Project
SNU 4910.210, Spring 2017
Chung-Kil Hur
due: 6/22(Thu) 23:59

Problem 1 (30 Points) In Scala, implement an interpreter `myeval` for the programming language E given below.

$$\texttt{myeval} : E \to V$$

- A parser is provided to translate a string to E.
- Design the value type V, and implement the `ConvertToScala` type class for V.
- For ill-typed inputs, you can return arbitrary values, or raise exceptions.

C	::=	n	Integer
	|	`true`	true
	|	`false`	false
	|	`nil`	list nil
E	::=	C	constant
	|	x	name
	|	`(if` E E E`)`	conditional
	|	`(cons` E E`)`	pair construction
	|	`(hd` E`)`	head of a pair
	|	`(tl` E`)`	tail of a pair
	|	`(fun (`x^*`)` E`)`	function
	|	`(`E E^*`)`	function call
	|	`(let (`B^*`)` E`)`	name binding to def/val
	|	`(+` E E`)`	integer addition
	|	`(-` E E`)`	integer subtraction
	|	`(*` E E`)`	integer multiplication
	|	`(=` E E`)`	integer equality
	|	`(<` E E`)`	integer less-than
	|	`(>` E E`)`	integer greater-than
B	::=	`(def` x E`)`	def
	|	`(val` x E`)`	val

- A^*: A appears several times (including 0 times). For example, the `(fun (`x^*`) E)` rule constructs both `(fun (a b) (+ a b))` and `(fun () 3)`.
- `let` creates a new scope, like a 'block' in Scala. `def` and `val` work in the same way as Scala.
  - `def` assigns a name to an expression.
  - `val` assigns a name to a value obtained by evaluating the given expression. The evaluation of `val`s in one `let` should be performed in sequential order: a latter `val` can refer names assigned by former `val`s.
  - To make a closure from an expression, just write the assigned name. (Unlike Scala, do not put '_' after the name.)
- For additional information, post questions on the GitHub course webpage.

Example programs:

- `myeval( (hd (cons 1 2)) )`
  Result: `1`
- `myeval( (let ((val p (cons 1 (cons true nil)))) (cons 0 p)) )`
  Result: `(0, (1, (2, nil)))`
- `myeval( (if true 10 20) )`
  Result: `10`
- `myeval( ((fun (x y) (+ x y)) 1 2) )`
  Result: `3`
- `myeval( (let ((val f (fun () (+ 1 2)))) (f)) )`
  Result: `3`
- `myeval( (let ((val a 10) (val b (+ a 1))) (* b 3)) )`
  Result: `33`
- `myeval( (let ((def f (fun (x) (if (= x 0) 0 (+ x (f (- x 1)))))))) (f 5)) )`
  Result: `15`
- `myeval( (let ((def f (fun (n) (g n 1))) (def g (fun (a b) (> a b)))) (f 3)) )`
  Result: `true`
- `myeval( ((fun (f) (fun (x) (f x))) (fun (x) (+ x 1))) )`
  Result: `function (that increments the input by 1)`

You can find the actual code of every example program in this document from `Test.scala` in the skeleton code.

Problem 2 (10 Points) Optimize `myeval` to handle tail recursive input programs, such as the example code shown below.
(Hint: Use Scala's tail recursion.)

- `myeval( (let ((def f (fun (x n)`
  `(if (= x 0) n (f (- x 1) (+ n x)))) )) (f 9999 0)) )`
  Result: 49995000

Problem 3 (10 Points)

Implement another interpreter `myeval_memo` that uses the 'memoization' technique to optimize the evaluation time.

A memoized function records input-output pairs every time it is called. When the function is called with a recorded input, it just returns the recorded correspoinding output, instead of computing it again.

`myeval_memo` do not have to be tail-recursive.

(Hint: You can use `scala.collection.immutable.HashMap` in this problem.)

- `myeval( (let ((def f (fun (n) (if (= n 0) 1`
  `(if (= n 1) 0 (if (> (f (- n 1)) (f (- n 2))) 0 1))))))`
  `(f 100)) )`
  Result: `1`