

04834580 Software Engineering (Honor Track) 2025-26

# Functional Programming Patterns

Sergey Mechtaev

[mechtaev@pku.edu.cn](mailto:mechtaev@pku.edu.cn)

School of Computer Science, Peking University



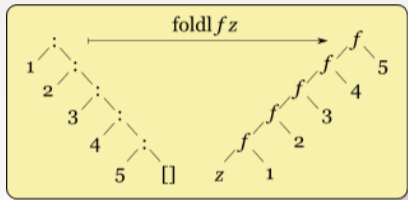
John Backus, in his 1977 Turing Award lecture [1]:

*Can programming be liberated from the von Neumann style? [...] Associated with the functional style of programming is an algebra of programs whose variables range over programs and whose operations are combining forms.*

- ▶ Functional patterns provide powerful **abstractions** for combining computations [2]
- ▶ **Folds** capture common patterns of recursion over data structures
- ▶ **Monads** structure sequential computations with effects (errors, IO, logging)
- ▶ These patterns are now mainstream: Python, Java, C++ all adopt functional idioms

The reduce (or foldl) function takes a list of values and reduces it to a single value.

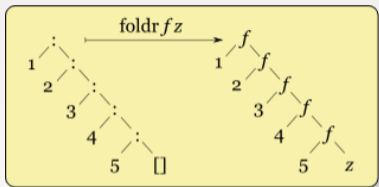
```
from functools import reduce
f = lambda x, y: x * y
product = reduce(f, [1, 2, 3, 4, 5])
```



```
foldl = lambda func, acc, xs: reduce(func, xs, acc)
>>> foldl(operator.sub, 0, [1,2,3])
-6
>>> foldl(operator.add, 'L', ['1','2','3'])
'L123'
```

Step-by-step:  $\text{reduce}(*, [1,2,3,4,5]) = (((((1*2)*3)*4)*5) = (((2*3)*4)*5) = ((6*4)*5) = (24*5) = 120$

The foldr function applies a function against an accumulator and each value of a foldable structure from right to left, folding it to a single value.



```
import functools
import operator
foldr = lambda func, acc, xs: reduce(lambda x, y: func(y, x), xs[::-1], acc)
>>> foldr(operator.sub, 0, [1,2,3])
2
>>> foldr(operator.add, 'R', ['1', '2', '3'])
'123R'
```

Note: `xs[::-1]` is the Python idiomatic way to return the reverse of a list. Contrast:

```
foldr(-, 0, [1,2,3]) = 1-(2-(3-0)) = 1-(2-3) = 1-(-1) = 2 vs foldl(-, 0, [1,2,3]) =
((0-1)-2)-3 = -6
```

Finding max and min elements, computing the product:

```
lmax = lambda xs: reduce(lambda x, y: x if x > y else y, xs)
>>> lmax([1,2,3,4,5])
5
lmin = lambda xs: reduce(lambda x, y: x if x < y else y, xs)
>>> lmin([1,2,3,4,5])
1
lsum = lambda xs: reduce(operator.add, xs)
>>> lsum([1,2,3,4,5])
15
product = lambda xs: reduce(operator.mul, xs)
>>> product([1,2,3,4,5])
120
```

Predicates on list elements:

```
lany = lambda pred, xs: reduce(lambda x, y: x or pred(y), xs, False)
>>> lany(lambda x: x > 3, [1,2])
False
>>> lany(lambda x: x > 3, [1,2,3,4,5,6])
True
lall = lambda pred, xs: reduce(lambda x, y: x and pred(y), xs, True)
>>> lall(lambda x: x > 3, [4,5,6,7])
True
>>> lall(lambda x: x > 3, [1,2])
```

Defining other high-level functions:

```
lmap = lambda func, xs: reduce(lambda x, y: x + [func(y)], xs, [])
```

```
>>> lmap(lambda x: x + 2, [1,2,3,4,5])
```

```
[3, 4, 5, 6, 7]
```

```
lfilter = lambda func, xs: reduce(lambda x, y: x + [y] if func(y) else x, xs, [])
```

```
>>> lfilter(lambda x: x % 2 == 0, [1,2,3,4,5,6,7,8,9])
```

```
[2, 4, 6, 8]
```

Insertion fold algorithm:

```
def insert(y, xs):
    if not xs:
        return [y]
    x, *rest = xs
    if y < x:
        return [y] + [x] + rest
    else:
        return [x] + insert(y, rest)

isort = lambda xs: foldl(insert, [], xs)
```

While `fold` reduces a structure to a value, `unfold` generates a structure from a seed [3]:

```
def unfold(pred, func, seed):  
    """Generate a list from a seed value.  
    pred: stopping condition  
    func: produces (element, next_seed)"""  
    result = []  
    while not pred(seed):  
        val, seed = func(seed)  
        result.append(val)  
    return result
```

```
>>> unfold(lambda n: n <= 0,  
...         lambda n: (n, n - 1),  
...         5)  
[5, 4, 3, 2, 1]
```

```
>>> unfold(lambda n: n >= 100,  
...         lambda n: (n, n * 2),  
...         1)  
[1, 2, 4, 8, 16, 32, 64]
```

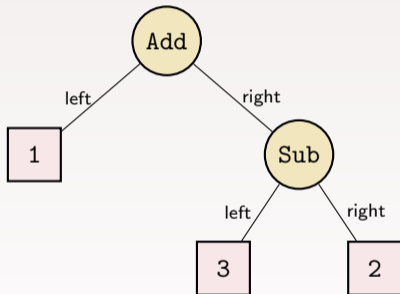
Arithmetic expressions:

```
class Num:
    def __init__(self, value):
        self.value = value

class Add:
    def __init__(self, left, right):
        self.left = left
        self.right = right

class Sub:
    def __init__(self, left, right):
        self.left = left
        self.right = right
```

The expression `Add(Num(1), Sub(Num(3), Num(2)))` as a tree:



`reduce_tree` traverses this tree bottom-up, applying functions at each node.

Traverse the tree recursively and apply the function `func` to combine the results of traversing each subtree:

```
def reduce_tree(fleaf, fadd, fsub, tree):
    if isinstance(tree, Num):
        return fleaf(tree.value)
    elif isinstance(tree, Add):
        return fadd(reduce_tree(fleaf, fadd, fsub, tree.left),
                    reduce_tree(fleaf, fadd, fsub, tree.right))
    elif isinstance(tree, Sub):
        return fsub(reduce_tree(fleaf, fadd, fsub, tree.left),
                    reduce_tree(fleaf, fadd, fsub, tree.right))
```

Using `reduce_tree` to print and evaluate expressions:

```
print_expr = lambda t: reduce_tree(str,  
    lambda x, y: f"({x} + {y})",  
    lambda x, y: f"({x} - {y})",  
    t)
```

```
eval_expr = lambda t: reduce_tree(lambda x: x,  
    lambda x, y: x + y,  
    lambda x, y: x - y,  
    t)
```

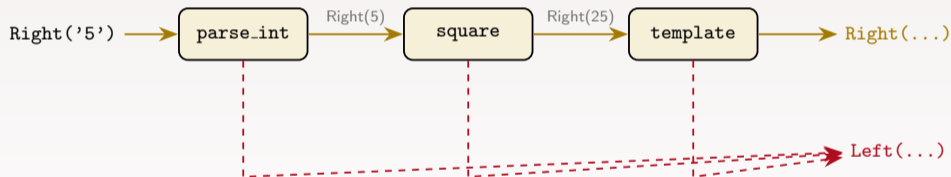
For `tree = Add(Num(1), Sub(Num(3), Num(2)))`:

```
>>> print_expr(tree)  
'(1 + (3 - 2))'  
>>> eval_expr(tree)  
2
```

- ▶ A **Monad** is a design pattern used to represent computations chained together.
- ▶ It consists of two main components (in Haskell notation):
  1. **A type constructor**: It takes a type  $T$  and returns a new type  $M[T]$ .
  2. **Two operations**:
    - ▶ **bind**: `bind :: M[T] -> (T -> M[U]) -> M[U]`
    - ▶ **return (or unit)**: `return :: T -> M[T]`

The bind (or then) operation routes values through a chain of computations:

**Happy path:** Right values pass through each function



**Error path:** a Left bypasses remaining functions

```
from pymonad.either import Left, Right
from pymonad.tools import curry

def parse_int(value):
    try:
        return Right(int(value))
    except ValueError as e:
        return Left(f"Error parsing integer: {e}")

def square(value):
    return value ** 2

def string_template(value):
    return f"FINAL RESULT {value}"

>>> Right('5').then(parse_int).then(square).then(string_template)
Right FINAL RESULT 25

>>> Right('abc').then(parse_int).then(square).then(string_template)
Left Error parsing integer: invalid literal for int()...
```

```
from pymonad.either import Left, Right
from pymonad.tools import curry

@curry(2)
def divide(a, b):
    return (
        Right(a/b)
        if b != 0
        else Left("Error: Division by zero")
    )

result = Right(10).then(divide(2)).then(divide(5))

print(result)
```

```
from pyMonad.io import IO

def read_csv(filename):
    return IO(lambda: open(filename, 'r').read())

def process_csv(content):
    lines = content.split('\n')
    headers = lines[0].split(',')
    return IO(lambda: {'headers': headers,
                       'data': [line.split(',') for line in lines[1:]]})

filename = 'example.csv'
csv_program = read_csv(filename).then(process_csv)

result = csv_program.run()
print("CSV Headers:", result['headers'])
print("CSV Data:", result['data'])
```

```
@curry(2)
def add(x, y):
    return Writer(x + y,
        f"Called function 'add' with arguments {x} and {y}. Result: {x + y}")

@curry(2)
def mul(x, y):
    return Writer(x * y,
        f"Called function 'mul' with arguments {x} and {y}. Result: {x * y}")

logged_arithmetic = (Writer
    .insert(0)
    .then(add(1))
    .then(mul(2)))
```

- [1] John Backus.  
Can programming be liberated from the von Neumann style? a functional style and its algebra of programs.  
*Communications of the ACM*, 21(8):613–641, 1978.
- [2] Harold Abelson and Gerald Jay Sussman.  
*Structure and interpretation of computer programs*.  
MIT Press, 1996.
- [3] Jeremy Gibbons.  
Origami programming.  
2003.