CPSC 323: Compilers and Languages

Introduction

Mr. Param Venkat Vivek Kesireddy
Teaching Associate
Mail: pkesireddy@fullerton.edu

Inputs from Prof Doina Bein & Prof James Choi

What we will learn

- Understanding basic concepts of languages
  - High-level programming languages, Assembly languages, Machine Languages
- Understanding compilers as the means to implement programming languages
  - Compilation vs. Interpretation
  - Phases of a compiler
  - Fundamental theories and algorithms in each phase
  - Lexical Analyzer
  - Syntactic Analyzer
  - Semantic Analyzer
  - Intermediate Code Generation
  - Simple code optimization
  - Target Code generation
  - Practice implementing some phases (tentative)

Languages

- Natural languages
  - Tools for expressing information
    - ideas, knowledge, commands, questions, ...
    - Facilitate communication between people
  - Different natural languages
    - English, Spanish, Chinese, French, German, ...

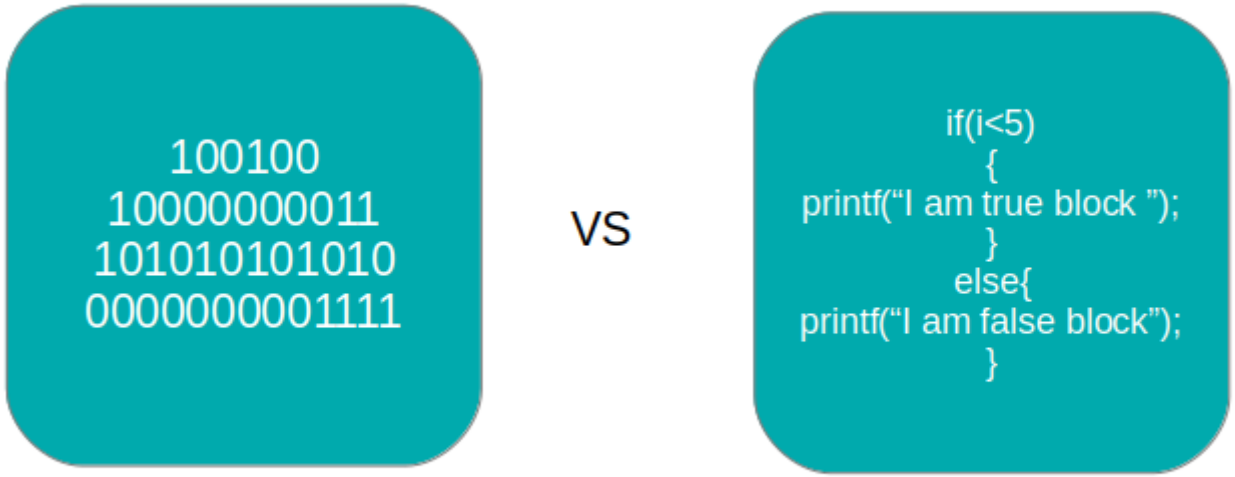
- Formal languages
  - Tools for use in specific situations, such as math or computer programming
  - They often use symbols, numbers, and characters that natural languages do not.
  - Languages with precise syntax and semantics are called formal languages. Programming languages are examples of formal languages.

- Programming languages
  - Tools for expressing data and algorithms
    - Instructing machines what to do
    - Facilitate communication between programmers and computers
  - Different levels of programming languages
    - High-level, low-level

Levels of Programming Languages

High-level (HL) language

Low-level (LL) language



High Level Languages

- High-level languages are more user-friendly and enable programmers to write code that is closer to natural language, making it easier to develop complex software applications.
- Does not require addressing hardware constraints.
- Every single program written in a high-level language must be interpreted into machine language before being executed by the computer.
- C,C++, Python and Java are popular examples of high-level languages.

Low level Languages

Assembly language (ASM)

- A type of low-level programming language that is intended to communicate directly with a computer's hardware
- A single line of assembly-language code normally corresponds to a single machine-language instruction (1:1)
- Assembly language is a low-level programming language that provides a human-readable representation of the machine code instructions executed by a computer's central processing unit (CPU).

Machine language

- the native language of the computer consisting of binary or hexadecimal instructions which a computer can respond to directly.
- This is literally the only language the computer can properly be said to "understand".
- Ex: A typical machine-language instruction in the IBM 370 family of computers looks like: 0001100000110101 or 1835 (written in hexadecimal).

Advantages and Disadvantages of HLL

Advantages

- Expressions
- Control structures/abstractions
- Data types
- Encapsulation
- Strongly-typed language: everything must be declared by the programmer before use

Disadvantages

- Execution is slow
- Occupies more memory
- Hardware control is less
- Not Time-efficient

Advantages and Disadvantages of LLL

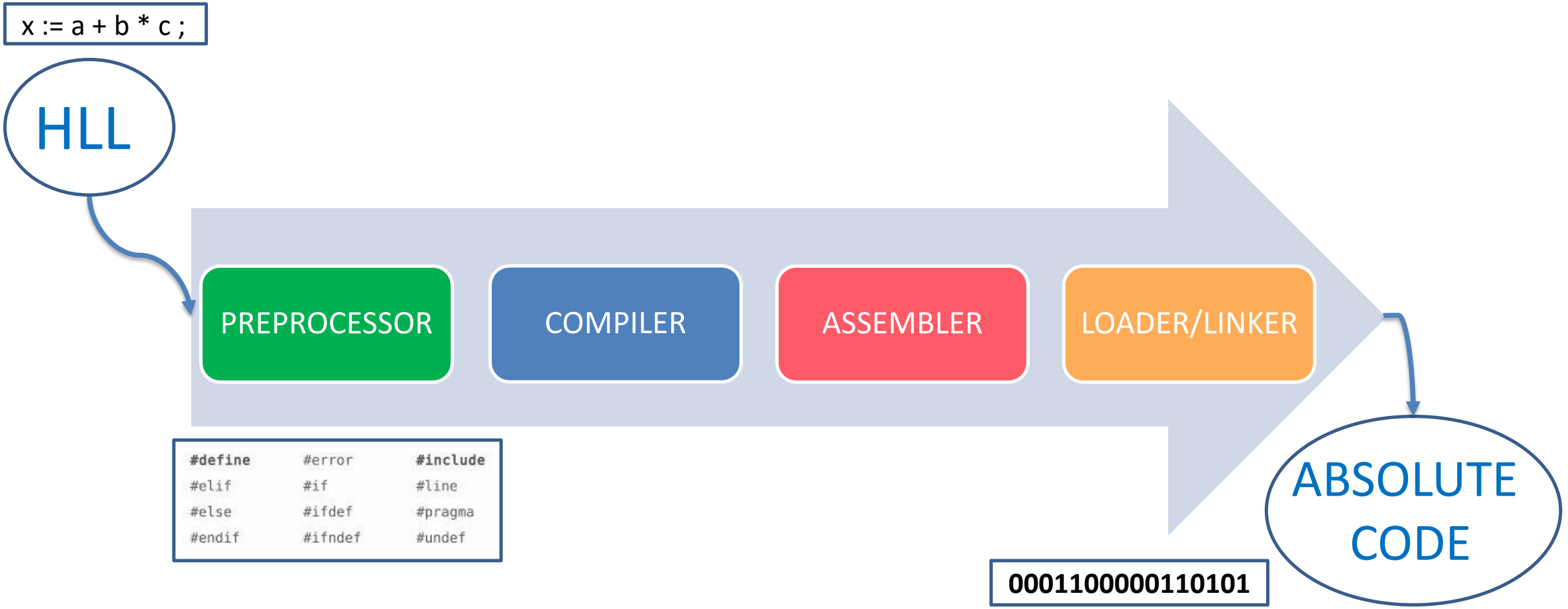
Advantages

- Fast and memory efficient
- Utilize processor and memory in better way.
- Time efficient
- Direct manipulation of computer registers and storage.
- Directly communicate with hardware devices.

Disadvantages

- Difficult to develop, debug and maintain.
- Machine dependent and are not portable.
- Error prone.
- Poor programming productivity
- Must have additional knowledge of the computer architecture of particular machine.

Levels of Programming languages



Economy of Programming Languages

Three obvious questions:

- Why are there so many programming languages?
- Why are there new programming languages?
- What is a good programming languages?

Economy of Programming Languages

Three obvious questions:

- Why are there so many programming languages?
  - Application domains have distinctive/conflicting needs.
  - It is hard to design one system for all.
    - Ex., scientific computing (good float points, good arrays, parallelism, etc.) – Fortran
    - Ex., business computing (persistence, good report facilities, data analysis, etc.) – SQL
    - Ex., system programming (control of resources, real time constraints, etc.) – C/C++

Economy of Programming Languages

Three obvious questions:

- Why are there new programming languages?
  - Old languages are not easy to change, it is much easier to design new languages for new opportunities
- What is a good programming languages?
  - There is no universally accepted metric for language design

Basic Terminology

- *Compiler*: is a software (program) that translates a program written in a source language (source code) into the code in the object language of a target machine (object code).
- *Source Language*: Programming language that the compiler accepts as an input (e.g., Pascal, C, C++, Fortran)
- *Object Language*: A particular machine (or assembly) language that is used to generate as the output of a compiler (Object Code).
- *Object file*: an external file storing object code (E.g., Myprog.obj)
- Target Machine: the computer on which the object code is to be run

Why should we study compiler?

Compilers are everywhere!

Many applications of compiler technology

- Parsers for HTML in web browser
- Interpreters for JavaScript/Flash
- Machine code generation for high-level programming languages
- Design of new computer architectures
- Hardware synthesis: VHDL to RTL translation
- Software productivity tools

History of compiler

- A-0 System
  - First implemented compiler was written by Grace Hopper (1951)
  - Here A-0 compiler converted symbolic mathematical code into machine code that could be executed by a computer.

- Fortran I (Formula Translator)
  - Was the 1st successful HL programming language
  - The first commercially available compiler (by John Backus at IBM, 1950s)
  - Huge impact on computer science

- Modern compilers preserve the outline of Fortran I

Structure of Compiler

1. Lexical Analysis
2. Syntax Analysis (Parsing)
3. Semantic Analysis
4. Intermediate Code Generation
5. Code Optimization
6. Target Code Generation