Project
SNU 4190.210, Programming Principles Fall 2023
Chung-Kil Hur

due: 12/13(Wed) 23:59

Problem 1 (50 Points) In Scala, implement an interpreter `interp` for the programming language E given below.

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

A	::=	x	call by value
	|	`(by-name` x`)`	call by name
B	::=	`(def` fn `(`A^*`)` E`)`	def
	|	`(val` x E`)`	val
	|	`(lazy-val` x E`)`	lazy val
E	::=	x	name
	|	n	integer
	|	f	float
	|	s	string
	|	`nil`	pair nil
	|	`(cons` E E`)`	pair constructor
	|	`(fst` E`)`	the first component of a pair
	|	`(snd` E`)`	the second component of a pair
	|	`(int?` E`)`	is int
	|	`(float?` E`)`	is float
	|	`(string?` E`)`	is string
	|	`(nil?` E`)`	is nil
	|	`(pair?` E`)`	is pair
	|	`(substr` E E E`)`	substring
	|	`(len` E`)`	length of a string or list
	|	`(if` E E E`)`	conditional
	|	`(let` B^* E`)`	name binding of def/val
	|	`(app` E E^*`)`	function call
	|	`(+` E E`)`	addition / string concat
	|	`(-` E E`)`	subtraction
	|	`(*` E E`)`	multiplication
	|	`(/` E E`)`	division
	|	`(%` E E`)`	remainder
	|	`(=` E E`)`	equality
	|	`(<` E E`)`	less than
	|	`(>` E E`)`	greater than

- For ill-typed inputs, you can return arbitrary values, or raise exceptions.
- X^* denotes that X can appear 0 or more times.
- `let` clauses create a new scope like a 'block' in Scala. Name bindings `def` and `val` work the similar way as in Scala.
  - `(def f (`A^*`)` E`)` assigns name `f` to expression `E` with arguments A^*. Examples include `(def f (a (by-name b)) (+ a b))` and `(def g () 3)`.
  - `(val` x E`)` assigns name x to the value obtained by evaluating E.
  - We do not allow the same name to be defined twice in the frame.
  - You do not have to consider forward reference in `val`. For example, `(val` x `(cons 1` x`))`.
- `Environment` is collection of `Frames`. `Frame` is created when a new scope is created.

- Identifier `x` should be an alphanumeric word which does not start with a number.
- `nil` and `(cons` v_1 v_2`)` are pair type. `pair?` should return 1 for these values. Otherwise, it should return 0.
- `(int? E)` first evaluates `E` into value v. If v is `integer`, it returns 1. Otherwise, it returns 0. Also `nil?`, `float?`, `string?`, and `pair?` behave the same way.
- `(substr` E_1 E_2 E_3`)` first evaluates E_1 into string s (If E_1 is not a string, raise any exception). E_2 and E_3 are the start and the end position of the substring of s. (You can simply use `String.substring` method of Scala)
- `(len` E`)` first evaluates E into value v. If v is a string or a pair (Cons or Nil), return the length of v. Otherwise, raise any exception.
- `len` of pair works simliar to Scala's `List[Any].length`. Since the last element of cons list from our language can be non-Nil element, `len` should caculate the number of the elements in the cons list, but must ignore the last Nil.
- e.g.) `(len (cons (cons (cons 5 4) 2) (cons 3 4)))` = 3, `(len (cons 2 (cons 3 nil)))` = 2, `(len nil)` = 0.
- For the binary operators (`+, -, *, /, %, =, <, >`), the types of two operands must be number (except `=`). If one of the operand is float type, the result also have to be a float value. Otherwise, the result will be an integer value.
- As an exception, `+` is a string concatenation when the two operands are string values. Also you can use `=` to compare two strings.
- Comparison expressions (`=, <, >`) returns 1 if the comparison is right. Otherwise, it returns 0.
- `(if` E_1 E_2 E_3`)` first evaluates E_1 into value v. If v is 0 or 0.0, it returns the result of E_3. Otherwise, it returns the result of E_2.
- `(lazy-val` x E`)` assigns name x to the value obtained by evaluating E lazily.
- Hint: Use `LazyOps`.
- For additional information, post questions on the GitHub course webpage.
- examples in `src/test/scala/TestSuite.scala`.

Problem 2 (15 Points) Optimize `interp` to handle tail recursive input programs, such as the example code shown below.

```
(let (def f (x sum) (if (> x 0) (app f (- x 1) (+ x sum)) sum))
(app f 10 0))
```

Hint: You don't need to reuse `Frame`. Just make `app` handler tail recursive, then you will get what you want.

Problem 3 (15 Points) Add IO action to `interp` by implementing `defIO`, `runIO`, `readline`, and `print` following:

B	::=	$\cdots$	
	$\mid$	(`defIO` fn (x^*) IO^* E)	define an IO action
IO	::=	(`runIO` x E)	run an IO action and bind to x
	$\mid$	(`readline` x)	read a line text and bind to x
	$\mid$	(`print` E)	print

- (`defIO` $fn(A^*)$ (IO^*) E) defines a function that returns **IO action**. fn takes a list of parameters A^* like a normal function. Then, it executes IO^* sequentially, and finally E becomes the result of the IO action.
- The result of (`app` fn E^*) is **NOT** an evaluation of E. Instead, it returns an runnable *thunk* (pending action) which has a type of `IO[Value]` where the last E is `Value` type. From this stage, it does not actually runs (IO^*) inside fn. Those will be run when `runIO` calls the thunk which will be explained below.
- (`runIO` x E) actually **runs** an IO action when E returns an IO action. (i.e. `IO[Value]`). The result of the IO action is bound to x (i.e. The type of x is `Value`.). You can also bind any value to x other than IO action. x can be used in the following IO actions and the result E, but not in the previous IO actions. (i.e. does not allow forward reference).
- (`readline` x) reads a line from the input (e.g. standard input) and binds it to x. The read value does not contain newline characters (`\n, \r, \r\n, ...`). You can assume that there will be only ASCII characters in the input.
- (`print` E) prints the result of E to the output (e.g. standard output). It does not automatically append newline characters.
- If the top-level (i.e. the most outer) expression is an IO action, that IO action will be automatically executed if you run a main function.

You must use `Reader` and `Writer` trait to actually execute `readline` and `print`. Also, you have to use `Show` trait to convert a value to string for `print`.

For example,

```
(let ((defIO f1 (x) (runIO a readline) (+ a x)) (defIO f2 (b) (runIO
ax (f1 b)) (runIO b readline) (+ ax b))) (app f2 10))
```

reads two numbers `a, b`, and prints the value of `a + b + 10`.

Problem 4 (20 Points) Implement a calaulator REPL (read-eval-print-loop) with the language E.

$$\begin{array}{rcll} Exp & ::= & n & \text{positive integer} \\ & | & Exp + Exp & \text{add} \\ & | & Exp - Exp & \text{sub} \\ & | & Exp * Exp & \text{mul} \\ & | & Exp / Exp & \text{div} \end{array}$$

- For each text taken from `readline`, if the text is a valid expression, evaluate the expression and print the result.
- If the text is not a valid expression, print `parse error`.
- If the text is `exit`, terminate the REPL.
- There will be no whitespaces in the input.