CS 558 Programming Languages Fall 2017 – Practice Midterm Exam

This exam has five questions. Each question is worth 20 points. Some questions have multiple sub-parts, whose worth is indicated in brackets. You have 75 minutes for the exam. Please write your answers on the exam paper in the spaces provided.

You may use a single one-sided 8.5x11 sheet of handwritten notes. Otherwise, the exam is closed-book, closed-notes. No computers or other electronic devices are allowed. You must work independently, and you cannot share your notes with other students.

In all the questions that ask for brief answers, it is recommended that you show your reasoning, to increase the chances of getting partial credit even if your final answer is wrong.
1. [20 pts.] **Grammars and Order of evaluation**

Consider the following concrete grammar for a language A of arithmetic expressions, where \( exp \) is the start symbol:

\[
\begin{align*}
exp & ::= \quad \text{term} \\
& \quad | \quad \text{let } \text{id} \ ' '=' \ exp \ \text{in} \ exp \\
& \quad | \quad \text{id} \ ' ':'=\ ' \ exp \\
\text{term} & ::= \quad \text{term} \ '+' \ \text{term} \\
& \quad | \quad \text{term} \ '*' \ \text{term} \\
& \quad | \quad \text{id} \\
& \quad | \quad \text{num}
\end{align*}
\]

The intended semantics are just as in the lab interpreters: \text{id} \ represents variables; \text{num} \ represents integer literals; \( x := e \) evaluates \( e \) to a value \( v \), sets the value of \( x \) to \( v \), and yields \( v \) as result; + is addition; * is multiplication; \text{let } x = e_1 \ \text{in} \ e_2 \) binds \( x \) to the value of \( e_1 \) inside \( e_2 \) and yields the value of the latter.

(a) [10 pts.] Is this grammar ambiguous? Why or why not?

(b) [10 pts.] Suppose we have two interpreters for this language. Interpreter #1 evaluates the operands of + and * from left-to-right; interpreter #2 evaluates them from right-to-left. Is there an A program that behaves differently under the two interpreters? Either show such a program or explain why it can’t exist.
2. [20 pts] Scope and Binding.

Consider the following Scala interpreter code for a simple statically-scoped language L of top-level functions and expressions. The s-expression forms for each language construct are shown in comments on the AST case class definitions. Since there is no assignment operator, we do not bother to model the store; instead, the environment binds variables directly to (integer) values.

```scala
case class Program(fdefs: List[FunDef], body: Expr) {} // ((fdefs) body)

case class FunDef(name: String, param: String, body: Expr) {} // (name param body)
sealed abstract class Expr {}

case class Num(n: Integer) extends Expr // n

case class Var(x: String) extends Expr // x

case class Add(l: Expr, r: Expr) extends Expr // (+ e e)

case class Apply(f: String, e: Expr) extends Expr // (@ f e)

case class Let(x: String, e: Expr, b: Expr) extends Expr // (let x e b)

type Env = Map[String, Int]

val emptyEnv : Env = Map[]

def interp(p: Program): Int = {
def interpE(env: Env, e: Expr): Int = e match {
case Num(n) => n
case Var(x) => env.getOrElse(x, throw InterpException("undefined variable:" + x))
case Add(l, r) => interpE(env, l) + interpE(env, r)
case Let(x, e, b) => {
  val v = interpE(env, e)
  interpE(env + (x->v), b)
}
case Apply(f, e) => {
  val v = interpE(env, e)
  for (fdef <- p.fdefs)
    if (fdef.name == f)
      return interpE(emptyEnv + (fdef.param->v), fdef.body)
  throw InterpException("undefined function:" + f)
}
}
interpE(emptyEnv, p.body)
}
```

(a) [6 pts.] In the interpreter code above, consider the three identifiers p, e, and env. Draw a box around each binding of these identifiers, and a circle around each use of these identifiers.

(b) [3 pts.] Draw an arrow from each use of identifier e (only) to the corresponding binding for e to which it refers.

(c) [3 pts.] What is the result of interpreting the following L program?

```lisp
((g z (+ z y))
(let y 21
  (@ g y)))
```

(d) [3 pts.] Suppose L used dynamic scope instead of static scope. What would be the result of the program from part (c)?

(e) [5 pts.] Change the interpreter code above to implement dynamic scope instead of static scope. (Hint: This requires changing just one variable use!) Mark your change clearly in the code listing.
3. [20 pts.] **Reference Equality**

Recall the toy language from lab 3, including `let`, `setFst`, and `setSnd` expressions (as you added them in the lab assignment). For reference, the s-expression EBNF syntax for this language is at the bottom of the page.

A value is either an integer or a (boxed) pair. As usual, the first and second elements of a pair can each be an integer or a further pair.

Assume `==` is a *structural* equality operator: it returns 1 if its arguments are identical integers or pairs with structurally equal contents, and 0 otherwise. (This is just the original meaning of `==` in the lab assignment before you modified it.)

Show that a *reference* equality operator on pairs can be defined inside language L4. In other words, write an L4 function `eqpair` such that `(@ eqpair e1 e2)` returns 1 if `e1` and `e2` are the *same* pair (have the same location), and 0 otherwise. You may assume that `eqpair` is only called on pairs. Use the s-expression syntax to write your function.

Hints: (1) Make use of the fact that pairs are mutable. (2) You may find it easier to develop the code in two stages. First, write a version of `eqpair` that works on the assumption that pairs being compared never contain the integer 42 as an element. Then, modify your code to work properly even on pairs that might contain 42.

---

**s-expression EBNF Syntax**

```
program ::= (gdecs fdecs exp)
gdecs ::= ({{ id exp }})
fdecs ::= ({{ id {(id) exp} }})
exp ::= num | id | (id exp) | (while exp exp) | (if exp exp exp) | (block {exp}) |
| (let id exp exp) | (write exp) | (+ exp exp) | (- exp exp) | (* exp exp) | (/ exp exp) | (\ exp exp) | (<= exp exp) | (== exp exp) |
| (pair exp exp) | (fst exp) | (snd exp) | (isPair exp) | (setFst exp exp) | (setSnd exp exp)
```
4. [20 pts.] **Parameter passing**

Consider the following program in Scala-like syntax.

```scala
case class L(var hd:Int, var tl:L) // see note below

def last(var x:L) : L = { // see note below
  if (x != null)
    while (x.tl != null)
      x = x.tl
  x
}

def rev(var x:L) : L = { // see note below
  var y:L = null
  while (x != null) {
    val z = x.tl
    x.tl = y
    y = x
    x = z
  }
  y
}

def main (argv:Array[String]) = {
  var w = L(1,L(2,L(3,null))) // see note below
  println("w1 = " + w) // prints "w1 = L(1,L(2,L(3,null)))"
  val a = last(w)
  println("a1 = " + a)
  println("w2 = " + w)
  var u = rev(w)
  println("u1 = " + u)
  println("w3 = " + w)
  println("a2 = " + a)
}

Note: By default, case class fields are immutable, but the `var` keywords in the definition for `L` makes `hd` and `tl` mutable fields. Function parameters are always immutable in real Scala, but we'll pretend that the `var` keywords before the parameters in the definitions of `last` and `rev` (which aren’t valid in real Scala) declare those parameters to be mutable variables that can be modified in the bodies of the functions. Otherwise, assume that the program has Scala-like semantics, except possibly for parameter passing. In particular, assume that objects of class `L` are boxed.

(a) [10 pts.] What does the program print under call-by-value semantics?

(b) [10 pts.] What does the program print under call-by-reference semantics?

```
5. [20 pts.] Axiomatic Semantics

Recall the very simple imperative language and set of rules used to illustrate axiomatic semantics in Lecture 2b, which are repeated for reference at the bottom of this page. Suppose we want to add a new repeat-until statement. The form of this statement is

\[ \text{repeat } S \text{ until } E \]

The semantics of this statement follow from the fact that it compiles to the following low-level code sequence (in the style of Lecture 2a):

\[
\begin{align*}
top: & \quad S \\
& \quad \text{evaluate } E \text{ into } t \\
& \quad \text{cmp } t, \text{true} \\
& \quad \text{brneq top}
\end{align*}
\]

Here is a valid triple describing the behavior of a particular program fragment involving repeat-until:

\[
\{ x = -1 \land y > 0 \} \\
\text{repeat} \\
x := x + y; \\
y := y - 1 \\
\text{until} \ (y \leq 0) \\
\{ x \geq 0 \land y \leq 0 \}
\]

(a) [5 pts.] Give a combination of while and sequential composition (\( ; \)) statements that is equivalent to \( \text{repeat } S \text{ until } E \).

(b) [15 pts.] Give the strongest proof rule you can for repeat-until statements. In particular, your rule should be strong enough to justify the above triple. It is not necessary to write down a proof of this triple as part of your answer, but you may find that doing so is helpful to check that your rule works as expected.

\[
\begin{align*}
\{ P[E/x] \} & \quad x := E \quad \{ P \} \quad (\text{ASSIGN}) \\
\{ P \land E \} & \quad S_1 \quad \{ Q \} \quad \{ P \land \neg E \} \quad S_2 \quad \{ Q \} \\
\{ P \} & \quad \text{if } e \text{ then } S_1 \text{ else } S_2 \quad \{ Q \} \quad (\text{COND}) \\
\{ P \land E \} & \quad S \quad \{ P \} \quad (\text{WHILE}) \\
\{ P \} & \quad \text{skip} \quad \{ P \} \quad (\text{SKIP}) \\
\{ P \} & \quad S_1 \quad \{ Q \} \quad \{ P \land \neg E \} \\
\{ P \} & \quad S_2 \quad \{ R \} \quad (\text{COMP}) \\
\{ P \} & \quad \text{while } E \text{ do } S \text{ end } \{ P \land \neg E \} \quad \{ Q \} \quad (\text{CONSEQ}) \\
\end{align*}
\]