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 (at least in part). 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.

If you are new to Lisp, and most students will be, please check out the accompanying "Notes on Lisp for CSC173" that I have posted with this project description. **It will take some time for you to get comfortable programming in Lisp.** Be prepared to go to a study session if it is new to you.

You should probably bookmark the Common Lisp HyperSpec, which documents all of Common Lisp (but is a reference, not a tutorial). However see Project Requirements, below, regarding which builtin features of Common Lisp you may usein this project.

On Functional Programming

What is "functional" programming? Surely all programs should be "functional" in the sense of functioning properly. That is true. It's also not what functional programming is about.

Functional programming is a way of specifying how to compute something by giving its definition rather than by giving a set of steps that must be followed to perform the computation. This type of programming is called *declarative*, as opposed to the *imperative*[1] languages like Java and Python with which you are already familiar. Computational definitions are often *recursive*, so recursion is very widely used in functional programming.

For example, take the problem of telling a computer how to add up the numbers from 1 to some number n. If you wrote the program in an imperative language like Java it would look something like this:

```
int sum(int n) {
    int sum = 0;
    for (int i=1; i <= n; i++) {
        sum += i;
    }
    return sum;
}
```

As an alternative, try to think of how you would state the *definition* of the sum of the numbers from 1 to n, rather than describing the steps required to compute it.

What's the simplest case?

Think about it...

If $n = 1$, the sum of the numbers up to n is... 1, right?

What about for numbers greater than 1?

Think about it also...

If $n > 1$, if you know the sum up to $n - 1$, and then the sum up to n is... that plus n, right? Note that this a *recursive* definition.

[1] imperative: from Latin *imperare* meaning "to command."

In mathematical notation, we might write that as follows:

$$sum(n) = \begin{cases} 1 & \text{if} \quad n = 1 \\ sum(n-1) + n & \text{if} \quad n > 1 \end{cases}$$

In a functional programming language, *the program will look just like the definition*! Perhaps something like this:

```
def sum(n):
    n = 1: 1
    n > 1: sum(n-1) + n
```

Note that there is no "return" statement here. We're not telling the computer what to do. If n is 1, the sum is 1. That's what it says. If $n > 1$, the sum is something else and we're given an expression for computing it.

Of course, you could also write the Java method recursively:

```
int sum(int n) {
    if (n == 1) return 1;
    if (n > 1) return sum(n-1) + n;
}
```

BTW: If you actually try this code, your Java compiler should complain. But you could fix it up.

The key to functional programming is to think declaratively ("say what something is not how to compute it"). Very often that will involve recursion. 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? That's how you need to think in order to program functionally.

All (almost all?) functional programming languages support some aspects of imperative progranmming. In fact, 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. For that reason, you are only allowed to use a subset of Lisp features, as described below.

Here's what Paul Graham, the founder of legendary tech incubator Y Combinator, has 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)

Graham 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.[2]

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)

[2] And by the way, the original "Y Combinator" is a actually a fascinating functional programming language construct, first described by Haskell Curry, after whom the functional programming language Haskell is named. But I digress. . .

Project Requirements

You must implement the **four** functions from **each** of the "List," "Set," and "Math" sections below. So twelve (12) functions total. 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. We could simply let your definitions replace the builtin ones. But to ensure that we are testing *your* functions, **all the functions that you must write have names that start with a period (or dot, "`.`")**. Your functions **must** use the correct names so that we can test them. Of course you may also define helper functions, or reuse functions that you have written yourself to implement another required function.

NOTE: You may use *only* the following builtin features from Common Lisp:

- Constants: `t`, `nil`, numbers
- Quotation: `quote` or "`'`"
- Type functions: `listp`, `consp`, `numberp`, `atom` (no "`p`"), *etc.*
- List functions: `cons`, `car`, `cdr`, `list`, `null`, `caar`, `cadr`, `first`, `second`, *etc.*
- Function functions (!): `defun`, `funcall`, `apply`, `lambda` (not really a function but...)
- Equality functions: `eq`, `eql`, `equal`, `equalp`
- Math operators: `+` (addition), `-` (subtraction), `*` (multiplication), `/` (division); you may **not** use the built-in modulus (`mod`) or remainder (`rem`) functions, but they have straightforward recursive definitions so you can easily write your own if you need them
- Comparison operators: `>`, `>=`, `<`, `<=`
- Boolean operators: `and`, `or`, `not` (you could easily write these yourself)
- Conditional functions: `cond`, `if`, `when`, `unless` (you could easily write all these using `cond`)
- 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`, `loop`, or `return` in the definitions of your functions. Be functional. You might also find the freely-available book *How to Design Programs* helpful.

Your submission **must** be a single Lisp (`.lisp`) file, in addition to your README or other documentation. We will load your file into Lisp and then test the required functions on a range of inputs. You will get points for the functions that are correct.

There is no opportunity for extra credit on this project.

Preliminaries

For functions that take another function or functions as parameter(s), you will need to use `apply` or `funcall` to invoke the given function. For example:

```
(defun math (f x y) (funcall f x y)) => MATH
(math '+ 2 3) => 5
(math '- 2 3) => -1
(math '* 2 3) => 6
(math '/ 2 3) => 2/3
```

List Functions

You may assume that any arguments to these functions that are supposed to be lists are in fact lists. That is, you do **not** need to check that they are lists (using `listp`, for example).

1. `(.member X L)`: Return true (non-`nil`) if `X` is a member of list `L` (test with `equalp`)

   ```
   (.member 3 '(1 3 x a)) => T
   (.member 4 '(1 3 x a)) => NIL
   ```

2. `(.remove-all X L)`: Return a new list which contains all the elemnts of `L` that are *not* `X` (test using `equalp`)

   ```
   (.remove-all 'a '(b a c a a d a)) => (B C D)
   ```

3. `(.foldl L F Z)`: Fold-left: for list `L`, binary function `F`, and initial value `Z`: If the list is empty, the result is the initial value `Z`. If not, fold the tail of the list `L` using as new initial value the result of applying `F` to the old initial value `Z` and the first element of `L`.

```
(.foldl '(1 2 3 4 5) '+ 0) => 15
(.foldl '(1 3 2) '- 10) => 4
(.foldl '(1 2 3) 'cons nil) => (((NIL . 1) . 2) . 3)
```

   Note: In Java, this is `java.util.stream.reduce`. In Python it is the builtin function `reduce`.

4. Sum of the (numeric) elements in a list

```
(.sum '(1 2 3 4)) => 10
(.sum nil) => 0
```

Set Functions

Set functions use lists to represent sets, but of course sets may not contain duplicate elements. You may assume that any sets (lists) given as arguments to these functions are lists that satisfy that requirement. That is, you do **not** need to check that it is true for sets (lists) given as argument. But of course you **do** need to ensure that it is true for sets that are the result of any of your functions.

1. `(.add-element X L)`: Return the set resulting from adding element `X` to set `L`

```
(.add-element 'a '(b c d)) => (A B C D)
(.add-element 'a '(a b c d)) => (A B C D)
```

2. `(.intersection S1 S2)`: Return the set that is the intersection of sets `S1` and `S2` (that is: $S_1 \cap S_2$)

```
(.intersection '(a b c) '(a c d)) => (A C)
```

3. `(.supersetp S1 S2)` : Return true (non-`nil`) if set `S1` is a superset of or equal to `S2`, otherwise return `nil` (that is: if $S_1 \supseteq S_2$)

```
(.supersetp '(a b c d) '(a b)) => T
(.supersetp '(a b) '(a b)) => T
(.supersetp '() '(a b)) => NIL
```

4. `(cardinality S)`: Return the cardinality (number of elements in) set `S`

```
(.cardinality '()) => 0
(.cardinality '(a b c)) => 3
```

Math Functions

You may assume that any arguments to these functions that are supposed to be numbers of some kind are in fact numbers of that kind. That is, you do **not** have to check that they are numbers (using `numberp`, for example).

1. `(.factorial N)`: Return the factorial ($N!$) of the given non-negative integer

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

2. `(.gcd X Y)`: Return the greatest common divisor (GCD) of the two given positive integers (the GCD is the largest integer that divides both integers)

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

   Hint: Euclid could have written this function.

3. `(.pow X Y)`: Return the value of `X` to the power `Y` (that is: x^y; you may assume that `Y` is an integer)

```
(.pow 2 8) => 256
(.pow 2 0) => 1
(.pow 2 -2) => 1/4
```

4. `(.with-annual-interest P R N)`: Return value V of principal amount P plus interest at rate r compounded annually for n years:

$$V = P(1 + r)^n$$

```
(.with-annual-interest 100 0.10 1) => 110.0
(.with-annual-interest 100 0.05 10) => 162.8894
```

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

Late Policy

Late projects will receive a grade of 0. You MUST 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

I assume that you are in this course to learn. You will learn the most if you do the projects **YOURSELF**.

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

- Teams of no more than 3 students, all currently taking CSC173.
- You MUST be able to explain anything you or your team submit, IN PERSON AT ANY TIME, at the instructor's or TA's discretion.
- One member of the team should submit code on the team's behalf in addition to their writeup. Other team members MUST submit a README (only) indicating who their collaborators are.

- All members of a collaborative team will get the same grade on the project.

Working in a team only works if you actually do all parts of the project together. If you only do one part of, say, three, then you only learn one third of the material. If one member of a team doesn't do their part or does it incorrectly (or dishonestly), all members pay the price.

Academic Honesty

I assume that you are in this course to learn. You will learn nothing if you don't do the projects yourself.

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*, http://htdp.org

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

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.