Language Design & Implementation

Introduction

Vassilis Markos,
Mediterranean College, Spring 2024 - 2025

Week 01

About

What Is This Module About

- Low–level Programming.
- We will explore fundamental concepts of language design, such as parsing, abstract syntax trees, grammars, deterministic automata and much more.
- We will also develop at least one programming language that will be capable of:
    - Computing simple math expressions.
    - Assigning variables.
    - Having a type system.
    - Flow control.
    - (Maybe) Executing loops.
    - (Maybe) Utilising functions.

Why Is This Useful?

- Designing a language is fun!
- Aren’t there enough mainstream programming languages?
  - Yes, there are, but there are also those thousands “little” languages which are being used everyday under the hood.
  - Make, Emacs Lisp, Jinja, lex, yacc, bison, XML, CSS, …
  - Also, it is not uncommon to have to build a parser on your own for some tasks, e.g., parsing a certain file–type that noone has already built.
- Designing a language is a great way to test your algorithmic thinking:
  - Which data structures really fit your needs?
  - How to structure such an enormous project?
  - Which approach would be better?

Why Is This Useful?

- To challenge yourselves:
  - Compared to most other projects you have encountered so far, this is the most wholesome one.
  - You have to build everything from scratch (even if you use Python as your development language).
  - This means that you will have to get to know your implementation language really well, which also provides you with a good chance to take a deep dive into a language you like.
  - ...or, dislike (it is never really late to learn lisp or Haskell).

Module Assessment

You will be assessed on the basis of two courseworks:

- **Coursework 1 (40%):** A ~1500 literature review (+- 10% without any penalty for under / over) about a single area of cutting-edge developments in language design and implementation, or interpretation and compilation.
- **Coursework 2 (60%):** Implement a custom programming language from scratch, using an existing language of your choice.

Formative feedback will be received for both courseworks provided you ask for it! ;)

Coursework 1: Brainstorming

Some ideas for topics to investigate:

- Static / dynamic code analysis
- Parallelism and concurrency
- Code optimisation
- Garbage collection and language security
- Interaction of AI and programmers / languages
- LLMs and compilation / programming languages

Coursework 1: Rubric

You will be assessed on the following axes:

- **Introduction (10%):** You should clearly provide the topic, scope and motivation for the review.
- **Organisation and Structure (10%):** Well-organised content, with a clear logical flow, proper utilisation of headings / subheadings, tables, figures, code examples etc.
- **Coverage and Synthesis (30%):** Comprehensive coverage of relevant works.
- **Critical Evaluation (20%):** Synthetic review of the presented works.
- **Clarity (10%):** Well–structured sentences.
- **Conclusion (10%):** Well–summarised results and findings.
- **Referencing (10%):** Properly structured bibliography.

Coursework 1: Logistics

- **Word limit:** 1500 words, not strict, but indicative.
- **Deadline:** TBA, but expect it around Week 7–8 of this semester, i.e., 21–28 March.
- **Submission:** Blackboard
- **Plagiarism:** Checked by Turnitin
- **Submission format:** PDF / DOCX
- Do not use your name as identification.
- Use your **student ID** to name your submission file (optionally, the course code).
- **Draft** submission and guaranteed review / feedback: until one week (168 hours) prior to the deadline.

Coursework 2: Brainstorming

- What will your language look like?
    - Procedural, like C?
    - Object oriented, like C++, Java?
    - Declarative, like SQL?
    - Functional, like Haskell, Lisp?
- What **data structures** will you need the most by your implementation language (i.e., the language you will build your interpreter / compiler upon)?
- Will you need **discriminated values** (e.g., Java's enums)?
- What about **polymorphism**?
- How does your implementation language **handle memory**?

Coursework 2: Rubric

- **Stage 1 (0-20%):** Basic Arithmetic Operations
- **Stage 2 (20-40%):** Boolean Logic
- **Stage 3 (40-50%):** Text Values
- **Stage 4 (50-60%):** Global Data
- **Stage 5 (60-80%):** Flow Control
- **Stage 6 (80-100%):**
  - List–like data structure (10%)
  - Dictionary data structure (10%)
  - Functions (10%)
  - Local variables (15%)
  - Anything else (graded based on difficulty)
- Total grade == max(sum of stage grades, 100).

Coursework 2: Logistics

- Start working early, as this is a really **tough project!**
- You should submit a **single ZIP** file containing:
  - **BUILD.txt**, which explains how to build your project from source + any dependencies.
  - **README.txt**, which explains how to use your project.
  - **Five example source files** of your implemented language which should all run and execute successfully.
  - All **required source files** of your project.
- Use your **student ID** to identify yourself.
- Use your **student ID** to name your submission **ZIP**.
- **(Final) Deadline:** TBA, most probably around May 23rd / 26th.

Coursework 2: More Logistics

- You will be required to submit your work in **three separate submissions,** one for every two stages, starting from Week 6 (deadlines on Mondays to align with our labs).
- After each submission, you will receive formative feedback on your project.
- Draft submissions with guaranteed feedback until one week (168 hours) prior to the final deadline (no drafts for pre−final submissions).
- You can also receive formative feedback during our lab sessions.
- **Failure to build is graded with 0%.**
- **Failure to run your example code is capped at 40%.**
    - You are strongly encouraged to build / test your solution on a VM besides your own machine!

Coursework 2: Viva

- Assessment will also be based on a thorough **presentation** of your project.
- You will have to prepare a **15 minute presentation** (not much longer / shorter) addressing how you have tackled each of the stages / features you have implemented.
- You will also be handed a set of **questions** about your project that you will have to address in your viva.
  - Expect them around weeks 8 – 10 (right before Easter break).
- **Failure to submit a recording** of your presentation results in a **grade of 0%**.

Coursework 2: Allowed Tools

- You will have to build your language on top of an existing language of your choice.
- However, **you are NOT free to utilise everything** any existing language has to offer. In particular:
    - You cannot use **parsing libraries**, such as Haskell's Parsec, Python's pyparsing / Lark, Java's JFlex, etc.
    - You cannot use **evaluation functionalities** such as Python's eval(), Java's ScriptEngine, etc.
    - You cannot use anything that actually implements a feature you want to implement!
- As a rule of thumb, you should **build everything from scratch!**

Module Materials

Each week you will find materials from three different sources:

- Our “main” slides, which are baked according to our needs and are based on Robert Nystrom’s *Crafting Interpreters*.
  - This also includes our Python labs on crafting our own language.
- Mr. Perakis’s slides, which you can utilise as a more in depth analysis of several theoretical concepts.
- Mr. Windmill’s slides, which you can also use to further study some theoretical aspects of the module.

Further Reading / Resources

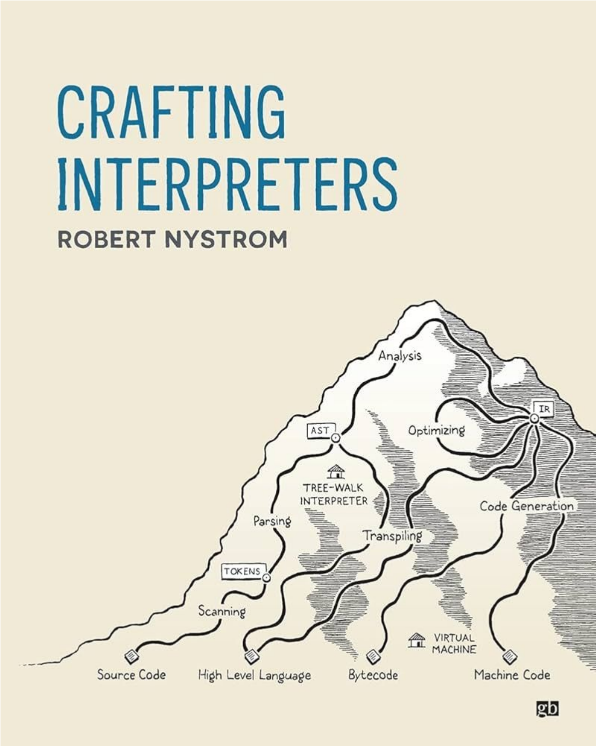
Some really useful resources include:

- **Crafting Interpreters**, Robert Nystrom, available here: https://craftinginterpreters.com/contents.html
- List of relevant **repositories** and **tutorials**: https://github.com/codecrafters-io/build-your-own-x?tab=readme-ov-file#build-your-own-programming-language
- **F# For Fun and Profit:** https://fsharpforfunandprofit.com/

Our Holy Bible

We will heavily rely on Rober Nystrom's *Crafting Interpreters*.

- You can find the book on our shared drive folder (use it for educational purposes only!).
- You are strongly encouraged to read it!
- Any shapes / figures etc included in the slides without any reference to their source are either crafted by me or borrowed shamelessly from that book.

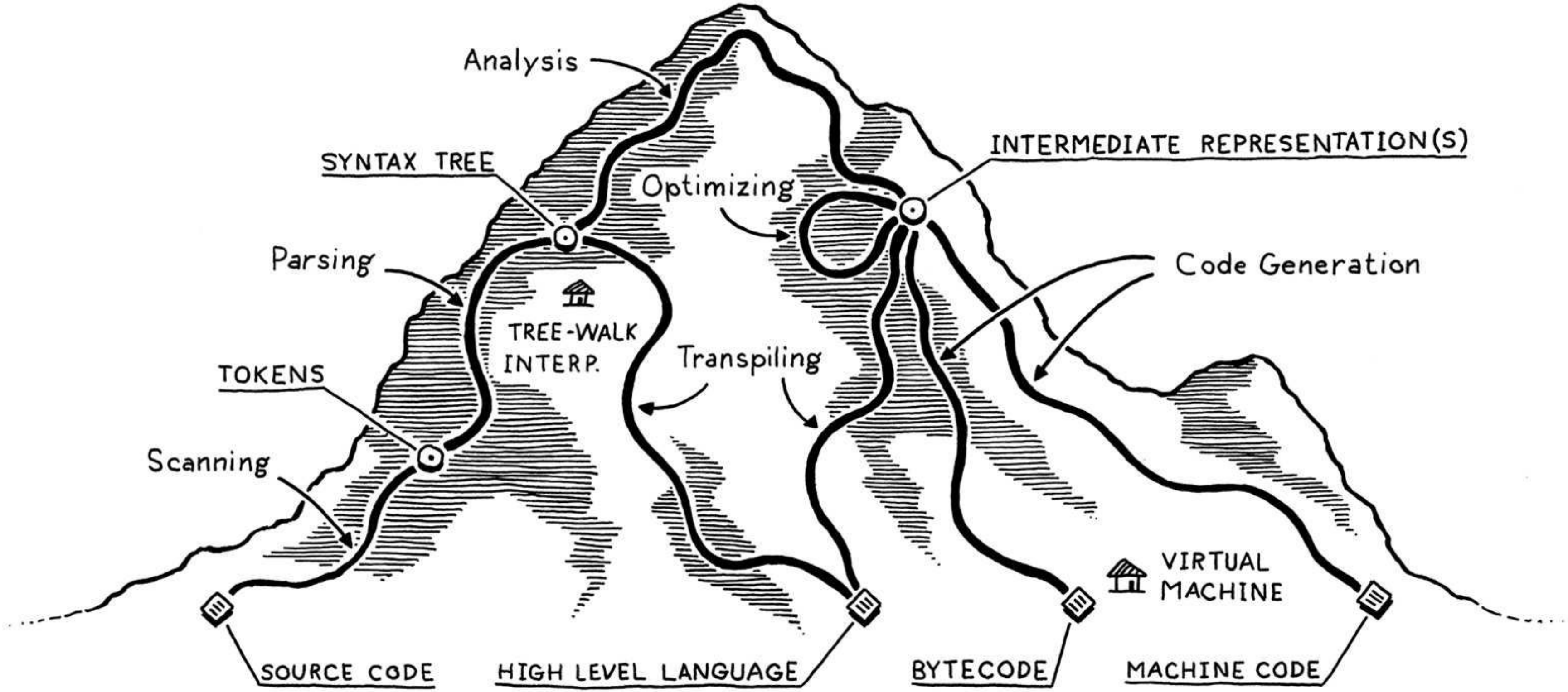


Before We Get Started…

- Expect this to be a **heavily lab–oriented module.**
- That is, we will work mostly on lab exercises in class, trying to implement Python on top of Python itself.
    - This already exists, it is called PyPy, and is marvellous.
    - We will follow our own track, however, mostly for teaching purposes.
- We will also **explore any theoretical concepts** as they come and might be useful to our implementation.
- Of course, you can **borrow ideas but not reuse** lab implementations per se in your projects.
- Feel free to also **work on** your own **Coursework 2** for this module during labs.

Dissecting A Language

The Trails of LDI



Scanning / Lexxing

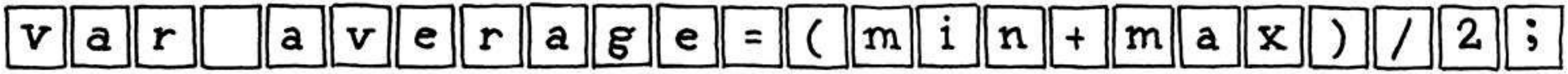
- You are given a Python script, say `foo.py`, which you are asked to execute.
- What is the first thing you should do?
  - Read it!
- The process of reading, i.e., making sense of the actual contents of the source code is often called scanning or lexing (< lex < λέξη).
- The output of a scanner / lexer is, typically, a list of tokens, so you can also silently read “scanning” as “tokenisation”.

Parsing

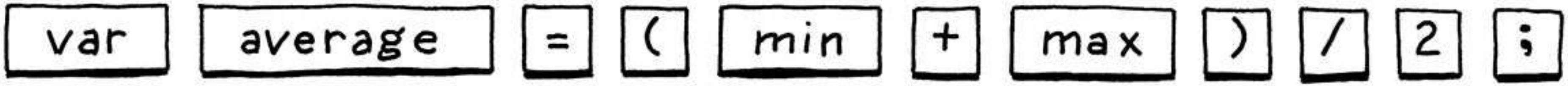
- Assume that you have now lexed your source code, which means that you have a list of tokens that might make more sense to you. What’s next?
  - You have to actually make sense of those, by capturing the subtle structure of the source code.
  - This process is called **Parsing.**
- Put more formally, parsing is the process of creating a data structure (typically an **Abstract Syntax Tree**, AST) that captures the syntactic structure of the underlying code.
- Often, ASTs are called simply Syntax Trees or, even, Trees.

Parsing Example

- Consider this piece of code:

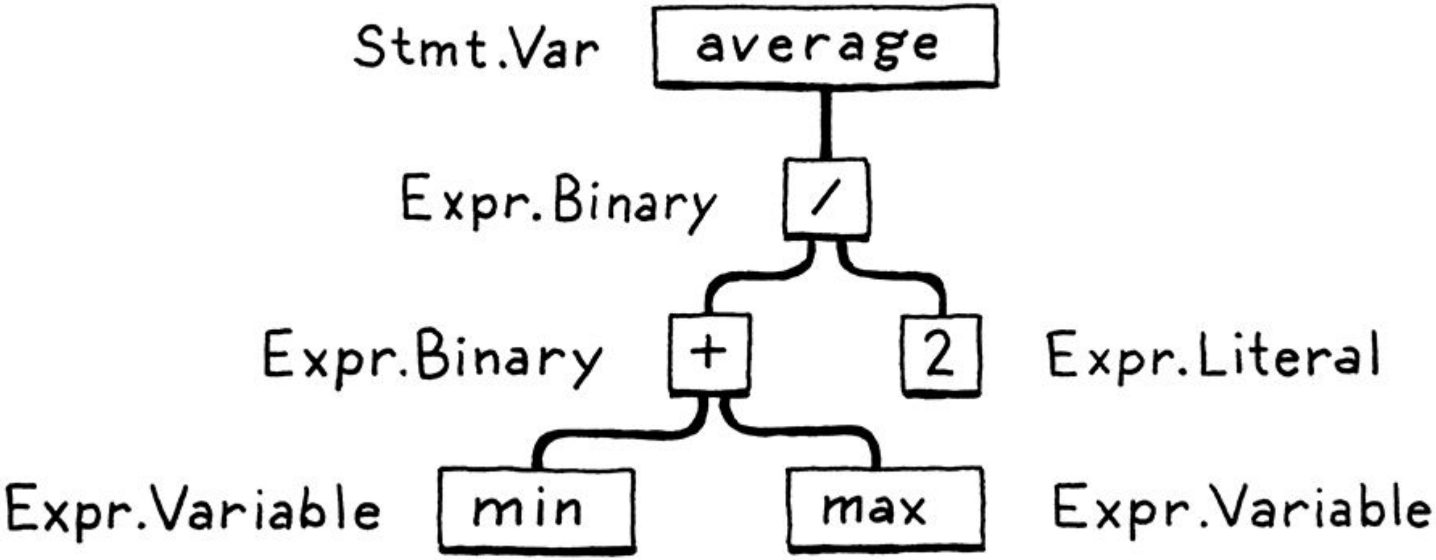


- How would you expect the lexxed version of this to look like?
- Most probably, something like that:



- What about the corresponding AST?

Parsing Example



Static Or Dynamic?

- Languages are split into two broad categories with respect to how they handle variable / data types:
  - **Statically typed** languages require data types (and scope, if needed) to be declared for each variable.
  - **Dynamically typed** languages do not require data types to be declared in any way, which means any types are handled at runtime.
- Going static gives one the chance to handle types at compilation time and, thus, prevent some bad stuff from happening at runtime, while going dynamic works the other way around.

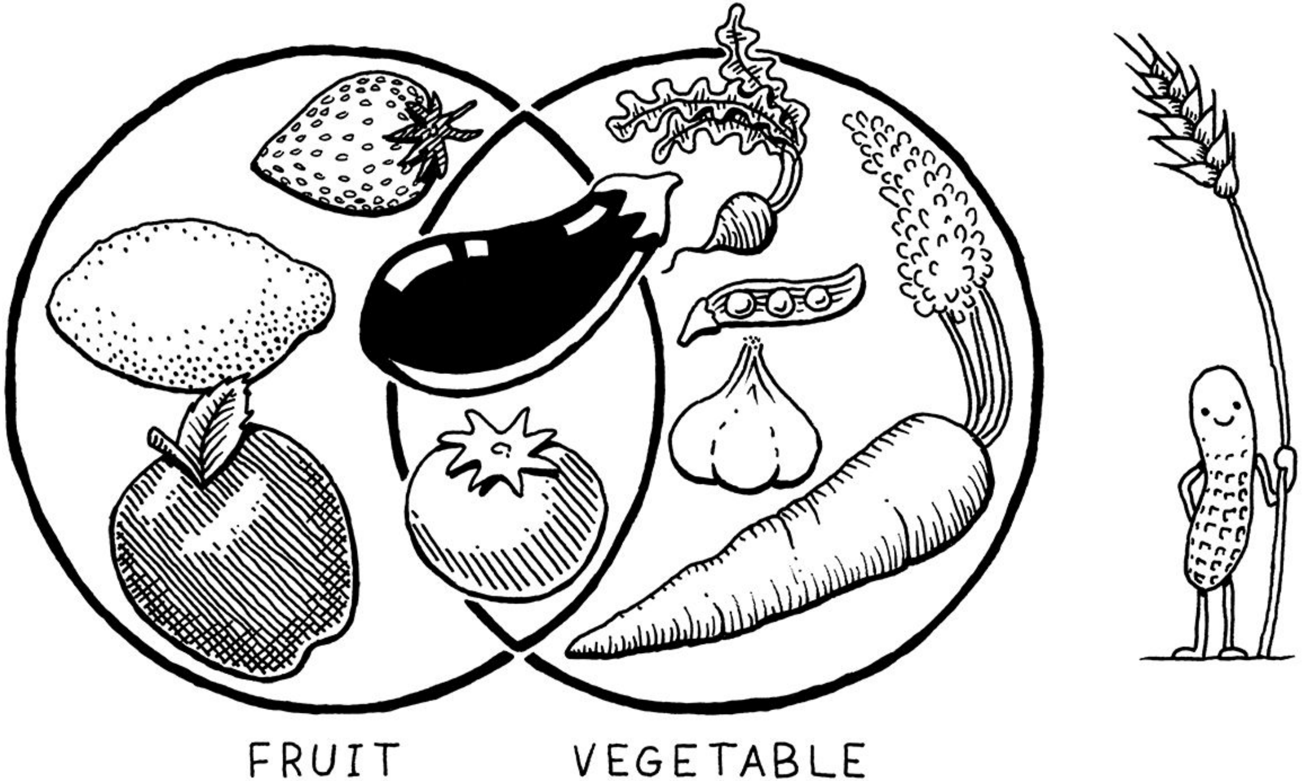
Beyond Parsing...

- Having parsed some source code, we have to go all the way down to generate some code that can be parsed by the machine.
- One simple choice would be to translate this directly to some sort of assembly–like instructions targeted to some certain machine.
  - But, this means that we cannot target many machines with our language.
- Another idea is to create some **Intermediate Representation** (IR), which we then map to any assembly instruction set we want to.
  - For instance, Python bytecode is such an IR.
- One step further, we can generate Virtual Machine (VM) code, i.e., code for a hypothetical (virtual) chip on which our compiled instructions will run.
  - This makes our language as portable as installing our VM on another machine.

Strange Beasts And Where To Find Them

- **Single–pass compilers** merge scanning, lexing, parsing and code generation at one single stage:
  - This is a really resource light approach, adopted by C and Pascal.
  - However, this is not relevant in an era of abundant resources, as it severely restricts the capacity of our compiler / language.
- **Tree–walk interpreters** execute code right after generating the AST.
  - Useful for little languages but not for larger projects.
- **Transpilers** are language implementations on top of other languages.
- **Just–In–Time (JIT)** compilers compile source directly to the machine's native instruction set, with performance as a primary desideratum.

Tomato == Fruit? Tomato == Vegetable?

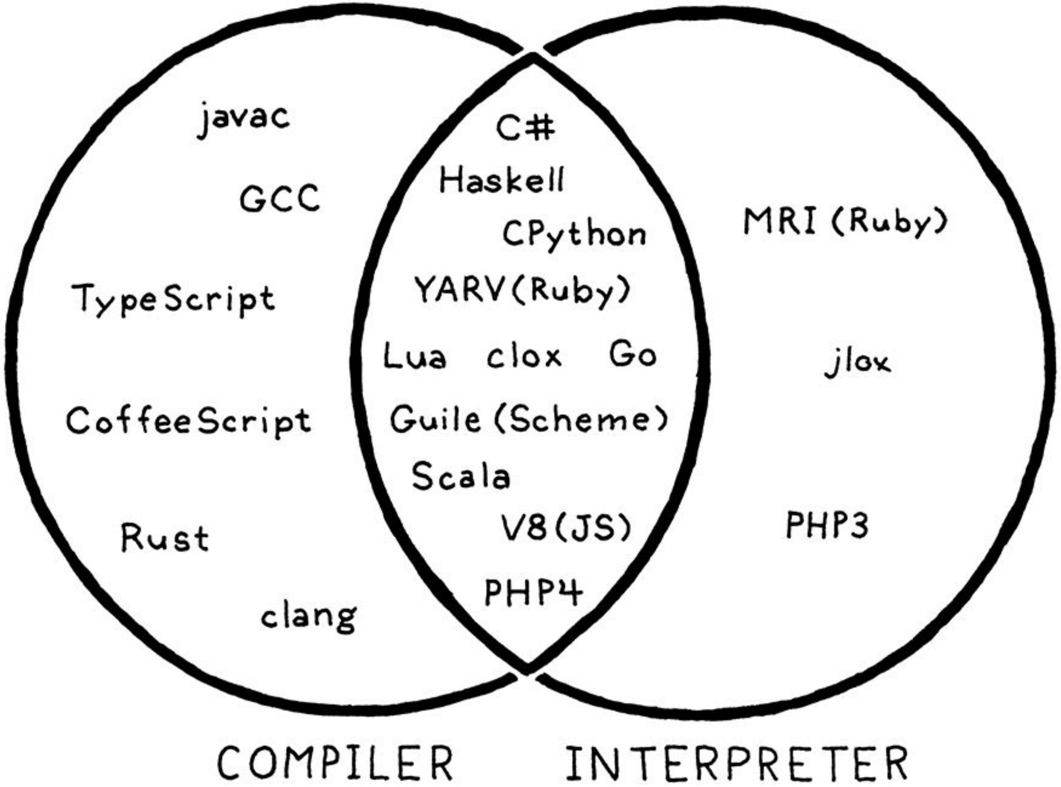


Compilers vs Interpreters

As discussed in the past:

- A **language** is not compiled or interpreted.
- A **language implementation** is compiled or interpreted!
- **Compiling** is a way to implement a language involving translating to another (typically lower–level) target language.
- So, a **compiler** is a program that implements compiling.
- An **interpreter** takes the source code and executes it immediately.
- Is Python compiled or interpreted?
    - Well… Both. (look around the web for that, we have also discussed it in the past)

Compiled / Interpreted Languages



Fun Time!

Scanning Or Lexing?

- To get a taste of what building a language feels, start working on our first lab, which will also help you organise your work for Coursework 2.
- You can find this lab at:

```
./labs/lab_01.pdf
```

- You can also find relevant resources at:

```
./source
```

Any Questions?

Office Hours:
TBA

Don’t forget to fill-in the questionnaire (check right)

https://forms.gle/aj7pYsK1ksufD6xYA