TB141Ic – ICT System Engineering and Rapid Prototyping

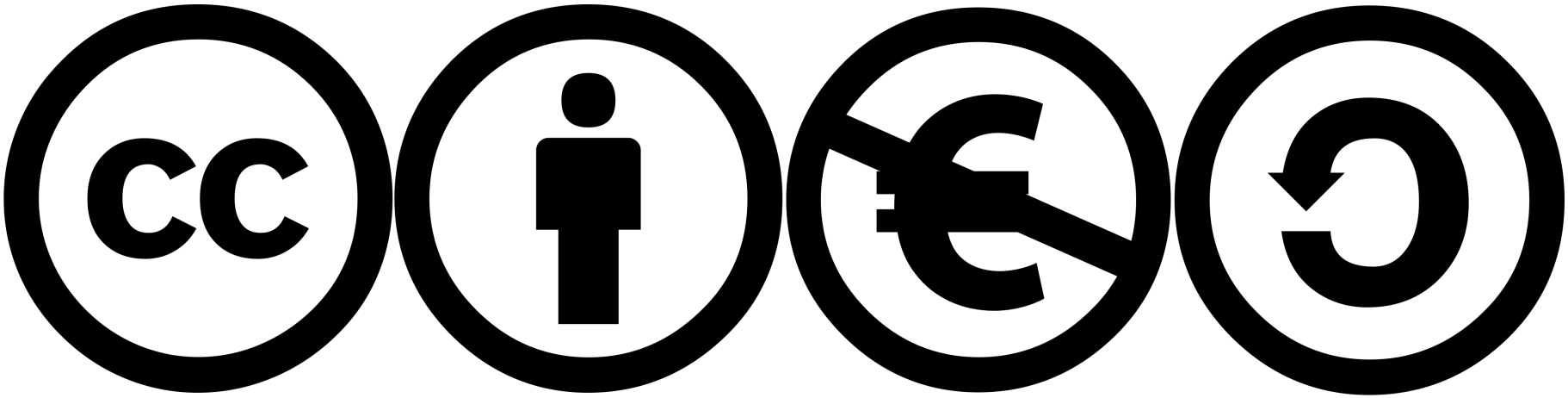
Programming Languages

Jacopo De Stefani, Ph.D.

Delft University of Technology, The Netherlands

28/02/2022

Licensing information



Learning objectives and related literature

Learning objectives

- Define a programming language
- Recognize the differences between compiled and interpreted languages.
- Recall the different taxonomies for programming languages
- Recognize and categorize the most common programming languages
- Recognize the most common programming paradigms.

Related literature

- Based on https://cs.lmu.edu/ ray/notes/pltypes/
- Wikipedia (see references)

Programming Languages Basics

What is a programming language?

Definition

A programming language is any set of rules that converts strings, or graphical program elements in the case of visual programming languages, to various kinds of machine code output.

What is a programming language?

Definition

A programming language is any **set of rules** that converts strings, or graphical program elements in the case of visual programming languages, to various kinds of machine code output.

- **Syntax** ~ Grammatical rules

What is a programming language?

Definition

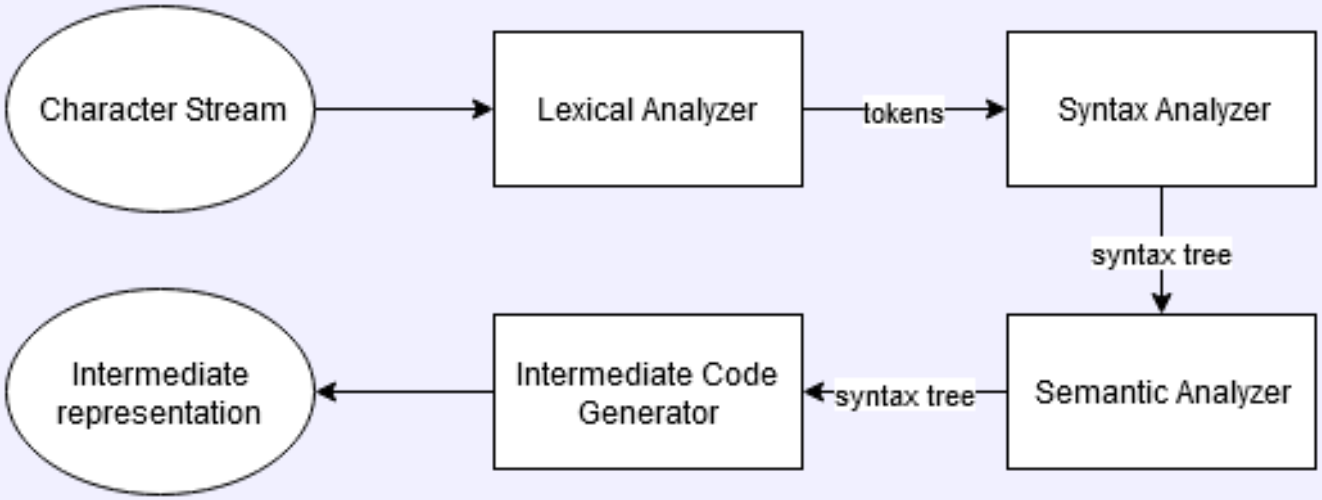
A programming language is any set of rules that **converts strings, or graphical program elements** in the case of visual programming languages, **to various kinds of machine code output**.

- **Semantics** ~ Meaning

Example - Programming Language

```
1     import random
2     n = random.randint(1, 99)
3     guess = int(raw_input("Enter an integer from 1 to 99:
      "))
4     while n != "guess":
5     print()
6     if guess < n:
7     print("guess is low")
8     guess = int(raw_input("Enter an integer from 1 to 99:
      "))
9     elif guess > n:
10    print("guess is high")
11    guess = int(raw_input("Enter an integer from 1 to 99:
      "))
12    else:
13    print ("you guessed it!")
14    break
15
```

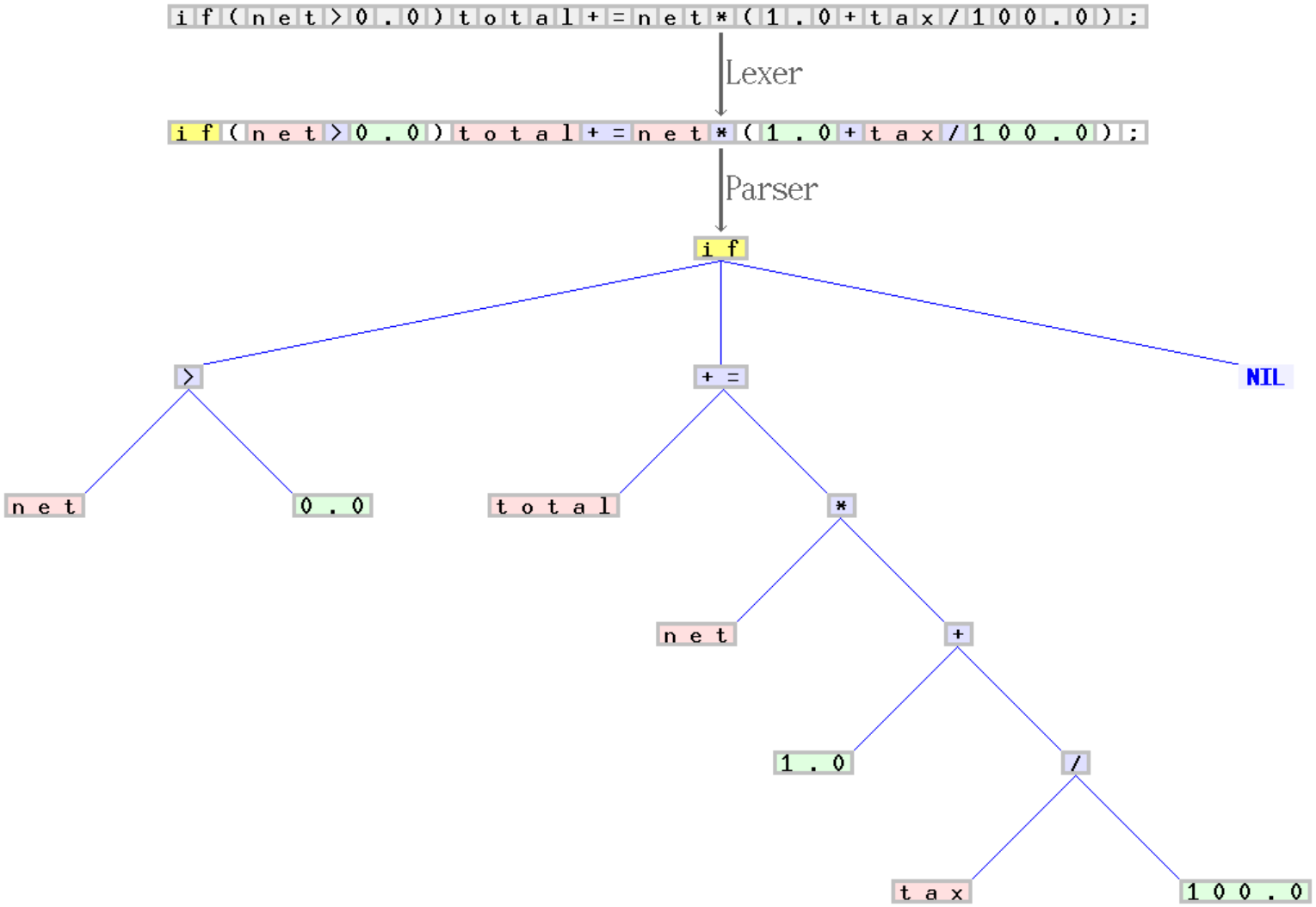
Compiler - Front End



Source: Compiler vs Interpreter - Vervuka-dev

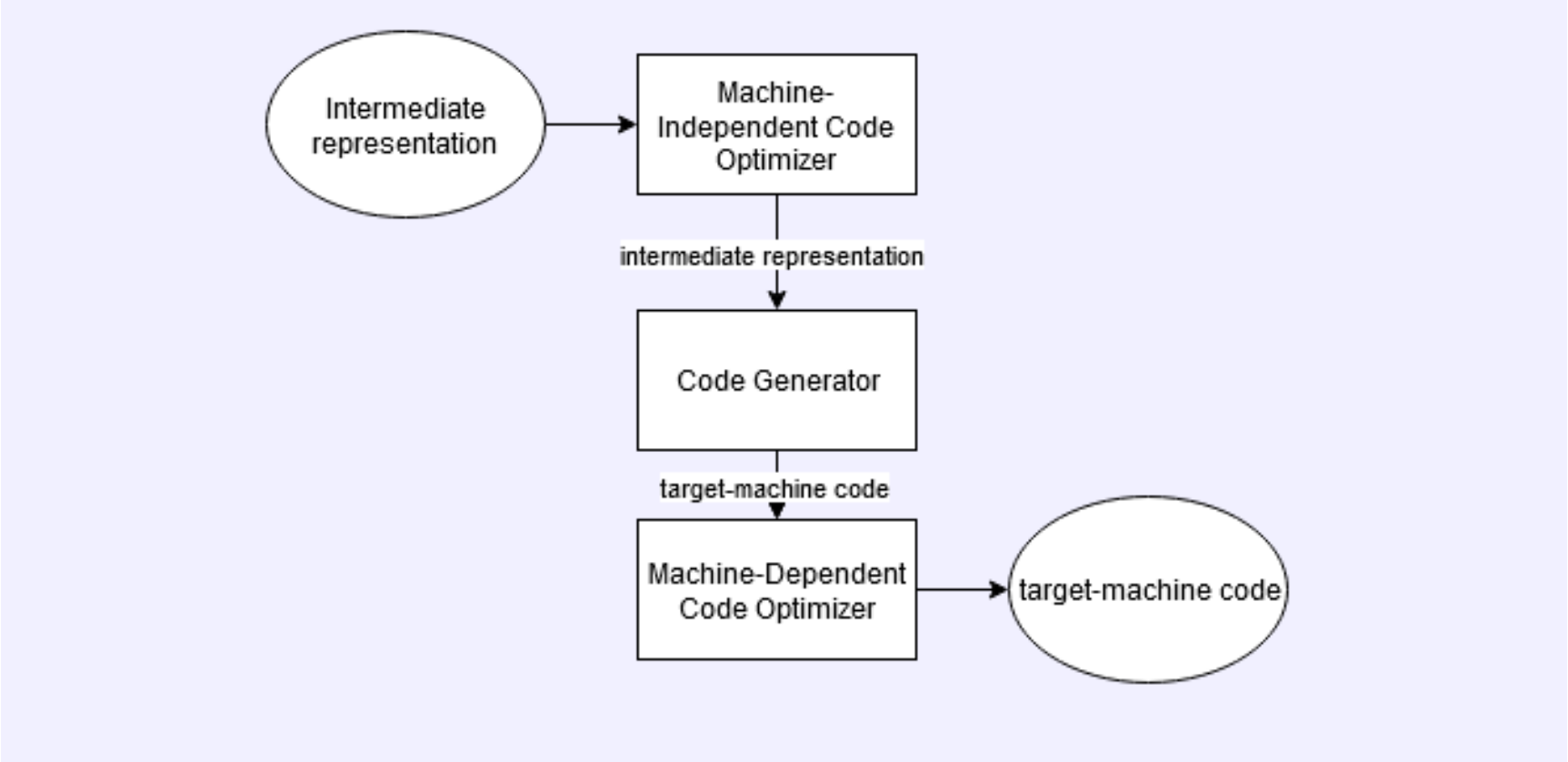
- **Lexical Analyzer** - does lexical analyze, so it divides input program(character stream) to tokens (they are just meaningful parts, for example each variable name is a token)
- **Syntax Analyzer** - it analyzes syntax, usually it uses context-free grammar definition of a language and checks if input fits to it.
- **Semantic Analyzer** - it analyzes if meaning of input is correct also can checks types. For example it can check if after every if there is a boolean expression.
- **Intermediate Code Generator** - generates some abstraction of input code.

Compiler - Lexical Analyzer example



Source: Wikimedia

Compiler - Back End

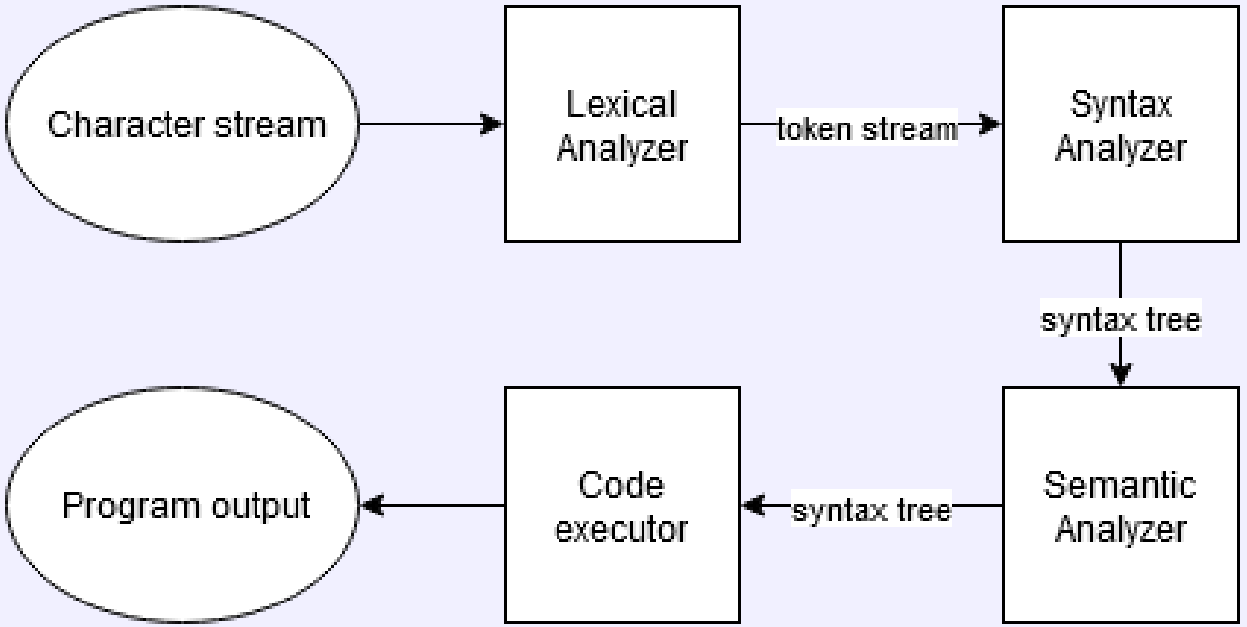


Source: Compiler vs Interpreter - Vervuka-dev

Backend of a compiler is focused on generating source code in target programming language along with some optimizations. Also it is responsible for making code to fit to a desired architecture.

It starts from intermediate representation of a code from a frontend part of a compiler, then it does optimizations on this abstract code, generates code in target language and again does some optimizations.

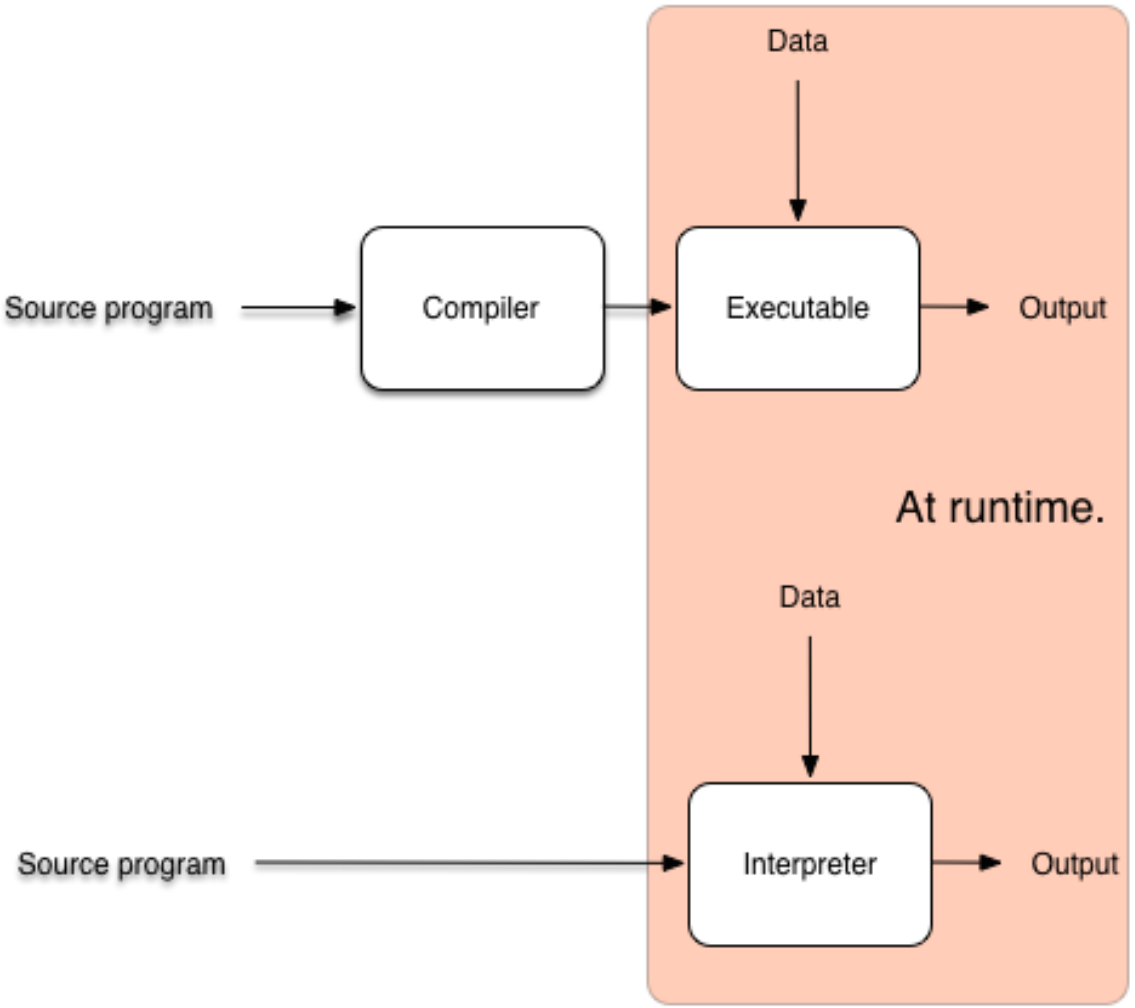
Interpreter



Source: Compiler vs Interpreter - Vervuka-dev

Interpreter executes a program line/block by line/block. It means it executes instructions almost directly. It has to store the actual state of execution (like variables) in its memory (usually it uses some virtual machine).

Compiler vs Interpreter



Source: JIT, AOT, Interpretation: What are differences between them?

Compiler vs Interpreter

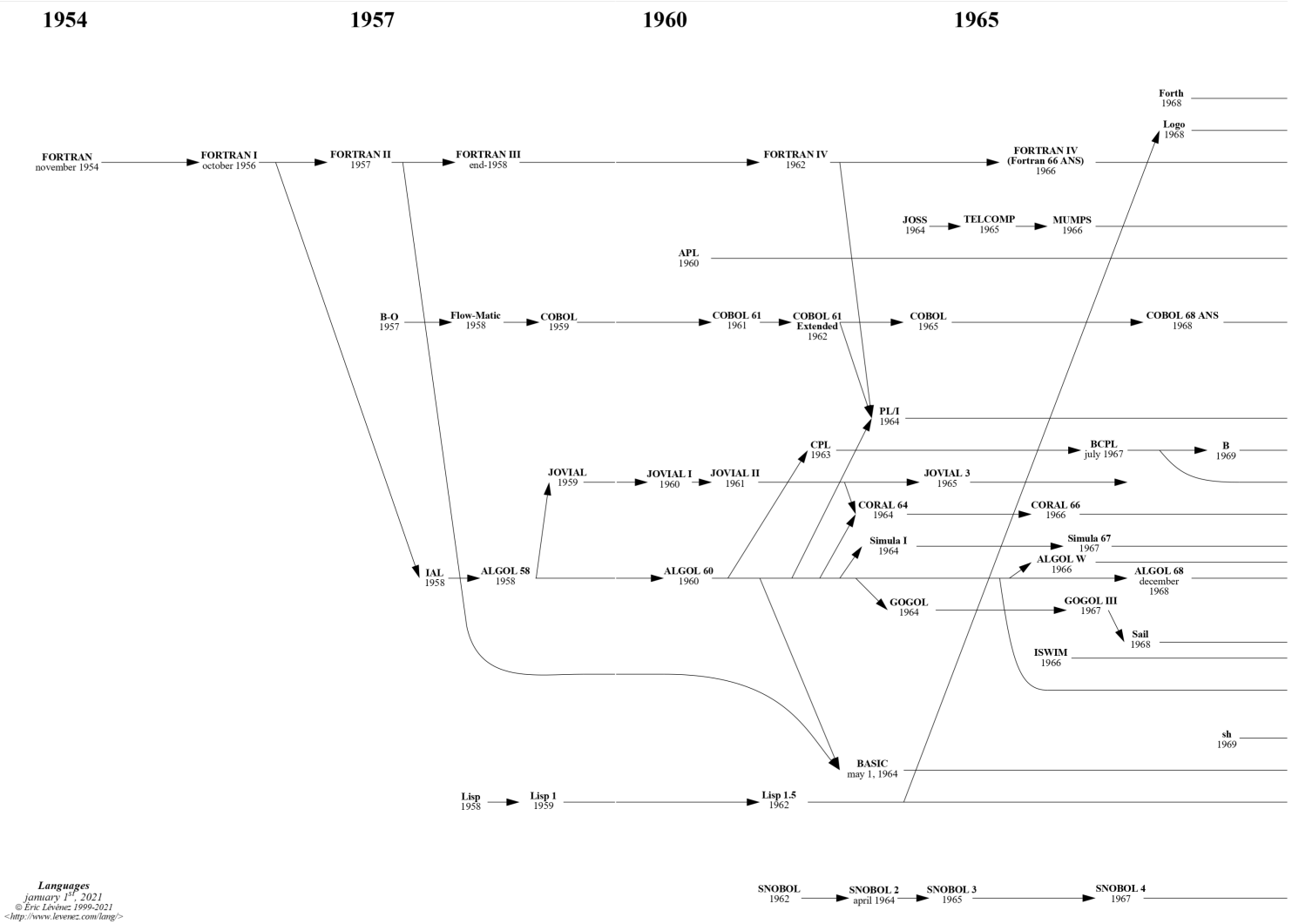
Criteria	Compiler	Interpreter
Speed	Fast, creates executable file that runs directly on the CPU	Slower, interprets code one line at a time
Debug	Debugging is more difficult. One error can produce many spurious errors	Debugging is easier. Each line of code is analysed and checked before being executed
Fault-tolerance	More likely to crash the computer. The machine code is running directly on the CPU	Less likely to crash as the instructions are being carried out either on the interpreters' command line or within a virtual machine environment which is protecting the computer from being directly accessed by the code.
IP Protection	Easier to protect Intellectual Property as the machine code is difficult to understand	Weaker Intellectual property as the source code (or bytecode) has to be available at run time. For example if you write a Flash Actionscript application, you can easily get de-compilers that convert the p-code back into actionscript source code (unless you use encryption, but that is another story).
Memory usage	Uses more memory - all the execution code needs to be loaded into memory, although tricks like Dynamic Link Libraries lessen this problem	Uses less memory, source code only has to be present one line at a time in memory
Safety	Unauthorised modification to the code more difficult. The executable is in the form of machine code. So it is difficult to understand program flow.	Easier to modify as the instructions are at a high level and so the program flow is easier to understand and modify.

Programming Languages Classification

Programming Languages Classification

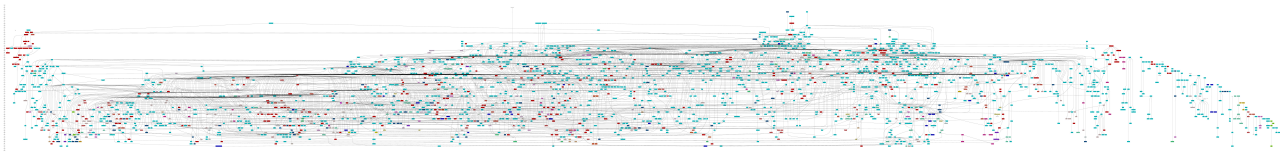
- Many different criteria exist to classify programming languages, based both on external (such as name and year of development) and internal properties (such as textual/graphical, memory management).
- The Online Historical Encyclopedia for Programming languages proposed a formal **taxonomy**
- However, with 8945 different programming languages and exact taxonomical approach becomes very complicated to use.
- For our purposes:
  - **Alphabetical:** By alphabetical ordering of the language names
  - **Categorical:** By intrinsic properties of the language
  - **Chronological:** By chronological order of invention
  - **Generational:** By genealogy of the languages

Programming Languages Classification - Chronological



Source: Eric Levenez

Programming Languages Classification - Generational



Source: Online Historical Encyclopaedia of Programming Languages

Programming Languages Classification - Categorical

Technical aspects of languages will consider linguistic structure, expressive features, possibility of efficient implementation, direct support for certain programming models, and similar concerns.

- **Machine languages**, that are interpreted directly in hardware
- **Assembly languages**, that are thin wrappers over a corresponding machine language
- **High-level languages**, that are anything machine-independent
- **System languages**, that are designed for writing low-level tasks, like memory and process management
- **Scripting languages**, that are generally extremely high-level and powerful
- **Visual languages:** that are non-text based
- **Esoteric languages:** that are not really intended to be used, but are very interesting, funny, or educational in some way

Machine code

Machine language is the direct representation of the code and data run directly by a computing device. Machine languages feature:

- Registers to store values and intermediate results
- Very low-level machine instructions (add, sub, div, sqrt) which operate on these registers and/or memory
- Labels and conditional jumps to express control flow
- A lack of memory management support — programmers do that themselves

The machine instructions are carried out in the hardware of the machine, so machine code is by definition machine-dependent. Different machines have different instruction sets. The instructions and their operands are all just bits.

```
1    89 F8 A9 01 00 00 00 75 06 6B C0 03 FF C0 C3 C1 E0 02
    83 E8 03 C3
2
```

Assembly code

- An assembly language is an encoding of machine code into something more readable.
- It assigns human-readable labels (or names) to storage locations, jump targets, and subroutine starting addresses.
- Isomorphic to its machine language.

```
1   .globl  f
2   .text
3   f:
4     mov     %edi, %eax        # Put first parameter into eax register
5     test    $1, %eax          # Examine least significant bit
6     jnz     odd               # If it's not a zero, jump to odd
7     imul    $3, %eax          # It's even, so multiply it by 3
8     inc     %eax              # and add 1
9     ret                       # and return it
10  odd:
11    shl     $2, %eax          # It's odd, so multiply by 4
12    sub     $3, %eax          # and subtract 3
13    ret                       # and return it
14
```

High level language

A high-level language gets away from all the constraints of a particular machine. HLLs may have features such as:

- Names for almost everything: variables, types, subroutines, constants, modules
- Complex expressions (e.g. `2 * (y**5) >= 88 && sqrt(4.8) / 2 % 3 == 9`)
- Control structures (conditionals, switches, loops)
- Composite types (arrays, structs)
- Type declarations
- Type checking
- Easy, often implicit, ways to manage storage (global, local and heap storage)
- Subroutines with their own private scope
- Abstract data types, modules, packages, classes
- Exceptions

High level language - Examples

C

```c
int f(const int n) {
  return (n % 2 == 0) ? 3 * n + 1 : 4 * n - 3;
}

```

Java

```java
class
ThingThatHoldsTheFunctionUsedInTheExampleOnThisPage {
  public static int f(int n) {
    return (n % 2 == 0) ? 3 * n + 1 : 4 * n - 3;
  }
}

```

Python

```python
def f(n):
  return 3 * n + 1 if n % 2 == 0 else 4 * n - 3

```

System language

- System software is computer software designed to operate and control the computer hardware, and to provide a platform for running application software.
- System software includes software categories such as operating systems, utility software, device drivers, compilers, and linkers.
- The system programming languages are for low level tasks like memory management or task management.
- A system programming language usually refers to a programming language used for system programming
- Writing system software, which usually requires different development approaches when compared with application software.

Examples of system languages include:

- Ada
- C/C++
- Rust

Scripting language

- In a traditional sense, scripting languages are designed to automate frequently used tasks that usually involve calling or passing commands to external programs.
- Many complex application programs provide built-in languages that let users automate tasks.
- Recently, many applications have built-in traditional scripting languages, such as Perl or Visual Basic, but there are quite a few native scripting languages still in use.
- Many scripting languages are compiled to bytecode and then this (usually) platform-independent bytecode is run through a virtual machine (compare to Java virtual machine).

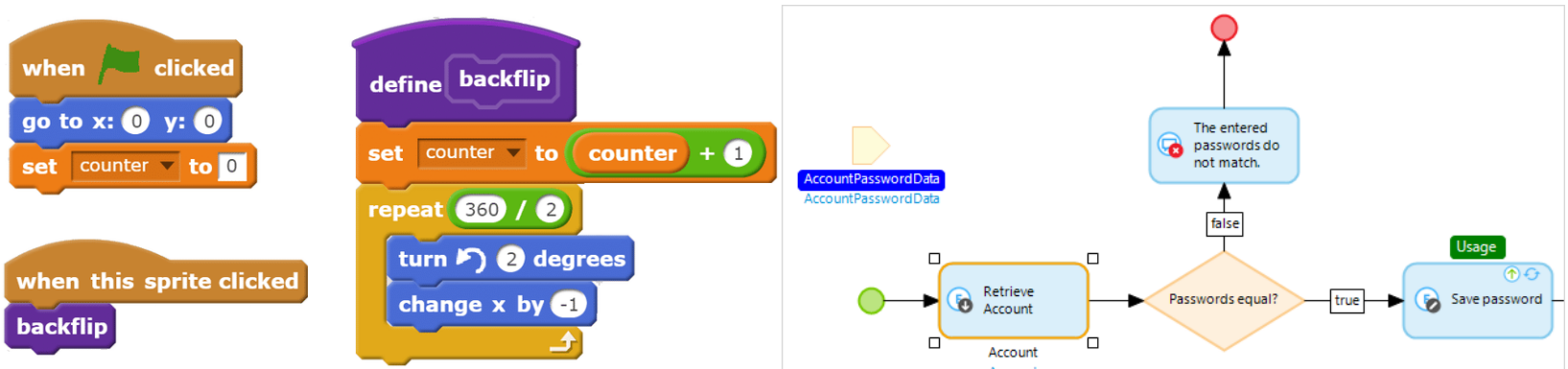
Examples include:

- Bash
- Zsh
- PowerShell
- Python
- R

Visual programming language

Visual programming languages let users specify programs in a two-(or more)-dimensional way, instead of as one-dimensional text strings, via graphic layouts of various types.

Examples include:



Scratch **Mendix**

Esoteric programming language

An esoteric programming language is a programming language designed as a test of the boundaries of computer programming language design, as a proof of concept, or as a joke.
Examples include:

Brainfuck

```
++++++++[>++++[>++>+++>+++>+<<<<-]>+>+>->>+[<]<-]>>.>---
.+++++++..+++.>>.<-.<.+++.------.--------.>>+.>++.
```

LOLCODE

```
HAI 1.2
CAN HAS STDIO?
VISIBLE "HAI WORLD!"
KTHXBYE
```

Malebolge

```
(=<`#9]~6ZY327Uv4-QsqpMn&+Ij"'E%e{Ab~w=_:]Kw%o44Uqp0/Q?xNvL:`H%c#DD2^WV>gY;
dts76qKJImZkj
```


Break

Programming Paradigms

Programming paradignms

Programming paradigms are a way to classify programming languages based on their features. Languages can be classified into multiple paradigms:

- **Imperative:** in which the programmer instructs the machine how to change its state. Examples are:
  - **Procedural** which groups instructions into procedures,
  - **Object-oriented** which groups instructions with the part of the state they operate on,
- **Declarative:** in which the programmer merely declares properties of the desired result, but not how to compute it. Examples are:
  - **Functional** in which the desired result is declared as the value of a series of function applications,
  - **Logic** in which the desired result is declared as the answer to a question about a system of facts and rules,

Imperative paradigm

Control how in imperative programming is explicit: commands show how the computation takes place, step by step. Each step affects the global state of the computation.

```
result = []
i = 0
start:
  numPeople = length(people)
  if i >= numPeople goto finished
  p = people[i]
  nameLength = length(p.name)
  if nameLength <= 5 goto nextOne
  upperName = toUpper(p.name)
  addToList(result, upperName)
nextOne:
  i = i + 1
  goto start
finished:
  return sort(result)
```

Structured paradigm

Structured programming is a kind of imperative programming where control flow is defined by nested loops, conditionals, and subroutines, rather than via gotos. Variables are generally local to blocks (have lexical scope).

```
result = [];
for i = 0; i < length(people); i++ {
  p = people[i];
  if length(p.name)) > 5 {
    addToList(result, toUpper(p.name));
  }
}
return sort(result);
```

Object-oriented paradigm

OOP is based on the sending of messages to objects. Objects respond to messages by performing operations, generally called methods . Messages can have arguments. A society of objects, each with their own local memory and own set of operations has a different feel than the monolithic processor and single shared memory feel of non object oriented languages.

Pure OOP - Smalltalk

```
^people filter: [:p | p name length greaterThan: 5] map:
  [:p | p name upper] sort

```

Hybrid - Java

```
result = []
for p in people {
  if p.name.length > 5 {
    result.add(p.name.toUpper);
  }
}
return result.sort;
```

Declarative paradigm

Control flow in declarative programming is implicit: the programmer states only what the result should look like, not how to obtain it

SQL

```
1    SELECT upper(name)
2    FROM people
3    WHERE length(name) > 5
4    ORDER by name
```

Functional paradigm

In functional programming , control flow is expressed by combining function calls, rather than by assigning values to variables:

In functional programming:

- There are no commands, only side-effect free expressions
- Code is much shorter, less error-prone, and much easier to prove correct
- There is more inherent parallelism, so good compilers can produce faster code

Haskell-like

```
people |> map (p => to_upper (name p)) |> filter (s =>
  length s > 5) |> sort
```

List comprehensions are an example of functional programming that combine map (apply the same operation) and filter (select based on a condition):

```
    L = [isdigit(str(i)) for i in range(100) if i % 2 ==
    0]
```

Logic and constraint paradigm

Logic programming and constraint programming are two paradigms in which programs are built by setting up relations that specify facts and inference rules, and asking whether or not something is true (i.e. specifying a goal .) Unification and backtracking to end solutions (i.e.. satisfy goals) takes place automatically.

Languages and Paradigms

- One of the characteristics of a language is its support for particular programming paradigms.
- Very few languages are pure (i.e implement a paradigm 100%.)
  - A lot of languages have a few escapes; for example in OCaml, you will program with functions 90% or more of the time, but if you need state, you can get it.
- If a language is purposely designed to allow programming in many paradigms is called a multi-paradigm language.
  - **Example:** In Scala you can do imperative, object-oriented, and functional programming quite easily.

Other relevant paradigms/languages

Literate programming

Literate programming is a programming paradigm introduced by Donald Knuth in which a computer program is given an explanation of its logic in a natural language, such as English, interspersed with snippets of macros and traditional source code, from which compilable source code can be generated.

- The approach is used in scientific computing and in data science routinely for reproducible research and open access purposes.
- Literate programming tools are widely employed today.
- Examples include:
  - Jupyter Notebooks
  - Observable
  - Maple
  - Sweave/KnitR

Is HTML a programming language?

Markup languages

Definition

A markup language is a set of rules governing what markup information may be included in a document and how it is combined with the content of the document in a way to facilitate use by humans and computer programs.

- Markup refers to data included in an electronic document which is distinct from the document's content.
- It is typically not included in representations of the document for end users.
- Markup is often used to control the display of the document or to enrich its content to facilitate automated processing.
- The idea and terminology evolved from the "marking up" of paper manuscripts (i.e., the revision/instructions by editors)

Key points

- A programming language is a set of rules that converts high-level descriptions (text/graphical) to various kinds of machine code output.
- Two main process exist to make the translation: Compilation and Interpretation
  - In compilation, a compiler transforms the whole high-level representation through one or more intermediate representations to create executable machine code.
  - The compilation and execution are two distinct phases.
  - In interpreting, the high-level interpretation is read element-by-element by an interpreter which executes the code.

Key points

- Several different categorizations exists to classify programming languages.
- In practice, the most used categorizations deal with the levels of abstraction of the code (categorical) and the programming "style" (paradigms).
- Other forms of languages (e.g. literate programming and markup/markdown) are commonly used in the everyday IT-practice.
- Language processing systems are used to translate texts from one language into another and to carry out the instructions specified in the input language. They include a translator and an abstract machine that executes the generated language.