Standard ML and Objective Caml, Side by Side

This page gives a quick side by side comparison of program fragments in the two ML dialects Standard ML ('97 revision) and Objective Caml (version 3.12). It is primarily targetted at people who need to convert code between the two dialects. Where suitable we also mention common extensions to SML, or recent extensions of Ocaml. The comparison does not cover features that do not have an appropriate counter part in the sibling dialect (e.g. Ocaml's object sublanguage, SML's user-defined operator fixity, or advanced library issues).

The first section is an interaction with the respective toplevel system, in order to show the built-in types. The rest just consists of example expressions and definitions. Keywords and other reserved symbols are type-set in blue.

- Literals
- Expressions
- Functions
- Control Flow
- Value Declarations
- Type Declarations
- Matching
- Tuples
- Records
- References
- Comparisons
- List Functions
- String Functions
- Array Functions
- Input/Output
- Exceptions
- Local Declarations
- Structures
- Functors
- Signatures

Literals

SML	Ocaml
- 3; > val it = 3 : int	# 3;; - : int = 3
- 3.141; > val it = 3.141 : real	# 3.141;; - : float = 3.141
- "Hello world"; > val it = "Hello world" : string	# "Hello world";; - : string = "Hello world"
- #"J"; > val it = #"J" : char	# 'J';; - : char = 'J'

`- true;`
`> val it = true : bool`
	`# true;;`
	`- : bool = true`

`- ();`
`> val it = () : unit`
	`# ();;`
	`- : unit = ()`

`- (3, true, "hi");`
`> val it = (3, true, "hi") : int * bool * string`
	`# (3, true, "hi");;`
	`- : int * bool * string = 3, true, "hi"`

`- [1, 2, 3];`
`> val it = [1, 2, 3] : int list`
	`# [1; 2; 3];;`
	`- : int list = [1; 2; 3]`

`- #[1, 2, 3];`
`> val it = #[1, 2, 3] : int vector`
Standard does not have vector literals but most implementations support them – use library functions otherwise
	Does not have vectors – use arrays

Does not have array literals – use library functions	`# [|1; 2; 3|];;`
	`- : int array = [|1; 2; 3|]`

Expressions

SML	Ocaml
`~3*(1+7) div 2 mod 3`	`-3*(1+7)/2 mod 3`
`~1.0/2.0 + 1.9*x`	`-1.0 /. 2.0 +. 1.9 *. x`

`a orelse b andalso c`	`a || b && c`
	or (deprecated)
	`a or b & c`

Functions

SML	Ocaml

`fn f => fn x => fn y => f(x,y)`	`fun f -> fun x -> fun y -> f (x,y)`
	or
	`fun f x y -> f (x,y)`

`fn 0 => 0`
`  | n => 1`
	`function 0 -> 0`
	`         | n -> 1`

`f o g`	`fun x -> f (g x)`
`map SOME xs`	*Does not have first-class constructors – use function instead, e.g.* `map (fun x -> Some x) xs`

`map #2 triples`
`map #lab records`
	Does not have first-class selectors – use function instead, e.g.
	`map (fun (_,x,_) -> x) triples`
	`map (fun x -> x.lab) records`

`f (inputLine stdIn) (inputLine stdIn)`	*Evaluation order is undefined for application – use `let`, e.g.*
	`let line1 = read_line () in`
	`let line2 = read_line () in`
	`f line1 line2`

Control Flow

SML	**Ocaml**
`if 3 > 2 then "X" else "Y"`	`if 3 > 2 then "X" else "Y"`
`if 3 > 2 then print "hello" else ()`	`if 3 > 2 then print_string "hello"` *Note: expression has to have type* `unit`

`while true do`
`     print "X"`
	`while true do`
	`     print_string "X"`
	`done`

Does not have `for` *loops – use recursion or* `while`	`for i = 1 to 10 do`
	`     print_endline "Hello"`
	`done`

`(print "Hello ";`
`  print "world")`
	`print_string "Hello ";`
	`print_string "world"`
	or
	`(print_string "Hello ";`
	`  print_string "world")`
	or
	`begin`
	`     print_string "Hello ";`
	`     print_string "world"`
	`end`

Value Declarations

SML	**Ocaml**
`val name = expr`	`let name = expr`
`fun f x y = expr`	`let f x y = expr`

`val rec fib = fn n =>`
`     if n < 2`
`     then n`
`     else fib(n-1) + fib(n-2)`
or
`fun fib n =`
`     if n < 2`
`     then n`
`     else fib(n-1) + fib(n-2)`
	`let rec fib = fun n ->`
	`     if n < 2`
	`     then n`
	`     else fib (n-1) + fib (n-2)`
	or
	`let rec fib n =`
	`     if n < 2`
	`     then n`
	`     else fib (n-1) + fib (n-2)`

Type Declarations

SML	**Ocaml**
`type t = int -> bool`	`type t = int -> bool`
`type ('a,'b) assoc_list = ('a * 'b) list`	`type ('a,'b) assoc_list = ('a * 'b) list`
`datatype 'a option = NONE | SOME of 'a`	`type 'a option = None | Some of 'a`

`datatype t = A of int | B of u`
`withtype u = t * t`
	`type t = A of int | B of u`
	`and    u = t * t`

`datatype v = datatype t`	`type v = t = A of int | B of u`

`datatype complex = C of real * real`
`fun complex xy = C xy`
`fun coord (C xy) = xy`
	`type complex = C of float * float`
	`let complex (x,y) = C (x,y)`
	`let coord (C (x,y)) = (x,y)`
	or (note parentheses in type declaration)
	`type complex = C of (float * float)`
	`let complex xy = C xy`
	`let coord (C xy) = xy`

Matching

SML	Ocaml

`fun getOpt(NONE, d) = d`
`  | getOpt(SOME x, _) = x`
	`let get_opt = function`
	`    (None, d) -> d`
	`  | (Some x, _) -> x`

`fun getOpt (opt, d) =`
`  case opt of`
`      NONE => d`
`    | SOME x => x`
	`let get_opt (opt, d) =`
	`  match opt with`
	`      None -> d`
	`    | Some x -> x`

`fun take 0 xs = []`
`  | take n nil = raise Empty`
`  | take n (x::xs) = x :: take (n-1) xs`
	`let rec take n xs =`
	`  match n, xs with`
	`      0, xs -> []`
	`    | n, [] -> failwith "take"`
	`    | n, x::xs -> x :: take (n-1) xs`

Does not have guards – use `if`	`let rec fac = function`
	`    0 -> 1`
	`  | n when n>0 -> n * fac (n-1)`
	`  | _ -> raise Hell`

`fun foo(p as (x,y)) = (x,p,y)`	`let foo ((x,y) as p) = (x,p,y)`

Tuples

SML	Ocaml
`type foo = int * float * string`	`type foo = int * float * string`

`val bar = (0, 3.14, "hi")`	`let bar = 0, 3.14, "hi"`
	or
	`let bar = (0, 3.14, "hi")`

`#2 bar`	*Does not have tuple selection – use pattern matching instead, e.g.* `let _,x,_ = bar in x`

`#2`	*Does not have first-class selectors – use function instead, e.g.*
	`function _,x,_ -> x`
	or
	`fun (_,x,_) -> x`

`(inputLine stdIn, inputLine stdIn)`	*Evaluation order is undefined for tuples – use `let`, e.g.*
	`let line1 = read_line () in`
	`let line2 = read_line () in`
	`(line1, line2)`

Records

SML	Ocaml
`type foo = {x:int, y:float, s:string ref}` *Note: record types need not be declared*	`type foo = {x:int; y:float; mutable s:string}` *Note: mutable field does not have the same type as a reference*
`val bar = {x=0, y=3.14, s=ref ""}`	`let bar = {x=0; y=3.14; s=""}`

`#x bar`
`#y bar`
`!(#s bar)`
	`bar.x`
	`bar.y`
	`bar.s`

`#x`	*Does not have first-class selectors – use function instead, e.g.* `fun r -> r.x`

`val {x=x, y=y, s=s} = bar`
`val {y=y, ...} = bar`
or
`val {x, y, s} = bar`
`val {y, ...} = bar`
	`let {x=x; y=y; s=s} = bar`
	`let {y=y} = bar`
	or (since Ocaml 3.12)
	`let {x; y; s} = bar`
	`let {y; _} = bar`

`{x = 1, y = #y bar, s = #s bar}`	`{x = 1; y = bar.y; s = bar.s}`
	or
	`{bar with x = 1}`

`#s bar := "something"`	`bar.s <- "something"`
Does not have polymorphic fields	`type bar = {f:'a.'a -> int}`

`{a = inputLine stdIn, b = inputLine stdIn}`	*Evaluation order is undefined for records – use `let`, e.g.*
	`let line1 = read_line () in`
	`let line2 = read_line () in`
	`{a = line1; b = line2}`

References

SML	Ocaml
`val r = ref 0`	`let r = ref 0`

`!r`	`!r`
	or
	`r.contents`

`r := 1`	`r := 1`
	or
	`r.contents <- 1`

`fun f(ref x) = x`	`let f {contents=x} = x`

`r1 = r2`
`r1 <> r2`
	`r1 == r2`
	`r1 != r2`

Comparisons

SML	Ocaml

`2 = 2`
`2 <> 3`
	`2 = 2`
	`2 <> 3`

`val r = ref 2`
`r = r`
`r <> ref 2`
	`let r = ref 2`
	`r == r`
	`r != ref 2`

`(2, r) = (2, r)`
`(2, r) <> (2, ref 2)`
	Does not have a proper generic equality (on one hand `(2, r) != (2, r)`, on the other `(2, r) = (2, ref 2)`)

`case String.compare(x,y) of`
`      LESS => a`
`    | EQUAL => b`
`    | GREATER => c`
	`match compare x y with`
	`      n when n < 0 -> a`
	`    | 0 -> b`
	`    | _ -> c`

`fun f x y = (x = y)`
`val f : ''a -> ''a -> bool`
	`let f x y = (x = y)`
	`val f : 'a -> 'a -> bool`
	Does not have equality type variables –

	comparison allowed on all types but may raise Invalid_argument *exception*
eqtype t	type t *Does not have equality types – comparison allowed on all types but may raise* Invalid_argument *exception*

Lists

SML	Ocaml
[1, 2, 3]	[1; 2; 3]
[(1, 2), (3, 4)]	[1, 2; 3, 4]
List.length xs	List.length xs
List.map f xs	List.map f xs
List.app f xs	List.iter f xs
List.foldl op+ 0 xs List.foldr op- 100 xs	List.fold_left (+) 0 xs List.fold_right (-) xs 100
List.all (fn x => x=0) xs List.exists (fn x => x>0) xs	List.for_all (fun x -> x=0) xs List.exists (fun x -> x>0) xs
val xys = ListPair.zip (xs, ys)	let xys = List.combine xs ys
val (xs, ys) = ListPair.unzip xys	let (xs, ys) = List.split xys
ListPair.app f (xs, ys)	List.iter2 f xs ys
[inputLine stdIn, inputLine stdIn]	*Evaluation order is undefined for lists – use* let*, e.g.* let line1 = read_line () in let line2 = read_line () in [line1; line2]

Strings

SML	Ocaml
"Hello " ^ "world\n"	"Hello " ^ "world\n"
Int.toString 13 Real.toString 3.141	string_of_int 13 string_of_float 3.141
String.size s	String.length s
String.substring(s, 1, 2)	String.sub s 1 2
String.sub(s, 0)	String.get s 0 *or* s.[0]
Strings are immutable, use CharArray *for mutability*	String.set s 0 'F' *or* s.[0] <- 'F'

Array Functions

SML	Ocaml
Array.array(20, 1.0)	Array.make 20 1.0
Array.fromList xs	Array.from_list xs

`Array.tabulate(30, fn x => x*x)`	`Array.init 30 (fun x -> x*x)`

`Array.sub(a, 2)`	`Array.get a 2`
	or
	`a.(2)`

`Array.update(a, 2, x)`	`Array.set a 2 x`
	or
	`a.(2) <- x`

`Array.copy{src=a, si=10, dst=b, di=0, len=20}`	`Array.blit ~src:a ~src_pos:10 ~dst:b ~dst_pos:0 ~len:20`

Input/Output

SML	Ocaml

`fun copyFile(name1, name2) =`
`    let`
`        val file1 = TextIO.openIn name1`
`        val s      = TextIO.inputAll file1`
`        val _      = TextIO.closeIn file1`
`        val file2 = TextIO.openOut name2`
`    in`
`        TextIO.output(file2, s);`
`        TextIO.closeOut file2`
`    end`
	`let copy_file name1 name2 =`
	`    let file1 = open_in name1 in`
	`    let size = in_channel_length file1 in`
	`    let buf = String.create size in`
	`    really_input file1 buf 0 size;`
	`    close_in file1;`
	`    let file2 = open_out name2 in`
	`    output_string file2 buf;`
	`    close_out file2`
	Caveat: above code actually contains a race condition.

Exceptions

SML	Ocaml
`exception Hell`	`exception Hell`
`exception TotalFailure of string`	`exception Total_failure of string`
`raise TotalFailure "Unknown code"`	`raise (Total_failure "Unknown code")`

`expr handle TotalFailure s =>`
`    ouch()`
	`try expr with`
	`    Total_failure s -> ouch ()`

Local Declarations

SML	Ocaml

`fun pyt(x,y) =`
`    let`
`        val xx = x * x`
`        val yy = y * y`
`    in`
`        Math.sqrt(xx + yy)`
`    end`
	`let pyt x y =`
	`    let xx = x *. x in`
	`    let yy = y *. y in`
	`    sqrt (xx +. yy)`

`local`
`    fun sqr x = x * x`
`in`
`    fun pyt(x,y) = Math.sqrt(sqr x + sqr y)`
`end`
	Does not have `local` *– use global declarations, an auxiliary module, or* `let`

`let`
`    structure X = F(A)`
`in`
`    X.value + 10`
`end`
	`let module X = F (A) in`
	`    X.value + 10`

Standard does not have structure declarations in `let` *but some implementations support them*	*Experimental language extension*
`let open M in expr end`	`let open M in expr` *Note: since Ocaml 3.12*

`let`
`    datatype t = A | B`
`    exception E`
`in`
`    expr`
`end`
	Does not have local type or exception declarations – use global declarations or `let module`

Structures

SML	**Ocaml**

`structure X :> S =`
`struct`
`    type t = int`
`    val x = 0`
`end`
	`module X : S =`
	`struct`
	`    type t = int`
	`    let x = 0`
	`end`

`X :> S`	`(X : S)`
`X : S`	*Does not have transparent signature ascription – use opaque ascription and* `with` *constraints*
`open X`	`include X`

`local open X in`
`    (* ... *)`
`end`
	`open X`
	`(* ... *)`

Functors

SML	**Ocaml**

`functor F(X : S) =`
`struct`
`    (* ... *)`
`end`
	`module F (X : S) =`
	`struct`
	`    (* ... *)`
	`end`
	or
	`module F = functor (X : S) ->`
	`struct`
	`    (* ... *)`
	`end`

`functor F(X : sig type t end) = body`
`structure X = F (struct type t = int end)`
or
`functor F(type t) = body`
`structure X = F(type t = int)`
	`module F (X : sig type t end) = body`
	`module X = F(struct type t = int end)`

`functor F (X : S) (Y : T) = body`
Standard does not have higher-order functors but several implementations support them
	`module F (X : S) (Y : T) = body`
	or
	`module F = functor (X : S) -> functor (Y : T) -> body`

`functor F(X : S) =`
`let`
`    structure Y = G(X)`
`in`
	Does not have `let` *for modules*

`    Y.A`
`end`	

Signatures

SML	Ocaml

`signature S =`
`sig`
`    type t`
`    eqtype u`
`    val x : t`
`    structure M : T`
`end`
	`module type S =`
	`sig`
	`    type t`
	`    type u`
	`    val x : t`
	`    module M : T`
	`end`

`functor F(X : S) : S`
Standard does not have higher-order functors but several implementations support them
	`module F (X : S) : S`
	or
	`module F : functor (X : S) -> S`

`include S`	`include S`
Does not have `open` *in signatures*	`open X`

`structure X : A`
`structure Y : B`
`sharing type X.t = Y.u`
	Does not have `sharing` *constraints – use* `with`

`S where type t = int`	`S with type t = int`
`S where X = A.B` *Standard does not have* `where` *for structures but several implementations support it – use* `where type` *otherwise*	`S with X = A.B`

`signature S =`
`sig`
`    signature A`
`    signature B = A`
`end`
Standard does not have nested signatures but some implementations support them
	`module type S =`
	`sig`
	`    module type A`
	`    module type B = A`
	`end`

Original version by Jens Olsson, jenso@csd.uu.se.
Module stuff, other additions, and HTMLification by Andreas Rossberg, rossberg@mpi-sws.org.
Last modified: 2011/01/18 / Imprint / Data protection