Fall 2017
Programming Languages
Homework 2

- Due on Monday, October 16, 2017 at 5:00 PM, Eastern time.
- The homework must be submitted through NYU Classes—do not send by email. Due to the timing of the midterm exam, late submissions will not be accepted for credit.
- I **strongly recommended** that you submit your solutions well in advance of the deadline, in case you have issues using the system or are unfamiliar with NYU Classes. Be very careful while submitting to ensure that you follow all required steps.
- Do not collaborate with any person for the purposes of answering homework questions.
- Use the Racket Scheme interpreter for the programming portion of the assignment. *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.
- When you're ready to submit your homework upload a single file, `hw2-<netID>.zip`, to NYU classes. The .zip archive should contain two files: `hw2-<netID>.pdf` containing solutions to the first four questions, and `hw2-<netID>.rkt` containing solutions to the Scheme programming question. Make sure that running your .rkt file in the DrRacket interpreter does not cause any errors. *Non-compiling programs will not be graded.*

1. [30 points] **Bindings and Nested Subprograms**

   Consider an imperative programming language which attempts to permit an object of a dynamic size to be returned from a subprogram. That is, an object whose size cannot be computed at compile time. In this question we explore how such a feature might be implemented in the presence of various assumptions regarding the language.

   We use the term *object* to refer to some previously allocated memory block which is bound to a variable. In general, an object *may* reference other objects, such as through a data member in the case of an object-oriented program. Bear this in mind while answering the questions. We say an object is *local* to a subprogram if it is allocated within the activation record of the subprogram. For the purposes of this question, we require that once an object is returned, the lifetime of the returned object must survive at least as long as any references to the object exist. For this question, we do not care about the details of deallocation or whether any object is ever deallocated.

   For each set of assumptions provided below, explain at a high level how you might implement this feature. Do not write code. An explanation of your general approach is sufficient. In answering the question, you may conclude that it is not possible to provide any such implementation, in either certain cases or in any case. If so, explain your reasoning. You are making an informal argument—proofs are not necessary.

   Your answers may differ depending on certain assumptions about the language. If you make any such assumptions, state what they are. You will not be graded on the basis of any preconceived "correct answer," but rather based on your demonstration of knowledge regarding this topic and the proper use of reasoning. It is not necessary that your answer be perfect, but it must be logical and demonstrate knowledge of the subject matter. Strive to make your answer concise and use correct terminology. Remember that you cannot earn full credit if we cannot understand what you wrote.

   1. Assume that for all subprograms defined in a given input program, the object returned from a subprogram is always nonlocal to the subprogram. (Remember, "nonlocal" tells you *where* in the program the object is declared, but not where it is stored [heap, stack, etc.] You may need to break your answer into cases).
   2. Now assume the unrestricted case where the object returned may or may not be local to the subprogram. Further assume that no other objects are referenceable from the returned object.
   3. Thinking about question 2 above, now assume that the returned object *can* reference other objects. Does this change your answer above? Explain.
   4. Thinking about question 2, now assume that instead of an object being returned, a subprogram is returned. Is there any parallel between returning subprograms versus returning objects containing other referenceable objects? Explain.
   5. Pick your favorite language. How does that language deal with the issue of returning dynamic objects from subprograms? How about returning subprograms from subprograms? Are these features supported? If so, how?

2. [10 points] **Nested Procedures and Runtime Bindings**

   Consider the following pseudo-code:

```
procedure outer ( ) is
  b : integer = 3;

    procedure inner ( c : integer ) is
       d : boolean = False;

       procedure innermost ( e : integer ) is
          b: real = 3.14;
          f: integer = -50;
       begin
          f = f + 10;

          if e == 0 then
              print b, c, d, f;
          else
              innermost(e-1);
          end if;
       end;

    begin
       innermost(c);
    end;

begin
   inner(b);
end;
```

   Please answer the following:

   a. Draw the runtime stack (i.e., for each activation record, the name of the procedure and its local variable bindings) as it will exist when the base case (`e == 0`) is reached, upon invoking function `outer()`. For the purposes of drawing the stack, assume that a function `main` (not shown) calls `outer()`.

   b. On the runtime stack you have drawn above, for each variable `b`, `c`, `d` and `f` referenced by the `print` statement within the base case above, draw an arrow indicating where each variable's value is located in the runtime stack. In other words, draw a display and show where it would point at the moment that the `innermost` base case is reached.

3. [10 points] **Parameter Passing**

   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[] = { 1, 3, 2, 7 };
integer i = 0;
mystery(i, A[i+1])

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

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

end procedure;
```

4. [25 points] **Lambda Calculus**

a. Let S and K stand for the following lambda expressions:

$$\text{S} \ = \lambda x.\lambda y.\lambda z.(xz(yz))$$

and

$$\text{K} \ = \lambda x.\lambda y.x$$

The composition SKK is a lambda expression again:

$$\text{SKK} \ = (\lambda x.\lambda y.\lambda z.(xz(yz)))\text{K K}$$

Apply a sequence of beta reductions to show that SKK is equivalent to the identity function:

$$\text{SKK} \ = \lambda z.z$$

b. Reduce the following expression in two different ways. Give a brief explanation of your reduction steps.

$$(\lambda x. * \ x \ x)(+ \ 2 \ 3)$$

c. Consider the following lambda expressions. For each of the above, 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). Do the two methods reduce to the same expression?

i. $(\lambda xy \,.\, yx)(\lambda x \,.\, x\, y)$

ii. $(\lambda x \,.\, xz)(\lambda xz \,.\, x\, y)$

iii. $(\lambda x \,.\, x\, y)(\lambda x \,.\, x)$

d. In the lecture slides, we discussed lambda expressions representing the function `PLUS` and the numerals $\ulcorner 0 \urcorner$, $\ulcorner 1 \urcorner$, ...
Reduce the lambda expression `PLUS` $\ulcorner 1 \urcorner$ $\ulcorner 1 \urcorner$ and show that it reduces to $\ulcorner 2 \urcorner$.

If the tools you are using to submit your solution supports the λ character, please use it in your solution. If not, you may write \lam as a substitute for λ.

5. [25 points] **Scheme**

In all parts of this assignment, 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 `append` appends two lists.
- 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:

a. The function `foldr` ("fold right") accepts the following parameters: a binary function f, a *seed value* s and a list $L = l_1, \ldots, l_n$. The algorithm works by first evaluating $x_1 = f(l_n, s)$, then $x_2 = f(l_{n-1}, x_1), \ldots$. For example, a call to `foldr + 0 '(1 2 3)` evaluates to `(1 + (2 + (3 + 0)))`. If the list is empty, `foldr` evaluates to s. Write the function for `foldr` in Scheme.

b. Write a function `paramreverse` that takes a function F and a list of arguments AL and calls function F, passing the arguments to AL in reverse order. Example: `(paramreverse - '(2 1))` should evaluate to -1.

c. Write a function `(highest L k)` which takes a list of integers and an integer $k > 0$ as arguments and returns a new list containing the k highest numbers in the original list. The integers in the returned list do not have to appear in any particular order. Example: `(highest '(1 7 4 5 3) 2)` should evaluate to `(5 7)`.

   Hint: break the problem into subproblems and write a separate function for each subproblem. Use the `let` *clauses.*

d. Write a function `(mapfun FL L)` that takes a list of functions `FL` and a list `L` and applies each element of `FL` to the corresponding element of `L`. For instance,

   ```
   (mapfun (list cadr car cdr) '((A B) (C D) (E F)))
   ```

   should return `(B C (F))` since `(cadr '(A B)) => B, (car '(C D)) => C`, and `(cdr '(E F)) => (F)`.

   If `FL` and `L` are of different lengths, it should stop whenever it reaches the end of either one; e.g.

   ```
   (mapfun (list cadr car) '((A B) (C D) (E F)))
   ```

   and

   ```
   (mapfun (list cadr car cdr) '((A B) (C D)))
   ```

   should both return (B C).

e. Write a function `(filter pred L)`, where `pred` is a predicate and `L` is a list of atoms. Function `filter` outputs a list such that an item i in L will appear in the output list if `pred(i)` is true. For instance, `(filter even? '(1 2 3 4))` should return `(2 4)`. Write the function `filter` in Scheme.