Pervasives	
val raise : exn -> 'a	Raise the given exception
val (=) : 'a -> 'a -> bool	tests for structural equality of e1 and e2
val (<>) : 'a -> 'a -> bool	Negation of (=).
val (==) : 'a -> 'a -> bool	e1 == e2 tests for physical equality of e1 and e2.
val (!=) : 'a -> 'a -> bool	Negation of (==).
val (<) : 'a -> 'a -> bool	
val (>) : 'a -> 'a -> bool	
val (<=) : 'a -> 'a -> bool	
val (>=) : 'a -> 'a -> bool	
val compare : 'a -> 'a -> int	compare x y returns 0 if x is equal to y, a negative integer if x is less than y, and a positive integer if x is greater than y.
val min : 'a -> 'a -> 'a	Return the smaller of the two arguments.
val max : 'a -> 'a -> 'a	Return the greater of the two arguments.
val not : bool -> bool	The boolean negation.
val (&&) : bool -> bool -> bool	
val (||) : bool -> bool -> bool	
val (|>) : 'a -> ('a -> 'b) -> 'b	Reverse-application operator: x |> f |> g is exactly equivalent to g (f (x)).
val (~-) : int -> int	Unary negation.
val (mod) : int -> int -> int	Integer remainder.
val abs : int -> int	Return the absolute value of the argument.
val (**) : float -> float -> float	Exponentiation.
val sqrt : float -> float	Square root.
val log : float -> float	Natural logarithm.
val log10 : float -> float	Base 10 logarithm.
val ceil : float -> float	Round above to an integer value.
val floor : float -> float	Round below to an integer value.
val abs_float : float -> float	abs_float f returns the absolute value of f.
val float_of_int : int -> float	Convert an integer to floating-point.
val int_of_float : float -> int	
val (^) : string -> string -> string	String concatenation.
val int_of_char : char -> int	
val char_of_int : int -> char	
val fst : 'a * 'b -> 'a	Return the first component of a pair.

Pervasives (cont)

val snd : 'a * 'b -> 'b	Return the second component of a pair.
val (@) : 'a list -> 'a list -> 'a list	List concatenation.

Types and Type Inference

Function	Type
fun x y –> x + y	int -> int -> int

Include variables that are passed in inside of the type.

List basics

val length : 'a list -> int	Return the length
val nth : 'a list -> int -> 'a	Return the n-th element of the given list.
val rev : 'a list -> 'a list	List reversal.
val concat : 'a list list -> 'a list	Concatenate a list of lists. The elements of the argument are all concatenated together (in the same order) to give the result.

Prefix with `List.`

List Iterators

val map : ('a -> 'b) -> 'a list -> 'b list	
val mapi : (int -> 'a -> 'b) -> 'a list -> 'b list	Same as List.map, but the function is applied to the index of the element as first argument (counting from 0), and the element itself as second argument.
val fold_left : ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a	
val fold_right : ('a -> 'b -> 'b) -> 'a list -> 'b -> 'b	
val map2 : ('a -> 'b -> 'c) -> 'a list -> 'b list -> 'c list	
val fold_left2 : ('a -> 'b -> 'c -> 'a) -> 'a -> 'b list -> 'c list -> 'a	
val fold_right2 : ('a -> 'b -> 'c -> 'c) -> 'a list -> 'b list -> 'c -> 'c	

Prefix with `List.`

List Scanning

val for_all : ('a -> bool) -> 'a list -> bool	checks if all elements of the list satisfy the predicate p
val exists : ('a -> bool) -> 'a list -> bool	checks if at least one element of the list satisfies the predicate p
val for_all2 : ('a -> 'b -> bool) -> 'a list -> 'b list -> bool	
val exists2 : ('a -> 'b -> bool) -> 'a list -> 'b list -> bool	

List Scanning (cont)

val mem : 'a -> 'a list -> bool	mem a l is true if and only if a is equal to an element of l.
val memq : 'a -> 'a list -> bool	Same as List.mem, but uses physical equality

Prefix with `List.`

List Searching

val find : ('a -> bool) -> 'a list -> 'a	returns the first element of the list l that satisfies the predicate p
val filter : ('a -> bool) -> 'a list -> 'a list	returns all the elements of the list l that satisfy the predicate p
val partition : ('a -> bool) -> 'a list -> 'a list * 'a list	returns a pair of lists (l1, l2), where l1 is the list of all the elements of l that satisfy the predicate p, and l2 is the list of all the elements of l that do not satisfy p

Prefix with `List.`

Lists of pairs

val split : ('a*'b) list -> 'a list * 'b list	Transform a list of pairs into a pair of lists
val combine : 'a list -> 'b list -> ('a * 'b) list	Transform a pair of lists into a list of pairs

Prefix with `List.`

List Sorting

val sort : ('a -> 'a -> int) -> 'a list -> 'a list	
val merge : ('a -> 'a -> int) -> 'a list -> 'a list -> 'a list	Merge two lists: Assuming that l1 and l2 are sorted according to the comparison function cmp, merge cmp l1 l2 will return a sorted list containting all the elements of l1 and l2.

Prefix with `List.`

String basics

val length : string -> int	
val get : string -> int -> char	returns the character at index n
val make : int -> char -> string	String.make n c returns a fresh string of length n, filled with the character c.
val sub : string -> int -> int -> string	returns a fresh string of length len, containing the substring of s that starts at position start and has length len.

String basics (cont)

val concat : string -> string list -> string	String.concat sep sl concatenates the list of strings sl, inserting the separator string sep between each.
val map : (char -> char) -> string -> string	String.map f s applies function f in turn to all the characters of s (in increasing index order) and stores the results in a new string that is returned.
val trim : string -> string	
val index : string -> char -> int	returns the index of the first occurrence of character c in string s
val rindex : string -> char -> int	returns the index of the last occurrence of character c in string s
val contains : string -> char -> bool	
val uppercase_ascii : string -> string	all lowercase letters translated to uppercase
val lowercase_ascii : string -> string	all uppercase letters translated to lowercase
val capitalize_ascii : string -> string	first character set to uppercase
val uncapitalize_ascii : string -> string	first character set to lowercase

Prefix with `String.`

Map

```
let rec map ( f : 'a -> 'b) (lst : 'a list) : 'b list =
match lst with
| [] -> []
| hd :: tl -> f hd :: map f tl ;;
```

fold_left

```
let rec foldleft (f : 'a -> 'b -> 'a)
(acc : 'a)
(lst : 'b list)
: 'a =
match lst with
| [] -> acc
| hd :: tl -> foldleft f (f acc hd) tl ;;
```

Different Data Types

list	<type> list	Group together data of the same type
tuple	(<type1> * <type2>)	Group together data of different types
Record	{field one: <type1>; field two: <type2>;...}	Groud together data of different types, but it is easier to access data.
Algebraic Data type	Name_one of <type1> | Name_two of <type2>	One piece of data can have many forms

Module

```
module type Integer =
sig
val value : int
end

module ThreeSig : Integer =
struct
let value = 3
let is_positive : bool = true
end
```

Applying the signature Integer, via ": Integer" restricts ThreeSig makes only "value" public.

Functor

Takes in a module and returns a new module. Ex.

```
module Increment ( I : Integer ) : Integer =
struct
let value = I.value + 1
end
```

Functor can take in multiple modules.

Abstraction Barrier

If a signature has a type

```
module type OPERATORS =
sig
type t
end ;;
```

Then it can be overridden in the implementation using the "with" keyword

```
module IntOps : OPERATORS with type t = int =
struct
type t = int
let times = ( * )
let divide = ( / )
let plus = ( + )
let int_to_t x = x
end ; ;
```