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:

```
stmt → 'if' expr 'then' stmt [ 'else' stmt ] | var ':=' expr
expr → expr '+' term | expr '-' term | term
term → term '*' factor | term '/' factor | factor
factor → var | num
```

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.

```
IfS : Stmt → Expr Stmt Stmt
AsgnS : Stmt → (string) Expr
PlusE : Expr → Expr Term
MinusE : Expr → Expr Term
TimesT : Term → Term Term
DivideT : Term → Term Term
VarT : Term → (string)
NumT : Term → (int)
```

Describe at least two significant defects in this proposed abstract syntax, and suggest how to fix them.

**Answer:**

(i) There is no way to represent ‘if’ statements that are missing the optional ‘else’ clause. To fix this, we could add a new node label for such “elseless ‘if’ s”, or (probably better) add a new EmptyS node label to represent “empty” (i.e., missing) statements.

(ii) There is no way to express an expression that is just a term. To fix this, we could add a production like

```
TermE : Expr → Term
```

(iii) But in any case, there is no reason to maintain the distinction between ‘expr’ and ‘term’ in the abstract syntax; it is introduced in the concrete syntax purely to capture operator precedence for parsing purposes. In the AST, syntactic category ‘term’ should be removed just as ‘factor’ was, by making Times, Divide, Var, and Num all node labels for Exprs.
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 might this 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.)

Answer:

(a) The result of `h` is the result of `g`, which is the value of `b`, which was never initialized and is therefore undefined.

(b) In the generated code, the stack location of `b` is the same as the location of local variable `a` in `f`. The call to `f` sets this location to 42; the code in `g` reads this location and uses it as the value for `b`, which is then returned. (Note: Without knowing what assembly code `gcc` generates, you can’t be sure this is why the program behaves as it does, but it’s the most probable explanation.)

(c) A Java compiler would reject the program (during semantic analysis), because the use of `b` before it has been defined violates the “definite assignment” property.
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.

$$\langle e_1, E, S \rangle \Downarrow \langle v_1, S' \rangle \quad \langle e_2, E, S' \rangle \Downarrow \langle v_2, S'' \rangle \quad v = v_1 \& v_2$$

$$\langle (\text{and } e_1, e_2), E, S \rangle \Downarrow \langle v, S'' \rangle \quad \text{(And)}$$

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).

**Answer:**

$$\langle e_1, E, S \rangle \Downarrow \langle \text{true}, S' \rangle \quad \langle e_2, E, S' \rangle \Downarrow \langle v_2, S'' \rangle$$

$$\langle (\text{and } e_1, e_2), E, S \rangle \Downarrow \langle v_2, S'' \rangle \quad \text{(And1)}$$

$$\langle e_1, E, S \rangle \Downarrow \langle \text{false}, S' \rangle$$

$$\langle (\text{and } e_1, e_2), E, S \rangle \Downarrow \langle \text{false}, S' \rangle \quad \text{(And2)}$$
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:

```
for (S₁; E; S₂) S₃
```

The semantics of this statement are exactly equivalent to

```
begin
S₁;
while E do begin S₃; S₂ end
end
```

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?

**Answer:**

(a) 

\[
\frac{\{ P \} S₁ \{ Q \} \{ Q \wedge E \} S₃ \{ R \} \{ R \} S₂ \{ Q \}}{\{ P \} \text{ for } (S₁; E; S₂) S₃ \{ Q \wedge \neg E \}} \quad \text{(FOR)}
\]

(b) There are two reasonable answers. If we assume indefinitely large magnitude integers, then the program will loop forever if \( c < 0 \). The assertion remains true in the sense of partial correctness: it says that if the fragment terminates, then \( y = c \); it says nothing in the case when the fragment doesn’t terminate.

Alternatively, if we assume machine-like integers of fixed size, they will eventually “wrap around,” and the fragment will terminate with \( y = c \).
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?

**Answer:**

(a) Under call-by-value, **twiddle** has no effect on **p0** or **p1**, so the first two outputs are **0 1**: **swizzle** actually exchanges the contents of the **a** fields, so the second two outputs are **1 0**.

(b) Under call-by-reference, **twiddle** actually exchanges the values of **p0** and **p1**, so the first two outputs are **1 0**: **swizzle** exchanges things as before, so the second two outputs are **0 1**.

(c) Assuming un-boxed semantics, the parameters to **twiddle** and **swizzle** are copied when they are passed, creating new objects, and similarly for the assignment to **z** in **twiddle**. This has no effect on the visible behavior of **twiddle**, so the first two outputs are again **0 1**. But **swizzle** now operates on local copies of its arguments, so it has no effect on the variables in **main**, and the second two outputs are also **0 1**.