Project
SNU 4910.210, Programming Principles Fall 2022
Chung-Kil Hur
due: 12/21(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	::=	n	integer
	|	f	float
	|	s	string
	|	x	name
	|	`nil`	pair nil
	|	`(cons` E E`)`	pair constructor
	|	`(fst` E`)`	the first component of a pair
	|	`(snd` E`)`	the second component of a pair
	|	`(nil?` E`)`	is nil
	|	`(int?` E`)`	is int
	|	`(float?` E`)`	is float
	|	`(string?` E`)`	is string
	|	`(pair?` E`)`	is pair
	|	`(substr` E E E`)`	substring
	|	`(len` E`)`	length of string or list
	|	`(if` E E E`)`	conditional
	|	`(let (`B^*`)` E`)`	name binding of def/val
	|	`(app` E E^*`)`	function call
	|	`(+` E E`)`	addition
	|	`(-` 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))`.
  - Also, you do not have to consider forward reference of two `def`s. For example, `(let (def f1 () (app f2)) (def f2 () (app f1)))`.

- `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.
- `(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 numeric operators (`+, -, *, /, %`), the types of two operands must be number. 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. You can compare two numeric values, or two strings. The comparison between any other types should return 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/InterpreterTest.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 algebraic effect handler to `interp` by implementing `effect`, `handle`, and `case` following:

C	::=	(`case` E x E)	effect handler
E	::=	$\cdots$	
	|	(`try` E C^*)	evaluate with handlers
	|	(`effect` E E)	call effect

Algebraic effect is a **resumable exception handler**.
See `https://overreacted.io/ko/algebraic-effects-for-the-rest-of-us/`

- (`try` E C^*) first evaluates E. While evaluating E, there can be an `effect` expression to call one of the handlers.
  - If there is a handler which can handle the effect, evaluate that handler and **resume** at the call site of the effect. The result of the effect should be the result of the effect handler.
  - If there is no handler to handle the effect in this `try` block, propagate the effect to the outer `try` block just like the regular `try-catch` blocks.
  - If there is no proper handler in the whole context, raise any exception.
  - If there is more than two handlers which takes the same effect, call the first (inner-most, upper-most) one.
- (`effect` E_c E_x) calls an effect of E_c with the value E_x. If the proper handler is found, call that handler and resume at this point. The result of `effect` should be the result of that handler.
- (`case` E_c x E_h) is a handler for the case E_c. If the E_c effect is called, the value E_x from the above `effect` is bound to x. The result of the effect should be the result of E_h with the given x.

Problem 4 (20 Points) Implement an interpreter of Brainfuck language.
We will give you a skeleton of the interpreter and basic functions to handle pointers and print ASCII code. Assume that the memory consists of 32 circular cells.

- Input command `,` will take a single character from the pre-defined list.
- Output command `.` will append a character to the output list.
- We will test your Brainfuck interpreter with our language E interpreter.