Homework 3
SNU 4190.310, 2014 Fall
Kwangkeun Yi
Due: 10/28(Tue), 24:00

Purpose of this homework:

- Understand the exact meaning of commonsense imperative language which we learned in class and implement the interpreter.
- Open eyes to the points which leave to be desired programming with that language.

Exercise 1 (40pts) "K- Interpreter"

Let's think about an imperative language K-[1] that we defined in class.

The homework is to implement an interperter that runs K- program as defined.

Make module `K` that have below `KMINUS` shape. (you don't have to include below signature module code.)

```
module type KMINUS =
sig
  exception Error of string
  type id = string
  type exp = NUM of int | TRUE | FALSE | UNIT
           | VAR of id
           | ADD of exp * exp
```

[1] Exact syntax and semantics are in TA page.

```
               | SUB of exp * exp
               | MUL of exp * exp
               | DIV of exp * exp
               | EQUAL of exp * exp
               | LESS of exp * exp
               | NOT of exp
               | SEQ of exp * exp             (* sequence *)
               | IF of exp * exp * exp        (* if-then-else *)
               | WHILE of exp * exp           (* while loop *)
               | LETV of id * exp * exp           (* variable binding *)
               | LETF of id * id list * exp * exp      (* procedure binding *)
               | CALLV of id * exp list       (* call by value *)
               | CALLR of id * id list        (* call by reference *)
               | RECORD of (id * exp) list    (* record construction *)
               | FIELD of exp * id            (* access record field *)
               | ASSIGN of id * exp           (* assign to variable *)
               | ASSIGNF of exp * id * exp    (* assign to record field *)
               | READ of id
               | WRITE of exp
      type program = exp
      type memory
      type env
      type value
      val emptyMemory: memory
      val emptyEnv: env
      val run: memory * env * program -> value
    end
```

K.run (K.emptyMemory, K.emptyEnv, S)

will compute the final value of a program S. If the program is not right type, raise Error exception and halt the program. “Error” means (if and only if) program semantic is not able to be defined with given semantics.

Type of input and output values is only integer in this program.

- Use read_int for read semantics and print_int for write. Do not put `print_endline` when doing `write`

- `write` : Print when the exp's value is integer. raise exception other cases.
- Do not print the result of K.run.
- Please do “`raise (Error ‘‘Invalid_argument’’)`“ when the number of arguments in `callv, callr` is different from the number of parameters in those function.
- You may raise `Error` with any string without upper case.

□

Exercise 2 (10pts) “K- Programming: The number of cases making changes”

Make K- program below, test with `K.run` you implemented.

Korea has currencies 1 won, 10 won, 100 won, 500 won, 1000 won, 5000 won, 10000 won, 50000 won. `numch`(100) is 12: From 100 1 won to 1 10 won and 90 1 won, · · ·, 1 100 won.

Hint: It may be like following code.

```
numch(n) = if n<10 then numch1(n)
           else if n<100 then numch10(n)
           ...

numch1(n) = 1
numch10(n) = if n<10 then numch1(n)
             else if ...
             else numch1(n) + numch10(n-10)
...
```

- Write your code relative to the parser (parser.mly). (Rule of thumb: use enough parenthesis)
- Please write `read (input)` and `write (numch(input))` at the end of the function like following.

```
let input := 0 in
let proc numch (x) = ... in (
  read input;
  write (numch(input))
)
```

- Assume `numch` takes just positive number as an argument.
- Actually another korean currency is 50 won.. but ignore it. We will follow the homework specification.

□

Exercise 3 (10pts) "K- Programming: compound data"

Make K- program below, test with `K.run` you implemented.

Make below functions that make and use binary tree:

`leaf`:	int → tree	(* a leaf tree *)
`makeLtree`:	int × tree → tree	(* a tree with only a left subtree *)
`makeRtree`:	int × tree → tree	(* a tree with only a right subtree *)
`makeTree`:	int × tree × tree → tree	(* a tree with both subtrees *)
`isEmpty`:	tree → bool	(* see if empty tree *)
`rTree`:	tree → tree	(* right subtree *)
`lTree`:	tree → tree	(* left subtree *)
`nodeVal`:	tree → int	(* node value *)
`dft`:	tree → unit	(* print node values in depth-first order *)
`bft`:	tree → unit	(* print node values in breath-first order *)

Make binary tree using upper functions and check if `dft` and `bft` print values in right order.

- Write your code relative to the parser (parser.mly). (Rule of thumb: use enough parenthesis)
- After testing, in submission, please terminate your code with just `in` (It will give you parsing error.)
- Make `dft` in *preorder* (root node) - (left subtree) - (right subtree).
- The order of traversing tree with `bft` is (root node) - (left node) - (right node) - . . . .

- Your code should satisfies following relation.
  - isEmpty(rTree(makeLtree (1, leaf (2)))) = true
  - isEmpty(Ltree(makeRtree (1, leaf (2)))) = true
- TA will not test following cases. First, takes empty tree or leaf as an argument of rTree/lTree. Second, takes empty tree as an argument of nodeVal.
- Print nothing when traversing empty tree.

□