1. Consider the following program, written in ML syntax.

   let fun h (z) =
   let val a = z
   fun f (x) = x + a
   fun g (y) =
     let val a = 10
     in f (y + a)
     end
   in g (z - 10)
   end
in h(0)
end

(a) What is the value of this expression assuming lexical scoping?
(b) What is the value of this expression assuming dynamic scoping?

Answer: (a) 0 (b) 10
2. Suppose you want to determine experimentally the order in which your C compiler evaluates function arguments. You can assume that the compiler consistently uses either left-to-right or right-to-left evaluation order on all calls. Write a C program that will print the string "left-to-right" or "right-to-left" as appropriate.

One simple answer:

```c
void f (int x, int y) {}

main () {
    int z;
    f (z = 0, z = 1);
    if (z == 0)
        printf ("right-to-left\n");
    else
        printf ("left-to-right\n");
}
```
3. 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 `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 `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.
4. Consider the following grammar, where the non-terminals are \{\text{INT},+,\ast\} and the start symbol is \text{expr}:

\[
\text{expr} := '+' \text{ expr } \text{ expr} \\
| '\ast' \text{ expr } \text{ expr} \\
| \text{ INT}
\]

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.)
5. Java implementations ordinarily allocate objects (class instances) in the heap, but clever compilers can sometimes determine that it is safe to allocate them in the stack frame of the function that created them, because they are no longer accessible after that function returns. Consider the following Java program, which creates 101 instances of class T (having v values 0,1,…,100) and ultimately prints the value 100.

```java
class example {
    static class T {
        int v;
        T (int v) {this.v = v;}
    }
    static T f (T x) {
        T y = new T(x.v + 1);
        if (y.v < 100) {
            return f(y);
        } else
            return y;
    }
    public static void main (String args[]) {
        T z = f (new T(0));
        System.out.println(z.v);
    }
}
```

(a) For which, if any, values of v, would it be safe to stack-allocate the corresponding T object? Why?

(b) Assume that the compiler internally marks each occurrence of the new keyword in the static program with a fixed allocation decision (“stack” or “heap”). Which, if either, of the two new keywords in this program could it safely mark as “stack”? Why?

Answer: (a) All the instances except 100 are only passed downward, and their lifetime does not exceed that of their creating function, so they could be stack-allocated. Instance 100 is returned upwards from the innermost call of f all the way to main; since its lifetime exceeds that of its creating function, it must be heap-allocated.

(b) The new within f must be marked “heap” because instance 100 is created there. Only the new within main can be marked “stack.” (Note that based on these markings, only one dynamic instance (with value 0) would be stack-allocated.)
6. Consider the following ML program:

```ml
fun f(x:int ref, y:int ref) : int ref = 
  if !x > !y then x else y
fun g(x:int ref) : unit = 
  x := !x * 2
val a = ref 1
val b = ref 2
fun main() : unit = 
  (g(f(a,b));
   print (Int.toString (!a + !b)))
```

(a) What will be printed when `main()` is executed?

(b) Translate this program as closely as possible into either C or C++. Treat `a` and `b` as global variables.

(Note: If you choose C, you’ll need to use pointers. If you choose C++, you may use either pointers or references, whichever you prefer.)

**Answer:** Here’s a C solution (on the left) and a C++ solution using references (on the right):

```c
#include <stdio.h>
int *f (int *x, int *y) {
  if (*x > *y)
    return x;
  else
    return y;
}
void g(int *x) {
  *x = *x * 2;
}
int a = 1;
int b = 2;
main () {
  g(f(&a,&b));
  printf ("%d\n", a+b);
}
```

```c++
#include <iostream.h>
int &f (int &x, int &y) {
  if (x > y)
    return x;
  else
    return y;
}
void g (int &x) {
  x = x * 2;
}
int a = 1;
int b = 2;
main () {
  g(f(a,b));
  cout << a+b << "\n";
}
```
7. The GNU C compiler supports an extension to ANSI C called “statement expressions,” which allows any compound statement to be treated as 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.
8. Control Statements

Consider a counted for loop statement, with the general form

\[ \text{for } \text{var} := \text{exp}_\text{first} \text{ to } \text{exp}_\text{last} \text{ by } \text{exp}_\text{step} \text{ do } \text{stmt} \]

We can give the semantics for this statement form by giving its translation into lower-level constructs as follows (where last, step, top and done are fresh names):

\[
\begin{align*}
\text{var} & := \text{exp}_\text{first}; \\
\text{last} & := \text{exp}_\text{last}; \\
\text{step} & := \text{exp}_\text{step}; \\
\text{top}: & \text{ if } \text{var} > \text{exp}_\text{last} \text{ goto done;} \\
& \text{ stmt;} \\
& \text{ var} := \text{var} + \text{exp}_\text{step}; \\
& \text{ goto top;} \\
\text{done}: &
\end{align*}
\]

For example, according to this semantics, the program

\[ \text{for } \text{i} := 1 \text{ to } 10 \text{ by } 2 \text{ do } \text{print i} \]

would print 1, 3, 5, 7, 9.

(a) Describe what is printed by the following program, based on the given semantics:

\[
\begin{align*}
\text{j} & := 8; \\
\text{i} & := 3; \\
\text{for } \text{i} := 1 \text{ to } \text{j} \text{ step } \text{i} \text{ do} \\
& \text{ begin} \\
& \text{ print i;} \\
& \text{ j} := 5; \\
& \text{ end;} \\
& \text{ print i;}
\end{align*}
\]

(b) There are many plausible alternatives to the given semantics that make different choices about where and when the bounds and step value get computed. Give such an alternative which would make the program in part (a) prints 1, 2, 4, 8. (Keep your modification as simple as possible.)

Answer: (a) 1, 2, 3, 4, 5, 6, 7, 8, 9. (b) Don’t evaluate \( \text{exp}_\text{last} \) and \( \text{exp}_\text{step} \) into variables. Instead, re-evaluate them each time around the loop:

\[
\begin{align*}
\text{var} & := \text{exp}_\text{first}; \\
\text{top}: & \text{ if } \text{var} > \text{exp}_\text{last} \text{ goto done;} \\
& \text{ stmt;} \\
& \text{ var} := \text{var} + \text{exp}_\text{step}; \\
& \text{ goto top;} \\
\text{done}: &
\end{align*}
\]