Functional Programming

Ionut G. Stan - OpenAgile 2010

Functional Programming

- what
- why
- how

Functional Programming

- what
- why
- how

What is FP

- a programming style

What is FP

- a programming style
- conceptually derived from lambda calculus (1930s)

What is FP

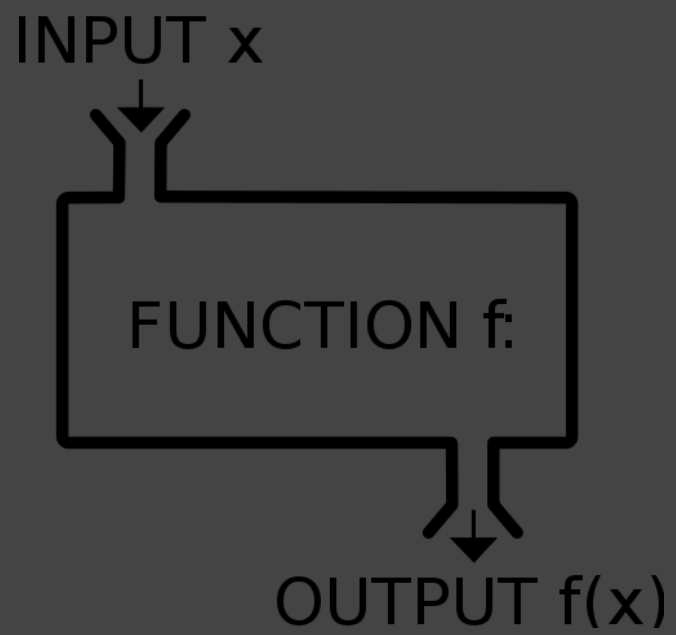
- a programming style
- conceptually derived from lambda calculus (1930s)
- not procedural programming

What is FP

- a programming style
- conceptually derived from lambda calculus (1930s)
- not procedural programming
- functions as in mathematical functions

What is FP

- a programming style
- conceptually derived from lambda calculus (1930s)
- not procedural programming
- functions as in mathematical functions
- math function: input completely determines the output



Imperative Programming

- programming: telling a computer what to do

Imperative Programming

- programming: telling a computer what to do
- imperative languages are leaky abstractions

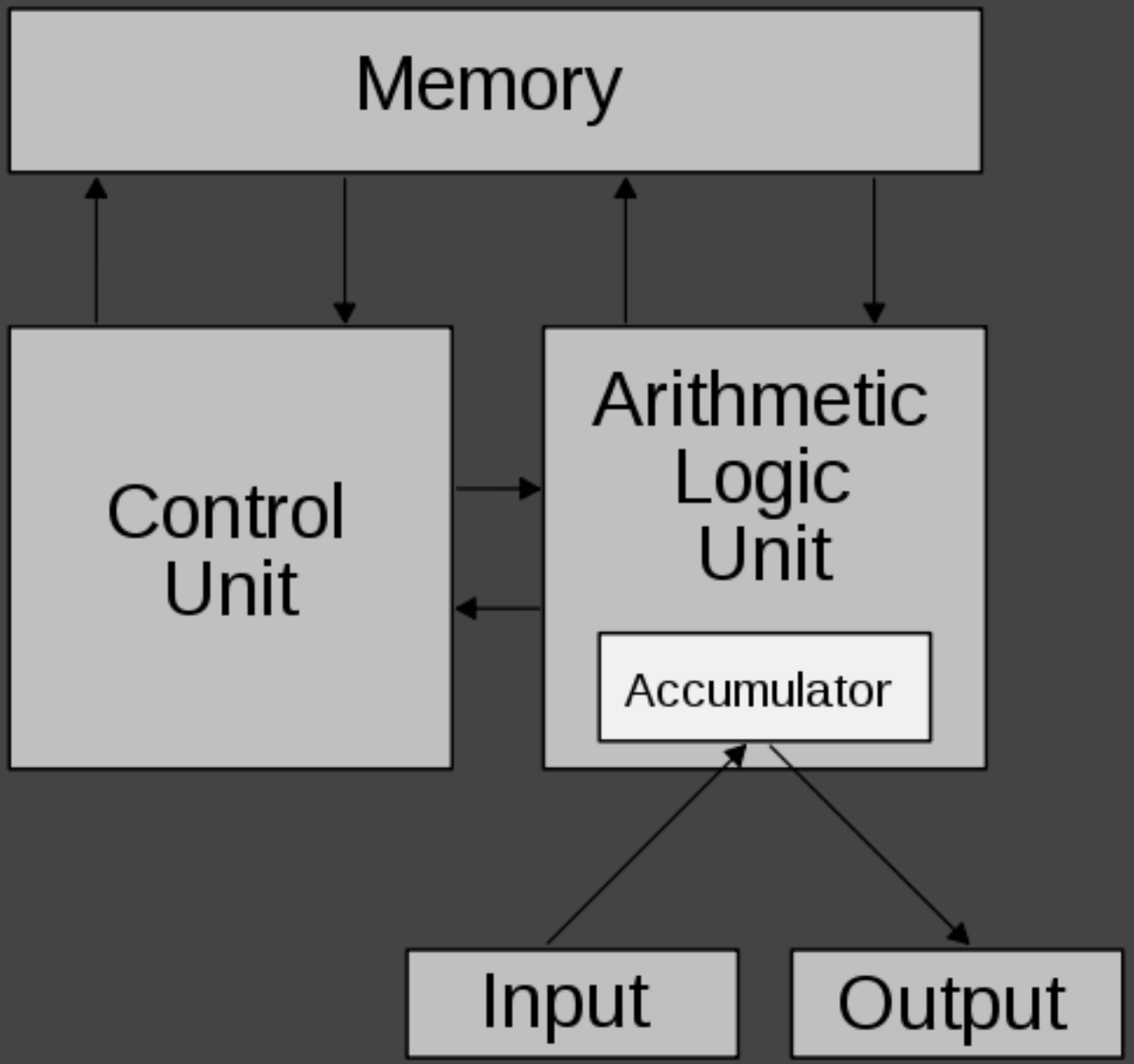
Imperative Programming

- programming: telling a computer what to do
- imperative languages are leaky abstractions
- also called von Neumann languages

Imperative Programming

- programming: telling a computer what to do
- imperative languages are leaky abstractions
- also called von Neumann languages
- von Neumann architecture is about modifying the state of the computer

von Neumann architecture



Imperative Programming

- programming: telling a computer what to do
- imperative languages are leaky abstractions
- also called von Neumann languages
- von Neumann architecture is about modifying the state of the computer
- computation model in imperative languages reflects von Neumann architecture

Imperative Programming

- programming: telling a computer what to do
- imperative languages are leaky abstractions
- also called von Neumann languages
- von Neumann architecture is about modifying the state of the computer
- computation model in imperative languages reflects von Neumann architecture
- imperative programming: what and how to do

What is FP

- a programming style
- conceptually derived from lambda calculus (1930s)
- not procedural programming
- functions as in mathematical functions
- trying to plug the abstraction leak

What is FP

- a programming style
- conceptually derived from lambda calculus (1930s)
- not procedural programming
- functions as in mathematical functions
- trying to plug the abstraction leak
- tell the computer what to do, not how

What is FP

- a programming style
- conceptually derived from lambda calculus (1930s)
- not procedural programming
- functions as in mathematical functions
- trying to plug the abstraction leak
- tell the computer what to do, not how
- mutations not allowed (no variables, just identifiers)

What is FP

- a programming style
- conceptually derived from lambda calculus (1930s)
- not procedural programming
- functions as in mathematical functions
- trying to plug the abstraction leak
- tell the computer what to do, not how
- mutations not allowed (no variables, just identifiers)
- no statements, just expressions (if/then/else is expression)

What is FP

- a programming style
- conceptually derived from lambda calculus (1930s)
- not procedural programming
- functions as in mathematical functions
- trying to plug the abstraction leak
- tell the computer what to do, not how
- mutations not allowed (no variables, just identifiers)
- no statements, just expressions (if/then/else is expression)
- functions are deterministic and side-effect free

What is FP

- a programming style
- conceptually derived from lambda calculus (1930s)
- not procedural programming
- functions as in mathematical functions
- trying to plug the abstraction leak
- tell the computer what to do, not how
- mutations not allowed (no variables, just identifiers)
- no statements, just expressions (if/then/else is expression)
- functions are deterministic and side-effect free
- functions are all we need to model computation

What is FP

- a programming style
- conceptually derived from lambda calculus (1930s)
- not procedural programming
- functions as in mathematical functions
- trying to plug the abstraction leak
- tell the computer what to do, not how
- mutations not allowed (no variables, just identifiers)
- no statements, just expressions (if/then/else is expression)
- functions are deterministic and side-effect free
- functions are all we need to model computation
- execution order is not guaranteed

Functional Programming

- what
- why
- how

Why FP

- easier to reason about programs

Why FP

- easier to reason about programs
- heisenbugs

Why FP

- easier to reason about programs

- heisenbugs
- race conditions

Why FP

- easier to reason about programs

- heisenbugs
- race conditions
- off by one errors

Why FP

- easier to reason about programs

- heisenbugs
- race conditions
- off by one errors
- objects trashing another object's internal state

Why FP

- easier to reason about programs

```
// does this program terminate?
var n = 0;

while (--n) {
    n++;
}
```

Why FP

- easier to reason about programs
- easier to parallelize

Why FP

- easier to reason about programs
- easier to parallelize
- program correctness proving

Why FP

- easier to reason about programs
- easier to parallelize
- program correctness proving
- composability results in greater and easier reuse

Just for fun - reuse in OO languages

"You wanted a banana but what you got was a gorilla holding the banana and the entire jungle."

Joe Armstrong, creator of Erlang

```
Jungle jungle = new Jungle();
Banana banana = jungle.getGorilla().getBanana();
```

Why FP

- easier to reason about programs
- easier to parallelize
- program correctness proving
- composability results in greater and easier reuse
- try out new perspectives

Functional Programming

- what
- why
- how

Functional Programming

- what
- why
- how (even in imperative languages)

How to FP

- avoid side-effects/mutation as much as possible

How to FP

- avoid side-effects/mutation as much as possible
- at least keep them as private as possible in OO

How to FP

- avoid side-effects/mutation as much as possible
- at least keep them as private as possible in OO
- treat variables as immutable (constants/final)

How to FP

- avoid side-effects/mutation as much as possible
- at least keep them as private as possible in OO
- treat variables as immutable (constants/final)
- return values (output) based on params (input) only

How to FP

- avoid side-effects/mutation as much as possible
- at least keep them as private as possible in OO
- treat variables as immutable (constants/final)
- return values (output) based on params (input) only
- play with a functional language

Functional Programming

- what
- why
- how
- example

Example

```
// imperative JavaScript
var sum = function (list) {
    var length = list.length;
    var total  = 0;

    while (length--) {
        total += list[length];
    }

    return total;
};
```

Example

```
// functional JavaScript
var sum = function (list) {
    return list.length === 0                    // base case
          ? 0
          : list[0] + sum( list.slice(1) ); // recursive rule
};
```

Example

```haskell
-- Haskell using pattern matching
sum :: [Int] -> Int
sum []           = 0                    -- base case
sum (first:rest) = first + sum rest -- recursive rule
```

"Conventional programming languages are growing ever more enormous, but not stronger. Inherent defects at the most basic level cause them to be both fat and weak: their primitive word-at-a-time style of programming inherited from their common ancestor -- the von Neumann computer, their close coupling of semantics to state transitions, their division of programming into a world of expressions and a world of statements, their inability to effectively use powerful combining forms for building new programs from existing ones, and their lack of useful mathematical properties for reasoning about programs."

John Backus, known for Fortran, Algol and BNF

Thank You

igstan.ro | ionut.g.stan@gmail.com | @igstan

Questions?

igstan.ro | ionut.g.stan@gmail.com | @igstan