15-110 Principles of Computing – S19

Lecture 2:

Computer Architecture
Programming Languages

Teacher:
Gianni A. Di Caro

جامعة كارنيجي ميلون في قطر
Carnegie Mellon University Qatar

Road map

- Ways to tell a computer what to do
- General architecture and way of functioning of a computer
- Computer languages vs. natural languages
- Compilers vs. interpreters

What does a computer do?

✓ **Calculations** (> billions per second) + **Remember and access results** (memory, TB of storage)

- What kind of calculations?

1. Built-in to some *restricted language* that the machine understands

2. Those that the **programmer defines**, based on a more flexible language (e.g., Python!)

Types of knowledge to specify what to do

✓ In order to do what the programmer wants them to do, computers need <u>precise</u> **knowledge** **(**about the actions to execute, the task to perform, the environment the computer interact with)

- There are two main types of knowledge that can be used to program a computer:

1. **Declarative knowledge**
   - Involves knowing WHAT is the case
   - Collection of statements of **facts** (including **goals**) and **truth**
     - Square root y of a number x is such that $y \cdot y = x$
     - This pen is red; this is a mac notebook
     - (I need) a table for two
   - WHAT is true
   - Conscious, can be verbalized

2. **Imperative knowledge**
   - Involves knowing HOW to do something
   - It is a recipe, a sequence of steps / commands (*imperative*), possibly based on declarative knowledge
     - How to drive a car
     - How to make a strawberry cake
     - Based on declarative knowledge of the square root, an iterative procedure for finding $\sqrt{x}$
   - HOW to do things

Examples: Declarative vs. Imperative approach

Declarative approach

Imperative approach

At the front desk of the restaurant:

✓ Table for two, please

✓ I see that table for two located under the tree, on right side, is empty. My wife and I are going to walk over there over the free path and sit down.

I need to go from CMU-Q to QF:

Address is:
Qatar Foundation
Al-Rayyan, Qatar

Types of knowledge and programming paradigms: Declarative

Declarative knowledge, WHAT is true ➡ **Declarative programming languages**

- ✓ Allow to specify *what* it is **known** and *what* are the **goals,** *what* to compute
- ✓ No need to specify the implementation, or *how* the goals are achieved
- ✓ Express the **logic of a computation** without describing its *control flow*

15-150 (Prof. Giselle)

- Examples: SQL, HTML, Prolog, Lisp, Haskell, ...
- SQL: `SELECT * FROM Doctors WHERE Specialty='Dentist';`
- HTML: <img src="cake.jpg"> This combination of text and image looks bad on most browsers.

Types of knowledge and programming paradigms: Declarative

Prolog:

Knowledge base

```
woman(mary).
man(joe).
happy(mary).
listens2Music(joe).
listens2Music(mary):- happy(mary).
playsAirGuitar(joe):- listens2Music(joe).
playsAirGuitar(mary):- listens2Music(mary).
```

Facts

Rules

Rule: `A :- B` A is implied by B: if B is true then also A is true

Query:

```
?- woman(mary).
?- woman(giselle).
?- playsAirGuitar(joe).
```

Types of knowledge and programming paradigms: Imperative

Imperative knowledge, HOW to do things ➡ **Imperative programming languages**

- ✓ Allow / need to tell the machine what we want to happen, *step by step*
- ✓ In *natural languages* the imperative mood expresses *commands*: *RUN!*
- ✓ In *programming languages*, an imperative program consists of a list of ***commands* for the computer to perform**
- ✓ Imperative programming describes how step by step a program operates to achieve its goals
- ✓ NOTE: We might use *declarative knowledge* to build the algorithm!

- Examples: C, C++, Python, C#, Java, Visual Basic, Fortran, …

We will focus on algorithms expressed using imperative knowledge and implemented using the imperative programming paradigm (as specifically encoded in python language)

Types of knowledge and programming paradigms: Imperative

What is the general "recipe" of imperative knowledge for problem solving?

- ✓ sequence of (simple) **steps**
- ✓ **flow control**: specifies when each step is being (conditionally, if-then-else) executed or repeated (while, until, for)
- ✓ means of determining when to **stop**

Algorithm

Types of knowledge and programming paradigms: Imperative

Good reasons to focus on the imperative approach (and on python):

- **Most used approach**, coming in a number of different flavors
- Quite *intuitive* as a first approach to computing
- The **hardware** architecture of almost all computers is designed after the imperative approach
- Nearly all computer hardware is designed to execute **machine code**, which is native to the computer and is written in the imperative style

Language Rank	Types	Spectrum Ranking
1. Python		100.0
2. C		99.7
3. Java		99.4
4. C++		97.2
5. C#		88.6
6. R		88.1
7. JavaScript		85.5
8. PHP		81.4
9. Go		76.1
10. Swift		75.3
11. Arduino		73.0
12. Ruby		72.4
13. Assembly		72.1
14. Scala		68.3
15. Matlab		68.0
16. HTML		67.0
17. Shell		66.3
18. Perl		57.6
19. Visual Basic		55.4
20. Cuda		53.9

https://adtmag.com/articles/2017/07/24/ieee-spectrum-ranking.aspx

Let's get a closer look into the *machine*

How to capture an algorithm in a process operated by a *machine*?

Fixed program computer:
machines that perform one or a limited set of predefined actions

Stored program computer:
machine that stores and execute custom instructions ⟶ machines that can perform a large variety of tasks

What is the basic architecture of a (stored program) computer, that enables it to be a versatile, "programmable" machine?

General computer architecture

Keyboard
Mouse
Joypad
Disk
....

Disk
External memory
Monitor
....

Internal memory
RAM: volatile
ROM: permanent

General computer architecture

Kind of analogy: use of multi-belt system in warehousing to move items across compartments, issue requests, respond to requests, coordinate job, …

Steps of execution: loading the computer program

Boot process:
Operating System
(Windows, Mac OSX, Linux, Android)

Storage memory:
OS programs

Internal memory
ROM: permanent

Steps of execution: loading the computer program

Storage memory:
User programs

Program encoding:

- Sequence of binary digits (01) that the machine can understand
- Sequence of instructions

Internal memory
RAM: volatile

Steps of execution: Fetching instructions

CPU:
Clock-based system

GHz(s)!

0000 1001 1100 0110 1010 1111 0101 1000
1010 1111 0101 1000 0000 1001 1100 0110
1100 0110 1010 1111 0101 1000 0000 1001
0101 1000 0000 1001 1100 0110 1010 1111

- Control Unit:
  - The (first) next instruction if fetched from memory and stored in the IR register
  - The PC register keeps track of the instruction to be executed next

Steps of execution: Performing operations

ALU:
Where operations are performed

What operations?

- Addition
- Comparison (equal, different)
- Moving data
- … (optional)

- ALU:
  - Gets the instructions from the CU
  - Gets data from the memory
  - Store data in the memory

Steps of execution: Repeating and branching

CU:
Keep iterating the process
Perform branching if necessary

- CU:
  - E.g., a test operation has returned False, then the next operation is not the next in the sequence
  - PC gets updated accordingly

Programming languages

- A ***program*** is just a <u>sequence of instructions</u> telling the computer <u>what to do</u>: an encoding of an *algorithm for solving a problem*

Easy!

Programming languages

- Program instructions need to be expressed in a language that computers can understand
  - We refer to this kind of a language as a ***programming language***
  - Python, Java, C, C++ are notable examples of *imperative programming languages*

```
summation = 0
count = 0
while count < 5:
    n = input("Input an integer number: ")
    summation = summation + int(n)
    count = count + 1
print("The sum of the 5 entered values is:", summation)
```

Programming languages

- A **programming language** is a language, the same as a **natural language**, (e.g., English, Arabic) but more strict and precise (no ambiguities!)
  - Programming language → **Formal language**
  - A formal language is defined by hand and by reasoning for a purpose, it is not *evolved by practice*

Symbols to construct words and sentences

Rules that describe how to make admissible words

Syntactic rules (a *grammar*) to construct well-formed sentences (*expressions*) out of the admissible words

Static semantics that defines which syntactically valid expressions have meaning in the language

Language

Semantics, which associates a meaning with each syntactically correct expression

Programming languages

Symbols to construct words and sentences

- English: letters $\{a - z\} \cup \{A - Z\}$, punctuation, accents, parentheses, …
- *Python*: {a-z} {A-Z} {0,1,..9} + - * / % = == += *= /= %= != > < <= >= >> << ++ [] () {} ~ ! & | ; , …

Rules that describe how to make admissible words

- English: School, ai: ✓ ?w ✗
- *Python*: this_is_a_variable ✓, 0this_is:a_variable ✗

Syntactic rules (a *grammar*) to construct well-formed sentences (*expressions*) out of the admissible words

- "Cat dog student" it is not admissible: *<noun> <noun> <noun>*
- "You are CMU students" it is well-formed!
- *Python:* 1.5 + 1.5 * a ✓, 1.5 1.5 a ✗

Static semantics that defines which syntactically valid expressions have meaning in the language

- "I have tired", "I are short", are syntactically correct (*<pronoun> <linking verb> <adjective>)* but have not meaning in English!
- *Python:* 4.5 / "abc" is syntactically well formed *(<literal> <operator> <literal>*) but produces a static semantic *error* since it doesn't make sense to divide a number by a string of text!

Programming languages: semantics

Semantics, which associates a meaning with each syntactically correct expression

Semantics: study of meaning in a language

- In English, semantics can be **ambiguous** (and context-dependent)
- *Look at the dog with one eye.*
  - Look at the dog that only has one eye.
  - Look at the dog using only one *of* your eyes.
  - Perhaps the dog has found an eye somewhere, and we're looking at the dog.

Programming languages: semantics

- In Python (and any other formal language) semantics of an expression is **uniquely defined**
- → Each "legal" program has exactly <u>one meaning</u>!
  - One set of outputs, depending on the inputs

- Python's formal language is powerful enough to program (compute) *<u>anything that can be computed</u>!*
  - Python is **Turing complete** 👍

  - <u>Warning:</u> YOU will have to take care of writing programs that are syntactically correct, complies with the static semantics of the language, and are semantically correct (they do what you expect them to do!)

Machine Languages

- Python, Java, C, and C++ are, indeed, examples of *high-level* languages
- Strictly speaking, computer hardware can only understand a very *low-level* language known as *machine language*
- If you want a computer to add two numbers, a and b, the instructions that the CPU will carry out might be something like this:

Load the number a from memory location 2001 into the CPU
Load the number b from memory location 2002 into the CPU
Add the two numbers in the CPU (using the ALU)
Store the result into the memory location 2003

A Lot of Work! 😰

- In Python:

```
c = 1 + 3
```

or

```
a = 1
b = 3
c = a + b
```

Much easier! 😊

A hierarchy of languages

Compiling a high-level language

- A *compiler* is a complex software that takes a program written in a high-level language and *translates* it into an equivalent program in the machine language of some computer
    - C, C++

Interpreting a high-Level language

- An *interpreter* is a software that analyzes and executes the source code instruction-by-instruction (*on-the-fly*) as necessary

Python is an <u>interpreted language</u>

Compiling vs. Interpreting

- Compiling is a **static** (i.e., pre-execution), one-shot translation
  - Once a program is compiled, it may be run over and over again without further need for the compiler or the source code
- Interpreting is **dynamic** (i.e., happens during execution)
  - The interpreter and the source code are needed every time the program runs
- Compiled programs tend to be faster, while interpreted ones lend themselves to a more flexible programming environments (*they can be developed and run interactively*)

Notes on Portability

- The translation process highlights another advantage that high-level languages have over machine languages, namely, ***portability***

- A program for an Intel-based machine will not run on an IBM-based machine since each computer type has its own machine language

- On the other hand, a program written in a high-level language (say, a Python program) can be run on many different kinds of computers as long as there is a suitable compiler or interpreter

- Python programs are said to be highly portable!

Summary: what you should know

- Types of knowledge and of programming language: imperative vs. declarative
- Understanding the general architecture of a computer and its general functioning
- Components of a language, Programming languages vs. Natural languages
- Machine language vs. High-level programming language
- Compilers vs. Interpreters