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`)

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.

Note on functions that take other functions as arguments

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 twice (f x)
  "Returns the result of applying function F twice to argument X."
  (funcall f (funcall f x)))

(twice 'sqrt 16) ; => 2.0
(twice '1+ 7) ; => 9
(twice 'cdr '(a b c d)) ; => (C D)
(twice 'car '((a b c) '(d e f))) ; => A
```

The difference between `funcall` and `apply` is that `funcall` takes a function as its first argument and passes any remaining arguments to the function, while `apply` takes a function and a list as its two arguments, and “applies” the function to the list of arguments, meaning that that the elements of the list become to arguments to the function.

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).

`(.contains L X)`: Returns true (non-`nil`) if the list `L` contains the element `X` (test elements with `equalp`), otherwise false (`nil`)

```
(.contains '(1 3 x a) 3) => T
(.contains '(1 3 x a) 2) => NIL
(.contains '() 2) => NIL
(.contains '((a 1) (b 2) (c 3)) '(b 2)) => T
```

`(.element-at L I)`: Returns the element from list `L` at index `I` (a number), or `nil` if there is no such element

```
(.element-at '(a b c) 1) => B
(.element-at '(a b c) 3) => NIL
(.element-at '(a b c) -1) => NIL
(.element-at '() 0) => NIL
(.element-at '((a () 2)) 1) => NIL)
```

`(.addtoend X L)`: Returns a list which contains the contents of `L` in order, followed by the element `X`

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

The following function doesn't operate on lists but I'm including it here anyway.

`(.ntimes F X N)`: Returns the result of applying function `F` `N` times to argument `X`

```
(.ntimes 'sqrt 16 2) => 2.0
(.ntimes '1+ 7 4) => 11
(.ntimes 'cdr '(a b c d) 3) => (D)
(.ntimes 'car '(((a 1) (b 2)) ((c 3) (d 4))) 3) => A
```

Hint: This is the generalization of the function `twice`, shown previously.

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.

`(.element-of X S)`: Returns true (non-`nil`) if the element `X` is an element of the set `S` (test elements with `equalp`), otherwise false (`nil`)

```
(.element-of 'a '(b c a d)) => T
(.element-of 'z '(b c a d)) => NIL
```

`(.insert S X)`: Returns the set resulting from adding element `X` to set `S` (test elements with `equalp`)

```
(.insert '(b c d) 'a) => (A B C D)
(.insert '(a b c d) 'a) => (A B C D)
```

`(.union S1 S2)`: Returns the set that is the union of sets `S1` and `S2` (that is: $L_1 \cup L_2$); test elements with `equalp`

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

`(.superseteq S1 S2)`: Returns true (non-`nil`) if set `S1` is a superset of or equal to `S2` (that is, if $S_1 \supseteq S_2$), otherwise return false (`nil`); test elements with `equalp`

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

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).

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

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

`(.nth-tri N)`: Returns the `N`'th triangular number, which is the sum of the sum of the `N` natural numbers from 0 to `N`.

```
(.nth-tri 1) => 1
(.nth-tri 3) => 6
(.nth-tri 10) => 55
```

`(.log2int X)`: Returns the integer part of the binary (base 2) logarithm of the given non-negative number `X`, which is the exponent of the largest power of 2 less than or equal to `X`:

```
(.log2int 1) => 0
(.log2int 2) => 1
(.log2int 3) => 1
(.log2int 4) => 2
(.log2int 6) => 2
(.log2int 8) => 3
(.log2int 15) => 3
(.log2int 16) => 4
```

`(.monthly-payment P R N)`: Monthly mortgage payment p for principal amount P, monthly interest rate r, for n months:

$$p = \frac{rP}{1 - \frac{1}{(1+r)^n}}$$

Example with principal $\$150{,}000$, annual interest rate 6% (so monthly interest rate $r = 0.06/12$) for 25 years (so $n = 25 \cdot 12$):

```
(.monthly-payment 150000 (/ 0.06 12) (* 25 12)) => 966.4525
```