Fall 2013
Programming Languages
Homework 2

- Due on Friday, October 4, 2013 at 11:55 PM, Eastern Standard Time.
- This homework contains general questions and a Scheme programming assignment.
- The homework must be submitted through NYU Classes—do not send by email. No late submissions will be accepted. A zero homework score will be recorded if no homework is submitted by this time—**no exceptions**.
- The homework must be submitted entirely through NYU Classes—do not send by email. Late submissions will not be accepted for credit. Submissions using NYU Classes are a multi-step process. Make sure you complete the *entire* submission process in NYU Classes before leaving the page. It is recommended that you not wait until the last minute to submit, in case you encounter difficulties. Report all issues with submissions to the graders.
- While you are working on this assignment, you may consult the lecture material, class notes, textbook, discussion forums, books, and/or recitation materials for general information concerning the lambda calculus, Scheme, or any general topics related to the assignment. You may **under no circumstance** collaborate with other students or use materials from an outside source (i.e., people, books, Internet, etc.) in answering **specific** homework questions.
- You must use the Racket Scheme interpreter for the programming portion of the assignment. You may download and install Racket for free from the course page. Racket is available for most popular platforms. *Important*: Be sure to select "R5RS" from the language menu before beginning the assignment. You can save your Scheme code to an `.rkt` file by selecting *Save Definitions* from the File menu. Be sure to comment your code appropriately and submit the `.rkt` file.

Nested Subprograms [10] Consider the following Ada code fragment:

```
1  function MergeSort (lst : CollectionType) : Path is
2     left, right : CollectionType;
3     idx : IndexType;
4
5        procedure Partition (lst : in CollectionType, left : out CollectionType,
6                             right : out CollectionType) is
7           indexElem : IndexType;
8        begin
9           ....
10       end Partition;
11
12       procedure Sort (lst : in out CollectionType) is
13       begin
14          SortHelper(lst);
15       ...
16       end;
17
18       procedure SortHelper (lst : in out CollectionType) is
19       begin
20       ...
21       end;
22
23       function Merge (lst1, lst2 : CollectionType, elem : IndexType) : CollectionType is
24       begin
25          ....
26       end Merge;
27
28 begin
29   Partition(lst, left, right);
30
31   Sort(left);
32   Sort(right);
33
34   return Merge(left, right, indexElem);
35 end;
```

Please answer the following:

1. What would happen if we remove the first parameter to Partition (on lines 5 and 29)?
2. The syntax of the program above is well-formed, but the body of MergeSort contains a semantic error. Explain the error.
3. Including the subprogram that calls MergeSort, at most how many activation records could exist on the stack? What are they?

Parameter Passing [10] Trace the following code under both *call-by-name* and *call-by-value* semantics. For each, explain how you arrived at the value of `tmp` on each iteration of the loop and write the final result.

```
A[] = { 4, 2, 3, 5, 2 };
integer i = 0;
mystery(A[i]+i, A[i+1])

procedure mystery (a1, a2)
  integer tmp = 3;

    for c from 1 to 3 do
      tmp = tmp + a1 + a2;
      i++;
    end for;

end procedure;
```

Lambda Calculus [25] Identify the *free* and *bound* variables in each expression below:

1. $(\lambda x.\lambda y.\lambda z.zx(\lambda x.x))$
2. $(\lambda x.\lambda y.(\lambda z.zy)(\lambda x.zx))$
3. $(\lambda x.(y(\lambda y.xy)))(\lambda y.xy)$

Evaluate the following expressions. Show your work. If submitting a text file, write $\backslash y$ to denote λy:

1. `PLUS` $\ulcorner 1 \urcorner \ulcorner 2 \urcorner$
2. `IF FALSE` 1 0
3. `AND TRUE FALSE`

For each of the lambda expressions below, explain whether the expression can be legally β-reduced without any α-conversion at any step. For any expression below requiring an α-conversion, write the β-reduction twice: once after performing the α-conversion (the correct way) and once after not performing it (the incorrect way). (The difference in results should convince you that forgetting to do the *alpha*-conversion when necessary can lead to an incorrect β reduction.)

1. $(\lambda xy\,.\,yx)(\lambda x\,.\,x\,y)$
2. $(\lambda x\,.\,xz)(\lambda xz\,.\,x\,y)$
3. $(\lambda x\,.\,x\,y)(\lambda x\,.\,x)$

Bindings [10] Consider the following program:

```
program main;
  var a, b, c: integer;

  procedure sub1;
    var x, y, b: integer;
    begin {sub1}
    ...
    end; {sub1}

  procedure sub2;
    var x, b, t: integer;

    procedure sub3;
      var y, a: integer;
      begin {sub3}
      ...
      end; {sub3}
    begin {sub2}
    ...
    end; {sub2}

begin {main}
  ...
end {main}
```

Complete the following table listing all of the variables, along with the program units where they are declared, that are visible in the bodies of sub1, sub2, and sub3, assuming static scoping is used.

Unit	Var	Where Declared

`sub1`	`x, y, b`
	`a, c`
		`sub1`
		`main`

`sub2`		
`sub3`		

Parameter Passing [10] Consider the following:

```
void foo1( Foo x )
{
  for (int x=0; x < K; x++)
    std::cout << x << std::endl;
}

void foo2 ( Foo& x )
{
  for (int x=0; x < K; x++)
    std::cout << x << std::endl;
}
```

Assume the following:

- Function `foo1` uses pass-by-value semantics.
- Function `foo2` uses pass-by-reference semantics.
- Foo is 32 bytes large
- 1 unit of time is required to copy 8 bytes to the stack frame. (Assume zero time to perform any other stack frame initialization).
- A memory address is 64 bits (8 bytes)
- Accessing a local object on the stack by value consumes 1 unit of time
- Accessing a local object on the stack by reference requires 2 units of time

What is the performance of the two functions if $K = 3$? What about $K = 15$? Explain your results.

Scheme [35] Turn in your solution to all questions in this section in a single Scheme (.rkt) file, placing your prose answers in source code comments. Multi-line comments start with `#|` and end with `|#`.

In all parts of this section, implement iteration using recursion. Do NOT use the iterative `do` construction in Scheme. Do not use any function ending in "!" (e.g. `set!`).

Some helpful tips:

- Scheme library function `apply` takes a function and list as arguments and *applies* the contents of the list as actual parameters to the function. For example, `(apply + ('2 3))` evaluates the expression (+2 3).
- Scheme library function `list` turns an atom into a list.
- You might find it helpful to define separate "helper functions" for some of the solutions below.
- the conditions in "if" and in "cond" are considered to be satisfied if they are not `#f`. Thus `(if '(A B C) 4 5)` evaluates to 4. `(cond (1 4) (#t 5))` evaluates to 4. Even `(if '() 4 5)` evaluates to 4, as in Scheme the empty list `()` is not the same as the Boolean `#f`. (Other versions of LISP conflate these two.)

Please complete the following. You may not look at or use solutions from any source when completing these exercises. Plagiarism detection will be utilized for this portion of the assignment:

1. Write 2 unary functions (functions with 1 argument) of your choice which accept an integer input. For example, increment and decrement.
2. *Currying* is a method of expressing an n-argument function as a 1 argument function. Given a list of formal parameters $(a_1, \ldots, a_n)$, a curried function will accept argument a_1 and evaluate to another curried *function* which accepts a_2, and so on. In the λ-calculus, the curried version of multiplication would be defined as $\lambda x.(\lambda y.(\texttt{MUL}\ x\ y))$

   Write a function `mulcur`, which is the curried form of the Scheme binary multiplication operator, *.

   A note about curried functions: currying functions allows us to take advantage of a feature common to functional languages, called *partial application*. This is used to pass some (but not all) arguments to a curried function. Doing so evaluates to a curried function accepting the remaining parameters.

   For example we can use partial application to define a new unary function `mulby5` ("multiply by 5") as follows:

   ```
   (define mulby5 (mulcur 5))
   ```

   Now you can invoke this as you would any unary function:

   ```
   (mulby5 2)
   > 10
   (mulby5 5)
   > 25
   ```

3. Write a function `compose` which is defined to perform the same function as in slide 29 of the Subprogram lecture. That is, it should return a *function* that, when invoked and supplied an argument, will execute the function composition with the argument as input. For example, `(compose inc inc)` should evaluate to `#<procedure>`, whereas `(compose inc inc) 5` should evaluate to 7.
4. In your own words, explain specifically why Ada cannot handle the program on slide 29, but Scheme can.
5. A well-known function among the functional languages is `map`. This function accepts a unary function f and list $l_1, \ldots, l_n$ as inputs and evaluates to a new list $f(l_1), \ldots, f(l_n)$. Write a similar function `map2` which accepts a list $j_1, \ldots, j_n$, another list $\ell_1, \ldots, \ell_n$ (note they are of equal length), a unary predicate p and a unary function f. It should evaluate to an n element list which, for all $1 \leq i \leq n$, yields $f(\ell_i)$ if $p(j_i)$ holds, or ℓ_i otherwise. Example:

```
(map2 '(1 2 3 4) '(2 3 4 5) (lambda (x) (> x 2)) inc)
```

should yield: `(2 3 5 6)`. Additionally, your solution should evaluate to an error message (i.e., a string) if the two lists are not of the same size.

6. Write a function `skip` which behaves as follows:

```
> ((skip 0) 'foo)
foo
> (((skip 1) 'foo) 'bar)
bar
> ((((skip 2) 'foo) 'bar) 'baz)
baz
```

   Your solution doesn't have to handle other cases (e.g. `((((skip 1) 'foo) 'bar) 'baz)`).

7. Consider the following Scheme code:

```
(lambda ()
   (if (a)
    (let ((x (g)))
  x)
(and (b) (a))))
```

   Is the above code tail recursive? Explain. If the outcome depends on any assumptions, state the assumptions.

8. **5 Point Bonus!** Now consider the following:

```
(lambda (b) (if a (x b) #f))
```

   Is the above code tail recursive? Explain. If the outcome depends on any assumptions, state the assumptions.