

# Functional programming

From Wikipedia, the free encyclopedia.

**Functional programming** is a programming paradigm that treats computation as the evaluation of mathematical functions.

In contrast to procedural / imperative programming, functional programming emphasizes the evaluation of functional expressions, rather than execution of commands. The expressions in these languages are formed by using functions to combine basic values.

## Contents

- 1 Introduction
- 2 History
- 3 Comparison with imperative programming
- 4 Functional programming languages
- 5 Higher-order functions
- 6 Speed and space considerations
- 7 Example languages
- 8 See also
- 9 References
- 10 External links

## Introduction

Mathematical functions have great strengths in terms of flexibility and analysis. For example, if a function is known to be idempotent, then a call to a function which has itself as its argument, and which is known to have no side-effects, may be efficiently computed without multiple calls.

A function in this sense has zero or more parameters and a single return value. The parameters—or arguments, as they are sometimes called—are the inputs to the function, and the return value is the function's output. The definition of a function describes how the function is to be evaluated in terms of other functions. For example, the function  $f(x) = x^2 + 2$  is defined in terms of the power and addition functions. At some point, the language has to provide basic functions that require no further definition.

Functions can be manipulated in a variety of ways in a functional programming language. Functions are treated as first-class values, which is to say that functions can be parameters or inputs to other functions and can be the return values or outputs of a function. This allows functions like `mapcar` in LISP and `map` in Haskell that take both a function and a list as input and apply the input function to every element of the list. Functions can be named, as in other languages, or defined anonymously (sometimes during program execution) using a lambda abstraction and used as values in other functions.

Functional languages also allow functions to be "curried". Currying is a technique for rewriting a function with multiple parameters as a function with one parameter that maps to another function of one parameter and so on until all parameters are exhausted. The curried function can be applied to just a subset of its parameters. The result is a function where the parameters in this subset are now fixed as constants, and the values of the rest of the parameters are still unspecified. This new function can be applied to the remaining parameters to get the final function value. For example, a function  $\text{add}(x, y) = x + y$  can be curried so that the return value of  $\text{add}(2)$ —notice that there is no  $y$  parameter—will be an anonymous function that is equivalent to a function  $\text{add2}(y) = 2 + y$ . This new function has only one parameter and corresponds to adding 2 to a number. Again, this is possible only because functions are treated as first-class values.

## History

Lambda calculus could be considered the first functional programming language, though it was not designed to be executed on a computer. Lambda calculus is a model of computation designed by Alonzo Church in the 1930s that provides a very formal way to describe function evaluation. The first computer-based functional programming language was Information Processing Language (IPL), developed by Newell, Shaw, and Simon at RAND Corporation for the JOHNNIAC computer in the mid 1950s. A much-improved functional programming language was LISP, developed by John McCarthy while at the Massachusetts Institute of Technology for the IBM 700/7000 series scientific computers in the late 1950s. While not a purely functional programming language, LISP did introduce most of the features now found in modern functional programming languages. Scheme was a later attempt to simplify and improve LISP. In the 1970s the language ML was created at the University of Edinburgh, and David Turner developed the language Miranda at the University of Kent. The language Haskell was released in the late 1980s in an attempt to gather together many ideas in functional programming research.

## Comparison with imperative programming

Functional programming can be contrasted with imperative programming. Functional programming appears to be missing several constructs often (though incorrectly) considered essential to an imperative language, such as C or Pascal. For example, in strict functional programming, there is no explicit memory allocation and no explicit variable assignment. However, these operations occur automatically when a function is invoked; memory allocation occurs to make space for the parameters and the return value, and assignment occurs to copy the parameters into this newly allocated space and to copy the return value back into the calling function. Both operations can only occur on function entry and exit, so side effects of function evaluation are eliminated. By disallowing side effects in functions, the language provides referential transparency. This ensures that the result of a function will be the same for a given set of parameters no matter where, or when, it is evaluated. Referential transparency greatly eases both the task of proving program correctness and the task of automatically identifying independent computations for parallel execution.

Looping, another imperative programming construct, is accomplished through the more general functional construct of recursion. Recursive functions invoke themselves, allowing an operation to be performed over and over. In fact, it can be proven that looping is equivalent to a special type of recursion called tail recursion. Recursion in functional programming can take many forms and is in general a more powerful technique than looping. For this reason, almost all imperative languages also support it (with FORTRAN 77 and COBOL, before 2002, as notable exceptions).

## Functional programming languages

"Pure" functional programs require no variables and have no side-effects, and are therefore automatically thread-safe. They are also automatically verifiable provided that any recursive cycle eventually stops. Nested functions just pass their results back to the main function. Functional languages commonly make quite sophisticated use of the stack.

Functional programming often depends heavily on recursion. The Scheme programming language even requires certain types of recursion (tail recursion) to be recognized and automatically optimized by a compiler.

Furthermore, functional programming languages are likely to enforce referential transparency, which is the familiar notion that 'equals can be substituted for equals': if two expressions are defined to have equal values, then one can be substituted for the other in any larger expression without affecting the result of the computation. For example, in

```
z = f(sqrt(2), sqrt(2));
```

we can factor out `sqrt(2)` and write

```
s = sqrt(2);
z = f(s, s);
```

thus eliminating the extra evaluation of the square-root function.

As intuitive as it sounds, this is not always the case with imperative languages. A case in point is the C programming language's `getchar()` "function", which is strictly a function not of its arguments but of the contents of the input stream `stdin` and how much has already been read. Following the example above:

```
z = f(getchar(), getchar());
```

we *cannot* eliminate `getchar()` as we did for `sqrt(2)`, because in C, "`getchar()`" might return two different values the two times it is called.

Hidden side-effects are in general the rule, rather than the exception, of traditional programming languages. Whenever a procedure reads a value from or writes a value to a global or shared variable, the potential exists for hidden side effects. This leakage of information across procedure boundaries in ways that are not explicitly represented by function calls and definitions greatly increases the hidden complexity of programs written in conventional non-functional languages.

By removing these hidden information leaks, functional programming languages offer the possibility of much cleaner programs which are easier to design and debug. However, they also offer other benefits.

Many programmers accustomed to the imperative paradigm find it difficult to learn functional programming, which encompasses a whole different way of composing programs. This difficulty, along with the fact that functional programming environments do not have the extensive tools and libraries available for traditional programming languages, are among the main reasons that functional programming has received little use in the software industry. Functional languages have remained largely the domain of academics and hobbyists, and what little inroads have been made are due to impure functional languages such as Erlang and Common Lisp. It could be argued that the largest influence of functional programming on the software industry has been by those academically trained programmers who have gone on to apply the impure functional programming style to their work in traditional imperative languages.

## Higher-order functions

A powerful mechanism sometimes used in functional programming is the notion of higher-order functions. Functions are higher-order when they can take other functions as arguments, and/or return functions as results. (The derivative in calculus is a common example of a function that maps a function to a function).

Higher-order functions were studied in the lambda calculus theory long before the notion of functional programming existed, and have influenced the design of several functional programming languages, such as Haskell, and have even spawned the function-level programming paradigm, which includes languages such as Backus' FP.

## Speed and space considerations

Functional languages have long been criticised as resource-hungry, both in terms of CPU resources and memory. This was mainly due to two things:

- some early functional languages were implemented with no effort at efficiency
- non-functional languages achieved speed at least in part by leaving out features such as bounds checking or garbage collection which are viewed as essential parts of modern computing frameworks, the overhead of which was built-in to functional languages by default

As modern imperative languages and their implementations have started to include greater emphasis on correctness, rather than raw speed, and the implementations of functional languages have begun to emphasise speed as well as correctness, the performance of functional languages and imperative languages has begun to converge. For programs which spend most of their time doing numerical computations, some functional languages (such as OCaml and Clean) can approach C speed, while for programs that handle large matrices and multidimensional databases, array functional languages (such as J and K) are usually faster than most non-optimized C programs. However, purely functional languages can be considerably slower when manipulating large data structures, due to less efficient memory usage. Lazy evaluation also adds an extra overhead in languages such as Haskell.

## Example languages

The oldest example of a functional language is Lisp, though neither the original LISP nor modern Lisps are pure-functional. The modern canonical examples are Haskell and members of the ML family including SML and Ocaml. Others include Scheme, Erlang, Clean, and Miranda. Tcl can also be well used as a functional programming language, with higher-order functions, abstractions and such, though not everybody does.

Category:Functional languages provides an exhaustive list.

## See also

- Procedural programming (contrast)
- Imperative programming (contrast)
- Programming paradigms
- lazy evaluation
- eager evaluation
- list of functional programming topics

## References

- Cousineau, Guy and Michel Mauny. *The Functional Approach to Programming*. Cambridge, UK: Cambridge University Press, 1998.
- Graham, Paul. *ANSI Common LISP*. Englewood Cliffs, New Jersey: Prentice Hall, 1996.
- Hudak, Paul. "Conception, Evolution, and Application of Functional Programming Languages." *ACM Computing Surveys* **21**, no. 3 (1989): 359-411.
- Pratt, Terrence, W. and Marvin V. Zelkowitz. *Programming Languages: Design and Implementation*. 3rd ed. Englewood Cliffs, New Jersey: Prentice Hall, 1996.
- Salus, Peter H. *Functional and Logic Programming Languages*. Vol. 4 of Handbook of Programming Languages. Indianapolis, Indiana: Macmillan Technical Publishing, 1998.
- Thompson, Simon. *Haskell: The Craft of Functional Programming*. Harlow, England: Addison-Wesley Longman Limited, 1996.

## External links

- Why Functional Programming Matters (<http://www.math.chalmers.se/~rjmh/Papers/whyfp.html>)
- "Functional Programming" (<ftp://ftp.aw.com/cseng/authors/finkel/apld/finkel04.pdf>)-- Chapter 4 of *Advanced Programming Language Design* by Raphael Finkel, an introductory explanation of functional programming

---

This article includes parts of an earlier version (<http://www.nupedia.com/article/short/Functional+Programming/>) (stable link (<http://www.nupedia.com/article/677/>)) posted on 19 June 2001 on Nupedia; reviewed and approved by the Computers group; editor, Michael Witbrock ; lead reviewer, Nancy Tinkham; lead copyeditors, Ruth Ifcher. and Larry Sanger.

Retrieved from "[http://en.wikipedia.org/wiki/Functional\\_programming](http://en.wikipedia.org/wiki/Functional_programming)"