1. Consider the following program, written in (a concrete syntax for) the “Toy Language” homework two.

```plaintext
letfun h (z) =
  let a = z
  in letfun f (x) = x + a
  in letfun g (y) =
    let a = 10
    in f (y + a)
    in g (z - 10)
  in h(0)
```

(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 two possible definitions of the \texttt{exists} function in ML.

\begin{verbatim}
fun exists1 (p,l) = foldr (fn (h,u) => (p h) orelse u,false) l

fun exists2 (p,l) =  
case l of  
  [] -> false  
  | (h::t) -> (p h) orelse (exists2(p,t))
\end{verbatim}

Assume \texttt{foldr} is defined exactly as in class and in homework 3:

\begin{verbatim}
fun foldr (f,n) =  
  let fun r l =  
    case l of  
      nil -> n  
      | h::t -> f(h, r t)  
  in r  
end
\end{verbatim}

Suppose that \( l \) is a very long list, and that \( p \) returns \texttt{true} on the first element of \( l \). Is one version of \texttt{exists} likely to run faster than the other when applied to \( p \) and \( l \)? If so, which one and why? (Notes: \texttt{orelse} is ML's boolean or operator; it is short-circuiting. ML uses applicative order evaluation.)

\textbf{Answer:} \texttt{exists2} is likely to run much faster. It applies \( p \) to each element of the list from left to right, and so stops examining the list (and return \texttt{true}) as soon as it looks at the first element (because \texttt{orelse} is short-circuiting). But \texttt{exists1} uses \texttt{foldr}, which applies \( p \) to the \texttt{every} element of the list (from right to left), regardless of whether it finds an element for which \( p \) is \texttt{true}, because the recursive call to \( r \) is evaluated \textit{before} function \( f \) is called.
4. Consider the following attribute grammar:

\[
\begin{align*}
\text{exp} & := \text{'let' id '=' exp1 'in' exp2} \\
& \quad \{ \text{exp1.env} = \text{extend(exp.env, id.s, exp2.val)}; \\
& \quad \text{exp2.env} = \text{exp.env}; \\
& \quad \text{exp.val} = \text{exp1.val}; \} \\
| \text{exp1} '+' \text{exp2} & \quad \{ \text{exp1.env} = \text{exp.env}; \\
& \quad \text{exp2.env} = \text{exp.env}; \\
& \quad \text{exp.val} = \text{exp1.val * exp2.val} \} \\
| \text{exp1} '*' \text{exp2} & \quad \{ \text{exp1.env} = \text{exp.env}; \\
& \quad \text{exp2.env} = \text{exp.env}; \\
& \quad \text{exp.val} = \text{exp1.val - exp2.val} \} \\
| \text{id} & \quad \{ \text{exp.val} = \text{lookup(exp.env, id.s)} \} \\
| \text{num} & \quad \{ \text{exp.val} = \text{num.val} \}
\end{align*}
\]

Here id and num are terminal symbols with attributes s and val respectively. Function lookup(env, var) returns the value bound to var in environment env, and extend(env, var, value) returns a new environment just like env except that it binds var to value.

Using this grammar, draw a parse tree for the expression

\[
\text{let } x = x + 10 \text{ in } 6 * 4
\]

and annotate each node of your tree with the value of its .val attribute. (Hint: Read the attribute equations carefully!)

**Answer:** Here’s the annotated parse tree. (Some of the attribute equations are rather strange!)

```
_.val = 20

| exp
  |---
     |---
        |---
           |---
              |---
                |---
                   |---
                      |---

'.val =20

's=''x''

let' id

'='

exp

'in'

exp

'.val =2

 '+'

exp

'.val =10

exp

'*'

exp

'.val =6

exp

'.val =4

'id'

'.val =10

num

'.val =6

num

'.val =4

num
```

4
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. Consider the following Java program, which creates 101 instances of class \( T \) (having \( v \) values \( 0, 1, \ldots, 100 \)) 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 schematic description of the code generated for a `while` statement:

```
stm := 'while' '(' bexp ')' stm
lab1:
  <code for bexp, with trueLabel = lab2, falseLabel = lab3>
lab2:
  <code for stm1>
goto lab1
lab3:
```

Here the code for `bexp` is parameterized by a `trueLabel` and `falseLabel`, just in the interpreter for homework 4.

Write a similar schematic description of the code generated for a `do...while` statement, giving it the same semantics as in Java, C++ or C. (Ignore the possibility of `break`, `continue`, or `goto` statements within the loop body.) The grammar for the statement is:

```
stm := 'do' stm1 'while' '(' bexp ')'```

**Answer:**

```
lab1:
  <code for stm1>
  <code for bexp, with trueLabel = lab1, falseLabel = lab2>
lab2:
```
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

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

Compare this to the more usual C definition

```c
#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,

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

expands to

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

so `i` and `j` are each incremented just once. But

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

expands to

```c
( (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. 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.