Yunke Xiao

COMP 302

1 Midterm 1

1.1 OCaml

OCaml is an elegant combination of a few key language features that have been developed over the past 40 years:

- *Garbage Collection*
- *First-Class Functions*
- *Stat Type-Checking*
- *Parametric Polymorphism*
- *Immutable Programming*
- *Automatic Type Inference*
- *Algebraic Datatypes and Pattern Matching*

1.2 Boolean, Types and Expressions

Similar to many other languages, the main OCaml types are string, boolean and integer

```
# 3;;
int = 3
# "Hello";;
string = "Hello"
# true;;
bool = true
```

Float type also exists in OCaml and float operations are different than integer operations; they're followed by a period

```
# 3.0 /. 3.0;;
float = 1.0
# 1.0 + 2.0;;
float = 3.0
```

Arithmetic operations are not *overloaded* in OCaml, as it needs to be able to determine variable and return types at runtime through looking at the syntax of the code. As for boolean operators

```
(bool) = (bool) : equal
(bool) == (bool) : not equal
not (bool) : not
(bool) || (bool) : or
(bool) && (bool) : and
```

Once we finish writing an expression, we need to type `;;` to tell the toplevel that it should evaluate an expression. Once an expression is evaluated, the toplevel prints the type of the result as well as the result itself.

Other ways to use booleans are in `For`, `While` and `If` statements

```
# if (1 = 1)
    then "equal"
    else "not equal";;
string = "equal"
# for i = 0 to 5 do
~~~ DO SOMETHING ~~~
  done
# while i < 5 do
~~~ DO SOMETHING ~~~
  done
```

1.3 Variables, Bindings and Functions

Ocaml is a call-by-value language

```
# let a = 3.0 * 2.0;;
val a : float = 6.0
```

This means we bind values to variables and not references. Once a value is bound to a variable, we can only *overshadow* it, not update its reference

```
# let m = 3;;
val m : int = 3
# let n = m * m;;
val n : int = 9
# let m = n* n;;
val m : int = 81
```

We use a let-expression that has the following structure:

```
let <name> = <expression 1> in <expression 2>
```

We can use `<name>` in `<expression 2>` with `<expression 1>` bound to `<name>`, however once `<expression 2>` is done evaluating, the binding between `<name>` and `<expression 1>` is deleted. We say that the scope of `<name>` ends after `<expression 2>`

```
# let v = 4;;
val k : int = 4
# let v = 3 in v * v;;
int = 9
# k;;
int = 4
```

Functions are also declared using `let`, however, unlike many other languages, its arguments are separated by spaces, and not parentheses and commas:

```
# let add x y = x + y;;
val add : int -> int -> int = <fun>
# add 3 4;;
int = 7
```

Additionally, variables must start with a lowercase letter or an underscore and functions describe their arguments and return values using `->`. In the example above, `int -> int ->` describes the first two inputs of the function, which are integers, and the final `int` describes its return value. The `<fun>` indicates that `add` is a function. However, since OCaml has first-class functions, you can write functions with other functions as arguments

```
# let subs x y = x - y;;
val subs : int -> int -> -> int = <fun>
# let abs_subs f x y =
    if x > y then f x y
    else f y x;;
val abs_subs : ('a -> 'a -> 'b) -> a' -> a' -> 'b = <fun>
# abs_subs subs 5 3;;
int = 2
# abs_subs subs 3 10;;
int = 8
```

Since OCaml doesn't know exactly what type certain variables are in `abs_subs`, it assigns `'a` and `'b` as two distinct types. It knows that since we compare `x` and `y`, therefore they must be the same type and the function returns a type `'b` that could either be the same type as `'a` or not. Since we use `f` with both `x` and `y` as arguments, `f`'s type is represented with `('a -> 'a -> 'b)`, with the parentheses distinguishing `f` from `abs_subs`'s other arguments.

To help improve the readability of our code, we can manually type our arguments using the symbole `:`, but they don't change how OCaml infers data types. It only serves as useful documentation

```
# let sum_if_true (test: int -> bool) (x : int) (y : int) : int =
    (if test x then x else 0) + (if test y then y else 0);;
val sum_if_true : (int -> bool) -> int -> int -> int = <fun>
```

When using functions, we need to pad our function definition with the keyword `rec` if our function is a recursive function, otherwise OCaml will run into errors

```
# let rec recursive_add x =
~~~ RECURSIVE FUNCTION ~~~
```

1.4 Pattern Matching

Pattern matching is a very useful tool in OCaml and functions similarly to `switch` statements in many other popular languages

```
# let bool_to_string a_bool =
    match a_bool with
    | true -> "true"
    | false -> "false";;
val bool_to_string : bool -> string = <fun>
# bool_to_string true
string = "true"
```

Pattern matching is initialized with `match <value> with` followed by several `| <value> -> <expression>` Pattern matching must be exhaustive, so wildcard _ is used to help represent any value

```
# let int_to_bool a_int =
    match a_int with
    | 0 -> false
    | _ -> true;;
val int_to_bool : int -> bool = <fun>
```

The example above would therefore return `false` only if the argument is equal to `0`, otherwise the function returns `true`.

The first data structure we'll see are tuples. They're groupings of different variables that can be of different types

```
# let tuple = (3, "three")
val tuple: int * string
```

Tuples are very useful and must not be confused with functions calls

1.5 Datatypes

OCaml allows us to create our own datatypes beyond the basic datatypes using the keywords `type`.

```
# type point2d = (x : int, y : int)
type point2d = (x : int, y : int)
# (0, 1)
point2d = (0, 1)
```

Types are not limited to the existing datatypes, so they don't have to be a collection of integers, strings, etc.

```
# type suit = Clubs | Spades | Hearts | Diamonds
type suit = Clubs | Spades | Hearts | Diamonds
# let a = Clubs
val a : suit = Clubs
```

You can even build types upon types

```
# type suit = Clubs | Spades | Hearts | Diamonds;;
    type suit = Clubs | Spades | Hearts | Diamonds
# type rank = Two | Three | Four | Five | Six | Seven | Eight |
              Nine | Ten | Jack | Queen | King | Ace;;
    type rank = Two | Three | Four | Five | Six | Seven |
                Eight | Nine | Ten | Jack | Queen | King | Ace
# type card = rank * suit;;
type card = rank * suit
# let ace_diamonds = (Ace, Diamonds);;
val ace_diamonds : suit = Ace * Diamonds
```

We can define constructors to our types (continuing from code above)

```
# type hand = Empty | Hand of card * hand
type hand = Empty | Hand of card * hand
# let hand0 = Empty
val hand0 : hand = Empty
# let hand1 = Hand(card(Jack, Hearts), hand0)
val hand1 : hand = (Jack * Hearts) * Empty
```

The keyword `of` determines the arguments of the constructor and consequently the type's content. When matching types, we can use their constructors in order to access their arguments

```
# let rec extract (s:suit) (h:hand) : hand =
    match h with
    | Empty -> Empty
    | Hand((r',s'), h') ->
        if s = s then Hand((r',s'), extract s h') else extract s h'
```

The function above returns a hand that contains all cards of a given suit.

1.6 Lists

Lists in OCaml must contain values of the same type and can be constructed in two different ways

```
# let string_list = ["hello";"world";"!"];;
val string_list : string list = ["hello"; "world; "!"]
# let int_list = 0::1::2::3::4::[];;
let int_list : int list = [0;1;2;3;4]
```

The symbol `::` is used to add elements to the beginning of a list. If we want to merge two lists together, we can use the symbol `@`

```
# [1;2;3]@[4;5;6]
int list = [1;2;3;4;5;6]
```

When pattern matching with lists, we can use `::` to isolate the head of the list from its tail

```
# let rec is_five_in_list (li : int list) : bool =
    match l with
    | [] -> false
    | hd::tl -> if head = 5 then true else is_five_in_list tl
val is_five_in_list : int list -> bool = <fun>
```

In the example above, we take a list and recursively check if the head of the list is equal to 5. If it reaches `[]`, that means none of the elements in that list are equal to 5, therefore you return `false`.

1.7 Options and Fun

Another common data structure in OCaml is the option. It is used to express that a value might or might be present

```
# let divide x y =
    if y = 0 then None else Some(x/y);
val divide : int -> int -> int -> int option = <fun>
```

Paired with pattern matching, we can see options as specialized lists that either have zero or one elements.

Anonymous functions are functions declared without being named. They are declared using the keyword `fun`

```
# (fun x -> x + 1);;
int -> int = <fun>
```

In order to evaluate anonymous functions, we must it arguments at the end of its declaration

```
# (fun x -> x + 1) 5;;
int = 6
```

Anonymous functions are often passed into other functions, namely iteration functions like `List.map`

```
# List.map (f x -> x +1) [0;1;2];;
int list = [1;2;3]
```

1.7.1 Higher Order Functions

Higher order functions are functions that take in other functions as arguments.

```
# let rec map f l = match l with
    | [] -> []
    | hd::tl -> (f h)::(map f tl);;
val map : ('a -> 'b) -> 'b list
```

The example above functions similarly to List.map, and represents one of the most common uses of higher order functions. Other popular uses of higher order functions include `filter` and `fold_right`/`fold_left`(see `hofun-1.ml`). Another particular use case of higher order functions is currying/uncurrying

```
# let curry f = (fun x y -> f(x,y));;
val curry : 'a * 'b -> 'c
# let sum(x,y) = x + y;;
val sum : int * in -> int
# curry f 10 5;;
int = 15
```

An important application of higher-order functions and the ability to return functions lies in partial evaluation

```
# let funky_plus x y = x * x + y;;
val funky_plus : int -> int -> int = <fun>
# let plus9 = funky_plus 3;;
val plus9 : int -> int = <fun>
# plus9 3;;
int = 12

# let funky x = let r = x * x in fun y -> r + y;;
val funky : int -> int -> int = <fun>
# let plus9 = funky 3;;
val plus9 : int -> int = <fun>
# plus9 1
int = 10
```

This allows us to easily reuse and create new similar code.