

# CS415 — Discussion Section Notes 1

Pieter Hooimeijer

2007-01-28

## Outline

1. Functional Programming
  - Recap
  - Examples: Fold, Filter
  - Type inference
2. The COOL Programming Language
  - List data type

## Functional Programming

### Recap: What are they?

Most popular programming languages are *imperative* — we get stuff done by changing variables, messing around with pointers, etc. Getting the correct output at the end is sort of a ‘happy coincidence’ of all these actions. This is **not** the only way to do things.

Functional programming avoids mutable variables and direct modification of the state. Programming is viewed as the *evaluation of mathematical functions*.

This may seem counter intuitive (and counter productive, if you’re working on PA1...). Nevertheless, functional programming is **just as powerful** as imperative programming. In other words, there is nothing you can do with C that you can’t do with OCAML...

Consider this exciting piece of code:

$$x = x + 5$$

Suppose this is valid code in some imperative language (say C), and some functional language (like OCAML or Haskell). This is what happens:

- In C: The value  $x + 5$  is computed. Variable  $x$  is set equal to the result of this computation (the original value of  $x$  is replaced).
- In Some Functional Language: Infinite recursion! Symbol  $x = x + 5 = (x + 5) + 5 = ((x + 5) + 5) + 5$ , etc. Each time the interpreter sees  $x$ , it treats it as a mathematical symbol. The expression diverges.

In general, we use recursion (of the non-infinite variety) to get stuff done in functional programming languages.

### Higher order function examples

Let's do some folding, but this time we'll use OCAML rather than the home-grown stuff you saw in class. First, we'll define some **useful** functions using **fold**.

```
open List (* for fold_left, map, filter, etc. *)

let odd x = x mod 2 != 0

let add x y = x + y
let sum = fold_left add 0
```

Loading this in OCAML yields the following:

```
mymachine:~/cs415$ ocaml
Objective Caml version 3.09.1

# #use "section1.ml";;
val add : int -> int -> int = <fun>
val sum : int list -> int = <fun>
# sum [1;2;3];;
- : int = 6
# sum (filter odd [1;2;3;4;5;6;7;8;9]);;
- : int = 25
# sum [1;3;5;7;9];;
- : int = 25
```

I threw in **filter** for free. Note that **odd** takes an integer  $x$  and returns 'false' if it's even, and 'true' otherwise. The real question is: what does `filter odd [...]` return?

Let's try a slightly more complicated example: cumulative sum. Rather than calculating just the sum of a list, we want a running total.

```
# let accumulator lst y = match lst with
```

```

        | []          -> [y]
        | (x::xs)    -> (x + y)::(x::xs)
    ;;
val accumulator : int list -> int -> int list = <fun>

# let accumulate = fold_left accumulator []
    ;;
val accumulate : int list -> int list = <fun>

# accumulate [1;2;3;4;5]
    ;;
- : int list = [15; 10; 6; 3; 1]

```

The `accumulator` function is a bit more complicated than `add` from before. This is because the accumulator keeps track of a list of sums, rather than just the one running total.

## The COOL Programming Language

### The List datatype in COOL

Make sure you don't start from scratch with your COOL implementation for PA1 — use the hint file that's provided. As it turns out, the hint file gives you a `List` datatype. This is useful because COOL does not have a standard library to speak of. Or arrays... Or anything else...

The `List` datatype looks a little odd: it is constructed of *nested pairs*. The pairs are implemented by 'cons cells,' a bit of naming that is left over from the LISP and SCHEME world. In addition to `Cons` there is the `Nil` class, which denotes both the empty list and the end of a nonempty list.

The class hierarchy for List looks as follows:



The `Cons` class contains two references: one to a string (its element) and one to a `List`. This `List` can either be another cons cell (for the next element), or it can be an instance of `Nil`, which always ends the list.

Let's have some practice with this list datatype. We'll add an **append** method. Instead of **insert** (which does an insertion sort), **append** will always add its element to the end of the list.

```
Class List inherits IO {
  -- ...

  append(i : String) : List { self };
};

Class Nil inherits List { -- Nil is an empty list
  -- ...

  append(i : String) : List { (new Cons).init(i, self) };
};

Class Cons inherits List { -- a Cons cell is a non-empty
  xcar : String;           -- list
  xcdr : List;

  -- ...

  append(i : String) : List {
    (new Cons).init(xcar, xcdr.append(i))
  };
};
```

Note that our **append** method is recursive. For an empty list (consisting of just one Nil object), the method returns a new pair. The new list consists of a single pair that has a reference to the string and to the original Nil object, which now serves as the end-of-list marker.

For any nonempty list, the **append** method will have to traverse to the end of the list. This is done by creating a new cons cell at every step. The new cell has the same element, but its 'tail' is updated with a recursive call to **append**. Eventually this call chain will reach a Nil object.

This list data structure is very inefficient. Appending an element to the end of the list takes  $O(n)$  operations, and it creates a whole new set of cons cells. Note to self: how much time would it take to 'prepend' an element (i.e. tack it to the front of the list rather than the back)?