CS131, Spring19 – Discussion 1B

Week 1 (04/05/19)

Administrations

1. TA: Wenhao Zhang (wenhaoz@cs.ucla.edu)
2. Office hour:
    - Time: 9:30am ~ 11:30am, Monday
    - Location: 3rd floor, common area in Eng VI
3. Class Announcements
    - Piazza
    - CCLE
    - Course Website (http://web.cs.ucla.edu/classes/spring19/cs131/)
    - Seasnet account
4. HW Guidelines
    - **Done independently**
    - Questions answered on Pizza
    - HW uploaded and submitted through CCLE
    - **We will run plagiarism checker on your submission**
        - MOSS
5. HW Gradings
    - Code submission will be graded automatically with scripts
    - Lateness policy: -1%, -2%, -4%, -8%, etc.
6. Steps to get full credits:
    - Make sure your code compiles on local machine and Seasnet servers
    - Do exactly what the spec says
    - Start working on your hw early
7. HW1 dues next Tuesday (04/09/2019) by 11:55pm on **CCLE**.
8. Today's agenda
    - OCaml
        - Basic syntax and data types
        - Function writing in OCaml
        - Recursion with OCaml
        - List and tuples
        - Pattern matching
        - Debugger – `ocamlc` and `ocamldebug`

OCaml

Install from www.ocaml.org, or use the preinstall runtime on seasnet servers.

OCaml is a *functional programming language.*

> Main feature of functional programming language is that **functions** are "first-class objects". This means you can pass them into functions and treat them as if they were any other variables.

You'll see an example (write a doTwice function) on this later.

Hello World!

```
(* our first program *)
let x = print_string "Hello, World!\n";;
```

Basic syntax

- #use "hw1.ml"
  - load file into interpreter
- Line endings
  - `;;`, Double semicolon terminates expression in a global scope
  - Used for global variables and function declarations
- Comments
  - (* text *)

Basic types

- int : 31-bit signed int or 63-bit signed int
- float
- bool : written as `true` or `false`
- char : 8-bit char
- string
- unit : written as (), similar to `void` in **C**.

Variable declaration

Variables are immutable in ocaml. Once declared, there is no way to modify it.
=> Less side effects

```
(* global var declaration *)
let five = 5;;

(* local var declaration *)
```

```
let average a b =
    let sum = a + b in
        sum / 2;;
```

Type inference

No need to tell OCaml variable types. Ocaml will figure by itself.

e.g.

- let five = 5;; val five : int = 5
- let t = true;; val t : bool = true
- let pi = 3.14159;; val pi : float = 3.14159

When you use a let expression, everything to the right side of the equal side is evaluated, then that resulting value is bound to the name that you define.

e.g.

- let sum = 1 + 1;; val sum : int = 2

These examples might seem trivial, but we'll see how powerful this feature is when you write functions, and ocaml can automatically infer the types of args and rets.

Basic algebraic operations

- `+`, `-`, `*`, `/` are all operations on integers.
- `+.`, `-.`, `*.`, `/.` are all operations on floats.
- `mod` is the modulo operator.
- `>`, `>=`, `<`, `<=` are integer or float comparison operators
- `=` compares two values and returns if they are equal.
- `^` is the string concatenation operator.
- `&&`, `not`, `||` logic

Ocaml will report error when you accidentally dont comply to these contracts.

e.g.

Expression `5 * pi` will throw exceptions and wont run in ocaml.

Write functions

Recall that OCaml is a functional programming language. Functions are just vars and you can pass them around. This is achieved by **anonymous function, lambda**

```
(* syntax for a function expression *)
fun x -> x * 2;;

(* applying a function to a value *)
(fun x -> x * 2) 3;;            (* => 6 *)
```

You can also assign the function to a variable and give it a name. Again, functions are just variables.

```
let double x = fun x -> x * 2;;
(* syntactic sugar makes it more readable*)
let double x = x * 2;;
```

Once the double function is declared, call it in this way:

```
double 3;;
```

Notice: No parenthesis required around function arguments. Parenthesis can still be used to assign precedence of evaluation.

e.g.

- double (1+2) (* => 6 *)
- (double 1) + 2 (* => 4 *)

How would we write a function that takes two ints as arguments and returns the product of the two ints?

```
(* syntactic sugar *)
let mul x y =
    x * y;;

(full version)
let mul x = fun y -> x * y;;

(* more explicitly on type of args *)
let mul (x: int) = fun (y: int) - > x * y;;
```

The type of this `mul` func is:

```
val mul: int -> int -> int = <fun>
```

Note: OCaml function can only takes one argument, the simplified version of `mul` is just a syntactic sugar. Please feel free to read more on **function currying** or **higher order function**.

What happens if we call this function with a single argument?

e.g.

- what would `mul 2` return?

```
int -> int = <fun>
```

This is a simple example of function currying by partially apply the args. It's ok if you dont know about function currying for now, and we will cover this important concept next week. For now, just understand that we can reuse existing functions to define new functions. `mul 2` is equivalent to `double` func we defined previously.

Let's see something more advanced.

Q: How would we write a function that takes two arguments, some function f and some variable x and applies the function to x twice.

```
let doTwice f x = f (f x);;
```

What is the type of this?

```
val doTwice : (`a -> `a) -> `a -> `a = <fun>
```

How to call `doTwice`?

```
doTwice double 2;;  (* => 8*)
```

What is the type if we just pass in func `double` into `doTwice`?

```
doTwice double;;
int ->  int = <fun>
```

Recursive funcs

We use the `let rec` keyword to tell the compiler that the function is a recursive function.

```
(* an example of factorial function*)
let rec f n = if n == 1 then 1 else n * f (n-1);;
f 3                                       (* => 6 *)
```

Lists and tuples

- Lists are homogenous. In other words, every element must be of the same type.
- Lists are immutable.
- Please read the API documents on the **List** module as it might help with you with the list manipulation.

e.g.

```
(* empty list* )
[];;

(* list of ints *)
[1;2;3;4;5];;
```

We can use `::` to build lists.

e.g.

```
1::[]        (* => [1]*)
1::2::[]     (* => [1;2]*)
```

We can also concatenate two lists together using `@`

e.g.

```
[1;2] @ [3;4]    (* => [1;2;3;4] *)
```

Excercise:

Q: Write a function that takes two arguments, a variable x and a list l. This func will insert x at the begining of l.

```
let insertAtBegining x l = x::l;;
```

Tuples are formed by putting parenthesis with commas separating multiple values. Tuples are heterogeous.

e.g.

```
("3", 3)
("3", 4, 0.5, "turtles")

(* get the first item in tuple with `fst` *)
fst ("1", 2)      (* => "1" *)
snd ("1", 2)      (* => 2 *)
```

Note that `fst` and `snd` only work on tuples with size 2.

Pattern Matching

syntax

```
match v with
| _ -> ...
| patter -> ...
```

`_` is the placeholder, and it matches with everything.

An example of pattern matching,

```
(* example of pattern matching; multiply two items in a tuple*)
let multTuple t = match t with
| (a, b) -> a * b;;


(* call multTuple*)
multTuple (2,3);;    (* => 6 *)
```

Pattern matching can be used for iterate over lists. Here is an example of sum up all items in a list using pattern matching.

```
let rec sumList l = match l with
| f :: r -> f + (sumList r)
| [] -> 0;;
```

Exercise

Q: Write a recursive function that iterates through a list of tuples (int * int) and adds the second value of the tuple if the first value of the tuple is the int 0.

HW1

Grammars

- Symbol
  - Terminal: A symbol which you cannot replace with other symbols
  - Non-terminal: A symbol which you can replace with other symbols
- Rule
  - From a non terminal symbol, derive a list of symbols
- Grammar
  - A starting symbol, and a set of rules that describe what symbols can be derived from a non terminal symbol

Example of a simple grammar:

- symbols: S, A, B, a, b
- Non-terminals: S, A, B
- Terminals: a, b
- Starting symbols: S

Rules: - S -> A - S -> B - A ->aA - A -> a - B -> bB - B -> b

How to derive: aaa

Debug your OCaml code

There are two ways to debug your ocaml code

- Trace function, `trace`
- OCaml debugger which allows analysing programs compiled with `ocamlc`

e.g.

1. we compile the program in debug mode `ocamlc -g hw1.ml`
2. we launch the debugger: `ocamldebug a.out`
3. set break point in hw1.ml:
   - set a break point at line 3 (ocd) break @hw1 3
   - run program to the break point (ocd) r

- print values (ocd) p x

The code I tested using debugger during discussion is here:

```
(* demo code for debugger *)
(*debug using ocamlc*)
let rec member (x:int) = function
[] -> false
| h::t -> if x=h then
                true
          else
                member x t;;

member 1 [1;2;3]
```