CS 558 Programming Languages Fall 2016 – Practice Midterm

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.] Abstract Syntax Design

Consider a toy language of expressions with the following EBNF concrete syntax:

\[
\begin{align*}
\text{stmt} & \rightarrow \text{if expr then stmt} [ \text{else stmt} ] | \text{var ::= expr} \\
\text{expr} & \rightarrow \text{expr} + \text{term} | \text{expr} - \text{term} | \text{term} \\
\text{term} & \rightarrow \text{term} \ast \text{factor} | \text{term} / \text{factor} | \text{factor} \\
\text{factor} & \rightarrow \text{var} | \text{num}
\end{align*}
\]

Now consider the following tree grammar for a proposed abstract syntax for the same language, intended to support semantic analysis and evaluation in the usual way.

\[
\begin{align*}
\text{IfS} : \text{Stmt} & \rightarrow \text{Expr} \text{ Stmt} \text{ Stmt} \\
\text{AsgnS} : \text{Stmt} & \rightarrow (\text{string}) \text{ Expr} \\
\text{PlusE} : \text{Expr} & \rightarrow \text{Expr} \text{ Term} \\
\text{MinusE} : \text{Expr} & \rightarrow \text{Expr} \text{ Term} \\
\text{TimesT} : \text{Term} & \rightarrow \text{Term} \text{ Term} \\
\text{DivideT} : \text{Term} & \rightarrow \text{Term} \text{ Term} \\
\text{VarT} : \text{Term} & \rightarrow (\text{string}) \\
\text{NumT} : \text{Term} & \rightarrow (\text{int})
\end{align*}
\]

Describe at least two significant defects in this proposed abstract syntax, and suggest how to fix them.
2. [20 pts.] Storage

Consider the following C program fragment:

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

(a) [6 pts.] According to the C language definition, it is undefined what result value is returned by \( h \). Why?

(b) [7 pts.] The code generated by `gcc -O0` actually returns the result value 42 from \( h \). Give a plausible explanation for why this might happen.

(c) [7 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.)
3. [20 pts.] **Operational Semantics of Boolean Expressions**

Consider a simple imperative language that includes boolean expressions such as `and`, `or`, and `not`.

Here is a possible operational semantics rule that describes evaluation for the `and` expression, where the (mathematical) expression `a & b` returns `true` if both `a` and `b` are the value `true`, and `false` otherwise.

\[
\begin{align*}
\langle e_1, E, S \rangle & \Downarrow \langle v_1, S' \rangle \quad \langle e_2, E, S' \rangle & \Downarrow \langle v_2, S'' \rangle \quad v = v_1 \& v_2 \\
\langle \text{and} \ e_1, e_2, E, S \rangle & \Downarrow \langle v, S'' \rangle
\end{align*}
\]

(And)

Suppose we want our language to use *short-circuit evaluation* semantics for `and` expressions. The rule given above does not express these semantics. Give alternative rules that do (you will need two of them).
4. [20 pts.] Axiomatic Semantics

Recall the very simple imperative language and set of rules used to illustrate axiomatic semantics in Lecture, which are repeated for reference at the bottom of this page.

Suppose we add to the language a C/C++/Java-style for statement:

\[
\text{for } (S_1; E; S_2) \ S_3
\]

The semantics of this statement are exactly equivalent to

\[
\begin{align*}
\text{begin} \\
S_1; \\
\text{while } E \text{ do begin } S_3; S_2 \text{ end} \\
\text{end}
\end{align*}
\]

Here is a valid triple describing the behavior of a particular program fragment involving for:

\[
\{ x = c \} \for \ (y := 0; x != 0; x := x - 1) \ y := y + 1 \ \{ y = c \}
\]

(a) [16 pts.] Give the strongest sound proof rule you can for for statements. It should be strong enough to justify the above triple. (Hint: The pre- and post-conditions mentioned in your rule should be built from three general propositions \( P \), \( Q \), and \( R \), as well as the expression \( E \).)

(b) [4 pts.] What happens when \( c < 0 \)? Why doesn’t this make the above triple invalid?

\[
\begin{align*}
\{P[E/x]\} & \ x := E \ \{P\} & \text{(ASSIGN)} \\
\{P \land E\} & \ S_1 \ \{Q\} & \{P \land \neg E\} \ S_2 \ \{Q\} & \text{(COND)} \\
\{P\} & \text{if } e \text{ then } S_1 \text{ else } S_2 \ \{Q\} \\
\{P \land E\} & \ S \ \{P\} & \text{(WHILE)} \\
\{P\} & \text{while } E \text{ do } S \ \{P \land \neg E\} \\
\{P\} & \text{skip} \ \{P\} & \text{(SKIP)} \\
\{P\} & S_1 \ \{Q\} & \{Q\} \ S_2 \ \{R\} & \text{(COMP)} \\
\{P\} & \text{if } e \text{ then } S_1 \text{ else } S_2 \ \{R\} \\
\{P\} & \text{while } E \text{ do } S \ \{P \land \neg E\} \\
P & P' \ \{P'\} & S \ \{Q'\} & Q' \Rightarrow Q & \text{(CONSEQ)}
\end{align*}
\]
5. [20 pts.] Parameter passing

Consider the following program in Scala-like syntax.

```scala
case class P(var a:Int) // see note below

def twiddle(var x:P, var y:P) = { // see note below
  val z = x
  x = y
  y = z
}

def swizzle(x:P, y:P) = {
  val z = x.a
  x.a = y.a
  y.a = z
}

def main () = {
  var p0 = P(0)
  var p1 = P(1)
  twiddle(p0,p1)
  println (p0.a + " " + p1.a)
  swizzle(p0,p1)
  println (p0.a + " " + p1.a)
}
```

Note: Scala statically distinguishes between identifiers for immutable values and identifiers for mutable variables. For local declarations, this distinction is marked by using the keyword `val` or `var`, respectively. By default, case class fields are immutable, but the `var` keyword in the definition for `P` makes `a` a mutable field. Function parameters are always immutable in real Scala, but we'll pretend that the `var` keywords before the parameters in the definition of `twiddle` (which aren't valid in real Scala) declare those parameters to be mutable variables that can be modified in the body of `twiddle`.

Otherwise, assume that the program has Scala-like semantics, except for parameter passing. In particular, assume that objects of class `P` are boxed.

(a) What does the program print under call-by-value semantics?

(b) What does the program print under call-by-reference semantics?

(c) Now suppose that objects of class `P` are not boxed, and the semantics of assignment are adjusted appropriately. Now what does the program print under call-by-value semantics?