CS 51
CODE REVIEW 1

SAMUEL GREEN

1. Philosophy

There is more than one way to solve a problem.
Some ways are better than others.

2. Important Terminology

A value is the unit of reasoning in a functional programming language. value

A type is the abstract analog of a value. Every value in `OCaml` has exactly 1 type. type

Primitive data types refer to the smallest, most essential building blocks of a programming language. By composing primitives, we build up a richer and richer vocabulary with which to represent the world. Examples of primitives are `int`, `bool`, `float`, and so on. Primitive data types

Compound types are compositions of multiple primitive types. Examples include functions, tuples, lists, among others. Compound types

A function is a special type of value. Abstractly, functions convert one value (an argument) into another value, a return value, possibly of a different type. `OCaml` has syntax for functions of multiple arguments, but it's very helpful to think of `OCaml` functions as always taking one value and returning another. Because they are values, functions can be passed as arguments to other functions or can be returned as return values by other functions. function argument return value

An expression is a composition of some number of values and functions. expression

An Identifier is used to specify and keep track of a single value in `OCaml`. Prof. Shieber also calls these variables, but I find it helpful to call them identifiers, since identifiers themselves don't contain or store anything: they are instead used to *identify*, or name, values to which they are associated. For example, in the expression Identifier variables

```
let x = 5;;
```

Date: February 3, 2017.

The identifier is `x`, because it *identifies* the value 5.

Abstract syntax

Abstract syntax, and discussion of abstraction in general, refers to the *ideas* and *concepts* that we use to reason about problems and programs and that ultimately inform how we write code. Understanding the abstract syntax you are working with is at least as important, if not more important, than the concrete syntax. Ideas about abstraction are definitely the most important in this course, particularly because it's not unfair to observe that `OCaml` is not a especially widespread.

Concrete Syntax

Concrete Syntax refers to the nuts and bolts of a programming language: the semicolons, spaces and tabs, and so forth. There's a fair amount of concrete syntax to keep track of in `OCaml`, but it will be much easier to keep everything clear if you have a good grasp on the abstract ideas that are always present underneath.

type constructor
value constructor

A type constructor refers to the concrete syntax used to specify a type. Every type constructor necessarily has at least one corresponding value constructor, which is the syntax used to create values of that type.

Scope

Scope refers to the "area" of a program from which the value associated with an identifier can be reached.

Higher-order

Higher-order is used to describe functions that operate on other functions or produce other functions. To check that you are comfortable with this idea, convince yourself that every function that takes multiple arguments in OCaml is technically higher order.

anonymous

An anonymous value (or more typically, an anonymous function) is a value that is never associated with an identifier. Examples are

```
5;;
fun y -> y;;
```

Each of these is a valid value in `OCaml`, despite never being bound to an identifier. An identifier that starts with _) is also called anonymous.

More terminology next week.

3. Git Cheatsheet

Make sure that you've watched TF Brian Yu's awesome video on Git.

There are 3 (important) states that a file can exist in: unstaged, staged, and committed. If a file is unstaged, either it is not being tracked by Git, or it has been changed since the last time it was committed but has not been marked for commit yet. Files that are staged have changed since the last time they were committed, but haven't been committed yet. Staged files are "snapshotted" when they're committed, at which point they move back to being unstaged once they are edited again.

Key commands:

(1) `git add`: Move a file from unstaged to stage.
(2) `git commit <-a> <-m “message”>`: Commit the files current in the staging area (i.e take a snapshot). The *-a* flag automatically adds files that have been previously tracked into the staging area before committing. The *-m “message”* flag associates a message with the commit; if you don’t include it, you’ll be prompted to type one into a text editor pop-up.
(3) `git push`. Send your local changes to a remote repository (i.e. GitHub).
(4) `git pull`. Get changes from a remote repository.
(5) Many more – Google is your friend, and no one ever *really* knows everything there is to know about Git.

4. General Advice

(1) Read through the whole lab, even if you don’t finish it. There will often by wisdom and/or guidance and/or hints for your problem sets (i.e. on design and style) in the comments. This is really worth your time.
(2) The type system is your friend. It’s tricky at the beginning, but learning to rely on and love it will pay dividends. Thinking about and writing down the types of functions you intend to write is a great first step.
(3) Don’t starting writing concrete syntax until you understand the abstract problem. Writing symbols that are confusing to begin with will rarely make things clearer.
(4) Keep your code organized, and be systematic in the way you write it. Creating code that “looks” clear to read (uniform spacing, etc) is present you’s way of helping future you make progress.
(5) The problem sets are not (only) about writing code. This comes back to the abstract versus concrete syntax idea. None of the exercises in this course is written with “it’s very important that students know how to build *x* piece of software’ in mind. Rather, they are designed to teach underlying concepts, and identifying those concepts will help you see the value of the exercises. Don’t hesitate to ask if the connections are always clear.

5. Exercises

Exercise 1. *Write the type of the following value:*

```
# let myfun1 =
  let greet y = "Hello " ^ y
  in fun x -> string_of_float x ^ greet "World!";;
val myfun1 : float -> string = <fun>
```

□

Exercise 2. *Write the type of the following function, and say what it does:*

```
# let rec myfun2 = fun l ->
    match l with
    | [] -> [0]
    | hd :: tl -> hd :: (myfun2 tl) ;;
      val myfun2 : int list -> int list = <fun>
```

Solution: *The function takes a list of integers and appends 0 to it.*

□

Exercise 3. *Write the type of the following value:*

```
# let myfun3 =
  let greet y = "Hello " ^ y
  in fun x -> let (z, y) = x in
                string_of_float z ^ greet "World" ^ string_of_int y;;
val myfun3 : float * int -> string = <fun>
```

□

Exercise 4. *What's the type of the function* `devious` *defined below?*

```
let devious x : ??? =
    match x with
    | [] -> true
    | [_y] -> false
    | h :: n :: _tl -> (h *. n) < 0.0 ;;
```

Solution: `val devious :  float list -> bool`

□

Exercise 5. *Annotate the type of the following function as it is applied to arguments one at a time.*

```
# let myst q x z =
    if q then float_of_int x +. z
    else z ** float_of_int x ;;
# myst  ;;
- : bool -> int -> float -> float = <fun>
# myst false ;;
- : int -> float -> float = <fun>
# myst false 51 ;;
  - : float -> float = <fun>
# myst false 51 5.1 ;;
  - : float = 1.2192113050 9464412e+36
```

□

Exercise 6. *Define a function* `zip` *that converts two lists into a list of tuples.*

> ***Solution:*** *Version 1: we could start with a version like this. Some problems though: how are mismatched lengths handled?*

```
let rec zip (x : int list) (y : int list) : (int * int) list =
  match x, y with
  | [], [] -> []
  | xhd :: xtl, yhd :: ytl -> (xhd, yhd) :: (zip xtl ytl) ;;
```

> ***Solution:*** *Version 2:*

```
# let rec zip (x : int list) (y : int list) : (int * int) list =
  match x, y with
  | [], [] -> []
  | xhd :: xtl, yhd :: ytl -> (xhd, yhd) :: (zip xtl ytl)
  | _, _ -> raise (Invalid_argument "zip: mismatched list lengths") ;;
val zip : int list -> int list -> (int * int) list = <fun>
```

What's going on with the `match x,y` *magic? Remember that* `match` *statements are used to destructure values, and that we can create values out of other values. In this case, we create the tuple value* `(x,y)` *(parentheses implied by the type) and then deconstruct its component parts. This is a very handy approach!* □

Exercise 7. *Using a* `fold`*, write a function that calculates the cumulative exponentiation of a* `float list`*, using 1 as the initial value.*

```
# let cum_exp = List.fold_left ( ** ) 1.;;
val cum_exp : float list -> float = <fun>
```

Solution: *Aside: what does this function always return?*

□