The OCaml Language

Syntax
Implementations are in .ml files, interfaces are in .mli files. Comments can be nested, between delimiters (*...*)
Integers: 123, 1_000, 0x433, 0x073, 0x1e10101
Chars: 'a', '255', '\xFF', '\n'
Flots: 0.1, -1.234e-34

Data Types

- unit: void, takes only one value: ()
- int: integer of either 31 or 63 bits, like 42
- int32: 32 bit Integer, like 42L
- int64: 64 bit Integer, like 42L
- float: double precision float, like 1.0
- bool: boolean, takes two values: true or false
- char: simple ASCII characters, like 'f'
- string: strings, like "Hello" or foo|Hello|foo
- mutable string of chars
- '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 Values

- type record = new record type
  { field1: bool; immutable field
  mutable field2: int; } mutable field
- type enum = new variant type
  | Constant: Constant constructor
  | Param of string: Constructor with arg
  | Pair of string * int: Constructor with args
  | Gadt : int -> enum:
  | Inline of ( x : int ):

Constructed Types

- 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)
- let c = Gadt 0
- let c = Inline { x = 3 }

References, Strings and Arrays

- let x = ref 3 integer reference
- x := 4 reference assignment
- print_int !x: reference access
- s.[0]: string char access
- t.(0): element access
- t.(0) <= x element modification

Imports — Namespaces

- open Unix
- let open Unix in expr
- Unix.(expr)

Functions

- let f x = expr
- let rec f x = expr
- function with one arg
- apply: f x
- with two args
- let f x y = expr
- apply: f x y
- with a pair as arg
- let f (x,y) = expr
- apply: f (x,y)
- anonymous function
- let f = function None -> act
  | Some -> act
  | by cases
  function definition
- let f = -str-len = expr
  apply: f (Some x)
  with labeled args
- let f ?len-str -expr
  apply: f (None)
  with optional arg (option)
- apply: f -str len
  with optional arg default
- apply: (for -str:)
  apply: f -str -len
- apply: [with omitted arg]:
- apply: [with commuting]:
- apply: (len: int option)
- apply: (explicitly omitted):
- apply: f
  arg has constrained type
- let f : a b. 'a' 'b' -> 'a
  fun (x,y) -> x
  function with constrained type

Modules

- module M = struct .. end
- module M = Unix
- include M
- module type Sg = module type of M
- module Make (S: Sg) = struct
- module M = (module M : Sg)
- module M = Unix
- module M:
- module Make(S: Sg) = struct .. end
- module M = Make(M')

Pattern-matching

- match expr with
  | pattern -> action
  | pattern when guard -> action
  | _ -> action
- conditional case
- default case

Patterns:

- | Pair (x,y) -> variant pattern
- | { field = 3; _ } -> record pattern
- | head :: tail -> list pattern
- | [1;2;x] -> list pattern
- | (Some x) as y -> or-pattern
- | (l.x) | (x,0) -> try&match
- | exception expr -> try&match

Conditionals

- Do NOT use on closures
- Structural | Physical

<table>
<thead>
<tr>
<th>&lt;=</th>
<th>=&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>Polyomorphic Inequality</td>
<td></td>
</tr>
<tr>
<td>Polyomorphic Equality</td>
<td></td>
</tr>
</tbody>
</table>

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
- exception MyExn of t * t'
- exception MyFail = Failure
- raise MyExn
- raise (MyExn (args))
- try expr
- with MyExn -> ...

Objects and Classes

- class virtual foo x =
- let y = x+2 in
- object (self: 'a)
- val mutable variable = x
- method get = variable
- method set z = variable
- method virtual copy = 'a
- method! set z = variable
- method virtual copy : 'a
- virtual class with arg
- init before object creation
- object with self reference
- mutable instance variable
- accessor
- class bar =
- let var = 42 in
- non-virtual class
- class variable
- constructor argument
- inheritance and ancestor reference
- mutable instance variable
- access to ancestor
- copy with change
- virtual method
- init after object creation
- class super =
- constructor
- method
- method set y = super
- super get (y)
- method virtual copy : 'a
- virtual method
- init after object creation
- immediate object

Polymorphic variants

- type t = ['A | 'B of int]
- type u = ['A | 'C of float]
- type v = ['t | 'u | 'f]
- union of variants
- let f : [< t ] -> int = function
- argument must be
- 'A -> 0
- method set y = super
- type 'A -> 0
- 'B n -> n
- of the argument
- t is a subtype of t
- 'A -> 0
- 'B n -> n
- _ -> 1