Functional programming

From Wikipedia, the free encyclopedia

In computer science, **functional programming** is a programming paradigm—a style of building the structure and elements of computer programs—that treats computation as the evaluation of mathematical functions and avoids changing-state and mutable data. It is a declarative programming paradigm, which means programming is done with expressions[1] or declarations[2] instead of statements. In functional code, the output value of a function depends only on the arguments that are passed to the function, so calling a function *f* twice with the same value for an argument *x* will produce the same result *f(x)* each time; this is in contrast to procedures depending on local or global state, which may produce different results at different times when called with the same arguments but different program state. Eliminating side effects, i.e. changes in state that do not depend on the function inputs, can make it much easier to understand and predict the behavior of a program, which is one of the key motivations for the development of functional programming.

Functional programming has its origins in lambda calculus, a formal system developed in the 1930s to investigate computability, the Entscheidungsproblem, function definition, function application, and recursion. Many functional programming languages can be viewed as elaborations on the lambda calculus. Another well-known declarative programming paradigm, *logic programming*, is based on relations.[3]

In contrast, imperative programming changes state with commands in the source code, the simplest example being assignment. Imperative programming does have functions—not in the mathematical sense—but in the sense of subroutines. They can have side effects that may change the value of program state. Functions without return values therefore make sense. Because of this, they lack referential transparency, i.e. the same language expression can result in different values at different times depending on the state of the executing program.[3]

Functional programming languages have largely been emphasized in academia rather than in commercial software development. However, prominent programming languages which support functional programming such as Common Lisp, Scheme,[4][5][6][7] Clojure,[8][9] Wolfram Language[10] (also known as Mathematica), Racket,[11] Erlang,[12][13][14] OCaml,[15][16] Haskell,[17][18] and F#[19][20] have been used in industrial and commercial applications by a wide variety of organizations. Functional programming is also supported in some domain-specific programming languages like R (statistics),[21] J, K and Q from Kx Systems (financial analysis), XQuery/XSLT (XML),[22][23] and Opal.[24] Widespread domain-specific declarative languages like SQL and Lex/Yacc use some elements of functional programming, especially in eschewing mutable values.[25]

Programming in a functional style can also be accomplished in languages that are not specifically designed for functional programming. For example, the imperative Perl programming language has been the subject of a book describing how to apply functional programming concepts.[26] This is also true of the PHP programming language.[27] C++11, Java 8, and C# 3.0 all added constructs to facilitate the functional style. The Julia language also offers functional programming abilities. An interesting case is that of Scala[28] – it is frequently written in a functional style, but the presence of side effects and mutable state place it in a grey area between imperative and functional languages.

Contents

History

Lambda calculus provides a theoretical framework for describing functions and their evaluation. Although it is a mathematical abstraction rather than a programming language, it forms the basis of almost all functional programming languages today. An equivalent theoretical formulation, combinatory logic, is commonly perceived as more abstract than lambda calculus and preceded it in invention. Combinatory logic and lambda calculus were both originally developed to achieve a clearer approach to the foundations of mathematics.[29]

An early functional-flavored language was Lisp, developed in the late 1950s for the IBM 700/7000 series scientific computers by John McCarthy while at Massachusetts Institute of Technology (MIT).[30] Lisp first introduced many paradigmatic features of functional programming, though early Lisps were multi-paradigm languages, and incorporated support for numerous programming styles as new paradigms evolved. Later dialects, such as Scheme and Clojure, and offshoots such as Dylan and Julia, sought to simplify and rationalise Lisp around a cleanly functional core, while Common Lisp was designed to preserve and update the paradigmatic features of the numerous older dialects it replaced.[31]

Information Processing Language (IPL) is sometimes cited as the first computer-based functional programming language.[32] It is an assembly-style language for manipulating lists of symbols. It does have a notion of "generator", which amounts to a function accepting a function as an argument, and, since it is an assembly-level language, code can be used as data, so IPL can be regarded as having higher-order functions. However, it relies heavily on mutating list structure and similar imperative features.

Kenneth E. Iverson developed APL in the early 1960s, described in his 1962 book *A Programming Language* (ISBN 9780471430148). APL was the primary influence on John Backus's FP. In the early 1990s, Iverson and Roger Hui created J. In the mid-1990s, Arthur Whitney, who had previously worked with Iverson, created K, which is used commercially in financial industries along with its descendant Q.

John Backus presented FP in his 1977 Turing Award lecture "Can Programming Be Liberated From the von Neumann Style? A Functional Style and its Algebra of Programs".[33] He defines functional programs as being built up in a hierarchical way by means of "combining forms" that allow an "algebra of programs"; in modern language, this means that functional programs follow the principle of compositionality. Backus's paper popularized research into functional programming, though it emphasized function-level programming rather than the lambda-calculus style which has come to be associated with functional programming.

In the 1970s, ML was created by Robin Milner at the University of Edinburgh, and David Turner initially developed the language SASL at the University of St Andrews and later the language Miranda at the University of Kent. Also in Edinburgh in the 1970s, Burstall and Darlington developed the functional language NPL.[34] NPL was based on Kleene Recursion Equations and was first introduced in their work on program transformation.[35] Burstall, MacQueen and Sannella then incorporated the polymorphic type checking from ML to produce the language Hope.[36] ML eventually developed into several dialects, the most common of which are now OCaml and Standard ML. Meanwhile, the development of Scheme, a simple lexically scoped and (impurely) functional dialect of Lisp, as described in the influential Lambda Papers and the classic 1985 textbook *Structure and Interpretation of Computer Programs*, brought awareness of the power of functional programming to the wider programming-languages community.

In the 1980s, Per Martin-Löf developed intuitionistic type theory (also called *constructive* type theory), which associated functional programs with constructive proofs of arbitrarily complex mathematical propositions expressed as dependent types. This led to powerful new approaches to interactive theorem proving and has influenced the development of many subsequent functional programming languages.

The Haskell language began with a consensus in 1987 to form an open standard for functional programming research; implementation releases have been ongoing since 1990.

Concepts

A number of concepts and paradigms are specific to functional programming, and generally foreign to imperative programming (including object-oriented programming). However, programming languages are often hybrids of several programming paradigms, so programmers using "mostly imperative" languages may have utilized some of these concepts.[37]

First-class and higher-order functions

Higher-order functions are functions that can either take other functions as arguments or return them as results. In calculus, an example of a higher-order function is the differential operator d/dx, which returns the derivative of a function f.

Higher-order functions are closely related to first-class functions in that higher-order functions and first-class functions both allow functions as arguments and results of other functions. The distinction between the two is subtle: "higher-order" describes a mathematical concept of functions that operate on other functions, while "first-class" is a computer science term that describes programming language entities that have no restriction on their use (thus first-class functions can appear anywhere in the program that other first-class entities like numbers can, including as arguments to other functions and as their return values).

Higher-order functions enable partial application or currying, a technique in which a function is applied to its arguments one at a time, with each application returning a new function that accepts the next argument. This allows one to succinctly express, for example, the successor function as the addition operator partially applied to the natural number one.

Pure functions

Pure functions (or expressions) have no side effects (memory or I/O). This means that pure functions have several useful properties, many of which can be used to optimize the code:

- If the result of a pure expression is not used, it can be removed without affecting other expressions.
- If a pure function is called with arguments that cause no side-effects, the result is constant with respect to that argument list (sometimes called referential transparency), i.e. if the pure function is again called with the same arguments, the same result will be returned (this can enable caching optimizations such as memoization).
- If there is no data dependency between two pure expressions, then their order can be reversed, or they can be performed in parallel and they cannot interfere with one another (in other terms, the evaluation of any pure expression is thread-safe).
- If the entire language does not allow side-effects, then any evaluation strategy can be used; this gives the compiler freedom to reorder or combine the evaluation of expressions in a program (for example, using deforestation).

While most compilers for imperative programming languages detect pure functions and perform common-subexpression elimination for pure function calls, they cannot always do this for pre-compiled libraries, which generally do not expose this information, thus preventing optimizations that involve those external functions. Some compilers, such as gcc, add extra keywords for a programmer to explicitly mark external functions as pure, to enable such optimizations. Fortran 95 also allows functions to be designated "pure".

Recursion

Iteration (looping) in functional languages is usually accomplished via recursion. Recursive functions invoke themselves, allowing an operation to be performed over and over until the base case is reached. Though some recursion requires maintaining a stack, tail recursion can be recognized and optimized by a compiler into the same code used to implement iteration in imperative languages. The Scheme language standard requires implementations to recognize and optimize tail recursion. Tail recursion optimization can be implemented by transforming the program into continuation passing style during compiling, among other approaches.

Common patterns of recursion can be factored out using higher order functions, with catamorphisms and anamorphisms (or "folds" and "unfolds") being the most obvious examples. Such higher order functions play a role analogous to built-in control structures such as loops in imperative languages.

Most general purpose functional programming languages allow unrestricted recursion and are Turing complete, which makes the halting problem undecidable, can cause unsoundness of equational reasoning, and generally requires the introduction of inconsistency into the logic expressed by the language's type system. Some special purpose languages such as Coq allow only well-founded recursion and are strongly normalizing (nonterminating computations can be expressed only with infinite streams of values called codata). As a consequence, these languages fail to be Turing complete and expressing certain functions in them is impossible, but they can still express a wide class of interesting computations while avoiding the problems introduced by unrestricted recursion. Functional programming limited to well-founded recursion with a few other constraints is called total functional programming.[38]

Strict versus non-strict evaluation

Functional languages can be categorized by whether they use *strict (eager)* or *non-strict (lazy)* evaluation, concepts that refer to how function arguments are processed when an expression is being evaluated. The technical difference is in the denotational semantics of expressions containing failing or divergent computations. Under strict evaluation, the evaluation of any term containing a failing subterm will itself fail. For example, the expression:

```
print length([2+1, 3*2, 1/0, 5-4])
```

will fail under strict evaluation because of the division by zero in the third element of the list. Under lazy evaluation, the length function will return the value 4 (i.e., the number of items in the list), since evaluating it will not attempt to evaluate the terms making up the list. In brief, strict evaluation always fully evaluates function arguments before invoking the function. Lazy evaluation does not evaluate function arguments unless their values are required to evaluate the function call itself.

The usual implementation strategy for lazy evaluation in functional languages is graph reduction.[39] Lazy evaluation is used by default in several pure functional languages, including Miranda, Clean, and Haskell.

Hughes 1984 argues for lazy evaluation as a mechanism for improving program modularity through separation of concerns, by easing independent implementation of producers and consumers of data streams.[40] Launchbury 1993 describes some difficulties that lazy evaluation introduces, particularly in analyzing a program's storage requirements, and proposes an operational semantics to aid in such analysis.[41] Harper 2009 proposes including both strict and lazy evaluation in the same language, using the language's type system to distinguish them.[42]

Type systems

Especially since the development of Hindley–Milner type inference in the 1970s, functional programming languages have tended to use typed lambda calculus, rejecting all invalid programs at compilation time and risking false positive errors, as opposed to the untyped lambda calculus, that accepts all valid programs at compilation time and risks false negative errors, used in Lisp and its variants (such as Scheme), although they reject all invalid programs at runtime, when the information is enough to not reject valid programs. The use of algebraic datatypes makes manipulation of complex data structures convenient; the presence of strong compile-time type checking makes programs more reliable in absence of other reliability techniques like test-driven development, while type inference frees the programmer from the need to manually declare types to the compiler in most cases.

Some research-oriented functional languages such as Coq, Agda, Cayenne, and Epigram are based on intuitionistic type theory, which allows types to depend on terms. Such types are called dependent types. These type systems do not have decidable type inference and are difficult to understand and program with. But dependent types can express arbitrary propositions in predicate logic. Through the Curry–Howard isomorphism, then, well-typed programs in these languages become a means of writing formal mathematical proofs from which a compiler can generate certified code. While these languages are mainly of interest in academic research (including in formalized mathematics), they have begun to be used in engineering as well. Compcert is a compiler for a subset of the C programming language that is written in Coq and formally verified.[43]

A limited form of dependent types called generalized algebraic data types (GADT's) can be implemented in a way that provides some of the benefits of dependently typed programming while avoiding most of its inconvenience.[44] GADT's are available in the Glasgow Haskell Compiler, in OCaml (since version 4.00) and in Scala (as "case classes"), and have been proposed as additions to other languages including Java and C#.[45]

Referential transparency

Functional programs do not have assignment statements, that is, the value of a variable in a functional program never changes once defined. This eliminates any chances of side effects because any variable can be replaced with its actual value at any point of execution. So, functional programs are referentially transparent.[46]

Consider C assignment statement `x = x * 10`, this changes the value assigned to the variable `x`. Let us say that the initial value of `x` was `1`, then two consecutive evaluations of the variable `x` will yield `10` and `100` respectively. Clearly, replacing `x = x * 10` with either `10` or `100` gives a program with different meaning, and

so the expression *is not* referentially transparent. In fact, assignment statements are never referentially transparent.

Now, consider another function such as `int plusone(int x) {return x+1;}` *is* transparent, as it will not implicitly change the input `x` and thus has no such side effects. Functional programs exclusively use this type of function and are therefore referentially transparent.

Functional programming in non-functional languages

It is possible to use a functional style of programming in languages that are not traditionally considered functional languages.[47] For example, both D and Fortran 95 explicitly support pure functions.[48]

JavaScript, Lua[49] and Python had first class functions from their inception.[50] Amrit Prem added support to Python for "lambda", "map", "reduce", and "filter" in 1994, as well as closures in Python 2.2,[51] though Python 3 relegated "reduce" to the `functools` standard library module.[52] First-class functions have been introduced into other mainstream languages such as PHP 5.3, Visual Basic 9, C# 3.0, and C++11.

In Java, anonymous classes can sometimes be used to simulate closures;[53] however, anonymous classes are not always proper replacements to closures because they have more limited capabilities.[54] Java 8 supports lambda expressions as a replacement for some anonymous classes.[55] However, the presence of checked exceptions in Java can make functional programming inconvenient, because it can be necessary to catch checked exceptions and then rethrow them—a problem that does not occur in other JVM languages that do not have checked exceptions, such as Scala.

In C#, anonymous classes are not necessary, because closures and lambdas are fully supported. Libraries and language extensions for immutable data structures are being developed to aid programming in the functional style in C#.

Many object-oriented design patterns are expressible in functional programming terms: for example, the strategy pattern simply dictates use of a higher-order function, and the visitor pattern roughly corresponds to a catamorphism, or fold.

Similarly, the idea of immutable data from functional programming is often included in imperative programming languages,[56] for example the tuple in Python, which is an immutable array.

Data structures

Purely functional data structures are often represented in a different way than their imperative counterparts.[57] For example, array with constant-time access and update is a basic component of most imperative languages and many imperative data-structure, such as hash table and binary heap, are based on arrays. Arrays can be replaced by map or random access list, which admits purely functional implementation, but the access and update time is logarithmic. Therefore, purely functional data structures can be used in languages which are non-functional, but they may not be the most efficient tool available, especially if persistency is not required.

Comparison to imperative programming

Functional programming is very different from imperative programming. The most significant differences stem from the fact that functional programming avoids side effects, which are used in imperative programming to implement state and I/O. Pure functional programming completely prevents side-effects and provides referential transparency.

Higher-order functions are rarely used in older imperative programming. A traditional imperative program might use a loop to traverse and modify a list. A functional program, on the other hand, would probably use a higher-order “map” function that takes a function and a list, generating and returning a new list by applying the function to each list item.

Simulating state

There are tasks (for example, maintaining a bank account balance) that often seem most naturally implemented with state. Pure functional programming performs these tasks, and I/O tasks such as accepting user input and printing to the screen, in a different way.

The pure functional programming language Haskell implements them using monads, derived from category theory. Monads offer a way to abstract certain types of computational patterns, including (but not limited to) modeling of computations with mutable state (and other side effects such as I/O) in an imperative manner without losing purity. While existing monads may be easy to apply in a program, given appropriate templates and examples, many students find them difficult to understand conceptually, e.g., when asked to define new monads (which is sometimes needed for certain types of libraries).[58]

Another way in which functional languages can simulate state is by passing around a data structure that represents the current state as a parameter to function calls. On each function call, a copy of this data structure is created with whatever differences are the result of the function. This is referred to as 'state-passing style'.

Impure functional languages usually include a more direct method of managing mutable state. Clojure, for example, uses managed references that can be updated by applying pure functions to the current state. This kind of approach enables mutability while still promoting the use of pure functions as the preferred way to express computations.

Alternative methods such as Hoare logic and uniqueness have been developed to track side effects in programs. Some modern research languages use effect systems to make the presence of side effects explicit.

Efficiency issues

Functional programming languages are typically less efficient in their use of CPU and memory than imperative languages such as C and Pascal.[59] This is related to the fact that some mutable data structures like arrays have a very straightforward implementation using present hardware (which is a highly evolved Turing machine). Flat arrays may be accessed very efficiently with deeply pipelined CPUs, prefetched efficiently through caches (with no complex pointer chasing), or handled with SIMD instructions. It is also not easy to create their equally efficient general-purpose immutable counterparts. For purely functional languages, the worst-case slowdown is logarithmic in the number of memory cells used, because mutable memory can be represented by a purely functional data structure with logarithmic access time (such as a balanced tree).[60] However, such slowdowns are not universal. For programs that perform intensive numerical computations, functional languages such as OCaml and Clean are only slightly slower than C according to The Computer Language Benchmarks Game.[61] For programs that handle large matrices and multidimensional databases, array functional languages (such as J and K) were designed with speed optimizations.

Immutability of data can in many cases lead to execution efficiency by allowing the compiler to make assumptions that are unsafe in an imperative language, thus increasing opportunities for inline expansion.[62]

Lazy evaluation may also speed up the program, even asymptotically, whereas it may slow it down at most by a constant factor (however, it may introduce memory leaks if used improperly). Launchbury 1993[41] discusses theoretical issues related to memory leaks from lazy evaluation, and O'Sullivan *et al.* 2008[63] give some practical advice for analyzing and fixing them. However, the most general implementations of lazy evaluation making extensive use of dereferenced code and data perform poorly on modern processors with deep pipelines and multi-level caches (where a cache miss may cost hundreds of cycles).

Coding styles

Imperative programs have the environment and a sequence of steps manipulating the environment. Functional programs have an expression that is successively substituted until it reaches normal form. An example illustrates this with different solutions to the same programming goal (calculating Fibonacci numbers).

Python

Printing first 10 Fibonacci numbers, iterative

```
def fibonacci(n, first=0, second=1):
    while n != 0:
        print(first, end="\n") # side-effect
        n, first, second = n - 1, second, first + second # assignment
fibonacci(10)
```

Printing first 10 Fibonacci numbers, functional expression style

```
fibonacci = (lambda n, first=0, second=1:
    "" if n == 0 else
    str(first) + "\n" + fibonacci(n - 1, second, first + second))
print(fibonacci(10), end="")
```

Printing a list with first 10 Fibonacci numbers, with generators

```
def fibonacci(n, first=0, second=1):
    while n != 0:
        yield first
        n, first, second = n - 1, second, first + second # assignment
print(list(fibonacci(10)))
```

Printing a list with first 10 Fibonacci numbers, functional expression style

```
fibonacci = (lambda n, first=0, second=1:
    [] if n == 0 else
    [first] + fibonacci(n - 1, second, first + second))
print(fibonacci(10))
```

Haskell

Printing first 10 Fibonacci numbers, functional expression style[1]

```
fibonacci_aux = \n first second->
    if n == 0 then "" else
    show first ++ "\n" ++ fibonacci_aux (n - 1) second (first + second)
fibonacci = \n-> fibonacci_aux n 0 1
main = putStr (fibonacci 10)
```

Printing a list with first 10 Fibonacci numbers, functional expression style[1]

```
fibonacci_aux = \n first second->
    if n == 0 then [] else
    [first] ++ fibonacci_aux (n - 1) second (first + second)
fibonacci = \n-> fibonacci_aux n 0 1
main = putStrLn (show (fibonacci 10))
```

Printing the 11th Fibonacci number, functional expression style[1]

```
fibonacci = \n-> if n == 0 then 0
                 else if n == 1 then 1
                      else fibonacci(n - 1) + fibonacci(n - 2)
main = putStrLn (show (fibonacci 10))
```

Printing the 11th Fibonacci number, functional expression style,[1] tail recursive

```
fibonacci_aux = \n first second->
    if n == 0 then first else
    fibonacci_aux (n - 1) second (first + second)
fibonacci = \n-> Fibonacci_aux n 0 1
main = putStrLn (show (fibonacci 10))
```

Printing the 11th Fibonacci number, functional expression style[1] with recursive lists

```
fibonacci_aux = \first second-> first : fibonacci_aux second (first + second)
select = \n zs-> if n==0 then head zs
                 else select (n - 1) (tail zs)
fibonacci = \n-> select n (fibonacci_aux 0 1)
main = putStrLn (show (fibonacci 10))
```

Printing the 11th Fibonacci number, functional expression style[1] with primitives for recursive lists

```
fibonacci_aux = \first second-> first : fibonacci_aux second (first + second)
fibonacci = \n-> (fibonacci_aux 0 1) !! n
main = putStrLn (show (fibonacci 10))
```

Printing the 11th Fibonacci number, functional expression style[1] with primitives for recursive lists, more concisely

```
fibonacci_aux = 0:1:zipWith (+) fibonacci_aux (tail fibonacci_aux)
fibonacci = \n-> fibonacci_aux !! n
main = putStrLn (show (fibonacci 10))
```

Printing the 11th Fibonacci number, functional declaration style,[2] tail recursive

```
fibonacci_aux 0 first _ = first
fibonacci_aux n first second = fibonacci_aux (n - 1) second (first + second)
fibonacci n = fibonacci_aux n 0 1
main = putStrLn (show (fibonacci 10))
```

Printing the 11th Fibonacci number, functional declaration style, using lazy infinite lists and primitives

```
fibs = 1 : 1 : zipWith (+) fibs (tail fibs)
-- an infinite list of the fibonacci numbers
-- fibs is defined in terms of fibs
fibonacci = (fibs !!)
main = putStrLn $ show $ fibonacci 11
```

Perl 6

As influenced by Haskell and others, Perl 6 has several functional and declarative approaches to problems. For example, you can declaratively build up a well-typed recursive version (the type constraints are optional) through signature pattern matching:

```
# define constraints that are common to all candidates
proto fib ( UInt:D \n --> UInt:D ) {*}
```

```
multi fib ( 0 --> 0 ) { }
multi fib ( 1 --> 1 ) { }

multi fib ( \n ) {
    fib(n - 1) + fib(n - 2)
}

for ^10 -> $n { say fib($n) }
```

An alternative to this is to construct a lazy iterative sequence, which appears as an almost direct illustration of the sequence:

```
my @fib = 0, 1, *+* ... *; # Each additional entry is the sum of the previous two
                           # and this sequence extends lazily indefinitely
say @fib[^10];             # Display the first 10 entries
```

Erlang

Erlang is a functional, concurrent, general-purpose programming language. A Fibonacci algorithm implemented in Erlang (Note: This is only for demonstrating the Erlang syntax. Use other algorithms for fast performance[64]):

```
-module(fib).    % This is the file 'fib.erl', the module and the filename must match
-export([fib/1]). % This exports the function 'fib' of arity 1

fib(1) -> 1; % If 1, then return 1, otherwise (note the semicolon ; meaning 'else')
fib(2) -> 1; % If 2, then return 1, otherwise
fib(N) -> fib(N - 2) + fib(N - 1).
```

Elixir

Elixir is a functional, concurrent, general-purpose programming language that runs on the Erlang virtual machine (BEAM).

The Fibonacci function can be written in Elixir as follows:

```
defmodule Fibonacci do
  def fib(0), do: 0
  def fib(1), do: 1
  def fib(n), do: fib(n-1) + fib(n-2)
end
```

Lisp

The Fibonacci function can be written in Common Lisp as follows:

```
(defun fib (n &optional (a 0) (b 1))
  (if (= n 0)
      a
      (fib (- n 1) b (+ a b))))
```

The program can then be called as

```
(fib 10)
```

Clojure

The Fibonacci function can be written in Clojure as follows:

```
(defn fib
  [n]
  (loop [a 0 b 1 i n]
    (if (zero? i)
      a
      (recur b (+ a b) (dec i)))))
```

The program can then be called as

```
(fib 7)
```

Kotlin

The Fibonacci function can be written in Kotlin as follows:

```
fun fib(x: Int): Int = if (x == 0 || x == 1) x else fib(x - 1) + fib(x - 2)
```

The program can then be called as

```
fib(7)
```

D

D has support for functional programming:

```
import std.stdio;
import std.range;

void main()
{
    /* 'f' is a range representing the first 10 Fibonacci numbers */
    auto f = recurrence!((seq, i) => seq[0] + seq[1])(0, 1)
             .take(10);

    writeln(f);
}
```

R

R is an environment for statistical computing and graphics. It is also a functional programming language.

The Fibonacci function can be written in R as a recursive function as follows:

```
fib <- function(n) {
 if (n <= 2) 1
 else fib(n - 1) + fib(n - 2)
}
```

Or it can be written as a singly recursive function:

```
fib <- function(n,a=1,b=1) {
 if (n == 1) a
 else fib(n-1,b,a+b)
}
```

Or it can be written as an iterative function:

```
fib <- function(n) {
 if (n == 1) 1
 else if (n == 2) 1
 else {
  fibval<-c(1,1)
  for (i in 3:n) fibval<-c(0,fibval[1])+fibval[2]
  fibval[2]
 }
}
```

The function can then be called as

```
fib(10)
```

SequenceL

SequenceL is a functional, concurrent, general-purpose programming language. The Fibonacci function can be written in SequenceL as follows:

```
fib(n) := n when n < 2 else
          fib(n - 1) + fib(n - 2);
```

The function can then be called as

```
fib(10)
```

To reduce the memory consumed by the call stack when computing a large Fibonacci term, a tail-recursive version can be used. A tail-recursive function is implemented by the SequenceL compiler as a memory-efficient looping structure:

```
fib(n) := fib_Helper(0, 1, n);

fib_Helper(prev, next, n) :=
    prev when n < 1 else
    next when n = 1 else
    fib_Helper(next, next + prev, n - 1);
```

Tcl

The Fibonacci function can be written in Tcl as a recursive function as follows:

```
proc fibo {x} {
    expr {$x<2? $x: [fibo [incr x -1]] + [fibo [incr x -1]]}
}
```

Use in industry

Functional programming has long been popular in academia, but with few industrial applications.[65]:page 11 However, recently several prominent functional programming languages have been used in commercial or industrial systems. For example, the Erlang programming language, which was developed by the Swedish company Ericsson in the late 1980s, was originally used to implement fault-tolerant telecommunications systems.[13] It has since become popular for building a range of applications at companies such as T-Mobile, Nortel, Facebook, Électricité de France and WhatsApp.[12][14][66][67][68] The Scheme dialect of Lisp was used as the basis for several applications on early Apple Macintosh computers,[4][5] and has more recently been applied to problems such as training simulation software[6] and telescope control.[7] OCaml, which was

introduced in the mid-1990s, has seen commercial use in areas such as financial analysis,[15] driver verification, industrial robot programming, and static analysis of embedded software.[16] Haskell, although initially intended as a research language,[18] has also been applied by a range of companies, in areas such as aerospace systems, hardware design, and web programming.[17][18]

Other functional programming languages that have seen use in industry include Scala,[69] F#,[19][20] (both being functional-OO hybrids with support for both purely functional and imperative programming) Wolfram Language,[10] Lisp,[70] Standard ML[71][72] and Clojure.[73]

In education

Functional programming is being used as a method to teach problem solving, algebra and geometric concepts.[74] It has also been used as a tool to teach classical mechanics in Structure and Interpretation of Classical Mechanics.

See also

- Purely functional programming
- Comparison of programming paradigms
- Eager evaluation
- List of functional programming topics
- Nested function
- Inductive functional programming
- Functional reactive programming

References

1. "Declaration vs. expression style - HaskellWiki" (https://wiki.haskell.org/Declaration_vs._expression_style#Expression style).
2. "Declaration vs. expression style - HaskellWiki" (https://wiki.haskell.org/Declaration_vs._expression_style#Declaration_style).
3. Hudak, Paul (September 1989). "Conception, evolution, and application of functional programming languages" (http://www.dbnet.ece.ntua.gr/~adamo/languages/books/p359-hudak.pdf) (PDF). *ACM Computing Surveys*. **21** (3): 359–411. doi:10.1145/72551.72554 (https://doi.org/10.1145%2F72551.72554)
4. Clinger, Will (1987). "MultiTasking and MacScheme" (http://www.mactech.com/articles/mactech/Vol.03/03.12/Multitasking/index.html). *MacTech*. **3** (12). Retrieved 2008-08-28.
5. Hartheimer, Anne (1987). "Programming a Text Editor in MacScheme+Toolsmith" (http://www.mactech.com/articles/mactech/Vol.03/03.1/SchemeWindows/index.html). *MacTech*. **3** (1). Retrieved 2008-08-28.
6. Kidd, Eric. *Terrorism Response Training in Scheme* (http://cufp.galois.com/2007/abstracts.html#EricKidd). CUFP 2007. Retrieved 2009-08-26.
7. Cleis, Richard. *Scheme in Space* (http://cufp.galois.com/2006/abstracts.html#RichardCleis). CUFP 2006. Retrieved 2009-08-26.
8. "The useR! 2006 conference schedule includes papers on the commercial use of R" (http://www.r-project.org/useR-2006/program.html). R-project.org. 2006-06-08. Retrieved 2011-06-20.
9. Chambers, John M. (1998). *Programming with Data: A Guide to the S Language*. Springer Verlag. pp. 67–70. ISBN 978-0-387-98503-9.
10. "Wolfram Language Guide: Functional Programming" (http://reference.wolfram.com/language/guide/FunctionalProgramming.html). 2015. Retrieved 2015-08-24.
11. "State-Based Scripting in Uncharted 2" (https://web.archive.org/web/20121215014637/http://www.gameenginebook.com/gdc09-statescripting-uncharted2.pdf) (PDF). Archived from the original (http://www.gameenginebook.com/gdc09-statescripting-uncharted2.pdf) (PDF) on 2012-12-15. Retrieved 2011-08-08.
12. "Who uses Erlang for product development?" (http://www.erlang.org/faq/faq.html#AEN50) *Frequently asked questions about Erlang*. Retrieved 2007-08-05.

13. Armstrong, Joe (June 2007). *A history of Erlang* (http://doi.acm.org/10.1145/1238844.1238850). Third ACM SIGPLAN Conference on History of Programming Languages. San Diego, California. Retrieved 2009-08-29.
14. Larson, Jim (March 2009). "Erlang for concurrent programming". *Communications of the ACM* **52** (3): 48. doi:10.1145/1467247.1467263 (https://doi.org/10.1145%2F1467247.1467263).
15. Minsky, Yaron; Weeks, Stephen (July 2008). "Caml Trading — experiences with functional programming on Wall Street" (http://journals.cambridge.org/action/displayAbstract?aid=1899164). *Journal of Functional Programming*. Cambridge University Press. **18** (4): 553–564. doi:10.1017/S095679680800676X (https://doi.org/10.1017%2FS095679680800676X). Retrieved 2008-08-27.
16. Leroy, Xavier. *Some uses of Caml in Industry* (http://cufp.galois.com/2007/slides/XavierLeroy.pdf) (PDF). CUFP 2007. Retrieved 2009-08-26.
17. "Haskell in industry" (http://www.haskell.org/haskellwiki/Haskell_in_industry). *Haskell Wiki*. Retrieved 2009-08-26. "Haskell has a diverse range of use commercially, from aerospace and defense, to finance, to web startups, hardware design firms and lawnmower manufacturers".
18. Hudak, Paul; Hughes, J.; Jones, S. P.; Wadler, P. (June 2007). *A history of Haskell: being lazy with class* (http://dl.acm.org/citation.cfm?doid=1238844.1238856). Third ACM SIGPLAN Conference on History of Programming Languages. San Diego, California. doi:10.1145/1238844.1238856 (https://doi.org/10.1145%2F1238844.1238856). Retrieved 2013-09-26.
19. Mansell, Howard (2008). *Quantitative Finance in F#* (http://cufp.galois.com/2008/abstracts.html#MansellHoward) CUFP 2008. Retrieved 2009-08-29.
20. Peake, Alex (2009). *The First Substantial Line of Business Application in F#* (http://cufp.galois.com/2009/abstracts.html#AlexPeakeAdamGranicz) CUFP 2009. Retrieved 2009-08-29.
21. Department of Applied Math, University of Colorado. "Functional vs. Procedural Programming Language" (https://web.archive.org/web/20071113175801/http://amath.colorado.edu/computing/mmm/funcproc.html). Archived from the original (http://amath.colorado.edu/computing/mmm/funcproc.html) on 2007-11-13. Retrieved 2006-08-28.
22. Dimitre Novatchev. "The Functional Programming Language XSLT — A proof through examples" (http://www.topxml.com/xsl/articles/fp/). *TopXML*. Retrieved May 27, 2006.
23. David Mertz. "XML Programming Paradigms (part four): Functional Programming approached to XML processing" (http://gnosis.cx/publish/programming/xml_models_fp.html). *IBM developerWorks*. Retrieved May 27, 2006.
24. OPtimized Applicative Language
25. Donald D. Chamberlin and Raymond F. Boyce (1974). "SEQUEL: A structured English query language". *Proceedings of the 1974 ACM SIGFIDET*: 249–264.
26. Dominus, Mark J. (2005). *Higher-Order Perl*. Morgan Kaufmann. ISBN 1-55860-701-3.
27. Holywell, Simon (2014). *Functional Programming in PHP*. php[architect]. ISBN 9781940111056.
28. "Effective Scala" (http://twitter.github.com/effectivescala/?sd). *Scala Wiki*. Retrieved 2012-02-21. "Effective Scala."
29. Haskell Brooks Curry; Robert Feys (1958). *Combinatory Logic* (https://books.google.com/books?id=fEnuAAAAMAAJ). North-Holland Publishing Company. Retrieved 10 February 2013.
30. McCarthy, John (June 1978). "History of Lisp" (http://citeseer.ist.psu.edu/mccarthy78history.html). *In ACM/SIGPLAN History of Programming Languages Conference*: 217–223. doi:10.1145/800025.808387 (https://doi.org/10.1145%2F800025.808387).
31. Guy L. Steele; Richard P. Gabriel (February 1996). "The Evolution of Lisp" (http://dreamsongs.com/Files/HOPL2-Uncut.pdf) (PDF). *In ACM/SIGPLAN Second History of Programming Languages*: 233–330. doi:10.1145/234286.1057818 (https://doi.org/10.1145%2F234286.1057818).
32. The memoir of Herbert A. Simon (1991), *Models of My Life* pp. 189-190 ISBN 0-465-04640-1 claims that he, Al Newell, and Cliff Shaw are "commonly adjudged to be the parents of [the] artificial intelligence [field]", for writing Logic Theorist, a program which proved theorems from *Principia Mathematica* automatically. In order to accomplish this, they had to invent a language and a paradigm which, which viewed retrospectively, embeds functional programming.
33. "Can Programming Be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs" (http://worrydream.com/refs/Backus-CanProgrammingBeLiberated.pdf) (PDF).
34. R.M. Burstall. Design considerations for a functional programming language. Invited paper. Proc. Infotech State of the Art Conf. "The Software Revolution", Copenhagen, 45–57 (1977)
35. R.M. Burstall and J. Darlington. A transformation system for developing recursive programs. Journal of the Association for Computing Machinery 24(1):44–67 (1977)
36. R.M. Burstall, D.B. MacQueen and D.T. Sannella. HOPE: an experimental applicative language. Proc. 1980 LISP Conference, Stanford, 136–143 (1980).
37. Dick Pountain. "Functional Programming Comes of Age" (https://web.archive.org/web/20060827094123/http://byte.com/art/9408/sec11/art1.htm). *BYTE.com (August 1994)* Archived from the original (http://byte.com/art/9408/sec11/art1.htm) on 2006-08-27. Retrieved August 31, 2006.
38. Turner, D.A. (2004-07-28). "Total Functional Programming" (http://www.jucs.org/jucs_10_7/total_functional_programming). *Journal of Universal Computer Science* **10** (7): 751–768. doi:10.3217/jucs-010-07-0751 (https://doi.org/10.3217%2Fjucs-010-07-0751)

39. The Implementation of Functional Programming Languages (http://research.microsoft.com/~simonpj/papers/slpj-book-987/index.htm). Simon Peyton Jones, published by Prentice Hall, 1987
40. Hughes, John (1984). "Why Functional Programming Matters" (http://www.cse.chalmers.se/~rjmh/Papers/whyfp.html)
41. John Launchbury (1993). "A Natural Semantics for Lazy Evaluation" (http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.35.2016).
42. Robert W. Harper (2009). *Practical Foundations for Programming Languages* (http://www.cs.cmu.edu/~rwh/plbook/book.pdf) (PDF).
43. "The Compcert verified compiler" (http://compcert.inria.fr/doc/index.html)
44. Simon Peyton Jones; Dimitrios Vytiniotis; Stephanie Weirich; Geoffrey Washburn. "Simple unification-based type inference for GADTs" (http://research.microsoft.com/en-us/um/people/simonpj/papers/gadt/). *ICFP 2006*. pp. 50–61.
45. Andrew Kennedy; Claudio Russo (October 2005). "Generalized Algebraic Data Types and Object-Oriented Programming" (https://web.archive.org/web/20061229164852/http://research.microsoft.com/~akenn/generics/gadtoop.pdf) (PDF). *OOPSLA*. San Diego, California. Archived from the original (http://research.microsoft.com/~akenn/generics/gadtoop.pdf) (PDF) on 2006-12-29. source of citation (http://lambda-the-ultimate.org/node/1134)
46. Huges, John. "Why Functional Programming Matters" (http://www.cse.chalmers.se/~rjmh/Papers/whyfp.pdf) (PDF). *http://www.chalmers.se/cse*. Chalmers Tekniska Hˆogskola. External link in `|website=` (help)
47. Hartel, Pieter; Henk Muller; Hugh Glaser (March 2004). "The Functional C experience" (http://www.ub.utwente.nl/webdocs/ctit/1/00000084.pdf) (PDF). *Journal of Functional Programming*. **14** (2): 129–135. doi:10.1017/S0956796803004817 (https://doi.org/10.1017%2FS0956796803004817); David Mertz. "Functional programming in Python, Part 3" (http://wayback.archive.org/web/20071016124848/http://www-128.ibm.com/developerworks/linux/library/l-prog3.html). *IBM developerWorks*. Archived from the original (http://www-128.ibm.com/developerworks/linux/library/l-prog3.html) on 2007-10-16. Retrieved 2006-09-17. (Part 1 (https://web.archive.org/web/20071016124848/http://www-128.ibm.com/developerworks/linux/library/l-prog.html), Part 2 (https://web.archive.org/web/20071016124848/http://www-128.ibm.com/developerworks/linux/library/l-prog2.html))
48. "Functions — D Programming Language 2.0" (http://www.digitalmars.com/d/2.0/function.html#pure-functions). Digital Mars. Retrieved 2011-06-20.
49. "Lua Unofficial FAQ (uFAQ)" (http://www.luafaq.org/#T1.2).
50. "Brendan Eich" (https://brendaneich.com/2008/04/popularity/).
51. van Rossum, Guido (2009-04-21). "Origins of Python's "Functional" Features" (http://python-history.blogspot.de/2009/04/origins-of-pythons-functional-features.html). The History of Python (http://python-history.blogspot.de/). Retrieved 2012-09-27. External link in `|publisher=` (help)
52. "functools — Higher order functions and operations on callable objects" (https://docs.python.org/dev/library/functools.html#functools.reduce) Python Software Foundation. 2011-07-31. Retrieved 2011-07-31.
53. Skarsaune, Martin (2008). *The SICS Java Port Project Automatic Translation of a Large Object Oriented System from Smalltalk to Java*.
54. Gosling, James. "Closures" (http://blogs.oracle.com/jag/entry/closures). *James Gosling: on the Java Road*. Oracle. Retrieved 11 May 2013.
55. "Java SE 8 Lambda Quick Start" (https://blogs.oracle.com/javatraining/entry/java_se_8_lambda_quick)
56. Bloch, Joshua. *Effective Java* (Second ed.). pp. Item 15.
57. *Purely functional data structures* (http://www.cambridge.org/us/academic/subjects/computer-science/algorithmics-complexity-computer-algebra-and-computational-g/purely-functional-data-structures) by Chris Okasaki, Cambridge University Press, 1998, ISBN 0-521-66350-4
58. Newbern, J. "All About Monads: A comprehensive guide to the theory and practice of monadic programming in Haskell" (http://monads.haskell.cz/html/index.html/html/). Retrieved 2008-02-14.
59. Larry C. Paulson (28 June 1996). *ML for the Working Programmer* (https://books.google.com/books?id=XppZdaDs7e0C). Cambridge University Press. ISBN 978-0-521-56543-1. Retrieved 10 February 2013.
60. Daniel Spiewak. "Implementing Persistent Vectors in Scala" (http://www.codecommit.com/blog/scala/implementing-persistent-vectors-in-scala) Retrieved Apr 17, 2012.
61. "Which programs are fastest? | Computer Language Benchmarks Game" (http://benchmarksgame.alioth.debian.org/u32/which-programs-are-fastest.php?gcc=on&ghc=on&clean=on&ocaml=on&sbcl=on&fsharp=on&racket=on&clojure=on&hipe=on&calc=chart) benchmarksgame.alioth.debian.org. Retrieved 2011-06-20.
62. Igor Pechtchanski; Vivek Sarkar (2005). "Immutability specification and its applications". *Concurrency and Computation: Practice and Experience* **17** (5–6): 639–662. doi:10.1002/cpe.853 (https://doi.org/10.1002%2Fcpe.853)
63. "Chapter 25. Profiling and optimization" (http://book.realworldhaskell.org/read/profiling-and-optimization.html#x_eK1). Book.realworldhaskell.org. Retrieved 2011-06-20.
64. [1] (https://web.archive.org/web/20131226033005/http://www.aquabu.com:80/2008/02/16/fibonacci-sequence-recursion-in-erlang)
65. Odersky, Martin; Spoon, Lex; Venners, Bill (December 13, 2010). *Programming in Scala: A Comprehensive Step-by-step Guide* (http://www.artima.com/shop/programming_in_scala_2ed) (2nd ed.). Artima Inc. pp. 883/852. ISBN 978-0-9815316-4-9.

66. Piro, Christopher (2009). *Functional Programming at Facebook* (http://cufp.galois.com/2009/abstracts.html#ChristopherPiroEugeneLetuchy) CUFP 2009. Retrieved 2009-08-29.
67. "Sim-Diasca: a large-scale discrete event concurrent simulation engine in Erlang" (http://research.edf.com/research-and-the-scientific-community/software/sim-diasca-80704.html) November 2011.
68. 1 million is so 2011 (http://blog.whatsapp.com/index.php/2012/01/1-million-is-so-2011/) // WhatsApp blog, 2012-01-06: "the last important piece of our infrastracture is Erlang"
69. Momtahan, Lee (2009). *Scala at EDF Trading: Implementing a Domain-Specific Language for Derivative Pricing with Scala* (http://cufp.galois.com/2009/abstracts.html#LeeMomtahan) CUFP 2009. Retrieved 2009-08-29.
70. Graham, Paul (2003). "Beating the Averages" (http://www.paulgraham.com/avg.html) Retrieved 2009-08-29.
71. Sims, Steve (2006). *Building a Startup with Standard ML* (http://cufp.galois.com/2006/slides/SteveSims.pdf) (PDF). CUFP 2006. Retrieved 2009-08-29.
72. Laurikari, Ville (2007). *Functional Programming in Communications Security*. (http://cufp.galois.com/2007/abstracts.html#VilleLaurikari) CUFP 2007. Retrieved 2009-08-29.
73. Lorimer, R. J. "Live Production Clojure Application Announced" (http://www.infoq.com/news/2009/01/clojure_production).
74. Emmanuel Schanzer of Bootstrap (https://twit.tv/shows/triangulation/episodes/196/) interviewed on the TV show Triangulation on the TWiT.tv network

Further reading

- Abelson, Hal; Sussman, Gerald Jay (1985). *Structure and Interpretation of Computer Programs*. MIT Press.
- Cousineau, Guy and Michel Mauny. *The Functional Approach to Programming*. Cambridge, UK: Cambridge University Press, 1998.
- Curry, Haskell Brooks and Feys, Robert and Craig, William. *Combinatory Logic*. Volume I. North-Holland Publishing Company, Amsterdam, 1958.
- Curry, Haskell B.; Hindley, J. Roger; Seldin, Jonathan P. (1972). *Combinatory Logic*. Vol. II. Amsterdam: North Holland. ISBN 0-7204-2208-6.
- Dominus, Mark Jason. *Higher-Order Perl*. Morgan Kaufmann. 2005.
- Felleisen, Matthias; Findler, Robert; Flatt, Matthew; Krishnamurthi, Shriram (2001). *How to Design Programs*. MIT Press.
- Graham, Paul. *ANSI Common LISP*. Englewood Cliffs, New Jersey: Prentice Hall, 1996.
- MacLennan, Bruce J. *Functional Programming: Practice and Theory*. Addison-Wesley, 1990.
- O'Sullivan, Brian; Stewart, Don; Goerzen, John (2008). *Real World Haskell*. O'Reilly.
- 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

- Ford, Neal (2012-01-29). "Functional thinking: Why functional programming is on the rise". Retrieved 2013-02-24.
- Akhmechet, Slava (2006-06-19). "defmacro – Functional Programming For The Rest of Us". Retrieved 2013-02-24. An introduction
- *Functional programming in Python* (by David Mertz): part 1, part 2, part 3