The OCaml Language

Syntax

Implementations are in `.ml` files, interfaces are in `.mli` files.
Comments can be nested, between delimiters `(*...*)`
Integers: `123`, `1_000`, `0x4533`, `0o773`, `0b1010101`
Chars: `'a'`, `'\255'`, `'\xFF'`, `'\n'` Floats: `0.1`, `-1.234e-34`

Data Types

`unit`	Void, takes only one value: `()`
`int`	Integer of either 31 or 63 bits, like `32`
`int32`	32 bits Integer, like `32l`
`int64`	64 bits Integer, like `32L`
`float`	Double precision float, like `1.0`
`bool`	Boolean, takes two values: `true` or `false`
`char`	Simple ASCII characters, like 'A'
`string`	Strings of chars, like `"Hello"`
`'a list`	Lists, like `head :: tail` or `[1;2;3]`
`'a array`	Arrays, like `[|1;2;3|]`
t_1 `* ... *` t_n	Tuples, like `(1,"foo", 'b')`

Constructed Types

<table>
<tr><td><code>type record =</code></td><td>new record type</td></tr>
<tr><td><code>{ field1 : bool;</code></td><td>immutable field</td></tr>
<tr><td><code>mutable field2 : int; }</code></td><td>mutable field</td></tr>
<tr><td><code>type enum =</code></td><td>new variant type</td></tr>
<tr><td><code>| Constant</code></td><td>Constant constructor</td></tr>
<tr><td><code>| Param of string</code></td><td>Constructor with arg</td></tr>
<tr><td><code>| Pair of string * int</code></td><td>Constructor with args</td></tr>
</table>

Constructed Values

```
let r = { field1 = true; field2 = 3; }
let r' = { r with field1 = false }
r.field2 <- r.field2 + 1;
let c = Constant
let c' = Param "foo"
let c'' = Pair ("bar",3)
```

References, Strings and Arrays

`let x = ref 3`	integer reference (mutable)
`x := 4`	reference assignation
`print_int !x;`	reference access
`s.[0]`	string char access
`s.[0] <- 'a'`	string char modification
`t.(0)`	array element access
`t.(0) <- x`	array element modification

Imports — Namespaces

`open Unix;;`	global open
`let open Unix in` *expr*	local open
`Unix.(`*expr*`)`	local open

Functions

<table>
<tr><td><code>let f x = expr</code></td><td>function with one arg</td></tr>
<tr><td><code>let rec f x = expr</code></td><td>recursive function</td></tr>
<tr><td>apply:</td><td><code>f x</code></td></tr>
<tr><td><code>let f x y = expr</code></td><td>with two args</td></tr>
<tr><td>apply:</td><td><code>f x y</code></td></tr>
<tr><td><code>let f (x,y) = expr</code></td><td>with a pair as arg</td></tr>
<tr><td>apply:</td><td><code>f (x,y)</code></td></tr>
<tr><td><code>List.iter (fun x -> e) l</code></td><td>anonymous function</td></tr>
<tr><td><code>let f= function None -> act</code></td><td>function definition</td></tr>
<tr><td><code>| Some x -> act</code></td><td>by cases</td></tr>
<tr><td>apply:</td><td><code>f (Some x)</code></td></tr>
<tr><td><code>let f ~str ~len = expr</code></td><td>with labeled args</td></tr>
<tr><td>apply:</td><td><code>f ~str:s ~len:10</code></td></tr>
<tr><td>apply (for <code>~str:str</code>):</td><td><code>f ~str ~len</code></td></tr>
<tr><td><code>let f ?len ~str = expr</code></td><td>with optional arg (<code>option</code>)</td></tr>
<tr><td><code>let f ?(len=0) ~str = expr</code></td><td>optional arg default</td></tr>
<tr><td>apply (with omitted arg):</td><td><code>f ~str:s</code></td></tr>
<tr><td>apply (with commuting):</td><td><code>f ~str:s ~len:12</code></td></tr>
<tr><td>apply (<code>len: int option</code>):</td><td><code>f ?len ~str:s</code></td></tr>
<tr><td>apply (explicitely ommited):</td><td><code>f ?len:None ~str:s</code></td></tr>
<tr><td><code>let f (x : int) = expr</code></td><td>arg has constrainted type</td></tr>
<tr><td><code>let f : 'a 'b. 'a*'b -> 'a</code></td><td>function with constrainted</td></tr>
<tr><td><code>= fun (x,y) -> x</code></td><td>polymorphic type</td></tr>
</table>

Modules

`module M = struct .. end`	module definition
`module M: sig .. end= struct .. end`	module and signature
`module M = Unix`	module renaming
`include M`	include items from
`module type Sg = sig .. end`	signature definition
`module type Sg = module type of M`	signature of module
`let module M = struct .. end in ..`	local module
`let m = (module M : Sg)`	to 1^{st}-class module
`module M = (val m : Sg)`	from 1^{st}-class module
`module Make(S: Sg) = struct .. end`	functor
`module M = Make(M')`	functor application

Module type items:
`val`, `external`, `type`, `exception`, `module`, `open`, `include`, `class`

Pattern-matching

<table>
<tr><td><code>match expr with</code></td><td></td></tr>
<tr><td><code>| pattern -> action</code></td><td></td></tr>
<tr><td><code>| pattern when guard -> action</code></td><td>conditional case</td></tr>
<tr><td><code>| _ -> action</code></td><td>default case</td></tr>
</table>

Patterns:

<table>
<tr><td><code>| Pair (x,y) -></code></td><td>variant pattern</td></tr>
<tr><td><code>| { field = 3; _ } -></code></td><td>record pattern</td></tr>
<tr><td><code>| head :: tail -></code></td><td>list pattern</td></tr>
<tr><td><code>| [1;2;x] -></code></td><td>list-pattern</td></tr>
<tr><td><code>| (Some x) as y -></code></td><td>with extra binding</td></tr>
<tr><td><code>| (1,x) | (x,0) -></code></td><td>or-pattern</td></tr>
</table>

Conditionals

Structural	Physical	
`=`	`==`	Polymorphic Equality
`<>`	`!=`	Polymorphic Inequality

Polymorphic Generic Comparison Function: `compare`

	x < y	x = y	x > y
`compare x y`	-1	0	1

Other Polymorphic Comparisons : `>`, `>=`, `<`, `<=`

Loops

```
while cond do ... done;
for var = min_value to max_value do ... done;
for var = max_value downto min_value do ... done;
```

Exceptions

`exception MyExn`	new exception
`exception MyExn of t * t'`	same with arguments
`exception MyFail = Failure`	rename exception with args
`raise MyExn`	raise an exception
`raise (MyExn (`*args*`))`	raise with args
`try` *expression*	catch `MyException` if raised
`with Myn -> ...`	in *expression*

Objects and Classes

`class virtual foo x =`	virtual class with arg
`let y = x+2 in`	init before object creation
`object (self: 'a)`	object with self reference
`val mutable variable = x`	mutable instance variable
`method get = variable`	accessor
`method set z =`	
`variable <- z+y`	mutator
`method virtual copy : 'a`	virtual method
`initializer`	init after object creation
`self#set (self#get+1)`	
`end`	
`class bar =`	non-virtual class
`let var = 42 in`	class variable
`fun z -> object`	constructor argument
`inherit foo z as super`	inheritance and ancestor reference
`method! set y =`	method explicitely overriden
`super#set (y+4)`	access to ancestor
`method copy = {< x = 5 >}`	copy with change
`end`	
`let obj = new bar 3`	new object
`obj#set 4; obj#get`	method invocation
`let obj = object .. end`	immediate object

Polymorphic variants

<table>
<tr><td><code>type t = [`A | `B of int]</code></td><td>closed variant</td></tr>
<tr><td><code>type u = [`A | `C of float]</code></td><td></td></tr>
<tr><td><code>type v = [t | u |]</code></td><td>union of variants</td></tr>
<tr><td><code>let f : [< t] -> int = function</code></td><td>argument must be</td></tr>
<tr><td><code>| `A -> 0 | `B n -> n</code></td><td>a subtype of <code>t</code></td></tr>
<tr><td><code>let f : [> t] -> int = function</code></td><td><code>t</code> is a subtype</td></tr>
<tr><td><code>| `A -> 0 | `B n -> n | _ -> 1</code></td><td>of the argument</td></tr>
</table>