Recitation 1: Introduction to OCaml

CS3110 Fall 2013

1 Lecture Outline

1. Style
2. OCaml Overview
3. Basic Expressions and Types
4. Options and Higher Order Functions

2 Style

This course emphasizes clean, efficient programming, which encompasses style. Please consult the CS3110 style guide often early on in the course so you will become familiar with the guidelines.
Some guidelines that we want to emphasize:

- 80 character limit per line
  In the terminal, a line with 80 characters will either wrap or cut off from view. Likewise, when we print out your problem sets to grade, lines with more than 80 characters get cut off, making it very difficult for us to grade your problem sets fairly. You will be penalized for this, so don't do it.

- Use spaces instead of tabs
  It easier to format your code when you use spaces because every text editor is unique in its tab spacing. Along with that,when we print out problem sets to grade, tabs appear as special characters instead of spaces. Simply, it is better to just use spaces instead of tabs.

3 OCaml Overview

Tuples

Every function in OCaml takes exactly one value and returns exactly one result. The advantage of always taking one argument and returning one result is that the language is extremely uniform. One of the features of OCaml that allows us to keep this uniformity but allowing us more flexibility with arguments and returns is the tuple.

A simple example of a tuple is:

```
let (y,x) = (1,2)
```

We can extract information quite easily from a tuple.

```
let first p =
  match p with
  |(a,_) -> a

let second p =
  match p with
  |(_,b) -> b
```

We can have a tuple of any length as well. We call these n-tuples (with n being the length of the tuple).

```
let five_tuple_match p =
  match p with
  |(_,_,c,_,_) -> c
```

There is a more succinct way to pattern match tuples.

```
let five_tuple_match2 (a,b,c,d,e) = c
```

You can even use wild cards with this pattern match.

```
let five_tuple_match3 (_,_,c,_,_) = c
```

4 Basic Expressions and Types

Expressions evaluate to values. Values can be classified according to their type:

Type	Example values
int	0,1,2,-1,-2
bool	true, false
float	3.141592, 1.618034
string	"Hello world!","xyzzy"
char	'A','Z'

Let us start looking at simple expressions, with the following syntax:

```
e ::= c | unop e | e1 binop e2 | if e1 then e2 else e3 | (e)
```

where c are constants (the values described above), unop are unary operations, binop are binary operations. We expressed this syntax using Backus-Naur Form (BNF), a common notation for the grammar of computer languages.

Unary operators include:

Operator	Meaning
-	take an int (or a float) and return its negation
not	take a bool and return its negation

Binary operators include:

Operator	Meaning
+, -, *, /	take two ints and return their sum, difference, product, or quotient
+. , -. , *. , /.	take two floats and return their sum, difference, product, or quotient
mod	take two ints and return their integer remainder
>, >=, <, <=	take two ints (or two floats) and return their comparison
=	take two values and return whether they are equal
^	take two strings and return their concatenation into a new string

Expressions are transformed into values by the application of evaluation rules. The evaluation rules for simple expressions are:

- Constants are already values.
- Unary and binary operators first evaluate their arguments to values, and then perform the operation.

- A conditional `if e1 then e2 else e3` evaluates `e1` to a value (of type bool): if it is `true`, `e2` is evaluated, if it is `false`, `e3` is evaluated.

Evaluation of operators only makes sense if the types of the operands agree. For example, the + operator is defined if both operands are integers, but adding an integer to a boolean is meaningless. So type checking is performed to ensure that expressions are meaningful, and this requires giving a type to every expression. How do we determine the type of an expression?

Every constant has a type (42 has type int, true has type bool, etc). Operators also have types (which we gave informally above); in OCaml syntax, these types are effectively:

```
-      :  int -> int                    float -> float
not    :  bool -> bool
(+)      :  int -> int -> int
(+.)     :  float -> float -> float
>      :  int -> int -> bool            float -> float -> bool
^      :  string -> string -> string
```

Every operator specifies the types of values it takes and the type of the value it returns.

Now we can give rules for determining the types of expressions. The principle embodied in these rules is:

The type of an expression is always the type of its result.

- For expression `unop e`, if `e` has type `t1` and `unop` has type `t1 -> t2`, then `unop e` has type `t2`.

- For expression `e1 binop e2`, if `e1` has type `t1`, and `e2` has type `t2`, and `binop` has type `(t1 * t2) -> t3`, then `e1 binop e2` has type `t3`.

- For conditional `if e1 then e2 else e3`, if `e1` has type `bool`, and then if `e2` and `e3` both have the same type `t`, then the conditional itself has type `t`.

  - Why must `e2` and `e3` have the same type?
    From the compiler's point of view, either branch could be taken. If the then branch is taken, the resulting value will have the same type as `e2`. Similarly, the else branch produces a value of the same type as `e3`. So the type of the conditional must somehow generalize the types of `e2` and `e3`. A safe way to guarantee that

this is possible is to require that both branches return the same type.

If the conditions in these rules are not satisfied by an expression, then it is not possible to give a type to that expression. It does not type check, and the compiler rejects it as a program. This prevents the evaluation of meaningless expressions. It is important to do so, because such programming errors could otherwise cause run-time crashes.

5 Options and Higher Order Functions

Ocaml does not have null. To work around this, there is a type called option.

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

Can we just treat options as any other type? Such as, can we add two int options (e.g. `Some 4 + Some 5`)?

Sadly, we cannot do that. However, operations on options are possible and are not difficult!

We can write

```
let option_plus a b =
  match a with
  | None -> None
  | Some x -> begin
    match b with
    | None -> None
    | Some y -> Some (x + y)
  end
```

This gives us...

`option_plus (None) (Some 5) = None`

`option_plus (Some 4) (None) = None`

`option_plus (Some 4) (Some 5) = Some 9`

This code is longer than what we want, so let's clean up the match statement

```
let option_plus1 a b =
  match a,b with
  | Some x, Some y -> Some (x + y)
  | _ , _ -> None
```

Now let's multiply two int options.

```
let option_mult a b =
  match a,b with
  | Some x, Some y -> Some (x * y)
  | _ , _ -> None
```

```
option_mult (None) (Some 5) = None
option_mult (Some 4) (None) = None
option_mult (Some 4) (Some 5) = Some 20
```

Wait, these methods look very similar. The only difference is the operator... Is there a way we can make this more efficient?
Yes, we can by using abstraction with a higher order function.

```
let option_apply f a b =
  match a,b with
  | Some x, Some y -> Some (f x y)
  | _ , _ -> None
```

Now using `option_apply`, we can redefine `option_plus` and `option_mult`.

```
let option_plus2 = option_apply (+)
```

```
let option_mult2 = option_apply ( * )
```

NOTE: In order to supply the multiplication operation, make sure to have spaces on either side to distinguish it from a comment