Chapter 2: The Basics of OCaml

- Five essential components to learning a language:
    1. Syntax: defined rules that constitute whether a program in the language is well-formed, including the keyword, restrictions, formatting, punctuation, and operators.
    2. Semantics: the rules that define the behavior of programs, or the meaning of a program.
    3. Idioms: common approaches to using the language features.
    4. Libraries: bundles of code that have been pre-written to offer more productivity to other programmers.
    5. Tools: language implementations provide compiler/interpreter as a tool for interacting with the computer using the language

2.1 The OCaml Toplevel

Start utop from terminal: utop

;; - double semi-clone, tells utop user done entering expression, OCaml should process the code

```
utop # 3110;;
```

```
utop # 3110;;
- : int = 3110
```

read response from right to left(OCaml evaluates the expression, informs the resulting value, and the value's type)

- 3110 is the value
- int is the type of the value
- value wasn't given a name, hence the -(dash symbol)

```
utop # true;;
- : bool = true
─( 16:36:00 )─< command 5
utop # false;;
- : bool = false
```

booleans

```
utop # 310 > 120 ;;
- : bool = true
```

integer expression

```
utop # "Hello world";;
- : string = "Hello world"
```

string

```
utop # "Today" ^ "is" ^ "tuesday";;
- : string = "Todayistuesday"
```

concatenate string

```
utop # 15.25 *. 45.5;;
- : float = 693.875
```

float number multiplication (*.)

```
utop # (3110 : int);;
- : int = 3110
```

replace 3110 with value & int with datatype

```
utop # let a = 342;;
val a : int = 342
—( 17:08:39 )—< comman
utop # a;;
- : int = 342
—( 17:08:44 )—< comman
```

- value of 342, whose type was int, and is bound to the name a(from right to left)
- value name x, which has type int, and is equal to 342(from left to right)

```
utop # a+y;;
- : int = 3452
```

add values from name variables(name bounds)

```
utop # let increaseByFive x = x+ 5;;
val increaseByFive : int -> int = <fun>
—( 17:12:25 )—< command 8 >——————————————
utop # increaseByFive 5;;
- : int = 10
—( 17:23:37 )—< command 9 >——————————————
utop # increaseByFive 15;;
- : int = 20
—( 17:23:52 )—< command 10 >—————————————
utop # increaseByFive (30);;
- : int = 35
—( 17:24:03 )—< command 11 >—————————————
utop # increaseByFive(increaseByFive(1));;
- : int = 11
```

OCaml functions

```
utop # 3.14 *. (float_of_int 3);;
- : float = 9.42
```

convert between int & float. Built-in functions: int_of_float and float_of_int

Datatype conversion (from x to String)

```
utop # string_of_int 43;;
- : string = "43"
```

string_of_int method to convert to Strings (from int to string)

```
utop # string_of_float 44.34;;
- : string = "44.34"
```

string_of_float method to convert to string (from float to string)

```
utop # string_of_bool true;;
- : string = "true"
```

string_of_bool method to convert to string (from boolean to string)

```
utop # String.make 3 'z';;
- : string = "zzz"
```

String.make to convert to string (from char to string)

Datatype conversion (from String to x)

```
utop # int_of_string "123456";;
- : int = 123456
```

int_of_string to convert to int datatype (from string to int)

```
utop # float_of_string "45.24334";;
- : float = 45.24334
```

float_of_string to convert to float datatype (from string to float)

```
utop # bool_of_string "true";;
- : bool = true
```

bool_of_string from to convert to boolean datatype (from string to boolean)

```
utop # "HelloWorld".[5];;
- : char = 'W'
```

individuals of a string can be accessed by a 0-based index. Syntax: “String”.[#];;

```
utop # if 3+1 > 5 then "true" else "false";;
- : string = "false"
```

if expressions

```
utop # let x = 40 in x+7;;
- : int = 47
```

another way of let expressions

```
utop # let a = "Hello";;
val a : string = "Hello"
─( 08:49:37 )─< command 18 >─
utop # let b = "World";;
val b : string = "World"
─( 09:01:54 )─< command 19 >─
utop # let c = a ^ b;;
val c : string = "HelloWorld"
```

string concatenation by let expressions

```
utop # let x = 5 in x + 2;;
- : int = 7
```

dynamic semantics

```
utop # let increaseByTwo = fun num -> num + 2;;
val increaseByTwo : int -> int = <fun>
─( 17:50:13 )─< command 3 >──────────────
utop # increaseByTwo 22;;
- : int = 24
```

named function(increaseByTwo), num(user passed-in parameter)

```
utop # let increaseBySeven x = x + 7;;
val increaseBySeven : int -> int = <fun>
─( 18:58:37 )─< command 3 >──────────────
utop # increaseBySeven 6;;
- : int = 13
```

another way of direct functions (more simple solution)

```
utop # let avg x y = (x +. y) /. 2.;;
val avg : float -> float -> float = <fun>
─( 19:02:10 )─< command 10 >──────────────
utop # avg 18.2 3.2;;
- : float = 10.7
```

direct functions with 2 parameters

```
─( 19:45:49 )─< command 1
utop # (fun x -> x+1) 2;;
- : int = 3
```

anonymous functions: inside the parenthesis is the function to be applied, #2 is the argument to be applied to

2.4.5 Polymorphic Functions

Identity function: the function that simply returns its input

```
utop # let id x = x;;
val id : 'a -> 'a = <fun>
—( 05:29:04 )—< command 2
utop # id 5;;
- : int = 5
—( 05:29:15 )—< command 3
utop # id "hello world";;
- : string = "hello world'
—( 05:29:20 )—< command 4
utop # id true;;
- : bool = true
```

returns the datatype and value to what the user pass in. Type of x would be ‘a’, pronounced ‘alpha’, similar to Java’s <T> type variable. Stands for unknown variable similar to know variable. Close to Java’s generics, polymorphism. Behave in many ways.

```
utop # let id (x:int) : int = x;;
val id : int -> int = <fun>
—( 05:39:41 )—< command 9 >————————————————————————————————
utop # id 5;;
- : int = 5
—( 05:41:14 )—< command 10 >———————————————————————————————
utop # id "hello world";;
Error: This expression has type string but an expression was expected of type
         int
```

restricts type to a polymorphic function. Example: restricting the data type of x to be an integer, disallowing other datatypes to be passed in.

2.4.6 Labeled and Optional Arguments: label arguments to its functions

```
utop # let addFunction ~num1:arg1 ~num2:arg2 = arg1 + arg2;;
val addFunction : num1:int -> num2:int -> int = <fun>
—( 05:52:31 )—< command 17 >——————————————————————————
utop # addFunction ~num1:5 ~num2:7;;
- : int = 12
```

labeling arguments to its functions,

```
utop # let subtractNumbers ~num1 ~num2 = num1 - num2;;
val subtractNumbers : num1:int -> num2:int -> int = <fun>
─( 05:55:52 )─< command 21 >─────────────────────────
utop # subtractNumbers ~num1: 5 ~num2:3;;
- : int = 2
```

shorthand for the above equivalent syntax

```
let f ~name1:(arg1 : int) ~name2:(arg2 : int) = arg1 + arg2
```

syntax to write both labeled argument and explicit type annotation

```
utop # f ~name1:2 ~name2:5;;
- : int = 7
```

```
utop # let f ?num:(arg1=10) arg2 = arg1 + arg2;;
val f : ?num:int -> int -> int = <fun>
─( 06:03:11 )─< command 35 >─────────────────────
utop # f ~num:2 9;;
- : int = 11
─( 06:04:13 )─< command 36 >─────────────────────
utop # f 7;;
- : int = 17
```

optional arguments, a default value must be provided. If user passes in a parameter for the optional value, that value will be used. Otherwise, the default value will be used.

2.4.7 Partial Application

```
utop # let add x y = x+y;;
val add : int -> int -> int = <fun>
—( 06:25:47 )—< command 40 >——————————
utop # let add5 = add 5;;
val add5 : int -> int = <fun>
—( 06:26:04 )—< command 41 >——————————
utop # add5 2;;
- : int = 7
```

2.4.8 Function Associativity

```
let f x1 x2 ... xn = e
```

is semantically equivalent to

```
let f =
  fun x1 ->
    (fun x2 ->
       (...
          (fun xn -> e)...))
```

```
e1 e2 e3 e4
```

really means the same as

```
((e1 e2) e3) e4
```

2.4.9 Operators as Functions

```
utop # ( + ) 4 2;;
- : int = 6
—( 06:45:25 )—< comm
utop # ( - ) 6 4 ;;
- : int = 2
—( 06:49:49 )—< comm
utop # ( * ) 5 2 ;;
- : int = 10
—( 06:49:56 )—< comm
utop # ( / ) 140 2;;
- : int = 70
```

using operators as functions example

```
utop # let ( ^^ ) x y = max x y;;
val ( ^^ ) : 'a -> 'a -> 'a = <fun>
—( 06:51:31 )—< command 55 >————————
utop # 5 ^^ 92;;
- : int = 92
```

max function, compares two numbers and return the maximum number using operators as functions

2.4.10 Tail Recursion

2.6 Printing

2.6.1 Unit

Unit: when programmer needs to take an argument or return a value, but there's no interesting value to pass or return. Similar to Java's void. A datatype with only one unit, its value is ()

2.6.2 Semicolon

```
utop # let _ = print_endline "Today" in
let _ = print_endline "is" in
print_endline "monday";;
Today
is
monday
- : unit = ()
```

Nested let expressions, print one thing after another.

2.6.4 Printf

```
utop # let print_msg week date = Printf.printf "%s %F \n " week date;;
val print_msg : string -> float -> unit = <fun>
─( 15:09:22 )─< command 73 >─────────────────────────────────────────
utop # print_msg "Monday" 9.11;;
Monday 9.11
 - : unit = ()
```

use OCaml's Printf module to print statements using format specifier

2.7 Debugging

Rob Miller:

1. The first defense against bugs is to make them impossible.
2. The second defense against bugs is to use tools to find them.
3. The third defense against bugs is to make them immediately visible.
4. The fourth defense agains bugs is extensive testing.

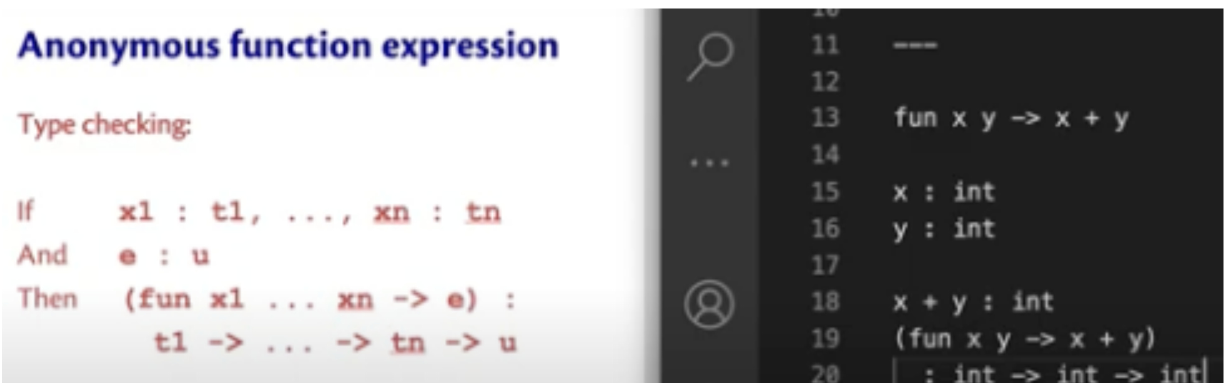
2.7.3 Debugging in OCaml

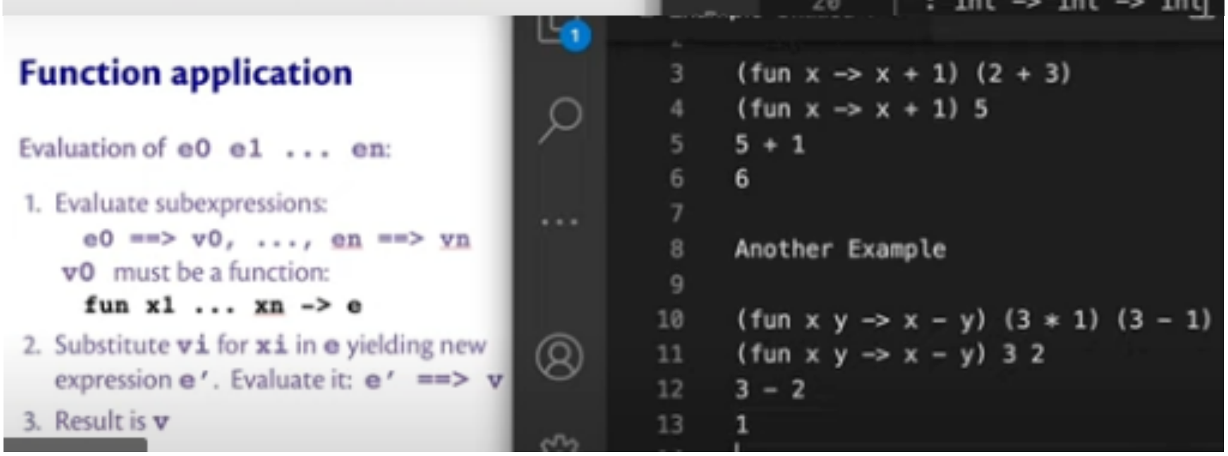
- print statements: print statements to ascertain the value of a variable
- function traces: use the #trace directive to see the trace of recursive calls and returns for a function
- debugger: debugging tool ocamldebug

Additional Notes:
- double semicolon is needed for interactive sessions(terminal) at top-level, no need double semicolon to write in .ml file
- two operator for Caml arithmetic operators : int & double
- two equality operators in OCaml, = and ==.   Inequality operators <> and != .   = and <> examine structural equality, whereas == and != examine physical equality
- everything is strictly a function, not a method
- control + l to “clear” terminal screen
- state recursive function definition: let rec f…

```
let inc = fun x -> x + 1
let inc x = x + 1
```

function syntactically different but semantically equivalent





Datatypes:
- bool: Booleans: written as true and false. Short-circuit conjunction && and

disjunction ll operators are available
- char: Characters: written with single quotes, such as ‘a’, ‘b’, ‘c’. Can convert characters to and from integers with char_of_int and int_of_char.
- string: Strings: sequence of characters, written with double quotes, such as “abcd”. Stain concatenation operator is ^