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 Obj=ctive Caml (version 3). It is primarily targetted at people who need to conve=t code between the two dialects. Where suitable we also mention common extens=ons to SML, or recent extensions of Ocaml. The comparison does not cover featur=s that do not have an appropriate counter part in the sibling dialect (eg. Ocaml's object sublanguage, SML's user-defined operator fixity, o= 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 express=ons and definitions. Keywords and other reserved symbols are type-set in blue.

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 & =`

Functions

SML	Ocaml
`fn f => fn x =>fn y =>`	`fun f -> fun x -> fun y ->=f (x,y)` *or* `fun f x y -> f (x,y)`
`fn 0 => =` `  | n => 1`	`function 0 -> 0` `  | = -> 1`
`f o g`	`fun x -> f=(g x)`
`map SOME xs`	*Does not have first-class constructors – use function instead, eg.* `map (fun x -> Some x) xs`
`map #2 triples` `map #lab records`	*Does not have first-class selectors – use function instead, eg.*

	`map (fun (_,x,_) -> x) triples` `map (fun x -> x.lab) records`

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 = =xpr`	`let name = =xpr`
`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 =3D None | Some of 'a`
`datatype t =of int | B of u` **`withtype u =* 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 ==/font> C of real *<=font> real` `fun complex xy = C xy` `fun coord (C xy) ==/font> xy`	`type complex = C of float * float` `let complex (x,y) =3D C (x,y)` `let coord (C (x,y)) =3D (x,y)` *or (note parentheses in type declaration)* `type complex = C of (float * float)` `let complex xy = C xy` `let coord (C xy) ==/font> xy`

Matching

SML	Ocaml
`fun getOpt(NONE, d) =3D d` `  | getOpt(SOME x, _) = x`	`let get_opt = function` `  (None, d) -> d` `  | (Some x, _) -> x =br>`
`fun getOpt (opt, d) =` `  case opt of` `  NONE =>  | SOME x => x`	`let get_opt (opt, d)` `  match opt with` `  None -> =` `  | Some x -> x`
`fun take 0 xs = []` `  | take n nil =3D raise Empty` `  | take n (x::xs) = x :: take (n-1) xs`	`let rec take = 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 whe= 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, eg.* `match bar with _,x,_ -> x` *or* `let _,x,_ = bar in x`
`#2`	*Does not have first-class selectors – use function instead, eg.* `function _,x,_ -> x` *or* `fun (_,x,_) -> x`

Records

SML	Ocaml
`type foo = {x:int, y:f=oat, s:string ref}` *Note: record types do not need to be declared*	`type foo = {x:int; y:f=oat; mutable s:s=ring}` *Note: mutable field has not the same type as a reference*
`val bar = {x=0, y==ref ""}`	`let bar = {x=0; y==""}`
`#x bar` `#y bar` `!(#s bar)`	`bar.x` `bar.y` `bar.s`
`#x`	*Does not have first-class selectors – use function instead, eg.* `fun r -> r.x`
`val {x=x, y=y, s==} = bar` `val {y=y, .=.} = bar` *or* `val {x, y, s} = bar` `val {y, ...} = bar`	`let {x=x; y=y; s==} = bar` `let {y=y} = bar`
`{x = 1, y = #y ba=, s = #s ba=}`	`{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}`

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
`1 = 1` `1 <> 3`	`1 = 1` `1 <> 3`
`val r = ref=1` `r = r` `r <> ref 1`	`let r = ref=1` `r == r` `r != ref 1`
`(1, r) = (1, r)` `(1, r) <> (1, ref 1)`	*Does not have a proper generic equality (on one hand=20* `(1, r) != (1, r)`*, on the other* `(1, r) = (1, ref 1)`*)*
`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 t=pes but may raise* `Invalid_argument` *exception*

List Functions

SML	Ocaml
`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 =&g=; x=0) xs` `List.exists (fn x =3D> x>0) xs`	`List.for_all (fun x => x=0) xs` `List.exists (fun x -=gt; x>0) xs`
`ListPair.app f (xs, ys)`	`List.iter2 f xs ys`

String Functions

SML	Ocaml
`"Hello " ^ "world"`	`"Hello " ^ "world"`
`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] <- &=x27;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`	`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`

`TextIO.output(file2, s);` `TextIO.closeOut file2` `end`	*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"` `=br>`	`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` *Standard does not have structure declarations in `let` but some implementations support them*	`let module X = F (A) in X.value + 10` *Experimental language extension*
`let` `open M` `datatype t = A | B` `<=r> exception E` `in` `expr` `end`	*Does not have local `open`, 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 ascriptio= 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 :->` `struct` `  (* ... *)` `end`
`functor F(X : sig type t end) = body` `module X = F (struct type t = =nt` `end)` *or* `functor F (type t) = body` `structure X = F(type t =`	`module F (X : sig type t end) = body` `module X = F(struct type t = =nt end)`
`functor F (X :: 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 :-> functor (Y :-> body`
`functor F(X :=S) =` `let` `  structure Y = G(X)` `in` `  Y.A` `end`	*Does not have* `let` *for modules*

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`
`include S`	`include S`
Does not have `open` *in signature=*	`open X`
`structure X :=A` `structure Y :=B` `sharing type =.t = Y.u`	*Does not have* `sharing` *constrain=s - use* `with`
`S where type = = 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 = =.B`
`signature S =` `sig` `  signature A` `  signature B = A` `end` *Standard does not have nested signatures but some implementations support t=em*	`module type S =` `sig` `  module type A` `  module type B = =` `end`