Fall 2021
Programming Languages
Homework 2

- Due on Monday, October 11, 2021 at 11:59 PM, Eastern Daylight time.
- 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.

      Answer: We can push string's address after we push the arguments into the stack. First we push the string into the stack and store the address, then we push all other arguments into the stack and then push the address of string into the stack. So the frame pointer is pointing to the address of the string. And it can go through the stack and read arguments.

   2. In Fortran 77, local variables were typically allocated statically. In Algol and its descendants (e.g., Ada and C), they are typically allocated in the stack. In Lisp they are typically allocated at least partially in the heap. Why do you think such differences exist?

      Answer: Because they use different scope so they need different location to store variables. For Fortran 77, it doesn't have recursion so it doesn't need to store in stack, while Algol and its descendants have, so they store local variables in stack. And for Lisp, it can dynamically allocate variables, so it doesn't need to worry about allocation and deallocation, which is why it stores local variables into heap.

   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?

      Answer: If the object is stack-allocated, then its destructor will be called when the object passed out of scope. If the objects is heap-allocated, then its destructor will be called when the program explicitly deletes the object or the program ends.

   4. Consider the following C code:

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

```
    {   int g, h;
        ...
        {
            int i;
        }
    }
    ...
}
```

Assuming that each variable occupies four bytes, how much total space is required for the variables in this code? Provide the most space-optimal solution you can. Explain your solution.

Answer: The most space optimal solution should be 24 bytes. Because in the most optimal situation, six variables are activated, which are a, b, c, g, h, i and d, e, f are not activated.

5. Consider the following pseudo code:

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

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

         procedure P (C,D : real)
         Z : 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.)

Answer: A, B, C, X, Z, P, Q, 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.

Answer: MAIN → BIGSUB → SUB2(7) → SUB3 → SUB1 → print(7, 3, 4) → print(10)

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.

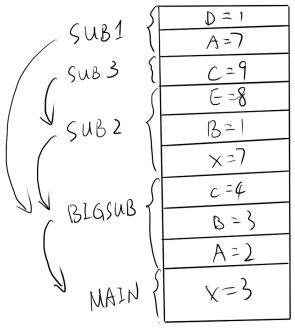
Answer: MAIN.X, BIGSUB.B, BIGSUB.C, SUB1.D, SUB2.E, SUB2.B, SUB2.X, SUB3.C

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

Answer: Yes, it changes the behavior. Because for procedure SUB1, if dynamic scoping is applied, then it would print out (10, 1, 9), rather than (7, 3, 4), which is under static scoping.

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`?

Answer: `MAIN` cannot invoke `SUB3` because `SUB3` is not visible to `MAIN`. But `MAIN` is visible to `SUB3` so `SUB3` can invoke `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.)

      Answer:
      After iteration 1: a1 = 2, a2 = 3, a3 = 8, a4 = 22, a5 = 22
      After iteration 2: a1 = 25, a2 = 26, a3 = 100, a4 = 252, a5 = 252
      After iteration 3: a1 = 278, a2 = 279, a3 = 1112, a4 = 2782, a5 = 2782

   2. Perform the same trace as above, where a1, a4, a5 are passed by *call-by-reference* and a2, a3 are passed by *call-by-need* semantics.

      Answer:
      After iteration 1: a1 = 2, a2 = 2, a3 = 8, a4 = 20, a5 = 20
      After iteration 2: a1 = 22, a2 = 2, a3 = 8, a4 = 20, a5 = 20
      After iteration 3: a1 = 22, a2 = 2, a3 = 8, a4 = 20, a5 = 20

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

      Answer:
      After iteration 1: a1 = 2, a2 = 2, a3 = 4, a4 = 1, a5 = 12
      After iteration 2: a1 = 3, a2 = 2, a3 = 4, a4 = 2, a5 = 12
      After iteration 3: a1 = 4, a2 = 2, a3 = 4, a4 = 3, a5 = 12

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

mystery(i, i+1, i*4, j, j)

procedure mystery (a1, a2, a3, a4, a5)

    for count from 1 to 4 do   // 1 to 3 inclusive
      a1 = a2 + a4;
      a4 = a4 + 1;
      a5 = (a3 + a2)*2;
    end for;

end procedure;
```

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

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

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

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

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

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

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

(for easy format, I use red color as circle)

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

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

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

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 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. $(\lambda xy\,.\,zx)(\lambda x\,.\,x\,y)$
   needs α-conversion.
   without α-conversion:
   $(\lambda xy\,.\,zx)(\lambda x\,.\,x\,y)$
   $\rightarrow_\beta\ (\lambda x\,.\,\lambda y\,.\,zx)(\lambda x\,.\,x\,y)$
   $\rightarrow_\beta\ (\lambda y\,.\,z(\lambda x\,.\,xy))$
   with α-conversion:
   $(\lambda xy\,.\,zx)(\lambda x\,.\,x\,y)$
   $\rightarrow_\alpha\ (\lambda x_0 y_0\,.\,zx_0)(\lambda x\,.\,x\,y)$
   $\rightarrow_\beta\ (\lambda y_0\,.\,z(\lambda x\,.\,x\,y)$
   And they are different.

b. $(\lambda x\,.\,\lambda yz\,.\,xyz)(\lambda z\,.\,zx)$
doesn't need α-conversion.
$(\lambda x\,.\,\lambda yz\,.\,xyz)(\lambda z\,.\,zx)$
$\rightarrow_\beta (\lambda yz\,.\,(\lambda z\,.\,zx)yz))$
$\rightarrow_\beta (\lambda yz\,.\,yxz)$

c. $(\lambda x\,.\,xz)(\lambda xz\,.\,x\,y)$
needs α-conversion.
without α-conversion:
$(\lambda x\,.\,xz)(\lambda xz\,.\,x\,y)$
$\rightarrow_\beta (\lambda xz\,.\,xy)z$
$\rightarrow_\beta (\lambda z\,.\,zy)$
with α-conversion:
$(\lambda x\,.\,xz)(\lambda xz\,.\,x\,y)$
$\rightarrow_\alpha (\lambda x\,.\,xz)(\lambda xz_0\,.\,xy)$
$\rightarrow_\beta {}_\beta(\lambda xz_0\,.\,xy)z$
$\rightarrow_\beta (\lambda z_0\,.\,zy)$
the results of two methods are not the same.

d. $(\lambda x\,.\,x\,y)(\lambda x\,.\,y)$
doesn't need α-conversion.
$(\lambda x\,.\,x\,y)(\lambda x\,.\,y)$
$\rightarrow_\beta (\lambda x\,.\,y)y$
$\rightarrow_\beta y$

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. $((\lambda y\ .\ z\ y)\ x)(\lambda x\ .\ x\ y)$
$\rightarrow_\alpha (((\lambda y_0\ .\ z\ y_0)\ x)(\lambda x_0\ .\ x_0\ y)$
$\rightarrow_\beta (zx(\lambda x_0\,.\,x_0y))$

b. $(\lambda x\ .\ x\ x\ x)\ (\lambda x\ .\ x\ x\ x)$
No normal form.

c. $(\lambda x\ .\ x)(\lambda y\ .\ x\ y)(\lambda z\ .\ x\ y\ z)$
$\rightarrow_\alpha (\lambda x_0\,.\,x_0)(\lambda y_0\,.\,xy_0)(\lambda z\,.\,xyz)$
$\rightarrow_\beta (\lambda y_0\,.\,xy_0)(\lambda z\,.\,xyz)$
$\rightarrow_\beta x(\lambda z\,.\,xyz)$

d. PLUS $\ulcorner 1\urcorner\ \ulcorner 3\urcorner$
$\rightarrow (\lambda mnfx\,.\,mf(nfx))\ 1\ 3$
$\rightarrow_\beta (\lambda nfx\,.\,1f(nfx))\ 3$
$\rightarrow_\beta \lambda fx\,.\,1f(3fx)$
$\rightarrow \lambda fx\,.\,(\lambda f_0x_0\,.\,f_0x_0)f((\lambda f_1x_1\,.\,f_1(f_1(f_1x_1)))fx))$
$\rightarrow_\beta \lambda fx\,.\,(\lambda x_0\,.\,fx_0)(\lambda x_1\,.\,f(f(fx_1))x)$
$\rightarrow_\beta \lambda fx\,.\,f(\lambda x_1\,.\,f(f(fx_1))x)$
$\rightarrow_\beta \lambda fx\,.\,f(f(f(fx)))$

e. SUCC $\ulcorner 2\urcorner$
$\rightarrow \lambda nfx\,.\,f(nfx)\ 2$
$\rightarrow_\beta \lambda fx\,.\,f(2fx)$

$\to \lambda f x \,.\, f((\lambda f_0 x_0 \,.\, f_0(f_0 x_0)) f x)$
$\to_\beta \lambda f x \,.\, f((\lambda x_0 \,.\, f(f x_0)) x)$
$\to_\beta \lambda f x \,.\, f(f(f x))$

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** 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 largest (f a) among all $a \in A$.

      ```
      >(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. (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 `intersectlist` which given two lists, A and B, will return a list of all the numbers appearing in both A and B. Although the inputs are lists, the function should behave as if it is operating on sets. For example, order of items should not affect the output and duplicate items must be removed if present.

```
>(intersectlist '() '())
'()

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

>(intersectlist '(1 2) '(2 4))
'(2)

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

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.

```
>(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.

```
>(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:

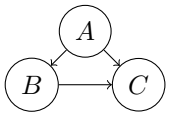
```
(map2 '(1 2 3 4) '(2 3 4 5) (lambda (x) (> x 2)) inc)
```

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. In this problem you will write code that converts the representation of a directed graph. You will work with two representations:
   - The first representation is a list of edges, where a single edge is represented by a two-element list. For example, the list (A B) represents an edge from A to B.

- The second representation uses a adjacency list: a graph is represented by an adjacency list which is a list. Each entry in this list (L) is a list in itself (of length 2) where the first element of L is a list containing a single element(X) denoting a vertex and the second element is a list of vertices to which there is an edge from X.

For example, the graph:



could be represented as an edge list by

```
'('(A B) '(B C) '(A C))
```

and as an adjacency list by

```
'('('(A) '(B C)) '('(B) '(C)) ('(C) '()))
```

(a) Write function edge-list-to-adjacency-list, which accepts a graph in edge list representation and returns a adjacency list representation of the same graph.

(b) Write function adjacency-list-to-edge-list, which accepts a graph in adjacency list representation and returns an edge list representation of the same graph.