CMSC 430
Introduction to Compilers
Fall 2018

Everything (else) you always wanted to know about OCaml (but were afraid to ask)

OCaml

- You know it well from CMSC 330
- All programming projects will be in OCaml
  - OCaml is well-designed for building language tools
- In 330, we covered all the basics
  - Tuples, lists, recursion, pattern matching, higher-order functions, currying, data types, modules, module types, updatable references
- For larger projects, there's more to know

Records

- Labeled tuples of values

```
# type course = {title:string; num:int};;
type course = { title : string; num : int; }
# let x = {title="Intro to Compilers"; num=430};;
val x : course = {title = "Intro to Compilers"; num = 430}
```

- Fields are referenced with the dot notation

```
# x.title;;
- : string = "Introduction to Compilers"
# x.number;;
- : int = 430
```

- All record types are named, and must be complete in any instance

```
# let y = {title="Intro to Compilers"};;
Error: Some record field labels are undefined: num
```

Records (cont'd)

- Record patterns can include partial matches

```
# let nextNum {num=x} = x;;
val nextNum : course -> int = <fun>
```

- The with construct can be used to modify just part of a record

```
# {x with num=431};;
- : course = {title = "Intro to compilers"; num = 431}
```

Records (cont'd)

- Record fields may be mutable

```
# type course = {title:string; mutable num:int};;
type course = { title : string; mutable num : int; }
# let x = {num=430; title="Intro to compilers"};;
val x : course = {title = "Intro to compilers"; num = 430}
# x.num <- 431;;
- : unit = ()
# x;;
- : course = {title = "Intro to compilers"; num = 431}
```

- In fact, this is what updatable refs translate to

```
# let y = ref 42;;
val y : int ref = {contents = 42}
```

Arrays and strings

- OCaml arrays are mutable and bounds-checked

```
# let x = [|1;2;3|];;
val x : int array = [|1; 2; 3|]
# x.(0) <- 4;;
- : unit = ()
# x;;
- : int array = [|4; 2; 3|]
# x.(4);;
Exception: Invalid_argument "index out of bounds".
# x.(-1);;
Exception: Invalid_argument "index out of bounds".
```

- OCaml strings are also mutable (this will change!)

```
# let x = "Hello";;
val x : string = "Hello"
# x.[0] <- 'J';;
- : unit = ()
# x;;
- : string = "Jello"
```

Design discussion

- OCaml has several similar constructs
  - Tuples
  - Lists
  - Records
  - Arrays
  - Data types

- Why have all these choices? Do other languages (e.g., Ruby) have all these different constructs?

Labeled arguments

- OCaml allows arguments to be labeled

```
# let f ~x ~y = x-y;;
val f : x:int -> y:int -> int = <fun>
# f 4 3;;
- : int = 1
# f ~y:4 ~x:3;;
- : int = -1
```

- Functions with labeled args can be partially applied

```
# let g = f ~y:4;;
val g : x:int -> int = <fun>
# g 3;;
- : int = -1
# g ~x:3;;
- : int = -1
```

Optional arguments

- Labeled arguments may be optional

```
# let bump ?(step = 1) x = x + step;;
val bump : ?step:int -> int -> int = <fun>
# bump 2;;
- : int = 3
# bump ~step:3 2;;
- : int = 5
```

- One note: type inference with partial applications of functions with labeled arguments may not always work

While and for

```
# while true do Printf.printf “Hello\n”;;
Hello
Hello
Hello
...
# for i = 1 to 10 do Printf.printf "%d\n" i done;;
1
2
...
10
```

- Can you encode while and for only using functions and recursion?

Modules

```
module type SHAPES =
  sig
    type shape
    val area : shape -> float
    val unit_circle : shape
    val make_circle : float -> shape
    val make_rect : float -> float -> shape
end;;

module Shapes : SHAPES =
  struct
    ...
    let make_circle r = Circle r
    let make_rect x y = Rect (x, y)
  end
```

Functors

- Modules can take other modules as arguments
  - Such a module is called a *functor*

```
module type OrderedType = sig
  type t
  val compare : t -> t -> int
end

module Make(Ord: OrderedType) =
struct ... end

module StringSet = Set.Make(String);;
(* works because String has type t, implements compare *)
```

- Other examples: Hashtbl, Map, Queue, Stack

Variants

- Recall OCaml data types (also called *variants*)

```
type shape =
| Circle of float
| Rect of float * float
```

- Each constructor name refers to a unique type
  - E.g., Circle always makes a shape
- Some downsides
  - Have to define all such types in advance of uses
  - Can’t accept data coming from two different variants

Polymorphic variants

- Like variants, but permit an unbounded number of constructors, created anywhere
  - Type inference takes care of matching up various uses

```
# [`On; `Off];;
- : [> `Off | `On ] list = [`On; `Off]
# `Number 1;;
- : [> `Number of int ] = `Number 1
# let f = function `On -> 1 | `Off -> 0 | `Number n -> n;;
val f : [< `Number of int | `Off | `On ] -> int = <fun>
# List.map f [`On; `Off];;
- : int list = [1; 0]
```

  - “<”—allow fewer tags   “>”—allow more tags
  - Can remove this ability by creating a named type

```
# type 'a vlist = [`Nil | `Cons of 'a * 'a vlist];;
type 'a vlist = [ `Cons of 'a * 'a vlist | `Nil ]
```

Regular vs. polymorphic variants

- Benefits of polymorphic variants:
  - More flexible
  - If used well, can improve modularity, maintainability

- Benefits of regular variants:
  - More type checking permitted
    - Only declared constructors used
    - Check for complete pattern matching
    - Enforce type constraints on parameters
  - Better error messages
    - Sometimes type inference with polymorphic variants subtle
  - Compiler can create slightly more optimized code
    - More is known at compile time

A note on OCaml versions

- We will use version 4.03.0
  - Add the following directory to your path:

    /afs/glue.umd.edu/class/fall2018/cmsc/430/0201/public/bin

    - Ask a TA if you don't know how
  - This version should be installed on submit now
- If you are installing OCaml yourself, we recommend using opam
  - We'll also use the ounit and Yojson packages
    - They are installed on GRACE at the path above