What is the difference between procedural programming and functional programming? [closed]

Asked 16 years, 4 months ago Modified 1 year, 6 months ago

Viewed 191k times

334

Closed. This question is opinion-based. It is not currently accepting answers.

Want to improve this question? Update the question so it can be answered with facts and citations by editing this post.

Closed 5 years ago.

The community reviewed whether to reopen this question last year and left it closed:

> Original close reason(s) were not resolved

Improve this question

I've read the Wikipedia articles for both procedural programming and functional programming, but I'm still slightly confused. Could someone boil it down to the core?

functional-programming glossary paradigms procedural-programming

Share

Improve this question

Follow

edited Aug 25, 2008 at 11:41

Steve M

10.6k • 12 • 53 • 63

asked Aug 22, 2008 at 19:29

Thomas Owens

116k • 99 • 317 • 436

Wikipedia implies that FP is a subset of (i.e. is always) declarative programming, but that is not true and conflates the taxonomy of IP vs. DP. – Shelby Moore III Dec 8, 2011 at 1:05

17 Answers

Sorted by: Highest score (default)

190

A functional language (ideally) allows you to write a mathematical function, i.e. a function that takes *n* arguments and returns a value. If the program is executed, this function is logically evaluated as needed.[1]

A procedural language, on the other hand, performs a series of *sequential* steps. (There's a way of transforming sequential logic into functional logic called continuation passing style.)

As a consequence, a purely functional program always yields *the same value* for an input, and the order of evaluation is not well-defined; which means that uncertain values like user input or random values are hard to model in purely functional languages.

---

[1] As everything else in this answer, that's a generalisation. This property, evaluating a computation when its result is needed rather than sequentially where it's called, is known as "laziness". Not all functional languages are actually universally lazy, nor is laziness restricted to functional programming. Rather, the description given here provides a "mental framework" to think about different programming styles that are not distinct and opposite categories but rather fluid ideas.

Share Improve this answer Follow

edited Aug 6, 2019 at 15:41

answered Aug 22, 2008 at 19:37

Konrad Rudolph

545k ● 139 ● 956 ● 1.2k

11 Uncertain values like user input or random values are hard to model in purely functional languages, but this is a solved problem. See monads. – Apocalisp Dec 23, 2008 at 15:16

"*sequential steps*, where the functional program would be nested" means providing for separation-of-concerns by emphasizing function composition, i.e. separating the dependencies among the subcomputations of a deterministic computation. – Shelby Moore III Dec 8, 2011 at 1:01

this seems wrong - procedures can be also nested, procedures can have parameters – Hurda Jul 7, 2016 at 13:12

2 @Hurda Yes, could have phrased this better. The point is that procedural programming happens stepwise in a pre-determined order, whereas functional programs are not executed stepwise; rather, values are computed when they are needed. However, the lack of a generally agreed upon definition of programming terminology makes such generalisations next to useless. I've amended my answer in that regard. – Konrad Rudolph Jul 7, 2016 at 13:55

119

I've never seen this definition given elsewhere, but I think this sums up the differences given here fairly well:

Functional programming focuses on **expressions**

Procedural programming focuses on **statements**

Expressions have values. A functional program is an expression who's value is a sequence of instructions for the computer to carry out.

Statements don't have values and instead modify the state of some conceptual machine.

In a purely functional language there would be no statements, in the sense that there's no way to manipulate state (they might still have a syntactic construct named "statement", but unless it manipulates state I wouldn't call it a statement in this sense). In a purely procedural language there would be no

expressions, everything would be an instruction which manipulates the state of the machine.

Haskell would be an example of a purely functional language because there is no way to manipulate state. Machine code would be an example of a purely procedural language because everything in a program is a statement which manipulates the state of the registers and memory of the machine.

The confusing part is that the vast majority of programming languages contain **both** expressions and statements, allowing you to mix paradigms. Languages can be classified as more functional or more procedural based on how much they encourage the use of statements vs expressions.

For example, C would be more functional than COBOL because a function call is an expression, whereas calling a sub program in COBOL is a statement (that manipulates the state of shared variables and doesn't return a value). Python would be more functional than C because it allows you to express conditional logic as an expression using short circuit evaluation (test && path1 || path2 as opposed to if statements). Scheme would be more functional than Python because everything in scheme is an expression.

You can still write in a functional style in a language which encourages the procedural paradigm and vice versa. It's just harder and/or more awkward to write in a paradigm which isn't encouraged by the language.

Share Improve this answer

Follow

answered Nov 28, 2012 at 8:32

Omnimike

1,575 ● 2 ● 15 ● 12

7 Best and most succinct explanation I've seen on the web, bravo! – tommed Aug 3, 2018 at 10:15

3 C has boolean expressions too – portalguy15837 Nov 19, 2020 at 20:24

1 Very good explanation. Comments: *"Python would be more functional than C because it allows you to express conditional logic as an expression using short circuit evaluation (test && path1 || path2 as opposed to if statements)"* In C you can also do `test && path1 || path2`. *"Statements don't have values and instead modify the state of some conceptual machine."* Hard to think of an example for this one, unless I misunderstand this definition. For example `return` I consider to be a statement, but on its own, does not modify state. – Nagev Feb 11, 2021 at 17:37

@Nagev a = sum(2, 3); the `return` in `sum` is modifying the variable `a` right? So basically state change? – Midhunraj R Pillai Mar 7, 2021 at 15:12

106

Basically the two styles, are like Yin and Yang. One is organized, while the other chaotic. There are situations when Functional programming is the obvious choice, and other situations were Procedural programming is the better choice. This is why there are at least two languages that have recently come out with a new version, that embraces both programming styles. *(Perl 6 and D 2)*

Procedural:

- The output of a routine does not always have a direct correlation with the input.
- Everything is done in a specific order.
- Execution of a routine may have side effects.
- Tends to emphasize implementing solutions in a linear fashion.

Perl 6 

```
sub factorial ( UInt:D $n is copy ) returns UInt {

  # modify "outside" state
  state $call-count++;
  # in this case it is rather pointless as
  # it can't even be accessed from outside

  my $result = 1;

  loop ( ; $n > 0 ; $n-- ){

    $result *= $n;

  }

  return $result;
}
```

D 2

```
int factorial( int n ){

  int result = 1;
```

```
  for( ; n > 0 ; n-- ){
    result *= n;
  }

  return result;
}
```

Functional:

- Often recursive.
- Always returns the same output for a given input.
- Order of evaluation is usually undefined.
- Must be stateless. i.e. No operation can have side effects.
- Good fit for parallel execution
- Tends to emphasize a divide and conquer approach.
- May have the feature of Lazy Evaluation.

Haskell

(copied from Wikipedia);

```
fac :: Integer -> Integer

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

or in one line:

```
fac n = if n > 0 then n * fac (n-1) else 1
```

Perl 6 

```
proto sub factorial ( UInt:D $n ) returns UInt {*}

multi sub factorial (  0 ) { 1 }
multi sub factorial ( $n ) { $n * samewith $n-1 } # {
```

D 2

```
pure int factorial( invariant int n ){
  if( n <= 1 ){
    return 1;
  }else{
    return n * factorial( n-1 );
  }
}
```

Side note:

Factorial is actually a common example to show how easy it is to create new operators in Perl 6 the same way you would create a subroutine. This feature is so ingrained into Perl 6 that most operators in the Rakudo implementation are defined this way. It also allows you to add your own multi candidates to existing operators.

```
sub postfix:< ! > ( UInt:D $n --> UInt )
  is tighter(&infix:<*>)
  { [*] 2 .. $n }
```

```
say 5!; # 120
```

This example also shows range creation (`2..$n`) and the list reduction meta-operator (`[ OPERATOR ] LIST`) combined with the numeric infix multiplication operator. (`*`)
It also shows that you can put `--> UInt` in the signature instead of `returns UInt` after it.

(You can get away with starting the range with `2` as the multiply "operator" will return `1` when called without any arguments)

Share Improve this answer

Follow

edited May 7, 2023 at 18:23

Orión González
315 ● 2 ● 15

answered Aug 22, 2008 at 22:46

Brad Gilbert
34.1k ● 11 ● 79 ● 130

Hi, can you please provide an example for the following 2 points mentioned for "Procedural" considering the example of factorial implementation in Perl 6. 1) The output of a routine does not always have a direct correlation with the input. 2) Execution of a routine may have side effects. – Naga Kiran Sep 27, 2009 at 14:32

`sub postfix:<!> ($n) { [*] 1..$n }` – Brad Gilbert Nov 6, 2009 at 23:00

@BradGilbert `No operation can have side effects` - Can you please elaborate it ? – HalfWebDev Feb 21, 2012 at 18:53

2 Probably the best answer I could ever found.... And, I did some research on those individual points.. that really helped me! :) – Navaneeth Jul 14, 2015 at 13:30

2 @AkashBisariya `sub foo( $a, $b ){ ($a,$b).pick }` ← does not always return the same output for the same input, while the following does `sub foo( $a, $b ){ $a + $b }` – Brad Gilbert Apr 22, 2018 at 16:10

74

Funtional Programming

```
num = 1
def function_to_add_one(num):
    num += 1
    return num


function_to_add_one(num) # expression that evaluates t
function_to_add_one(num) # expression that evaluates t
function_to_add_one(num) # expression that evaluates t
function_to_add_one(num) # expression that evaluates t
function_to_add_one(num) # expression that evaluates t

#Final Output: 2
```

Procedural Programming

```
num = 1
def procedure_to_add_one():
    global num
    num += 1
    return num
```

```
procedure_to_add_one() # statement that changes intern
procedure_to_add_one() # statement that changes intern
procedure_to_add_one() # statement that changes intern
procedure_to_add_one() # statement that changes intern
procedure_to_add_one() # statement that changes intern

#Final Output: 6
```

`function_to_add_one` is a function

`procedure_to_add_one` is a procedure

Even if you run the **function** five times, every time it will return **2**

If you run the **procedure** five times, at the end of fifth run it will give you **6**.

Expressions vs Statements

As you can see in the code above, functions always evaluate to the same output given the same input. And since they evaluate to a value, in technical terms they are expressions.

On the other hand, procedures don't evaluate to a value, they might not return anything, just change internal state and therefore they are NOT expressions but statements.

After seeing these examples, Omnimike's answer will be even easier to understand.

DISCLAIMER: Obviously this is a hyper-simplified view of reality. This answer just gives a taste of "functions" as opposed to "procedures". Nothing more. Once you have tasted this superficial yet deeply penetrative intuition, start exploring the two paradigms, and you will start to see the difference quite clearly.

Helps my students, hope it helps you too.

Share Improve this answer Follow

edited Jun 12, 2023 at 6:19

answered Jan 18, 2019 at 11:29

Hamza Zubair

1,400 ● 16 ● 23

19 this example is really simple to understand the term of "stateless" & "immutable data" in functional programming, reading through all the definitions & differences listed above didn't clear my confusion until reading this answer. Thank you! – maximus Mar 3, 2019 at 10:53

57

In computer science, functional programming is a programming paradigm that treats computation as the evaluation of mathematical functions and avoids state and mutable data. It emphasizes the application of functions, in contrast with the procedural programming style that emphasizes changes in state.

Share Improve this answer

answered Aug 22, 2008 at 19:34

Follow

juan

81.8k ● 52 ● 164 ● 198

4 While this is the explanation that has helped me the most, I am still fuzzy on the concept of functional programming. I am looking for a style of programming that doesn't depend on referencing external objects in order to run (every thing the function needs to run should be passed in as a parameter). For example, I would never put `GetUserContext()` in the function, the user context would be passed in. Is this functional programming? Thanks in advance. – Matt Cashatt Nov 18, 2014 at 15:34

26

I believe that procedural/functional/objective programming are about how to approach a problem.

The first style would plan everything in to steps, and solves the problem by implementing one step (a procedure) at a time. On the other hand, functional programming would emphasize the divide-and-conquer approach, where the problem is divided into sub-problem, then each sub-problem is solved (creating a function to solve that sub problem) and the results are combined to create the answer for the whole problem. Lastly, Objective programming would mimic the real world by create a mini-world inside the computer with many objects, each of which has a (somewhat) unique characteristics, and interacts with others. From those interactions the result would emerge.

Each style of programming has its own advantages and weaknesses. Hence, doing something such as "pure

programming" (i.e. purely procedural - no one does this, by the way, which is kind of weird - or purely functional or purely objective) is very difficult, if not impossible, except some elementary problems specially designed to demonstrate the advantage of a programming style (hence, we call those who like pureness "weenie" :D).

Then, from those styles, we have programming languages that is designed to optimized for some each style. For example, Assembly is all about procedural. Okay, most early languages are procedural, not only Asm, like C, Pascal, (and Fortran, I heard). Then, we have all famous Java in objective school (Actually, Java and C# is also in a class called "money-oriented," but that is subject for another discussion). Also objective is Smalltalk. In functional school, we would have "nearly functional" (some considered them to be impure) Lisp family and ML family and many "purely functional" Haskell, Erlang, etc. By the way, there are many general languages such as Perl, Python, Ruby.

edited Apr 10, 2015 at 3:07

Aaron Hall ♦

394k ● 91 ● 412 ● 339

answered Aug 22, 2008 at 19:57

Lam Luu

To expand on Konrad's comment:

13

> As a consequence, a purely functional program always yields the same value for an input, and the order of evaluation is not well-defined;

Because of this, functional code is generally easier to parallelize. Since there are (generally) no side effects of the functions, and they (generally) just act on their arguments, a lot of concurrency issues go away.

Functional programming is also used when you need to be capable of *proving* your code is correct. This is much harder to do with procedural programming (not easy with functional, but still easier).

Disclaimer: I haven't used functional programming in years, and only recently started looking at it again, so I might not be completely correct here. :)

Share Improve this answer

Follow

answered Aug 22, 2008 at 19:53

Herms

38.7k ● 13 ● 79 ● 104

13

One thing I hadn't seen really emphasized here is that modern functional languages such as Haskell really more on first class functions for flow control than explicit recursion. You don't need to define factorial recursively in Haskell, as was done above. I think something like

```
fac n = foldr (*) 1 [1..n]
```

is a perfectly idiomatic construction, and much closer in spirit to using a loop than to using explicit recursion.

Share Improve this answer

Follow

answered Aug 24, 2008 at 12:28

C Hogg

1,001 ● 10 ● 15

13

A functional programming is identical to procedural programming in which global variables are *not* being used.

Share Improve this answer

Follow

edited Mar 4, 2017 at 6:22

answered Mar 3, 2017 at 14:41

Nir O.

1,651 ● 1 ● 19 ● 29

10

Procedural languages tend to keep track of state (using variables) and tend to execute as a sequence of steps. Purely functional languages don't keep track of state, use immutable values, and tend to execute as a series of dependencies. In many cases the status of the call stack will hold the information that would be equivalent to that which would be stored in state variables in procedural code.

Recursion is a classic example of functional style programming.

Share Improve this answer

Follow

answered Aug 22, 2008 at 21:11

Wedge

19.8k ● 7 ● 50 ● 71

1 After reading this page i was thinking of the same thing ->"Recursion is a classic example of functional style programming", and you cleared it.Thanks, Now I Think Am getting Some thing. – Mysterious Jack Aug 1, 2014 at 9:29

6

Konrad said:

> As a consequence, a purely functional program always yields the same value for an input, and the order of evaluation is not well-defined; which means that uncertain values like user input or random values are hard to model in purely functional languages.

The order of evaluation in a purely functional program may be hard(er) to reason about (especially with laziness) or even unimportant but I think that saying it is not well defined makes it sound like you can't tell if your program is going to work at all!

Perhaps a better explanation would be that control flow in functional programs is based on when the value of a function's arguments are needed. The Good Thing about this that in well written programs, state becomes explicit: each function lists its inputs as parameters instead of arbitrarily munging global state. So on some level, *it is*

easier to reason about order of evaluation with respect to one function at a time. Each function can ignore the rest of the universe and focus on what it needs to do. When combined, functions are guaranteed to work the same[1] as they would in isolation.

> ... uncertain values like user input or random values are hard to model in purely functional languages.

The solution to the input problem in purely functional programs is to embed an imperative language as a DSL using a sufficiently powerful abstraction. In imperative (or non-pure functional) languages this is not needed because you can "cheat" and pass state implicitly and order of evaluation is explicit (whether you like it or not). Because of this "cheating" and forced evaluation of all parameters to every function, in imperative languages 1) you lose the ability to create your own control flow mechanisms (without macros), 2) code isn't inherently thread safe and/or parallelizable *by default*, 3) and implementing something like undo (time travel) takes careful work (imperative programmer must store a recipe for getting the old value(s) back!), whereas pure functional programming buys you all these things—and a few more I may have forgotten—"for free".

I hope this doesn't sound like zealotry, I just wanted to add some perspective. Imperative programming and especially mixed paradigm programming in powerful

languages like C# 3.0 are still totally effective ways to get things done and [there is no silver bullet](#).

[1] ... except possibly with respect memory usage (cf. foldl and foldl' in Haskell).

Share Improve this answer Follow

edited Nov 6, 2009 at 21:56

answered Aug 26, 2008 at 15:13

Jared Updike

7,268 ● 8 ● 48 ● 73

5

To expand on Konrad's comment:

> and the order of evaluation is not well-defined

Some functional languages have what is called Lazy Evaluation. Which means a function is not executed until the value is needed. Until that time the function itself is what is passed around.

Procedural languages are step 1 step 2 step 3... if in step 2 you say add 2 + 2, it does it right then. In lazy evaluation you would say add 2 + 2, but if the result is never used, it never does the addition.

Share Improve this answer Follow

edited Feb 16, 2009 at 14:10

Svante

51.4k ● 11 ● 83 ● 124

answered Aug 22, 2008 at 20:23

Brian Leahy
35.4k ● 12 ● 46 ● 60

4

If you have a chance, I would recommand getting a copy of Lisp/Scheme, and doing some projects in it. Most of the ideas that have lately become bandwagons were expressed in Lisp decades ago: functional programming, continuations (as closures), garbage collection, even XML.

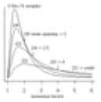
So that would be a good way to get a head start on all these current ideas, and a few more besides, like symbolic computation.

You should know what functional programming is good for, and what it isn't good for. It isn't good for everything. Some problems are best expressed in terms of side-effects, where the same question gives differet answers depending on when it is asked.

Share Improve this answer

Follow

answered Dec 23, 2008 at 15:10



Mike Dunlavey
40.6k ● 15 ● 94 ● 138

3

@Creighton:

In Haskell there is a library function called **product**:

```
prouduct list = foldr 1 (*) list
```

or simply:

```
product = foldr 1 (*)
```

so the "idiomatic" factorial

```
fac n = foldr 1 (*)  [1..n]
```

would simply be

```
fac n = product [1..n]
```

Share Improve this answer

Follow

answered Aug 26, 2008 at 15:20

Jared Updike

7,268 ● 8 ● 48 ● 73

This does not provide an answer to the question. To critique or request clarification from an author, leave a comment below their post. – Nick Kitto Apr 10, 2015 at 4:20

I believe this was posted many years ago, before the comment system was added, if you can believe it: stackoverflow.com/help/badges/30/beta?userid=2543 – Jared Updike Apr 12, 2015 at 16:25

3

Procedural programming divides sequences of statements and conditional constructs into separate blocks called procedures that are parameterized over arguments that are (non-functional) values.

Functional programming is the same except that functions are first-class values, so they can be passed as arguments to other functions and returned as results from function calls.

Note that functional programming is a generalization of procedural programming in this interpretation. However, a minority interpret "functional programming" to mean side-effect-free which is quite different but irrelevant for all major functional languages except Haskell.

Share Improve this answer

Follow

edited Apr 10, 2015 at 3:11

Aaron Hall ♦

394k ● 91 ● 412 ● 339

answered Oct 19, 2008 at 4:40

J D

48.6k ● 14 ● 174 ● 277

2

None of the answers here show idiomatic functional programming. The recursive factorial answer is great for representing recursion in FP, but the majority of code is not recursive so I don't think that answer is fully representative.

Say you have an arrays of strings, and each string represents an integer like "5" or "-200". You want to check this input array of strings against your internal test case (Using integer comparison). Both solutions are shown below

Procedural

```
arr_equal(a : [Int], b : [Str]) -> Bool {
    if(a.len != b.len) {
        return false;
    }

    bool ret = true;
    for( int i = 0; i < a.len /* Optimized with &&
ret*/; i++ ) {
        int a_int = a[i];
        int b_int = parseInt(b[i]);
        ret &= a_int == b_int;
    }
    return ret;
}
```

Functional

```
eq = i, j => i == j # This is usually a built-in
toInt = i => parseInt(i) # Of course, parseInt ===
toInt here, but this is for visualization

arr_equal(a : [Int], b : [Str]) -> Bool =
    zip(a, b.map(toInt)) # Combines into [Int,
Int]
   .map(eq)
   .reduce(true, (i, j) => i && j) # Start with
true, and continuously && it with each value
```

While pure functional languages are generally research languages (As the real-world likes free side-effects), real-world procedural languages will use the much simpler functional syntax when appropriate.

This is usually implemented with an external library like Lodash, or available built-in with newer languages like Rust. The heavy lifting of functional programming is done with functions/concepts like `map`, `filter`, `reduce`, `currying`, `partial`, the last three of which you can look up for further understanding.

Addendum

In order to be used in the wild, the compiler will normally have to work out how to convert the functional version into the procedural version internally, as function call overhead is too high. Recursive cases such as the factorial shown will use tricks such as tail call to remove O(n) memory usage. The fact that there are no side effects allows functional compilers to implement the `&& ret` optimization even when the `.reduce` is done last. Using Lodash in JS, obviously does not allow for any optimization, so it is a hit to performance (Which isn't usually a concern with web development). Languages like Rust will optimize internally (And have functions such as `try_fold` to assist `&& ret` optimization).

Share Improve this answer Follow

edited Nov 1, 2018 at 16:42

answered Nov 1, 2018 at 0:45

Nicholas Pipitone
4,122 ● 4 ● 25 ● 39

1

To Understand the difference, one needs to to understand that "the godfather" paradigm of both procedural and functional programming is the **imperative programming**.

Basically procedural programming is merely a way of structuring imperative programs in which the primary method of abstraction is the "procedure." (or "function" in some programming languages). Even Object Oriented Programming is just another way of structuring an imperative program, where the state is encapsulated in objects, becoming an object with a "current state," plus this object has a set of functions, methods, and other stuff that let you the programmer manipulate or update the state.

Now, in regards to functional programming, the *gist* in its approach is that it identifies what values to take and how these values should be transferred. (so there is no state, and no mutable data as it takes functions as first class values and pass them as parameters to other functions).

PS: understanding every programming paradigm is used for should clarify the differences between all of them.

PSS: In the end of the day, programming paradigms are just different approaches to solving problems.

PSS: this quora answer has a great explanation.

edited Nov 22, 2018 at 20:44

answered Nov 22, 2018 at 20:16

Fouad Boukredine
1,623 ● 15 ● 19