MTH 135

Structure & Comparison of Programming Languages

Prof. dr. Irma Ravkic

Course overview

- Chapter 1: why learning programming concepts
- Chapter 2: evolution of the major programming languages
- Chapter 3: Describing syntax and semantics
- Chapter 4: Lexical and Syntax analysis
- Chapter 5: Names, Bindings and Scopes
- Chapter 6: Data types
- Chapter 7: Expressions and Assignment Statements
- Chapter 8: Statement-Level Control Structure

We don't cover but might mention some concepts if I have time:

- abstractions and concurrency

Course overview

- After each chapter we will have homeworks and quizzes

● Homework – approximately 10 homework sets consisting of questions from the text. Assignments must be turned in on time. Two points will be deducted for each class day your homework set is late (this “two-point policy” does not apply to the final project...see “Final Project” Handout for details).

● Quizzes – we will have 6 quizzes, one quiz for each chapter (except chapter 8). Each quiz will be approximately 10 - 20 points and will cover material from class and/or from the text. All quizzes will be online and will be open book.

● Research Paper – Write a report on any computer language you choose (except C or C++). You must choose a language that you have not studied before and each student must pick a different language. The paper should be approximately 20 typewritten double-spaced pages in length and include the following sections: History of the Language, Syntax and Semantics, Program Examples, Evaluation, Resources.

Course overview

The breakdown of grades is as follows:

Homework	40%
Quizzes	35%
Final Project	25%

and the grade scale is:

93% - 100%A	77% - 79% C+
90% - 92% A-	73% - 76% C
87% - 89% B+	70% - 72% C-
83% - 86% B	60% - 69% D
80% - 82% B-	below 60% F

NOTE: If the Final Project is not completed you cannot earn a passing grade.

Something about my teaching philosophy

I am new at university teaching and MSMU, meaning we will work more closely to understand the material

I like making jokes and I have a very casual style, but I will warn you when you’re too loud :)

I will use Canvas to the best of its possibilities (I am still learning, though)

- **IDKE** (I don’t know either/everything) - I will post your interesting questions in the discussion section of each course

- I will post some interesting additional material at an appropriate place in Canvas

  I will say many more things than are written in the book to give you a bigger perspective, but I will ask you only about the things that are in the book (isn’t this great?).

English is not my native language, so sorry for all the silly mistakes

Chapter I

Why learning programming concepts?
What are criteria for programming languages?
Language design and categories
Trade-offs
Implementation methods
Programming environments

Chapter I

Why learning programming concepts?

Why?

Increased capacity to express ideas.

Human language

words
phrases
dialect / slangs

Programming languages

data and control
structures,
abstractions

transfer skills: language constructs can be simulated in other languages that do not support those constructs directly

Why?

Improved background for choosing appropriate languages.

often limited to one or two languages that are directly relevant to the current projects of the organization

when given a choice of languages for a new project, use the language with which they are most familiar, even if it is poorly suited for the project at hand

vis-a-vis feature simulation: it is preferable to use a feature whose design has been integrated into a language than to use a simulation of that feature, which is often less elegant, more cumbersome, and less safe

Why?

Improved background for choosing appropriate languages:

I want to write large desktop applications

Java

C/C++, C#

I want to make animations

Processing

I want to develop games for consoles and desktop applications

C++

C#

I want to do maths

R

Python

Haskell

OCAML

Lisp

I want to do machine learning

Python

Scala

R

C

Why?

Improved background for choosing appropriate languages.

```
echo “Hello World”
```

```
int main()
{
    std::cout << "Hello, world!
";
    return 0;
}
```

```
document.write('Hello, world!');
```

a thorough understanding
of the fundamental concepts of languages ->
easier to see how these concepts are
incorporated into the design of a new language

the better you
know the grammar of your native language, the
easier it is to learn a second language

Why?

Better understanding of the significance of implementation.

certain kinds of program bugs can be found and fixed only by a programmer who knows some related implementation details

knowing how execution works: programmers who know little about the complexity of the implementation of subprogram calls often do not realize that a small subprogram
that is frequently called can be a highly inefficient design choice

Why?

Better use of languages that are already known.

many contemporary programming languages are large and complex:

programmers can learn about previously unknown and unused parts of the languages they already use and begin to use those features

Why?

Overall advancement of computing.

ALGOL 60 should have displaced FORTRAN in 1960s:
it was more elegant and had much better control statements, among other reasons

Why not?
many programmers at the time did not clearly understand the conceptual design of ALGOL 60

Why?

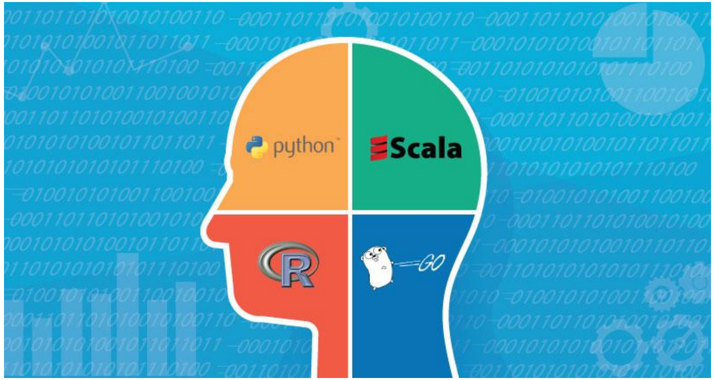
Versatile applications.

Scientific applications

large numbers of floating point arithmetic calculations

arrays and matrices

Fortran: first language for scientific applications



Why?

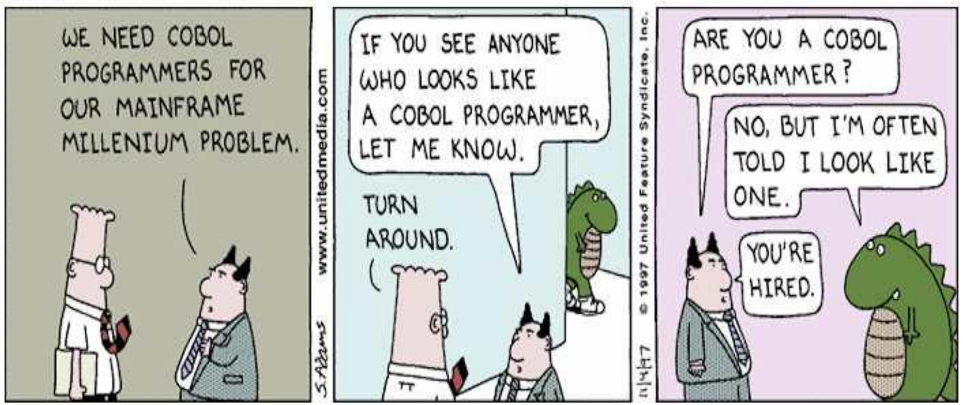
Versatile applications.

Business applications

produce elaborate reports, precise storing and manipulations of decimal numbers

the use began in 1950s

first successful: COBOL (COmmon Business-Oriented Language)



Interesting read on language extinction: https://devops.com/cobol-completely-obsolete-omnipresent-language/

Why?

Versatile applications.

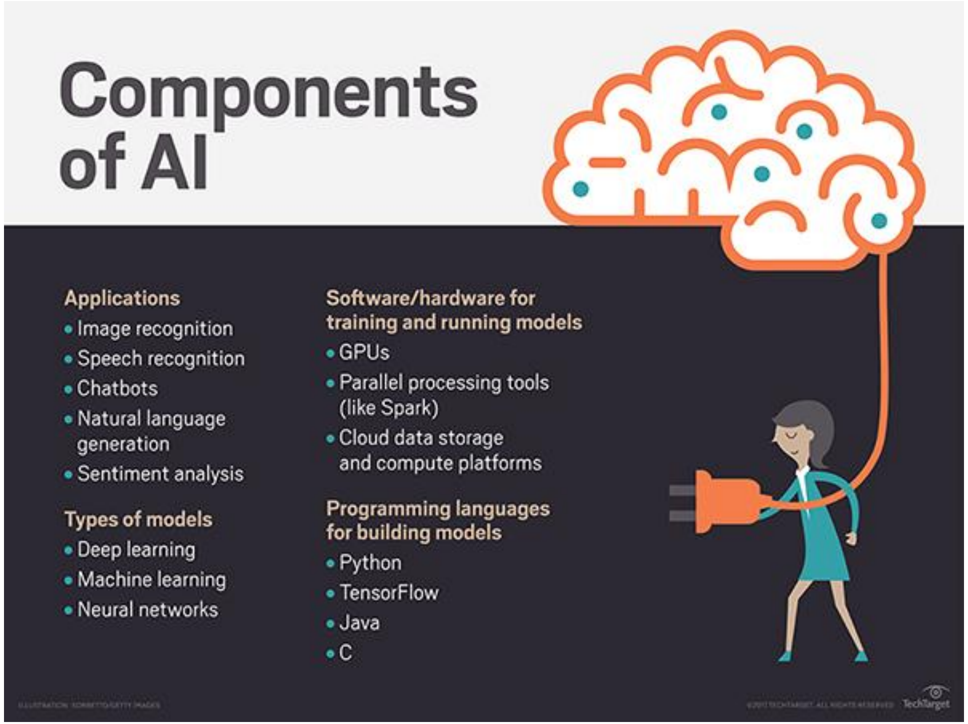
Artificial Intelligence applications

symbolic rather than numeric computations

linked lists (trees)

require more flexibility (create and execute code segments during execution)

functional language LISP; logic programming Prolog;



Why?

Versatile applications.

Systems programming

- runs continuously - efficient
- needs low level features to interact with the hardware
- 1960s - 1970s, special machine-oriented
- high-level languages for systems software: IBM (PL/S), Digital (BLISS), Burroughs (Extended ALGOL)
- most system software now written in C/C++ (e.g., UNIX, entirely in C)
- Why C? Low level, execution efficient, not many safety restrictions (!)

Why?

Versatile applications.

Web Applications

Markup languages (HTML - not a programming language), to general purpose languages Java

Dynamic web content: putting some programming code into HTML (PHP or JavaScript)

Chapter I

What are criteria for programming languages?

Criteria and their influences

Table 1.1 Language evaluation criteria and the characteristics that affect them

	CRITERIA		
Characteristic	READABILITY	WRITABILITY	RELIABILITY
Simplicity	•	•	•
Orthogonality	•	•	•
Data types	•	•	•
Syntax design	•	•	•
Support for abstraction		•	•
Expressivity		•	•
Type checking			•
Exception handling			•
Restricted aliasing			•

Readability

the ease with which programs can be read and understood

before: efficiency most important; programming for the point of computers not users

in 1970s understood that maintenance of systems very important (as the function of readability)

Readability

Feature 1: Simplicity

Problem 1: a language with a large number of basic constructs is more difficult to learn than one with a smaller number (learn a subset, ignore the rest)

Problem 2: feature multiplicity

```
count = count + 1
count += 1
count++
++count
```

Problem 3: Operator overloading
Even though useful, can lead to reduced readability (overloading +)
Assembly languages simple, but reduced readability

Readability

Feature 2: Orthogonality

a relatively small set of primitive constructs can be combined in a relatively small number of ways to build the control and data structures of the language

-independent of the context of its appearance in a program
(define a pointer to point to any specific type defined in the language)

VAX:
ADDL operand_1, operand_2
Semantics: operand_2 ← contents(operand_1) + contents(operand_2)
(a single instruction can use either registers or memory cells as the operands)

Vs. IBM:
A Reg1, memory_cell
AR Reg1, Reg2

Not orthogonal: different instructions, only two out of four combinations legal

Readability

Feature 2: Orthogonality

vs. simplicity: fewer exceptions -> higher regularity -> easier to learn, read

Lack of orthogonality in C:

1) Arrays and struct: struct can be returned from function, array cannot

2) Context dependence: a+b if *a* points to a float value that occupies four bytes, then the value of *b* must be scaled—in this case multiplied by 4—before it is added to *a*

Readability

Feature 2: Orthogonality

Too much orthogonality

Most orthogonal: ALGOL 68

1) Every language construct has a type
2) Most constructs produce value

Explosion of combinations!

Functional languages: greatest simplicity (everything is a function call); not very efficient though

Readability

Feature 3: Data Types

A numeric type is used for an indicator flag because there is no Boolean type in the language

```
timeOut = 1
```

But introducing a Boolean value denoting the truth of an expression:

```
timeOut = true
```

Readability

Feature 4: Syntax Design

the syntax, or form, of the elements of a language has a significant effect on the readability of programs

Special words: (while, for, class)
words for ending blocks instead of braces

Fortran95 and Ada: if...end if, loop...end loop

reduced simplicity - increased readability

In Fortran: Do, End can be variable names (confusing)

Form and meaning:

statement names indicating their purpose

Violation: same form, different meaning
static
(compile time vs. visible only in the file)

UNIX: grep (g/regular_expression/p)
g - global commands
p - print matching lines

Writability

the ease with which programs can be written

most of the language characteristics that affect readability also affect writability

Writability

Feature 1: Simplicity and orthogonality

too many language constructs: misuse of some features and a disuse of others that may be either more elegant

use unknown features accidentally, with bizarre results

a programmer can design a solution to a
complex problem after learning only a simple set of primitive constructs

too much orthogonality: if any combination possible, errors

Writability

Feature 2: Abstraction

the ability to define and then use complicated structures or operations in ways that allow many of the details to be ignored

Process abstraction: using subprograms

Data abstraction: In C++ and Java, the binary trees can be implemented by using an abstraction of a tree node in the form of a simple class with two pointers (or references) and an integer

Writability

Feature 3: Expressivity

lots of computation with small programs

- count++ much better than longer count = count +1
- short-circuit the conditions: and then in Ada
- for vs. while

Reliability

A program is said to be reliable if it performs to its specifications under all conditions

Reliability

Feature 1: Type checking

testing for type errors in a given program, either by the compiler or during program execution

run-time type checking is expensive, compile-time type checking is more desirable

For example: an int type variable could be used as an actual parameter in a call to a function that expected a float type as its formal parameter, and neither the compiler nor the run-time system would detect the inconsistency

Reliability

Feature 2: Exception Handling

the ability of a program to intercept run-time errors (as well as other unusual conditions detectable by the program), take corrective measures, and then continue

Reliability

Feature 3: Aliasing

having two or more distinct names that can be used to access the same memory cell

dangerous feature

for example, two pointers set to point to
the same variable, which is possible in most languages

Cost

the cost of training programmers to use the language

the cost of writing programs in the language

the cost of compiling in that language

the cost of executing the program (dependent on language design)

- trade-off can be made between compilation cost and execution speed of the compiled code - optimization

the cost of language implementation system (free compiler/interpreter)

the cost of poor reliability

the cost of maintenance

Related to writability and readability

Other possible criteria

Portability

the ease with which programs can be moved from one implementation to another

standardization of a language: a long and costly process

Generality

the applicability to a wide range of applications

Welldefinedness

the completeness and precision of the language’s official defining document

Chapter I

Influences on Language Design

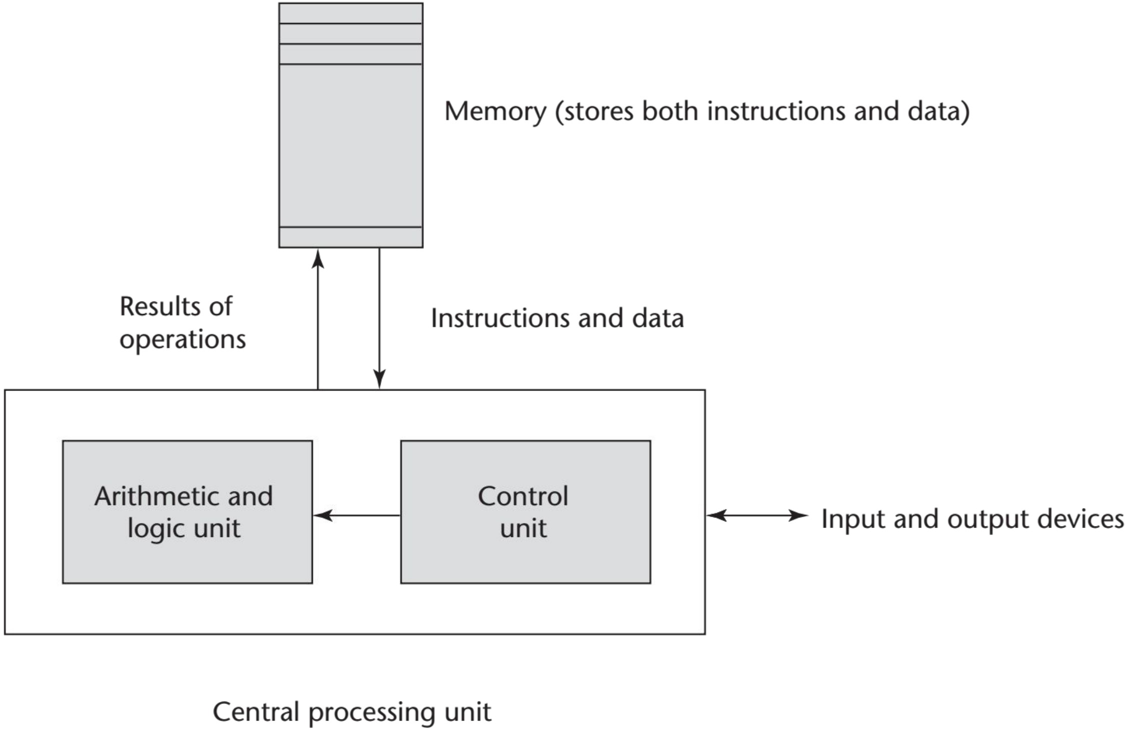
computer architecture

programming design methodologies

Von Neumann Computer Architecture

Figure 1.1

The von Neumann computer architecture



Von Neumann Computer Architecture

the central features of imperative languages are variables, which model the memory cells; assignment statements, based on the piping operation; the iterative form of repetition, the most efficient way to implement repetition on this architecture

iteration is fast on von Neumann computers: instructions stored in adjacent cells of memory and repeating the execution of a section of code requires only a branch instruction

Programming design methodologies

Early 60s: simple equations for solving problems; hardware costs high

Early 70s: hardware costs smaller than software/programmers costs

Late 70s: from procedure-oriented to data-oriented; emphasize data design, focusing on the use of abstract data (SIMULA 67) types to solve problems

Early 80s: object-oriented design; data abstraction, which encapsulates processing with data objects and controls access to data, and adds inheritance (reuse) and dynamic method binding (flexible use of inheritance); Smalltalk (1989), ADA 95, Java, C++, C#, F#

Chapter I

Language Categories

imperative

functional

logic

object-oriented

Language categories

imperative: visual (.NET; drag-n-drop), scripting languages

functional: treats computation as the evaluation of mathematical functions and avoids changing-state and mutable data

logic: rule-based language; no particular order of rules; no need to write extensive procedural instructions

object-oriented: a programming paradigm based on the concept of "objects", which may contain data, in the form of fields, often known as attributes; and code, in the form of procedures, often known as methods

Chapter I

Implementation methods

compiler implementation

pure interpretation

hybrid implementation

Implementation

computer: internal memory (programs + data)
processor (collection of circuits enabling operations)

machine language: set of its instructions

requires large collection of programs, called the operating system, which supplies higher-level primitives than those of the machine language

language implementation systems
need many of the operating system facilities, they
interface with the operating
system rather than directly with the processor (in
machine language)

Compilation

programs can be translated into machine language, which can be executed directly on the computer - fast execution (C, COBOL, C++, Ada)

lexical analyzer: gathers the characters of the source program into lexical units (ids, special words)

syntax analyzer: construct hierarchical structures called parse trees - the syntactic structure of the program