# Spring 2024
# Programming Languages
# Homework 2

- Due on Sunday, February 25, 2024 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. Consider a programming language named DynStack. In DynStack, programmers can "dynamically" declare local variables within a subprogram such that these variables reside within the subprogram's activation record (not the heap). Whereas a normal local variable's lifetime begins when the surrounding subprogram begins, a dynamic variable declaration delays the "birth" of the local variable until the declaration is executed for the first time. However, the lifetime of both local variables and dynamically declared local variables ends when the surrounding subprogram terminates. If the declaration is never executed, then the variable never comes into existence in the first place. To be clear, dynamically declared variables will come into existence in the same order as the declarations are first reached at runtime.

      Note that dynamically declared variables are a fictitious feature we are making up for the purposes of this thought exercise—they do not exist in real life.

      Explain at a high level how dynamically declared variables could be implemented, if at all. Full implementation details are unnecessary. We are mostly interested in how you, as a language designer, would set about making this feature a reality. You may assume that the language is a stack-based imperative language, which uses static typing (i.e., types are associated with variables, and so each variable has a known size), and static scoping. You may also make the simplifying assumption that the language does not support nested subprograms, like Ada, Pascal, or functional languages do. You must also assume that all data in the activation record—except dynamically declared variables—must continue to be locatable using a fixed offset from the frame pointer, as discussed in class. You may also state your own *reasonable* assumptions as well.

      To help guide your explanation, we are most interested in the following aspects:

      (a) Is this feature even possible to implement? If not, why not?

      (b) Activation records are normally a fixed size. Could we make the activation record a variable size by extending it as needed? How might that work? Where would the dynamically declared variables go relative to everything else in the activation record?

      (c) When a dynamically declared variable is referenced in a subprogram during runtime, how will the system "look up" where the variable exists within the activation record? Fixed offset from the frame pointer? Something else?

      (d) How would the compiler allocate space for dynamically declared variables within the activation record.

      *Note: we are not looking for a single correct answer. We are interested in your reasoning about the runtime stack and how you are using that reasoning to solve this problem.*

   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 subprograms). Is it possible for a language to enforce this rule statically? Assume NoRec has no function types (i.e., variables you can declare which "point" to functions).

      Please note that the answer may differ based upon your assumptions. There may not be a single correct answer. We are more interested in how you apply your reasoning. *Hint to get you thinking: how might the presence of control structures such as if-then-else or loops affect the answer? What about unused subprograms?*

   3. Now consider another programming language "AllRec," where every subprogram *must* terminate in a recursive call—again, either directly or indirectly through other subprograms. Is it possible to have such a language? If so, how might it work?

   4. Refer to slide 27 of the Subprograms lecture. This slide demonstrates the limits of stack-based allocation. Specifically, the activation record of Make_Incr will disappear after Make_Incr returns and along with it will go Make_Incr's formal parameter, X.

      The way functional languages overcome this problem is to place all activation records in the heap so that activation records like Make_Incr can be kept around longer than they ordinarily would.

This allows Make_Incr's formal parameter, X, to be available when "Add_Five" is called on the last line of the slide.

Suppose storing activation records in the heap wasn't an option. Can you think of other ways to overcome the "Limits of Stack Based Allocation"?

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

   Consider the following pseudo-code:

```
procedure MAIN;
    var X : integer = 5;

    procedure OUTER_SUB;
        var A : integer = 1;
        var B : integer = 2;

        procedure MIDDLE_SUB;
            var D : integer = 7;
            var E : integer = 8;

            procedure INNER_SUB1;
                var D : integer = 3;
                var F : integer = 4;

            begin {INNER_SUB1}
                D := E + F;  <---------------- (1)
                print(D, E, F);
            end; {INNER_SUB1}

            procedure INNER_SUB2(X : integer);
                var F : integer = 9;
                var G : integer = 10;

                procedure INNER_SUB3;
                    var G: integer = 5;

                begin {INNER_SUB3}
                    INNER_SUB1;
                    G := F + D;
                    print(G);
                end; {INNER_SUB3}

            begin {INNER_SUB2}
                INNER_SUB3;
                D := G;
            end; {INNER_SUB2}

        begin {MIDDLE_SUB}
            INNER_SUB2(7);
        end; {MIDDLE_SUB}

    begin {OUTER_SUB}
        MIDDLE_SUB;
    end; {OUTER_SUB}

begin
    OUTER_SUB;
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.

   Example: MAIN → OUTER_SUB → …

b. Over the lifetime of the program above, 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.

   Example: FOO.X, BAR.Y …

c. For the remaining variables that *did* change, what is the last value each variable held during their lifetime?

   Example: FOO.Y=11, BAR.Z=2 …

d. Assume now that dynamic scoping rules are in effect. Does this change the behavior of the program above? Explain why or why not. You don't have to perform an exhaustive trace, but identify at least one behavior that would be different under dynamic scoping rules.

e. 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.

f. According to the static scoping rules we've learned, can `MAIN` invoke `INNER_SUB1`? Give brief explanation for your answer. Can `INNER_SUB1` 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 evaluated 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 $a1$ and $a5$ are passed using *call-by-name* semantics, $a2$ and $a3$ are passed using *call-by-need* semantics, and $a4$ is passed using *call-by-value* semantics.

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

```
var i=1, j=10;

mystery(i, i+2, i*4, i, j+2)

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

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

end procedure;
```

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

   1. This first two sets 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. $\lambda x\ .\ y\ \lambda y\ .\ \lambda z\ .\ z\ x$

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

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

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

   2. Circle all of the free variables (if any) for each of the following lambda expressions (You may have to rewrite the expression using parentheses as you did in the above question in order to identify the free variables):

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

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

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

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

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

   3. These next two sets of questions is intended to help you understand more fully why $\alpha$-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 $\alpha$-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 $\beta$-reduced without any $\alpha$-conversion at any of the steps, according to the rule we learned in class. For any expression below requiring an $\alpha$-conversion, perform the $\beta$-reduction twice: once after performing the $\alpha$-conversion (the correct way) and once after not performing it (the incorrect way). Do the two methods reduce to the same expression?

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

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

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

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

      `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, $\beta$-reduce each to normal form (provided a normal form exists) using applicative order reduction. For each, perform $\alpha$ conversions where required. For clarity, please show each step individually—do not combine multiple reductions on a single line.

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

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

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

      d. `ISZERO` $\ulcorner 1 \urcorner$

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

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

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 `chain` that expects two arguments: a list of unary functions $f_1, f_2, \ldots, f_n$, and an initial value $v$. The `chain` function should compute $f_n(f_2(\ldots, f_1(v)))$

      ```
      >(define inc (a) (+ a 1))
      >(define dec (a) (- a 1))
      >(chain (list inc dec inc dec) 5)
      5
      >(chain (list dec dec dec) 5)
      2
      >(chain (list inc) 5)
      6
      >(chain '() 2)
      2
      ```

      Note: The type of $n$ will not necessarily be numeric. Moreover, for any function $f_i$ the input type will not necessarily be the same as the output type.

   2. Implement a function `chain_odd` that expects two arguments: a list of unary functions $f_1, f_2, \ldots, f_n$, and an initial value $v$. Like above the type of $v$ may not be numeric, but unlike above, the inputs/output types for each function $f_i$ are the same.

      The `chain_odd` function should behave in the same way as chain, except that for each list of functions provided as input $f_1, f_2, \ldots, f_n$, we define another list of functions $g_1, g_2, \ldots, g_n$ exhibiting the following behavior:

      Function $g_i(n) = f_i(n)$ if $n$ is a number and $n$ is odd. Otherwise, $g_i(n) = n$. The `chain_odd` function should then compute $g_n(g_2(\ldots, g_1(v)))$. You might find the built-in predicate `odd?` handy.

      ```
      >(define (inc n) (+ n 1))
      >(define (dec n) (- n 1))
      >(define (plus2 n) (+ n 2))
      ```

```
>(define (minus2 n) (- n 2))
>(define (ident s) s)

>(chainodd (list inc inc inc inc) 3)
4
>(chainodd (list plus2 plus2 dec minus2) 3)
6
>(chainodd '() 2)
2
>(chainodd (list ident ident) "hey")
"hey"
```

3. Implement a function `zip` which takes two input lists (assume they are both the same size $n$) and outputs a list of $n$ pairs, where pair $i$ contains values from the $i$th position of each input list. If the input lists are empty, the output list should be empty.

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

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

4. Implement a function `unzip` which computes the reverse of `zip`. That is, given a list of pairs of size $n$, evaluate to a list of two lists, each of size $n$ where output list $i$ contains values from the $i$th position of each input list. If the input list is empty, the two output lists should be empty.

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

>(unzip '((1 2) (2 3)))
>((1 2) (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. Implement a function `reverse2`, which expects a list X and returns the same list reversed.

```
>(reverse2 '(1 2 3))
(3 2 1)
```

7. Implement a function `interleave_outer`, which expects as arguments two lists X and Y, and returns a single list obtained by choosing elements from the beginning of X and the end of 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_outer '(1 2 3) '(a b c))
(1 c 2 b 3 a)

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

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

8. Write a function `count_occurrences` that takes two arguments: a list lst and an element x. The function should return the number of times x appears in lst.

   Example:

```
>(count-occurrences '(1 2 3 2 4) 2)
2

>(count-occurrences '(1 2 3 2 4) 1)
1

>(count-occurrences '(1 2 3 2 4) 6)
0
```