Programming Languages

Ege Emir Ozkan

Chapter 1

Languages and Creation

1.1 Types of Programming Languages

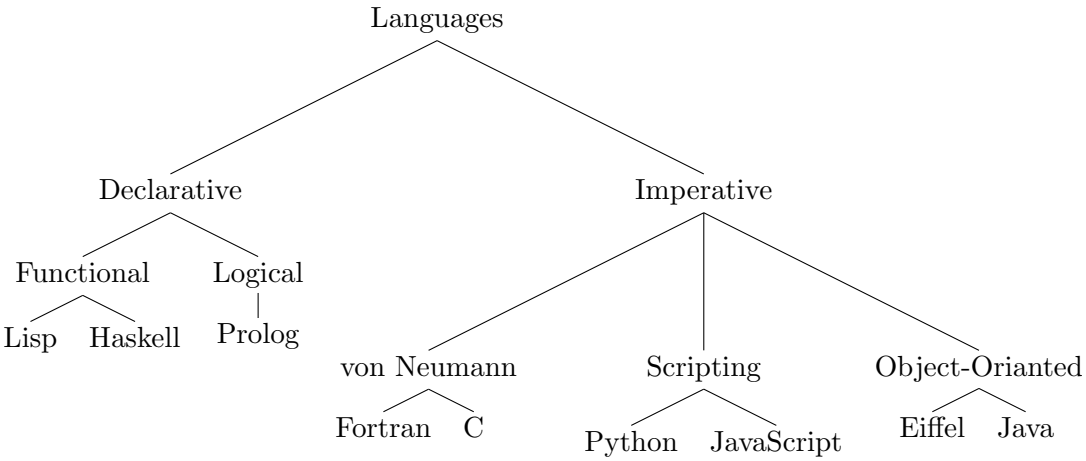
1.1.1 Why are there so many programming languages?

- Evolution of language constructs, such as the evolution of goto based control flows to loops to nested block struuctures to object-oriantation opens new possibilities to discover.
- Special purposes, languages for scientific computing, embedding systems programming, data science.
- Subjective ideas about what is nice to use.

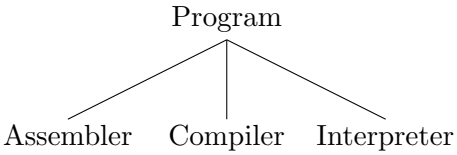
1.1.2 What makes a language successful?

- Expressive power of the language.
- Ease of use for the beginners (Python)
- Ease of implementation, easy to write a compiler/interpreter for.
- Standardization (Java standard)
- Open source (Java, Python)
- Good compilers (Rust, Go)
- Economic backing, patronage (Swift as the offical language for Apple products, Vala for Gnome desktop.)

1.1.3 Categorising Programming Languages by Syntax



1.1.4 Categorising Programming Langauges by Methods of Execution



Assembler

In the early days of computing, Machine Instructions were used to represent the programs, however, due to the complexity of this method, first Assemblers were produced, these assemblers assembled assembly instructions, which used one-to-one correspondence between machine instructions and their mnemonics, to machine instructions to create executable files.

Examples: x86 Assembly

Compiler

Compilers create executables from high level languages by translating them to the Assembly or Machine Code instructions. Compilers lack the one-to-one correspondence between mnemonics and instructions that is found in assemblers, but state-of-the art compilers produce better code than any human will. Labor costs now outweigh the hardware costs.W

Examples: C, Fortran

Interpreter

Interpreters do not produce output programs, rather, they take the input and output together and execute them together.

Examples: Python, R, Lua

Interpretation vs Compilation

Compilers provide better speed through analysis and nontrival transofrmation, while **Interpreters provide** greater flexibility and through error messages better diagnostics.

Virtual Machines

By providing a target architecture that can be shared by multiple architectures called a virtual machine, source program is compiled to an intermediate representation and this representation is then interpretered through the virtual machine.

Examples: C#, Java

1.2 Compiler Systems

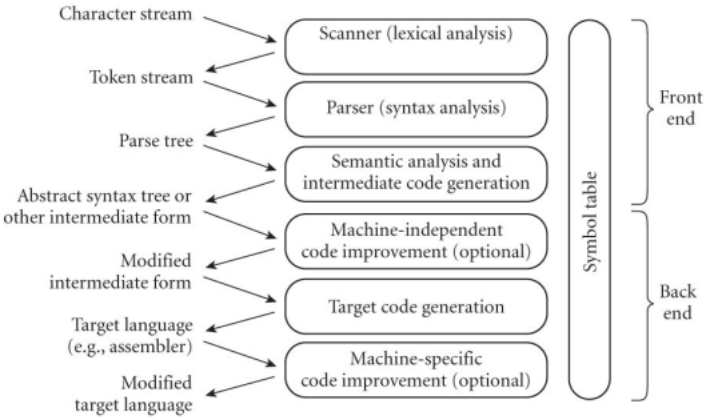


Figure 1.1: Model of a compiler

1.2.1 Auxillary Systems

Linker

Linker links external libraries to the program enabling usage of wider machine instructions, linker's linked library's instructions are added after the inital compilation of the source code to an incomplete set of machine instructions.

Preprocessor

Preprocessor modifies the existing source code prior to inital compilation.

1.2.2 Frontend

Scanning

Recognition of a regular language via a Definite Finite Automata to check if all of the program's tokens are well formed.

Parsing

Parsing is the checking if the source code conforms to the syntax defined by a context-free grammar.

Semantic Analysis

Semantic analysis is the checking of all the other rules not captured by the context-free grammar and the parse tree.

Chapter 2

Programming Language Syntax

Representation of the Context Free Grammars is the Backus-Naur Form, sometimes shortened to BNF, this representation can relaibly express any context free grammar.

2.0.1 Example

Example grammar