CS 11 Ocaml track: lecture 2

Today:
- comments
- algebraic data types
- more pattern matching
- records
- polymorphic types
- ocaml libraries
- exception handling
Previously...

- OCaml interactive interpreter
- Compiling standalone programs
- Basic data types and operators
- `let` expressions, `if` expressions
- Functions
- Pattern matching
- Higher-order functions
- Tail recursion
Comments

- Comments start with (* and end with *)
  - can be nested

- No single-line comments

(* This is a comment. *)

(* This is
  a (* nested comment *)
*)
Algebraic data types

- AKA "union types"
- Idea: want a new data type that can be any one of several different things
- Extremely useful!
  - makes it easy to define complex data types
- Pattern matching automatically works with the structure of these types
Example 1

- Example:
  
  ```
  type card = Spade | Heart | Diamond | Club
  ```

- `type` is a keyword
- `card` is the name of the type you're defining
- `Spade, Heart, Diamond, and Club` are type constructors
  - also instances of type `card`
- `type` names must start with lower-case letter
- constructors must start with upper-case letter
let string_of_card c =

match c with

| Spade   -> "Spade"
| Heart   -> "Heart"
| Diamond -> "Diamond"
| Club    -> "Club"

| means "or" (conceptually)

N.B. first | is optional
Example 2

type number = (* generic numbers *)
   Zero
  | Integer of int
  | Real of float

let float_of_number n =
  match n with
  Zero -> 0.0
  | Integer i -> float_of_int i
  | Real f -> f
Example 2 -- alternate

type number = (* generic numbers *)
  | Zero      (* note leading | *)
  | Integer of int
  | Real of float

let float_of_number n =
  match n with
  | Zero -> 0.0    (* note leading | *)
  | Integer i -> float_of_int i
  | Real f -> f
Aside: the function keyword

let float_of_number = function
    Zero -> 0.0
    | Integer i -> float_of_int i
    | Real f -> f

- Used for pattern matching with a one-argument function
- Just a shortcut
- Contrast: fun keyword doesn't match patterns
Example 2

let add n1 n2 = (* add generic numbers *)
    match n1, n2 with
    | Zero, n    (* fall through to next case *)
    | n, Zero -> n
    | Integer i1, Integer i2 -> Integer (i1 + i2)
    | Integer i, Real r  (* fall through *)
    | Real r, Integer i -> Real (r +. float_of_int i)
    | Real r1, Real r2 -> Real (r1 +. r2)
Example 3

- Abstract integer type:

  ```ocaml
type integer = (* recursive data type *)
    | Zero
    | Succ of integer
  ```

- NOTE: Can't re-use a constructor name (here, Zero) in the same module
Example 3

```ocaml
let rec add x y =
  match x with
  | Zero -> y
  | Succ x' -> Succ (add x' y)
```

- Recall: when defining a recursive function, need to use `let rec`
Defining your own operators

- In ocaml, can define your own operators
- Note that surrounding operator with () makes it into a function

```ocaml
# (+) ;;
```

- : int -> int -> int = <fun>

- Here, (+) is the function version of the + operator
Defining your own operators

- Want a +++ operator for our new integers:

```ocaml
let rec (++++) x y =
  match x with
  | Zero -> y
  | Succ x' -> Succ (x' +++ y)
```

- Recall: when defining a recursive function, need to use `let rec`

- New operators can only use non-alphanumerical characters (except for some built-in ones)
Defining your own operators

Why is this broken?

let rec (***) x y =

    match x with
    | Zero -> Zero
    | Succ Zero -> y
    | Succ x' -> y +++ (x' *** y)
Defining your own operators

Correct version:

```ocaml
let rec ( *** ) x y =
    match x with
    | Zero -> Zero
    | Succ Zero -> y
    | Succ x' -> y +++ (x' *** y)
```
Records

- A record bundles together different pieces of data
  - with possibly different types
- Like a tuple with a name for each position in the tuple

```python
type named_point = {
  name : string ;
  x : float;
  y : float;
}
```
Creating records

# { name="foo"; x=10.0; y=20.0 } ;;
- : named_point = {name = "foo"; x = 10.; y = 20.}

- NOTE: Type inference correctly determines that the above expression is a named_point

- Can also write this as
  
  { x=10.0; name="foo"; y=20.0 }

  (the fields don't have to be in any order)

- However, you can't leave out any of the field names
Using records

let add_points p1 p2 =
    match p1, p2 with
    {name=n1; x=x1; y=y1},
    {name=n2; x=x2; y=y2} ->
    {name=n1 ^ n2; x=x1 +. x2; y=y1 +. y2}
The `_` pattern

```
let add_points p1 p2 =
    match p1, p2 with
    {name=n1; x=x1; y=y1},
    {name=_; x=x2; y=y2} ->
    {name=n1; x=x1 +. x2; y=y1 +. y2}
```

- `_` in patterns means "don't care"
- ignores value in that position
Polymorphic types

- Consider this function:

```ocaml
let rec list_length lst =
    match lst with
    | [] -> 0
    | (h :: t) -> 1 + list_length t
```

- What's the type of list_length?

```ocaml
val list_length : 'a list -> int = <fun>
```
Polymorphic types

- What's the type of list_length?
val list_length : 'a list -> int = <fun>
- This is a *polymorphic* type
- Same type for lists of ints, lists of floats, etc.
  list_length [1;2;3;4;5] \(\rightarrow\) 5
  list_length ["foo"; "bar"; "baz"] \(\rightarrow\) 3
- However, list elements must all be of same type
- How do we define a type like that?
Polymorphic types

- Let's define our own list type:
  
  ```
  type 'a our_list =
      | Nil
      | Cons of 'a * 'a our_list
  ```

- 'a says that this is a polymorphic type

- Note: tuple types are printed with * e.g.

```ocaml
# (10, "foo") ;;
- : int * string = (10, "foo")
```
Polymorphic types

Let's use our new type:

```ocaml
let rec list_length our_lst =
  match our_lst with
  | Nil -> 0
  | Cons (h, t) -> 1 + list_length t
```
Note on the libraries

- There is a library function called **List.length**
- Lives in the **List** module
- Documented on www.ocaml.org web site
- You should browse through the standard libraries:
  - **Pervasives (built-in)**
  - **List**
  - **Array**
  - **Hashtbl**
  - **Printf**
Note on the libraries

- You don't have to have an "import" statement to use library functions

  # List.length [1;2;3;4;5]
  - : int = 5

- If you don't want to type List. all the time you can do

  open List

- but I recommend against it.
Exception handling

- Ocaml includes a simple and effective exception handling system
- ML language one of the first ones in which exception handling was incorporated
- New keywords:
  - raise
  - try
  - with
  - exception
Example

# let rec find x lst =
    match lst with
    | [] -> raise (Failure "not found")
    | h :: t -> if x = h then x else find x t

val find : 'a -> 'a list -> 'a = <fun>
Example

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

# find 0 [1;2;3;4;5];;
Exception: Failure "not found".

# Failure ("not found");;
- : exn = Failure "not found"
exceptions

- Exceptions have type `exn`
- Like an extensible union type
- Can add new constructors using the `exception` keyword:
  ```
  # exception Bad of string ;;
  exception Bad of string
  ```
- Recall: constructor must have first letter capitalized
Raise exceptions using the keyword `raise`:

```
# raise (Bad "this is really whacked!");;
Exception: Bad "this is really whacked!".
```
try/with (1)

- Catch exceptions in a try/with statement:

```
# try
    raise (Bad "this is really whacked!")
    with (Bad s) -> s ;;
- : string = "this is really whacked!"
```
try/with (2)

- Catching multiple exceptions:

```ocaml
# try
  raise (Bad "this is really whacked!")
with e ->
  match e with
    (Bad s) -> s
  | _   -> "whatever";;
- : string = "this is really whacked!"
```
try/with (3)

- Catching multiple exceptions, alternate way:

```ocaml
# try
  raise (Bad "this is really whacked!")
with (Bad s) -> s
  | (Failure f) -> f
  | _ -> "whatever" ;;
- : string = "this is really whacked!"
```
try/with (4)

- Slight variation:

```ocaml
# try
  raise (Bad "this is really whacked!")

with
  | (Bad s) -> s
  | (Failure f) -> f
  | _ -> "whatever" ;;
- : string = "this is really whacked!"
```
Next week

- Imperative programming in ocaml!
- The module system