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

Assignment 6: Scopes and Storage Management

(Due Thursday 11/8/18

This week's assignment deals with scopes and storage management. You are going to implement an interpreter with variable environments.

1. EL2

EL2 extends EL1 to support scopes and boxed data. An EL2 program consists of a sequence of global definitions and a main body expression. For simplicity, only part of EL1's exressions are inherited. But, there are several new expression forms: `let`, `pair`, `isPair`, `fst`, `snd`, `setFst`, and `setSnd`, and two equality comparison operators. EL2's grammar is given below:

```
 Program -> (List[GDef] Expr)
 GDef -> (String Expr)
 Expr -> Int
      |  String
      |  (Op Expr Expr)      // arithmetic ops
      |  (= Expr Expr)       // shallow comparison
      |  (== Expr Expr)      // deep comparison
      |  (:= String Expr)
      |  (if Expr Expr Expr)
      |  (write Expr)
      |  (seq Expr Expr)
      |  (skip)
      |  (let String Expr Expr)
      |  (pair Expr Expr)
      |  (isPair Expr)
      |  (fst Expr)
      |  (snd Expr)
      |  (setFst Expr Expr)
      |  (setSnd Expr Expr)
Op    -> + | - | * | <=
```

EL2 programs manipulate values, which can be either integers or pairs. A pair in turn contains two values. Here is a Scala representation of EL2's values:

```
sealed abstract class Value
case class NumV(num: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`) pointing to its storage location, together with a `PairV` tag. (We say 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:* Special implementation can be made to perform *structural* (a.k.a. *deep*) comparison.)

Informal semantics for the new features are as follows.

- A program is evaluated by elaborating each global definition in order, and then evaluating the main body expression, whose value is the program result. The program result value must be an integer (not a pair).
- A global definition `(x e)` is elaborated by evaluating its initializing expression `e` to a value `v` and then binding `x` to `v`. Each global is in scope from just below its definition to the end of the program, *i.e.* in subsequent global definitions and the main body. It is a checked run-time error to access a global's value before it has been initialized.

  If the same name appears twice in the list of globals, the behavior of the program is undefined (and unspecified: no such programs are used in any of the specification tests).
- The `let` expression introduces a local scope with a variable binding and a body. Evaluating `(let x e b)` evaluates `e`, binds the resulting value to the newly created local variable `x`, evaluates expression `b` 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.

  EL2 uses static scope rules. The scope of `x` is just the expression `b`. A local variable introduced by a `let` binding hides any global or outer local 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 whose left element is `v1` and right element is `v2`.
- Evaluating `(isPair e)` evaluates `e` and yields 1 if the result is a pair and 0 otherwise.
- Evaluating `(fst e)` evaluates `e` to a pair value, and extracts and yields the left element value. It is a checked run-time error if `e` evaluates to a non-pair value.
- Evaluating `(snd e)` evaluates `e` to a pair value, and extracts and yields the right element value. It is a checked run-time error if `e` evaluates to a non-pair value.
- Evaluating `(setFst 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 `(setSnd 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 value tested by a `if` must be an integer; otherwise a checked run-time error results.
- The value written by a `write` can be either an integer or a pair.
- The arithmetic operators (`+`,`-`,`*`,`<=`) work only on integers; it is a checked run-time error to apply them to 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.

2. Storage Model and Environment Support

In the EL2 interpreter implementation, all variables and all pairs are stored in storage. Three storage classes, static, stack, and heap, are used.

- Global variables (*i.e.* those defined by `GDefs`) are stored in the static (*i.e.* global) storage.
- Local variables (*i.e.* those introduced by `lets`) are stored on the stack.
- Pairs are stored in the heap.

The three storage classes are implemented as follows:

```
// Storage type declarations
type Index = Int
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
}

// Three actual storage
val global = new Store()
val heap = new HeapStore()
val stack = new StackStore()
```

Variable-storage bindings are maintained in an environment:

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

Initially, the environment `genv` contains only global variables. As the interpretation process enters and leaves scopes, the enviroment gets augmented and retracted with local variables.

Each storage class has its own address type:

```
sealed abstract class Addr() { ... }
case class GlobalAddr(index:Int) extends 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 GlobalAddr(i) => global.get(i)
  case HeapAddr(i)   => heap.get(i)
  case StackAddr(i)  => stack.get(i)
}
def set(a:Addr,v:Value) = a match {
  case GlobalAddr(i) => global.set(i,v)
  case HeapAddr(i)   => heap.set(i,v)
  case StackAddr(i)  => stack.set(i,v)
}
```

3. Your Tasks

The file `Interp2.scala` contains an incomplete EL2 interpreter. Read and understand as much as possible the given code in the file before adding new code.

There are two major differences comparing this interpreter with `Interp1` from Assignment 3. First, 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)`. Second, 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.

1. Your first task is to complete the EL2 interpreter, by providing the interpretation code for the expressions. Sample code for some expressions are included in the provided program file.

   Note that `setFst` or `setSnd` can be used to create cyclic structures, but for this assignment you should just assume that won't happen.

2. Your second, *optional*, task is to implement a deep equality comparison operator, `==`. Evaluating `(== 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 compares `(fst e1)` with `(fst e2)`, and `(snd e1)` with `(snd e2)`, and yields 1 if both yield 1; otherwise yields 0.

   Note that this recursive routine may go into an infinite loop on circular pair structures. You don't need to handle this case. You may assume it does not occur in test cases.

A test program, `TestInterp2.scala`, is provided. Run your interpreter with these tests and make sure they all pass. (If you are not doing the optional Task 2, ignore the failing of the Deq test.)

4. Submission

Submit your `Interp2.scala` through the "Assignment 6" 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.