**Definition**

A programming paradigm is a style, or “way,” of programming. Some languages make it easy to write in some paradigms but not others.

A paradigm is a way of doing something (like programming), not a concrete thing (like a language). Now, it’s true that if a programming language L happens to make a particular programming paradigm P easy to express, then we often say “L is a P language” (e.g. “Haskell is a functional programming language”) but that does not mean there is any such thing as a “functional language paradigm”.

**Some Common Paradigms**

* Imperative: Programming with an explicit sequence of commands that update state.
* Declarative: Programming by specifying the result you want, not how to get it.
* Structured: Programming with clean, goto-free, nested control structures.
* Procedural: Imperative programming with procedure calls.
* Functional (Applicative): Programming with function calls that avoid any global state.
* Function-Level (Combinator): Programming with no variables at all.
* Object-Oriented: Programming by defining objects that send messages to each other. Objects have their own internal (encapsulated) state and public interfaces. Object orientation can be:
* Class-based: Objects get state and behavior based on membership in a class.
* Prototype-based: Objects get behavior from a prototype object.
* Event-Driven: Programming with emitters and listeners of asynchronous actions.
* Flow-Driven: Programming processes communicating with each other over predefined channels.
* Logic (Rule-based): Programming by specifying a set of facts and rules. An engine infers the answers to questions.
* Constraint: Programming by specifying a set of constraints. An engine finds the values that meet the constraints.
* Aspect-Oriented: Programming cross-cutting concerns applied transparently.
* Reflective: Programming by manipulating the program elements themselves.
* Array: Programming with powerful array operators that usually make loops unnecessary.

Paradigms are not meant to be mutually exclusive; a single program can feature multiple paradigms.

**Structured Programming**

**Structured programming** is a kind of imperative programming where control flow is defined by nested loops, conditionals, and subroutines, rather than via gotos. Variables are generally local to blocks (have lexical scope).

result = [];  
for i = 0; i < length(people); i++ {  
    p = people[i];  
    if length(p.name)) > 5 {  
        addToList(result, toUpper(p.name));  
    }  
}  
return sort(result);

Early languages emphasizing structured programming: Algol 60, PL/I, Algol 68, Pascal, C, Ada 83, Modula, Modula-2. Structured programming as a discipline is sometimes though to have been started by a famous letter by Edsger Dijkstra entitled [Go to Statement Considered Harmful](http://david.tribble.com/text/goto.html).

**Functional Programming**

In **functional programming**, control flow is expressed by combining function calls, rather than by assigning values to variables:

sort(  
  fix(λf. λp.  
    if(equals(p, emptylist),  
      emptylist,  
      if(greater(length(name(head(p))), 5),  
        append(to\_upper(name(head(p))), f(tail(p))),  
        f(tail(people)))))(people))

Yikes! We’ll describe that later. For now, be thankful there’s usually syntactic sugar:

let  
    fun uppercasedLongNames [] = []  
      | uppercasedLongNames (p :: ps) =  
          if length(name p) > 5 then (to\_upper(name p))::(uppercasedLongNames ps)  
          else (uppercasedLongNames ps)  
in  
    sort(uppercasedLongNames(people))

Huh? That still isn’t very pretty. Why do people like this stuff? Well the real power of this paradigm comes from passing functions to functions (and returning functions from functions).

sort(  
    filter(λs. length s > 5,  
        map(λp. to\_upper(name p),  
            people)))

We can do better by using the cool |> operator. Here x |> f just means f(x). The operator has very low precedence so you can read things left-to-right:

people |> map (λp. to\_upper (name p)) |> filter (λs. length s > 5) |> sort

Let’s keep going! Notice that you wouldn’t write map(λx. square(x)), right? You would write map(square). We can do something similar above, but we have to use function composition, you know, (f o g)x is f(g(x)), so:

people |> map (to\_upper o name) |> filter (λs. length s > 5) |> sort

Here are three things to read to get the gist of functional programming:

* [Kris Jenkins’ article](http://blog.jenkster.com/2015/12/what-is-functional-programming.html)
* [Chris Done’s two-part article](https://www.fpcomplete.com/blog/2017/04/pure-functional-programming)
* [Joel Spolsky’s article on map and reduce](http://www.joelonsoftware.com/items/2006/08/01.html)

With functional programming:

* There are no commands, only side-effect free expressions
* Code is much shorter, less error-prone, and much easier to prove correct
* There is more inherent parallelism, so good compilers can produce faster code

Some people like to say:

* *Functional, or Applicative, programming* is programming without assignment statements: one just applies functions to arguments. Examples: Scheme, Haskell, Miranda, ML.
* *Function-level programming* does away with the variables; one combines functions with **functionals**, a.k.a. **combinators**. Examples: FP, FL, J.

***Exercise****: Write the above example in Miranda, ML, and J.*

***Exercise****: Research the following programming styles and state how they are similar and how they are different from each other: (a) Stack-based, (b) Concatenative, (c) Point-free, (d) Tacit.*

Many languages have a neat little thing called **comprehensions** that combine map and filter.

sorted(p.name.upper() for p in people if len(p.name) > 5)

**Object Oriented Programming**

OOP is based on the sending of **messages** to objects. Objects respond to messages by performing operations, generally called **methods**. Messages can have arguments. A society of objects, each with their own local memory and own set of operations has a different feel than the monolithic processor and single shared memory feel of non object oriented languages.

One of the more visible aspects of the more pure-ish OO languages is that conditionals and loops become messages themselves, whose arguments are often blocks of executable code. In a Smalltalk-like syntax:

result := List new.  
people each: [:p |  
  p name length greaterThan: 5 ifTrue: [result add (p name upper)]  
]  
result sort.  
^result

This can be shortened to:

^people filter: [:p | p name length greaterThan: 5] map: [:p | p name upper] sort

Many popular languages that call themselves OO languages (e.g., Java, C++), really just take some elements of OOP and mix them in to imperative-looking code. In the following, we can see that length and toUpper are methods rather than top-level functions, but the for and if are back to being control structures:

result = []  
for p in people {  
    if p.name.length > 5 {  
        result.add(p.name.toUpper);  
    }  
}  
return result.sort;

The first object oriented language was Simula-67; Smalltalk followed soon after as the first “pure” object-oriented language. Many languages designed from the 1980s to the present have labeled themselves object-oriented, notably C++, CLOS (object system of Common Lisp), Eiffel, Modula-3, Ada 95, Java, C#, Ruby.