Art
02

CST8152 Compilers

Algonquin College

Computer Engineering Technology

CST8152 Compilers

Fall, 2023

Art 02

Prof. Paulo Sousa

Algonquin College

Computer Engineering Technology

CST8152 Compilers

Fall, 2023

Art 02

Inside Compilers

Prof. Paulo Sousa

CST8152 — Compilers

Art 2: Inside Compilers

- *Compilers Context*

L1	L2
A2W01	A2W09
A2W02	A2W10
A2W03	A2W11
A2W04	A2W12
A2W05	A2W13
A2W06	A2W14
A2W07	A2W15

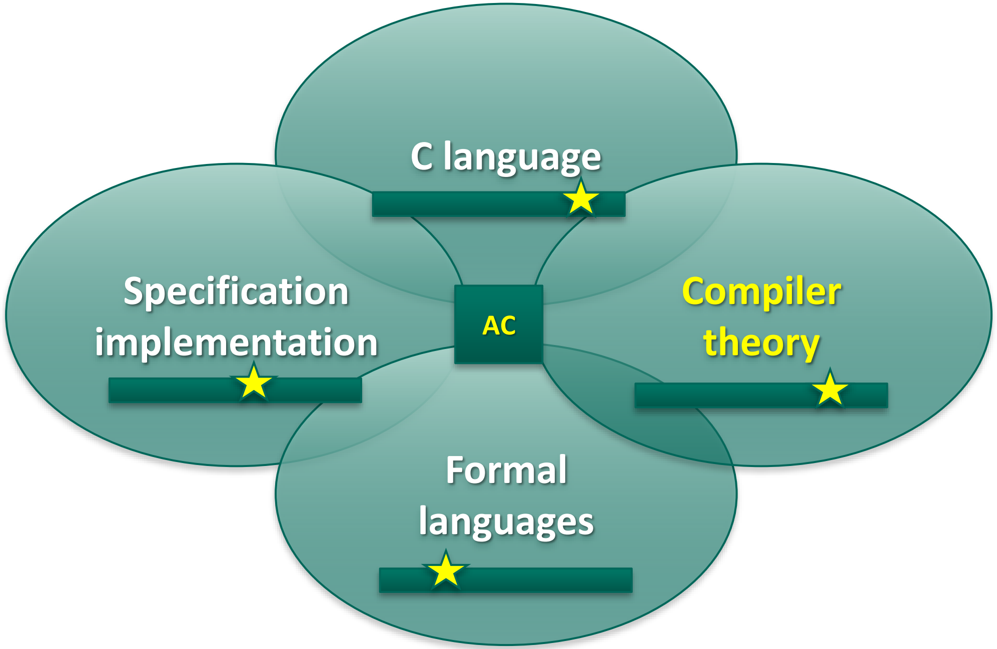
A2W08

ALGONQUIN COLLEGE

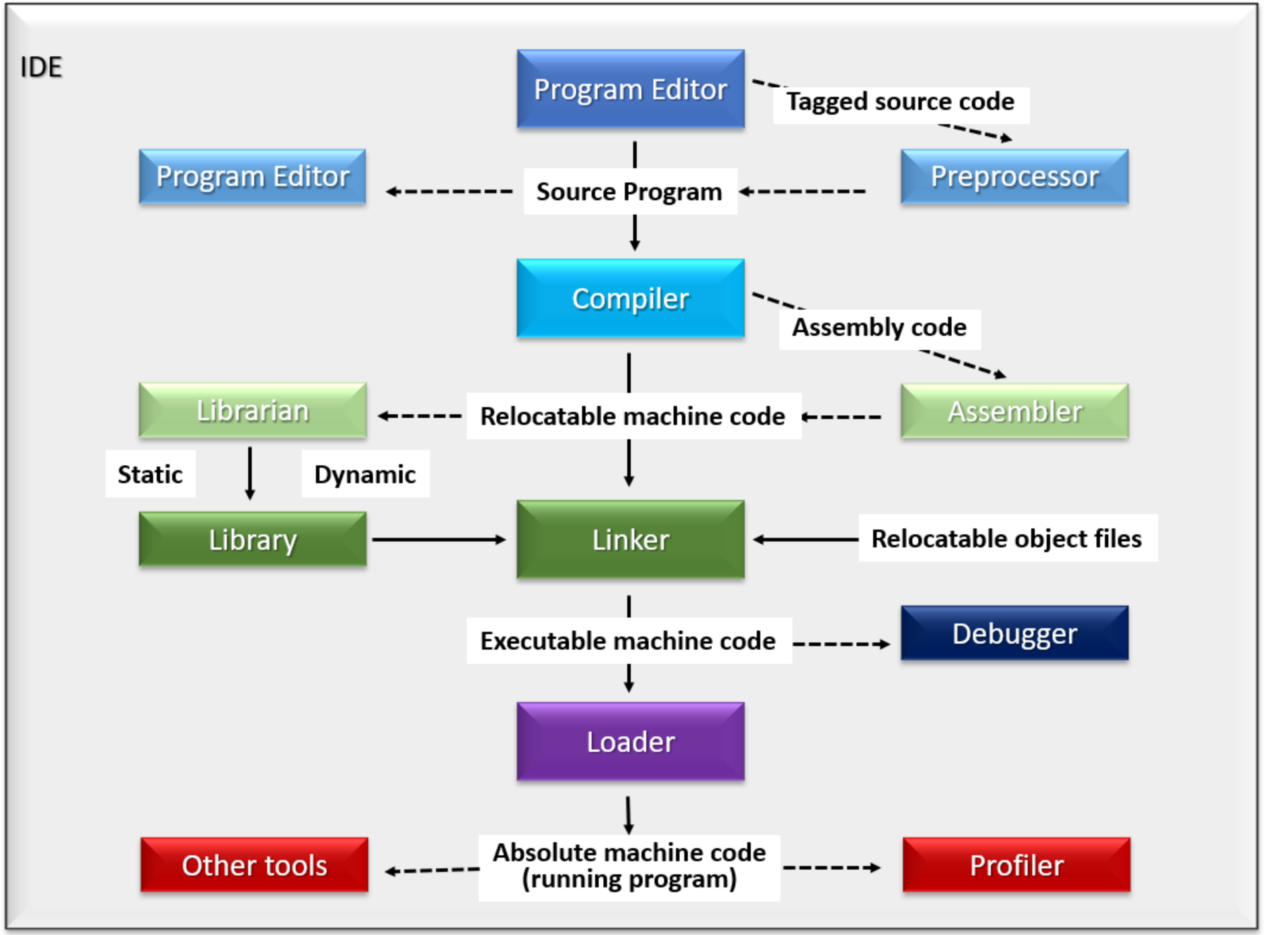
Compilers – Art 2

Context of a Compiler

Lets start…


C language
Specification implementation
AC
Compiler theory
Formal languages

2.1. General Diagram



Source: Algonquin College (Svillen Ranev)

2.1. Definitions

- **Assembler**: Assemblers are simple compilers which translate assembly language into machine code.
- **Compiler**: Translate the text of the program in another language - usually assembler or some form of machine code.
- **Debugger**: Allows the user to trace the execution of a program statement by statement and inspect the content of different parts of the program memory.

Source: https://towardsdatascience.com/top-10-in-demand-programming-languages-to-learn-in-2020

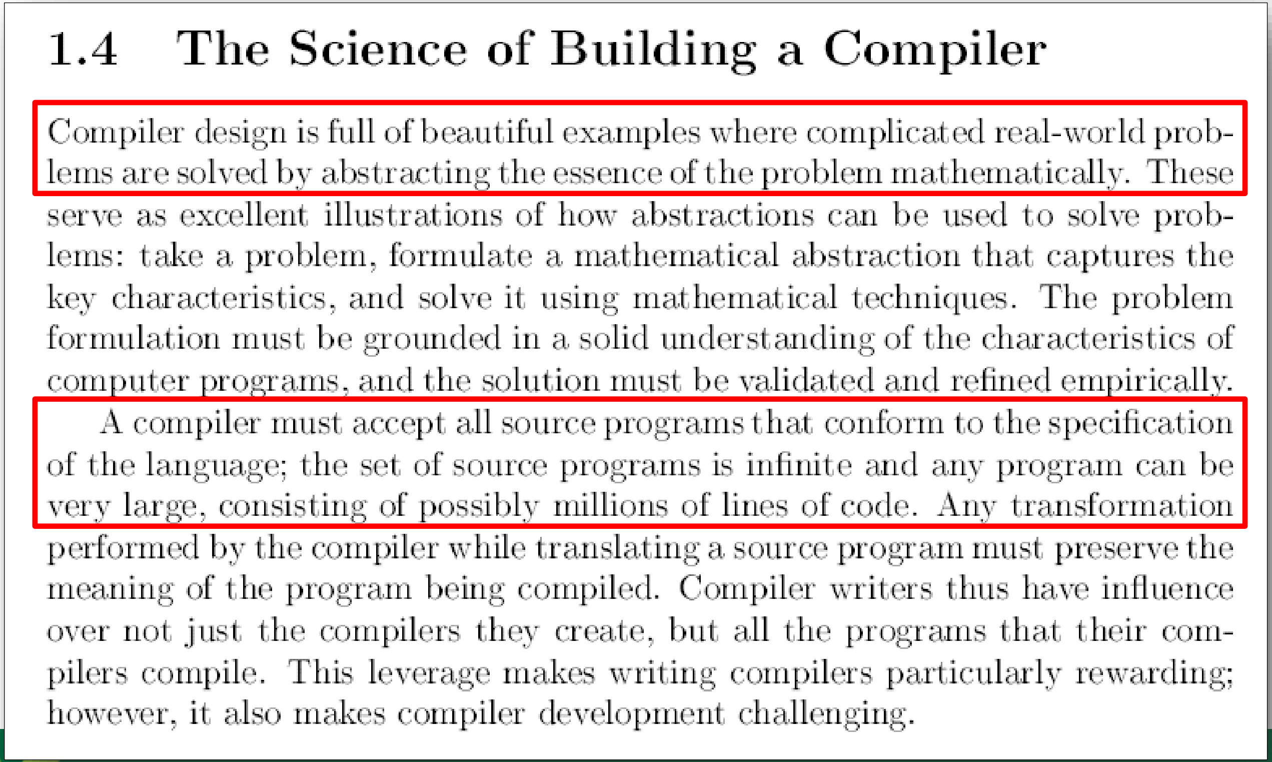
2.1. Definitions

- **Librarian**: Allows creating and maintaining libraries of pre-compiled component which can be used later without the need to be compiled again.
- **Linker**: Combines (links) all necessary components of a program into some executable form. Not all programming languages require linkers.
- **Loader**: Loads an executable program and passes the control to the program.

2.2. Other Tools

- Automatic or Unit testers (JUnit),
- Code Inspectors and Analyzers,
- Error loggers,
- Make and build scripting tools (Ant),
- Profilers,
- Project Managers (Maven)
- Refactoring tools,
- Run-time Inspectors,
- Style Formatters,
- Task Managers (Mylyn),
- Version Control and, Collaboration (Git).

2.3. General View



1.4 The Science of Building a Compiler

Compiler design is full of beautiful examples where complicated real-world problems are solved by abstracting the essence of the problem mathematically. These serve as excellent illustrations of how abstractions can be used to solve problems: take a problem, formulate a mathematical abstraction that captures the key characteristics, and solve it using mathematical techniques. The problem formulation must be grounded in a solid understanding of the characteristics of computer programs, and the solution must be validated and refined empirically.

A compiler must accept all source programs that conform to the specification of the language; the set of source programs is infinite and any program can be very large, consisting of possibly millions of lines of code. Any transformation performed by the compiler while translating a source program must preserve the meaning of the program being compiled. Compiler writers thus have influence over not just the compilers they create, but all the programs that their compilers compile. This leverage makes writing compilers particularly rewarding; however, it also makes compiler development challenging.

2.3. Example Hybrid Compiler

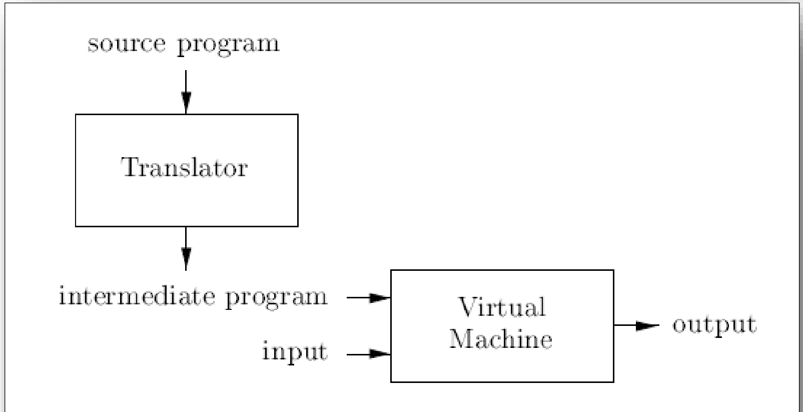


Figure 1.4: A hybrid compiler

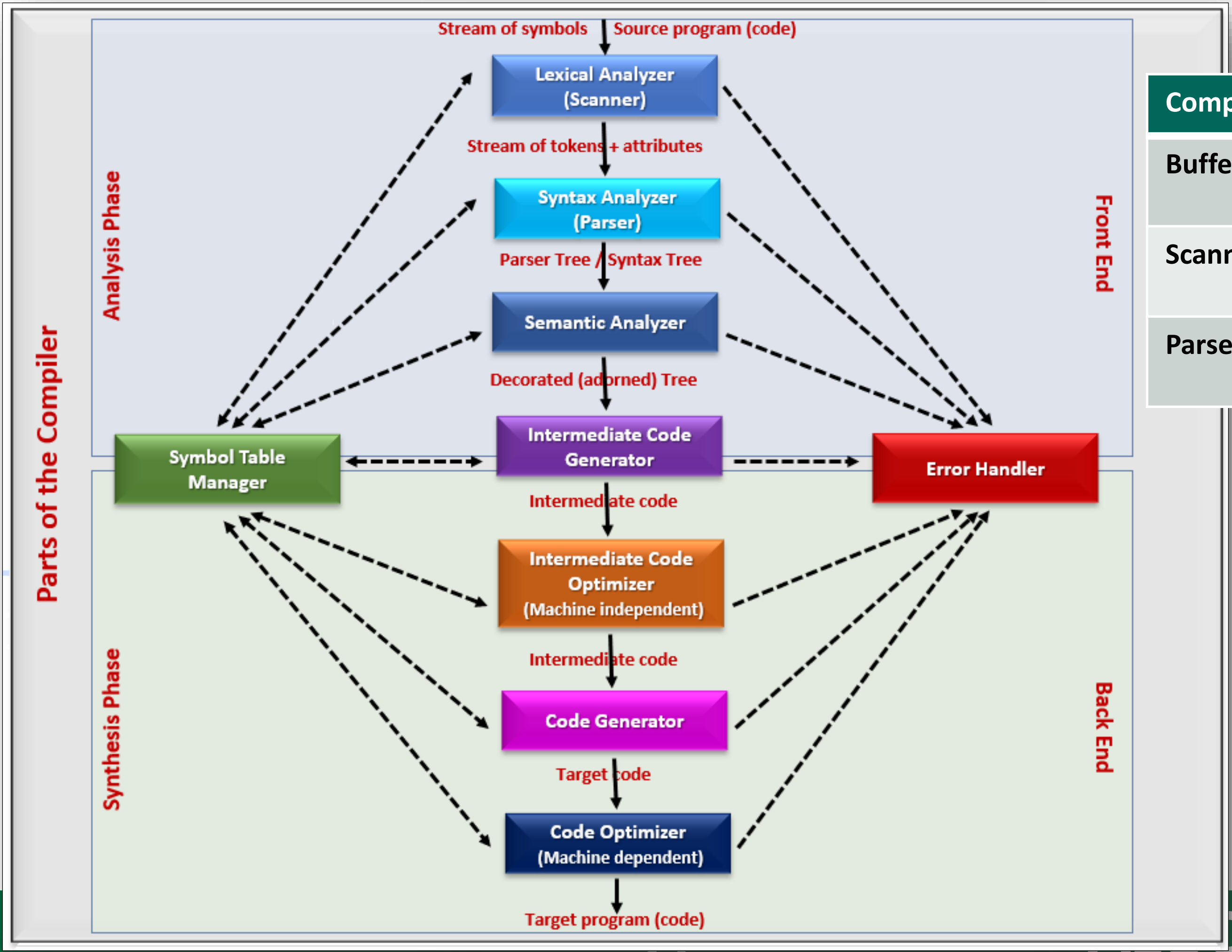
```
public class HelloWorld {
    public static void main(String[] args) {
        System.out.println("This will be printed");
    }
}
```

```
> javap HelloWorld.class
Compiled from "HelloWorld.java"
public class HelloWorld {
  public HelloWorld();
  public static void main(java.lang.String[]);
}
```

2.3. Example Hybrid Compiler

```
> javap -c HelloWorld.class
Compiled from "HelloWorld.java"
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
}
```



Component	Units
Buffer	Symbols (char)
Scanner	Tokens (lexeme)
Parser	Structures (BNF)

2.3. General View

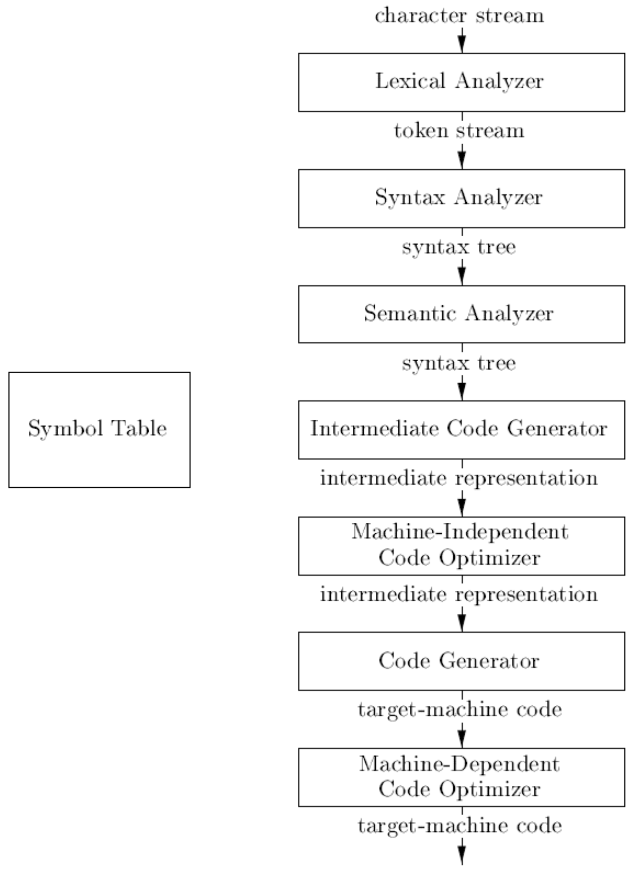
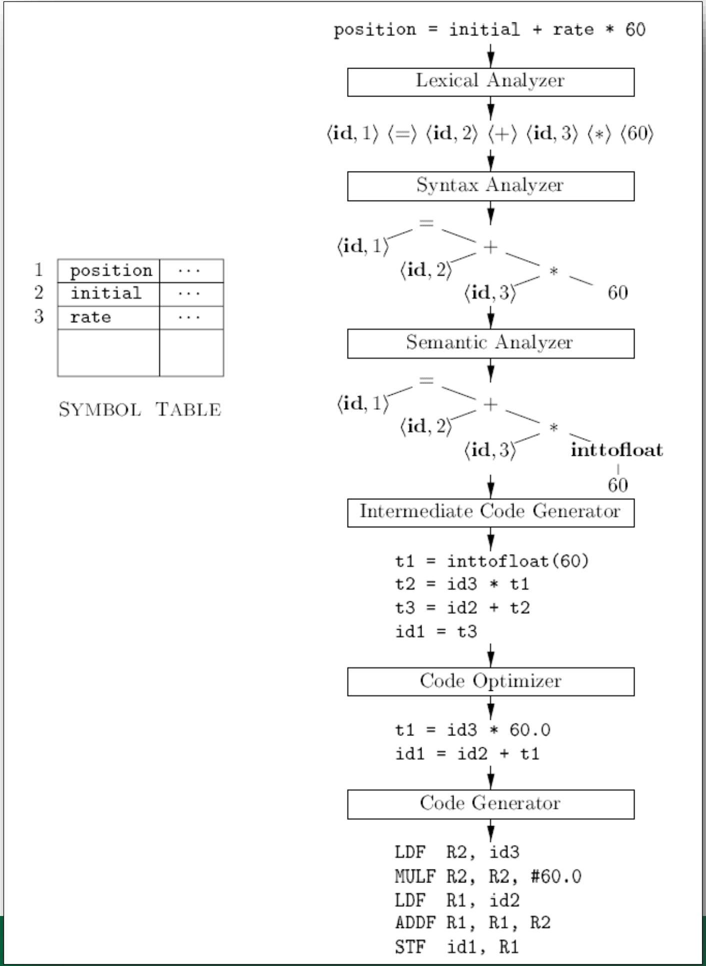


Figure 1.6: Phases of a compiler

2.3. General View



1	position	···
2	initial	···
3	rate	···

SYMBOL TABLE

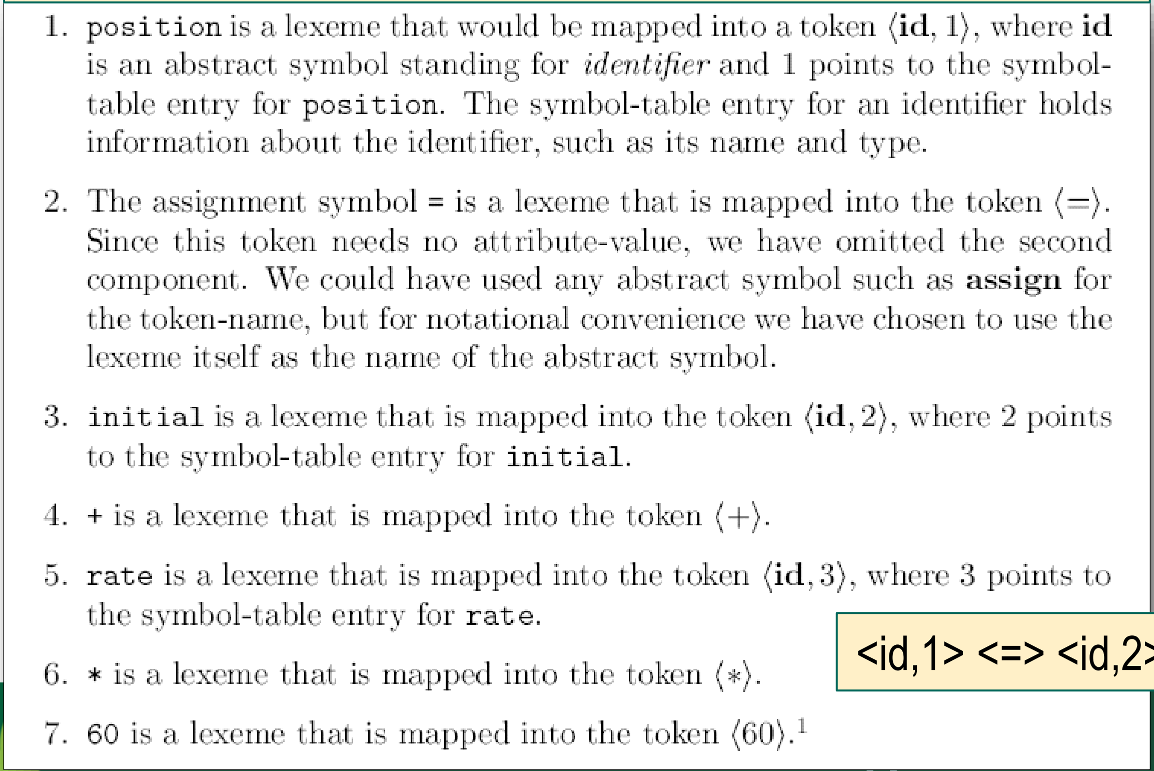
```
t1 = inttofloat(60)
t2 = id3 * t1
t3 = id2 + t2
id1 = t3
```

```
t1 = id3 * 60.0
id1 = id2 + t1
```

```
LDF  R2, id3
MULF R2, R2, #60.0
LDF  R1, id2
ADDF R1, R1, R2
STF  id1, R1
```

2.3. General View

position = initial + rate * 60



1. `position` is a lexeme that would be mapped into a token $\langle \textbf{id}, 1 \rangle$, where **id** is an abstract symbol standing for *identifier* and 1 points to the symbol-table entry for `position`. The symbol-table entry for an identifier holds information about the identifier, such as its name and type.
2. The assignment symbol `=` is a lexeme that is mapped into the token $\langle = \rangle$. Since this token needs no attribute-value, we have omitted the second component. We could have used any abstract symbol such as **assign** for the token-name, but for notational convenience we have chosen to use the lexeme itself as the name of the abstract symbol.
3. `initial` is a lexeme that is mapped into the token $\langle \textbf{id}, 2 \rangle$, where 2 points to the symbol-table entry for `initial`.
4. `+` is a lexeme that is mapped into the token $\langle + \rangle$.
5. `rate` is a lexeme that is mapped into the token $\langle \textbf{id}, 3 \rangle$, where 3 points to the symbol-table entry for `rate`.
6. `*` is a lexeme that is mapped into the token $\langle * \rangle$.
7. `60` is a lexeme that is mapped into the token $\langle 60 \rangle$.[1]

<id,1> <=> <id,2> <+> <id,3> <*> <60>

CST8152 — Compilers

Compilers – Art 2

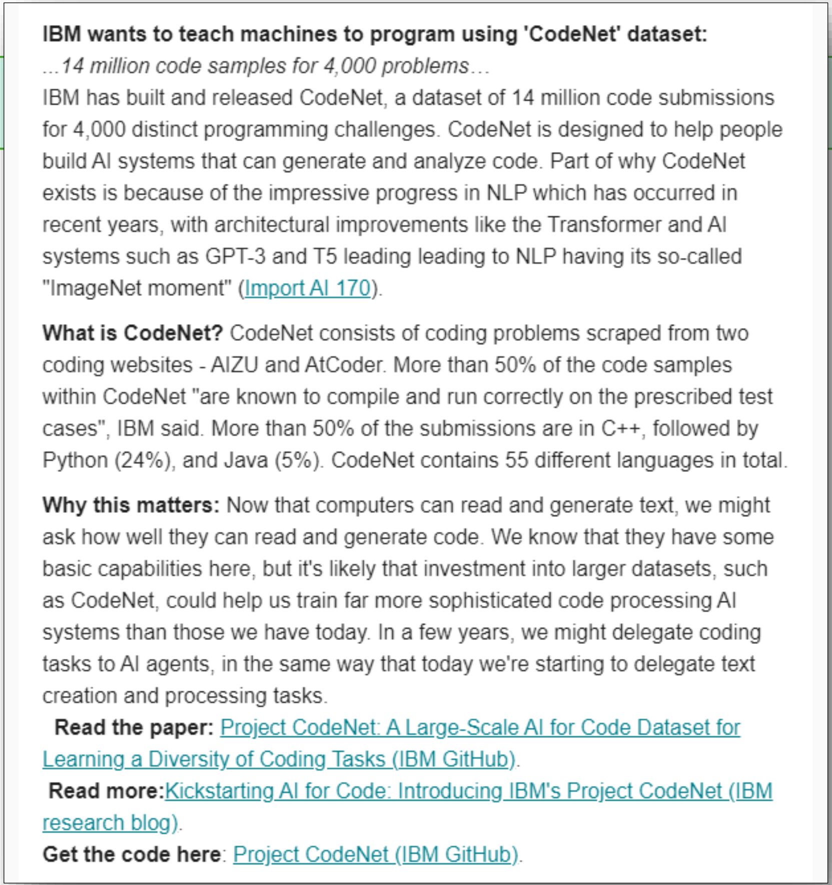
The Future...

ALGONQUIN COLLEGE

Recent News (17/05/2021)

- **About AI and Code Generation**

https://research.ibm.com/blog/codenet-ai-for-code


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).

Recent News (11/07/2023)

- **Top Artificial Intelligence (AI) Tools That Can Generate Code To Help Programmers**

https://www.marktechpost.com/2023/07/11/top-artificial-intelligence-ai-tools-that-can-generate-code-to-help-programmers/

- OpenAI Codex
  - https://openai.com/blog/openai-codex
- Tabnine
  - https://www.tabnine.com/
- CodeT5
  - https://blog.salesforceairesearch.com/codet5/
- GitHub Copilot
  - https://github.com/features/copilot/
- DeepCode
  - https://snyk.io/platform/deepcode-ai/
- TabNine
  - https://www.tabnine.com/

Compilers – Art 2

Concluding

Review

- *Define the elements of the context of the diagram.*

Some Questions

1. *Describe the compilation process using the previous concepts.*
2. *Give some errors that can happen in this process.*
3. *Propose a strategy to identify and fix (when possible) these errors.*

Source:
https://static.wixstatic.com/media/7594af_51a81a8ccc5f418281f52c8bdd2dd618~mv2.jpg

Open questions…

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

Image URL: https://cdn-az.allevents.in/events3/banners/26b150363d5757da8578fa6a1481585a368f12d4247adfe95837e6ea6c5ab2af-rimg-w1200-h549-gmir.jpg?v=1569691155

CST8152 —— Compilers

Compilers – Art 2

Thank you for your attention!

Contact: sousap@algonquincollege.com

ALGONQUIN COLLEGE