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

# Assignment 3: Scopes and Storage Management

## (Due Thursday 11/17/22)

This assignment deals with scopes and storage management in an interpreter.

## 1. ScopeLang

ScopeLang extends ImpLang (frome Assignment 2) to support scopes and boxed data. (For simplicity, ImpLang's `for` construct is omitted.) There are several new constructs: `let`, `pair`, `pair?`, `#1`, `#2`, `set#1`, and `set#2`, and a new equality comparison operator, `deq`. ScopeLang's grammar is given below:

```
Expr -> <num>                    // int const
     |  <id>                     // var
     |  (Op Expr Expr)           // binop
     |  (:= <id> Expr)           // ref assign
     |  (write Expr)             // output
     |  (seq Expr Expr)          // sequence
     |  (if Expr Expr Expr)      // if
     |  (while Expr Expr)        // while loop
     |  (let <id> Expr Expr)     // local scope
     |  (pair Expr Expr)         // pair constructor
     |  (pair? Expr)             // pair query
     |  (#1 Expr)                // 1st elm of pair
     |  (#2 Expr)                // 2nd elm of pair
     |  (set#1 Expr Expr)        // set 1st elm of pair
     |  (set#2 Expr Expr)        // set 2nd elm of pair
Op   -> + | - | * | / | %        // arith op
     |  < | > | ==               // relational op (== is ref comparison)
     |  deq                     // deep comparison
<num>: integer  // "wholeNumber"
<id> : string   // "ident"
```

ScopeLang programs manipulate values, which can be either integers or pairs. A pair in turn contains two values. Here is a Scala representation of ScopeLang's values:

```
sealed abstract class Value
case class NumV(n:Int) extends Value
case class PairV(a:Addr) extends Value
```

As shown, an integer value is represented by the number itself, together with a `NumV` tag, while a pair value is represented by a reference (of type `Addr`, which is defined in the interpreter) pointing to its storage location, together with a `PairV` tag. (Thus, pairs are *boxed* data.)

As an implication, when we copy a pair value, only its reference gets copied, and when we compare two pairs, only their references are compared. (*Note:* a separate comparison operator, `deq`, is defined in the language to perform *deep* (aka *structural*) comparison.)

Informal semantics for the new features are given below, followed by the necessary semantic changes to the existing constructs. The term "checked runtime error" refers to an error that will be caught by the language's runtime system, such as an interpreter.

- The `let` expression introduces a local scope with a variable binding and a body. Evaluating `(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. For example, the expression

  ```
  (let x 1 (+ x 2))
  ```

  introduces a variable `x` and binds 1 to it; its value is the body expression `(+ x 2)`'s value, which evaluates to 3.

  ScopeLang uses static scope rules. The scope of `x` is just the expression `e`. A variable introduced by a `let` binding hides any outer scope's variable with the same name.

- The `pair` expression is a pair constructor. Evaluating `(pair e1 e2)` evaluates `e1` and `e2` (in that order) to values `v1` and `v2` and yields a new pair (*i.e.* a value of type `PairV`) whose left element is `v1` and right element is `v2`.

- Evaluating `(pair? e)` evaluates `e` and yields 1 if the result is a pair and 0 otherwise.

- Evaluating `(#1 e)` evaluates `e` to a pair value, and extracts and yields the left element value. It is a checked runtime error if `e` evaluates to a non-pair value.

- Evaluating `(#2 e)` evaluates `e` to a pair value, and extracts and yields the right element value. It is a checked runtime error if `e` evaluates to a non-pair value.

- Evaluating `(set#1 p e)` evaluates `p` to a pair value `pv`, evaluates expression `e` to a value `v`, updates the left component of `pv` with `v`, and yield the (mutated) pair `pv` as the result of the expression. It is a checked runtime error is `pv` is not a pair.

- Evaluating `(set#2 p e)` evaluates `p` to a pair value `pv`, evaluates expression `e` to a value `v`, updates the right component of `pv` with `v`, and yield the (mutated) pair `pv` as the result of the expression. It is a checked runtime error if `pv` is not a pair.

- The equality comparison operator, `==`, works on both integers and pairs, but both operands must be of the same type. Evaluating `(== e1 e2)` evaluates `e1` and then `e2`, and compares their values based on representations (*e.g.* for pairs, compares their references). If the two values are equal, the expression yields 1; otherwise it yields 0. For example, `(== (pair 1 2) (pair 1 2))` yields 0, since the two pairs are unrelated objects. It is a checked runtime error if `e1`'s value and `e2`'s value are not of the same type.

- The deep equality comparison operator, `deq`, works on both integers and pairs, and it does not require both operands to be of the same type. Evaluating `(deq e1 e2)` evaluates `e1` and then `e2`, and compares their values *structurally*: If both values are integers, yields 1 if they are equal; otherwise yields 0. If both values are pairs, yields 1 if their references are equal; otherwise recursively deep compares `(#1 e1)` with `(#1 e2)`, and `(#2 e1)` with `(#2 e2)`, and yields 1 if both yield 1; otherwise yields 0. It yields 0 if `e1`'s value and `e2`'s value are not of the same type.

The following are additional semantic requirements:

- Every variable must be linked to a `let` binding. It is a checked runtime error to use an unbounded variable.

- The numeric operators (`+,-,*,/,%,<,>`) work only on integers; it is a checked runtime error to apply them to a pair.

- The value written by a `write` can be either an integer or a pair. In the case of a pair, e.g. `(pair x y)`, the print out format is `(x.y)`, i.e. two values within parentheses separated by a dot (and followed a `newline`). Also, in this case, `write` yields the pair's value.

- The value tested by a `if` or a `while` must be an integer; otherwise a checked runtime error results.

- The top-level expression (i.e. the program) must evaluates to an integer; otherwise a checked runtime error results.

## 2. Storage Model

In the ScopeLang interpreter implementation, all variables and all pairs are stored in storage. Two storage classes, stack, and heap, are used. (There is no global data in ScopeLang, so no need for static storage.)

- Variables introduced by `let`s are stored on the stack.
- Pairs are stored in the heap.

The storage classes are implemented as follows:

```
// Storage type declarations
type Index = Int
sealed abstract class Store {
  private val contents = collection.mutable.Map[Index,Value]()
  def get(i:Index) = contents.getOrElse(i, throw UndefinedContents("" + i))
  def set(i:Index,v:Value) = contents += (i->v)
}
class HeapStore extends Store {
  private var nextFreeIndex:Index = 0
  def allocate(n:Int): Addr = { ... } // allocates n units, returns a heap addr
  // there is no mechanism for deallocation
}
class StackStore extends Store {
  private var stackPointer:Index = 0;
  def push(): Addr = { ... }          // allocates 1 unit, returns a stack addr
  def pop() = stackPointer -= 1       // deallocates 1 unit
}
// Two actual storage
val heap = new HeapStore()
val stack = new StackStore()
```

Each storage class has its own address type:

```
sealed abstract class Addr() { ... }
case class HeapAddr(index:Int) extends Addr { ... }
case class StackAddr(index:Int) extends Addr { ... }
```

which makes it easy to lookup or set a variable's value:

```
def get(a:Addr) = a match {
  case HeapAddr(i)   => heap.get(i)
  case StackAddr(i)  => stack.get(i)
}
def set(a:Addr,v:Value) = a match {
  case HeapAddr(i)   => heap.set(i,v)
  case StackAddr(i)  => stack.set(i,v)
}
```

Note that for storing a pair value, two adjacent storage addresses are needed, since there are two values:

```
set(addr,leftv)
set(addr+1,rightv)
```

Similarly, when there is a need to fetch a pair's value (say, for printing), two fetches will be needed:

```
val lv = get(addr)
val rv = get(addr+1)
```

## 3. Environment Support

In the interpreter, program states, i.e. variable-storage bindings, are maintained in an environment:

```
type Env = Map[String,Addr]
var env: Env = Map[String,Addr]()
```

Initially, the environment `env` is empty. As the interpretation process enters and leaves scopes, the environment gets augmented and retracted with variable-storage bindings. Note that when a scope closes, it's local variable's value needs to be popped off from its stack storage, and its binding need to removed from the environment. At the end of program interpretation, the stack should be empty.

To augment `env` with a new binding `x->v`, you simply do

```
env + (x->v)
```

If you keep the previous version of the environment around, then there is no need to retract a binding; you may just revert back to the previous version.

## Your Tasks

The file `SLInterp.scala` contains an incomplete ScopeLang interpreter. Your task is to complete the interpreter implementation.

There are three major differences comparing the current interpreter with the `ImpLang` interpreter from Assignment 2.

1. There are two value types to deal with: `NumV` (for integers) and `PairV` (for pairs). This shows up everywhere, for example, instead of yielding a 0, now it's `NumV(0)`. Pairs are boxed data, they need to be stored in the heap, and accessed through references.
2. Due to the presence of scopes, the interpreter needs to rely on an environment to maintain valid bindings at any program point. Every interpretation routine now has an `env` parameter. Variables need to be stored on the stack. When a scope closes, the corresponding variable needs to be popped off from the stack.
3. The interpretation of the deep comparison construct, `(deq e1 e2)`, requires a new, recursive checking process. Note that `set#1` or `set#2` can be used to create cyclic pair structures, which may cause the recursive checking process go into an infinite loop. For this assignment, you should just assume that won't happen.

As usual, a test program, `TestSLI.scala`, is provided. While it is convenient to have many tests in a single file, it is not the best form for debugging. During the debugging period, it might be helpful to copy individual tests out to files, say, `test1`, ... `testk`, and run the interpreter on each file, with the debug flag on:

```
linux> scala SLInterp test1 2
```

## 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, `assign3-notes.txt`. Zip this write-up, along with the program, `SLInterp.scala`, into a single file, and submit it through the "Assignment 3" submission folder on the Canvas class website.