SNU 4541.574
Programming Language Theory

Tour of OCaml Programming

OCaml and this course

The material in this course is mostly conceptual and mathematical.

However:

- some of the ideas we will encounter are easier to grasp if you can “see them work”
- experimenting with small implementations of programming languages is an excellent way to deepen intuitions

For these purposes, we will use the OCaml language.

OCaml is a large and powerful language. For present purposes, though, we can concentrate just on the “core” of the language, ignoring most of its features. In particular, we will not need modules or objects.

Higher-order & Typed Programming

OCaml is a *HOT* programming language — i.e., a language in which the *functional programming style* is the dominant idiom.

Other well-known higher-order languages include Lisp, Scheme, Haskell, and Standard ML.

Higher-order & Typed Programming

The HOT style can be described as a combination of...

- *persistent* data structures (which, once built, are never changed)
- *recursion* as a primary control structure
- heavy use of *higher-order functions* (functions that take functions as arguments and/or return functions as results)

Imperative languages, by contrast, emphasize...

- *mutable* data structures
- *looping* rather than recursion
- *first-order* rather than higher-order programming (though many object-oriented design patterns involve higher-order idioms—e.g., Subscribe/Notify, Visitor, etc.)

Computing with Expressions

OCaml is an *expression language*. A program is an expression. The "meaning" of the program is the value of the expression.

```
# 16 + 18;;
- : int = 34

# 2*8 + 3*6;;
- : int = 34
```

The top level

OCaml provides both an interactive *top level* and a *compiler* that produces standard executable binaries. The top level provides a convenient way of experimenting with small programs.

The mode of interacting with the top level is typing in a series of expressions; OCaml *evaluates* them as they are typed and displays the results (and their types). In the interaction above, lines beginning with # are inputs, and lines beginning with - are the system's responses. Note that inputs are always terminated by a double semicolon.

Giving things names

The `let` construct gives a name to the result of an expression so that it can be used later.

```
# let inchesPerMile = 12*3*1760;;
val inchesPerMile : int = 63360

# let x = 1000000 / inchesPerMile;;
val x : int = 15
```

Functions

```
# let cube (x:int) = x*x*x;;
val cube : int -> int = <fun>

# cube 9;;
- : int = 729
```

We call `x` the *parameter* of the function `cube`; the expression `x*x*x` is its *body*. The expression `cube 9` is an *application* of `cube` to the *argument* 9.
The *type* printed by OCaml, `int->int` (pronounced "`int` arrow `int`") indicates that `cube` is a function that should be applied to an integer argument and that returns an integer.
Note that OCaml responds to a function declaration by printing just `<fun>` as the function's "value."

Here is a function with two parameters:

```
# let sumsq (x:int) (y:int)  = x*x + y*y;;
val sumsq : int -> int -> int = <fun>

# sumsq 3 4;;
- : int = 25
```

The type printed for `sumsq` is `int->int->int`, indicating that it should be applied to two integer arguments and yields an integer as its result.
Note that the syntax for invoking function declarations in OCaml is slightly different from languages in the C/C++/Java family: we write `cube 3` and `sumsq 3 4` rather than `cube(3)` and `sumsq(3,4)`.

The type boolean

There are only two values of type `boolean`: `true` and `false`. Comparison operations return boolean values.

```
# 1 = 2;;
- : bool = false

# 4 >= 3;;
- : bool = true
```

`not` is a unary operation on booleans.

```
# not (5 <= 10);;
- : bool = false

# not (2 = 2);;
- : bool = false
```

Conditional expressions

The result of the conditional expression `if B then E1 else E2` is either the result of `E1` or that of `E2`, depending on whether the result of `B` is `true` or `false`.

```
# if 3 < 4 then 7 else 100;;
- : int = 7

# if 3 < 4 then (3 + 3) else (10 * 10);;
- : int = 6

# if false then (3 + 3) else (10 * 10);;
- : int = 100

# if false then false else true;;
- : bool = true
```

Defining things inductively

In mathematics, things are often defined inductively by giving a "base case" and an "inductive case." For example, the sum of all integers from 0 to n or the product of all integers from 1 to n:

$$\begin{array}{rcll} \mathrm{sum}(0) & = & 0 & \\ \mathrm{sum}(n) & = & n + \mathrm{sum}(n-1) & \text{if} \quad n \geq 1 \\ \\ \mathrm{fact}(1) & = & 1 & \\ \mathrm{fact}(n) & = & n * \mathrm{fact}(n-1) & \text{if} \quad n \geq 2 \end{array}$$

It is customary to extend the factorial to all non-negative integers by adopting the convention $\mathrm{fact}(0) = 1$.

Recursive functions

We can translate inductive definitions directly into *recursive* functions.

```
# let rec sum(n:int) = if n = 0 then 0 else n + sum(n-1);;
val sum : int -> int = <fun>

# sum(6);;
- : int = 21

# let rec fact(n:int) = if n = 0 then 1 else n * fact(n-1);
val fact : int -> int = <fun>

# fact(6);;
- : int = 720
```

The `rec` after the `let` tells OCaml this is a recursive function — one that needs to refer to itself in its own body.

Making Change

Another example of recursion on integer arguments. Suppose you are a bank and therefore have an “infinite” supply of coins (pennies, nickles, dimes, and quarters, and silver dollars), and you have to give a customer a certain sum. How many ways are there of doing this?
For example, there are 4 ways of making change for 12 cents:

12 pennies
1 nickle and 7 pennies
2 nickles and 2 pennies
1 dime and 2 pennies

We want to write a function `change` that, when applied to 12, returns 4.

Making Change – continued

To get started, let's consider a simplified variant of the problem where the bank only has one kind of coin: pennies.
In this case, there is only one way to make change for a given amount: pay the whole sum in pennies!

```
# (* No. of ways of paying a in pennies *)
  let rec changeP (a:int) = 1;;
```

That wasn't very hard.

Making Change – continued

Now suppose the bank has both nickels and pennies.
If a is less than 5 then we can only pay with pennies. If not, we can do one of two things:

- Pay in pennies; we already know how to do this.
- Pay with at least one nickel. The number of ways of doing this is the number of ways of making change (with nickels and pennies) for a-5.

```
# (* No. of ways of paying in pennies and nickels *)
  let rec changePN (a:int) =
    if a < 5 then changeP a
    else changeP a + changePN (a-5);;
```

Making Change – continued

Continuing the idea for dimes and quarters:

```
# (* ... pennies, nickels, dimes *)
  let rec changePND (a:int) =
    if a < 10 then changePN a
    else changePN a + changePND (a-10);;

# (* ... pennies, nickels, dimes, quarters *)
  let rec changePNDQ (a:int) =
    if a < 25 then changePND a
    else changePND a + changePNDQ (a-25);;
```

Finally:

```
# (* Pennies, nickels, dimes, quarters, dollars *)
  let rec change (a:int) =
    if a < 100 then changePNDQ a
    else changePNDQ a + change (a-100);;
```

Some tests:

```
# change 5;;
- : int = 2

# change 9;;
- : int = 2

# change 10;;
- : int = 4
```

...

```
# change 29;;
- : int = 13
# change 30;;
- : int = 18

# change 100;;
- : int = 243

# change 499;;
- : int = 33995
```

Lists

One handy structure for storing a collection of data values is a *list*. Lists are provided as a built-in type in OCaml and a number of other popular languages (e.g., Lisp, Scheme, and Prolog—but not, unfortunately, Java).

We can build a list in OCaml by writing out its elements, enclosed in square brackets and separated by semicolons.

```
# [1; 3; 2; 5];;
- : int list = [1; 3; 2; 5]
```

The type that OCaml prints for this list is pronounced either "integer list" or "list of integers".

The empty list, written `[]`, is sometimes called "nil."

The types of lists

We can build lists whose elements are drawn from any of the basic types (`int`, `bool`, etc.).

```
# ["cat"; "dog"; "gnu"];;
- : string list = ["cat"; "dog"; "gnu"]

# [true; true; false];;
- : bool list = [true; true; false]
```

We can also build lists of lists:

```
# [[1; 2]; [2; 3; 4]; [5]];;
- : int list list = [[1; 2]; [2; 3; 4]; [5]]
```

In fact, for *every* type `t`, we can build lists of type `t list`.

Lists are homogeneous

OCaml does not allow different types of elements to be mixed within the same list:

```
# [1; 2; "dog"];;
Characters 7-13:
This expression has type string list but is here used
with type int list
```

Constructing Lists

OCaml provides a number of built-in operations that return lists. The most basic one creates a new list by adding an element to the front of an existing list. It is written `::` and pronounced “cons” (because it *cons*tructs lists).

```
# 1 :: [2; 3];;
- : int list = [1; 2; 3]

# let add123 (l: int list) = 1 :: 2 :: 3 :: l;;
val add123 : int list -> int list = <fun>

# add123 [5; 6; 7];;
- : int list = [1; 2; 3; 5; 6; 7]

# add123 [];;
- : int list = [1; 2; 3]
```

Some recursive functions that generate lists

```
# let rec repeat (k:int) (n:int) =  (* A list of n copies c
    if n = 0 then []
    else k :: repeat k (n-1);;

# repeat 7 12;;
- : int list = [7; 7; 7; 7; 7; 7; 7; 7; 7; 7; 7; 7]

# let rec fromTo (m:int) (n:int) = (* The numbers from m to
    if n < m then []
    else m :: fromTo (m+1) n;;

# fromTo 9 18;;
- : int list = [9; 10; 11; 12; 13; 14; 15; 16; 17; 18]
```

Constructing Lists

Any list can be built by “consing” its elements together:

```
-# 1 :: 2 :: 3  :: 2 :: 1 :: [] ;;;
- : int list = [1; 2; 3; 2; 1]
```

In fact, `[` x_1; x_2; $\ldots$; x_n `]` is simply a shorthand for

$$x_1 \texttt{ :: } x_2 \texttt{ :: } \ldots \texttt{ :: } x_n \texttt{ :: []}$$

Note that, when we omit parentheses from an expression involving several uses of `::`, we associate to the right—i.e., `1::2::3::[]` means the same thing as `1::(2::(3::[]))`. By contrast, arithmetic operators like `+` and `-` associate to the left: `1-2-3-4` means `((1-2)-3)-4`.

Taking Lists Apart

OCaml provides two basic operations for extracting the parts of a list.

- `List.hd` (pronounced "head") returns the first element of a list.

```
# List.hd [1; 2; 3];;
- : int = 1
```

- `List.tl` (pronounced "tail") returns everything *but* the first element.

```
# List.tl [1; 2; 3];;
- : int list = [2; 3]
```

More list examples

```
# List.tl (List.tl [1; 2; 3]);;
- : int list = [3]

# List.tl (List.tl (List.tl [1; 2; 3]));;
- : int list = []

# List.hd (List.tl (List.tl [1; 2; 3]));;
- : int = 3
```

More list examples

```
# List.hd [[5; 4]; [3; 2]];;
- : int list = [5; 4]

# List.hd (List.hd [[5; 4]; [3; 2]]);;
- : int = 5

# List.tl (List.hd [[5; 4]; [3; 2]]);;
- : int list = [4]
```

Modules – a brief digression

Like most programming languages, OCaml includes a mechanism for grouping collections of definitions into *modules*.

For example, the built-in module `List` provides the `List.hd` and `List.tl` functions (and many others). That is, the name `List.hd` really means "the function `hd` from the module `List`."

Recursion on lists

Lots of useful functions on lists can be written using recursion. Here's one that sums the elements of a list of numbers:

```
# let rec listSum (l:int list) =
    if l = [] then 0
    else List.hd l + listSum (List.tl l);;

# listSum [5; 4; 3; 2; 1];;
- : int = 15
```

Consing on the right

```
# let rec snoc (l: int list) (x: int) =
    if l = [] then x::[]
    else List.hd l  :: snoc(List.tl l) x;;
val snoc : int list -> int -> int list = <fun>

# snoc [5; 4; 3; 2] 1;;
- : int list = [5; 4; 3; 2; 1]
```

Reversing a list

We can use `snoc` to reverse a list:

```
# (* Reverses l -- inefficiently *)
  let rec rev (l: int list) =
    if l = [] then []
    else snoc (rev (List.tl l)) (List.hd l);;
val rev : int list -> int list = <fun>

# rev [1; 2; 3; 3; 4];;
- : int list = [4; 3; 3; 2; 1]
```

Why is this inefficient? How can we do better?

A better rev

```
# (* Adds the elements of l to res in reverse order *)
  let rec revaux (l: int list) (res: int list) =
    if l = [] then res
    else revaux (List.tl l) (List.hd l :: res);;
val revaux : int list -> int list -> int list = <fun>

# revaux [1; 2; 3] [4; 5; 6];;
- : int list = [3; 2; 1; 4; 5; 6]

# let rev (l: int list) = revaux l [];;
val rev : int list -> int list = <fun>
```

Tail recursion

The `revaux` function

```
let rec revaux (l: int list) (res: int list) =
  if l = [] then res
  else revaux (List.tl l) (List.hd l :: res);;
```

has an interesting property: the *result* of the recursive call to `revaux` is also the result of the whole function. I.e., the recursive call is the *last* thing on its “control path” through the body of the function. (And the other possible control path does not involve a recursive call.)
Such functions are said to be *tail recursive*.

Tail recursion

It is usually fairly easy to rewrite a recursive function in tail-recursive style. For example, the usual factorial function is not tail recursive (because one multiplication remains to be done after the recursive call returns):

```
# let rec fact (n:int) =
    if n = 0 then 1
    else n * fact(n-1);;
```

We can transform it into a tail-recursive version by performing the multiplication *before* the recursive call and passing along a separate argument in which these multiplications "accumulate":

```
# let rec factaux (acc:int) (n:int) =
    if n = 0 then acc
    else factaux (acc*n) (n-1);;
# let fact (n:int) = factaux 1 n;;
```

Basic Pattern Matching

Recursive functions on lists tend to have a standard shape: we test whether the list is empty, and if it is not we do something involving the head element and the tail.

```
# let rec listSum (l:int list) =
    if l = [] then 0
    else List.hd l + listSum (List.tl l);;
```

OCaml provides a convenient *pattern-matching* construct that bundles the emptiness test and the extraction of the head and tail into a single syntactic form:

```
# let rec listSum (l: int list) =
    match l with
      []   -> 0
    | x::y -> x + listSum y;;
```

Pattern matching can be used with types other than lists. For example, here it is used on integers:

```
# let rec fact (n:int) =
    match n with
      0 -> 1
    | _ -> n * fact(n-1);;
```

The _ pattern here is a *wildcard* that matches any value.

Complex Patterns

The basic elements (constants, variable binders, wildcards, `[]`, `::`, etc.) may be combined in arbitrarily complex ways in `match` expressions:

```
# let silly l =
    match l with
      [_;_;_] -> "three elements long"
    | _::x::y::_::_::rest ->
        if x>y then "foo" else "bar"
    | _ -> "dunno";;
val silly : int list -> string = <fun>
# silly [1;2;3];;
- : string = "three elements long"
# silly [1;2;3;4];;
- : string = "dunno"
# silly [1;2;3;4;5];;
- : string = "bar"
```

Type Inference

One pleasant feature of OCaml is a powerful *type inference* mechanism that allows the compiler to calculate the types of variables from the way in which they are used.

```
# let rec fact n =
    match n with
      0 -> 1
    | _ -> n * fact(n-1);;
val fact : int -> int = <fun>
```

The compiler can tell that `fact` takes an integer argument because `n` is used as an argument to the integer `*` and `-` functions.

Similarly:

```
# let rec listSum l =
    match l with
      []   -> 0
    | x::y -> x + listSum y;;
val listSum : int list -> int = <fun>
```

Polymorphism (first taste)

```
# let rec length l =
    match l with
      []   -> 0
    | _::y -> 1 + length y;;
val length : 'a list -> int = <fun>
```

The `'a` in the type of `length`, pronounced “alpha,” is a *type variable* standing for an arbitrary type.

The inferred type tells us that the function can take a list with elements of *any* type (i.e., a list with elements of type alpha, for any choice of alpha).

We’ll come back to polymorphism in more detail a bit later.

Tuples

Items connected by commas are “tuples.” (The enclosing parens are optional.)

```
# "age", 44;;
- : string * int = "age", 44

# "professor","age", 33;;
- : string * string * int = "professor", "age", 33

# ("children", ["bob";"ted";"alice"]);;
- : string * string list =
        "children", ["bob"; "ted"; "alice"]

# let g (x,y) = x*y;;
val g : int * int -> int = <fun>
```

How many arguments does g take?

Tuples are not lists

Please do not confuse them!

```
# let tuple = "cow", "dog", "sheep";;
val tuple : string * string * string =
      "cow", "dog", "sheep"

# List.hd tuple;;
This expression has type string * string * string
but is here used with type 'a list
```

```
# let tup2 = 1, "cow";;
val tup2 : int * string = 1, "cow"

# let l2 = [1; "cow"];;
This expression has type string but is here
used with type int
```

Tuples and pattern matching

Tuples can be “deconstructed” by pattern matching:

```
# let lastName name =
    match name with
      (n,_,_) -> n;;

# lastName  ("Pierce", "Benjamin", "Penn");;
- : string = "Pierce"
```

Example: Finding words

Suppose we want to take a list of characters and return a list of lists of characters, where each element of the final list is a "word" from the original list.

```
# split ['t';'h';'e';' ';'b';'r';'o';'w';'n';
         ' ';'d';'o';'g'];;
- : char list list =
      [['t'; 'h'; 'e']; ['b'; 'r'; 'o'; 'w'; 'n'];
       ['d'; 'o'; 'g']]
```

(Character constants are written with single quotes.)

An implementation of `split`

```
# let rec loop w l =
    match l with
      [] -> [w]
    | (' '::ls) -> w :: (loop [] ls)
    | (c::ls) -> loop (w @ [c]) ls;;
val loop : char list
        -> char list
        -> char list list
         = <fun>

# let split l = loop [] l;;
val split : char list -> char list list = <fun>
```

Note the use of both tuple patterns and nested patterns. The `@` operator is shorthand for `List.append`.

Aside: Local function definitions

The `loop` function is completely local to `split`: there is no reason for anybody else to use it — or even for anybody else to be able to see it! It is good style in OCaml to write such definitions as *local bindings*:

```
# let split l =
    let rec loop w l =
      match l with
        [] -> [w]
      | (’ ’::ls) -> w :: (loop [] ls)
      | (c::ls) -> loop (w @ [c]) ls  in
    loop [] l;;
```

In general, any let definition that can appear at the top level

```
# let ...;;
# e;;;
```

can also appear in a `let...in...` form.

```
# let ... in e;;;
```

A Better Split

Our `split` function worked fine for the example we tried it on. But here are some other tests:

```
# split ['a';' ';' ';'b'];;
- : char list list = [['a']; []; ['b']]

# split ['a';' '];;
- : char list list = [['a']; []]
```

Could we refine `split` so that it would leave out these spurious empty lists in the result?

Sure. First rewrite the pattern match a little (without changing its behavior):

```
# let split l =
    let rec loop w l =
      match w,l with
        _, [] -> [w]
      | _, (’ ’::ls) -> w :: (loop [] ls)
      | _, (c::ls) -> loop (w @ [c]) ls   in
    loop [] l;;
```

Then add a couple of clauses:

```
# let better_split l =
    let rec loop w l =
      match w,l with
        [],[] -> []
      | _,[] -> [w]
      | [], (’ ’::ls) -> loop [] ls
      | _, (’ ’::ls) -> w :: (loop [] ls)
      | _, (c::ls) -> loop (w @ [c]) ls   in
    loop [] l;;

# better_split [’a’;’b’;’ ’;’ ’;’c’;’ ’;’d’;’ ’];;
- : char list list = [[’a’; ’b’]; [’c’]; [’d’]]
# better_split [’a’;’ ’];;
- : char list list = [[’a’]]
# better_split [’ ’;’ ’];;
- : char list list = []
```

Basic Exceptions

OCaml's exception mechanism is roughly similar to that found in, for example, Java.
We begin by defining an exception:

```
# exception Bad;;
```

Now, encountering `raise Bad` will immediately terminate evaluation and return control to the top level:

```
# let rec fact n =
    if n<0 then raise Bad
    else if n=0 then 1
    else n * fact(n-1);;
# fact (-3);;
Exception: Bad.
```

(Not) catching exceptions

Naturally, exceptions can also be caught within a program (using the `try...with...` form), but let's leave that for another day.

Defining New Types of Data

Predefined types

We have seen a number of data types:

`int`	
`bool`	
`string`	
`char`	
`[x;y;z]`	`lists`
`(x,y,z)`	`tuples`

Ocaml has a number of other built-in data types — in particular, `float`, with operations like `+.`, `*.`, etc.

One can also create completely new data types.

The need for new types

The ability to construct new types is an essential part of most programming languages.

For example, suppose we are building a (very simple) graphics program that displays circles and squares. We can represent each of these with three real numbers...

A circle is represented by the co-ordinates of its center and its radius. A square is represented by the co-ordinates of its bottom left corner and its width. So we can represent *both* shapes as elements of the type:

```
float * float * float
```

However, there are two problems with using this type to represent circles and squares. First, it is a bit long and unwieldy, both to write and to read. Second, because their types are identical, there is nothing to prevent us from mixing circles and squares. For example, if we write

```
# let areaOfSquare (_,_,d) = d *. d;;
```

we might accidentally apply the `areaOfSquare` function to a circle and get a nonsensical result.
(Recall that numerical operations on the `float` type are written differently from the corresponding operations on `int` — e.g., `+.` instead of `+`. See the OCaml manual for more information.)

Data Types

We can improve matters by defining `square` as a new type:

```
# type square = Square of float * float * float;;
```

This does two things:

- It creates a *new* type called `square` that is different from any other type in the system.
- It creates a *constructor* called `Square` (with a capital `S`) that can be used to create a `square` from three floats. For example:

```
# Square(1.1, 2.2, 3.3);;
- : square = Square (1.1, 2.2, 3.3)
```

Taking data types apart

We take types apart with (surprise, surprise...) *pattern matching*.

```
# let areaOfSquare s =
    match s with
      Square(_, _, d) -> d *. d;;
val areaOfSquare : square -> float = <fun>

# let bottomLeftCoords s =
    match s with
      Square(x, y, _) -> (x,y);;
val bottomLeftCoords : square -> float * float
                          = <fun>
```

So we can use constructors like Square both as *functions* and as *patterns*.

These functions can be written a little more concisely by combining the pattern matching with the function header:

```
# let areaOfSquare (Square(_, _, d)) = d *. d;;
# let bottomLeftCoords (Square(x, y, _)) = (x,y);;
```

Continuing, we can define a data type for circles in the same way.

```
# type circle = Circle of float * float * float;;

# let c = Circle (1.0, 2.0, 2.0);;

# let areaOfCircle (Circle(_, _, r)) =
    3.14159 *. r *. r;;

# let centerCoords (Circle(x, y, _)) = (x,y);;

# areaOfCircle c;;
- : float = 12.56636
```

We *cannot* now apply a function intended for type square to a value of type `circle`:

```
# areaOfSquare c;;
This expression has type circle but is here used
with type square.
```

Variant types

Going back to the idea of a graphics program, we obviously want to have several shapes on the screen at once. For this we'd probably want to keep a list of circles and squares, but such a list would be *heterogenous*. How do we make such a list?

Answer: Define a type that can be *either* a circle *or* a square.

```
# type shape = Circle of float * float * float
             | Square of float * float * float;;
```

Now *both* constructors `Circle` and `Square` create values of type `shape`. For example:

```
# Square (1.0, 2.0, 3.0);;
- : shape = Square (1.0, 2.0, 3.0)
```

A type that can have more than one form is often called a *variant* type.

Pattern matching on variants

We can also write functions that do the right thing on all forms of a variant type. Again we use pattern matching:

```
# let area s =
    match s with
      Circle (_, _, r) -> 3.14159 *. r *. r
    | Square (_, _, d) -> d *. d;;

# area (Circle (0.0, 0.0, 1.5));;
- : float = 7.0685775
```

Here is a heterogeneous list:

```
# let l = [Circle (0.0, 0.0, 1.5);
           Square (1.0, 2.0, 1.0);
           Circle (2.0, 0.0, 1.5);
           Circle (5.0, 0.0, 2.5)];;

# area (List.hd l);;
- : float = 7.0685775
```

Mixed-mode Arithmetic

Many programming languages (Lisp, Basic, Perl, database query languages) use variant types internally to represent numbers that can be either integers or floats. This amounts to *tagging* each numeric value with an indicator that says what kind of number it is.

```
# type num = Int of int | Float of float;;

# let add r1 r2 =
    match (r1,r2) with
      (Int i1,   Int i2)   -> Int (i1 + i2)
    | (Float r1, Int i2)   -> Float (r1 +. float i2)
    | (Int i1,   Float r2) -> Float (float i1 +. r2)
    | (Float r1, Float r2) -> Float (r1 +. r2);;

# add (Int 3) (Float 4.5);;
- : num = Float 7.5
```

More Mixed-Mode Functions

```
# let unaryMinus n =
    match n with Int i -> Int (- i)
               | Float r -> Float (-. r);;

# let minus n1 n2 = add n1 (unaryMinus n2);;

# let rec fact n =
    if n = Int 0 then Int 1
    else mult n (fact (minus n (Int 1)));;

# fact (Int 7);;
- : num = Int 5040
```

What will happen if we write `fact 7`?

A Data Type for Optional Values

Suppose we are implementing a simple lookup function for a telephone directory. We want to give it a string and get back a number (say an integer). We expect to have a function `lookup` whose type is

```
lookup: string  -> directory -> int
```

where `directory` is a (yet to be decided) type that we'll use to represent the directory.
However, this isn't quite enough. What happens if a given string isn't in the directory? What should `lookup` return?
There are several ways to deal with this issue. One is to raise an exception. Another uses the following data type:

```
# type optional_int = Absent | Present of int;;
```

To see how this type is used, let's represent our directory as a list of pairs:

```
# let directory  = [("Joe", 1234); ("Martha", 5672);
                    ("Jane", 3456); ("Ed", 7623)];;

# let rec lookup s l =
    match l with
      [] -> Absent
    | (k,i)::t -> if k = s then Present(i)
                           else lookup s t;;

# lookup "Jane" directory;;
- : optional_int = Present 3456

# lookup "Karen" directory;;
- : optional_int = Absent
```

Built-in options

Because options are often useful in HOT programming, OCaml provides a built-in type `t option` for each type `t`. Its constructors are `None` (corresponding to `Absent`) and `Some` (for `Present`).

```
# let rec lookup s l =
    match l with
      [] -> None
    | (k,i)::t -> if k = s then Some(i)
                            else lookup s t;;

# lookup "Jane" directory;;
- : optional_int = Some 3456
```

Enumerations

The `option` type has one variant, `None`, that is a “constant” constructor carrying no data values with it. Data types in which *all* the variants are constants can actually be quite useful...

```
# type color = Red | Yellow | Green;;
# let next c =
    match c with Green -> Yellow | Yellow -> Red
               | Red -> Green;;
```

```
# type day =   Sunday | Monday | Tuesday | Wednesday
             | Thursday | Friday | Saturday;;
# let weekend d =
    match d with
      Saturday -> true
    | Sunday   -> true
    | _        -> false;;
```

A Boolean Data Type

A simple data type can be used to replace the built-in booleans. We use the constant constructors `True` and `False` to represent *true* and *false*. We'll use different names as needed to avoid confusion between our booleans and the built-in ones:

```
# type myBool = False | True;;
# let myNot b =
    match b with False -> True | True -> False;;
# let myAnd b1 b2 =
    match (b1,b2) with
      (True,  True)  ->  True
    | (True,  False) ->  False
    | (False, True)  ->  False
    | (False, False) ->  False;;
```

Note that the behavior of `myAnd` is not quite the same as the built-in `&&`!

Recursive Types

Consider the tiny language of arithmetic expressions defined by the following (BNF-like) grammar:

```
exp  ::=  number
          ( exp + exp )
          ( exp - exp )
          ( exp * exp )
```

(We'll come back to these grammars in more detail next week...)

We can translate this grammar directly into a datatype definition:

```
type ast =
    ANum of int
  | APlus of ast * ast
  | AMinus of ast * ast
  | ATimes of ast * ast;;
```

Notes:

- This datatype (like the original grammar) is *recursive*.
- The type `ast` represents *abstract syntax trees*, which capture the underlying tree structure of expressions, suppressing surface details such as parentheses

An evaluator for expressions

Goal: write an evaluator for these expressions.

```
val eval : ast -> int = <fun>

# eval (ATimes (APlus (ANum 12, ANum 340), ANum 5));
- : int = 1760
```

The solution uses a recursive function plus a pattern match.

```
let rec eval e =
  match e with
    ANum i -> i
   | APlus (e1,e2) -> eval e1 + eval e2
   | AMinus (e1,e2) -> eval e1 - eval e2
   | ATimes (e1,e2) -> eval e1 * eval e2;;
```