Introduction

Why are we doing this? – Your course in context

- A science built on logic, yet speaking in metaphors?
- What are computers?
- What are programming languages and how do they relate to computers?
- The experience of programming ...
- Things which will not work ...
- Things which work ...

References

[Church1932] Church, Alonzo
A set of postulates for the foundation of logic

[Church1936] Church, Alonzo
An unsolvable problem of elementary number theory
American Journal of Mathematics 1936 vol. 58 (2) pp. 345-363

[Gödel1931] Gödel, Kurt
Über formal unentscheidbare Sätze der Principia Mathematica

Mathematische Annalen 1931 vol. 92 (1-2) pp. 173-198

[Neumann1945] von Neumann, John
The First Draft Report on the EDVAC
Technical report 1945 pp. 1-51

[Rice1953] Rice, Henry Gordon
Classes of Recursively Enumerable Sets and Their Decision Problems
Transactions of the American Mathematical Society 1953 vol. 76 (2) pp. 35-386

[Thompson2011] Thompson, Simon
Haskell - The craft of functional programming
Addison Wesley, third edition 2011

{Turing1937} Turing, Alan
On computable numbers, with an application to the Entscheidungsproblem
Proceedings of the London Mathematical Society 1937 vol. 2 (42)

The language of Computer Science

Metaphors and Anthropomorphic Language

While computer science is based on logic, physics and psychology, its technical terms can be colourful and confusing to outsiders:

Bus, window, word, language, desktop, file, open, kill, bootstrapping, memory, library, drive, driver, virus, mouse, web, machine, assembler, surfing, backbone, handshake, tunnel, gates, port, cookie, slave, buffers, intelligent control, running …

"Computing" and "404" themselves are also used as metaphors in other areas or disciplines, like the cognitive sciences.

Deus ex machina?
or:
Glorified collection of switches?
**Introduction**

**Computer Science**

What is Computer Science about?

- **Some forms of hardware**
  - "Computers" and all their components
- **Any form of "software"**
  - "Programs" ranging from a heating controller switch to "running" the International Space Station
- **Theory**
  - Knowing what works (and what does not) and what works efficiently and reliably

Related disciplines:
- Mathematics (mostly discrete), Philosophy, Logic, Physics, Psychology, Design, Engineering (mostly electronics), ...

---

**Definition**

**Algorithm**

Abstract, yet finite and formal description of a method to achieve an intended output or effect, dependent on a range of possible input values.

- ... can introduce a finite number of internal states.
- ... can be deterministic or non-deterministic.
- ... can be sequential or concurrent.

Origins of the word:
- Abū' Abdallāh Muhammad ibn Māsā' Al-Khwārizmī (Al-Khwarizmi) (9th century Persian mathematician – wrote in Arabic).
- *Algoritmi de numero Indorum* (12th century Latin translation of an original Arabic script).
- *Algorithm* (17th century French term denoting the decimal number system).
- Algorithm (19th century English term – current meaning formed in the early 20th century).
Introduction

Definition

Algorithm

Abstract, yet finite and formal description of a method to achieve an intended output or effect, dependent on a range of possible input values.

- … can introduce a finite number of internal states.
- … can be deterministic or non-deterministic.
- … can be sequential or concurrent.

Origins of the word:
- Abū'Abdallāh Muhammad ibn Māsā Al-Khwārizmī (ابو عبد الله محمد بن مسأ الخوارزمی) (9th century Persian mathematician – wrote in Arabic).
- Algoritmi de numero Indorum (12th century Latin translation of an original Arabic script).
- Algorithm (17th century French term denoting the decimal number system).
- Algorithm (19th century English term – current meaning formed in the early 20th century).

© 2015 Uwe R. Zimmer, The Australian National University page 41 of 793 (chapter 1: “Introduction” up to page 99)

Building blocks

Controllable Switches & Ratios

as transistors, relays, vacuum tubes, valves, etc.

Credit: Wikipedia

© 2015 Uwe R. Zimmer, The Australian National University page 42 of 793 (chapter 1: “Introduction” up to page 99)

Building blocks

Controllable Switches & Ratios

as transistors, relays, vacuum tubes, valves, etc.

Digital Computers

Analogue Computers

© 2015 Uwe R. Zimmer, The Australian National University page 44 of 793 (chapter 1: “Introduction” up to page 99)
Introduction

Ratios - building blocks for analogue computers

Analogue Computers

In signal processing and control - and especially in sound & graphics

Mixed signal systems

Special purpose

Introduction

Logic - the basic building blocks for digital computers

Constructing logic gates – for instance NAND in CMOS:

... and subsequently all other logic gates:

Half adder:

Full adder:

Ripple carry adder:

No, you did not end up in the wrong course!
... hang in there ...
**Definition: Processor**

**Hardware origins**

18th century machines

**L'Ecrivain**

1770

Programmable, yet not a computer in today's definition (not Turing complete).
**Computer Architectures**

**A simple processor (CPU)**

- **Decoder/Sequencer**
  Can be a machine in itself which breaks CPU instructions into concurrent micro code.

- **Execution Unit / Arithmetic-Logic-Unit (ALU)**
  A collection of transformational logic.

- **Memory**
  Instruction pointer, stack pointer, general purpose and specialized registers.

- **Flags**
  Indicating the states of the latest calculations.

- **Code/Data management**
  Fetching, Caching, Storing.

---

**Harvard Architecture**

- **Control unit**
  Concurrently addresses program and data memory and fetches next instruction. Controls next ALU operations and determines the next instruction (based on ALU status).

- **Arithmetic Logic Unit (ALU)**
  Fetches data from memory. Executes arithmetic/logic operation. Writes data to memory.

- **Input/Output**
  Program memory

- **Data memory**
  Programs can change themselves.

---

**von Neumann Architecture**

- **Control unit**
  Sequentially addresses program and data memory and fetches next instruction. Controls next ALU operations and determines the next instruction (based on ALU status).

- **Arithmetic Logic Unit (ALU)**
  Fetches data from memory. Executes arithmetic/logic operation. Writes data to memory.

- **Input/Output**
  Memory

  Programs and data is not distinguished

  Programs can change themselves.

---

**Computer Architectures**

**A simple processor (CPU)**

- **Decoder/Sequencer**
  Can be a machine in itself which breaks CPU instructions into concurrent micro code.

- **Execution Unit / Arithmetic-Logic-Unit (ALU)**
  A collection of transformational logic.

- **Memory**
  Instruction pointer, stack pointer, general purpose and specialized registers.

- **Flags**
  Indicating the states of the latest calculations.

- **Code/Data management**
  Fetching, Caching, Storing.

... all this can be build out of NAND gates, which are based on the controllable switch technology of the time!

In later courses you will learn about pipelines, multi-cores, hyper-threads, vector machines, virtual memory, and a whole lot more ...
Introduction

Hardware

Where are all the computers?

- ~98% of all CPUs are found in embedded devices.
  - Modern cars exceed 100 CPUs per vehicle.
- ~7.3 billion mobile phones worldwide (2014) > 1 per person.
- ~2% of all CPUs are found in laptops and desktops.
- High performances are achieved by special purpose GPUs
- ~1-2 billion laptop and desktop computers worldwide.
- The majority of CPUs are 8-bit and programmed in C or Assembler.

How many CPUs does your laptop sport?

... my key-ring torch has a CPU.

Programming processors

Assembler

<table>
<thead>
<tr>
<th>Label</th>
<th>Memory address</th>
<th>Memory contents</th>
<th>Assembler instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>Symbolic</td>
<td>Hexadecimal</td>
<td>Binary</td>
<td>Hexadecimal</td>
</tr>
<tr>
<td>00001000</td>
<td>01001000</td>
<td>11100111</td>
<td>48 E7</td>
</tr>
<tr>
<td>00001002</td>
<td>00000010</td>
<td>00000000</td>
<td>20 00</td>
</tr>
<tr>
<td>00001004</td>
<td>00110010</td>
<td>00000000</td>
<td>34 00</td>
</tr>
<tr>
<td>00001006</td>
<td>00000100</td>
<td>01000010</td>
<td>04 42</td>
</tr>
<tr>
<td>00001008</td>
<td>00000000</td>
<td>00000001</td>
<td>00 01</td>
</tr>
<tr>
<td>0000100A</td>
<td>01000010</td>
<td>10000001</td>
<td>42 81</td>
</tr>
<tr>
<td>00001010</td>
<td>11000101</td>
<td>11001010</td>
<td>51 CA</td>
</tr>
<tr>
<td>00001012</td>
<td>11111111</td>
<td>11111100</td>
<td>FF FC</td>
</tr>
<tr>
<td>00001014</td>
<td>10000010</td>
<td>11000000</td>
<td>82 C0</td>
</tr>
<tr>
<td>00001016</td>
<td>01001100</td>
<td>11011111</td>
<td>4C DF</td>
</tr>
<tr>
<td>00001018</td>
<td>00000000</td>
<td>000000100</td>
<td>00 04</td>
</tr>
</tbody>
</table>

A little more human friendly.

Machine code

<table>
<thead>
<tr>
<th>Memory address</th>
<th>Memory contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>00001000</td>
<td>01001000</td>
</tr>
<tr>
<td>00001002</td>
<td>00000010</td>
</tr>
<tr>
<td>00001004</td>
<td>00110010</td>
</tr>
<tr>
<td>00001006</td>
<td>00000100</td>
</tr>
<tr>
<td>00001008</td>
<td>00000000</td>
</tr>
<tr>
<td>00001010</td>
<td>11000101</td>
</tr>
<tr>
<td>00001012</td>
<td>11111111</td>
</tr>
<tr>
<td>00001014</td>
<td>10000010</td>
</tr>
<tr>
<td>00001016</td>
<td>01001100</td>
</tr>
<tr>
<td>00001018</td>
<td>00000000</td>
</tr>
</tbody>
</table>

... or: what the world looks like if you are a computer.

Semantic

<table>
<thead>
<tr>
<th>Label</th>
<th>Memory address</th>
<th>Memory contents</th>
<th>Assembler instruction</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>Symbolic</td>
<td>Hexadecimal</td>
<td>Binary</td>
<td>Hexadecimal</td>
<td></td>
</tr>
<tr>
<td>00001000</td>
<td>01001000</td>
<td>11100111</td>
<td>48 E7</td>
<td></td>
</tr>
<tr>
<td>00001002</td>
<td>00000010</td>
<td>00000000</td>
<td>20 00</td>
<td></td>
</tr>
<tr>
<td>00001004</td>
<td>00110010</td>
<td>00000000</td>
<td>34 00</td>
<td></td>
</tr>
<tr>
<td>00001006</td>
<td>00000100</td>
<td>01000010</td>
<td>04 42</td>
<td></td>
</tr>
<tr>
<td>00001008</td>
<td>00000000</td>
<td>00000001</td>
<td>00 01</td>
<td></td>
</tr>
<tr>
<td>00001010</td>
<td>11000101</td>
<td>11001010</td>
<td>51 CA</td>
<td></td>
</tr>
<tr>
<td>00001012</td>
<td>11111111</td>
<td>11111100</td>
<td>FF FC</td>
<td></td>
</tr>
<tr>
<td>00001014</td>
<td>10000010</td>
<td>11000000</td>
<td>82 C0</td>
<td></td>
</tr>
<tr>
<td>00001016</td>
<td>01001100</td>
<td>11011111</td>
<td>4C DF</td>
<td></td>
</tr>
<tr>
<td>00001018</td>
<td>00000000</td>
<td>000000100</td>
<td>00 04</td>
<td></td>
</tr>
</tbody>
</table>

Input: D0 discrete values which are stored beginning at memory address (A0).

Output: Discrete arithmetic mean in D1.

Method: D0 values are accumulated in D1 and divided by the number of values.

Side effects: none.
Programming processors

Machine level programming

Observations:
- Dependent on the specific CPU model.
- Error prone.
- Harder to handle complex problems.
- The machine dictates the language in which the programmer "thinks".
- Full control.

Need for more problem oriented languages.
Need for more abstract and safer languages.

P.S. How much control are you willing to or need to sacrifice for abstraction?

Programming languages

Early languages

Machine code inspired (imperative, extended macro assemblers):
- **Fortran** ('57), **Cobol** ('59), **Basic** ('64), **C** ('71)
- **Algol** ('58), **Algol60**, **Algol68**, **Pascal** ('70)

Based on lambda calculus (functional programming):
- **Lisp** ('58), **ML** ('73)

Structured and strongly typed (imperative):
- **Algol** ('58), **Algol60**, **Algol68**, **Pascal** ('70)

Based on message passing (object oriented):
- **Simula** ('67), **SmallTalk** ('69)

Based on declarative, first order logic:
- **Prolog** ('72)

Imperative means (informally) that the program is a sequence of instructions.

Functional programming means (informally) that the program is a set of functions transforming input to output without storing any information (much like a mathematical function).
Programming languages

Early languages

Machine code inspired (imperative, extended macro assemblers):

- Fortran ('57), Cobol ('59), Basic ('64), C ('71)

Based on lambda calculus (functional programming):

- Lisp ('58), ML ('73)

Structured and strongly typed (imperative):

- Algol ('58), Algol60, Algol68, Pascal ('70)

Based on message passing (object oriented):

- Simula ('67), SmallTalk ('69)

Based on declarative, first order logic:

- Prolog ('72)

While those languages provided a basis for most languages today, none of those over 40 year old editions are capable to handle large scale problems or modern hardware by themselves.

Find the right language

Ask a fanboy?

Follow the hype?

Choose what seems popular?

Choose what promises to be "easy"?

Use the first language you learned for everything else in your career?
Introduction

Programming languages

Find the right language

- Ask a fanboy?
- Follow the hype?
- Choose what seems popular?
- Choose what promises to be “easy”?
- Use the first language you learned for everything else in your career?
- Choose the best tool for the job.

Paradigms

- Control flow: Imperative ↔ Declarative
- Declarative: Functional ↔ Logic ↔ Finite State Machines
- Allocations and bindings: Static ↔ Dynamic
- Time: Event-driven ↔ Discrete ↔ Synchronous ↔ Continuous
- Focus: Control flow-oriented ↔ Data flow-oriented
- Degree of concurrency: Sequential ↔ Concurrent ↔ Distributed
- Structure: Modular ↔ Generics ↔ Templates ↔ Object-Oriented ↔ Aspect-Oriented ↔ Agent-Oriented
- Determinism: Deterministic ↔ Non-deterministic

Influential languages

Languages which could be considered influential:

\[ \lambda\text{-Calculus}/\text{Lisp} \rightarrow \text{Simula}/\text{Smalltalk} \rightarrow \text{Algol} \rightarrow \text{Prolog} \]

... conceptual foundations:

\[ \text{ML}/\text{Haskell} \rightarrow \text{Eiffel} \rightarrow \text{Ada} \rightarrow \text{C} \rightarrow \text{Java} \]

... other major influences:
Programming languages

Influential languages

... by counting how many other languages they influenced.

According to this method:

Haskell, Lisp, Smalltalk and C

have inspired the most other languages

... don't take this method too seriously.

© 2015 Uwe R. Zimmer, The Australian National University
Programming languages

Influential languages

Imperative, Algol family: (or: “begin-end languages”)
Algol 60, Algol 68, Pascal, Modula, Modula-2, Oberon, Object-Pascal, Ada, ...

Note: Influence does not imply popularity. Algol 60 is one of the most influential languages, yet it will be hard to find a compiler for it today.

Influential languages

Functional (λ-Calculus) languages:
Lisp, Common-Lisp, ML, OCaml, Scheme, Haskell, ...

Logic languages:
Planner, Prolog, ...

Scripting languages:
Perl, Ruby, Python, JavaScript, ...

Introduction
### Programming languages

**Influential languages**

- Perl, Ruby, Python, JavaScript, ...

**Scripting languages:**

- JavaScript
- Python
- Ruby

Note: Popularity does not imply influence. PHP is one of the most used languages today, yet it is commonly considered too flawed to base any new language design on it.

### Concrete languages

- Most languages offer a mixture of paradigms, yet a few are based on a small base of orthogonal primitives.

- Some are more universal than others, yet no language serves all purposes.

Orthogonal primitives means here: For most problems there is exactly one way to do it. Or “less freedom of choice, less keywords to learn.”
Introduction

Programming Languages

Concrete languages

- Most languages offer a mixture of paradigms, yet a few are based on a small base of orthogonal primitives.
- Some are more universal than others, yet no language serves all purposes.

Other pragmatic questions to consider:
- How good is the compiler (compilation speed, performance of the produced code)?
- How comfortable is the "development environment"?
- Does it work well on the platform(s) (computer system(s)) you need?

Translations

Programs need to be executed on an actual computer.
- All programs need to be translated into machine code.
- This happens sometimes in multiple stages (via intermediate languages and machines).

Compiler

Translates a higher level program as a whole into a lower level language.

Interpreter

Translates a higher level program into a lower level language while the higher level program progresses (slower but potentially more interactive).

Programming Languages

Concrete languages

- Most languages offer a mixture of paradigms, yet a few are based on a small base of orthogonal primitives.
- Some are more universal than others, yet no language serves all purposes.

Other pragmatic questions to consider:
- How good is the compiler (compilation speed, performance of the produced code)?
- How comfortable is the "development environment"?
- Does it work well on the platform(s) (computer system(s)) you need?

Translations

Programs need to be executed on an actual computer.
- All programs need to be translated into machine code.
- This happens sometimes in multiple stages (via intermediate languages and machines).

Compiler

Translates a higher level program as a whole into a lower level language.

Interpreter

Translates a higher level program into a lower level language while the higher level program progresses (slower but potentially more interactive).

Haskell

- Functional programming language (based on the λ-calculus).
- Statically typed with polymorphism and operator overloading.
- Uses monads to provide side-effects and imperative sequences.

Pragmatic aspects:
- Offers compilers as well as interactive environments.
- Concise and compact syntax.
- Compiler detects many errors (while error messages can be confusing).
- Performs mostly better than many yet slower than some languages (e.g., Ada, C++, ...).
- Commonly used for: Verifiable systems, High integrity (non-real time) systems
- Not directly usable for: Real-time systems, High performance systems.

Used as the main language for the course.
Introduction

Programming Languages

Human craft and correctness

Programming is a human activity ("The art of programming").
  - A program can be well crafted and aesthetical.
  - A program can be maintainable or unreadable.

Programming is an expression in (temporal) logic:
  - A program can be correct or incorrect (against a specification).

© 2015 Uwe R. Zimmer, The Australian National University
**Introduction**

**Programming Languages**

*Programming is ...*

... *essentially easy* – everybody can do it in a few easy tutorials?

What the salesman of the latest trendy language wants to make you believe.

... *intrinsically hard* – only the best and bravest will ever get it?

That’s what the “old boys” might tell you to keep you out of their feet.

... *a weird way to spend your day* – lock yourself into a nerdy cellar?

Many Hollywood movies about programmers show some of those.

... or ...

... *a specialized, comparatively recent, professional activity*, which

requires plenty of *care and focus* (just like any advanced profession).

---

**Theory of programming**

*Some things will never work*

Decision problems:

- Is a program syntactically correct? ✓
- Is $x$ a prime number? ✓
- Will a program stop given a certain input? ✗

Can one write programs which answer those questions for all cases?

---

The last one (the “Halting problem”) is a representative for a

class of algorithmically not solvable decision problems.

---

**Undecidable problems**

Kurt Gödel (‘31): in loose, short translation and paraphrasing:

Any consistent theory can express *true statements* which *cannot be proven* within this theory.

Turing (‘37): Implications from his article:

“On Computable Numbers With an Application to the Entscheidungsproblem”:

*No program can decide* whether

another arbitrary program *will terminate* on a given input.

*Non-trivial properties of programs are algorithmically undecidable*

(Rice’s theorem ’53), yet can often be proven for specific cases.
Some programs work reliably

- Most flight computers.
- Car braking controllers.
- Banking systems.
- High speed trains, subway systems.
- Internet search engines.
- Professional audio/video equipment.

Professional programming leads to many high integrity systems, some of which we often trust with our lives.

Programming

What makes a professional programmer?
or: How to migrate from “kinda works” to reliable?

- Is “fluent” in all essential programming concepts and paradigms.
- Knows the right tools for the job and uses them.
- Knows what the available hardware is capable of and takes this into consideration.
- Understands the translation into executable machine code and controls its essential parameters.
- Understands testing and verification and applies them adequately.
- Finds the best suited abstractions and modularisations (requires experience).
- Knows how to analyse unexpected problems (“debugging”).

This course is the first step.

Summary

Introduction

- Definitions
  - Algorithm.

- Computers
  - Basic hardware concepts.
  - Basic forms and distribution.

- Programming languages
  - Why are they needed?
  - Existing paradigms and relation to hardware.

- Things which work and things which won’t.
  - Impossibly and reliable systems.