



Art  
04

# CST8152 Compilers

**Algonquin College**

Computer Engineering  
Technology

# CST8152 Compilers

**Fall, 2023**



Art  
04

Based on resources developed  
by prof. **Svillen Ranev**.

**Prof. Paulo Sousa**



**Algonquin College**

Computer Engineering  
Technology

# CST8152 Compilers

**Fall, 2023**



**Art  
04**

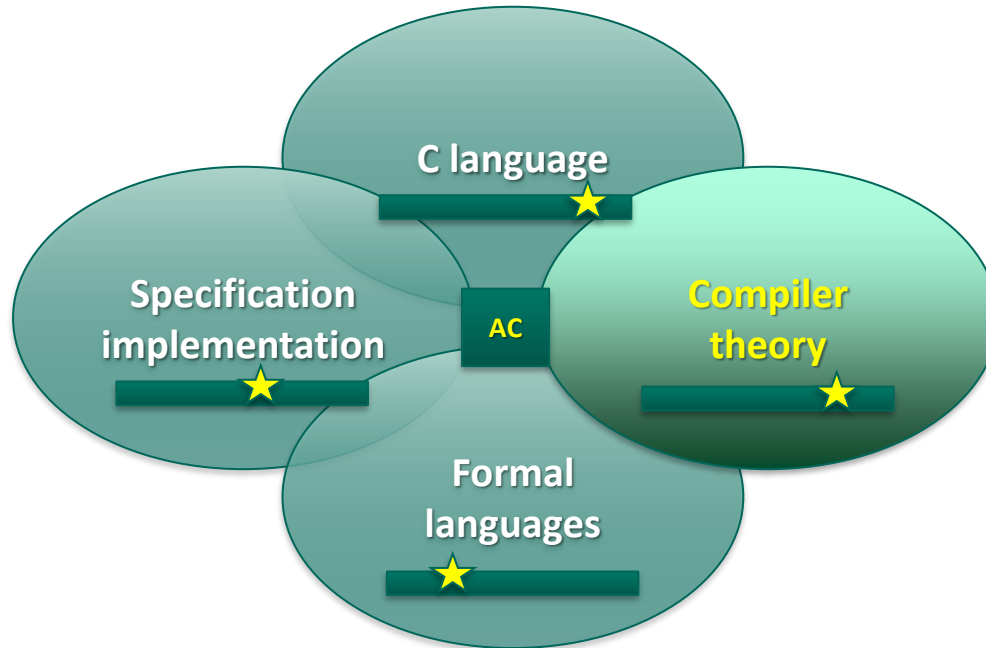
**Compiler Development**



Based on resources developed  
by prof. **Svillen Ranev**.

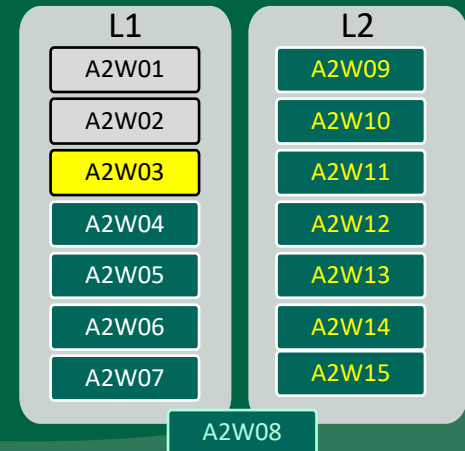
**Prof. Paulo Sousa**

# Let's start...



## Art 4: Compiler Development

- *Buffer and Front-End*
- *Bootstrapping*
- *Trends*



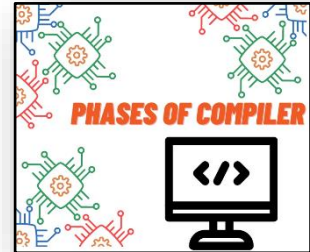


Compilers – Art. 4

# Compiler Development

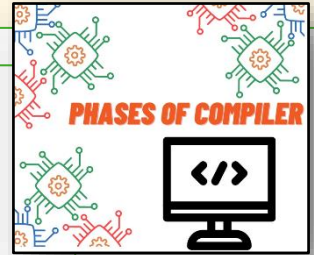
# Current Development

- **Definition:** The sequence of steps followed during the compilation process.
  - A **pass** reads the source and performs transformations, by intermediate files.
  - Some languages require at least two passes to generate code.  
So, basically, we have:
    - Single-phase
    - Multi-phase



# Single-pass compiler

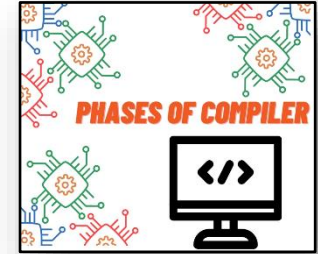
- **Single pass compiler:** The source code is scanned **only once**, done line by line. It is used to traverse the program only once. The one-pass compiler passes only once through the parts of each compilation unit. It translates each part into its final machine code.
  - When the line source is processed, it is scanned and the token is extracted.
  - Then the syntax of each line is analyzed and the tree structure is built. After the semantic part, the code is generated.
  - The same process is repeated for each line of code until the entire program is compiled.





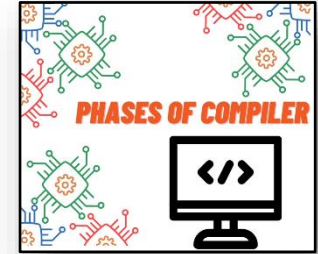
# Multi-pass compiler

- **Multi-pass compiler:** This is the **common case**, where we need to process the code several times and takes the result from one phase to another. We have independent machine code and splits the program into smaller pieces.
  - In the **first pass**, compiler can read the source program, scan it, extract the tokens and store the result in an output file.



# Multi-pass compiler

- In the **second pass**, compiler can read the output file produced by first pass, build the syntactic tree and perform the syntactical analysis.
  - The output of this phase is a file that contains the syntactical tree.
- In the **third pass**, compiler can read the output file produced by second pass and check that the tree follows the rules of language or not.
  - The output of semantic analysis phase is the annotated tree syntax.
  - This pass is going on, until the target output is produced.



# Resume

## SINGLE PASS COMPILER VERSUS MULTIPASS COMPILER

### SINGLE PASS COMPILER

A type of compiler that passes through the parts of each compilation unit only once, immediately translating each code section into its final machine code

Faster than multipass compiler

Called a narrow compiler

Has a limited scope

### MULTIPASS COMPILER

A type of compiler that processes the source code or abstract syntax tree of a program several times

Slower as each pass reads and writes an intermediate file

Called a wide compiler

Has a great scope

There is no code optimization

There is no intermediate code generation

Takes a minimum time to compile

Memory consumption is lower

Used to implement programming languages such as Pascal

There is code optimization

There is intermediate code generation

Takes some time to compile

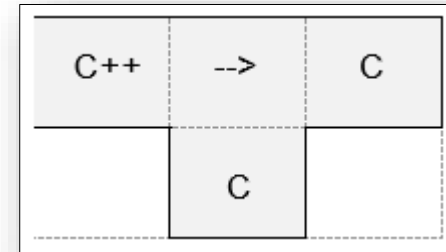
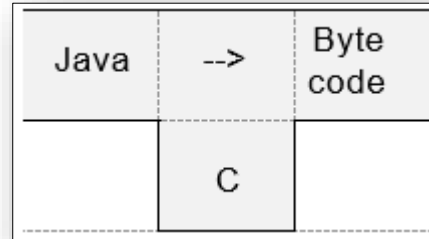
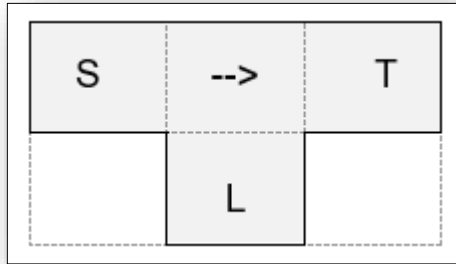
Memory consumption is higher

Used to implement programming languages such as Java

Visit [www.PEDIAA.com](http://www.PEDIAA.com)

# Current Development

- Diagrams:

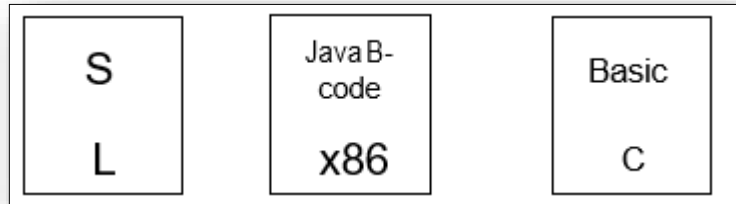


- Translators (Compilers).** T-shaped (T-shirt) tombstone represents translators (compilers).



# Current Development

- **Diagrams:**



- The head of the tombstone shows the translator's source language **S** and the target language **T**;
- The arrow indicates the direction of the translation. The base of the tombstone shows the translator's implementation language.

- **Interpreters.** An interpreter is a program that accepts a program written in some source language, and in most cases, runs it immediately without translating the entire source program into target program (machine code).

# More about front-end compiler

- See the buffer...

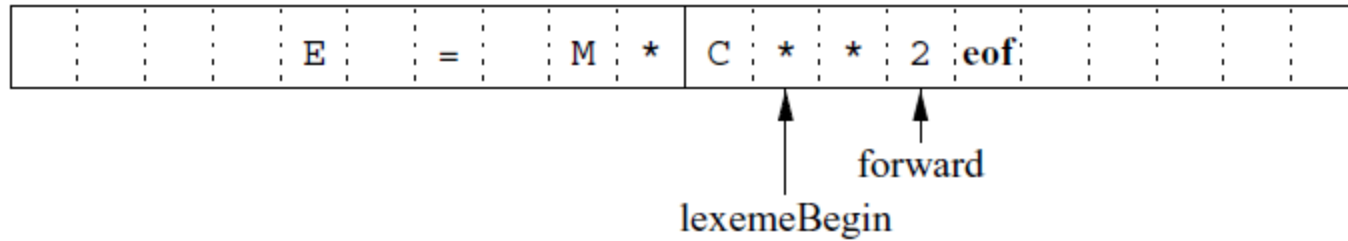


Figure 3.3: Using a pair of input buffers

# Limits

- **Check the basic architecture:**
  - For practical purposes, the buffer is in level 3 (physical memory).

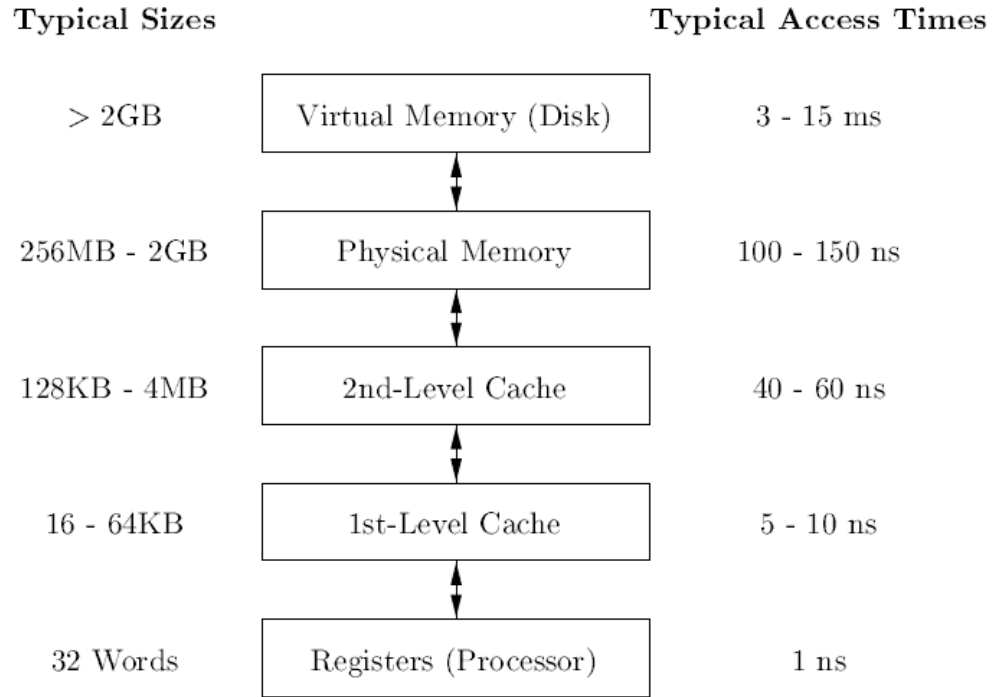


Figure 7.16: Typical Memory Hierarchy Configurations

# More about front-end compiler

- Connecting concepts...

TOKEN	INFORMAL DESCRIPTION	SAMPLE LEXEMES
<b>if</b>	characters i, f	if
<b>else</b>	characters e, l, s, e	else
<b>comparison</b>	< or > or <= or >= or == or !=	<=, !=
<b>id</b>	letter followed by letters and digits	pi, score, D2
<b>number</b>	any numeric constant	3.14159, 0, 6.02e23
<b>literal</b>	anything but ", surrounded by "'s	"core dumped"





## Compilers – Art. 4

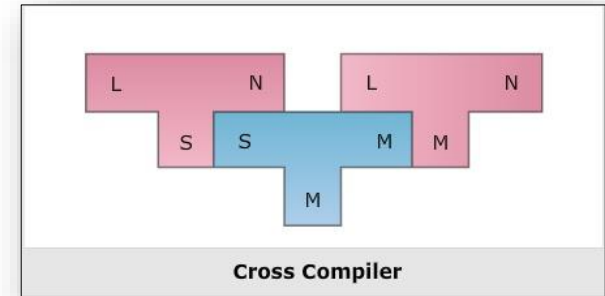
# Bootstrapping

# Bootstrapping (1)

## Bootstrapping:

When a compiler is written using the **same language** the compiler is supposed to compile from

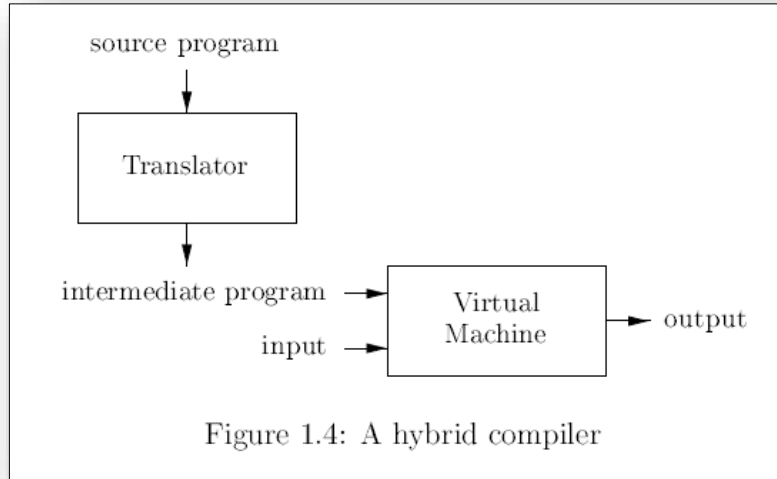
- **Half bootstrapping:** If a compiler for the language already exist but we want to write a compiler for a different machine using the same language the compiler compiles;
- **Full bootstrapping:** If the compiler is written from the scratch.



# Hybrid Compilation

Example:

- Hybrid compilation is very well used. The general scheme is shown here:



- Java Code:

```
public class HelloWorld {  
    public static void main(String[] args) {  
        System.out.println("Hello World!");  
    }  
}
```



# Intermediate Code

- Bytecode (JVM machine):

```
public class HelloWorld {  
    public HelloWorld();  
        Code:  
        0: aload_0  
        1: invokespecial #1    // Method java/lang/Object."<init>":()V  
        4: return  
    public static void main(java.lang.String[]);  
        Code:  
        0: getstatic  #2        // Field java/lang/System.out:Ljava/io/PrintStream;  
        3: ldc      #3          // String This will be printed  
        5: invokevirtual #4      // Method java/io/PrintStream.println:(Ljava/lang/String;)V  
        8: return  
}
```







## Compilers – Art. 4

# The Future...




# Thinking about the Future..

- Facebook AI Creates Its Own Language In Creepy Preview Of Our Potential Future:
  - <https://www.forbes.com/sites/tonybradley/2017/07/31/facebook-ai-creates-its-own-language-in-creepy-preview-of-our-potential-future/>
- The truth behind Facebook AI inventing a new language:
  - <https://towardsdatascience.com/the-truth-behind-facebook-ai-inventing-a-new-language-37c5d680e5a7>
- OpenAI API:
  - <https://openai.com/blog/openai-api/>



**Main idea:** Machines can create machines – remember that a compiler is a program = logical machine.

# Recent News...



## About automatic language for build compilers (GPT-3):

- GPT-3 Demo:  
<https://www.youtube.com/watch?v=8psgEDhT1MM>
- GPT-3 Paper:  
<https://arxiv.org/pdf/2005.14165.pdf>
- Kevin Lacker tests:  
<https://lacker.io/ai/2020/07/06/giving-gpt-3-a-turing-test.html>

## IBM wants to teach machines to program using 'CodeNet' dataset:

...14 million code samples for 4,000 problems...

IBM has built and released CodeNet, a dataset of 14 million code submissions for 4,000 distinct programming challenges. CodeNet is designed to help people build AI systems that can generate and analyze code. Part of why CodeNet exists is because of the impressive progress in NLP which has occurred in recent years, with architectural improvements like the Transformer and AI systems such as GPT-3 and T5 leading leading to NLP having its so-called "ImageNet moment" ([Import AI 170](#)).

**What is CodeNet?** CodeNet consists of coding problems scraped from two coding websites - AIZU and AtCoder. More than 50% of the code samples within CodeNet "are known to compile and run correctly on the prescribed test cases", IBM said. More than 50% of the submissions are in C++, followed by Python (24%), and Java (5%). CodeNet contains 55 different languages in total.

**Why this matters:** Now that computers can read and generate text, we might ask how well they can read and generate code. We know that they have some basic capabilities here, but it's likely that investment into larger datasets, such as CodeNet, could help us train far more sophisticated code processing AI systems than those we have today. In a few years, we might delegate coding tasks to AI agents, in the same way that today we're starting to delegate text creation and processing tasks.

**Read the paper:** [Project CodeNet: A Large-Scale AI for Code Dataset for Learning a Diversity of Coding Tasks](#) (IBM GitHub).

**Read more:** [Kickstarting AI for Code: Introducing IBM's Project CodeNet](#) (IBM research blog).

**Get the code here:** [Project CodeNet](#) (IBM GitHub).



## Compilers – Art. 4

# Concluding



# Review

- *Tools / strategy for creating compilers...*

## Some Questions

1. *Why to use diagrams to Compilers representation?*
2. *What is the importance of bootstrapping?*
3. *Can you create a compiler for interpret bytecodes?  
How?*
4. *What are the pros and cons of using AI?*



Source:  
[https://static.wixstatic.com/media/7594af\\_51a81a8ccc5f418281f52c8bdd2dd618~mv2.jpg](https://static.wixstatic.com/media/7594af_51a81a8ccc5f418281f52c8bdd2dd618~mv2.jpg)



# Open questions...

- Any doubts / questions?
- How we are until now?





## Compilers – Art. 4

**Thank you for your  
attention!**