

Formal Methods

Lecture 2

(B. Pierce's slides for the book “Types and Programming Languages”)

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 'alist
```

```
# 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 wl =  
    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 functional 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 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;;
```

Polymorphism

Polymorphism

We encountered the concept of polymorphism very briefly last time. Let's look at it now in a bit more detail.

```
# let rec last l
  = match l with
    [] -> raise Bad
  | [x] -> x
  | _::y -> last y
```

What type should we give to the parameter `l`?

Polymorphism

```
# let rec last l
  = match l with
    | [] -> raise Bad
    | [x] -> x
    | _::y -> last y
```

It doesn't matter what type of objects are stored in the list: we could make it `int list` or `bool list`, and OCaml would not complain. However, if we chose one of these types, would not be able to apply `last` to the other.

Instead, we can give `l` the type `'a list` (pronounced "alpha"), standing for an arbitrary type. When we use the function, OCaml will figure out what type we need.

Polymorphism

This version of `last` is said to be polymorphic, because it can be applied to many different types of arguments. ("Poly" = many, "morph" = shape.)

Note that the type of the elements of `l` is `'a` (pronounced "alpha"). This is a type variable, which can *instantiated*, each time we apply `last`, by replacing `'a` with any type that we like. The instances of the type `'a list -> 'a` include

```
int list -> int
string list -> string
int list list -> int list
etc.
```

In other words,

$$\text{last} : 'a \text{ list} \rightarrow 'a$$

can be read, "`last` is a function that takes a list of elements of any type `alpha` and returns an element of `alpha`."

A polymorphic append

```
# let rec append (l1: 'alist) (l2: 'alist)
  =if l1 = [] then l2
  else List.hd l1 :: append (List.tl l1) l2;;
val append : 'alist -> 'alist -> 'alist = <fun>

# append [4; 3; 2] [6; 6; 7];;
- : int list = [4; 3; 2; 6; 6; 7]

# append ["cat"; "in"] ["the"; "hat"];;
- : string list = ["cat"; "in"; "the"; "hat"]
```

A polymorphic rev

```
# let rec revaux (l: 'alist) (res: 'alist) =  
  if l = [] then res  
  else revaux (List.tl l) (List.hd l :: res);;  
val revaux : 'alist -> 'alist -> 'alist = <fun>  
  
# let rev (l: 'alist) = revaux l [];;  
val rev : 'alist -> 'alist = <fun>  
  
# rev ["cat"; "in"; "the"; "hat"];;  
- : string list = ["hat"; "the"; "in"; "cat"]  
-  
# rev [false; true];;  
- : bool list = [true; false]
```

Polymorphic repeat

```
# (* Alist of n copies of k *)  
let rec repeat (k:'a) (n:int) =  
    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]  
  
# repeat true 3;;  
- : bool list = [true; true; true]  
  
# repeat [6;7] 4;;  
- : int list list = [[6; 7]; [6; 7]; [6; 7]; [6; 7]]
```

What is the type of repeat?

Palindromes

A palindrome is a word, sentence, or other sequence that reads the same forwards and backwards.

```
# let palindrome (l: 'alist) =  
    l = (rev l);;  
val palindrome : 'alist -> bool = <fun>  
  
# palindrome ["a";"b";"l";"e"; "w";"a";"s";  
              "I"; "e";"r";"e"; "I";  
              "s";"a";"w"; "e";"l";"b";"a"];;  
- : bool = true  
-  
# palindrome [true; true; false];;  
- : bool = false
```

Digression: Approaches to Typing

- ▲ A *strongly typed* language prevents programs from accessing private data, corrupting memory, crashing the machine, etc.
- ▲ A *weakly typed* language does not.
- ▲ A *statically typed* language performs type-consistency checks at when programs are compiled.
- ▲ A *dynamically typed* language delays these checks until programs are executed.

	Weak	Strong
Static		ML, Java*, C#*
Dynamic	C, C++	Lisp, Scheme

*Strictly speaking, Java and C# should be called “mostly static”

Practice with Types

What are the types of the following functions?

- △ `let f (x:int) = x + 1`
- △ `let f x = x + 1`
- △ `let f (x:int) = [x]`
- △ `let f x = [x]`
- △ `let f x = x`
- △ `let f x = hd(tl x) :: [1.0]`
- △ `let f x = hd(tl x) :: []`
- △ `let f x = 1 :: x`
- △ `let f x y = x :: y`

- ▴ `let f x y = x :: []`
- ▴ `let f x = x @ x`
- ▴ `let f x = x :: x`
- ▴ `let f x y z = if x>3 then y else z`
- ▴ `let f x y z = if x>3 then y else [z]`

And one more:

```
let rec f x      =  
  if (tl x) = [] then x  
  else f (tl x)
```