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 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.)

**Answer:**

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

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

(c) 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.] **Scope and Parameters**

Consider the following program in Scala-like syntax.

```scala
1  var a = 3
2  def f (b:Int) =
3      a + b
4  def g (var c:Int) = { // see note below
5      val a = c + 1
6      c = f(a)
7      def h (d:Int) =
8          d + c
9          h(10)
10     }
11  def j () = {
12      val b = g(a)
13      b + a
14     }
```

Note: Function parameters are always immutable in real Scala, but we’ll pretend that the `var` keyword before the parameter in the definition of `g` (which isn’t valid in real Scala) declares this parameter to be a mutable variable that *can* be modified in the body of `g`.

(a) [5 pts.] Assuming standard Scala **static scoping**, we can describe the variable environment at line 3 by listing the names in scope and the line at which each was bound, as follows:

```
{ a: line 1, b: line 2 }
```

Use the same style to describe the variable environments at line 8 and at line 13.

**Answer:**

At line 8: `{ a: line 5, c: line 4, d: line 7 }`.
At line 13: `{ a: line 1, b: line 12 }`.

(b) [5 pts.] Still assuming **static scoping** and also assuming **call-by-value** parameter passing, what value is returned by function `j`?

**Answer:** 20

(c) [5 pts.] Now assuming **static scoping** but **call-by-reference** parameter passing, what value is returned by `j`?

**Answer:** 24

(d) [5 pts.] Now assuming **dynamic scoping** of variables and **call-by-value** parameter passing, what is the environment seen when evaluating line 3, and what value is returned by `j`?

**Answer:**

At line 3: `{ a: line 5, b: line 2, c: line 4 }`.

Returned value: 21
4. [20 pts.] **Operator Precedence and Order of evaluation**

Consider a language L with addition (+), multiplication (*) and assignment (:=) operators, where, as usual, the value of an assignment expression is the value of its right-hand side.

Suppose we know that addition and multiplication both have higher precedence than assignment and that they do not have the same precedence as each other, but we don’t know which has higher precedence. Suppose further that we know that operands are either always evaluated left-to-right or always evaluated right-to-left, but we don’t know which.

Finally, suppose that in order to determine these aspects of L’s semantics we evaluate the following concrete expression in an environment and store where the value of x is initially 1.

\[(x := 3) + (x := x + 4) * (x := x + 2)\]

For each of the following possible answers, indicate which combination of precedence ordering and evaluation ordering would lead to that answer. If no combination would, say “impossible.”

(a) 90
(b) 24
(c) 38
(d) 66

**Answer:**

(a) addition higher, left-to-right
(b) multiplication higher, right-to-left
(c) impossible
(d) multiplication higher, left-to-right

The missing combination (addition higher, right-to-left) generates a result of 30.
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 loop-exitif statement, which provides loops that break out in the middle. The form of this statement is

\[
\text{loop } S_1 \text{ exitif } (E) \text{ } S_2 \text{ end}
\]

This statement can be compiled to the following low-level code sequence (as shown in Lecture 2a):

```
top: $s_1$
  evaluate $e$ into $t$
  cmp $t$,true
  breq done
  $s_2$
  br top
done:
```

Here is a valid triple describing the behavior of a particular program fragment involving loop-exitif:

\[
\{ x \geq 0 \land y \geq 0 \}\n\text{loop}
  \begin{align*}
  & x := x - 1 \\
  & \text{exitif } (x < 0) \\
  & y := y + x \\
  \end{align*}
\text{end} \\
\{ x = -1 \land y \geq 0 \}
\]

(a) [5 pts.] Give a combination of while and sequential composition (;) statements that is equivalent to loop $S_1$ exitif $E$ $S_2$ end.

Answer:

$S_1; \text{while } (\neg E) \text{ do } S_2; S_1 \text{ end}$

(b) [15 pts.] Give the strongest proof rule you can for loop-exitif 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.

Answer:

\[
\frac{ \{P\}S_1\{Q\} \quad \{Q \land \neg E\}S_2\{P\} }{ \{P\} \text{loop } S_1 \text{ exitif } (E) \text{ } S_2 \text{ end } \{Q \land E\} } \quad \text{(LOOP-EXITIF)}
\]

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

\[
\begin{align*}
\{P\} & \text{skip } \{P\} & \text{(SKIP)} \\
\{P\} & S_1 \{Q\} \{Q\} S_2 \{R\} & \text{(COMP)} \\
\{P\} & S_1; S_2 \{R\} & \text{(COMP)} \\
\{P\} & \Rightarrow P' \{P'\} S \{Q'\} & \text{(CONSEQ)} \\
\{P\} & S \{Q\} & \text{(CONSEQ)}
\end{align*}
\]