WIKIPEDIA

Caml

Caml (originally an acronym for **Categorical Abstract Machine Language**) is a multi-paradigm, general-purpose programming language which is a dialect of the ML programming language family. Caml was developed in France at INRIA and ENS.

Caml is statically typed, strictly evaluated, and uses automatic memory management. OCaml, the main descendant of Caml, adds many features to the language, including an object layer.

Caml

Caml

Paradigm	Multi-paradigm: functional, imperative
Family	ML
Designed by	Gérard Huet, Guy Cousineau, Ascánder Suárez, Pierre Weis, Michel Mauny (Heavy Caml), Xavier Leroy (Caml Light)
First appeared	1985
Stable release	0.75[1] / January 26, 2002
Typing discipline	Inferred, static, strong
OS	Cross-platform
Website	caml.inria.fr (http://caml.inria.fr)
Influenced by	
ML	
Influenced	
OCaml	

Contents

Examples

In the following, # represents the OCaml prompt.

Hello World

```
print_endline "Hello, world!";;
```

Factorial function (recursion and purely functional programming)

Many mathematical functions, such as factorial, are most naturally represented in a purely functional form. The following recursive, purely functional Caml function implements factorial:

```
let rec fact n = if n=0 then 1 else n * fact(n - 1);;
```

The function can be written equivalently using pattern matching:

```
let rec fact = function
  | 0 -> 1
  | n -> n * fact(n - 1);;
```

This latter form is the mathematical definition of factorial as a recurrence relation.

Note that the compiler inferred the type of this function to be `int -> int`, meaning that this function maps ints onto ints. For example, 12! is:

```
 # fact 12;;
 - : int = 479001600
```

Numerical derivative (higher-order functions)

Since Caml is a functional programming language, it is easy to create and pass around functions in Caml programs. This capability has an enormous number of applications. Calculating the numerical derivative of a function is one such application. The following Caml function `d` computes the numerical derivative of a given function `f` at a given point `x`:

```
let d delta f x =
  (f (x +. delta) -. f (x -. delta)) /. (2. *. delta);;
```

This function requires a small value `delta`. A good choice for delta is the cube root of the machine epsilon.

The type of the function `d` indicates that it maps a `float` onto another function with the type `(float -> float) -> float -> float`. This allows us to partially apply arguments. This functional style is known as currying. In this case, it is useful to partially apply the first argument `delta` to `d`, to obtain a more specialised function:

```
# let d = d (sqrt epsilon_float);;
val d : (float -> float) -> float -> float = <fun>
```

Note that the inferred type indicates that the replacement `d` is expecting a function with the type `float -> float` as its first argument. We can compute a numerical approximation to the derivative of $x^3 - x - 1$ at $x = 3$ with:

```
# d (fun x -> x *. x *. x -. x -. 1.) 3.;;
- : float = 26.
```

The correct answer is $f'(x) = 3x^2 - 1 \rightarrow f'(3) = 27 - 1 = 26$.

The function `d` is called a "higher-order function" because it accepts another function (`f`) as an argument. We can go further and create the (approximate) derivative of f, by applying `d` while omitting the `x` argument:

```
# let f' = d (fun x -> x *. x *. x -. x -. 1.) ;;
val f' : float -> float = <fun>
```

The concepts of curried and higher-order functions are clearly useful in mathematical programs. In fact, these concepts are equally applicable to most other forms of programming and can be used to factor code much more aggressively, resulting in shorter programs and fewer bugs.

Discrete wavelet transform (pattern matching)

The 1D Haar wavelet transform of an integer-power-of-two-length list of numbers can be implemented very succinctly in Caml and is an excellent example of the use of pattern matching over lists, taking pairs of elements (`h1` and `h2`) off the front and storing their sums and differences on the lists `s` and `d`, respectively:

```
# let haar l =
   let rec aux l s d =
     match l, s, d with
       [s], [], d -> s :: d
     | [], s, d -> aux s [] d
     | h1 :: h2 :: t, s, d -> aux t (h1 + h2 :: s) (h1 - h2 :: d)
     | _ -> invalid_arg "haar"
     in aux l [] [];;
val haar : int list -> int list = <fun>
```

For example:

```
  # haar [1; 2; 3; 4; -4; -3; -2; -1];;
   - : int list = [0; 20; 4; 4; -1; -1; -1; -1]
```

Pattern matching allows complicated transformations to be represented clearly and succinctly. Moreover, the OCaml compiler turns pattern matches into very efficient code, at times resulting in programs that are shorter and faster than equivalent code written with a case statement (Cardelli 1984, p. 210.).

History

The first Caml implementation was written in Lisp by Ascánder Suárez in 1987 at the French Institute for Research in Computer Science and Automation (INRIA).[2]

Its successor, *Caml Light*, was implemented in C by Xavier Leroy and Damien Doligez,[2] and the original was nicknamed "Heavy Caml" because of its higher memory and CPU requirements.[2]

Caml Special Light was a further complete rewrite that added a powerful module system to the core language. It was augmented with an object layer to become *Objective Caml*, eventually renamed OCaml.

See also

- Categorical abstract machine
- OCaml

References

1. "Latest Caml Light release" (http://caml.inria.fr/caml-light/release.en.html). Retrieved 22 February 2020.
2. "A History of Caml" (https://caml.inria.fr/about/history.en.html), inria.fr

Bibliography

- The Functional Approach to Programming with Caml (http://pauillac.inria.fr/cousineau-mauny/main.html) by Guy Cousineau and Michel Mauny.
- Cardelli, Luca (1984). Compiling a functional language (https://dx.doi.org/10.1145/800055.802037) *ACM Symposium on LISP and functional programming*, Association of Computer Machinery.

External links

- Official website (http://caml.inria.fr) – Caml language family