Prof. Jingke Li (FAB120-06, lij@pdx.edu), Class: TR 10:00-11:15 @ EB 103; Office Hour: TR 11:20-12:20.

Assignment 4: Functional Language Implementation

(Due Thursday 12/1/22)

This assignment deals with a couple of issues related to functional language's implementation, including function value representation and by-name parameter passing.

1. FuncLang

FuncLang is a simple functional language, with the following grammar:

```
Expr -> <num>
     |  <id>
     |  (Op Expr Expr)
     |  (if Expr Expr Expr)
     |  (let <id> Expr Expr)      // local scope
     |  (let* <id> Expr Expr)     // recursive version of let
     |  (fun <id> Expr)           // lambda func
     |  (@ Expr Expr)             // func application
Op   -> + | - | * | / | %
     |  < | > | ==
<num>: integer  // "wholeNumber"
<id>:  string   // "ident"
```

FuncLang supports two types of values, *integers* and *closures*. The latter is for representing function values. Here is a Scala representation of FuncLang's values:

```
sealed abstract class Value
case class NumV(n:Int) extends Value
case class ClosureV(x:String,b:Expr,env:Env) extends Value
```

As shown, a closure includes a function definition and an environment. Unlike the pair values of ScopeLang, closure values are not boxed values, rather, they are treated just like integers: they exist on their own, and can be stored in variables. (This choice is to make the implementation easier.)

Note: This closure definition is different from the version discussed in class, where a closure is defined to comprise a code-pointer and a set of bindings for (only) free-variables. However, both versions serve the same purpose, *i.e.* to preserve relevant information of a function for later invocations.

The new constructs of FuncLang are defined below:

- There is support for lambda-expressions (anonymous functions) in the form of `(fun x b)` expression, where `x` is the function's formal parameter, and `b` is the body. Evaluating `(fun x b)` creates a closure, of the form `(x b env)`, as defined above. `env` is a copy of the current environment, within which the function is defined. (This environment thus includes bindings for all free variables of the function.)

  To introduce a named function, use `let` to bind a name to a `fun` expression. For example, we could define and use an incremental function as follows:

  ```
  (let incr (fun x (+ x 1)) (@ incr 5))
  ```

- There is a function application expression `(@ f e)`. Evaluating it evaluates `f` to a function value (*i.e.* a closure), evaluates `e` to a value and binds it to the formal parameter of the function, evaluates the function body, in the closure's environment augmented with the new binding, and yields the resulting value. It is a checked run-time error if `f` doesn't evaluates to a function value.

- There is support for recursive `let*` expressions to build recursive functions. Recall that, evaluating the regular `(let x b e)` evaluates `b`, binds the resulting value to the newly created local variable `x`, evaluates expression `e` in the resulting environment, and yields the resulting value.

  The semantics for `(let* f b e)` is slightly different: it puts `f` in scope within `b` as well as within `e`. Evaluating it first adds a *forward* binding for the new local variable `f` to the environment (a forward binding is an incomplete binding whose missing value will be patched later), then evaluates `b` in the resulting environment and patches `f`'s binding with `b`'s value, finally evaluates `e` and yields the resulting value.

  With `let*`, we could define and use a recursive factorial function as follows:

  ```
  (let* fac (fun x (if (< x 1)
                       1
                       (* x (@ fac (- x 1)))))
        (@ fac 6))
  ```

  Although the grammar allows `b` to be any expression, the FuncLang interpreter only works when `b` is a `fun` expression. It is a checked run-time error if `b` is not a `fun` expression.

Your Tasks

The file `FLInterp.scala` contains an incomplete FuncLang interpreter. Your task is to complete the interpreter implementation, with the following three steps.

1. **Naive version (stack only).** In this first version, the heap storage is *not* used at all. All storage-bound values are stored on the stack. More specifically, every time a new variable is introduced by a `let` or a `let*` construct, its value (an integer or a closure) is stored on the stack; every time a function is called, its (sole) parameter's value is also stored on the stack.

   After completing the program, test it with `TestFLI.scala`. It should pass all the tests that do not have a `true` flag in its `FLInterp()` call.

2. **Correct version (using heap).** As shown by the failure on the "first-class functions" tests, the naive version does not support first-class functions correctly. This is because closures may refer to stack-allocated values that are no longer exist. (*Aside:* Analyze the runtime memory activities of any of the `example[1-4]` programs in the `TestFLI.scala` file to see why it fails.)

   Modify the interpreter to store all variables and parameters into the heap. Keep the existing stack-storage code as is and place the new heap-storage code under an `if` statement, at all places where changes are needed:

   ```
   if (useHeap)
     <new heap-storage code>
   else
     <existing stack-storage code>
   ```

   This way both the naive version and the new version of the interpreter will be available, and can be selected by a Boolean flag `useHeap`. The `apply` driver routine has a parameter for setting this flag:

   ```
   def apply(s:String, useHeap:Boolean=false, ...): Int = {...}
   ```

   Now test the interpreter with `TestFLI.scala` again. It should pass those `useHeap` tests now (but will fail on the last `lazy-eval` test).

3. **Call-by-name parameter passing.** FuncLang's function application `(@ f e)` follows the call-by-value semantics: It evaluates the argument `e` to a value and binds it function `f`'s formal parameter, before executing `f`'s body.

   Implement a second parameter passing semantics, call-by-name. Under this new semantics, all occurrences of the formal parameter in `f`'s body are replaced by the argument `e`. You may assume that there is no name conflicts in the process, in other words, you don't need to implement α-reduction.

Note: Multiple definitions for the same variable in nested scopes, e.g.

```
(fun x (let x 3 (+ x 1))),
```

is not an example of name conflict. Your program should be able to handle this function's application without needing α-reduction, e.g.

```
(@ (fun x (let x 3 (+ x 1))) 4)
```

(*Hint:* The call-by-name substitution process should skip bound variables.)

Use the same flag arrangement for making this change:

```
if (callByName)
  <new call-by-name code>
else
  <existing call-by-value code>
```

This time, there is only one place this switching needs to happen, *i.e.* in the function application section. However, the substitution process needs to recursively visit all expression forms. As a hint, you may consider implementing the following recursive function:

```
// substitute argument y for parameter x in expression e
// (skip bound occurrences of x)
def substitute(e:Expr, x:String, y:Expr): Expr = {
  e match {
    case Num(n)   => e
    case Var(x)   => ... // this is where actual substitution happens
    case Add(l,r) => Add(substitute(l,x,y), substitute(r,x,y))
    ....
  }
}
```

Note: In processing a binding construct, i.e. `let`, `let*`, or `fun`, the substitution routine should check to see if `x` matches the construct's binding variable, if so, the construct should be skipped. (Since in this case the occurrences of `x` within the construct are bound to the construct's variable.)

For debugging, it might be helpful to print out the result of substitution for each function call:

```
val e2 = substitute(...)
if (debug > 0)
  println("CBN: " + Apply(f,e) + " => " + e2)
```

So that you can be sure that the β-reduction is performed correctly.

Summary and Submission

Write a short summary (half page or one page) covering your experience with this assignment. Save this write-up in a text file, `assign4-notes.txt`. Zip this write-up, along with the program, `FLInterp.scala`, into a single file, and submit it through the "Assignment 4" submission folder on the Canvas class website.