Prof. Jingke Li (FAB120-06, lij@pdx.edu), Class: TR 12:00-13:15 @ KMC 185; Office Hour: TR 11:00-11:55.

Assignment 7&8: Functional Programming

(Due Thursday 11/22/18)

This double assignment deals with several issues relating to functional programming, including function values, parameter passing, and various forms of recursions. (*Fair Warning:* This assignment contains double amount of work. So start early; and if you have questions or issues, seek help early as well!)

1. EL3

EL3 is a simple functional language, with the following grammar:

```
Expr -> Int
      |  String
      |  (Op Expr Expr)
      |  (if Expr Expr Expr)
      |  (seq Expr Expr)
      |  (skip)
      |  (let String Expr Expr)
      |  (letRec String Expr Expr)
      |  (fun String Expr)
      |  (@ Expr Expr)
Op    -> + | - | * | / | <=
```

EL3 supports two types of values, *integers* and *closures*. The latter is for representing function values. Here is a Scala representation of EL3's values:

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

As shown, a closure includes a function definition (see below) and an environment (holding variable-storage mappings). This environment is the one the function is defined in, which includes bindings for all free variables of the function. (*Note:* This closure definition is slightly different from the version discussed in class, where a closure is defined to include 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 for later function invocations.)

EL3 has the following new features:

- 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, which includes both `x` and `b`, as well as a copy of the current environment.

  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 `letRec` expressions to build recursive functions. Writing `(letRec f b e)` puts `f` in scope within `b` as well as within `e`. For example, we could define and use a recursive factorial function as follows:

```
(letRec fac (fun x (if (<= x 0)
                       1
                       (* x (@ fac (- x 1)))))
        (@ fac 6))
```

  Although the grammar allows `b` to be any expression, the EL3 interpreter only works when `b` is a `fun` expression. It is a checked run-time error if `b` is not a `fun` expression.

2. Storage Model and Environment Support

The EL3 interpreter implements two storage classes, stack and heap. In the default mode, all variables and parameters are stored on the stack. More specifically, every time a new variable is introduced by a `let` or a `letRec` construct, its value is stored on the stack; every time a function is called, its (sole) parameter's value is stored on the stack. (*Note:* In the default mode, the heap storage is *not* used at all.)

Unlike the treatment of pair values in the last assignment, closure values are not assigned storage in the interpreter. (This is due to a technical reason: We don't have a storage model for program code, so we can't have "code pointers.") Closures are treated just like integers, they exist on their own, and can be stored in variables.

3. EL3 Interpreter

- **Default Version**

  The file `Interp3.scala` contains an incomplete EL3 interpreter. Read and understand as much as possible the given code in the file. Complete the implementation for the default version, *i.e.* store all variables and parameters on the stack. Test it with `TestInterp3.scala`. Ignore the failings of the "use heap storage" and "call-by-name" tests.

- **Analysis**

  Look at the following two simple EL3 expressions:

```
(let f (fun x (fun y x)) (@ (@ f 2) 1))
(let f (let y 4 (fun x y)) (@ f 1))
```

  1. Based on EL3's semantics, what should be the value of each expression?
  2. Both expressions are coded in `TestInterp3.scala` (as `example2` and `example4`). What are their values according to the interpreter?
  3. Using the storage model to explain the behavior of the interpreter. Be specific, *e.g.* explain with info such as which value is stored where and when.

  Save your answers in a text file, `analysis.txt`.

- **First-Class Function Support**

  As you may have seen with the above analysis, the default mode does not support truly first-class functions, because closures can refer to stack-allocated values.

  Now modify the interpreter to store all variables and parameters into the heap. Make this modification by placing the new storage code under an if statement, at all places where changes are needed:

```
if (useHeap)
  <new code>
else
  <existing code>
```

This approach allows both the default version and the new version of the interpreter be available, and be selectable by a Boolean flag `useHeap`. The `process` driver routine has a parameter for setting this flag:

```
def process(s:String,heap:Boolean=false,cbn:Boolean=false,debug:Int=0): Int = {...}
```

Now test the interpreter with `TestInterp3.scala` again. It should pass the "use heap storage" test.

- **Call-By-Name**

  EL3'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.

  Now you are asked to 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`. (This is called *beta-reduction*.) You may assume that there is no name conflicts in the process, in other words, you don't need to implement the *alpha-reduction* in this first version.

  Use the same flag arrangement for making the change:

  ```
  if (callByName)
    <new code>
  else
    <existing code>
  ```

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

  ```
  // Replacing x by y in e
  def replaceE(e:Expr,x:String,y:Expr): Expr = {
    e match {
      case Num(n) => e
      case Var(x) => ...
      case Add(l,r) => Add(replaceE(l,x,y),replaceE(r,x,y))
      ....
    }
  }
  ```

  Test your new version with `TestInterp3.scala`. This time a couple of "call-by-name" tests should pass, although the one requires "alpha-reduction" will still fail.

- **Alpha-Reduction**

  When performing a beta-reduction across `let`, `letRec`, and `Fun` constructs, you need to be aware of the possibility of "name conflicts," *i.e.* the bound variable in these constructs may share the same name with a variable in the beta-reduction. As shown in class, in this case, one can perform an alpha-reduction first, which changes the bound variable's name.

  For this last part, you are to implement the alpha-reduction. A simple trick that substantially simplifies the task is to perform the alpha-reduction *preemptively*, *i.e.* without checking whether conflicts would happen or not. This approach would work if we change a bound variable's name to something we know for sure does not conflict with the beta-reduction. In our case, we assume EL3's identifiers do not contain symbols. So we can change a bound variable `x` to `x_`, and be assured that there will be no conflicts.

For this part, you should just go ahead and modify the interpreter. After the modification, the new version of interpreter should pass all tests in `TestInterp3.scala`. (*Hint:* You should be able to use the same recursive function, `replaceE()`, to implement the alpha-reduction.)

4. Programming in EL3

While simple, EL3 is a powerful functional language. In this part, you are going to convert several Scala programs into EL3. Write your solution EL3 programs in the program file `TestNew.scala`. If your heap version of the interpreter is working, test them with your interpreter.

Here are the programs you need to convert.

- **Multi-Argument Functions:**

```
def f1(x:Int,y:Int):Int = x - y
f1(4,2)
```

```
def f2(x:Int,y:Int,z:Int):Int = x + y - z
f2(2,3,4)
```

Since EL3 has only single-argument functions, you need to use curried functions to implement the above programs.

- **Fibonacci Functions:**

```
// regular version
def fib(n:Int):Int = {
  if (n <= 2) 1
  else fib(n-1) + fib(n-2)
}
fib(10)
```

```
// tail-recursive
def fibt(n:Int):Int = {
  def helper(n:Int,a:Int,b:Int):Int = {
    if (n <= 2) a
    else helper(n-1, a+b, a)
  }
  helper(n,1,1)
}
fibt(10)
```

```
// continuation-passing style
def fibc(n:Int):Int = {
  def helper(n:Int,c:Int=>Int):Int = {
    if (n <= 2) c(1)
    else helper(n-1, x => helper(n-2, y => c(x+y)))
  }
  helper(n,x=>x)
}
fibt(10)
```

The last program is *optional*. You'll earn exra credits if you correctly implement it.

5. Submission

Zip your `Interp3.scala`, `TestNew.scala`, and `analysis.txt` into a single ZIP file, and submit it through the "Assignment 7" submission folder on the D2L class website (under the "Activities/Assignments" tab). Keep your original files untouched in case there is a need to show their timestamp.