You can expect the exam to consist of about ten questions similar to the following ones. Each question will probably be worth the same number of points. You’ll have about two hours for the exam. Remember that it is an open-book, open-notes exam; you can use any reference materials you wish. However, you must work independently, and you cannot share reference materials with other students.

1. The GNU C compiler supports an extension to ANSI C called “statement expressions,” which allows any compound statement to be treated an expression by enclosing it in parentheses. For example, we can write

```
#define maxint(a,b) ({ int a1 = (a), b1 = (b); a1 > b1 ? a1 : b1; })
```

Compare this to the more usual C definition

```
#define max(a,b) ((a) > (b) ? (a) : (b))
```

Give an example where these two macros behave differently, and explain why maxint is probably a better definition?

**Answer:** Consider passing expressions that have side-effects. For example,

```
maxint(i++,j++)
```

expands to

```
({ int a1 = (i++); b1 = (j++); a1 > b1 ? a1 : b1; })
```

so i and j are each incremented just once. But

```
max(i++,j++)
```

expands to

```
( (i++) > (j++) ? (i++) : (j++); )
```

so either i or j is incremented twice. The former is more likely to be what the user of the macro expected to happen.
2. Consider the following C program fragment:

```c
void f() {
    int a = 2;
}
int g() {
    int b;
    return b;
}
int h() {
    f();
    return g();
}
```

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

(b) The code generated by \texttt{gcc -O0} actually returns the result value 2 from \( h \). Why might this happen?

(c) What would happen if this code were given to a Java compiler? (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 2; 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 \texttt{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. Consider the following grammar, where the non-terminals are \{\text{INT},+,*\} and the start symbol is $\text{expr}$:

$$
\text{expr} \ := \ '+' \ \text{expr} \ \text{expr} \\
      \ | \ '('* \ \text{expr} \ \text{expr} \\
      \ | \ \text{INT}
$$

Is this grammar ambiguous or not? Give brief but convincing evidence for your answer.

Answer: This grammar is not ambiguous. To be ambiguous, a grammar must allow two different parse trees, and hence two different left-most derivations, for the same input string. Therefore, at some point in a left-most derivation, there must be choice of which rule to use to expand the left-most non-terminal. But in this grammar, all the rules match a different initial symbol, so at most one rule can ever be successfully applied at any given point in the input string. Hence the grammar must not be ambiguous.

(Note: In general, it is quite hard to prove non-ambiguity except for very simple grammars like this one; proving ambiguity (by exhibiting two different parse trees for a single input) is much easier.)

4. The Oberon programming language has local (nested) procedures and also permits procedure variables. To quote the language definition: it takes “the view that procedures are themselves objects that can be assigned to variables.” But only top-level (non-nested) procedures may be assigned into variables. What is the most likely motivation for this restriction?

Answer: It is reasonable to assume that Oberon was designed to use a stack to store procedure activation records. If a nested procedure could be assigned into a variable (say into a global variable), it could remain alive (and callable) after its enclosing procedure returned. Since the nested procedure might reference local variables or parameters of the enclosing procedure, they would need to remain alive as long as the nested procedure did. But if stack allocation of activation records is used, this wouldn’t generally be true. By only allowing top-level procedures to be stored in variables, Oberon guarantees that this problem can’t occur.
5. Consider the following attribute grammar, where \texttt{id} and \texttt{num} are terminal symbols with attributes \texttt{s} and \texttt{val} respectively, and \texttt{extend} and \texttt{lookup} behave just like the environment operators in the homework interpreters.

\[
\text{exp} ::= \text{'let'}\ assigns \text{'in'}\ \text{expl} \quad \{ \text{assigns.env} = \text{exp.env}; \\
\text{expl.env} = \text{assigns.envout}; \\
\text{exp.val} = \text{expl.val} \} \\
| \ \text{id} \quad \{ \text{exp.val} = \text{lookup(\text{exp.env}, \text{id.s})} \} \\
| \ \text{num} \quad \{ \text{exp.val} = \text{num.val} \}
\]

\[
\text{assigns} ::= \text{id} \text{ '=' } \text{exp} \quad \{ \text{exp.env} = \text{assigns.env}; \\
\text{assigns.envout} = \\
\text{extend(\text{assigns.env}, \text{id.s}, \text{exp.val})} \} \\
| \ \text{assigns1 'and' id } \text{ '=' } \text{exp} \quad \{ \text{assigns1.env} = \text{assigns.env}; \\
\text{exp.env} = \text{assigns.env}; \\
\text{assigns.envout} = \\
\text{extend(\text{assigns1.envout}, \text{id.s}, \text{exp.val})} \}
\]

We take the semantics of an expression to be the synthesized .\texttt{val} attribute for the root of its parse tree, assuming that the initial inherited .\texttt{env} attribute for the root is the empty environment. Now consider the following two expressions:

\[
\text{let x = 1} \quad \text{in let x = 2 and y = x} \quad \text{in y} \quad \text{let x = 1} \quad \text{in let x = 2} \quad \text{in let y = x} \quad \text{in y}
\]

Do these expressions have the same semantics? If so, what value do they (both) compute? If not, what is a sensible way to change the attribute grammar so that both expressions \textit{do} compute the same value?

\textbf{Answer:} The expression on the left computes 1, because the second binding of \texttt{x} does not hide the first binding of \texttt{x} for purposes of computing the value of \texttt{y}. The expression on the right computes 2, because the second binding of \texttt{x} hides the first binding of \texttt{x} completely. To change the semantics so that the expressions compute the same value, just change the second attribute rule for last grammar production to

\[
\text{exp.env} = \text{assigns1.envout}
\]

\textit{(Note: The usual terminology is that the given semantics describes a “parallel” form of multi-assignment \texttt{let} whereas the revised semantics describe a “sequential” form.)}

\textit{(Further note: This problem is a probably a little too long to appear on the actual exam. However, you can expect to see similar, though shorter, problems.)}