Spring 2021
Programming Languages
Homework 2

- Due on Monday, March 1, 2021 at 11:55 PM, Eastern Daylight time.
- The homework must be submitted through NYU Classes—do not send by email. Due to timing considerations, late submissions will not be accepted after the deadline above. **No exceptions will be made.**
- I **strongly recommend** 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. [25 points] **Activation Records and Lifetimes**

   1. In class, we discussed an implementation issue in C relating to the `printf` function. Recall that reversing the order of the arguments was the solution to the problem of not being able to access the format string using a constant frame pointer offset. Now assume that reversing the arguments is not an option and that some other method has to be devised to solve this problem instead. Formulate a different solution and explain how it works in a conversational level of detail. The solution cannot involve a fixed number of arguments, or a fixed size of arguments.
   2. In Ada 83, "out" parameters could be written but not read. In Ada 95, this rule was modified so that "out" parameters could be both read and written, but the parameter begins uninitialized. If the parameter is never written, the caller effectively sees an uninitialized variable. The language C# is similar, but requires the variable to be initialized before returning and it cannot be read by the method before it is first written. Why do you suppose these restrictions are in place? Explain.
   3. In C++, a *destructor* in object-oriented programming is a special method belonging to a class which is responsible for cleaning up resources. Destructors cannot be called explicitly. Rather, they are called automatically by the language runtime. Thinking back to the lecture slides on activation records, where do you suppose the destructor call is triggered by the language? (No special knowledge of C++ is assumed for this question). There are two cases to consider: heap-allocated objects (using the **new** and **delete** operators) and stack-allocated objects which are locally declared inside a function. How might the destructor be called in both these cases?
   4. Consider the following C code:

```
{  int a, b, c;
   ...
   {   int d, e;
   ...
       {  int f;
       ...
       }
     ...
   }
   ...
   { int g, h, i;
     ...
   }
   ...
 }
```

      Assuming that each variable occupies four bytes, how much total space is required for the variables in this code? Provide two answers: one assuming no space optimizations and a second answer including space optimizations. Explain the space optimizations.
   5. Consider the following pseudocode:

```
procedure P (A,B : real)
  X : real

     procedure Q (B,C : real)
     Y : real
     ...

     procedure R (A,C : real)
     Z : real
     ...  **
```

. . .

What is the referencing environment of the line marked with an asterisk? That is, what names are visible from the specified location? (Hint: names include not just variable names, but procedure names as well.)

2. [15 points] **Nested Subprograms**

Consider the following pseudo-code:

```
procedure outer () is
a : int = 7
n : int = 2
b : int = 1

    procedure inner (int n) is
    a : integer = 3

        procedure innermost (int n) is
        begin
           if n == 1 then
              print(b)
           else
              b := a + b
              innermost(n-1)
           end if;
        end;

    begin
      innermost(a)
    end;

begin

   for j from 1 to n do
      inner(b);
   end for

end;
```

Please answer the following:

a. Write the name and actual parameter value of every subprogram that is called, in the order in which each is activated, starting with a call to `outer`. Assume static scoping rules apply.

Example: outer, inner(2), …

b. While the program above is running, which variables hold values that never change (e.g. are never assigned in the scope in which they exist)? Don't forget to consider formal parameters. Identify the scope of the variables to be clear about which declaration you are referring to. For example: outer.a, outer.b, inner.n, inner.a, etc.

c. Assume now that dynamic scoping rules are in effect. Does this change the behavior of the program above? Explain why or why not.

d. Draw the runtime stack as it will exist when the base case (`n == 1`) is reached for the first time, after first invoking function `outer()`. Assume that `outer` is called from function `main` (not shown above). Your drawing must contain the following details:

- Write the activation records in the proper order. The position of *outer* in your stack will imply the stack orientation, so no need to specify that separately.
- Each activation record must show the name of the procedure and its local variable bindings.

- Assume static linkages are used and draw them.

e. According to the static scoping rules we've learned, can `innermost` invoke `outer`? Can `outer` invoke `innermost`?

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

   1. Trace the following code assuming all parameters are passed using *call-by-name* semantics. Evaluate each formal parameter and show its value after each loop iteration (as if each one was referenced at the bottom of the loop.)

      Example:
      After iteration 1: a1 = ?, a2 = ?, a3 = ?, a4 = ?
      After iteration 2: a1 = ?, . . .

   2. Perform the same trace as above, where a1, a4 are passed by *call-by-name* and a2, a3, are passed by *call-by-need* semantics.

   3. Perform the same trace as above, where all arguments are passed by *call-by-value*.

```
var i=0, j=1;

mystery(i, i+1, i*3,j)

procedure mystery (a1, a2, a3, a4)

    for count from 1 to 3 do   // 1 to 3 inclusive
      a1 = a2 + a3 + a4;
      a4 = a4 + 1;
    end for;

end procedure;
```

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

These questions are to be completed after the February 22 lecture.

This first set of problems will require you to correctly interpret the precedence and associativity rules for Lambda calculus and also properly identify free and bound variables. For each of the following expressions, rewrite the expression using parentheses to make the structure of the expression explicit (make sure it is equivalent to the original expression). Remember the "application over abstraction" precedence rule together with the left-associativity of application and right-associativity of abstraction. Make sure your solution covers both precedence *and* associativity.

Now, the expressions:

a. $(\lambda x.x)\ y\ z$

Example: $(((\lambda x.x)\ y)\ z)$ (Only associativity is necessary in this example since parentheses are already present to force abstraction)

b. $\lambda x\ .\ \lambda y\ .\ y\ x$

c. $(\lambda x.x)\ \lambda x.x\ (\lambda y.y)\ z$

d. $(\lambda y.\lambda z.f\ \lambda x.z\ y)\ p\ x$

e. $\lambda z.((\lambda s.s\ q)\ (\lambda q.q\ z))\ \lambda z.z\ z$

Circle all of the free variables (if any) for each of the following lambda expressions:

a. $\lambda x\ .\ z\ x\ \lambda z\ .\ y\ z$

b. $(\lambda x\ .\ x\ y)\ \lambda z\ .\ w\ \lambda w\ .\ w\ x\ z$

c. $x\ \lambda z\ .\ x\ \lambda w\ .\ w\ z\ y$

d. $\lambda x\ .\ x\ y\ \lambda x\ .\ y\ x$

e. $(\lambda x\ .\ x)\ x\ y$

This next set of questions is intended to help you understand more fully what α-conversions are needed for: namely, to avoid having a free variable in an actual parameter captured by a formal parameter of the same name. This would result in a different (incorrect) solution. Remember that when performing an α-conversion, we always change the name of the formal parameter—never the free variable. Consider the following lambda expressions. For each of the expressions below, state whether the expression can be legally β-reduced without any α-conversion at any of the steps, according to the rule we learned in class. For any expression below requiring an α-conversion, perform 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?

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

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

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

For each of the expressions below, β-reduce each to normal form (provided a normal form exists) using applicative order reduction. For each, perform α conversions where required. For clarity, please show each step individually—do not combine multiple reductions on a single line.

a. $(\lambda x\ .\ xy)((\lambda y\ .\ z\ y)\ x)$

b. $(\lambda x\ .\ x\ x\ x)\ (\lambda x\ .\ x\ x\ x)$

c. $(\lambda z\ .\ z)(\lambda z\ .\ z\ z)(\lambda z\ .\ z\ q)$

d. `MULT` $\ulcorner 2 \urcorner\ \ulcorner 3 \urcorner$

e. `AND TRUE FALSE`

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**

These questions are to be completed after the February 22 lecture.

For the questions below, turn in your solutions 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 features such as `set`, `while`, `display`, `begin`, etc. Do not use any function ending in "!" (e.g. `set!`). These are imperative features which are not permitted in this assignment. Use only the functional subset discussed in class and in the lecture slides. Do not use Scheme library functions in your solutions, except those noted below.

Some helpful tips:

- 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. Consider using one of the `let` forms for these.
- 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.)
- You may call any functions defined in the Scheme lecture slides in your solutions. (For that reason, you may obviously include the source code for those functions in your solution without any need to cite the source.)
- You may not look at or use solutions from any other source when completing these exercises. Plagiarism detection will be utilized for this portion of the assignment. **DO NOT PLAGIARIZE YOUR SOLUTION.**

Please complete the following:

1. Provide an answer to Exercise 3.13 in the Scott textbook.
2. Write a function `findremove` which takes a list of numbers and a value and returns the same list with all occurrences of the specified value removed, if it exists.

```
>(findremove '(1 20 38 5) 0)
(1 20 38 5)

>(findremove '(1 20 38 5) 20)
(1 38 5)

>(findremove '(1 20 38 5) 5)
(1 20 38)

>(findremove '() 5)
()

>(findremove '(1 2 2 3 3) 2)
(1 3 3)
```

3. Implement a function `zip` which takes an arbitrary number of lists as input and returns a list of those lists. (Hint: this is much simpler than you think.)

```
>(zip '(1 2 3) '(2 3 5))
((1 2 3) (2 3 5))

>(zip '(1 2 3) '(2 3 5) '(5 6 7))
((1 2 3) (2 3 5) (5 6 7))
```

4. Implement a function `unzip` which given a list and a number n, returns the nth item in the list. Return the empty list if the index is out of range.

```
>(unzip '( (1 2 3) (5 6 7) (5 9 2 )) 1)
(5 6 7)

>(unzip '( (1 2 3) (5 6 7) (5 9 2 )) 0)
(1 2 3)
```

5. Implement a function `cancellist` which given two lists, will remove from *both* lists all occurrences of numbers appearing in both.

```
>(cancellist '() '())
(() ())

>(cancellist '(1 3) '(2 4))
((1 3) (2 4))

>(cancellist '(1 2) '(2 4))
((1) (4))

>(cancellist '(1 2 3) '(1 2 2 3 4))
(() (4))
```

6. 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.

7. 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.

8. Write a function `getitem` which given a list and a zero-based index, retreives the item from the list at that index. Should return `#f` if the index is out of bounds.

```
>(getitem '(1 2 3 4 5) 2)
3

>(getitem '(1 2 3 4 5) 0)
1

>(getitem '(1 2 3 4 5) 6)
#f
```

9. Exercise 11.6 in Scott

10. Exercise 3.12 in Scott