CMSC330: OCaml Basics

Chris Kauffman

Last Updated:
Thu Sep 28 09:09:59 AM EDT 2023

Logistics

Assignments

- Project 3 Due Fri 06-Oct: Regex $\rightarrow$ NFA $\rightarrow$ DFA
- **Quiz 2 on Fri 29-Sep in Discussion**
  REMINDER: Past Semester Quizzes available under **Resources** on class web page
- Exam 1 on Thu 05-Oct, covers topics through Thu 28-Sep

Reading: OCaml Docs `https://ocaml.org/docs`

- Tutorial: Your First Day with OCaml
- Tutorial: OCaml Language Overview

Goals: OCaml Overview

- Static Types / Type Inference
- Pattern Matching
- Aggregate Data

A bit of History...

- 1930s: Alonzo Church invents the **Lambda Calculus**, a notation to succinctly describe computable functions.
- 1958: John McCarthy and others create **Lisp**, a programming language modeled after the Lambda Calculus. Lisp is the second oldest programming language still widely used.
  - Descendants of Lisp include Common Lisp, Emacs Lisp, Scheme, **Racket**, etc.
  - Lisp influenced almost **every other language** that followed it
- 1972: Robin Milner and others at Edinburgh/Stanford develop the Logic For Computable Functions (LCF) Theorem Prover to do mathy stuff
- To tell LCF how to go about its proofs, they invent a **Meta Language (ML)** which is like **Lisp with a type system** (Hindley-Milner type system)
- Folks soon realize that ML is a damn fine general purpose programming language and start doing things with it besides programming theorem provers

Origins of OCaml

Circa 1990, Xavier Leroy at France's INRIA looks at the variety of ML implementations and declares

"C'est nul" == "It's crap!"

- No command line compiler: only top level REPL
- Run only on Main Frames, not Personal Computers (a la Unix to Linux)
- Hard to experiment with adding new features

Xavier Leroy in 2010

Leroy develops the ZINC[a] system for INRIA's flavor of ML: Categorical Abstract Machine Language (CAML) to allow

- Separate **compilation to bytecode** and linking

Later work introduces

- Object system: **Objective Caml**, shortened to **OCaml**
- Native code compiler
- Various other tools sweet tools like a **time traveling debugger**

Question: Bytecode? Native Code? What are these?

---

[a] Xavier Leroy. The ZINC Experiment. Technical report 117, INRIA, 1990

Bytecode versus Native Code Compilation

Native Code Compilation

Convert source code to a form directly understandable by a CPU (an executable program)

Bytecode Compilation

Convert source code to an intermediate form (bytecode) that is must be further converted to native code by an interpreter.

Source Code Interpreter

Directly execute source code as it is read by doing on-the-fly conversions to native code.

System	*Compilation/Execution Model*
Java	Compile to Bytecode: `javac`, Interpret to native: `java`
C / C++	Native Code Compilation: `gcc / clang`
Python	Interpret Source Code with on-the-fly bytecode creation: `python`
	REPL: `python`
OCaml	Compile to Bytecode: `ocamlc`, Interpret to native: `ocamlrun`
	Native Code Compilation: `ocamlopt`
	REPL: `ocaml`

Bytecode versus Native-Code Compilation

```
# BYTECODE COMPILER : ocamlc
> ocamlc speedtest.ml    # compile to bytecode
> file a.out             # show file type
a.out: a /usr/bin/ocamlrun script executable (binary data)

> time ./a.out           # time execution
33554432

real   0m0.277s          # about a quarter second passed
user   0m0.276s          # full debug features available
sys    0m0.000s

# NATIVE CODE COMPILER: ocamlopt
> ocamlopt speedtest.ml  # compile to native code
> file a.out             # show file type
a.out: ELF 64-bit LSB pie executable x86-64

> time ./a.out
33554432

real   0m0.022s          # about 1/10th the time: WAY FASTER
user   0m0.022s          # BIG BUT: can't use native code with
sys    0m0.000s          #          OCaml's debugger
```

Influence of Functional Programming and ML

Why are we studying OCaml? No one uses it...
– Every Student ever Tasked to Study OCaml

You may never use OCaml for a job, but you will definitely feel its effects via the adoption of **Functional Programming** and **ML-inspired static type systems**

- Java 8 added lambdas, enabled Map/Reduce, uses a Generics system that is verbose substitute for ML's polymorphic types
- C++ and C have added `auto` var types inferred by compiler
- F# (Microsoft) : OCaml + .NET framework
- Swift (Apple) : ML + Objective-C library access
- Scala : JVM language with type inference, algebraic data types, functional features, OO features, every lang feature known and unknown
- Rust: C + Some OCaml syntax + Some Type Inference + Manage memory entirely at compile time
  Incidentally, the first Rust compiler was written in OCaml

Exercise: Collatz Computation An Introductory Example

- `collatz.ml` prompts for an integer and computes the Collatz Sequence starting there
- The current number is updated to the next in the sequence via

```
if cur is EVEN cur=cur/2; else cur=cur*3+1
```

- This process is repeated until it converges to 1 (mysteriously) or the maximum iteration count is reached
- The code demonstrates a variety of Python features and makes for a great crash course intro
- With a neighbor, study this code and identify the features you should look for in every programming language

Exercise: Collatz Computation An Introductory Example

```
(* collatz.ml: *)
open Printf;;                     (* use printf *)
let verbose = true;;              (* module-level var *)

let collatz start maxsteps =      (* func of 2 params *)
  let cur = ref start in          (* local variable *)
  let step = ref 0 in             (* refs for mutation *)
  if verbose then
    begin
      printf "start: %d maxsteps %d\n" start maxsteps;
      printf "Step  Current\n";
    end;
  while !cur != 1 && !step < maxsteps do
    if verbose then
      printf "%3d: %5d\n" !step !cur;
    begin match !cur mod 2 with   (* pattern matching *)
    | 0 -> cur := !cur/2;         (* := is ref-assigment *)
    | _ -> cur := !cur*3+1;       (* !x is dereference *)
    end;
    step := !step + 1;
  done;
  (!cur,!step)                    (* return value *)
;;
let _ =                           (* main block *)
  print_string "Collatz start val:\n";
  let start = read_int () in
  let (final,steps) = collatz start 500 in
  printf "Reached %d after %d iters\n" final steps;
;;
```

Look for. . . Comments, Statements/Expressions, Variable Types, Assignment, Basic Input/Output, Function Declarations, Conditionals, Iteration, Aggregate Data, Library System

```
>> ocamlc collatz.ml
>> ./a.out
Collatz start val:
10
  0:    10
  1:     5
  2:    16
  3:     8
  4:     4
  5:     2
  6:     1
```

Answers: Collatz Computation An Introductory Example

- ☒ Comments: `(* comment between *)`
- ☒ Statements/Expressions: expressions sort of normal like

  `x+1      a && b      t < m     printf "%d" a;`

  Variables introduced via `let x = ..  in`
- ☒ Variable Types: string, integer, boolean are obvious as values, no type names mentioned... *isn't OCaml statically typed?*
- ☒ Assignment: via `let x = expr in` or `x := expr;`
- ☒ Basic Input/Output: `printf() / read_int()`
- ☒ Function Declarations: `let funcname param1 param2 =`
- ☒ Conditionals (if-else): `if cond then ...  else ...`

  Multiple statements require `begin/end`

  We'll get to know this sexy `match/with` *character as soon...*
- ☒ Iteration (loops): clearly `while cond do`, others soon
- ⊟ Aggregate data (arrays, records, objects, etc): `(ocaml,has,tuples)` and others we'll discuss soon
- ⊟ Library System: `open Printf` is like `from Printf import *`

Type Inference

- All vars/values are statically typed by compiler BUT…
- Compiler uses **type inference** to determine types so programs rarely states them explicitly; REPL shows this

```
>> ocaml                                (* start the REPL *)
OCaml version 5.0.0
Enter #help;; for help.
                   (* TYPE INFERENCE *)
# let x = 7;;                          (* bind x to 7 *)
val x : int = 7                        (* x must be an integer *)
# let doubler i = 2*i;;                (* bind doubler to a function *)
val doubler : int -> int = <fun>       (* int argument, int returns *)
(*            arg  return *)

                   (* TYPE CHECKING *)
# doubler 9;;                          (* call doubler on 9 *)
- : int = 18                           (* result is an integer *)
# doubler x;;                          (* call on x *)
- : int = 14                           (* ok: x is an integer *)
# doubler "hello";;                    (* call doubler "hello" *)
Line 1, characters 8-15:               (* Type Checker says: *)
1 | doubler "hello";;                  (* NO SOUP FOR YOU! *)
            ^^^^^^^
Error: This expression has type string but an
       expression was expected of type int
```

Type Inference During Compilation

While explicit types don't appear during normal compilation, they are always present and will appear in error messages

```
>> cat type_inference_errors.ml
```

```
1  open Printf;;
2
3  let doubler i = 2*i;;
4
5  let _ =
6    let four = doubler 2 in
7    let eight = doubler four in
8    let yesyes = doubler "yes" in      (* Like in Python, right? *)
9    printf "%d %d %d\n" four eight;
10   printf "%d\n" yesyes;
11 ;;
```

```
>> ocamlc type_inference_errors.ml   ## Have a tall glass of NOPE
File "type_inference_errors.ml", line 8, characters 23-28:
8 |   let yesyes = doubler "yes" in
                           ^^^^^
Error: This expression has type string but an expression
       was expected of type int
```

Types and Type Notations

Basic Types

Expected basic types for high-level langs are present like `int float bool string`
A few other special types like `unit` and `'a` are common that will be discussed momentarily

Aggregate Types

OCaml has various built-in aggregate types as well

```
# let ia = [|1; 2; 3|];;
val ia : int array = [|1; 2; 3|]

# let sl = ["a"; "b";];;
val sl : string list = ["a"; "b"]

# let tup = (true,4.56,"hi");;
val tup : bool * float * string
          = (true, 4.56, "hi")
```

Function Types

Functions have types with each param separated by an arrow `->` including the final return type

```
# let add a b = a+b;;
val add : int -> int -> int = <fun>

# add;;
- : int -> int -> int = <fun>

# let selfcat s = s^s;;
val selfcat : string -> string = <fun>

# int_of_string;;
- : string -> int = <fun>

# let add_pair (a,b) = a+b;;
val add_pair : int * int -> int = <fun>

# let give_meaning () = 42;;
val give_meaning : unit -> int = <fun>

# let poly_meaning x = 42;;
val poly_meaning : 'a -> int = <fun>
```

Type Annotations

- Types are inferred but one can **annotate** code with types
- Be aware that conflicts between annotations and inferred types will generate compiler errors
- May look at OCaml's Module System which includes Interface Files that state the types of all functions/variables, known as the module *signature*

```
# let a = 1;;
val a : int = 1

# let x : int = 5;;
val x : int = 5

# let y : int = "hi";;
Line 1, characters 14-18:
1 | let y : int = "hi";;
                  ^^^^
Error: This expression has type string but an
       expression was expected of type int

# let add (a : int) (b : int) : int = a+b;;
val add : int -> int -> int = <fun>

# let selfcat (s : string) : string = s+s;;
Line 1, characters 36-37:
1 | let selfcat (s : string) : string = s+s;;
                                        ^
Error: This expression has type string but an
       expression was expected of type int
```

Unit Type for Printing / Side-Effects

- The notation () means `unit` and is the return value of functions that only perform side-effects
- Roughly equivalent to `void` in C / Java / etc.
- Often appears as return type for output functions
- Usually don't about `unit` returns; don't bind result and. . .
- Functions with no parameters are passed () to call them
- **End statements returning unit with a semi-colon (;)** except at the top level where ;; is used instead

```
# print_string;;
- : string -> unit = <fun>

# print_string "hi\n";;
hi
- : unit = ()

# printf "%d\n" 42;;
42
- : unit = ()

# let meaning () = 42;;
val meaning : unit -> int = <fun>

# meaning;;
- : unit -> int = <fun>

# meaning ();;
- : int = 42
```

Exercise: Infer Those Types

- Determine the **type** of each of the following entities
- Tuples are created via `(a,b)` (parens optional)
- Lists are created via `[x;y;z]`
- Function types notated with `type1 -> type2 -> type3 -> ...`

Each function is a one-liner with its return value on its sole line

```
# let sum_diff a b =
    (a+b,a-b);;
val sum_diff : ????

# let catlist x y z =
    [x; x^y; x^z; y^z];;
val catlist : ????

# let diff_props a b =
    (a*b=0, a*b>0, a*b<0) ;;
val diff_props : ????

# let samy_print str =
     printf "%s - but Samy is my hero\n" str;;
val samy_print : ????

# let cur = 42;;
val cur : ????

# let print_cur () =
    printf "cur: %d\n" cur;;
val print_cur : ????
```

Answers: Infer Those Types

- Determine the **type** of each of the following entities
- Tuples are created via `(a,b)` (parens optional)
- Lists are created via `[x;y;z]`
- Function types notated with `type1 -> type2 -> type3 -> ...`

Each function is a one-liner with its return value on its sole line

```
# let sum_diff a b =
    (a+b,a-b);;
val sum_diff : int -> int -> int * int = <fun>

# let catlist x y z =
    [x; x^y; x^z; y^z];;
val catlist : string -> string -> string -> string list = <fun>

# let diff_props a b =
    (a*b=0, a*b>0, a*b<0) ;;
val diff_props : int -> int -> bool * bool * bool = <fun>

# let samy_print str =
     printf "%s - but Samy is my hero\n" str;;
val samy_print : string -> unit = <fun>

# let cur = 42;;
val cur : int = 42

# let print_cur () =
    printf "cur: %d\n" cur;;
val print_cur : unit -> unit = <fun>
```

Top-Level Statements

- Names bound to values are introduced with the `let` keyword
- At the top level, separate these with double semi-colon `;;`

REPL

```
>> ocaml
OCaml version 5.0.0
Enter #help;; for help.

# let name = "Chris";;
val name : string = "Chris"
# let office = 327;;
val office : int = 327
# let building = "Shepherd";;
val building : string = "Shepherd"
# let freq_ghz = 4.21;;
val freq_ghz : float = 4.21
```

Source File

```
(* top_level.ml : demo of top level
   statements separated by ;; *)
let name = "Chris";;
let office = 327;;
let building = "Shepherd";;
let freq_ghz = 4.21;;
let doubler a =
  2*a
;;
let pair_to_list (a,b) =
  [a; b];;

(* Top-level ;; are optional
   but help clarity for new
   OCaml Coders *)
let inc_it x = x+1

let dec_it y = y-1
```

Syntax Note for ;; in Modules

When writing `.ml` files, known as **Modules**, ending top-level bindings with ;; is optional. Lecture examples will include them to make definition ends clear but your own code may omit them. This is a matter of taste in source files but are required in the REPL which apes top-level declarations.

```
(* mod_sans_semis.ml:  *)
let astring = "Hello OCaml!"

let coll_step cur step =
  let next =
    if cur mod 2 == 0 then
      cur / 2
    else
      cur * 3 + 1
  in
  (next,step+1)


let pi = 3.14159

let area radius =
  pi *. radius**2.0
```

```
(* mod_with_semis.ml: *)
let astring = "Hello OCaml!";;

let coll_step cur step =
  let next =
    if cur mod 2 == 0 then
      cur / 2
    else
      cur * 3 + 1
  in
  (next,step+1)
;;

let pi = 3.14159;;

let area radius =
  pi *. radius**2.0;;
```

Exercise: Local Statements

- Statements in ocaml can be nested somewhat arbitrarily, particularly `let` bindings
- Commonly used to do actual computations
- Local `let` statements are followed by keyword `in`

```
let first =                 (* first top level binding *)
  let x = 1 in              (* local binding *)
  let y = 5 in              (* local binding *)
  y*2 + x                   (* * + : integer multiply and add *)
;;

let second =                (* second top-level binding *)
  let s = "TAR" in          (* local binding *)
  let t = "DIS" in          (* local binding *)
  s^t                       (* ^ : string concatenate (^) *)
;;
```

What value gets associated with names `first` and `second`?

Answers: Local Statements

```
let first =                 (* first top level binding *)
  let x = 1 in              (* local binding *)
  let y = 5 in              (* local binding *)
  y*2 + x                   (* * + : integer multiply and add *)
;;

(* binds first to
     y*2 + x
   = 5*2 + 1
   = 11
*)

let second =                (* second top-level binding *)
  let s = "TAR" in          (* local binding *)
  let t = "DIS" in          (* local binding *)
  s^t                       (* ^ : string concatenate (^) *)
;;
(* binds second to
     "TAR"^"DIS"  (concatenate strings)
   = "TARDIS"
*)
```

Clarity

```
(* A less clear way of writing the previous code *)
let first = let x = 1 in let y = 5 in y*2 + x;;
let second = let s = "TAR" in let t = "DIS" in s^t;;
```

- Compiler treats all whitespace the same so the code evaluates identically to the previous version
- Most readers will find this much harder to read
- **Favor clearly written code**
  - Certainly at the expense of increased lines of code
  - In most cases clarity trumps execution speed
- Clarity is of course a matter of taste

Exercise: Explain the following Compile Error

- Below is a source file that fails to compile
- Compiler error message is shown
- Why does the file fail to compile?

```
> cat -n local_is_local.ml
1   (* local_is_local.ml : demo of local binding error *)
2
3   let a =                    (* top-level binding *)
4     let x = "hello" in       (* local binding *)
5     let y = " " in           (* local binding *)
6     let z = "world" in       (* local binding *)
7     x^y^z                    (* result *)
8   ;;
9
10  print_endline a;;          (* print value of a *)
11
12  print_endline x;;          (* print value of x *)
> ocamlc local_is_local.ml
File "local_is_local.ml", line 12, characters 14-15:
Error: Unbound value x
```

Answers: Local Bindings are Local

```
1   (* local_is_local.ml : demo of local binding error *)
2
3   let a =                        (* top-level binding *)
4     let x = "hello" in           (* local binding *)
5     let y = " " in               (* local binding *)
6     let z = "world" in           (* local binding *)
7     x^y^z                        (* result *)
8   ;;                             (* x,y,z go out of scope here *)
9
10  print_endline a;;              (* a is well defined *)
11
12  print_endline x;;              (* x is not defined *)
```

- **Scope**: areas in source code where a name is well-defined and its value is available
- a is bound at the top level: value available afterwards; has module-level scope (module? *Patience, grasshopper...*)
- The scope of `x` ends at Line 8: not available at the top-level
- Compiler "forgets" `x` outside of its scope

Exercise: Fix Binding Problem

- Fix the code below
- Make changes so that it actually compiles and prints **both** a and x

```
(* local_is_local.ml : demo of local binding error *)

let a =                         (* top-level binding *)
  let x = "hello" in            (* local binding *)
  let y = " " in                (* local binding *)
  let z = "world" in            (* local binding *)
  x^y^z                         (* result *)
;;                              (* x,y,z go out of scope here *)

print_endline a;;               (* print a, it is well defined *)

print_endline x;;               (* x is not defined *)
```

Answers: Fix Binding Problem

One obvious fix is below

```
> cat -n local_is_local_fixed.ml
     1  (* local_is_local_fixed.ml : fixes local binding
     2     error by making it a top-level binding
     3  *)
     4
     5  let x = "hello";;           (* top-level binding *)
     6
     7  let a =                     (* top-level binding *)
     8    let y = " " in            (* local binding *)
     9    let z = "world" in        (* local binding *)
    10    x^y^z                     (* result *)
    11  ;;                          (* x,y,z go out of scope here *)
    12
    13  print_endline a;;           (* print a, it is well defined *)
    14
    15  print_endline x;;           (* print x, it is well defined *)

> ocamlc local_is_local_fixed.ml
> ./a.out
hello world
hello
```

Mutable and Immutable Bindings

Q: How do I change the value bound to a name?
A: You don't.

- OCaml's default is **immutable or persistent** bindings
- Once a name is bound, it holds its value until going out of scope
- Each `let/in` binding creates a scope where a name is bound to a value
- Most **imperative** languages feature easily **mutable** name/bindings

```
> python
Python 3.6.5
>>> x = 5
>>> x += 7
>>> x
12

// C or Java
int main(...){
  int x = 5;
  x += 5;
  System.out.println(x);
}

(* OCaml *)
let x = 5 in
???
print_int x;;
```

Approximate Mutability with Successive `let/in`

- Can approximate mutability by successively rebinding the same name to a different value

```
let x = 5 in      (* local: bind FIRST_x to 5 *)
let x = x+5 in    (* local: SECOND_x is FIRST_x+5, FIRST_x gone *)
print_int x;;     (* prints 10: most recent x, SECOND_x  *)
                  (* top-level: SECOND_x out of scope *)
print_endline "";;
```

- `let/in` bindings are more sophisticated than this but will need functions to see how
- OCaml also has explicit mutability via several mechanisms
  - `ref`: references which can be explictily changed
  - arrays: cells are mutable by default
  - records: fields can be labelled `mutable` and then changed

  We'll examine these soon

Exercise: let/in Bindings

- Trace the following program
- Show what values are printed and why they are as such

```
let x = 7;;
let y =
  let z = x+5 in
  let x = x+2 in
  let z = z+2 in
  z+x;;

print_int y;;
print_endline "";;

print_int x;;
print_endline "";;
```

Answers: `let/in` Bindings

- A later `let/in` supersedes an earlier one BUT…
- Ending a local scope reverts names to top-level definitions

```
let x = 7;;        (* top-level x <-------+ *)
let y =            (* top-level y <---+   | *)
  let z = x+5 in   (* z = 12 = 7+5    |   | *)
  let x = x+2 in   (* x =  9 = 7+2    |   | *)
  let z = z+2 in   (* z = 14 = 12+2   |   | *)
  z+x;;            (* 14+9 = 23 ------+   | *)
                   (* end local scope |   | *)
print_int y;;      (* prints 23 ------+   | *)
print_endline "";; (*                     | *)
                   (*                     | *)
print_int x;;      (* prints 7 -----------+ *)
print_endline "";; (*                       *)
```

OCaml is a **lexically scoped** language: can determine name/value bindings purely from source code, not based on dynamic context.

Immediate Immutability Concerns

Q: What's with the whole `let/in` thing?

Stems for Mathematics such as...

Pythagorean Thm: Let c be they length of the hypotenuse of a right triangle and let a, b be the lengths of its other sides. Then the relation $c^2 = a^2 + b^2$ holds.

Q: If I can't change bindings, how do I get things done?

A: Turns out you can get lots done but it requires an adjustment of thinking. Often there is **recursion** involved.

Q: `let/in` seems bothersome. Advantages over mutability?

A: Yes. Roughly they are

- It's easier to formally / informally verify program correctness
- Immutability opens up possibilities for parallelism

Q: Can I still write imperative code when it seems appropriate?

A: Definitely. Some problems in CMSC330 will state constraints like "must not use mutation" to which you should adhere or risk deductions.

Exercise: Collatz Sans Mutation

```
(* collatz_rec.ml: *)
open Printf;;
let verbose = true;;
let collatz start maxsteps =

  let rec collatz_step cur step =
    if verbose then
      printf "%3d: %5d\n" step cur;
    let rem = cur mod 2 in
    match (cur,step=maxsteps,rem) with
      (1,_,   _) -> (cur,step)
    | (_,true,_) -> (cur,step)
    | (_,_,   0) -> collatz_step (cur/2)   (step+1)
    | (_,_,   _) -> collatz_step (cur*3+1) (step+1)
  in
  if verbose then
    begin
      printf "start: %d maxsteps %d\n" start maxsteps;
      printf "Step  Current\n";
    end;
  collatz_step start 0
;;
let _ =
  print_string "Collatz start val:\n";
  let start = read_int () in
  let (final,steps) = collatz start 500 in
  printf "Reached %d after %d iters\n" final steps;
;;
```

Consider this alternate version of our first Collatz sequence computation. How does it compute the sequence? See any new tricks?

Answers: Collatz Sans Mutation

- Uses a "helper function" which is nested in the local scope of the outer function as in

```
let collatz start maxsteps =      (* outer function *)

  let rec collatz_step cur step = (* nested / inner function *)
    ...                           (* can access outer func *)
  in                              (* vars like maxsteps *)
  ...
  collatz_step start 0
;;
```

- `collatz_step` uses recursion to generate the Collatz sequence
  - Doesn't that risk stack overflow for long Collatz sequences?
  - *Not with the tail call optimization used by most functional languages, OCaml and Scheme included*
- Recursive functions can be set up with a `let rec ...` binding *(annoying that this is not the default but no bigee)*
- Inner function uses somewhat more complex `match/with` statement for case analysis