CSCI 2041: Basic OCaml Syntax and Features

Chris Kauffman

Last Updated:
Wed Sep 12 14:37:38 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

Basic Syntax and Semantics in OCaml

Lab01

- First meetings on Mon/Tue
- Required attendance

Assignment 1

- Will go up over the weekend
- Due at **end of weeks** listed on schedule
- Due Monday 9/17

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

```
(* 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

On obvious fix is below

```
> cat -n local_is_local_fixed.ml
(* local_is_local_fixed.ml : fixes local binding
   error by making it a top-level binding
*)

let x = "hello";;             (* top-level binding *)

let a =                       (* top-level 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;;             (* 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 FIST-x+5 *)
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 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

```
(* basic_printing.ml : local scope, print variables *)
let x = "hi" in                (* local scope with x *)
let y = 5 in                   (* .. and y *)
print_string  "string: ";      (* single semi-colon for *)
print_string  x;               (* side-effects only statements *)
print_newline ();              (* that continue the local scope *)
print_string  "int: ";
print_int     y;               (* y still defined *)
print_newline ();
let z = 1.23 in                (* add z to local scope *)
print_string  "float: ";
print_float   z;
print_newline ();
print_endline "done";
;;                             (* end top-level statement *)
(* x,y,z no longer in scope *)
```

Exercise: Output or Error?

To the right are 3 code blocks. Determine:

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

```
(* Block 1 *)
let a = 7 in
print_endline "get started";
let b = 12 in
print_endline "another line";
print_int (a+b);
print_newline ();
;;

(* Block 2 *)
let c = 2 in
let d = a + 2 in
print_int d;
print_newline ();
;;

(* Block 3 *)
let a = 9
;;
print_endline "last one";
print_int a;
print_newline ();
;;
```

Answers: Output or Error?

```
(* Block 1 *)                    (* OK *)
let a = 7 in                     (* a in local scope *)
print_endline "get started";     (* continue local scope *)
let b = 12 in                    (* b in local scope *)
print_endline "another line";    (* continue local scope *)
print_int (a+b);                 (* a and b still in scope, all is well *)
print_newline ();
;;                               (* end local scope, a b undefined *)

(* Block 2 *)                    (* ERROR *)
let c = 2 in                     (* c in local scope *)
let d = a + c in                 (* ERROR: no binding for a *)
print_int d;
print_newline ();
;;

(* Block 3 *)                    (* OK *)
let a = 9                        (* a bound to 9 *)
;;                               (* at the top level *)
print_endline "last one";
print_int a;                     (* a is a top-level binding, in scope *)
print_newline ();
;;
```

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

```
(* printf_demo.ml : demonstrate the printf function
   for succinct output *)

open Printf;;       (* access functions from Printf module *)
(* function printf is now available *)

printf "hi there!\n";;
printf "sub in an int: %d\n" 17;;
(* Output:
   hi there!
   sub in an int: 17
*)

printf "string: %s integer %d float %f done\n"
                "hi"          5      1.23;;
(* output:
   string: hi integer 5 float 1.230000 done
*)
```

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

```
let x = "hi" in
let y = 5 in
print_string  "string: ";
print_string  x;
print_newline ();
print_string  "int: ";
print_int     y;
print_newline ();
let z = 1.23 in
print_string  "float: ";
print_float   z;
print_newline ();
print_endline "done";
;;
```

printf

```
let x = "hi" in
let y = 5 in
printf "string: %s\n" x;
printf "int: %d\n" y;
let z = 1.23 in
printf "float: %f\n" z;
printf "done\n";
;;
```

- 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

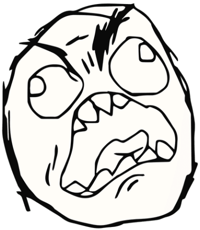
- Likely to encounter the following minor irritation early on

```
> 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 `int`s and produce an `int`
- It won't work for `float`s



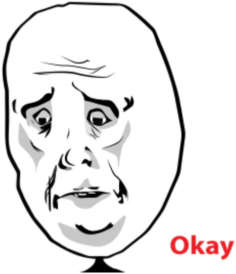
Integer vs. Floating Point Arithmetic

- 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
```



Annotating Types by Hand

- Can annotate types by hand using `: atype` as shown below
- Compiler complains if it disagrees
- Will examine this again wrt function types

```
(* type_annotations.ml : show type annotation syntax of colon
   for simple definitions *)

let a : int = 7;;              (* annotate a as int *)
let b = 7;;                    (* b inferred as int *)

(* fully annotated version *)
let c : int =                  (* annotate c as int *)
  let x : string = "hi" in
  let y : string = "bye" in
  let z : string = x^y in      (* concatenate *)
  String.length z              (* return string length : int *)
;;

(* fully inferred version *)
let d =                        (* inferred c as int <-----+  *)
  let x = "hi" in              (* inferred x as string    |  *)
  let y = "bye" in             (* inferred y as string    |  *)
  let z = x^y in               (* inferred z as string    |  *)
  String.length z              (* return string length : int *)
;;
```