CS 571
Homework 2 Solution

Due: Oct 19 **100 points**
No Late Submissions

Important Reminder: As per the course Academic Honesty Statement, cheating of any kind will minimally result in receiving an F letter grade for the entire course.

To be submitted on paper in class.

1. On a particular 32-bit architecture, a stack frame for a function with `n` parameters and `m` local variables is laid out as follows:

   Parameter `n` at offset `4 * (n + 1)` from the frame-pointer
   ...
   Parameter 1 at offset `8` from the frame-pointer
   Return address at offset `4` from the frame-pointer
   Saved frame-pointer at offset `0` from the frame-pointer
   Local variable 1 at offset `-4` from the frame-pointer
   Local variable 2 at offset `-8` from the frame-pointer
   ...
   Local variable `m` at offset `-4*m` from the frame-pointer.

   Assuming that the stack-frame contains only the above:

   (a) Give a formula in terms of `n` and `m` for the size (in bytes) of a stack-frame.

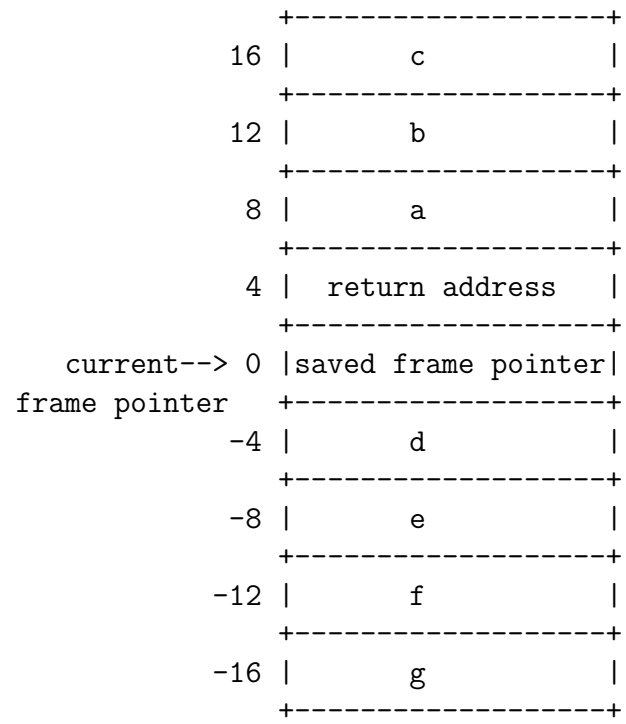
   (b) Given the function

```
f(int a, int b, int c) { //parameters a, b, c
  var d, e, f, g;        //local vars d, e, f, g.
   ...
}
```

Show the layout of the stack-frame for an invocation of f; Your answer should include the offset from the frame-pointer of each parameter and local variable. *15-points*

(a) The `n` parameters occupy `4 * n` bytes; the `m` local variables occupy `4 * m` bytes; the saved frame-pointer and return address occupy 4 bytes each. Hence the total is `4 * (n + m + 2)` bytes.

(b) The stack-frame will be laid out as follows:



2. Given the following program in a statically-scoped language which supports nested functions:

```
f(a, b) {                  //1
```

```
  var x = ...;              //2

  g(a, x) {                 //3
    var x = ...;            //4

    h(b) {                  //5
      var a = ...;          //6
      return a + b*x;       //refs to a, b, x.
    }

    //body of g()
    return b + h(a)*x;      //refs to a, b, x.
  }

  //body of f()
  return a*b + x;           //refs to a, b, x.
}
```

Local variable declarations are indicated using `var`. Points i where variables are defined are indicated using a comment `//`i.

For each line above which contains a comment `refs to a, b, x`, for each referenced variable show which declaration it refers to. *15-points*

Expression	`a` Declaration	`b` Declaration	`x` Declaration
`a + b*x`	`//6`	`//5`	`//4`
`b + h(a)*x`	`//3`	`//1`	`//4`
`a*b + x`	`//1`	`//1`	`//2`

3. Given the following program in a language which supports first-class functions as well as both lexically-scoped (indicated using a `lex` declaration) and dynamically-scoped variables (indicated using a `dyn` declaration) with `lambda` used to define anonymous functions.

```
//declare dynamically scoped var
dyn b = 3;

//Define function f with single parameter a
```

```
f(a) {
  lex static1 = a * 2;
  lex static2 = b;
  //return function which takes a single parameter x.
  return lambda (x) { return x + static1*static2 + b; }
}

//Define function f with single parameter a
sub h(a) {
  dyn b = a + 5
  return f(5)
}

//print result of calling return'd function from h(3) with
//actual parameter 5.
print h(3)(5);
```

What will be printed by the above program? Justify your answer by showing the values of all the variables. *15-points*

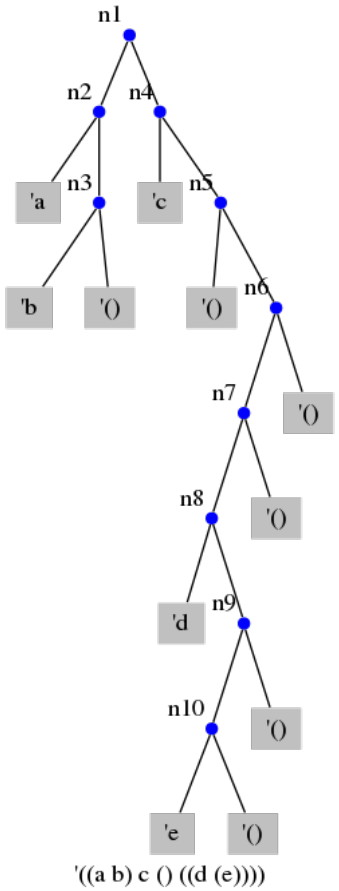
Annotating the program to show values of variables:

```
dyn b = 3;

f(a) {                          //f(5) called; hence a == 5
  lex static1 = a * 2;          //static1 = 5 * 2; hence static1 == 10
  lex static2 = b;              //static2 = value of b in caller h();
                                //hence static2 == 8
  //returns (lambda (x) { return x + 10*8 + b })
  return lambda (x) { return x + static1*static2 + b; }
}

sub h(a) {                      //h(3) called; hence a == 3
  dyn b = a + 5                 //b = 3 + 5; hence b ==  8
  return f(5)
}
```

```
//h() returns above lambda value, which is then called with parameter 5
//i.e. (lambda (x) { return x + 10*8 + b })(5)
//Since h() has returned, b reverts to 3.
//Hence the above lambda reduces to 5 + 10*8 + 3 == 88
print h(3)(5);                //88 is printed.
```

Note that Perl supports both lexical and dynamic variables as well as nested anonymous functions. The lex-dyn.pl program shows a translation of the above program into Perl syntax.

4. Consider the tree structure shown in the figure below (taken from the Scheme Slides).



By providing the Scheme expression equivalent to the subtree rooted at each internal node `n1` ... `n10`, show that the root node `n1` is equivalent to `'((a b) c () ((d (e))))`. *15-points*

Filling in the values of the internal nodes using a bottom-up traversal of the tree, we have:

```
n10 = '(e)
 n9 = '((e))
 n8 = '(d (e))
 n7 = '((d (e)))
 n6 = '(((d (e))))
 n5 = '( () ((d (e))))
 n4 = '( c () ((d (e))))
 n3 = '(b)
 n2 = '(a b)
 n1 = '( (a b) c () ((d (e))))
```

Hence the root is indeed equal to `'( (a b) c () ((d (e))))`.

5. Show that the `count-non-pairs` function discussed in the Scheme Slides will always terminate.

   (Termination is usually shown by identifying some quantity which cannot go below some value and showing that the quantity decreases as the computation progresses. Since the quantity cannot decrease below the minimum bound, the computation must terminate). *10-points*

   The `count-non-pairs` function code was given as:

```
(define (count-non-pairs ls)
  (if (not (pair? ls))
      1
      (+ (count-non-pairs (car ls))
    (count-non-pairs (cdr ls)))))
```

Termination can be shown by arguing that the depth of the argument decreases on each recursive call. Specifically, define depth(t) of a Scheme term t as follows:

- If t is not a pair, then depth(t) is 0.
- If t is a pair, then depth(t) is 1+max(depth(`car`(t))), depth(`cdr`(t))))

Hence for the first recursive call above, it is clear that the depth of the argument `(car ls)` is less than the depth of the incoming argument `ls`. Similary, for the second recursive call above, it is clear that the depth of the argument `(cdr ls)` is less than the depth of the incoming argument `ls`.

Since the size of the argument for each recursive call is strictly smaller than the size of the incoming argument, the function must terminate.

6. How would you build conceptually infinite lists in a language like Scheme.

   Specifically, if you are given

   `(next v)` A function which generates the next element in the list when given the value `v` of the previous element in the list.

   `init` A value representing the first element in the list.

   describe how you would build a data-structure which acts like an infinite list containing the elements generated by 0-or-more applications of the `next` function to `init`.

   For example, assume that the infinite list is constructed using the function `(inf-cons next init)` with parameters `next` (a function) and `init` (the initial value); `inf-car` and `inf-cdr` are accessor functions which return the head and tail of the constructed infinite list. Given these definitions, it should be possible to build and access an infinite list of natural numbers as follows:

```
> (define inf-natnums (inf-cons (lambda (x) (+ x 1)) 0))
> (inf-car inf-natnums)
0
> (inf-car (inf-cdr inf-natnums))
1
```

```
> (inf-car (inf-cdr (inf-cdr (inf-cdr inf-natnums))))
3
>
```

It is not required to show explicit code; it is sufficient to describe the essential idea. *15-points*

The essential idea is to squirrel away the value representing the current head of the list and the next function in a data-structure. Since 2 values need to be squirreled away, a Scheme pair is an obvious choice for the data-structure. Getting the head of the list would merely return the value; getting the tail of the list would use the next function to generate the next element and squirrel that value into a new copy of the data-structure.

This functionality could be implemented as follows (code is not required for this question):

```
(define (inf-cons next init) (cons init next))

(define (inf-car ls) (car ls))

(define (inf-cdr ls)
   (let ([next (cdr ls)])
        (cons (next (car ls)) next)))
```

Note that this idea of using a function to defer computation is not at all specific to Scheme and can be used in most popular languages. It is often a useful technique and can be used to solve problems with initialization loops or to build data-structures which appear cyclic while still being immutable.

7. Discuss the validity of the following statements:

   (a) Modules form a *closed scope*.

   (b) The *scope* of a variable is the same as it's *lifetime*.

   (c) It is possible to program without destructive assignment in any language which supports recursive functions.

(d) Scheme does not support destructive assignment.

(e) In Scheme, if for some expression x, `(list?` x`)` returns `#t`, then `(pair?` x`)` must also return `#t`. *15-points*

(a) Modules do indeed form a *closed scope* as declarations need to be explicitly imported or exported. Hence the statement is **true**.

(b) The *scope* of a variable is quite different from its *lifetime*. The former is a static concept, whereas the latter is a dynamic concept. Hence the statement is **false**.

(c) Loops depend on destructive assignment, but it is possible to replace all loops with recursive functions. Hence the statement is **true**.

(d) Scheme does support destructive assignment using the `set!` function. Hence the statement is **false**.

(e) `'()` is a list but not a pair. Hence `(list? '())` returns `#t` but `(pair? '())` returns `#f`. Hence the statement is **false**.