OCaml

- Mini starter presentation on OCaml
- This is going to tour the OCaml language
- Starting organizing material for this, I decided to do a bunch of presentations on the language which will build on top of each other
- Basically, I'm sorry but this might be a bit beginnery.

First install OPAM (If you didn't read your email)

- OPAM is the recommended package manager for OCAML
- Except packages you can also install certain versions of the compiler
- if you didn't do this when you received the e-mail you should
- install opam via the preferred method of your distribution
- run `opam switch 4.04.0`
- run `eval $(opam config env)`

We have an OCaml

- The compiler has two (main) backends
- can compile to interpreted code
- can compile to native code
- you can not use a combination of the above

Some things to go over

- Types
- The compiler will hate you
- Work was done for better compiler-human relationships
- The current state of the compiler is to stab you in the face
- If you get more seriousy of writing OCaml in the future, a linter may help you spot loads of stupid things you might do, and the compiler being adamant of throwing you in a fiery pit.

OCaml as an expression based language

- One of the most important things to remember when working with OCaml is the following (excuse my janky bnf notation plzkthxbai):

```
expr ::= assignment
expr ::= function
assignment ::= let <var> = <expr> [in <expr2>]
```

Functions

- You can define a function like this:

```
let funny x y = x + y
```

Types

- Let's take a look at types. OCaml is a strongly statically typed language with object orientation.

Ints and Floats

- Like many languages OCaml supports integers and floats.
- Unlike many languages specific operators exist for floating point
- Something like this is okay

```
# 1 + 2 ;;
- : int = 3
```

- Things go awry when you try something like this

```
# 1.12 + 2.123;;
Error: This expression has type float but an
      expression was expected of type int
```

Ints and Floats II

- To remedy the above, we can do the following (if you look hard enough, you'll notice a sneaky '.'):

```
# 12.12 +. 12.12 ;;
- : float = 24.24
```

- Any other operation you need to do with floats, you may use any of the operators: `*.`, `+.`, `/.`, `-.`

Ints to strings / strings to ints

- You can usually use functions in the form of string_of_x (where x is usually the type you want to stringify)
- In this example, int is the type so string_of_int is our function

```
# string_of_int 12;;
- : string = "12"
```

- There exists string_of_{bool, float, ...}
- There also exists int_of_float, float_of_int

Tuples and Tuple Destructuring

- OCaml is nice and supports tuples. You can do something like this:

```
# let x = (1, 2);;
val x : int * int = (1, 2)
```

- The type signature as you may observe of tuples above, is denoted in the form of `a * b * c * d`, where {a, b, c, d} are the types of the elements of the tuple.
- If you need to access elements, you can do tuple destructuring. We use the anonymous variable '_' to discard the value we don't care about.

```
# let (a, _) = x;;
val a : int = 1
```

Lists

- The bread and butter of most functional languages!
- One thing to watch out is that unlike tuples, list elements are delimited by semicolons

```
# let my_cute_list = [1;2;3;4];;
val my_cute_list : int list = [1; 2; 3; 4]
```

Variants

- You can think of them as just Constants
- or Constant containers
- Here's an example definition:

```
type colors = Red | Green | Blue | Ultraviolent
```

Variants with Values

- You can have variants with hold values as well. You can mix them up with ones which do not contain anything as well.

```
type complicated =
        | Something of string
        | SomethingElse of int
        | SomethingVerbose of int list
        | NothingAtAll
```

Variants with Values II

- You instantiate things like that, like this (thingy):

```
let derpy_declares =
    let derpy = Something "nice" in
    let herpy = SomethingElse 12 in
    let hurrr = SomethingVerbose [1;2;3;4] in
    let durrr = NothingAtAll in
    [derpy; herpy; hurrr; durrr]
```

Built in Variants

- A common variant you will see in OCaml, as well as any other programming language that has some sort of monadic/burrito functionality is the Option monad.
- The Option monad symbolizes a value that may or may not exist, allowing functions that are to be applied onto it, to treat it as if that value were there.
- In OCaml we have the Option type as follows (built in)

```
type 'a option =
  | Some of 'a
  | None
```

Functions with optional and labelled parameters

- OCaml supports optional and labelled parameters
- Labelled are not optional - they need to be provided, but in the order you feel like

Functions Labelled Param example

- The format in func decleration is `~name_to_be_referred_from_outside:name_inside`

```
let print_person ~name:n ~surname:sur ~age:a =
      Printf.printf
        "%s, %s, lived %d years stupidly\n" sur n a

let () =
      print_person ~age:54
                   ~surname:"doe"
                   ~name:"smith";
      ()
```

Functions: Optional Param example

- Optional params may or may not be there
- The actual params are of the built in type option as we saw earlier.

```
let opt_pars ?one:one ?two:two ()=
  match one, two with
  | None   , None      ->
      print_endline "  provided no arguments"
  | Some _, None      ->
      print_endline "  provided one argument"
  | None   , Some _    ->
      print_endline "  provided one argument"
  | Some _, Some _    ->
      print_endline "  provided 2 arguments"
```

Functions: Optional Param example II

You can call the previous declaration like this:

```
let () =
  opt_pars ~one:1 ~two:2 ();
  opt_pars ~one:1 ();
  opt_pars ~two:1123 ();
  opt_pars ~two:12312 ~one:123123 ();
  opt_pars ~one:"ASDF" ();
  ()
```

Pattern Matching with Simple Values

- You can think of pattern matching as a more powerful switch case.
- Let's take a look at a simple example

```
let simple_patmatch v : unit =
  match v with
  | 0 -> print_endline "provided a 0"
  | 1 -> print_endline "provided a 1"
  | x -> Printf.printf "provided a %d :(\n" x

let () =
  let _ = List.map
                (fun x -> simple_patmatch x)
                [0; 1; 2; 3] in
  ()
```

Pattern Matching lists

- You can obtain the head and tail of a list using pattern matching

```
let decapitate ls =
  match ls with
  | head :: _ -> head
  | [] -> ""

let () =
  print_endline
    (decapitate ["head"; "body"; "legs"])
```

Imperative Programming

- It's possible to do imperative programming, but it usually looks quite ugly

```
let impy_func =
  print_endline "hallo";
  print_endline "thar";
  print_endline "much ocaml";
  print_endline "very compile";
  let feelings = "wow" in
    print_endline feelings
```

String Basics

- OCaml has a bit of a strange operator than one may be used to for string concatenation

```
let how_to_concat () =
  let str = "hell" in
  let str2 = "oh " in
  let str3 = "world" in
  Printf.printf "%s\n" (str ^ str2 ^ str3)
```

String: Length, Substrings

- Here's some basic string manipulation
- **NB**: You need to run these with `ocaml str.cma yourfile.ml`

```
let how_to_str_length () =
  let len = String.length "hello world" in
  let ret = string_of_int len in
  print_endline ret

let how_to_str_substring () =
  let str = "this is my hello" in
  let sub =
      String.sub str 0 (String.length str - 1) in
  print_endline sub
```

String: Comparison, Regex

- Nothing too crazy here

```
let how_to_regex_split () =
  let str = "comfortably,delimited, string,  thingy" in
  let splitted = Str.split (Str.regexp ", *") str in
  let _ = List.map (Printf.printf "%s|") splitted in
  print_endline ""

let how_to_compare () =
  let str = "potato" in
  let str2 = "yotato" in
  let str3 = "potato" in
  Printf.printf "compare: %s\n"
     (string_of_int (String.compare str str2));
  Printf.printf "compare: %s\n"
     (string_of_int (String.compare str str3))
```

Writing OCaml outside the interpretter

- One thing to note is that you won't be doing double semi colons
- Another to note is how you make you "main" function
- Usually you want to have a bunch of definitions and call them in your main like below
- Notice the "assignment" to unit.

```
let my_funny_func = print_endline "blarg"
let () = my_funny_func
```

- You can write your code this way and simply run your code with the `ocaml` command.
- (You can also compile your code with ocamlc or ocamlopt for native binaries)

Exercise: Writing a state machine

- With what has been presented so far, we should have enough material to write a small silly state machine
- Make it so there is a light that turns from blue to red to yellow to green to blue . . .
- Hint: Use pattern matching and variants

Exercise: Writing a state machine solution

```
type color = Blue | Red | Yellow | Green

let transition s =
  match s with
  | Blue -> Red
  | Red -> Yellow
  | Yellow -> Green
  | Green -> Red

let () = let _ =
  (transition Red |> transition
    |> transition |> transition
    |> transitio) in
    ()
```

Recursion in OCaml

- You need to explicitly tell OCaml that a function is recursive
- You define that the expression is recursive by typing `let rec` ...
- Here's an example taken from the OCaml refman (this generates a sequence provided two numbers)

```
let rec range a b =
    if a > b then []
    else a :: range (a+1) b;;
```

Recursion Exercise

- Exercise! With what you know so far, you should be able to write a `print_list` function
- Hint: reminder that you can do some imperative thingies by postfixing your expressions with a semicolon

Recursion Exercise Solution

```
let print_list ll =
  let rec _print_list l =
    match l with
    | x :: y -> Printf.printf "%d, " x; _print_list y
    | [] -> () in
  _print_list ll;
  print_endline ""

let _main =
        print_list [1;2;3;4];
        ()
```

Recursive Variants

- You can define Variants which consume themselves via pattern matching.
- This can be handy for writting parsers
- A very simple example taken from the refman

```
type expr =
        | Plus of expr * expr
        | Minus of expr * expr
        | Times of expr * expr
        | Divide of expr * expr
        | Value of string
```

Recursive Variants II

```
let rec to_string e =
  match e with
  | Plus(left, right) ->
    "(" ^ to_string left ^ " + " ^ to_string right ^ ")"
  | Minus(left, right) ->
    "(" ^ to_string left ^ " - " ^ to_string right ^ ")"
  | Times(left, right) ->
    "(" ^ to_string left ^ " * " ^ to_string right ^ ")"
  | Divide(left, right) ->
    "(" ^ to_string left ^ " / " ^ to_string right ^ ")"
  | Value v -> v
```

Recursive Variants III (Example Invokation)

```
let print_expr expr =
  print_endline(to_string expr)
let () =
  print_expr(
    Divide(
      Plus(Value "y", Value "x"),
      Value "z"))
```

Modules

- Modules can be thought of as namespaces. We have already seen some of them. `String`, `Str`, `List` are modules that give you some functionality over things.
- Defining your own modules is pretty straightforward

```
module Best = struct
  let potato = 12.12
  let potato_times x = potato *. x
end

module Worst = struct
  let potato = -12.12
end
```

Modules II

- Naturally you can call the above in the following way:

```
let () =
  print_endline (string_of_float Best.potato);
  print_endline (string_of_float (Best.potato_times 2.12));
  print_endline (string_of_float Worst.potato)
```

OCaml has Objects and Classes

- You can declare and use an object without declaring a class
- This is kind of like an anonymous class which has been instantiated...
- Let's look at class definitions

Classes

- OCaml also supports objects
- Let us examine meatbags

Classes I: Meatbags

```
class meatbag = object
  val mutable name = ""
  val mutable surname = ""
  val mutable age = 0

  method get_name = name
  method get_surname = surname
  method get_age = age

  method set_name n = name <- n
  method set_surname s = surname <- s
  method set_age a = age <- a

  method to_string =
    Printf.sprintf
      "%s, %s, %d years of stupid\n" surname name age
end
```

Classes II

- You would invoke the above and instantiate a meatbag the following way:

```
let demo () =
  let person = new meatbag in
    person#set_name "john";
    person#set_surname "doe";
    person#set_age 43;
  Printf.printf "%s" (person#to_string)
```

- But how could we refer to a method inside the class, to another method inside the same class?
- We need some sort of reference just like in C++/Java (this).

Classes III

- You do that by adding self (or it could also be named potato - basically just a variable that stores the self reference)

```
class frivolous_life = object(self)
  val mutable balance = 32.42
  method applicable_interest =
    if balance > 100.0 then 0.01 else 0.001
  method calc_interest =
    (self#applicable_interest +. 1.0) *. balance
end

let your_account_owns_you  ()=
  let account = new frivolous_life in
    Printf.printf "your negligible balance: %f\n"
      account#calc_interest

let () = demo () ; your_account_owns_you ();
```

Further interesting topics

- Splitted file (mli + ml): you can have your signatures splitted to your actual implementation (very similarly to C/C++ with c files and header files)
- ocsigen is a web framework that you can work with in OCaml. Combined with the performance you get with native code, it offers interesting venues for API development.
- MirageOS is a microkernel framework in OCaml that allows you to build small operating systems that do a specific job very well.