CS 558 Programming Languages Fall 2023 – 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.

Write your name on your sheet of notes, and turn the sheet in with your exam when finished.

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:

$$
\begin{array}{rcl}
exp & ::= & term \\
 & | & \texttt{let}\ id\ \texttt{'='}\ exp\ \texttt{in}\ exp \\
 & | & id\ \texttt{':='}\ exp \\
term & ::= & term\ \texttt{'+'}\ term \\
 & | & term\ \texttt{'*'}\ term \\
 & | & id \\
 & | & num
\end{array}
$$

The intended semantics are just as in the lab interpreters: id represents variables; 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; $\star$ is multiplication; `let` x `=` e_1 `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 that both build the *same* AST as the result of parsing any given concrete input. Interpreter #1 evaluates the operands of + and $\star$ 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.

```
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[String,Int]()

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?

```
((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.] **Storage**

Consider the following C program fragment:

```
void f() {
  int a = 42;
}
int g() {
  int b;
  return b;
}
int h() {
  f();
  return g();
}
```

(a) [5 pts.] According to the C language definition, it is undefined what result value is returned by `h`. Why?

(b) [5 pts.] Suppose that when this code is given to the `gcc` C compiler (with default settings), it produces machine code that, when run, returns the result value `42` from `h`. Give a plausible explanation for why this might happen.

(c) [5 pts.] What would happen if this code were given to a Java compiler? (To make this a syntactically legal Java fragment, assume these are member functions of some class.)

(d) [5 pts.] What would we need to do to convert the body of `g` into legal Scala?

4. [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

repeat S 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):

```
top: S
     evaluate E into t
     cmp t,true
     brneq top
```

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

```
{ x = -1 ∧ y > 0}
repeat
  x := x + y;
  y := y - 1
until (y ≤ 0)
{ x ≥ 0 ∧ y ≤ 0 }
```

(a) [5 pts.] Give a combination of `while` and sequential composition (`;`) statements that is equivalent to `repeat` S `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.

$$\frac{}{\{P[E/x]\}\ \ x\mathtt{:=}E\ \ \{P\}}\ \text{(ASSIGN)} \qquad \frac{}{\{P\}\ \ \mathtt{skip}\ \ \{P\}}\ \text{(SKIP)}$$

$$\frac{\{P \wedge E\}\ \ S_1\ \ \{Q\} \quad \{P \wedge \neg E\}\ \ S_2\ \ \{Q\}}{\{P\}\ \ \mathtt{if}\ e\ \mathtt{then}\ S_1\ \mathtt{else}\ S_2\ \ \{Q\}}\ \text{(COND)} \qquad \frac{\{P\}\ \ S_1\ \ \{Q\} \quad \{Q\}\ \ S_2\ \ \{R\}}{\{P\}\ \ S_1\mathtt{;}S_2\ \ \{R\}}\ \text{(COMP)}$$

$$\frac{\{P \wedge E\}\ \ S\ \ \{P\}}{\{P\}\ \ \mathtt{while}\ E\ \mathtt{do}\ S\ \mathtt{end}\ \{P \wedge \neg E\}}\ \text{(WHILE)} \qquad \frac{P \Rightarrow P' \quad \{P'\}\ \ S\ \ \{Q'\} \quad Q' \Rightarrow Q}{\{P\}\ \ S\ \ \{Q\}}\ \text{(CONSEQ)}$$

5. [20 pts.] **Operational Semantics**

Consider a simple expression language similar to the one used to illustrate formal operational semantics in Lecture 4a, where we gave rules for evaluation judgments of the form $\langle e, E, S\rangle \Downarrow \langle v, S'\rangle$. Suppose we want to add a non-deterministic guarded-if expression to this language. For simplicity, we limit the expression to two arms; its syntax is `(gifnz` e_{c1} e_{b1} e_{c2} e_{b2}`)`. Informally, this expression is evaluated as follows: evaluate the guard conditions e_{c1} and e_{c2}, in that order, to values v_{c1} and v_{c2}; *non-deterministically* choose some $i \in \{1, 2\}$ such that v_{ci} is non-zero; evaluate the corresponding body e_{bi} and yield its value as the result of the entire `gifnz` expression. If it is impossible to choose such an i, the expression cannot be evaluated successfully.

Write down one or more formal operational evaluation rules that capture the semantics of `gifnz`.

$$\frac{l = E(x) \quad v = S(l)}{\langle x, E, S\rangle \Downarrow \langle v, S\rangle}\ \text{(Var)} \qquad \frac{}{\langle i, E, S\rangle \Downarrow \langle i, S\rangle}\ \text{(Int)} \qquad \frac{\langle e_1, E, S\rangle \Downarrow \langle v_1, S'\rangle \quad \langle e_2, E, S'\rangle \Downarrow \langle v_2, S''\rangle}{\langle \texttt{(+}\ e_1\ e_2\texttt{)}, E, S\rangle \Downarrow \langle v_1 + v_2, S''\rangle}\ \text{(Add)}$$

$$\frac{\begin{array}{c}\langle e_1, E, S\rangle \Downarrow \langle v_1, S'\rangle \qquad l \notin \mathit{dom}(S') \\ \langle e_2, E + \{x \mapsto l\}, S' + \{l \mapsto v_1\}\rangle \Downarrow \langle v_2, S''\rangle\end{array}}{\langle \texttt{(local}\ x\ e_1\ e_2\texttt{)}, E, S\rangle \Downarrow \langle v_2, S'' - \{l\}\rangle}\ \text{(Local)} \qquad \frac{\langle e, E, S\rangle \Downarrow \langle v, S'\rangle \quad l = E(x)}{\langle \texttt{(:=}\ x\ e\texttt{)}, E, S\rangle \Downarrow \langle v, S' + \{l \mapsto v\}\rangle}\ \text{(Assgn)}$$

$$\frac{\langle e_1, E, S\rangle \Downarrow \langle v_1, S'\rangle \quad v_1 \neq 0 \quad \langle e_2, E, S'\rangle \Downarrow \langle v_2, S''\rangle}{\langle \texttt{(if}\ e_1\ e_2\ e_3\texttt{)}, E, S\rangle \Downarrow \langle v_2, S''\rangle}\ \text{(If-nzero)} \qquad \frac{\langle e_1, E, S\rangle \Downarrow \langle 0, S'\rangle \quad \langle e_3, E, S'\rangle \Downarrow \langle v_3, S''\rangle}{\langle \texttt{(if}\ e_1\ e_2\ e_3\texttt{)}, E, S\rangle \Downarrow \langle v_3, S''\rangle}\ \text{(If-zero)}$$

$$\frac{\begin{array}{c}\langle e_1, E, S\rangle \Downarrow \langle v_1, S'\rangle \qquad v_1 \neq 0 \qquad \langle e_2, E, S'\rangle \Downarrow \langle v_2, S''\rangle \\ \langle \texttt{(while}\ e_1\ e_2\texttt{)}, E, S''\rangle \Downarrow \langle v_3, S'''\rangle\end{array}}{\langle \texttt{(while}\ e_1\ e_2\texttt{)}, E, S\rangle \Downarrow \langle v_3, S'''\rangle}\ \text{(While-nzero)} \qquad \frac{\langle e_1, E, S\rangle \Downarrow \langle 0, S'\rangle}{\langle \texttt{(while}\ e_1\ e_2\texttt{)}, E, S\rangle \Downarrow \langle 0, S'\rangle}\ \text{(While-zero)}$$