CSC173: Project 3
Functional Programming

In this unit, we are looking at Alan Turing's formalization of "computable functions" using Turing machines. I have posted Turing's famous paper *On Computable Numbers* (Turing, 1937a) on BlackBoard for you.

The lambda (λ) calculus is an alternative formulation of computable functions developed by the mathematician Alonzo Church around the same time that Turing was developing his machine model (Church, 1936). The two formalizations have been proven equivalent, in that any function computable using one of them can be computed using the other. Turing actually discussed this in an appendix to his famous paper, and in a subsequent paper (Turing, 1937b). The Church-Turing thesis states that these and other equivalent formalisms completely define the informal notion of an "algorithm" or computable function. So far no alternative model has been proposed that can compute anything that can't be done with either Turing machines or the lambda calculus.

Functional programming is based on the lambda calculus. In this project, we will take a short break from C and do a bit of functional programming. However just like actually programming a Turing machine is quite involved, programming directly using the lambda calculus would be painful. Very, very painful. So instead, this term we will use the original functional programming language: Lisp.

Lisp

Lisp, from "List Processing," was invented by John McCarthy in 1958 and first published in 1960 (McCarthy, 1960), making it one of the first programming languages ever. This section provides a very brief introduction to Lisp. I **strongly** urge you to read Paul Graham's "The Roots of Lisp," which lays it out very well for more modern readers IMO.

In Lisp, expressions are either *atoms*, which are sequences of characters, or *lists* of zero or more expressions enclosed in parentheses. For example (from Paul Graham):

```
foo
()
(foo)
(foo bar)
(a b (c) d)
```

Lisp expressions are defined *recursively*. We've seen that before. And BTW: atoms include both *symbols*, that look like variables in other programming languages, and numbers (sequences of digits, *etc.*).

If an expression is a list with at least one element, then we call the first element of the list the *operator* or *functor* and the rest of the elements (possibly none) the *arguments*.

List expressions represent the application of a function to its arguments. For example, `(eq x y)` represents the application of the `eq` function (which tests the most basic form of equality) to arguments `x` and `y`.

There are many builtin functions in Lisp. Paul Graham's article describes the "primitive" building block functions of Lisp in detail: `quote`, `atom`, `eq`, `car`, `cdr`, `cons`, and `cond`. You should probably read that now.

Function definitions are expressed as list expressions using the `lambda` operator:

(`lambda` ($p_1 \ldots p_n$) e)

where the atoms p_i are the *parameters* of the function and e is an expression that constiututes the body (or definition) of the function. For example:

```
(lambda (x y) (not (eq x y)))
```

is a function with two parameters that tests whether the parameters are not equal (in the sense of `eq`).

```
(lambda (z) (+ z 1))
```

is a function with one parameter that adds one to the value of the parameter and returns the sum.

An expression whose first element is a lambda expression is called a *function call*, and represents the application of the function to specific arguments for its parameters:

((`lambda` ($p_1 \ldots p_n$) e) $a_1 \ldots a_n$)

First each argument expression a_i is evaluated in order. Then e is evaluated with the values of each of the parameters p_i set equal to the value of the corresponding a_i. This is just like how arguments are passed to functions via parameters in other programming languages.

For example:

```
((lambda (x y) (not (eq x y))) 'a 'a)
((lambda (x y) (not (eq x y))) 'a 'b)
```

The quotes here mean "the symbol `a` itself, not whatever value has been assigned to the symbol," and similarly for `b`. This takes some getting used to. Don't want until the last minute to figure it out.

The first call returns false (`nil`) since its arguments are the same, and the second one returns true (`t`). Two more examples of function calls:

```
((lambda (z) (+ z 1)) 0)
((lambda (z) (+ z 1)) 99)
```

The first call returns 1; the second returns 100.

Function applications can be nested:

```
((lambda (z) (+ z 1))
 ((lambda (z) (+ z 1)) 0))
```

The outermost call to the "+1" function takes one argument. The argument expression is evaluated first, resulting in the inner call to the "+1" function being evaluated. Its argument is 0, so it returns 1. That value is used as the argument to the outer call to the "+1" function, which then returns 2. So the value of the entire expression is 2.

It is convenient for several reason to be able to name functions, like you name the functions and methods in your programs in other languages. This also allows the definition of recursive functions. There are several ways to do this in Lisp. I will describe only one:

(`defun` f ($p_1 \ldots p_n$) e)

An expression starting with `defun` defines the name (symbol) f to be the `lambda` expression with parameters p_i and body e, where f may be used in e. For example:

```
(defun plus1 (x) (+ 1 x))
(defun factorial (n)
    (* n (factorial (- n 1))))  ; Note: not quite correct...
```

Once a name is defined as a function (that is, as a `lambda` expression)), the name can be used in a function call:

```
(plus1 0)
(plus1 99)
(factorial 7)
```

The first two calls return 1 and 100, respectively. The last one would return 5040, if the definition of the `factorial` function were correct.

That's as far as I'm going to go. You have enough to get you started even if you've never programmed in Lisp before (and most people haven't). The next section gives some tips for installing and running Lisp. The section after that discusses what it means to program *functionally*, which is the key to using Lisp well.

Using Lisp

There are several free implementations of Lisp (and some commercial options). I recommend that you try `abcl`, Armed Bear Common Lisp, from abcl.org. Because `abcl` runs on the Java Virtual Machine (JVM), it runs on Linux, Mac, and Windows just like Java. You will need to run it from a command shell (a.k.a. terminal). If you aren't comfortable with your computer's command line yet, now's a good time to learn.

Download one of the "binary" packages from abcl.org. Unpack the archive to get a folder with `abcl.jar` among other files.

Next open a terminal window (Linux: `xterm` or equivalent; Mac: `Terminal`; Windows `CMD.COM` or `PowerShell` or other).

Change your working directory to the folder that you just unpacked. For example, if you downloaded the archive to your "Downloads" folder and then unpacked it to create the sub-folder "abcl-bin-1.5.0", you would type the following, followed by Return:

```
cd Downloads/abcl-bin-1.5.0
```

You can list the contents of the current directory (folder) with the `ls` command (`dir` in `CMD.COM`). If you need a tutorial on using the command shell, there are plenty on the web.

All you need to do to run `abcl` is to tell Java to run the `jar` (Java Archive) that you downloaded:

```
java -jar abcl.jar
```

You should see some startup messages, and finally a prompt from the main read-eval-print loop (REPL). Here's what I see:

```
Armed Bear Common Lisp 1.5.0
Java 12.0.1 Oracle Corporation
OpenJDK 64-Bit Server VM
Low-level initialization completed in 0.281 seconds.
Startup completed in 1.192 seconds.
Type ":help" for a list of available commands.
CL-USER(1):
```

Try entering just `1` followed by Return. You should see the value 1, which is the value of the expression you entered.

Try evaluating `(+ 1 2)`. (Don't type the period!) You should see the result of applying the function + (plus) to arguments 1 and 2, yielding result 3.

Try evaluating `"hello world"` (that is, a string enclosed in double-quotes). It should be echoed back to you, since that's the value of a string expression.

Try evaluating `(defun plus1 (x) (+ x 1))`. You should see the name `PLUS1` since you just defined a function with that name (Lisp is case-insensitive, that is, not case-sensitive, by default).

Then try evaluating `(plus1 8)`. You should see the value 9, which is the result of evaluating your function `PLUS1` with argument 8.

Type `:exit` to exit `abcl`. Type `:help` for more REPL commands.

You are ready to write some Lisp code! You can type at the REPL, which is good for testing and debugging, but usually you type your code into a file and tell Lisp to load it:

```
(load "project3")
```

This will load and run the code in the file named `project3.lisp` and return you to the REPL if there were no errors.

If you have questions about Lisp or `abcl`, bring them to a TA study session. Do not wait until the last minute!

On Functional Programming

The key to functional programming is to think recursively. What is a recursive definition of a list of, say, numbers? What is a recursive definition of the sum of a list of numbers? What does it mean to "filter" a list of numbers recursively to keep or remove some of them?

It is an amazing fact of Computer Science that any recursive algorithm can be implemented iteratively, and vice-versa. Similarly, it is possible to write almost purely imperative programs using Lisp (or most practical functional programming languages), but *that is not the goal for this project*. Your code should be as purely functional as possible.

Paul Graham is the founder of Y Combinator (which is named after a very cool functional programming construct), among other accomplishments. He has this to say about learning to program in Lisp:

> For alumni of other languages, beginning to use Lisp may be like stepping onto a skating rink for the first time. It's actually much easier to get around on ice than it is on dry land—*if* you use skates. Till then you will be left wondering what people see in this sport.
>
> What skates are to ice, functional programming is to Lisp. Together the two allow you to travel more gracefully, with less effort. But if you are accustomed to another mode of travel, this may not be your experience at first. One of the obstacles to learning Lisp as a second language is learning to program in a functional style. (Graham, 1994, p. 33)

He goes on to describe a way of making this mental shift. I've posted a longer excerpt from his book *On Lisp* on BlackBoard with the readings for this unit.

Our textbook also discusses this:

> There is a common belief that it is easier to learn to program iteratively, or to use nonrecursive function calls, than it is to learn to program recursively. While we cannot argue conclusively against that point of view, we do believe that recursive programming is easy once one has had the opportunity to practice the style. Recursive programs are often more succinct or easier to understand than their imperative counterparts. More importantly, some problems are more easily attacked by recursive programs than by iterative programs [for example, searching trees]. (Aho and Ullman, 1995, pp. 69–70)

Project Requirements

You must implement **four** functions from **each** of the "List," "Set," and "Math" sections below. You must also implement the three functions from the final "Required Functions" section. So fifteen (15) functions total are required. Don't worry—most of them can be done in no more than a few lines of Lisp.

In some cases, these functions are already defined in Lisp. That doesn't matter—your definitions will replace the builtin ones. [1] You may use *only* the following builtins:

- Constants: `t`, `nil`, numbers
- List functions: `cons`, `car`, `cdr`, `list`, `null`
- Function functions (!): `defun`, `funcall`, `apply`, `lambda` (not really a function but...)
- Equality functions: `eq`, `eql`, `equal`, `equalp`
- Math operators: `+` (addition), `-` (subtraction), `*` (multiplication), `/` (division)
- Comparison operators: `>`, `>=`, `<`, `<=`
- Boolean operators: `and`, `or`, `not` (you could easily do these yourself)
- Conditional functions: `if`, `cond`, `when`, `unless`
- Input/Output functions: `read`, `format`, `princ` and its relatives, and anything else in Sec. 22.3.1 of *Common Lisp: The Language* if you think you need it (`finish-output` is useful with prompts)

I think that list is sufficient. If you think otherwise, ask well before the deadline.

Note that you may **NOT** use imperative programming forms like `let`, `setq`, or `loop` in the definitions of your functions. Be functional. You may use such functions in your REPL if you like (see below).

[1] To supress redefinition warnings from ABCL, add the following at the top of your source file:

```
(setq ext:*warn-on-redefinition* nil)
```

or you may use some other name for your version of the functions.

Your submission **must** be a single Lisp (`.lisp`) file, in addition to your README or other documentation.

- Loading the file must define and run all the functions.
- Your code must print informative messages, typically using `format` or related output functions.
- Your code must demonstrate execution of the example used in the definitions given below below.
- For each function, your code must also allow the the user to test the function using a REPL. That is, it should print an informative prompt, use `read` (or equivalent) to read the argument(s), run the function on the user's input, and print the result. You should have one or at most a couple of functions to do this–cutting-and-pasting large chunks of code is bad form.

In other words: we will load your single Lisp file. If everything runs smoothly and it is clear what is going on and that your functions are working properly, you will get full points for execution. If not, you won't. There is some sample output at the end of this document.

Finally: write your code using good functional style. Use recursion rather than iteration. You might also find the freely-available book *How to Design Programs* helpful (see References).

List Functions

You must implement at least four of these list functions.

1. Append two lists

```
(append '(1 3 x a) '(4 2 b)) => (1 3 x a 4 2 b)
```

2. Reverse a list

```
(reverse '(a b c d)) => (d c b a)
```

3. Map a function over every element in a list (this is called `mapcar` in Lisp)

```
(defun add3 (x) (+ 3 x))
(map 'add3 '(1 2 3 4)) => (4 5 6 7)
```

4. Remove duplicates from a list

```
(nub '(1 1 2 4 1 2 5)) => (1 2 4 5)
```

5. Fold-left (arguments are: initial value, function, list)

```
(fold 10 '- '(1 3 2)) => 4
```

6. Filter

```
(defun lessthan3 (x) (< x 3))
(filter 'lessthan3 '(1 4 5 2 1 6)) => (1 2 1)
```

7. Merge two sorted lists

```
(merge '(1 3 4 7) '(2 3 6)) => (1 2 3 3 4 6 7)
```

8. Add an element to the end of a list. Cool hint: try using `reverse`

```
(addtoend 'd '(a b c)) => (a b c d)
```

9. Index of

```
(indexof 'a '(b c a d)) => 2
(indexof 'a '(b c d f)) => -1
```

10. Remove-all

```
(remove-all 'a '(b a c a a d a)) => (b c d)
```

Set Functions

Set functions use lists to represent sets, but of course sets may not contain duplicate elements. You must implement at least four of these set functions.

1. Set membership

```
(member 'a '(b c a d)) => t
(member 'z '(b c a d)) => nil
```

2. Insert element into set

```
(insert 'a '(b c d)) => (a b c d)
(insert 'a '(a b c d)) => (a b c d)
```

3. Set intersection

```
(intersection '(a b c) '(a c d)) => (a c)
```

4. Set union

```
(union '(a b c) '(a c d)) => (a b c d)
```

5. Set difference

```
(difference '(a b c) '(a c d)) => (b)
(difference '(a c d) '(a b c)) => (d)
```

6. Symmetric difference (disjunctive union)

```
(symdiff '(a b c) '(a c d)) => (b d)
```

7. Check if subset or equal ($\subseteq$; the `p` in the function name stands for "predicate," meaning something that is true or false)

```
(subsetp '(a b) '(a b c d)) => t
```

8. Check if superset or equal ($\supseteq$)

```
(supersetp '(a b c d) '(a b)) => t
```

9. Cardinality

```
(cardinality '(a b c)) => 3
```

10. Power set (if you need a hint, check Wikipedia)

```
(powerset '()) => (())
(powerset '(a b c)) =>
  '(()
    (a)
    (b)
    (c)
    (a b)
    (a c)
    (b c)
    (a b c))
```

Math Functions

You must implement at least four of these math functions.

1. Absolute value

```
(abs 7) => 7
(abs -7) => 7
```

2. Factorial

```
(factorial 5) => 120
```

3. Check if 3 integers can be the lengths of the two sides and the hypoteneuse of a right triangle (in that order)

```
(right-tri 3 4 5) => t
(right-tri 1 2 3) => nil
```

4. Greatest Common Divisor (GCD)

```
(gcd 8 12) => 4
```

Hint: Euclid could have written this function.

5. Least Common Multiple (LCM)

```
(lcm 4 6) => 12
```

6. Nth Fibonacci number (the 0^{th} Fibonacci number is 0, the first Fibonacci number is 1; see Wikipedia).

```
(nth-fibo 6) => 8
(nth-fibo 10) => 55
```

7. Test if a number is prime

```
(primep 5) => t
(primep 6) => nil
```

   See Wikipedia for test cases.

8. Nth prime number (the first prime number is 2)

```
(nth-prime 6) => 13
(nth-prime 26) => 101
```

Required Functions

You must implement all three of these mathematical functions on integers.

1. Check if a number is *perfect*: a number is perfect if the sum of its factors other than itself is equal to the number itself.

```
(perfectp 5) => nil
(perfectp 6) => t
```

2. Check if a number is *abundant*: an abundant number's sum of factors other than itself is greater than the number itself.

```
(abundantp 5) => nil
(abundantp 12) => t
```

3. Check if a number is *deficient*: a deficient number's sum of factors other than itself is less than the number itself.

```
(deficientp 5) => t
(deficientp 12) => nil
```

These can all be written very concisely using a functional language like Lisp. You may, of course, use any of the functions that you implemented from the lists above.

Recursive functions in general, and these functions in particular, may run out of memory space for their call stack, a so-called "stack overflow." If you encounter that, document it in your README. Your functions should work for *some* arguments though.... FYI: A good compiler will turn so-called "tail recursion" into iteration, so it will never run out of stack. Check out the Common Lisp function `compile-file` (there is also `compile`, but I had problems using that in ABCL).

Extra Credit

- Implement all the additional functions from the lists for extra credit up to 10% max if you do them all properly. Max extra credit is less if you do less extra functions.

Project Submission

Your project submission MUST include the following:

1. A README.txt file or PDF document describing:
   (a) Any collaborators (see below)
   (b) Acknowledgements for anything you did not code yourself (you should avoid this, other than the code we've given you, which you don't have to acknowledge)
   (c) Any issues we should know about, limitations, or special things that you did.
2. All source code for your project in a single Lisp file

As described above, we will load your single file into `abcl`. It should be self-explanatory, meaning that when we load the file, it is clear from the output what the code is doing. **The easier you make this for us, the better your grade will be.** It is your job to make this clear and simple for us.

Late Policy

Late projects will **not** be accepted. Submit what you have by the deadline. If there are extenuating circumstances, submit what you have before the deadline and then explain yourself via email.

If you have a medical excuse (see the course syllabus), submit what you have and explain yourself as soon as you are able.

Collaboration Policy

You will learn the most if you do the projects YOURSELF.

That said, collaboration on projects is permitted, subject to the following requirements:

- Groups of no more than 3 students, all currently taking CSC173.
- You must be able to explain anything you or your group submit, IN PERSON AT ANY TIME, at the instructor's or TA's discretion.
- One member of the group should submit on the group's behalf and the grade will be shared with other members of the group. Other group members should submit a short comment naming the other collaborators.
- All members of a collaborative group will get the same grade on the project.

Academic Honesty

Do not copy code from other students or from the Internet.

Avoid Github and StackOverflow completely for the duration of this course.

There is code out there for all these projects. You know it. We know it.

Posting homework and project solutions to public repositories on sites like GitHub is a violation of the University's Academic Honesty Policy, Section V.B.2 "Giving Unauthorized Aid." Honestly, no prospective employer wants to see your coursework. Make a great project outside of class and share that instead to show off your chops.

Acknowledgements

Thanks to the always-functional Mikayla Konst for her contributions to this project back in the day.

References

Aho, A., and Ullman, J. (1995). *Foundations of Computer Science*. W. H. Freeman and Company.

Church, A. (1936). "An unsolvable problem of elementary number theory." *American Journal of Mathematics* 58 (2), pp. 345–363.

Felleisen, M., Findler, R. B., Flatt, M., and Krishnamurthi, S. (2014) *How to Design Programs, Second Edition*. MIT Press. https://htdp.org/

Graham, P. (1994). *On Lisp*. Pentice-Hall.

Graham, P. (2001). "The Roots of Lisp". http://paulgraham.com/rootsoflisp.html.

McCarthy, J. (1960). Recursive functions of symbolic expressions and their computation by machine. *Communications of the ACM* 3(4), pp. 184–195.

Turing, A.M. (1937a). "On computable numbers, with an application to the Entscheidungsproblem." *Proceedings of the London Mathematical Society* 2(1), pp. 230–265.

Turing, A. M. (1937b). "Computability and λ-definability." *Journal of Symbolic Logic* 2(4), pp. 153–163.

Sample Output

Here is the output of my code on all the examples used above, but without invoking the REPL.

```
Armed Bear Common Lisp 1.5.0
Java 1.8.0_92 Oracle Corporation
Java HotSpot(TM) 64-Bit Server VM
Low-level initialization completed in 0.229 seconds.
Startup completed in 1.005 seconds.
***
*** CSC242 Intro to AI
*** Project 3: Functional Programming in Lisp
*** George Ferguson
***

*** List functions

(APPEND (1 3 X A) (4 2 B)) => (1 3 X A 4 2 B)
(REVERSE (A B C D)) => (D C B A)
(MAP ADD3 (1 2 3 4)) => (4 5 6 7)
(NUB (1 1 2 4 1 2 5)) => (1 2 4 5)
(FOLD 10 - (1 3 2)) => 4
(FILTER LESSTHAN3 (1 4 5 2 1 6)) => (1 2 1)
(MERGE (1 3 4 7) (2 3 6)) => (1 2 3 3 4 6 7)
(ADDTOEND D (A B C)) => (A B C D)
(ADDTOEND2 D (A B C)) => (A B C D)
(INDEXOF A (B C A D)) => 2
(INDEXOF A (B C D F)) => -1
(REMOVE-ALL A (B A C A A D A)) => (B C D)

*** Set functions

(MEMBER A (B C A D)) => T
(INSERT A (B C D)) => (A B C D)
(INSERT A (A B C D)) => (A B C D)
(INTERSECTION (A B C) (A C D)) => (A C)
(UNION (A B C) (A C D)) => (A B C D)
(DIFFERENCE (A B C) (A C D)) => (B)
(DIFFERENCE (A C D) (A B C)) => (D)
(SYMDIFF (A B C) (A C D)) => (B D)
(SUBSETP (A B) (A B C D)) => T
(SUBSETP (A B Q) (A B C D)) => NIL
(SUPERSETP (A B C D) (A B)) => T
(SUPERSETP (A B C D) (A Q B)) => NIL
```

```
(CARDINALITY (A B C)) => 3
(POWERSET (A B C)) => (NIL (C) (B) (B C) (A) (A C) (A B) (A B C))

*** Math functions

(ABS 7) => 7
(ABS -7) => 7
(FACTORIAL 5) => 120
(RIGHT-TRI 3 4 5) => T
(RIGHT-TRI 1 2 3) => NIL
(GCD 8 12) => 4
(LCM 4 6) => 12
(NTH-FIBO 6) => 8
(NTH-FIBO 10) => 55
(PRIMEP 1) => NIL
(PRIMEP 2) => T
(PRIMEP 4) => NIL
(PRIMEP 5) => T
(PRIMEP 10) => NIL
(PRIMEP 11) => T
(PRIMEP 101) => T
(NTH-PRIME 1) => 2
(NTH-PRIME 2) => 3
(NTH-PRIME 3) => 5
(NTH-PRIME 4) => 7
(NTH-PRIME 5) => 11
(NTH-PRIME 6) => 13
(NTH-PRIME 26) => 101

*** Required functions

(PERFECTP 5) => NIL
(PERFECTP 6) => T
(PERFECTP 36) => NIL
(PERFECTP 496) => T
(ABUNDANTP 5) => NIL
(ABUNDANTP 12) => T
(DEFICIENTP 5) => T
(DEFICIENTP 12) => NIL
```

Here is an example of using a REPL to test one of the functions.

```
(APPEND (1 3 X A) (4 2 B)) => (1 3 X A 4 2 B)
Enter a list of arguments for APPEND (NIL to stop): ((a b c) (1 2 3))
(APPEND (A B C) (1 2 3)) => (A B C 1 2 3)
Enter a list of arguments for APPEND (NIL to stop): ((a b c) ())
(APPEND (A B C) NIL) => (A B C)
Enter a list of arguments for APPEND (NIL to stop): (() ())
(APPEND NIL NIL) => NIL
Enter a list of arguments for APPEND (NIL to stop): nil
(REVERSE (A B C D)) => (D C B A)
Enter a list of arguments for REVERSE (NIL to stop): ...
```

Note that the arguments must be enclosed in parentheses (since they are a *list* of arguments, as requested by the prompt). There are other ways to do it, but this makes it easy to use `read` to get input from the user. Then you need to know how to apply a function to a list containing its arguments...

FWIW: My code contains a single function `test` that has only a single line of code in its body. My REPL is implemented as a single small function named `repl` that calls `test` and implements the "L" part without using `LOOP` or any other iteration constructs (those are not allowed for this project). Any questions: bring them to study session.