2. Any name not in the current environment is meaningless. For example,

The result of

```
LET("x", NUM 1,
    PLUS (LET("x", NUM 2, PLUS(VAR "x", VAR "x")),
          VAR "x")
```

```
                    )
```

is 5.

The result of

```
              LET("x", NUM 1,
                  PLUS (LET("y", NUM 2, PLUS(VAR "x", VAR "y")),
                        VAR "x")
                    )
```

is 4.

```
              LET("x", NUM 1,
                  PLUS (LET("y", NUM 2, PLUS(VAR "y", VAR "x")),
                        VAR "y")
                    )
```

is meaningless. `y` for outer `PLUS` expression is not defined in its environment.

`MAX` is integer expression for finding out the maximum integer in the list That is, the result of `MAX [NUM 1, NUM 3, NUM 2]` is 3. The result of `MAX` taking empty list is 0. □

- You don't have to print really using print function.
- If you meet free variable then raise `FreeVariable` exception.(you sould declare this `exception FreeVariable`)
- PLUS, MULT, DIVIDE follow integer arithmetic.