Lecture 1

Introduction to DALG, Programming Paradigms

TDDD86Data Structures, Algorithms and Programming Paradigms

Utskriftsversion av Lecture in *Data Structures, Algorithms and Programming Paradigms*
August 28-29, 2023

IDA, Linköping University

1.1

Content

Contents

1.2

1 Administrative aspects

Teacher and personal

Ahmed Rezine	Lecturer
Caspe N. Kanefall	Teaching assistant, group D2.a
Gustav Carlsson	Teaching assistant, group D2.b
Szilvia Varro-Gyapay	Teaching assistant, group D2.c
Oscar Karlsson	Teaching assistant, group U2.a
Hampuz Togeretz	Teaching assistant, group U2.b
Anders Luong	Teaching assistant
Benjamin Sundvall	Teaching assistant
Linus Gardshol	Teaching assistant
Anna Grabska Eklund	course administrator

1.3

Literature

- C++
  - C++ Primer 5/E, Lippman, Lajoie, Moo
  - The C++ Programming Language 4/E, Stroustrup
- Data structure and algorithms
  - OpenDSA
- Programming Paradigms
  - Lectures handouts and linked material

1.4

Examination

- UPG2 1hp — Report (U,G)
- UPG1 2hp — Computer-based assignments (U,G)
- DAT1 3hp — Computer-based exam (U,3,4,5)
- LAB1 5hp — Labs (U,3,4,5)
- The final grade is the weighted average of the grades of DAT1 and LAB1, rounded to the nearest integer. See: `www.ida.liu.se/~TDDD86/exam/`

1.5

Labs: 5hp (U,3,4,5)

- 8 labs (4 in HT1 and 4 in HT2)
  - Conducted in pair
  - To pass you need to reach grade 3
  - A higher grade can be achieved by collecting bonus points
- See `www.ida.liu.se/~TDDD86/info/labs.sv.shtml`
- You need to register to Webreg to get your points reported:
  - In pairs for the labs before September 6th : LAB1 under webreg `www.ida.liu.se/webreg3/TDDD86-2023-1/LAB1`
  - In same pairs (or individually) before December 1st: extra assignments `www.ida.liu.se/webreg3/TDDD86-2023-1/Extra%20labs`
  - Individually for the Kattis problems before January 1st: Bonus problems `www.ida.liu.se/webreg3/TDDD86-2023-1/Bonus%20problems`
- Use Gitlab, follow style guide: `www.ida.liu.se/~TDDD86/info/labs.sv.shtml`
- We might need to move some pairs from one group to the other in order to avoid bias situations.

1.6

Your involvement in the course

- Attend the lectures (if you want)
- Study under the whole course (including the OpenDSA part!)
- Do labs 1–8
- Do extra questions and bonus problems for a higher grade
- Do the written assignment
- Pass the exam. Planned for December 15th

Course homepage
`http://www.ida.liu.se/~TDDD86/`

1.7

Related course, to go further

TDDD20 Design and Analysis of Algorithms
Introduce greedy algorithms, decomposition and dynamic programming, NP-completness, inexact methods, randomized algorithms, etc.

TDDD38 Advanced Programming in C++
Give a deeper knowledge about constructs and mechanisms in C++. Focus is on advanced constructions and usage of C++.

1.8

2 DALG – introduction

DALG

Data structure
How to efficiently store data

- Theoretically, efficient data structures
- Practically, efficient data structures

Algorithms
How to solve problems efficiently

- Analyse complexity
- Examples of different types of algorithms
  - Sort algorithms
  - Graph algorithms
- Construction methods

1.9

Why study DALG?

Ancient origin, new opportunities

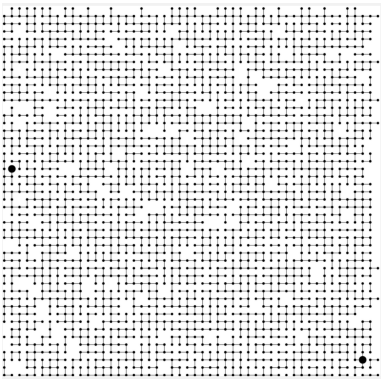
- The study of algorithms has been going on at least since Euclid (fl. 300 BCE)
- Formalized by Church and Turing in the 1930s
- The subject of numerous ongoing researches

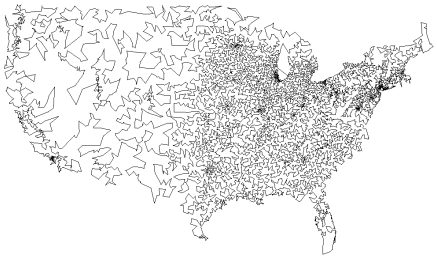
1.10

Why study DALG?

In order to solve otherwise unsolvable problems

- T.ex. network connectivity or the traveller salesperson problems

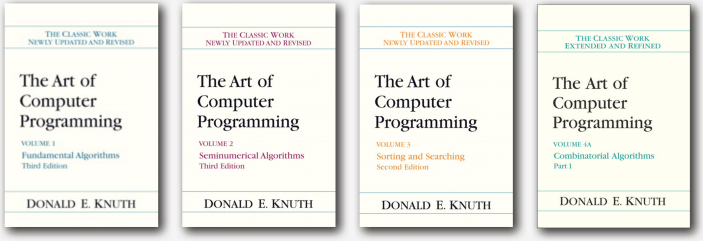




1.11

Why study DALG?

For intellectual stimulation



Donald Knuth
1974 Turing Award

1.12

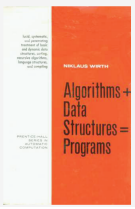
Why study DALG?

To become a proficient programmer

" I will, in fact, claim that the difference between a bad programmer and a good one is whether he considers his code or his data structures more important. Bad programmers worry about the code. Good programmers worry about data structures and their relationships. "

— Linus Torvalds (creator of Linux)

" Algorithms + Data Structures = Programs. " *— Niklaus Wirth*



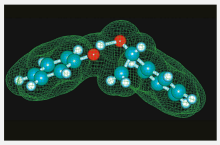
1.13

Why study DALG?

Because it can help us figure out things about life and universe

> *" Computer models mirroring real life have become crucial for most advances made in chemistry today…. Today the computer is just as important a tool for chemists as the test tube. "*
>
> — *Royal Swedish Academy of Sciences (Nobel Prize in Chemistry 2013)*

Martin Karplus, Michael Levitt, and Arieh Warshel



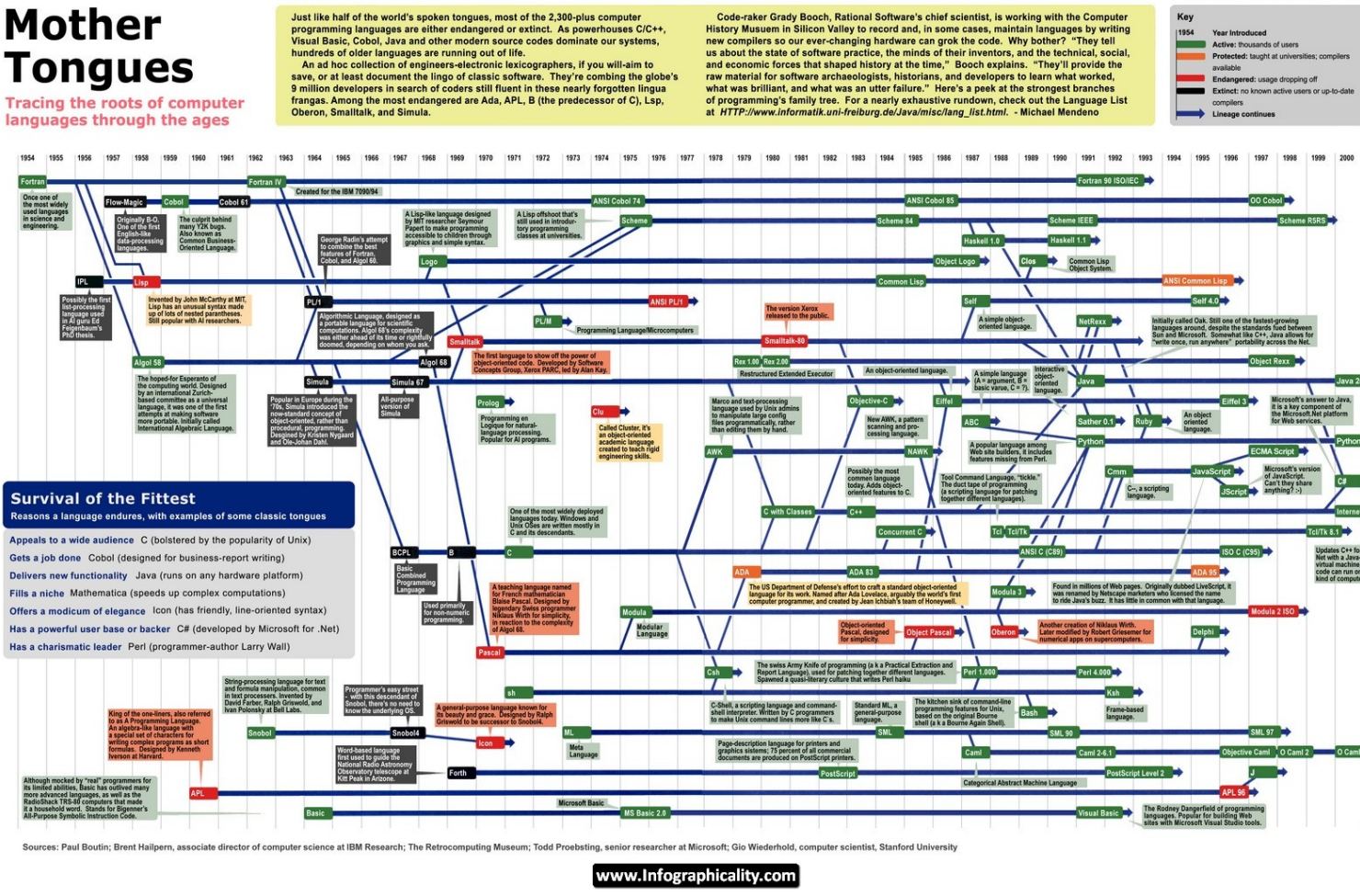
1.14

Why so many programming languages

- Some languages are focused on performance.
- Some languages are focused on making it easy to write code.
- Some languages are focused on performing a particular task extremely well.
- Some languages are focused on testing new concepts.

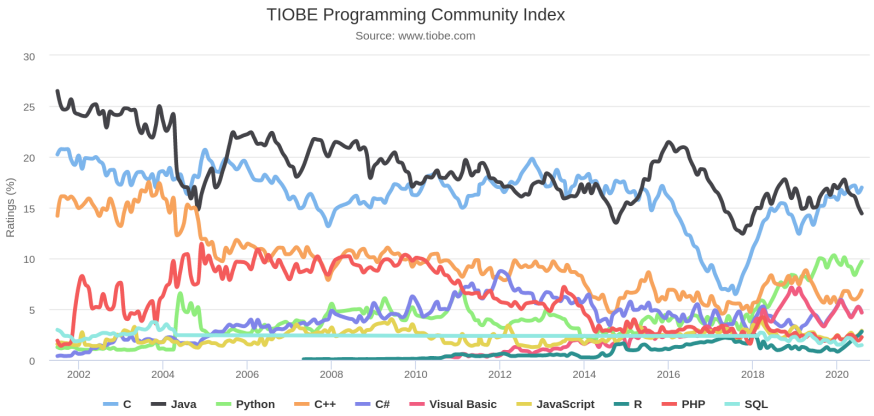
1.15

History



1.16

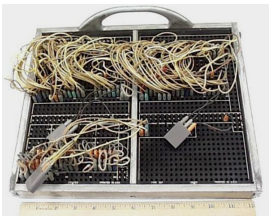
Language popularity



1.17

Origin of programming languages

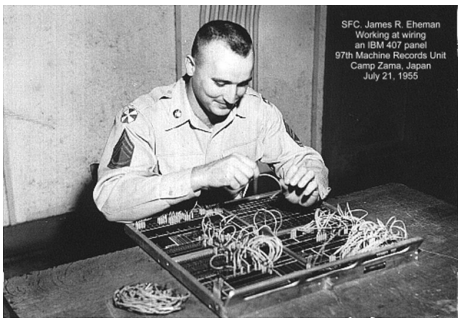
- A programming language is commonly defined as a *way to communicate to the computer what we want to do*.
- Before the mid-1940s, to perform a task it was required to switch the internal cable connections in a computer.
  - Example: IBM 407 Accounting Machine (1949)

  - It would be the equivalent of designing your own chip for each program
- Programming was invented to allow users to solve problems without having to reconfigure the hardware.

1.18

Programming a coupling board



Hmm, should I transfer this parameter by value or by reference?

1.19

Machine language

- John von Neumann suggested that the computer could keep the same cabling defining a set of general operations.
- The operator would input a series of binary codes to organize the basic hardware operations to solve specific problems.

1.20

Assembly

- Assembly language was introduced to provide symbolic abbreviation (“ADD”, “PUSH”...) to represent binary code.
- An improvement but...
  - Lacks abstraction of mathematical notation
  - Each type of hardware architecture has its own set of machine instructions, and therefore requires its own assembly dialect.
- Assembly language appeared in the 1950s and is still used today for machine-oriented tools and hand optimization.

1.21

Algol

- Algol: Algorithmic Language, released in 1960.
- Structured control statements
  - sequence
  - loops (for)
  - selection (if-else)
- Different numeric types
- Introduce array/field
- Supports procedure (and recursion)

```
procedure Absmax(a) Size:(n, m) Result:(y) Subscripts:(i, k);
value n, m; array a; integer n, m, i, k; real y;
comment The absolute greatest element of the matrix a, of size n by m
is transferred to y, and the subscripts of this element to i and k;
begin
integer p, q;
y := 0; i := k := 1;
for p := 1 step 1 until n do
for q := 1 step 1 until m do
if abs(a[p, q]) > y then
begin y := abs(a[p, q]);
i := p; k := q
end
end Absmax
```

1.22

4 Programming paradigms

Programming Paradigm?

- Programming: define and communicate to the computer what we want to do
- Paradigms: patterns, worldviews

Programming Paradigm: **conceptual way of looking at how to describe a programming language**

1.23

Example 1: Imperative Programming

First do A, then do B, then do C, etc... For example:

```
A: read integer n
B: set m = n*n
C: print m
D: ...
```

1.24

Example 2: Object-Oriented Programming

Class A, B, C has data and method.

```
class A {
  function read() ...
}
class B {
  function square() ...
}
class C {
  function print() ...
}
```

1.25

Declarative Programming

Describes what a program should output.

```
List all the students in the TDDD86 course.
```

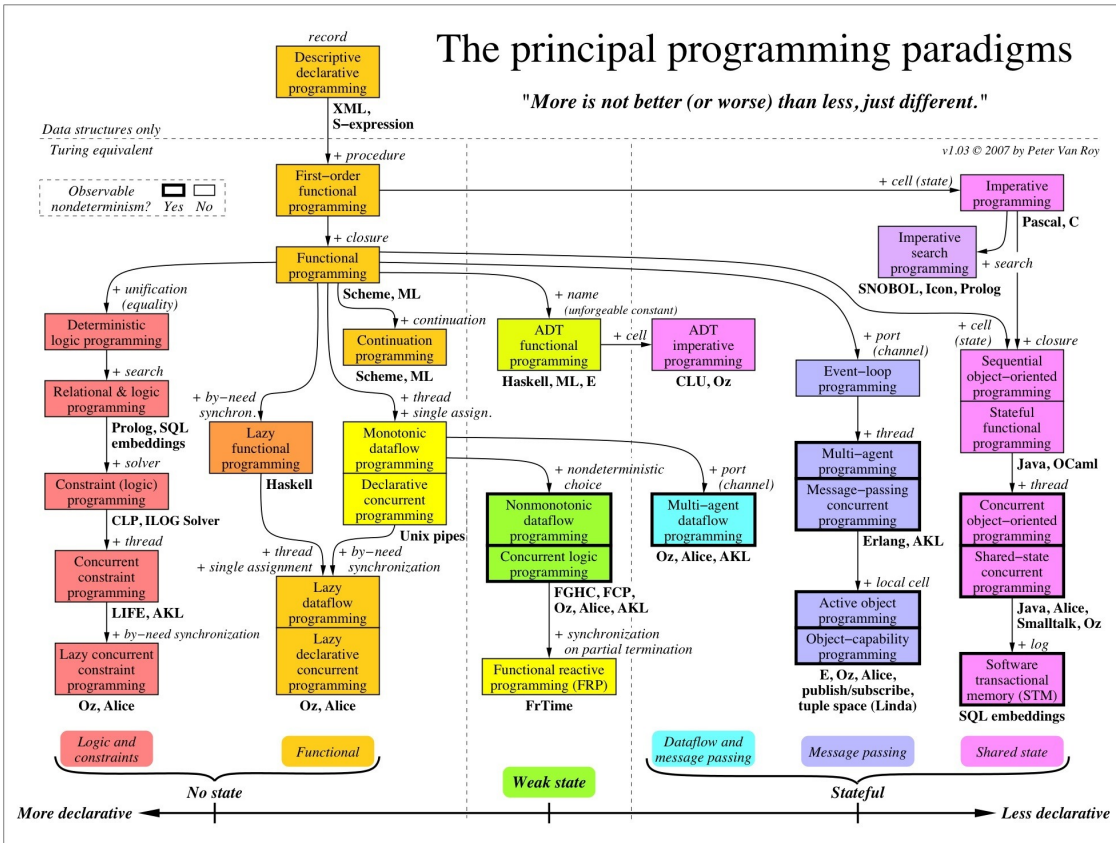
1.26

Programming Paradigm

Programming Paradigm: **conceptual way of looking at how to describe a programming language**

- A programming paradigm is a set of programming concepts.
- Examples of concepts:
  - threads / parallelism
  - items (object orientation)

1.27



1.28

Important Concept: State

In **Imperative Programming** a program has a **state** and step-by-step **instruction** to manipulate the state.

So far, the programming language you have used at LiU have been imperative: Python, Java, assembly.

Contrast: in **declarative programming** the code defines the end result and not how it should be achieved.

1.29

Important Concept: State

In **imperative programming** there are variables that represent the application state (along with the implicit variables, such as instruction pointer).

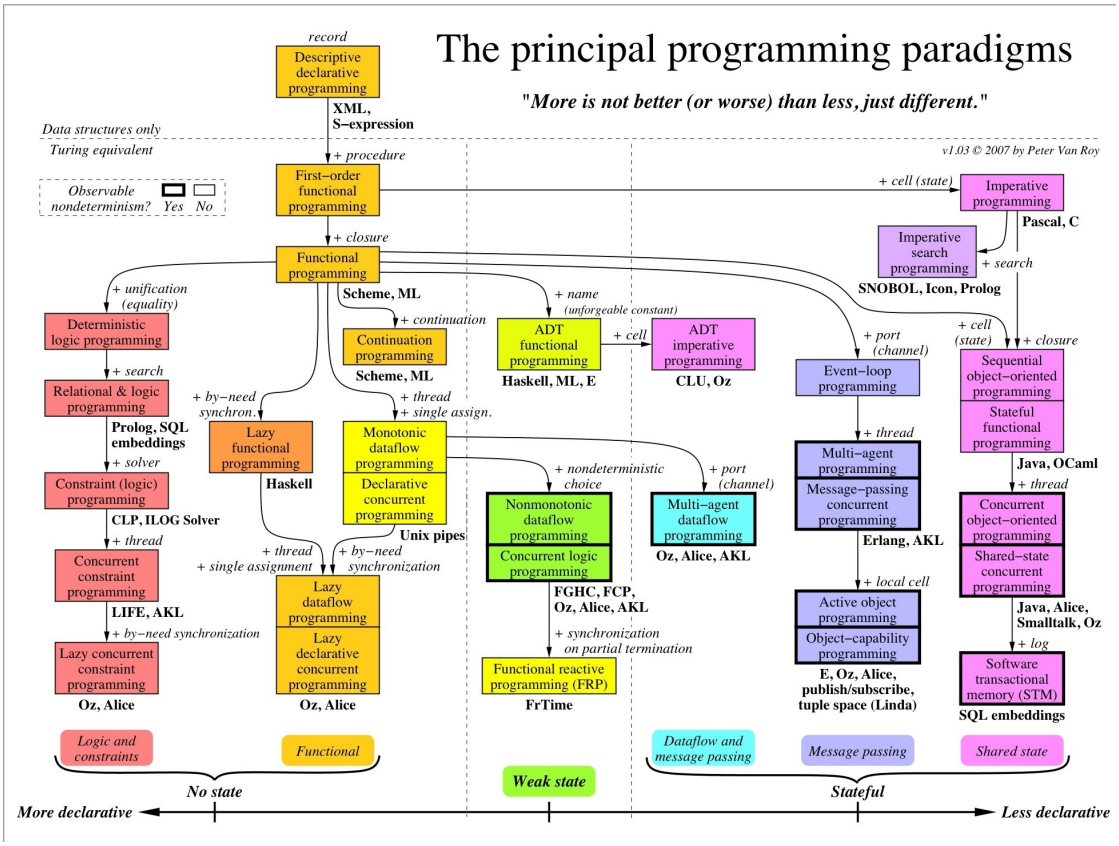
The instructions change the variables and affect the application state. The instructions have **side-effects**!

In **declarative programming**, you define what things are and there is no modification of the state.

The instructions have **no side-effects**!

It is a fundamental different way of thinking about computations compared to imperative programming.

1.30



$$\text{sort}(L) = \begin{cases} L & \text{if len}(L) \leq 1 \\ \text{merge}(\text{sort}(\text{firsthalf}(L)), \\ \qquad \text{sort}(\text{secondhalf}(L))) & \text{otherwise} \end{cases}$$

$$\text{merge}(L_1, L_2) =$$

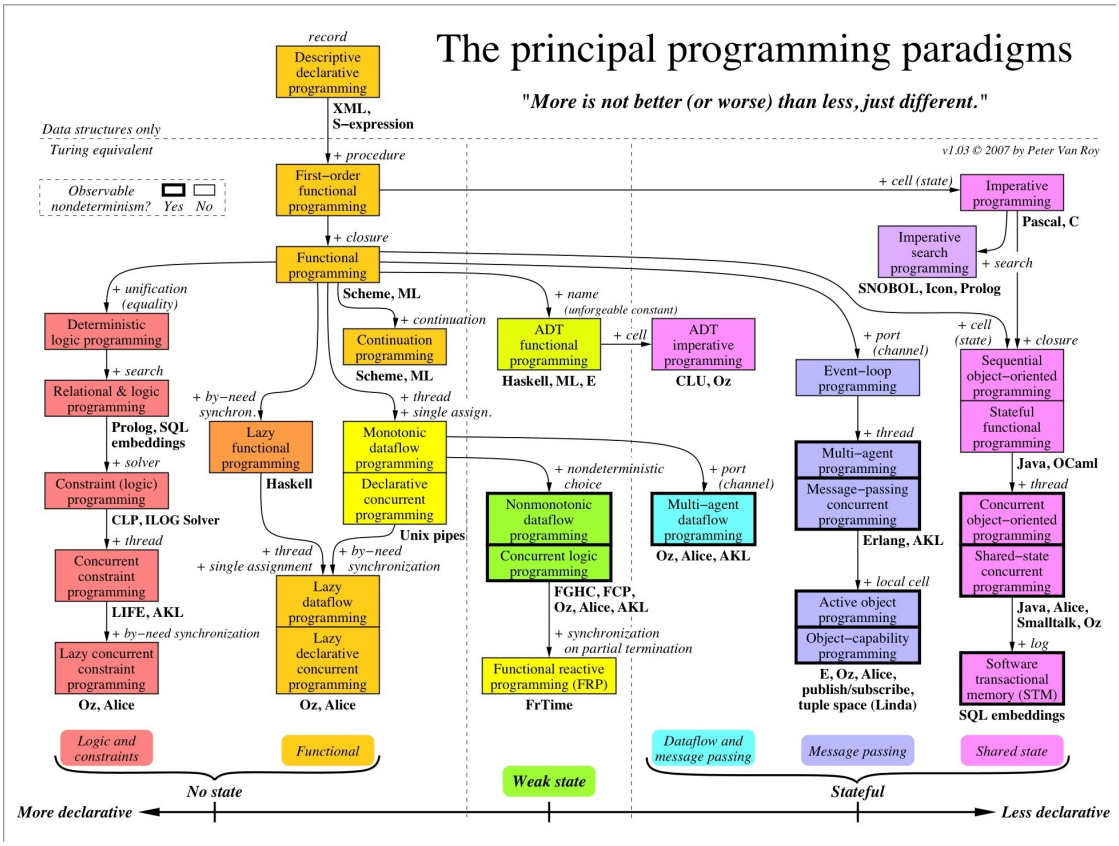
$$\begin{cases} L_1 & \text{if length}(L_2) = 0 \\ L_2 & \text{if length}(L_1) = 0 \\ \text{head}(L_1) | \text{merge}(\text{tail}(L_1), L_2) & \text{if head}(L_1) \leq \text{head}(L_2) \\ \text{head}(L_2) | \text{merge}(L_1, \text{tail}(L_2)) & \text{if head}(L_1) > \text{head}(L_2) \end{cases}$$

$\text{sort}([5,12,43,1])$ returns $[1,5,12,43]$

1.31

Logic programming

A type of declarative programming where the result of the calculations are specified as formal logic. Most common is first-order predicate logic.



Defines a predicate sort(L,S) is true if S is sorted list for L.
sort(L,S) = sorted(S) **and** permutation(L,S)
sort$([5,12,43,1],S)$ = **true**
Means that S must be $[1,5,12,43]$.

1.32

Other important paradigms

Object oriented programming:

- Object, class, inheritance, polymorphism...

Parallel programming:

- Thread, synchronisation, etc...

Distributed calculations:

- Calculation spread on different computers (e.g., cloud computing)

1.33

Different Paradigms = Different Strengths

Rule of thumb: the more declarative language, the further from the hardware
In the end, the program will run on the same hardware, regardless of which language it is written in.
Declarative language make it easier to express complex calculations.
...but they are harder to execute on a machine.
Imperative language to execute faster...
...but they require more code.

1.34

Why do you need to know all this?

> "It is essential for anyone who wants to be considered a professional in the areas of software to know several languages and several programming paradigms."
>
> – Bjarne Stroustrup

1.35

Why do you need to know all this?

- You are here (in the computer science/software engineering program at LiU) to become computer scientists, not to teach you to write simple programs in the < latest fashionable language >.
- Better understanding of programming and algorithms by getting multiple perspectives on how to think about programming
- The right tool for the right projects - different languages and paradigms are good at different things

1.36

5 Programming languages concepts

Foundation of programming languages

- Programming languages have many similarities to natural languages
  - There are rules of syntax and semantics, there are many dialects, etc...
- Lets have a look at a few concepts:
  - Compiled/interpreted
  - Syntax
  - Semantics
  - Typing

1.37

Compiled vs interpreted

Compiled languages translate into machine code that can be run directly on a computer CPU.

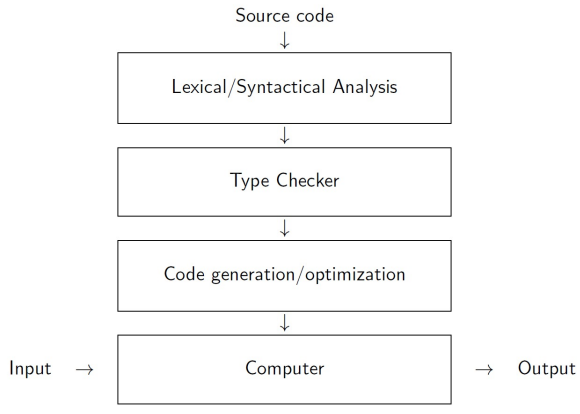
- The entire program is usually translated before running

Interpreted languages are processed by a virtual machine on a higher level

- Usually the program is transformed while running, batches-by-batches when it is needed

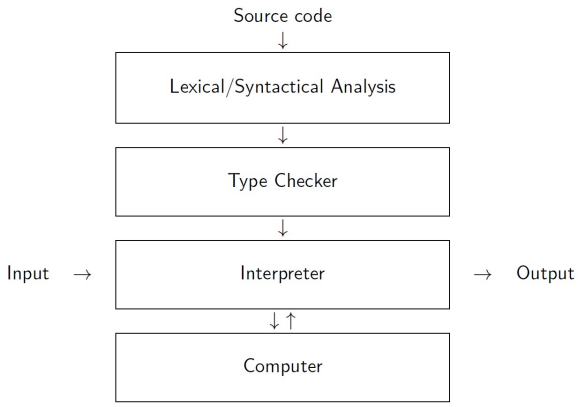
1.38

Compiled language



1.39

Interpreted language



1.40

Syntax

A language syntax describes how well-formed expression should look like:

- How to assemble symbols to form units
- How to assemble units to from expressions
- The formalism to describe a language's syntax is usually called a **grammar**

1.41

Semantic

The semantic is an important aspect of a (programming) language:

- Semantic is about the meaning of constructions.
- More difficult to define than the syntax
- A programmer should be able to predict what happens before the program run
- It is a precise description of what different **constructions** of a language means

Different approaches:

- Operational semantic:
  - Describes directly the execution of the statements of a language, for exemple by describing the transitions of an abstract machine.
- Axiomatic semantic:
  - Defines the meaning of commands in term of logical predicates and axioms on the program state, for example Hoare logic.
- Denotational semantic:
  - Uses mathematical objects, e.g., partial functions, to denote what programs do.

1.42

Type

- A program needs to handle data
- structures and mechanisms for doing this are called **type systems**
- types help with
  - program design
  - check correctness
  - determine storage requirements

1.43

Type system

- A program needs to handle data
- Types provide a specification for data
- Structures and mechanisms to handle types is called **type system**
- Type system usually includes
  - a set of predefined types
  - a mechanism to create new types
  - a mechanism to control types
    - When are two types the same?
    - When should a type replace the other?
    - What is the type of a compound term?
  - Rules for control: static/dynamic

1.44

Typing

- A language is said to be **typed** when it requires specification of the type of data when defining an operation or a variable
- Assembly languages are usually **untyped**
  - In assembly all data is represented as bytes array

1.45

Weak and strong typing

- There is a distinction between **weak typing** and **strong typing**
- With **strong typing**, the language will not allow operation on the wrong type of data
  - adding an integer to a float
- With **weak typing**, the language will perform implicit type conversion, i.e. a type will be interpreted as an other (for)
  - when adding an integer to a float, the integer is converted to a float
  - when adding an integer to a string, the integer is converted to a string

1.46

Static vs dynamic type control

- There is a distinction on when the types are checked
- With **statically types languages**, the types are checked before the program runs
- With **dynamic types languages**, the types are checked while the program is running

1.47