50.051 Programming Language Concepts

W13-S3 End

Matthieu De Mari

What we have seen this term

About the C/C++ language

In C/C++

A good introduction to the C/C++ programming language

- Ability to program in C and C++
- Understand how memory is used to store data and instructions, when dynamic typing is not a feature (data types declarations, arrays, pointers, etc.)
- Understand the imperative programming paradigm
- A cleaner object-oriented framework in the case of C++ (because the Python framework is kind-of-bad, to be honest).
- (The mother of all programming languages?)

Perfect your learning of C/C++

Important: 6 weeks of C/C++ is not enough to master the language!

- Keep on practicing!
- Try coming up with a portfolio of projects in C/C++?
- Some ideas could include translating your previous Python projects in C, or additional ideas.
- Keep it fun for you!

Some general ideas: https://hackr.io/blog/cpp-projects

Learn C#

In the same family of C/C++, it might be worth looking at C# as well

- C# is a high-level, object-oriented programming language that was developed by Microsoft as part of the .NET platform.
- C# is designed to be simple, modern, and easy to learn.
- It offers features such as garbage collection, type safety, and simplified memory management, that C/C++ does not have.

Good online course, here: https://learn.microsoft.com/en-us/shows/csharp-fundamentals-for-absolute-beginners/

Keep an eye on Google Carbon?

Carbon, or Carbon-Lang, is an experimental, general-purpose programming language. The project is open-source and was started by Google, following in the footsteps of previous Google-made programming languages (Go and Dart). Google engineer Chandler Carruth first introduced Carbon at the CppNorth conference in Toronto in July 2022. He stated that Carbon was created to be a C++ successor.[1][2][3] The language is expected to have a 1.0 release in 2024 or 2025.[4]

The language intends to fix several perceived shortcomings of C++[5] but otherwise provides a similar feature set. The main goals of the language are readability and "bi-directional interoperability", as opposed to using a new language like Rust (which, while being influenced by C++, is not two-way compatible with C++ programs). Changes to the language will be decided by the Carbon leads.[6][7][8][9]

From Wikipedia: https://en.wikipedia.org/wiki/Carbon_(programming_language)
Some courses start to emerge: https://betterprogramming.pub/carbon-programming-language-tutorial-6d67b4cc16ae

What we have seen this term

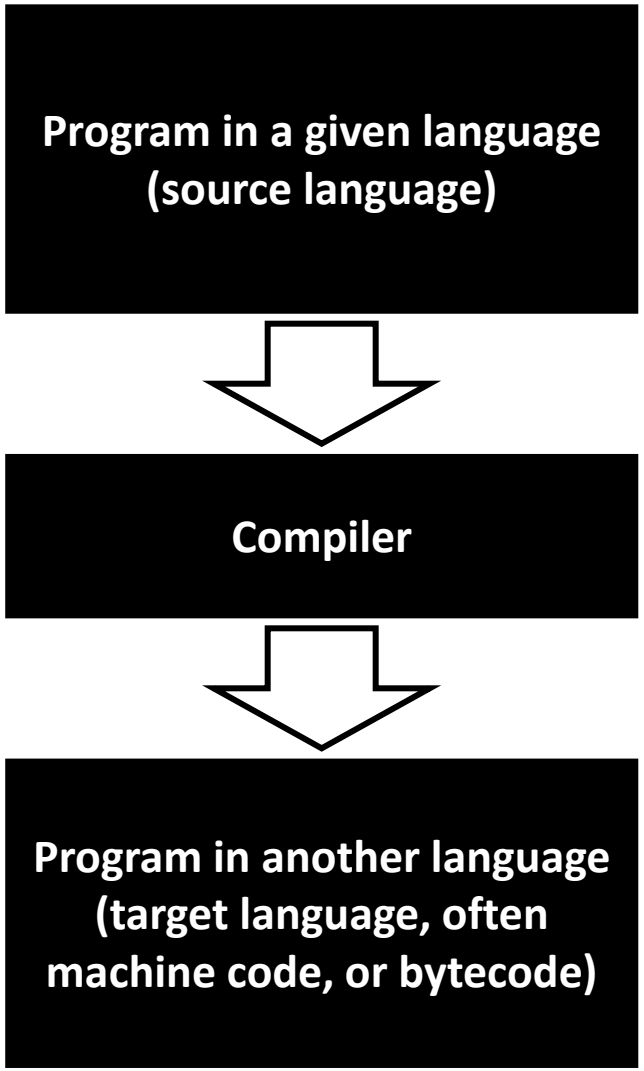
About compilers

Compiler: a definition

Definition (Compilers):
Compilers are **computer programs** whose purpose is to

- **Translate** a **program** written in **one language** (called **source language**),
- Into a **program** written in **another language** (called **target language**).

Target language is often machine code or bytecode.

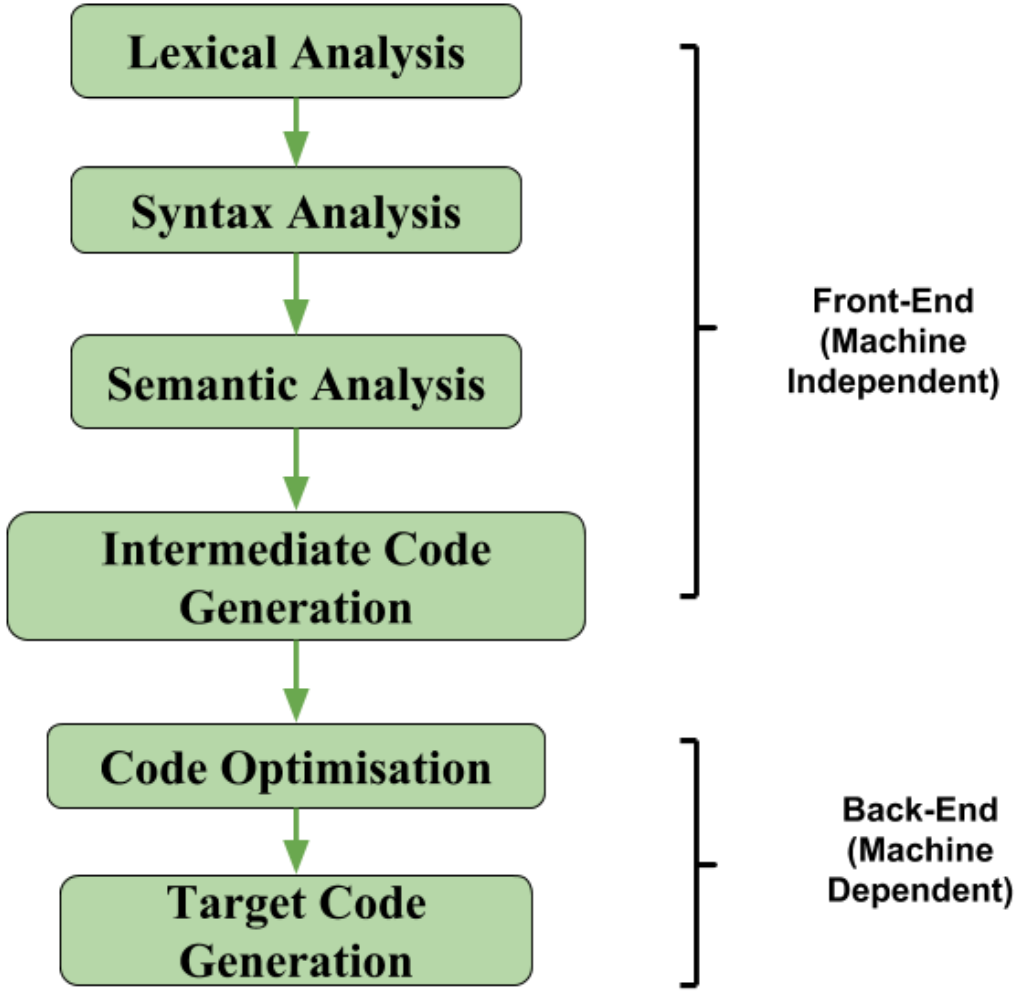


The architecture of a compiler

Definition (the three parts of a typical compiler architecture):

A typical compiler architecture consists of three main components: the **front-end**, **middle-end**, and **back-end**.

- **Front-end:** checks source code is legal (90% of the compiler job!).
- **Middle-end (optional?):** IR code generation and optimization.
- **Back-end:** Final translation and handover to CPU!



How to continue your learning

Because you should!

Continue your learning

W8S3 – W9S1: FSMs and their implementation

- Continue with more advanced types of FSMs (non-deterministic, push automatons, etc.).
- Learn how to convert any NDFSM into its equivalent FSM (tedious, but possible all the time).

Good online courses:

- MIT Theory of Computation
- Stanford Automata Theory

Continue your learning

W9S2: RegEx and their implementation

- Not much else in terms of implementing FSMs and RegEx.
- Maybe try making your own RegEx engine that takes any RegEx expression (basic operations only) and generates the correct FSM.
- Then use said FSM to check validity of a given string.
- Learn to recognize the RegEx behind any given FSM.

Good online courses:

- Few good courses will teach you about implementation in C…
- Maybe this one?

Continue your learning

W9S3-W10S1: Tokenization

- Not much else on this topic.
- Possible improvement #1: We generate/compile the RegEx FSM every time we call said RegEx function.
  Would be better to only generate it once?
- How to handle the C pre-processing instructions before tokenization?

Continue your learning

W10S2: CFGs

- How to implement a CFG?
- For a given set of production rules to use, write an algorithm that would build the parse tree matching a given derivation.

Good online courses:

- Very few, most of the time, better to check Github for implementations.
- Maybe this? (not C)

Continue your learning

W10S2: SDTs

- How to implement an SDT, and following the idea of the parse tree generating algorithm, how would you build an abstract syntax tree?

Good online courses:

- As with CFGs, very few courses online, most of the time, better to check Github for implementations.

Continue your learning

W11: Parsing

- Try implementing both the DFS and LFS algorithm in top-down parsing.
- Try implementing LR(0) and LR(1) algorithms as shown in HW2 and in class!
- Learn about more classes of LR parsers like SLR and LALR.

Good course:

- SUTD Term 7 course? **50.054 Compiler Design and Program Analysis, by Prof. Kenny Lu (in the case of functional programming).**

Continue your learning

W12S1-2: Semantics

- Try implementing a symbol table, as a simple stack, or better, a spaghetti stack! Later on, implement your own scope checking for the semantics analysis.
- Try implementing your own type checking rules and operation tables. Later on, implement your own type checking for the semantics analysis.

Good online courses:

- As with CFGs/SDTs, very few courses online, most of the time, better to check Github for implementations.

Continue your learning

W12S3: TAC code generation

- Write your own CFG and SDT for the TAC language.
- Write your own translator function that reads an AST and produces the TAC accordingly.
- Work out more cgen() functions for while loops, for loops, break, functions calls, class definitions and methods, etc.

Good online courses:

- As with CFGs/SDTs, very few courses online, most of the time, better to check Github for implementations.

Continue your learning

W13S1: TAC code optimization

- Try implementing the Copy Propagation optimization procedure.
- Try implementing a liveness analysis (challenging!).
- Try implementing a Dead Code Elimination optimization procedure.
- Chain them, rinse and repeat local optimization function for all basic blocks in your code!
- Try implementing more local optimization techniques (arithmetic simplifications, short-circuit evaluation, constant folding, etc.)
- Eventually, try writing a control-flow graph representation for your TAC code and study/implement more advanced (global) optimization procedures.

Continue your learning

W13S2: Backend

- Try writing a simple translator that reads TAC code (either as a string of text of a control-flow graph), and produces assembly code (following the Beta CPU instruction set from 50.002).
- Try implementing a few register allocation algorithm (naïve, linear scan, etc.)
- Try your hands on the graph coloring problem and the Chaitin algorithm implementation!
- More backend stuff (runtime, garbage collection, code optimization and clock cycles optimization on your CPU by performing instruction ordering/section, parallelism, etc.)

And interpreters in all of this?

Definition (Interpreters):

Many of you have probably heard about the **compilers vs. interpreters paradigm**, but what is it about?

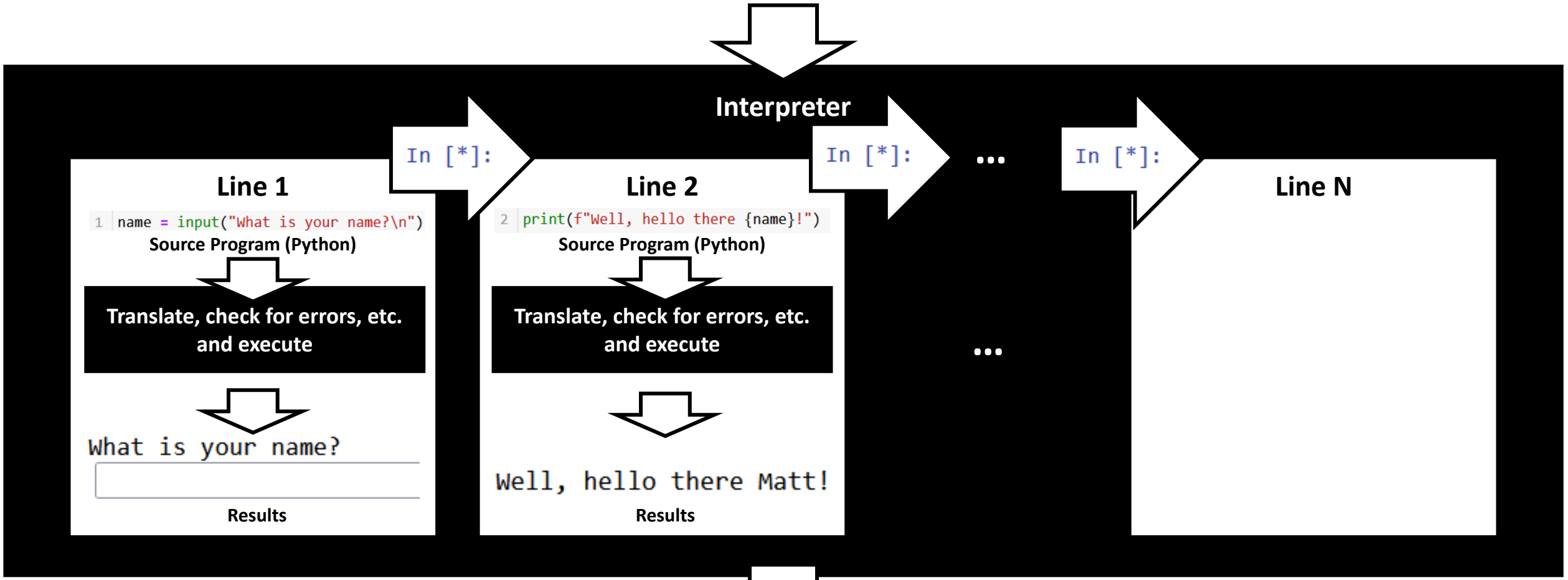
Interpreters have the **same objective as compilers**, i.e. **translate a source program into target code which can be executed by the CPU**.

What changes is the translation and execution procedure:

- **Compilers** will translate the source code into target code **in its entirety first**, and THEN will execute the target code.
- **Interpreters**, on the other hand, will translate each line of the source code **one line at a time**, and execute each one of them in succession.

```
1 name = input("What is your name?\n")
2 print(f"Well, hello there {name}!")
```

Source Program (Python)




```
What is your name?
Matt
Well, hello there Matt!
```

Results

Reference courses, in short

- If interested to learn more about parsers, the reference course is the Compiler course from Stanford

https://web.stanford.edu/class/cs143/

- Available for free online and comes with video recordings

https://www.edx.org/course/compilers

- SUTD Term 7 course? **50.054 Compiler Design and Program Analysis, by Prof. Kenny Lu (in the case of functional programming).**

(Not yet added to ISTD course catalog as of 20/04/2023.)

(Tentative syllabus to be shown in class!)

Your feedback matters!

It is our first time trying this course, and as instructors, we learned a lot by trying it, but also realized a few things did not work out…

- Typically, 6 weeks is too short to teach all the concepts of compilers correctly (maybe we should shorten the C/C++ part to make room for compilers?).
- A **lab session** or two to have you **implement some parsing/semantics checks, in a guided manner**, would have been nice.
- Will improve for next run.

Let us know if you have more feedback for us!

The End

Also, remember: if the code is legal, but does not produce the behavior you expect, then it is not the compiler's fault…

The problem is…

The End

Also, remember: if the code is legal, but does not produce the behavior you expect, then it is not the compiler's fault…

The problem, in that case, is… **You!**

It is not the compiler's job to figure out the logic you want your code to have!

And debug it for you if it fails!

(How would the compiler do it anyway?)