ML Programming Language Cheatsheet

A concise cheat sheet covering the essential syntax, features, and concepts of the ML programming language family (Standard ML, OCaml, F#). Useful for quick reference and learning.

Core Syntax and Types

Basic Syntax

Variable Binding	`val x = 5;` (SML) `let x = 5;;` (OCaml) `let x = 5` (F#)
Function Definition	`fun add x y = x + y;` (SML) `let add x y = x + y;;` (OCaml) `let add x y = x + y` (F#)
Comments	(* SML comment) *(SML)* `(* Nested comments are allowed *)` *(SML)* (OCaml comment) *(OCaml)* `(* Nested comments are allowed *)` *(OCaml)* (F# comment *) (F#) `(* Nested comments are allowed *)` (F#)
Sequential Execution	`;` (SML) `;;` (OCaml) `ignore` (F#)
Unit Type	`()` (SML/OCaml/F#)
String Concatenation	`^` (SML/OCaml) `+` (F#)

Data Types

Integer	`int` (SML/OCaml/F#)
Real/Float	`real` (SML), `float` (OCaml/F#)
Boolean	`bool` (SML/OCaml/F#)
String	`string` (SML/OCaml/F#)
Character	`char` (SML/OCaml/F#)
Unit	`unit` (SML/OCaml/F#)

Operators

Arithmetic	`+, -, *, div, mod` (SML) `+, -, *, /, mod` (OCaml) `+, -, *, /, %` (F#)
Comparison	`=, <>, >, <, >=, <=` (SML/OCaml/F#)
Boolean	`andalso, orelse, not` (SML) `&&, ||, not` (OCaml) `&&, ||, not` (F#)
Floating-point Arithmetic	`+, -, *, /` (OCaml/F#) `Real.+ ,Real.- ,Real.* ,Real./` (SML)
Integer division	`div` (SML) `/` (OCaml/F#)
Modulus	`mod` (SML/OCaml) `%` (F#)

Control Flow and Data Structures

Conditional Statements

If-Then-Else	`if condition then expr1 else expr2` (SML/OCaml/F#)

Case/Match Statements	`case expression of`
	`  pattern1 => result1`
	`| pattern2 => result2`
	`| _ => default_result;`
	
	`match expression with`
	`  pattern1 -> result1`
	`| pattern2 -> result2`
	`| _ -> default_result`
	
	`match expression with`
	`  | pattern1 -> result1`
	`  | pattern2 -> result2`
	`  | _ -> default_result`

Boolean Conditionals	`if true then 1 else 0;`
	`(* returns 1 *)`
	
	`if true then 1 else 0;;`
	`(* returns 1 *)`
	
	`if true then 1 else 0`
	`// returns 1`

String Conditionals	`if "a" = "a" then 1 else 0; (* returns 1 *)`
	
	`if "a" = "a" then 1 else 0;; (* returns 1 *)`
	
	`if "a" = "a" then 1 else 0  // returns 1`

Lists

List Creation	`[1, 2, 3]` (SML/OCaml) `[ 1; 2; 3 ]` (F#)
Cons Operator	`::` (SML/OCaml) `::` (F#)
Head and Tail	`hd list` (SML), `List.hd list` (OCaml), `List.head list` (F#) `tl list` (SML), `List.tl list` (OCaml), `List.tail list` (F#)
List Length	`length list` (SML), `List.length list` (OCaml), `List.length list` (F#)
Appending Lists	`@` (SML/OCaml) `@` (F#)
Map Function	`map f list` (SML), `List.map f list` (OCaml), `List.map f list` (F#)

Tuples

Tuple Creation	`(1, "hello", true)` (SML/OCaml/F#)
Accessing Elements	`#1 tuple` (SML/OCaml), `fst tuple`, `snd tuple` (OCaml for 2-tuples) `fst tuple`, `snd tuple` (F# for 2-tuples)
Deconstruction	`let (x, y, z) = tuple` (SML/OCaml/F#)

Example Tuple	`val example = (1, "text", 3.14);`
	`val (int_val, string_val, real_val) = example;`
	
	`let example = (1, "text", 3.14);;`
	`let (int_val, string_val, real_val) = example;;`
	
	`let example = (1, "text", 3.14)`
	`let (int_val, string_val, real_val) = example`

Functions and Modules

Function Definitions

Basic Function	`fun square x = x * x;` (SML) `let square x = x * x;;` (OCaml) `let square x = x * x` (F#)
Recursive Function	`fun factorial 0 = 1 | factorial n = n * factorial (n - 1);` (SML) `let rec factorial n = if n = 0 then 1 else n * factorial (n - 1);;` (OCaml) `let rec factorial n = if n = 0 then 1 else n * factorial (n - 1)` (F#)
Anonymous Function (Lambda)	`fn x => x * x` (SML) `fun x -> x * x` (OCaml) `fun x -> x * x` (F#)
Curried Function	`fun add x y = x + y;` (SML) `let add x y = x + y;;` (OCaml) `let add x y = x + y` (F#)
Higher-Order Function	`fun apply f x = f x;` (SML) `let apply f x = f x;;` (OCaml) `let apply f x = f x` (F#)

Modules

Module Definition	`structure MyModule = struct ... end;` (SML) `module MyModule = struct ... end;;` (OCaml) `module MyModule = struct ... end` (F#)

Module Signature (Interface)	`signature MY_MODULE_SIG = sig ... end;` (SML)
	`module type MY_MODULE_TYPE = sig ... end;;` (OCaml)
	`// F# uses signature files (.fsi)`
	`// or inline signatures`
	`type MY_MODULE_TYPE = sig ... end`

Module Implementation	`structure MyModule : MY_MODULE_SIG = struct ... end;` (SML) `module MyModule : MY_MODULE_TYPE = struct ... end;;` (OCaml)
Accessing Module Members	`MyModule.member` (SML/OCaml/F#)
Opening a Module	`open MyModule;` (SML/OCaml) `open MyModule` (F#)

Advanced Features

Exceptions

Exception Declaration	`exception MyException of string;` (SML) `exception MyException of string;;` (OCaml) `exception MyException of string` (F#)
Raising an Exception	`raise MyException "Error message";` (SML) `raise (MyException "Error message");;` (OCaml) `raise (MyException "Error message")` (F#)

Exception Handling	`try`
	`  expression`
	`with`
	`  MyException msg =>`
	`handle_error msg;`
	
	`try`
	`  expression`
	`with`
	`  MyException msg ->`
	`handle_error msg;;`
	
	`try`
	`  expression`
	`with`
	`  | MyException msg ->`
	`handle_error msg`

Standard Exceptions	`Fail`, `InvalidArg`, `Match` (SML/OCaml/F#)

References (Mutable State)

Creating a Reference	`ref value` (SML/OCaml/F#)
Accessing a Reference	`!ref_variable` (SML/OCaml/F#)
Updating a Reference	`ref_variable := new_value` (SML/OCaml/F#)

Example Usage	`val counter = ref 0;`
	`counter := !counter + 1;`
	`!counter; (* Returns 1 *)`
	
	`let counter = ref 0;;`
	`counter := !counter + 1;;`
	`!counter;; (* Returns 1 *)`
	
	`let counter = ref 0`
	`counter := !counter + 1`
	`!counter  // Returns 1`

Records (Structs)

Record Definition	`type person = {name : string, age : int};`
	`val john : person = {name = "John", age = 30};`
	
	`type person = {name : string; age : int};;`
	`let john : person = {name = "John"; age = 30};;`
	
	`type person = {name : string; age : int}`
	`let john : person = {name = "John"; age = 30}`

Accessing Record Fields	`#name john` (SML), `john.name` (OCaml/F#)

Record Update (Functional)	`val jane = {john with age = 31};`
	
	`let jane = {john with age = 31};;`
	
	`let jane = {john with age = 31}`