CSCI 2041: Basic OCaml Syntax and Features

Chris Kauffman

Last Updated:
Mon Sep 10 08:56:29 CDT 2018

Logistics

- OCaml System Manual: 1.1 - 1.3
- Practical OCaml: Ch 1-2
- OCaml System Manual: 25.2 (Pervasives Modules)
- Practical OCaml: Ch 3, 9

Goals

- Syntax and Semantics in OCaml
- Basic Polymorphism with Functions

Lab01

- First meetings on Mon/Tue
- Required attendance

Assignment 1

- Will go up over the weekend
- Due at **end of weeks** listed on schedule
- Likely around Sunday 9/16

Every Programming Language

Look for the following as it should almost always be there

- ☐ Comments
- ☐ Statements/Expressions
- ☐ Variable Types
- ☐ Assignment
- ☐ Basic Input/Output
- ☐ Function Declarations
- ☐ Conditionals (if-else)
- ☐ Iteration (loops)
- ☐ Aggregate data (arrays, structs, objects, etc)
- ☐ Library System

Comments

- Surround by `(* comment *)`
- Comment may span multiple lines until closing `*)`
- Will often provide commented programs to assist with learning
- Examples:

```
(* basics.ml : some basic OCaml syntax *)
let x = 15;;                    (* bind x to an integer *)
let y = "hi there";;            (* bind y to a string *)

(* Function to repeatedly print  *)
let repeat_print n str =        (* bind repeat_print to a function *)
  for i=1 to n do               (* of an integer and a string which *)
    print_endline str;          (* repeatedly prints the string *)
  done
;;
```

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 4.07.0

# 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;;
```

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
```

Answer: 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

```
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;;          (* print a, it is well defined *)
11
12 print_endline x;;          (* x is not defined *)
```

Answers: Fix Binding Problem

On 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

```
1  let x = 5 in        (* local: bind FIRST-x to 5 *)
2  let x = x+5 in      (* local: SECOND-x is FIST-x+5 *)
3  print_int x;;       (* prints 10: most recent x, SECOND-x  *)
4                      (* top-level: SECOND-x out of scope *)
5  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

```
 1  let x = 7;;
 2  let y =
 3    let z = x+5 in
 4    let x = x+2 in
 5    let z = z+2 in
 6    z+x;;
 7
 8  print_int y;;
 9  print_endline "";;
10
11  print_int x;;
12  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

```
 1  let x = 7;;          (* top-level x <-------+ *)
 2  let y =              (* top-level y <---+   | *)
 3    let z = x+5 in     (* z = 12 = 7+5    |   | *)
 4    let x = x+2 in     (* x =  9 = 7+2    |   | *)
 5    let z = z+2 in     (* z = 14 = 12+2   |   | *)
 6    z+x;;              (* 14+9 = 23 ------+   | *)
 7                       (* end local scope |   | *)
 8  print_int y;;        (* prints 23 ------+   | *)
 9  print_endline "";;   (*                     | *)
10                       (*                     | *)
11  print_int x;;        (* prints 7 ----------+ *)
12  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 2041 will state constraints like "must not use mutation" to which you should adhere or risk deductions.

Built-in Fundamental Types of Data

The usual suspects are present and conveniently named

```
> ocaml
        OCaml version 4.06.0

# let life = 42;;                          (* int : 31-bit are 63-bit *)
val life : int = 42                        (* integer (1 bit short??) *)

# let pie = 3.14159;;                      (* float  : 64-bit floating *)
val pie : float = 3.14159                  (* point number *)

# let greet = "Bonjour!";;                 (* string : contiguous array *)
val greet : string = "Bonjour!"            (* of character data *)

# let learning = true;;                    (* bool : Boolean value of *)
val learning : bool = true                 (* true or false only *)

# let result = print_endline greet;;       (* unit : equivalent to void *)
Bonjour!                                   (* in C/Java; side-effects only *)
val result : unit = ()                     (* such as printing or mutating *)

# result;;                                 (* Note that result has value (),
- : unit = ()                              (* NOT the output "Bonjour!" *)
```

Unit type and Printing

- The notation `()` means `unit` and is the return value of functions that only perform side-effects
- Primary among these are printing functions
  - Ex: `return_val` bound to `()` in code on right
- Don't usually care about `unit` so usually don't bind return values of printing functions
- Functions with no parameters are passed `()` to call them
  - Ex: `print_newline ()`

```
(* basic_printing.ml : printing and
   the unit value *)

let return_val =
  print_endline "hi there!\n";;
(* output: hi there! *)
(* val return_val : unit = () *)

(* built-in printing functions *)
print_string  "hi";; (* don't bother *)
print_int     5;;    (* binding unit *)
print_float   1.23;; (* return value *)
print_endline "done";;
(* output:
   hi51.23done
*)

print_int 7;;       (* pass unit to   *)
print_newline ();; (* functions with *)
print_int 8;;       (* no args like   *)
print_newline ();; (* print_newline  *)
(* output:
   7
   8
*)
```

Side-Effects and Local Scopes

- Side-effects only statements like printing can end with a single semi-colon; these should all have `unit` value
- Single semi-colons continue any existing local scope
- Double semi-colon ends top-level statements / local scopes

```
 1  (* basic_printing.ml : local scope, print variables *)
 2  let x = "hi" in                (* local scope with x *)
 3  let y = 5 in                   (* .. and y *)
 4  print_string  "string: ";      (* single semi-colon for *)
 5  print_string  x;               (* side-effects only statements *)
 6  print_newline ();              (* that continue the local scope *)
 7  print_string  "int: ";
 8  print_int     y;               (* y still defined *)
 9  print_newline ();
10  let z = 1.23 in                (* add z to local scope *)
11  print_string  "float: ";
12  print_float   z;
13  print_newline ();
14  print_endline "done";
15  ;;                             (* end top-level statement *)
16  (* x,y,z no longer in scope *)
```

Exercise: Output or Error?

To the right are 3 code blocks if

- Code compiles correctly, describe its output OR
- Won't compile and describe the error

```
 1  (* Block 1 *)
 2  let a = 7 in
 3  print_endline "get started";
 4  let b = 12 in
 5  print_endline "another line";
 6  print_int (a+b);
 7  print_newline ();
 8  ;;
 9
10  (* Block 2 *)
11  let c = 2 in
12  let d = a + 2 in
13  print_int d;
14  print_newline ();
15  ;;
16
17  (* Block 3 *)
18  let a = 9
19  ;;
20  print_endline "last one";
21  print_int a;
22  print_newline ();
23  ;;
```

Answers: Output or Error?

```
 1  (* Block 1 *)                    (* OK *)
 2  let a = 7 in                     (* a in local scope *)
 3  print_endline "get started";     (* continue local scope *)
 4  let b = 12 in                    (* b in local scope *)
 5  print_endline "another line";    (* continue local scope *)
 6  print_int (a+b);                 (* a and b still in scope, all is well *)
 7  print_newline ();
 8  ;;                               (* end local scope, a b undefined *)
 9
10  (* Block 2 *)                    (* ERROR *)
11  let c = 2 in                     (* c in local scope *)
12  let d = a + c in                 (* ERROR: no binding for a *)
13  print_int d;
14  print_newline ();
15  ;;
16
17  (* Block 3 *)                    (* OK *)
18  let a = 9                        (* a bound to 9 *)
19  ;;                               (* at the top level *)
20  print_endline "last one";
21  print_int a;                     (* a is a top-level binding, in scope *)
22  print_newline ();
23  ;;
```

This is Ridiculous

> *So you're telling me just to print an integer on its own line I've got to write* `print_int i;` *followed by* `print_newline ();`*? That's ridiculous. I've about had it with OCaml already.*

- Yup, printing with standard functions is pretty lame
- Folks with C experience, advanced Java experience, or perhaps Python know a better way to print an integer, a string, and a float in a one liner.
- Q: What's our favorite way to *print f*ormatted output?

Printf Module and `printf` function

- Output with previous functions is extremely tedious
- `printf` makes this much more succinct

```
 1  (* printf_demo.ml : demonstrate the printf function
 2     for succinct output *)
 3
 4  open Printf;;       (* access functions from Printf module *)
 5  (* function printf is now available *)
 6
 7  printf "hi there!\n";;
 8  printf "sub in an int: %d\n" 17;;
 9  (* Output:
10     hi there!
11     sub in an int: 17
12  *)
13
14  printf "string: %s integer %d float %f done\n"
15                 "hi"          5       1.23;;
16  (* output:
17     string: hi integer 5 float 1.230000 done
18  *)
```

`printf` gets type checked (!!!)

- OCaml's compiler checks the types of substitutions in `printf`
- After years of #^%@-ing this up in C and Java, I just about cried with joy when I found this out

```
> cat -n printf_typecheck.ml
     1  (* Demonstrate compiler checking substitution
     2     types in a printf format string  *)
     3  open Printf;;
     4
     5  let x = 42 in
     6  let y = 1.23 in
     7  printf "x is %f and y is %d" x y;;

> ocamlc printf_typecheck.ml
File "printf_typecheck.ml", line 7, characters 29-30:
Error: This expression has type int but an expression
       was expected of type float
```

Compare Printing: Standard vs. `printf`

Standard Functions

```
 1  let x = "hi" in
 2  let y = 5 in
 3  print_string  "string: ";
 4  print_string  x;
 5  print_newline ();
 6  print_string  "int: ";
 7  print_int     y;
 8  print_newline ();
 9  let z = 1.23 in
10  print_string  "float: ";
11  print_float   z;
12  print_newline ();
13  print_endline "done";
14  ;;
```

`printf`

```
1  let x = "hi" in
2  let y = 5 in
3  printf "string: %s\n" x;
4  printf "int: %d\n" y;
5  let z = 1.23 in
6  printf "float: %f\n" z;
7  printf "done\n";
8  ;;
```

- Kauffman is a big fan of `printf` in any language
- Often the fastest, easiest way to generate formatted output
- Will use it extensively in the course and others so well worth learning conversions specifiers associated format strings

Type Checking is a Harsh Master

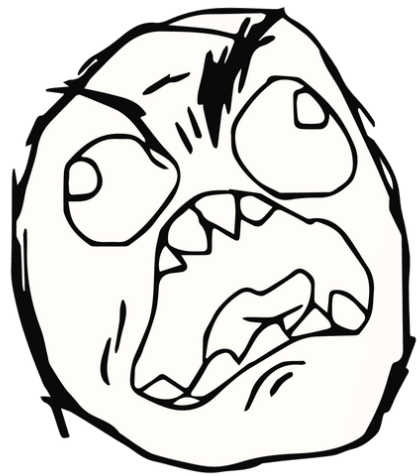
- Likely to encounter the following minor irritation early on

```
> ocaml
        OCaml version 4.07.0
# 1 + 5;;
- : int = 6
# 1.5 + 5.5;;
Characters 0-3:
  1.5 + 5.5;;
  ^^^
Error: This expression has type float but
an expression was expected of type int
```

- Type checking is **extremely thorough**
- So thorough that even basic arithmetic operations are specifically typed

```
# (+);;
- : int -> int -> int = <fun>
```

- `+` is a function that takes 2 `ints` and produce an `int`
- It won't work for `floats`



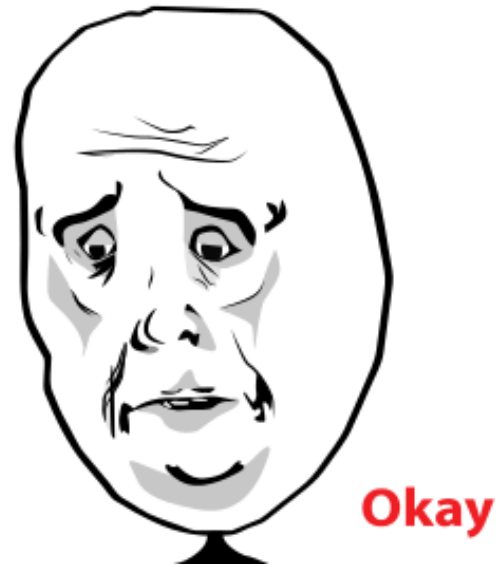
Integer vs. Floating Point Arithmetic

- Arithmetic operators `+ - * /` only work for `int` types
- Dotted operators `+. -. *. /.` only work for `float` types

```
# 1 + 5 * 2;;
- : int = 11
# 1.5 +. 5.5 *. 2.0;;
- : float = 12.5
```

- While many find it initially irritating, this is true to the underlying machine
  - Int/Float numbers differ in bit layout
  - Int/Float arithmetic instructions use different CPU circuitry
  - Conversions between Int/Float are CPU instructions that take time; OCaml reflects this with conversion functions

```
# float_of_int 15;;
- : float = 15.
# int_of_float 2.95;;
- : int = 2
```



Refs and Mutation

- Mutable bindings are often done via **references**
- These are set up to "point" at a mutable data location
- Initialize with `ref x` with `x` as the initial value
- Alter the location with ref assignment syntax `x := y;`
- Retrieve ref data with `!x`

```
(* ref_summing.ml : demonstrate use of mutable refs to sum *)
open Printf;;

let sum_1_to_n n =                  (* generate the sum of numbers 1 to n *)
  let sum = ref 0 in                (* initialize ref to 0 *)
  for i=1 to n do                   (* loop *)
    let next = !sum + i in          (* add on i to current sum *)
    sum := next;                    (* assign sum to next; RETURN TYPE unit *)
    (* sum := !sum + i; *)          (* one-liner version *)
  done;
  !sum                              (* return value of sum *)
;;
let sum10 = sum_1_to_n 10 in
let sum50 = sum_1_to_n 50 in
printf "summing 1 to 10 gives %d\n" sum10;
printf "summing 1 to 50 gives %d\n" sum50;
;;
```

Exercise: Function Definitions

- Have seen this several times: functions can be defined by binding a name with parameters
- Functions always have a type that gives their parameters and return type
- Notation for this in ML is with "arrows" like these examples

```
int -> float
(* 1 int param, return float *)

int -> int -> float
(* 2 int params, return float *)

string -> int -> unit
(* string and int params,
   return nothing *)
```

- What are the types of the following functions?

```
(* file func_defs.ml : simple
   function definitions *)

(* do some math  *)
let do_math x y =
  let z = x + y in
  let w = z*z + z in
  w
;;

(* make a word *)
let do_english s =
  let suffix = "-alicious" in
  s^suffix
;;

open Printf;;
(* Function to repeatedly print *)
let repeat_print n str =
  for i=1 to n do
    printf "%s\n" str;
  done
;;
```

Answers: Function Definitions

```
(* file func_defs.ml : simple
   function definitions *)

(* do some math  *)
let do_math x y =
  let z = x + y in
  let w = z*z + z in
  w
;;

(* make a word *)
let do_english s =
  let suffix = "-alicious" in
  s^suffix
;;

open Printf;;
(* Function to repeatedly print *)
let repeat_print n str =
  for i=1 to n do
    printf "%s\n" str;
  done
;;
```

Invoking the compiler as `ocamlc -i` will show the inferred types associated with top-level bindings like functions.

```
> ocamlc -i func_defs.ml
val do_math : int -> int -> int
val do_english : string -> string
val repeat_print : int -> string -> unit
```

Recursive Functions

Nesting Function Definitions

Lexical Binding and Scope Trickery

`for` loops

Arrays

Lists