Imperative: how to solve

- Procedural
- Object-Oriented
Programming Paradigms

**Imperative**: how to solve
- Procedural
- Object-Oriented

**Declarative**: what to solve
- Functional
- Logical
Imperative Languages

- The design of the imperative languages is based directly on the von Neumann architecture.
Imperative Languages

- The design of the imperative languages is based directly on the von Neumann architecture.

- Efficiency is the primary concern, rather than the suitability of the language for software development.
The design of the functional languages is based on **mathematical functions**.

A solid theoretical basis that is also closer to the user, but relatively unconcerned with the architecture of the machines on which programs will run.
A mathematical function is a mapping of members of one set, called the domain set, to another set, called the range set.
A mathematical function is a mapping of members of one set, called the domain set, to another set, called the range set.

- A lambda expression specifies the parameter(s) and the mapping of a function in the following form:
  \[ \lambda(x) \ x \times x \times x \] for the function \( \text{cube}(x) = x \times x \times x \)
Lambda Expressions

- Lambda expressions describe nameless functions.
- Lambda expressions are applied to parameter(s) by placing the parameter(s) after the expression.
  e.g., \((\lambda(x) \cdot x \times x \times x)(2)\) which evaluates to 8
A higher-order function, or **functional form**, is one that either takes functions as parameters or yields a function as its result, or both.
Functional Programming Concepts

Functional languages such as Lisp, Scheme, FP, ML, Miranda, and Haskell are an attempt to realize Church’s lambda calculus in practical form as a programming language.

The key idea:
do everything by composing functions
  - no mutable state
  - no side effects
Function Composition

A functional form that takes two functions as parameters and yields a function whose value is the first actual parameter function applied to the application of the second.
Function Composition

- A functional form that takes two functions as parameters and yields a function whose value is the first actual parameter function applied to the application of the second.

Form: \( h \equiv f \circ g \)
A functional form that takes two functions as parameters and yields a function whose value is the first actual parameter function applied to the application of the second.

Form: \( h \equiv f \circ g \) which means \( h(x) \equiv f(g(x)) \)

For \( f(x) \equiv x + 2 \) and \( g(x) \equiv 3 \times x \),
\( h \equiv f \circ g \) yields \( (3 \times x) + 2 \)
Apply-to-all

- A functional form that takes a single function as a parameter and yields a list of values obtained by applying the given function to each element of a list of parameters.
Apply-to-all

- A functional form that takes a single function as a parameter and yields a list of values obtained by applying the given function to each element of a list of parameters.

Form: $\alpha$

\[
\alpha(h(x)) \equiv x \times x
\]

$\alpha(h, (2, 3, 4))$ yields $(4, 9, 16)$
Apply-to-all

- A functional form that takes a single function as a parameter and yields a list of values obtained by applying the given function to each element of a list of parameters.

Form: $\alpha$
For $h(x) \equiv x \times x$
$\alpha(h, (2, 3, 4))$ yields $(4, 9, 16)$
The objective of the design of a FPL is to mimic mathematical functions to the greatest extent possible.
The objective of the design of a FPL is to mimic mathematical functions to the greatest extent possible.

The basic process of computation is fundamentally different in a FPL than in an imperative language.

- In an imperative language, operations are done and the results are stored in variables for later use.
- Management of variables is a constant concern and source of complexity for imperative programming.
The objective of the design of a FPL is to mimic mathematical functions to the greatest extent possible.

The basic process of computation is fundamentally different in a FPL than in an imperative language.
- In an imperative language, operations are done and the results are stored in variables for later use.
- Management of variables is a constant concern and source of complexity for imperative programming.

In an FPL, variables are not necessary, as is the case in mathematics.
Referential Transparency
In an FPL, the evaluation of a function always produces the same result given the same parameters.
Fundamentals of Functional Programming Languages

Referential Transparency
In an FPL, the evaluation of a function always produces the same result given the same parameters.

Tail Recursion
Writing recursive functions that can be automatically converted to iteration.
Haskell

- Statically scoped, strongly typed, type inferencing, pattern matching.
- Purely functional (e.g., no variables, no assignment statements, and no side effects of any kind).
Function Definitions with Different Parameter Ranges

fact n
  | n == 0 = 1
  | n > 0 = n * fact(n { 1)

sub n
  | n < 10 = 0
  | n > 100 = 2
  | otherwise = 1

square x = x * x

Works for any numeric type of x
List notation: Put elements in brackets
e.g., directions = ["north", "south", "east", "west"]
Lists

- List notation: Put elements in brackets
e.g., directions = ["north", "south", "east", "west"]

- Length: length
e.g., length directions is 4
List notation: Put elements in brackets
e.g., directions = ["north", "south", "east", "west"]

Length: length
e.g., length directions is 4

Arithmetic series with the .. operator
e.g., [2, 4 .. 10] is [2, 4, 6, 8, 10]
Lists

- **List notation**: Put elements in brackets
e.g., `directions = ["north", "south", "east", "west"]`

- **Length**: `length`
e.g., `length directions` is 4

- **Arithmetic series with the .. operator**
e.g., `[2, 4 .. 10]` is `[2, 4, 6, 8, 10]`

- **Concatenation is with ++**
e.g., `[1, 3] ++ [5, 7]` results in `[1, 3, 5, 7]`
Lists

- List notation: Put elements in brackets
e.g., directions = ["north", "south", "east", "west"]

- Length: length
e.g., length directions is 4

- Arithmetic series with the .. operator
e.g., [2, 4 .. 10] is [2, 4, 6, 8, 10]

- Concatenation is with ++
e.g., [1, 3] ++ [5, 7] results in [1, 3, 5, 7]

- Construct and take apart lists via the colon operator
e.g., 1:[3, 5, 7] results in [1, 3, 5, 7]
List Comprehension

- Set notation
- List of the squares of the first 20 positive integers: $[n \times n| n \leftarrow [1..20]]$
List Comprehension

- Set notation
- List of the squares of the first 20 positive integers: \([n \times n | n \leftarrow [1..20]]\)
- All of the factors of its given parameter:
  \(factors \ n = [i | i \leftarrow [1..n \ div \ 2], n \ mod \ i == 0]\)