

Functional Programming

Aayush Bajaj

July 10, 2026

Contents

1	the paradigm	1
2	lambda calculus	2
2.1	syntax and reduction	2
2.2	church numerals	2
2.3	the y combinator	3
3	higher-order functions	3
4	closures and currying	4
5	purity and effects: monads	4
6	lazy evaluation	5
7	pattern matching and algebraic data types	5
8	fp without a functional language	6
8.1	python, executed	6
8.2	elisp, executed	6
9	see also	7

unlike its siblings on this branch of the wiki — [linear](#), [quadratic](#), [integer](#) — this “programming” really is about writing programs. functional programming is the discipline of building software out of *expressions that are evaluated* rather than *statements that are executed*: no assignment, no mutation, no time. what remains is algebra, and algebra is something you can reason about.

*sicp is the canonical sermon:
<https://mit-press.mit.edu/9780262510875/>
— and learn you a haskell the
friendliest hymnal:
<https://learnyoua-haskell.github.io>*

1 the paradigm

three commitments define it:

- **first-class functions**: functions are values — passed as arguments, returned as results, stored in lists. the “higher-order” machinery below is just this taken seriously.
- **immutability**: data is never updated in place; “change” means building a new value that shares structure with the old. persistent [data structures](#) make this cheap (a cons prepend is $\mathcal{O}(1)$ and the old list survives untouched).
- **referential transparency**: an expression can always be replaced by its value.

$f(2)$ today equals $f(2)$ tomorrow — no hidden state, no spooky action. this is what makes equational reasoning, memoisation, parallelisation and caching *correct by default* rather than by audit.

the cost, of course, is that real programs must eventually mutate the world — print, write, launch the missiles. the paradigm’s genuinely deep contribution is not avoiding effects but *quarantining* them (see monads below).

2 lambda calculus

church’s 1932-33 calculus is the assembly language of the paradigm — three grammar rules and one rewrite rule, turing-complete.

2.1 syntax and reduction

$$M ::= x \mid \lambda x. M \mid M N \quad (1)$$

variables, abstraction (anonymous function), application. computation is a single rule, **beta reduction** — substitute the argument for the bound variable:

$$(\lambda x. M) N \rightarrow_{\beta} M[x := N], \quad (2)$$

plus alpha-conversion (bound names are arbitrary: $\lambda x.x = \lambda y.y$) and eta ($\lambda x. f x = f$ when x not free in f). a term with no beta-redex is in *normal form*; church-rosser guarantees normal forms are unique when they exist — evaluation order cannot change the answer, only whether you reach it.

2.2 church numerals

encode n as “apply f n times”:

$$\begin{array}{ll} 0 = \lambda f. \lambda x. x & \text{succ} = \lambda n. \lambda f. \lambda x. f (n f x) \\ 1 = \lambda f. \lambda x. f x & \text{plus} = \lambda m. \lambda n. \lambda f. \lambda x. m f (n f x) \\ 2 = \lambda f. \lambda x. f (f x) & \text{mult} = \lambda m. \lambda n. \lambda f. m (n f) \end{array}$$

a worked reduction of $\text{plus } 2 \ 1$ (each displayed arrow bundles the two β -steps of a curried application):

$$\begin{aligned} \text{plus } 2 \ 1 &= (\lambda m. \lambda n. \lambda f. \lambda x. m f (n f x)) \ 2 \ 1 \\ &\rightarrow_{\beta} \lambda f. \lambda x. 2 f (1 f x) \\ &\rightarrow_{\beta} \lambda f. \lambda x. 2 f (f x) \\ &\rightarrow_{\beta} \lambda f. \lambda x. f (f (f x)) = 3, \end{aligned}$$

where the second step unfolds $1 f x \rightarrow_{\beta} f x$ and the third unfolds $2 f$ applied to $f x$. arithmetic by function composition alone — numbers never appear.

booleans too: true = a function that keeps its first argument, false = one that keeps its second. if-then-else is just application

2.3 the y combinator

lambda terms are anonymous, so recursion cannot mention itself by name. the fix: manufacture self-reference from self-application. define

$$Y = \lambda f. (\lambda x. f (x x)) (\lambda x. f (x x)). \quad (3)$$

apply it to any g and watch it regenerate itself:

$$\begin{aligned} Y g &= (\lambda x. g (x x)) (\lambda x. g (x x)) \\ &\rightarrow_{\beta} g ((\lambda x. g (x x)) (\lambda x. g (x x))) \\ &= g (Y g). \end{aligned}$$

so Yg is a *fixed point* of g : feed g a recipe expecting “the recursive function” as an argument, and Y ties the knot. the honest caveats: the unfolding $Yg \rightarrow g(Yg) \rightarrow g(g(Yg)) \rightarrow \dots$ never terminates on its own — it is g ’s job to stop consuming it at the base case — and under strict call-by-value evaluation the inner xx is evaluated eagerly, so Y diverges immediately. strict languages use the eta-expanded **Z combinator**, $Z = \lambda f. (\lambda x. f(\lambda v. x x v))(\lambda x. f(\lambda v. x x v))$, which delays the self-application behind a lambda — the python demo below runs exactly this.

3 higher-order functions

the everyday trio, each a loop with the loop abstracted away:

- `map f xs` — apply f elementwise. structure preserved.
- `filter p xs` — keep the elements satisfying the predicate.
- `fold f z xs` — *collapse* the list: replace every constructor with f and the empty list with z .

fold is the senior partner. a list is built from two constructors — `cons (:)` and `nil []` — and `foldr f z` is precisely “reinterpret those constructors as f and z ”:

$$\text{foldr } (f, z, [x_1, x_2, x_3]) = f(x_1, f(x_2, f(x_3, z))). \quad (4)$$

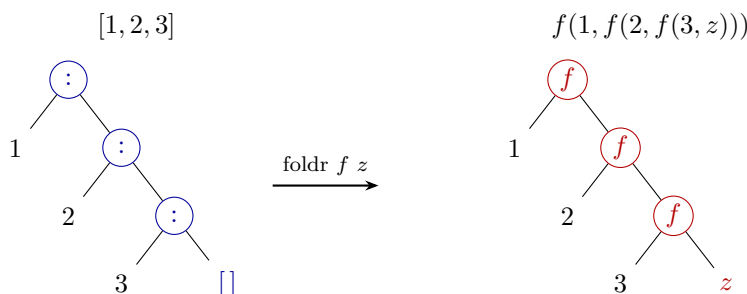


Figure 1: foldr is constructor replacement: the list $[1, 2, 3]$ is the left tree; $\text{foldr } (f, z)$ rebuilds it with every cons node relabelled f and nil relabelled z . with $f = +$, $z = 0$ the right tree evaluates to the sum.

fold is *universal* for structural recursion on lists: any function defined by “do something with the head, recurse on the tail, base case for nil” is a fold with the right f

and z . in particular `map` and `filter` are folds (proved by running them, below), sums and products are folds, and manipulating the recurrences that folds compute is a whole art (Graham, Ronald L. and Knuth, Donald E. and Patashnik, Oren, 1994). the left-handed cousin `foldl` associates the other way — tail-recursive, constant stack with strict accumulation, but it cannot short-circuit or digest infinite lists the way a lazy `foldr` can.

4 closures and currying

- a **closure** is a function value plus the environment it was born in: the returned lambda in `adder(7)` remembers `n = 7` forever. closures are how pure languages model objects — and how impure ones smuggle in private state.
- **currying** rewrites an n -argument function as a chain of one-argument functions: $f : (A \times B) \rightarrow C$ becomes $f : A \rightarrow (B \rightarrow C)$. in haskell every function is curried by construction — `plus 3` is not an error but a new function. partial application falls out for free, and so does the point-free style where programs are compositions of specialised combinators.

the venerable koan: objects are a poor man's closures, and closures are a poor man's objects

5 purity and effects: monads

the quarantine mechanism. a pure language cannot “just” do IO or “just” fail — instead, effectful computations are *values of a wrapped type*, and a monad is the interface for chaining them. two operations and three laws:

the laws: return is a left and right identity for bind, and bind is associative — a monoid-flavoured contract, nothing mystical

```
1 class Monad m where
2   return :: a -> m a           -- wrap a pure value
3   (>=)    :: m a -> (a -> m b) -> m b -- "bind": chain, threading the effect
```

maybe — computations that can fail. failure short-circuits the rest of the chain without a single if-null check:

```
1 data Maybe a = Nothing | Just a
2
3 instance Monad Maybe where
4   return = Just
5   Nothing >= _ = Nothing      -- failure is contagious
6   (Just x) >= f = f x         -- success feeds forward
7
8 safeDiv :: Int -> Int -> Maybe Int
9 safeDiv _ 0 = Nothing
10 safeDiv a b = Just (a `div` b)
11
12 -- safeDiv 100 5 >= safeDiv 300 ==> Just 15
13 -- safeDiv 100 0 >= safeDiv 300 ==> Nothing, and nothing crashed
```

io — the world itself as the threaded state. conceptually `IO a` behaves like a function `World -> (a, World)`; `main` is one big composed IO value that the runtime executes. purity survives because *composing* descriptions of effects is pure — execution happens off-stage:

```
1 greet :: IO ()
2 greet = getLine >= \name -> putStrLn ("hello, " ++ name)
```

(the haskell blocks on this page are notation for reading, not artefacts of a run —

no ghc on this machine; the python and elisp blocks below are executed for real.) the practical moral survives translation into any language: separate the *description* of an effect from its *execution*, and error handling becomes composition instead of boilerplate — rust’s `?` on `Result`, javascript promise chains, python’s `Optional` pipelines are all monadic `bind` wearing local costume.

6 lazy evaluation

haskell evaluates outermost-first and only on demand, so definitions may be self-referentially infinite:

```
1 naturals = 0 : map (+1) naturals      -- [0,1,2,3,...]
2 fibs    = 0 : 1 : zipWith (+) fibs (tail fibs)
3 -- take 8 fibs ==> [0,1,1,2,3,5,8,13]
```

`take 8` forces exactly eight cells; the rest of the infinite list stays an unevaluated thunk. laziness is what lets `foldr` short-circuit, lets control flow be written as ordinary functions, and separates *generation* from *consumption* — the producer needn’t know how much the consumer wants. python’s generators are the strict world’s translation, executed here:

```
1 from itertools import islice
2
3 def fibs():                # an "infinite list", lazily
4     a, b = 0, 1
5     while True:
6         yield a
7         a, b = b, a + b
8
9 print(list(islice(fibs(), 8)))
```

[0, 1, 1, 2, 3, 5, 8, 13]

7 pattern matching and algebraic data types

data in FP is built from **sums** (“one of these shapes”) and **products** (“these fields together”) — algebraic data types. functions are then defined by *pattern matching*, one equation per constructor, and the compiler checks you handled every case:

```
1 data Shape = Circle Double | Rect Double Double -- a sum of products
2
3 area :: Shape -> Double
4 area (Circle r) = pi * r * r
5 area (Rect w h) = w * h
6
7 data Tree a = Leaf | Node (Tree a) a (Tree a)
8
9 depth :: Tree a -> Int
10 depth Leaf = 0
11 depth (Node l _ r) = 1 + max (depth l) (depth r)
```

exhaustiveness checking is the quiet superpower: add a `Triangle` constructor and every non-updated `area` in the codebase becomes a compile error, not a 3am page. compare the object-oriented encoding — one class per shape, one method per operation — which makes adding *shapes* easy and adding *operations* hard; ADTs invert

the trade. the two decompositions of the “expression problem” are duals, and **object-oriented programming** is the other half of the story.

8 fp without a functional language

8.1 python, executed

everything above compiles down to lambdas — including the numerals and the fixed-point combinator. run honestly:

```
1  from functools import reduce
2
3  # church numerals as python lambdas: n = "apply f n times"
4  zero = lambda f: lambda x: x
5  succ = lambda n: lambda f: lambda x: f(n(f)(x))
6  plus = lambda m: lambda n: lambda f: lambda x: m(f)(n(f)(x))
7  mult = lambda m: lambda n: lambda f: m(n(f))
8  to_int = lambda n: n(lambda k: k + 1)(0)
9
10 one, two, three = succ(zero), succ(succ(zero)), succ(succ(succ(zero)))
11 print("plus two one =", to_int(plus(two)(one)))
12 print("mult two three =", to_int(mult(two)(three)))
13
14 # the Z combinator (call-by-value Y): anonymous recursion
15 Z = lambda f: (lambda x: f(lambda v: x(x)(v)))(lambda x: f(lambda v: x(x)(v)))
16 fact = Z(lambda rec: lambda n: 1 if n == 0 else n * rec(n - 1))
17 print("Z-combinator factorial(6) =", fact(6))
18
19 # fold as the universal recursion scheme: map and filter are folds
20 foldr = lambda f, z, xs: z if not xs else f(xs[0], foldr(f, z, xs[1:]))
21 mapf = lambda g, xs: foldr(lambda h, acc: [g(h)] + acc, [], xs)
22 filterf = lambda p, xs: foldr(lambda h, acc: [h] + acc if p(h) else acc, [],
23                               ↪ xs)
24 print("map (^2) via fold:", mapf(lambda v: v * v, [1, 2, 3, 4, 5]))
25 print("filter odd via fold:", filterf(lambda v: v % 2, [1, 2, 3, 4, 5]))
26 print("sum via reduce:", reduce(lambda a, b: a + b, [1, 2, 3, 4, 5], 0))
27
28 # closures + currying
29 def adder(n): # returns a closure over n
30     return lambda m: m + n
31 add7 = adder(7)
32 print("closure add7(35) =", add7(35))
```

```
plus two one = 3
mult two three = 6
Z-combinator factorial(6) = 720
map (^2) via fold: [1, 4, 9, 16, 25]
filter odd via fold: [1, 3, 5]
sum via reduce: 15
closure add7(35) = 42
```

church arithmetic actually reducing to 3 and 6 inside a mainstream runtime is the whole thesis of this page in seven lines.

8.2 elisp, executed

emacs lisp is the functional dialect this wiki is written in. with lexical binding it has honest closures; run with `emacs --batch -l fp-demo.el`:

elisp is a lisp-2: functions and variables live in separate namespaces, hence all the funcalls

```
1  ;; -*- lexical-binding: t; -*-
```

```

2 ;; higher-order functions and closures in elisp (a lisp-2: note the funcalls)
3 (require 'cl-lib)
4
5 (defun compose (f g) (lambda (x) (funcall f (funcall g x))))
6 (defun curry2 (f) (lambda (a) (lambda (b) (funcall f a b))))
7
8 (let* ((inc (lambda (n) (1+ n)))
9        (dbl (lambda (n) (* 2 n)))
10       (inc-then-dbl (compose dbl inc))
11       (add10 (funcall (curry2 #'+) 10)))
12   (princ (format "compose: %d\n" (funcall inc-then-dbl 20)))
13   (princ (format "curried add 10 -> %d\n" (funcall add10 32))))
14
15 ;; map / filter / fold
16 (princ (format "mapcar: %S\n" (mapcar (lambda (x) (* x x)) '(1 2 3 4 5))))
17 (princ (format "seq-filter: %S\n" (seq-filter #'cl-oddp '(1 2 3 4 5))))
18 (princ (format "seq-reduce: %d\n" (seq-reduce #'+ '(1 2 3 4 5) 0)))
19
20 ;; a closure that counts – private state without a global
21 (defun make-counter ()
22   (let ((n 0)) (lambda () (setf n (1+ n)))))
23 (let ((c (make-counter)))
24   (funcall c) (funcall c)
25   (princ (format "counter after three calls: %d\n" (funcall c))))

```

```

compose: 42
curried add 10 -> 42
mapcar: (1 4 9 16 25)
seq-filter: (1 3 5)
seq-reduce: 15
counter after three calls: 3

```

note `make-counter`: a closure *mutating its captured variable* — elisp is unbothered by purity, and the counter is exactly the poor-man’s-object koan made executable. more on the [elisp notes](#) page.

9 see also

- [object-oriented programming](#) — the dual decomposition of the expression problem
- [dynamic programming](#) — memoisation is referential transparency, cashed in
- [data structures](#) — persistence and structure sharing, the price of immutability
- [elisp notes](#) — this wiki’s own functional dialect
- [linear programming](#) — the unrelated “programming” next door

references

Graham, Ronald L. and Knuth, Donald E. and Patashnik, Oren (1994). *Concrete Mathematics: A Foundation for Computer Science*, Addison-Wesley.