Spring 2023
Programming Languages
Homework 2

- Due on Friday, March 3, 2023 at 11:59 PM, Eastern Time (ET).
- The homework must be submitted through NYU BrightSpace—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 BrightSpace. 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 BrightSpace. 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.

      To be honest, I'm not sure if this would work, but I have two ideas: the first is that the prinf function could write instructions that access the frame pointer to the frame below it (the previous activation frame), and traverse upwards to collect arguments for the printf function.

      The second idea: have the printf function does not store any arguments to the stack, but rather a location in the heap, and then automatically pass two references to the stack, a pointer to an dynamic array of variables of different types, (strings, ints, etc), and one for the string literal that you want to augment.

   2. Consider a programming language "NoRec." In this language, programmers cannot write programs with recursion (i.e., procedures cannot call themselves, directly or indirectly through other procedures).

      One runtime optimization possible here is based on the observation that, without recursion, at any point of execution, each method can be active at most once. Hence, at any given moment, there is at most one active activation record for each procedure. This observation simplifies the implementation of activation records: because there is at most one copy of local variables and actual arguments for each procedure, they need not be stored on the stack; instead, they can be stored in static (global) memory at a fixed location. The benefit of this optimization is that there is less pushing and popping from the stack.

      (a) With these optimizations in place, for each of the following, explain (with reason) whether it is still needed or no longer needed in an activation record:

      - frame pointer

        Frame pointers would no longer be needed. Frame pointers main task is to provide a dynamic pointer to a known location within an activation record. Local variables within an activation record, even if dynamically initialized, will be located at a fixed offset from the frame pointer, something that can be determined "statically", i.e. by the compiler before run-time. Thus, in recursive-stack based languages, frame pointers are a necessity to point to the correct local variables, that may or may not share the same name as another variable. Since non-recursive languages don't have this issue with local variables, the concept of a frame pointer is unecessary.

      - return address

        A return address would still be necessary in an activation record, as its essentially just an instruction to help return the value generated by a function at the end of an activation record. In our NoRec language, the return address would just serve as a pointer to the statically stored location of whatever variable we save the value into.

      (b) How can this *no recursion* behavior be enforced, i.e., what runtime and/or compile time checks are needed to ensure that there is *no recursion*?

   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?

For heap-allocated objects, most languages run times have a "garbage collector" that will check a "reference count" i.e. how many variables on the stack or on the heap point to said object, and in the event an objects reference count is 0, a desctrutor is implicitly called and the memory is freed up. Of course, there is the case where a programmer writes a line of code explicitly deleting an object themselves by calling the destructor i.e. "delete object".

In the case of stack-allocated objects, last homework we learned that there are certain procedures put in place by language run-times that tear-down objects (calling their destructors), when stack frames are popped off the stack. This situation occurs if the object and all other sub-objects contained within are created on the stack frame about to be popped. Before popping off the stack frame, the destructors are called, the objects unwound, and the memory freed up responsibly.

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, what is the minimum space required for the variables in this code? Explain your answer briefly.
`Hint:` Think about the possible space optimizations.

Assuming static scoping rules, and a stack-allocated memory scheme, we should consider the maximum amount of variables that are "visible" at any given time. We start by looking at the nested scopes, as variables a,b, and c are present through the body of the program given in psuedocode, they will be visible for the whole duration of the program. In the first nested scope within the main scope, we define d,e and f. That's 6 variables in total. We then exit that nested scope block and enter a new one (thus losing three visible variables). As we enter the new scope, we initialize 3 more variables, g,h and i, thus our maximum amount of variables visible at any given time is 6. With each variable taking up 4 bytes, at minimum we would need 6x4 bytes worth of space, or 24 bytes.

5. Consider the following pseudo code:

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

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

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

    ...
end
```

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.
`Note:` Procedures Q and R are nested directly inside procedure P.

Assuming static binding: in procedure R (where the asterisks are), the following variable names are visible: Z, X, A, B, and C. Three procedure names are visible: P,Q and R.

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

Consider the following pseudo-code:

```
procedure MAIN;
    var X : integer = 3;

    procedure BIGSUB;
        var A : integer = 2;
        var B : integer = 3;
        var C : integer = 4;

        procedure SUB1;
            var A : integer = 5;
            var D : integer = 1

        begin {SUB1}
            A := B + C;     <---------------- (1)
            print(A, B, C);
        end; {SUB1}

        procedure SUB2(X : integer);
            var B : integer = 1;
            var E : integer = 8;
            procedure SUB3;
                var C: integer = 9;

            begin {SUB3}
                SUB1;
                E := B + C;
                print(E);
            end; {SUB3}

        begin {SUB2}
            SUB3;
            A := E;
        end; {SUB2}

    begin {BIGSUB}
        SUB2(7);
    end; {BIGSUB}
begin
    BIGSUB;
end; {MAIN}
```

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 `MAIN`. Assume static scoping rules apply.

MAIN (Initializes X=3) → BIGSUB (Initializes A=2, B=3,C=4) → SUB2 (with 7 being passed as the actual parameter to the formal parameter X, B=1, E=8 initialized) → SUB3 (C=3 initialized) → SUB1 (A=5, D=1 initialized).

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.

Changed variable values: SUB1.A, SUB2.E, BIGSUB.A

Unchanged Variables: MAIN.X BIGSUB.B, BIGSUB.C SUB2.X, SUB2.B SUB3.C SUB1.D

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

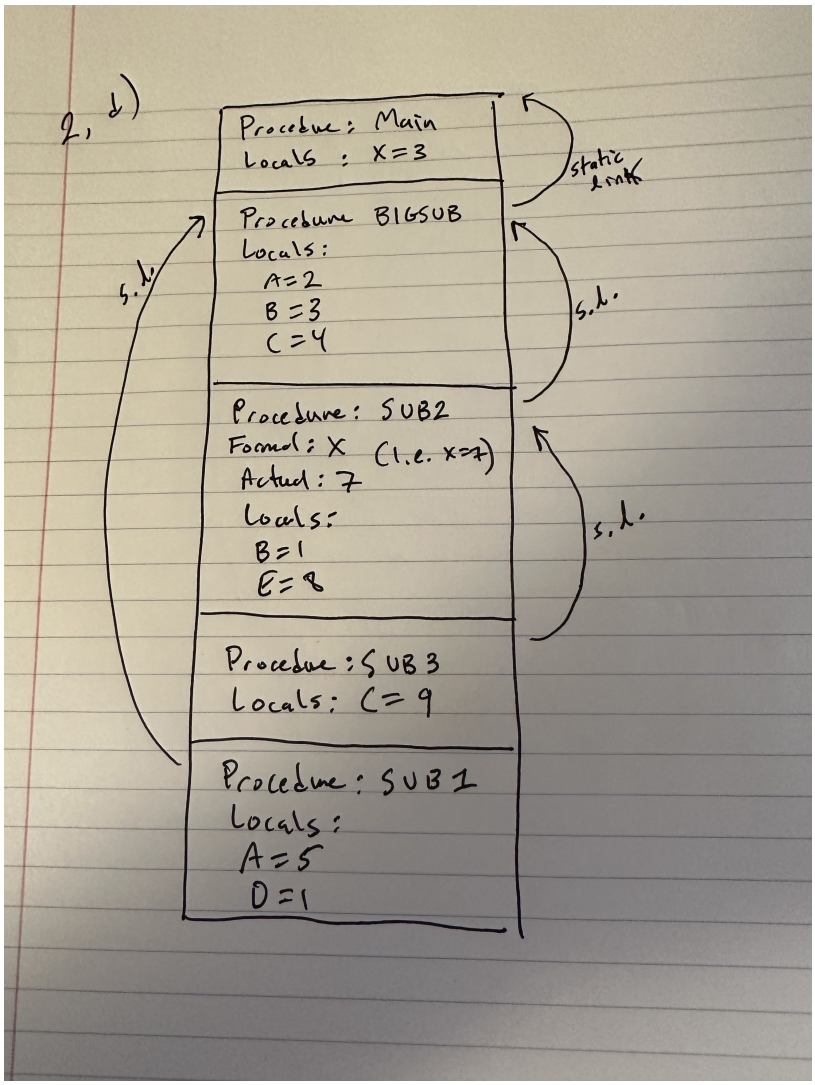
Dynamic scoping rules would certainly change the behavior of the program. For static scoping, our compiler knows to write instructions to look for the most recent declaration of a variable in the current activation frame, and then begin searched down the links of the static chain. In this form, the state of variables defined in different scopes is fairly predictable.

In dynamic scoping, the most recent binding is chosen at run time, which yields un-intuitive program behavior, especially during conditional control flow at run time.

In our program, behavior would change, for instance when we get to the line marked (1) in the diagram, where $A := B + C$, the C being referred to is no longer the one defined in BIGSUB, but rather the most recent C variable which is defined in the beginning of the SUB3 procedure. This changes the value assigned to A, and what is output by the print statement in that code block.

d. Draw the runtime stack as it will exist when execution finishes the line marked `(1)`, after first invoking procedure `MAIN`. Your drawing must contain the following details:

- Write the activation records in the proper order. The position of *MAIN* 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 `MAIN` invoke `SUB3`? Give brief explanation for your answer. Can `SUB3` invoke `MAIN`?

MAIN would not be able to invoke SUB3 as it's hidden within multiple layers of inner nested scopes. MAIN would only be able to invoke SUB3 if it was declared within the same scope, or in an outer scope. In our example, MAIN can only see functions defined within the same scope, or an outer scope. However, within MAIN's scope, only BIGSUB is defined, and there is no outer scope shown in the psuedocode for MAIN to evoke any other functions. Therefore, in MAIN, only BIGSUB could be invoked.

SUB3, however, would be able to invoke MAIN as MAIN is defined in an outer scope. This would lead to recursive behavior. SUB3 can invoke SUB2, SUB1, BIGSUB, and MAIN. You can see this in the picture as SUB3 has static links that can reach MAIN.

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.)

      After Iteration 1, a1=5, a2=2, a3=3, a4=1
      After Iteration 2, a1=22, a2=6, a3=15, a4=2
      After Iteration 3, a1=91, a2=23, a3=66, a4=3

   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

      After Iteration 1, a1=5, a2=2, a3=3, a4=1
      After Iteration 2, a1=6, a2=2, a3=3, a4=2
      After Iteration 3, a1=7, a2=2, a3=3, a4=3

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

      Call by value copies the values of the actual parameters into the formal, making the trace straight forward to perform:

      Before Iteration 0: a1 = 1, a2= 2, a3=3, a4=0
      After Iteration 1, a1=5, a2=2, a3=3, a4=1
      After Iteration 2, a1=6, a2=2, a3=3, a4=2
      After Iteration 3, a1=7, a2=2, a3=3, a4=3

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

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**

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

   b. $\lambda x\ .\ \lambda y\ .\ \lambda z\ .\ z\ y\ x$
      Answer: $(\lambda x.(\lambda y.(\lambda z.(z\ (y\ x)))))$

   c. $\lambda x\ .\ x\ y\ \lambda z\ .\ w\ \lambda w\ .\ w\ x\ z$
      Answer: $(\lambda x.(x\ (y\ (\lambda z.(w\ (\lambda w.(w(x\ z))))))))$

   d. $x\ \lambda z\ .\ x\ \lambda w\ .\ w\ z\ y$
      Answer: $(x\ \lambda z.(x\ \lambda w.(w\ (z\ y))))$

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

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

   a. $\lambda z\ .\ z\ x\ \lambda y\ .\ y\ z$
      Answer: only x is free

   b. $(\lambda x.x)\ (\lambda x.x\ (\lambda y.y))\ z$
      Answer: Only the z is free at the end

   c. $\lambda p.(\lambda z.f\ \lambda x.z\ y)\ p\ x$
      Answer: The variables in the parenteses f and y are free variables, as well as the p and x on the end after the parantheses.

   d. $\lambda x\ .\ x\ y\ \lambda x\ .\ y\ x$
      Answer: All of the y's are free

   e. $\lambda x\ .\ x\ (x\ y)$
      Answer: only the y is free

3. This next set of questions is intended to help you understand more fully why α-conversions are needed: 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. $(\lambda xy\,.\,z\,x)(\lambda x\,.\,x\,y)$

   (Incorrect) β-reduction without α-reduction:

   $$(\lambda xy\,.\,z\,x)[x \to (\lambda x\,.\,x\,y)] \underset{\beta}{\to} \lambda y.(z\ \lambda x.(x\ y))$$

   Beta reduction after changing the y to a w in the first expression $(\lambda xy\,.\,z\,x)[y/w] \underset{\alpha}{\to} (\lambda xw\,.\,z\,x)$
   Now the β-reduction:

   $$(\lambda xw\,.\,z\,x)[x \to (\lambda x\,.\,x\,y)] \underset{\beta}{\to} \lambda w.(z\ \lambda x.(x\ y))$$

b. $(\lambda x\,.\,\lambda yz\,.\,x\,y\,z)(\lambda z\,.\,z\,x)$

First we note that $(\lambda x\,.\,\lambda yz\,.\,x\,y\,z) = \lambda xyz\,.\,x\,y\,z$

(Incorrect) With no α-reduction:

$$(\lambda xyz\,.\,x\,y\,z)[x \to (\lambda z\,.\,z\,x)] \underset{\beta}{\to} \lambda yz.((\lambda z.zx)yz)$$

With 2 α-reductions, $x \to w$ for the left expression,

$$\lambda xyz\,.\,x\,y\,z[x/w] \underset{\alpha}{\to} \lambda wyz\,.\,w\,y\,z$$

And $z \to a$ for the right expression

$$(\lambda z\,.\,z\,x)[z/a] \underset{\alpha}{\to} (\lambda a\,.\,a\,x)$$

The correct resulting β-reduction:

$$(\lambda wyz\,.\,w\,y\,z)[w \to (\lambda a\,.\,a\,x)] \underset{\beta}{\to} \lambda yz\,.\,((\lambda a\,.\,a\,x)\,y\,z)$$

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

(Incorrect) Without α-reduction:

$$(\lambda x\,.\,x\,z)[x \to (\lambda xz\,.\,x\,y)] \underset{\beta}{\to} (\lambda xz\,.\,x\,y)z$$

with α-reduction for the formal parameter z in the second expression

$$(\lambda xz\,.\,x\,y)[z/a] \underset{\alpha}{\to} (\lambda xa\,.\,x\,y)$$

Then we beta reduce:

$$(\lambda x\,.\,x\,z)[x \to (\lambda xa\,.\,x\,y)] \underset{\beta}{\to} (\lambda xa\,.\,x\,y)z$$

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

Alpha reduction is not necessary here: we cannot change the name of any formal parameters to avoid confusion between formal and free parameters.
Beta reduction:

$$[x \to \lambda x\,.\,x\ y] \underset{\beta}{\to} (\lambda x\,.\,y)\ y$$

`Note:` All the variables are single letters {x, y, z}, i.e, expression $(\lambda x\,.\,\lambda yz\,.\,x\,y\,z)$ is equivalent to $(\lambda x\,.\,\lambda y\,.\,\lambda z\,.\,(x\,y\,z))$.

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

$$\begin{aligned}((\lambda y\ .\ z\ y)\ x)(\lambda x\ .\ x\ y) &\to (\lambda y\,.\,z\,y)[y \to x] \underset{\beta}{\to} (z\,x) \\ &= (z\,x)(\lambda x\,.\,x\,y)\end{aligned} \tag{1}$$

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

This cannot be β reduced into a normal form as it the expression grows each time you try to β-reduce.

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

$$\begin{aligned}(\lambda x\ .\ x)(\lambda y\ .\ x\ y)(\lambda z\ .\ x\ y\ z) &\to ((\lambda x\ .\ x))[x \to (\lambda y\ .\ x\ y)] \underset{\beta}{\to} (\lambda y\ .\ x\ y) \\ &= (\lambda y\ .\ x\ y)(\lambda z\ .\ x\ y\ z) \\ (\lambda y\ .\ x\ y)[y \to (\lambda z\ .\ x\ y\ z)] &\underset{\beta}{\to} x(\lambda z\ .\ x\ y\ z) \\ &= x(\lambda z\ .\ x\ y\ z)\end{aligned} \tag{2}$$

d. `MULT` $\ulcorner 0 \urcorner \ulcorner 3 \urcorner$

MULT $= \lambda mnf.(m\ (n\ f))$
$\ulcorner 0 \urcorner = \lambda fx.x$
$\ulcorner 3 \urcorner = \lambda fx.f(f(fx))$

$$\begin{aligned}\text{MULT}\ulcorner 0 \urcorner \ulcorner 3 \urcorner &= (\lambda mnf.(m\ (n\ f)))\,(\lambda fx.x)(\lambda fx.f(f(fx))) \\ (\lambda mnf.(m\ (n\ f)))\,[m \to (\lambda fx.x)] &\underset{\beta}{\to} (\lambda nf\,.\,((\lambda fx.x)\ (n\ f))) \\ &= (\lambda nf\,.\,((\lambda fx.x)\ (n\ f)))\,(\lambda fx.f(f(fx))) \\ (\lambda nf\,.\,((\lambda fx.x)\ (n\ f)))\,[n \to (\lambda fx.f(f(fx)))] &\underset{\beta}{\to} (\lambda f\,.\,((\lambda fx.x)\ ((\lambda fx.f(f(fx)))\ f))) \\ (\lambda fx.f(f(fx)))))[f \to f] &\underset{\beta}{\to} (\lambda fx.f(f(fx))))) \\ &= (\lambda f\,.\,((\lambda fx.x)\ ((\lambda fx.f(f(fx))) \\ (\lambda fx.x)[f \to (\lambda fx.f(f(fx))] &\underset{\beta}{\to} (\lambda x.x) \\ (\lambda f\,.\,(\lambda x.(x))) &= \lambda fx\,.\,x = \ulcorner 0 \urcorner\end{aligned} \tag{3}$$

e. `EXP` $\ulcorner 2 \urcorner\ \ulcorner 1 \urcorner$

EXP $= \lambda mn.(n\ m)$
$\ulcorner 1 \urcorner = \lambda fx.fx$
$\ulcorner 2 \urcorner = \lambda fx.f(fx)$

$$\begin{aligned}\text{EXP}\ulcorner 2 \urcorner \ulcorner 1 \urcorner &= \lambda mn.(n\ m)(\lambda fx.f(fx))(\lambda fx.fx) \\ \lambda mn.(n\ m)[m \to (\lambda fx.f(fx))] &\underset{\beta}{\to} \lambda n.(n\ (\lambda fx.f(fx))) \\ &= \lambda n.(n\ (\lambda fx.f(fx)))(\lambda fx.fx) \\ \lambda n.(n\ (\lambda fx.f(fx)))[n \to (\lambda fx.fx)] &\underset{\beta}{\to} (\lambda fx.fx)\ (\lambda fx.f(fx))) \\ (\lambda fx.fx)[f \to (\lambda fx.f(fx)))] &\underset{\beta}{\to} (\lambda x.(\lambda fx.f(fx)))x) \\ (\lambda x.(\lambda fx.f(fx)))[x \to x] &\underset{\beta}{\to} \lambda fx.f(fx))) \\ &= \lambda fx.f(fx))) = \ulcorner 2 \urcorner\end{aligned} \tag{4}$$

5. [25 points] **Scheme** 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 and in the lecture slides.

   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. Write a function `arg-max` that expects two arguments: a unary function f which maps a value to a number, and a nonempty list A of values. It returns the a for which $(f\ a)$ is largest among all $a \in A$. If the list A is empty, return -1.

      ```
      >(define square (a) (* a a))
      > (arg-max square '(5 4 3 2 1))
      5

      > (define invert (a) (/ 1000 a))
      > (arg-max invert '(5 4 3 2 1))
      1

      > (arg-max (lambda (x) (- 0 (square (- x 3)))) '(5 4 3 2 1))
      3
      ```

   2. Implement a function `zip` which takes an arbitrary number of lists as input and returns a list of those lists. If there are no input lists, return an empty list as output.(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))
      ```

   3. 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)
```

4. 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))
```

5. Implement function `sortedmerge`, which expects two sorted lists of numbers and returns a single sorted list containing exactly the same elements as the two argument lists together. You may assume that both the input lists are always *sorted*.

```
>(sortedmerge '(1 2 3) '(4 5 6))
'(1 2 3 4 5 6)

>(sortedmerge '(1 3 5) '(2 4 6))
'(1 2 3 4 5 6)

>(sortedmerge '(1 3 5) '())
'(1 3 5)
```

6. Implement a function `interleave`, which expects as arguments two lists X and Y, and returns a single list obtained by choosing elements alternately, first from X and then from Y. If the sizes of the lists are not the same, the excess elements on the longer list will appear at the end of the resulting list. Input lists can be empty as well.

```
>(interleave '(1 2 3) '(a b c))
'(1 a 2 b 3 c)

>(interleave '(1 2 3) '(a b c d e f))
'(1 a 2 b 3 c d e f)

>(interleave '(1 2 3 4 5 6) '(a b c))
'(1 a 2 b 3 c 4 5 6)
```

7. A well-known function among the functional languages is `map`, which we saw in the lecture slides. 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 a string containing an error message if the two lists are not of the same size.

8. 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`. Here, the unary function `inc` means `(lambda (x) (+ x 1))`.