LMU LUDWIG-MAXIMILIANS-UNIVERSITÄT MÜNCHEN

P6 – Scientific Programming

Marcus Mohr
Jens Oeser

Geophysics Section
Department of Earth and Environmental Sciences
Ludwig-Maximilians-Universität München

SoSe 2021

Part #3

Programming Languages

basic concepts, history, classification

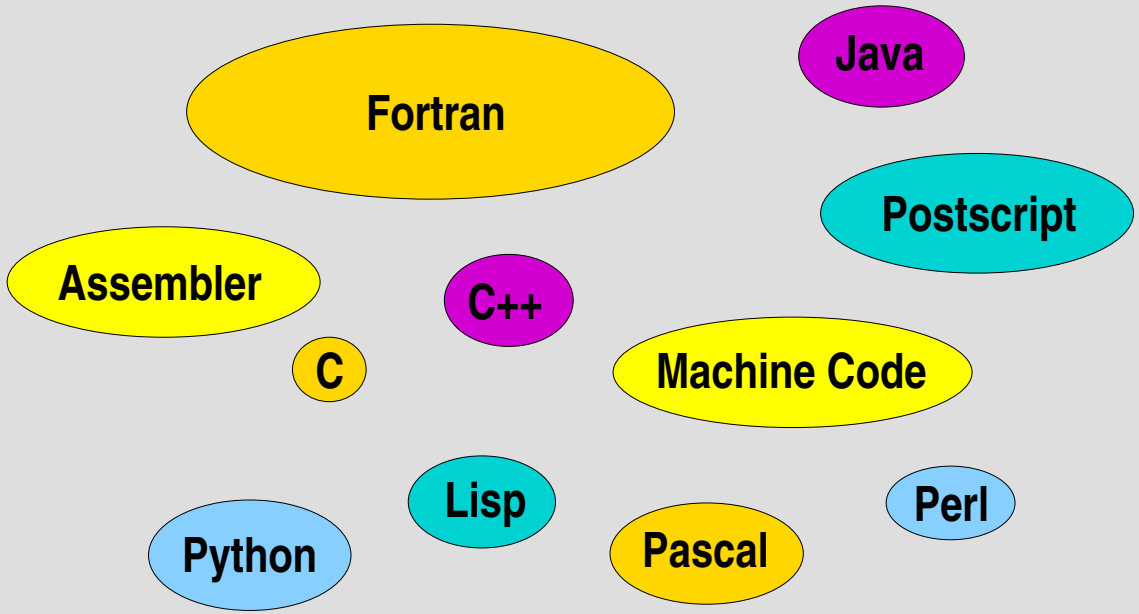
What is Programming?

Programming is the act of implementing a certain algorithm in a certain programming language.

A programming language forms an interface between a human and the computer.

Programming Languages

Since the 1950ies over a 1000 different programming languages have been developed, some examples are



and more are in the making . . .

Important for Computational Science

In the context of writing simulation codes for numerical models only a small amount of languages is currently of real importance:

- the Fortran language family, i.e. FORTRAN77, Fortran $\{90, 95, 2003, 2008\}$
- C (originally designed for programming Unix OS)
- C++ (object-oriented language)
- Python (is getting a lot attention currently $\rightarrow$ see e.g. ObsPy)

Two Basic Concepts

Syntax:

The syntax of a programming language defines what are the allowed words of the language and how sentences/programs are constructed, something like the grammar of the language.

Semantics:

This term denotes the meaning of the words/symbols of the language and the programs constructed form them.

Historical Development

The development of programming languages directly followed that of computing machinery. It can roughly be divided into 5 phases:
(see e.g. Henning & Vogelsang)

generation	languages	since
1	binary machine code	1950
2	assembler languages	1950
3	problem oriented, imperative/procedural	1954
4	4GL-languages (application oriented)	1960
5	declarative languages	1978

1st Generation

- Initially computers could only be programmed using machine language, this was referred to as coding. Commands are binary codes that are send to the processor together with the data they act on.
- Example: 10110000 01100011 (hex: B0 63) is the command to move operand 0x63 into the AL register on an x86 processor
- writing complex software is 'very tedious'
- language is processor dependent and not portable

2nd Generation

- pure machine code was soon replaced by assembler languages
- assembler languages replace hexadecimal codes by short descriptive abbreviations (so called mnemonics) and offer other improvements like e.g. symbolic address references
- previous example now reads: `mv al, 63h`
- a program (often also called assembler) is needed to transform the assembler program into executable machine code
- assembler programming is sometimes still used, e.g. to write highly optimised routines (especially for new architectures)

3rd Generation

- to achieve portability, make programming easier and allow scientists to enter this realm the first so called high-level languages were developed
- the first such language that succeeded was FORTRAN in 1954
- high-level languages are abstract, i.e. they are no longer written for a certain processor
- a compiler is required to transform the source code into executable (machine code)
- object-oriented (starting 1970ies) and some declarative (starting 1960ies) languages also belong to the 3rd generation

4^{th} and 5^{th} Generation

- here classification starts to become a little blurred
- 4^{th} generation languages may be characterised by the idea to only specify what should be achieved, but not the details how it is to be done
- examples are SQL (formulation of data-base queries), Postscript or XML (as a descriptive language)
- 5^{th} generation consists of declarative languages like PROLOG ($\rightarrow$ programming paradigm)

Classification I

Programming languages may be classified by the underlying programming paradigm:

imperative / procedural

example: Fortran, C

object oriented

example: C++

object based

example: JavaScript

declarative

example: LISP

Note, however, that a language may support more than one paradigm.

Classification I (cont.)

Imperative/procedural:

- computations consist of a sequence of statements that change the state of the program, such as assignments to variables or subroutine calls
- re-use and encapsulation of functionality is based on subroutines (functions and procedures)
- natural extension of machine language

Declarative:

- focusses on the what not the how; esp. *functional* and *logical* languages
- functional programs only consist of series of functions transforming input into output, but know no assignments (stateless)
- logical programming is based on specifying known facts (predicates) and using mathematical logic/reasoning for problem solving

Classification I (cont.)

Object-based:

- computation is based on objects and their interaction
- objects combine data and methods acting on these data
- re-use and encapsulation of functionality is based on objects/classes and their methods

Object-oriented:

- no clear distinction
- like object-based, but support enhanced concepts, such as *inheritance* and *polymorphism*

Classification II

- Another way to classify programming languages is by the approach taken to execute the instructions in the source code.
- Note for picky people: Formally this is not a language feature, but a distinction based on common implementation practice.
- One distinguishes between compiled and interpreted languages and mixed approaches.

Compiled Languages

The source code of the program is translated into executable machine code by another program, a so called compiler.

- improves execution time
- compiler can check for errors during compilation
- typical edit-compile-test cycle
- examples: Fortran, C, C++

Interpreted Languages

At runtime a so called interpreter reads in the source code of the program and executes it one command after the other.

- no edit-compile-test cycle
- interpreter can check execution ($\rightarrow$ safe environment)
- execution may be much slower
- examples: Perl, JavaScript, R, Matlab, (Python)

Mixed Approaches

The separation into *compiled* and *interpreted* is no longer complete. Java source code e.g. is converted into an intermediate format by a compiler (so called byte code) which is executed by an interpreter.

- allows execution in a safe environment
- significantly reduces runtime penalty

Same holds for Python (`*.pyc` files); again this is in the end a question of definition (does a compiler need to generate machine code, or not?)

References

- Henning, Vogelsang, Handbuch Programmiersprachen, Hanser Verlag
- Hans Jürgen Schneider: (Anmerkungen zur) Geschichte der Programmiersprachen, Folien zur Vorlesung, WS 07/08, LS Informatik 2, FAU Erlangen