WIKIPEDIA

ML (programming language)

ML (**Meta Language**) is a general-purpose functional programming language. It is known for its use of the polymorphic Hindley–Milner type system, which automatically assigns the types of most expressions without requiring explicit type annotations, and ensures type safety – there is a formal proof that a well-typed ML program does not cause runtime type errors.[1] ML provides pattern matching for function arguments, garbage collection, imperative programming, call-by-value and currying. It is used heavily in programming language research and is one of the few languages to be completely specified and verified using formal semantics. Its types and pattern matching make it well-suited and commonly used to operate on other formal languages, such as in compiler writing, automated theorem proving, and formal verification.

ML

Paradigm	Multi-paradigm: functional, imperative
Designed by	Robin Milner and others at the University of Edinburgh
First appeared	1973
Typing discipline	Inferred, static, strong
Dialects	
OCaml, Standard ML, F#	
Influenced by	
ISWIM	
Influenced	
Clojure, Coq, Cyclone, C++, Elm, F#, F*, Haskell, Idris, Kotlin, Miranda, Nemerle, OCaml, Opa, Erlang, Rust, Scala, Standard ML	

Contents

Overview

Features of ML include a call-by-value evaluation strategy, first-class functions, automatic memory management through garbage collection, parametric polymorphism, static typing, type inference, algebraic data types, pattern matching, and exception handling. ML uses static scoping rules.

ML can be referred to as an *impure* functional language, because although it encourages functional programming, it does allow side-effects (like languages such as Lisp, but unlike a purely functional language such as Haskell). Like most programming languages, ML uses eager evaluation, meaning that all subexpressions are always evaluated, though lazy evaluation can be achieved through the use of closures. Thus one can create and use infinite streams as in Haskell, but their expression is indirect.

ML's strengths are mostly applied in language design and manipulation (compilers, analyzers, theorem provers), but it is a general-purpose language also used in bioinformatics and financial systems.

ML was developed by Robin Milner and others in the early 1970s at the University of Edinburgh,[2] and its syntax is inspired by ISWIM. Historically, ML was conceived to develop proof tactics in the LCF theorem prover (whose language, *pplambda*, a combination of the first-order predicate calculus and the simply-typed polymorphic lambda calculus, had ML as its metalanguage).

Today there are several languages in the ML family; the three most prominent are Standard ML (SML), OCaml and F#. Ideas from ML have influenced numerous other languages, like Haskell, Cyclone, Nemerle,[3] ATS, and Elm.[4]

Examples

The following examples use the syntax of Standard ML. Other ML dialects such as OCaml and F# differ in small ways.

Factorial

The factorial function expressed as pure ML:

```
fun fac (0 : int) : int = 1
  | fac (n : int) : int = n * fac (n - 1)
```

This describes the factorial as a recursive function, with a single terminating base case. It is similar to the descriptions of factorials found in mathematics textbooks. Much of ML code is similar to mathematics in facility and syntax.

Part of the definition shown is optional, and describes the *types* of this function. The notation E : t can be read as *expression E has type t*. For instance, the argument n is assigned type *integer* (int), and fac (n : int), the result of applying fac to the integer n, also has type integer. The function fac as a whole then has type *function from integer to integer* (int -> int), that is, fac accepts an integer as an argument and returns an integer result. Thanks to type inference, the type annotations can be omitted and will be derived by the compiler. Rewritten without the type annotations, the example looks like:

```
fun fac 0 = 1
  | fac n = n * fac (n - 1)
```

The function also relies on pattern matching, an important part of ML programming. Note that parameters of a function are not necessarily in parentheses but separated by spaces. When the function's argument is 0 (zero) it will return the integer 1 (one). For all other cases the second line is tried. This is the recursion, and executes the function again until the base case is reached.

This implementation of the factorial function is not guaranteed to terminate, since a negative argument causes an infinite descending chain of recursive calls. A more robust implementation would check for a nonnegative argument before recursing, as follows:

```
fun fact n = let
  fun fac 0 = 1
    | fac n = n * fac (n - 1)
  in
    if (n < 0) then raise Domain else fac n
  end
```

The problematic case (when n is negative) demonstrates a use of ML's exception system.

The function can be improved further by writing its inner loop as a tail call, such that the call stack need not grow in proportion to the number of function calls. This is achieved by adding an extra, *accumulator*, parameter to the inner function. At last, we arrive at

```
fun fact n = let
  fun fac 0 acc = acc
    | fac n acc = fac (n - 1) (n * acc)
  in
    if (n < 0) then raise Domain else fac n 1
  end
```

List reverse

The following function *reverses* the elements in a list. More precisely, it returns a new list whose elements are in reverse order compared to the given list.

```
fun reverse [] = []
  | reverse (x :: xs) = (reverse xs) @ [x]
```

This implementation of reverse, while correct and clear, is inefficient, requiring quadratic time for execution. The function can be rewritten to execute in linear time:

```
fun 'a reverse xs : 'a list = List.foldl (op ::) [] xs
```

Notably, this function is an example of parametric polymorphism. That is, it can consume lists whose elements have any type, and return lists of the same type.

Modules

Modules are ML's system for structuring large projects and libraries. A module consists of a signature file and one or more structure files. The signature file specifies the API to be implemented (like a C header file, or Java interface file). The structure implements the signature (like a C source file or Java class file). For example, the following define an Arithmetic signature and an implementation of it using Rational numbers:

```
signature ARITH =
sig
        type t
        val zero : t
        val succ : t -> t
        val sum : t * t -> t
end
```

```
structure Rational : ARITH =
struct
        datatype t = Rat of int * int
        val zero = Rat (0, 1)
        fun succ (Rat (a, b)) = Rat (a + b, b)
        fun sum (Rat (a, b), Rat (c, d)) = Rat (a * d + c * b , b * d)
end
```

These are imported into the interpreter by the 'use' command. Interaction with the implementation is only allowed via the signature functions, for example it is not possible to create a 'Rat' data object directly via this code. The 'structure' block hides all the implementation detail from outside.

ML's standard libraries are implemented as modules in this way.

See also

- Standard ML and Standard ML § Implementations
- Dependent ML: a dependently typed extension of ML
    - ATS: a further development of dependent ML
- Lazy ML: an experimental lazily evaluated ML dialect from the early 1980s
- PAL (programming language): an educational language related to ML
- OCaml: an "industrial strength"[5] ML dialect used to implement Coq
- F#: an open-source cross-platform functional-first language for the .NET Framework
- CoffeeScript and TypeScript: metalanguages for ECMAScript

References

1. Robin Milner. A theory of type polymorphism in programming. Journal of Computer and System Sciences, 17(3):348–375, 1978.
2. Gordon, Michael J. C. (1996). "From LCF to HOL: a short history" (http://www.cl.cam.ac.uk/~mjcg/papers/HolHistory.html). Retrieved 2007-10-11.
3. *Programming language for "special forces" of developers* (http://nemerle.org/About), Russian Software Development Network: Nemerle Project Team, retrieved January 24, 2021
4. Tate, Bruce A.; Daoud, Fred; Dees, Ian; Moffitt, Jack (2014). "3. Elm". *Seven More Languages in Seven Weeks* (Book version: P1.0-November 2014 ed.). The Pragmatic Programmers, LLC. pp. 97, 101. ISBN 978-1-941222-15-7. "On page 101, Elm creator Evan Czaplicki says: 'I tend to say "Elm is an ML-family language" to get at the shared heritage of all these languages.' ["these languages" is referring to Haskell, OCaml, SML, and F#.]"
5. "OCaml is an industrial strength programming language supporting functional, imperative and object-oriented styles" (http://ocaml.org/). Retrieved on January 2, 2018.

Further reading

- *The Definition of Standard ML*, Robin Milner, Mads Tofte, Robert Harper, MIT Press 1990; (revised edition adds author David MacQueen), MIT Press 1997, ISBN 0-262-63181-4.
- *Commentary on Standard ML*, Robin Milner, Mads Tofte, MIT Press 1997, ISBN 0-262-63137-7.
- *ML for the Working Programmer*, Lawrence Paulson, Cambridge University Press 1991, 1996, ISBN 0-521-57050-6.
- Harper, Robert (2011). *Programming in Standard ML* (https://www.cs.cmu.edu/~rwh/isml/book.pdf) (PDF). Carnegie Mellon University.
- *Elements of ML Programming*, Jeffrey D. Ullman, Prentice-Hall 1994, 1998, ISBN 0-13-790387-1.

External links

- Standard ML of New Jersey, another popular implementation (http://smlnj.org)
- F#, an ML implementation using the Microsoft .NET framework (http://msdn.microsoft.com/en-us/fsharp/default.aspx) Archived (https://web.archive.org/web/20100218004857/http://msdn.microsoft.com/en-us/fsharp/default.aspx) 2010-02-18 at the Wayback Machine
- MLton, a whole-program optimizing Standard ML compiler (http://mlton.org)
- Successor ML (https://github.com/SMLFamily/Successor-ML) – or sML
- CakeML, a read-eval-print loop version of ML with formally verified runtime and translation to assembler (https://cakeml.org)