Imperative and Functional Programming Paradigm

Imperative Programming

What Makes a Language Imperative?

- Programs written in imperative programming languages consist of
  - A program state
  - Instructions that change the program state
- Program instructions are "imperative" in the grammatical sense of imperative verbs that express a command

Concept of State

- In imperative programming, a name may be assigned to a value and later reassigned to another value.
- The collection of names and the associated values and the location of control in the program constitute the state.
- The state is a logical model of storage which is an association between memory locations and values.
- A program in execution generates a sequence of states.
- The transition from one state to the next is determined by assignment operations and sequencing commands

$$S_0 - O_0 \rightarrow S_1 - O_1 \rightarrow S_2 \ldots S_{n-1} - O_{n-1} \rightarrow S_n$$

Defining Characteristics of Imperative Languages

- Statements are commands
  - Command order is critical to correct execution;
  - Programmers control all aspects: algorithm specification, memory management, variable declarations, etc.
- They work by modifying program state
- Statements reflect machine language instructions.

Von Neumann Machines and Imperative Programming

- Commands in an imperative language are similar to the native machine instructions of traditional computer hardware – the von Neumann-Eckley model.
- John von Neumann: first person to document the basic concepts of stored program computers.
- Von Neumann was a famous Hungarian mathematician; came to US in 1930s & became interested in computers while participating in the development of the hydrogen bomb.

The "von Neumann" Computer

- A *memory* unit: able to store both data and instructions
  - Random access
  - Internally, data and instructions are stored in the same address space & and are indistinguishable
- A *calculating unit* (the ALU)
- A *control unit,* (the CPU)
  Stored program → an instruction set
- Duality of instructions and data → programs can be self modifying
- Von Neumann outlined this structure in a document known as the "First Draft of a Report on the EDVAC" June, 1945

History of Imperative Languages

- First imperative languages: assembly languages
- 1954-1955: Fortran (FORmula TRANslator) John Backus developed for IBM 704
- Late 1950's: Algol (ALGOrithmic Language)
- 1958: Cobol (COmmon Business Oriented Language) Developed by a government committee; Grace Hopper very influential.

Turing Completeness

- A language is Turing complete if it can be used to implement any algorithm.
- Central to the study of computability
- Alan Turing: A British mathematician, logician, and eventually computer scientist.

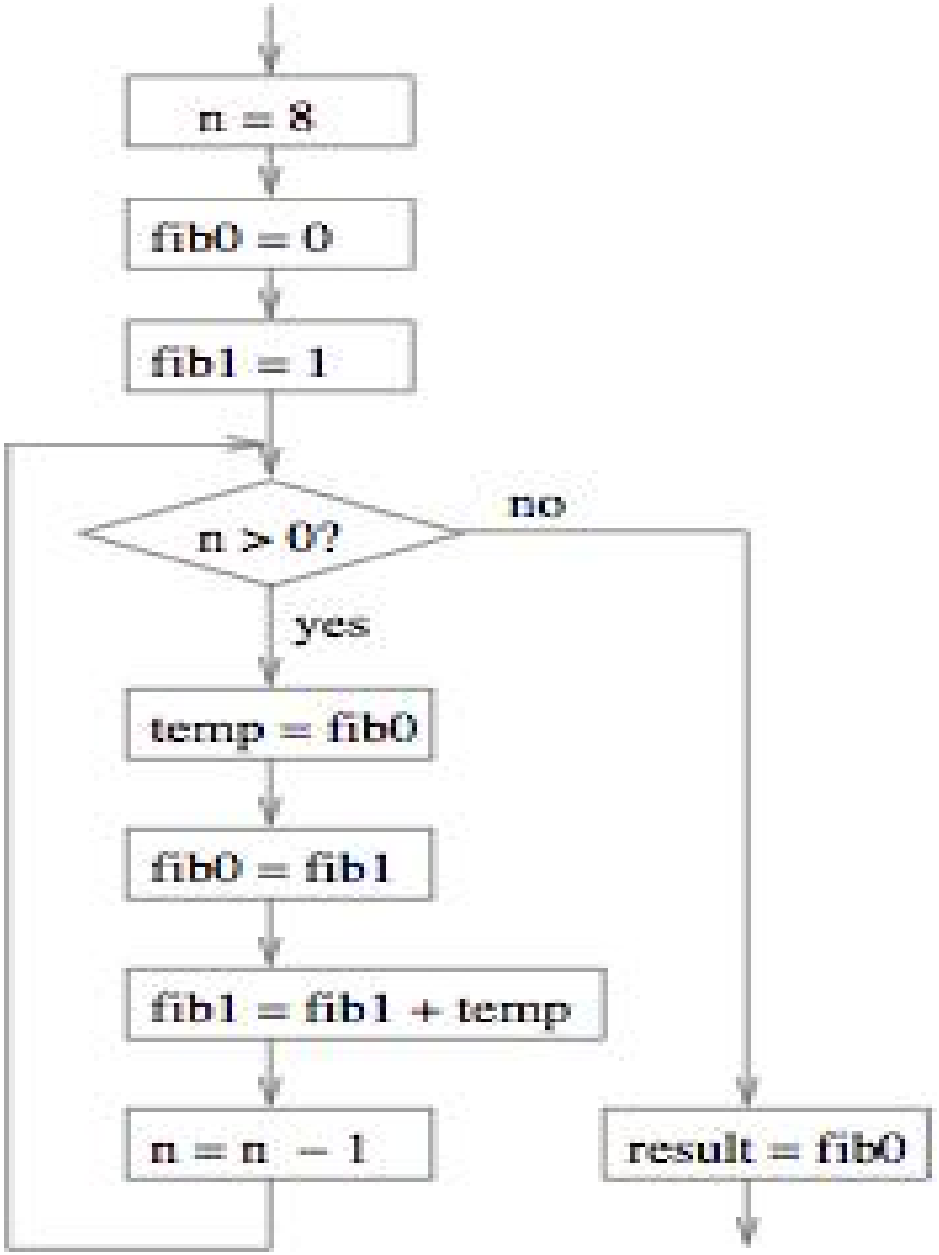
Imperative Programming

- Imperative languages are Turing complete if they support integers, basic arithmetic operators, assignment, sequencing, looping and branching.

- Modern imperative languages generally also include features such as
  - Expressions and assignment
  - Control structures (loops, decisions)
  - I/O commands
  - Procedures and functions
  - Error and exception handling
  - Library support for data structures

Flowchart

- Used to model imperative programs
- Based on the three control statements (sequential, selection and iteration) that are essential to have Turing machine capability
- Precursor of UML and other modern techniques
- Originated to describe process flow in general



Imperative versus Declarative Languages

- Imperative programming languages (Java, C/C++)
  - specify a sequence of operations for the computer to execute.
- Declarative languages (SQL, Haskell, Prolog)
  - describe the solution space
  - provide knowledge required to get there
  - don't describe steps needed to get there
- Functional languages and logic languages are declarative.

Procedural Abstraction

- Nicholas Wirth described [imperative] programs as being "algorithms plus data structures".
- Algorithms become programs through the process of procedural abstraction and stepwise refinement.
- Libraries of reusable functions support the process (functions = procedures)
- Imperative programming + procedures = procedural programming.

Procedural Abstraction

- Procedural abstraction allows the programmer to be concerned mainly with the interface between the function (procedure) and what it computes, ignoring the details of how the computation is accomplished.

- Abstraction allows us to think about *what* is being done, not *how* it is implemented.

Stepwise Refinement

- Stepwise refinement (also called functional decomposition) uses procedural abstraction by developing an algorithm from its most general form [the abstraction] into a specific implementation.

- Programmers start with a description of what the program should do, including I/O, and repeatedly break the problem into smaller parts, until the sub-problems can be expressed in terms of the primitive states and data types in the language.

Structured Programming

- A disciplined approach to imperative program design.
- Uses procedural abstraction and top-down design to identify program components (also called modules or structures)
- Program structures combined in a limited number of ways, so understanding how each structure works means you understand how the program works
- Program control flow is based on decisions, sequences, loops, but…
- Does not use goto statements
- Modules are developed, tested separately and then integrated into the whole program.

Functional Programming

Introduction

- Functional programming is a programming paradigm that treats computation as the evaluation of mathematical functions and avoids state and mutable data

  In other words

- functional programming promotes code with no side effects, no change of value in variables. It opposes to imperative programming, which emphasizes change of state

What this means?

- No mutable data (no side effect)
- No state (no implicit, hidden state)

Once assigned (value binding), a variable (a symbol) does not change its value.

Overview

- Functional programming mirrors *mathematical functions*:
  - domain = input, range = output
  - A computation maps inputs to outputs

- *Variables* are mathematical *symbols*; not associated with memory locations.

  (Compare to imperative programming)

Overview

- Pure functional programming is *state-free*: no assignment statements.
- Programs in pure functional languages consist of composite functions; output of each function is input to another.

Today, most functional languages have some imperative statements.

Example

Summing the integers 1 to 10 in imperative language C:

```
int total = 0, i;
for (i = 1; i <= 10; ++i){
    total = total+i;
}
```

Values change for both `total` and `i` during program execution

Summing integers 1 to 10 in a pure functional language

```
sum (m, n) : if (m > n) 0
     else m + sum (m+1, n)

sum (1, 10) // main function
```

No side effect => No assignments to variables!

All state is bad?

No, **hidden**, **implicit** state is bad.

Functional programming do not eliminate state, it just make it **visible** and **explicit** (at least when programmers want it to be).

- Functions are *pure* functions in the mathematical sense: their output depend only in their inputs, there is not "environment".
- Same result returned by functions called with the same inputs.

Background: Functions

- A function *f* from a set X (the domain) to a set Y (the range) maps each element x in X to a unique element y in Y.

For example, $f(x) = x^2$ maps the set of real numbers into the set of positive real numbers (ignoring imaginary numbers).

  - i.e., the domain X is the set of all real numbers.

Background: Functional Composition

- If f is a function from X to Y and g is a function from Y to Z, then $(g \circ f)(x)$ is a function from X to Z defined as

  $(g \circ f)(x) = g(f(x))$, for all x in X

  Simply put, it means to replace the x in $g(x)$ with $f(x)$.

Example:

$f(x) = x^2 + x$

$g(x) = 2x + 1$

$g \circ f = 2(x^2 + x) + 1$

Functions and the Lambda Calculus

Lambda calculus: The "assembly language" of functional programming

Typical mathematical function: $Square(n) = n^2$

funcName(args) = func-definition: an expression

Square : **R** $\rightarrow$ **R** maps from reals to reals

domain and range are the set of real numbers

A function is *total* if it is defined for all values of its domain. Otherwise, it is *partial*. e.g., *Square* is total.

Referential Transparency

- A function's result depends only upon the values of its arguments and not on any previous computation or the order of evaluation for its arguments.

  Or

- If we call a function with the same parameters, we know for sure the output will be the same (there is no state anywhere that would change it).

Referential Transparency

This is in contrast to the view of functions in imperative languages, where function values are based on arguments, order of evaluation, and can also have side effects (change state).

- Pure functional languages have no state in the sense of state in imperative programs & so no side effects.

Referential Transparency

In imperative programming a statement such as

```
x = x + y;
```

means "add the value of in memory cell x to the value in memory cell y and replace the value of x by the value of the function"

- The name x is used in two different ways here; *lhs* (a memory address) versus *rhs* (a value)

- Functional programs accept and return data, but have no side effects and do not have the concept of a variable as a memory address

Advantages enabled by Referential Transparency

- Memoization
  - Cache results for previous function calls.
- Idempotence
  - Same results regardless how many times you call a function.
- Modularization
  - We have no state that pervades the whole code, so we build our project with small, black boxes that we tie together, so it promotes bottom-up programming.

Advantages enabled by Referential Transparency

- Ease of debugging
  - Functions are isolated, they only depend on their input and their output, so they are very easy to debug.
- Parallelization
  - Functions calls are independent.
  - We can parallelize in different CPUs/process/computers/…

```
result = func1(a, b) + func2(a, c)
```

We can execute *func1* and *func2* in paralell because '**a**' won't be modified.

- Concurrence
  - With no shared data, concurrence gets a lot simpler:

No semaphores, No monitors, No locks, No race-conditions, No dead-locks.