Lecture 31:

- Intro to OCaml (cont)

Announcements:

- HW-6 out, due Mon

OCaml Basics

If-then-else expressions: `if` b `then` e_1 `else` e_2

- b is a boolean expression, e_1 and e_2 must have same types
- evaluated to either e_1 or e_2

```
# if true then 0 else 1 ;;
- : int = 0

# if false then 1 else if true then 2 else 3 ;;
- : int = 2

# if true then 1 else 2.0 ;;
Error: This expression has type float but an expression
       was expected of type int
```

Not a normal conditional *statement* ... it is an *expression*!

- e.g., `# let s = if true then 1 else 0 ;;`
- Can have `if-then` without else ... must be unit type (e.g., printing)
- Can use `( )`'s for formatting and also `begin ... end`

OCaml Basics

Primitive Types:

- `int` ... a signed 63-bit integer (extra bit for garbage collection)
- `float` ... similar to C++'s double type
- `bool` ... boolean value either `true` or `false`
- `char` ... a single-byte character value (e.g., `'a'`)
- `string` ... an immutable character sequence
- `unit` ... written as `()` and similar to `void` (i.e., a type without values)

OCaml Basics

Examples:

```
# Sys.int_size ;;
- : int = 63
# 'a' ;;
- : char = 'a'
# "foo" ;;
- : string = "foo"
# () ;;
- : unit = ()
# "foo" ^ "bar" ;;                              (* concatenation *)
- : string = "foobar"
# "foo".[0] ;;                                 (* character access *)
- : char = 'f'
# "foo".[3] ;;
Exception: Invalid_argument "index out of bounds".
# (float 2) +. 3.0 ;;                        (* explicit conversion *)
- : float = 5.
# int_of_float 3.14 ;;           (* explicit conversion, full name *)
- : int = 3
```

OCaml Basics

Expressions

- an expression evaluates to a value (e.g., to a literal)
- whereas not all statements are expressions (e.g., like in C++/Java/etc.)

Identifiers

- must start with a lowercase letter or an underscore
- can contain any number of letters, numbers, underscores, or '

Naming: binding an identifier to an expression (possibly parameterized)

- which is different than defining a variable (which can change) ...
- a parameterized name is a function

OCaml Basics

Let Bindings: `let` *name* = *expression*

```
# let var1 = 42 ;;
val var1 : int = 42

# var1;;
- : int = 42

# x = 5 ;;
Error: unbound value x

# let pi = 3.14159 ;;
val pi : float = 3.14159

# pi;;
- : float = 3.14159

# let pi' = 3.14 ;;
val pi' : float = 3.14
```

OCaml Basics

Let-In Expressions: `let` *name* = *expr1* `in` *expr2*

```
# let var2 = 42 in var2 ;;
- : int = 42

# var2 ;;
Error: Unbound value var2

# let pi = 3.14 in pi /. 2.0 ;;
- : float = 1.57

# let x, y = 3, 4 in x * y ;;
- : int = 12

# let x = 3 and y = 4 in x * y ;;
- : int = 12
```

Local bindings (versus global) ... useful when defining functions (clean code)

OCaml Basics

Functions: essentially just "parameterized" let bindings

```
# let inc x = x + 1 ;;
val inc : int -> int = <fun>
# inc 1 ;;
- : int = 2
# inc (inc 1) ;;   ... apply: # inc @@ inc 1    or pipeline: # 1 |> inc |> inc ;;
- : int = 3
# inc ;;
- : int -> int = <fun>
```

Note the function type: `inc : int -> int` ... `# #show inc;;`

- `inc` takes an `int` and returns an `int`
- we say `inc`'s type is "`int` *to* `int`"

Note how we call the function:

- instead of `inc(1)` ... uses λ-calculus style of function application

OCaml Basics

Check in: What is wrong with the following? How do we fix it?

```
# let inc x = x + 1 ;;
val inc : int -> int = <fun>

# let dec x = x - 1 ;;
val inc : int -> int = <fun>

# dec inc 1 ;;
Error: This function has type int -> int
       It is applied to too many arguments; maybe you forgot a `;'.

# dec (inc 1) ;;
- : int = 1

# inc (inc (inc 1)) ;;
- : int = 4
```


OCaml Basics

Can mix let-in expressions and let bindings:

```
# let deg_to_rad d =
    let pi = 3.14 and half_circle = 180.0 in
    d *. (pi /. half_circle) ;;
val deg_to_rad : float -> float = <fun>

# deg_to_rad 180.0 ;;
- : float = 3.14000000000000057
```

We'll talk more about functions and types as we go

OCaml Basics

Can also "run" a source file ... e.g., `hello.ml`:

```
print_endline "Hello, World!"
```

To run from the command line:

```
$ ocaml hello.ml
"Hello World!"
```

To compile and run as an executable

```
$ ocamlopt -o hello hello.ml
$ ./hello
"Hello World!"
```

Can also load into the REPL ... e.g., `funs.ml`:

```
let inc x = x + 1
let dec x = x -1
```

```
# #use "inc_dec.ml" ;;
# inc 1 ;;
- : int = 2
```