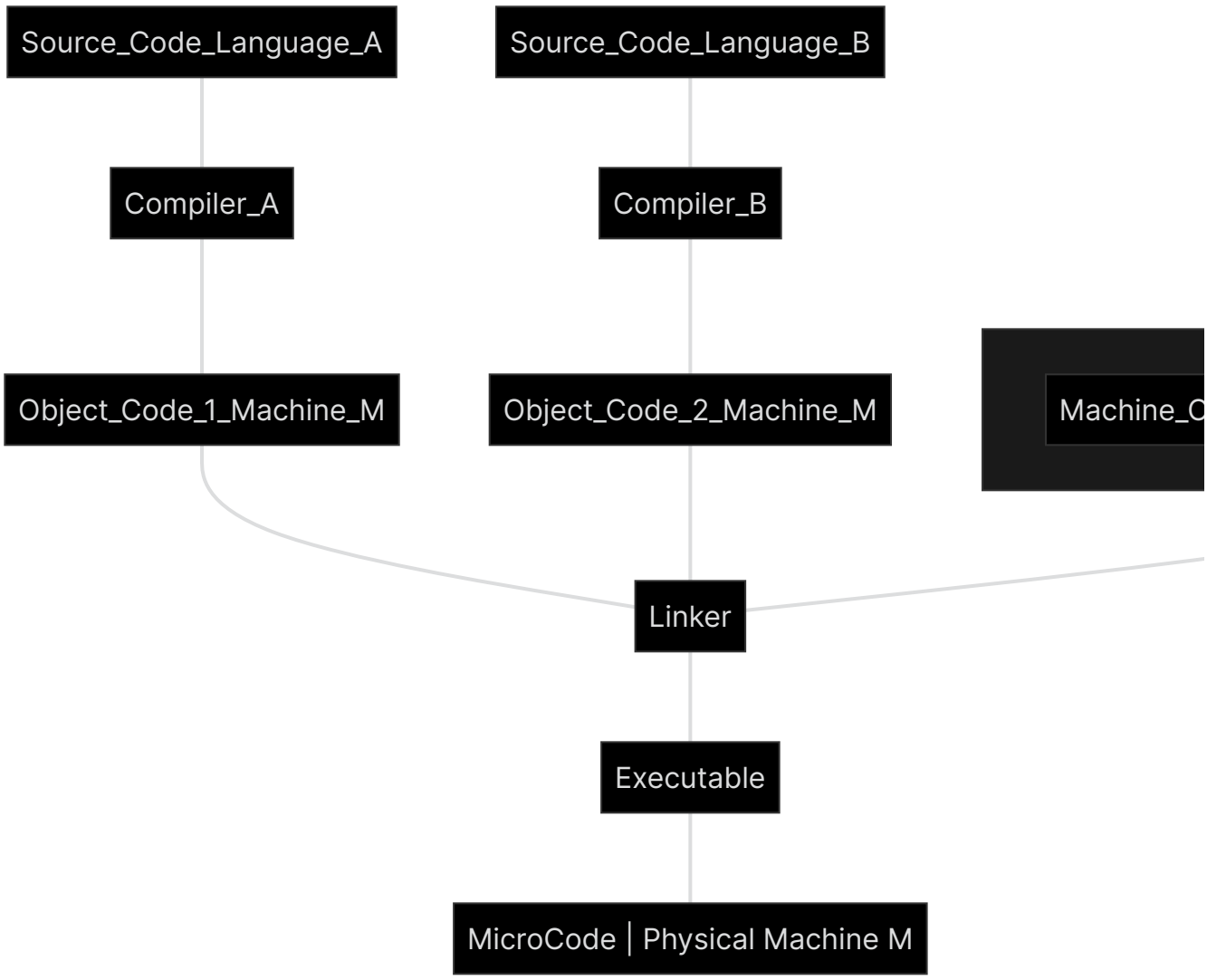
Types of Problems:

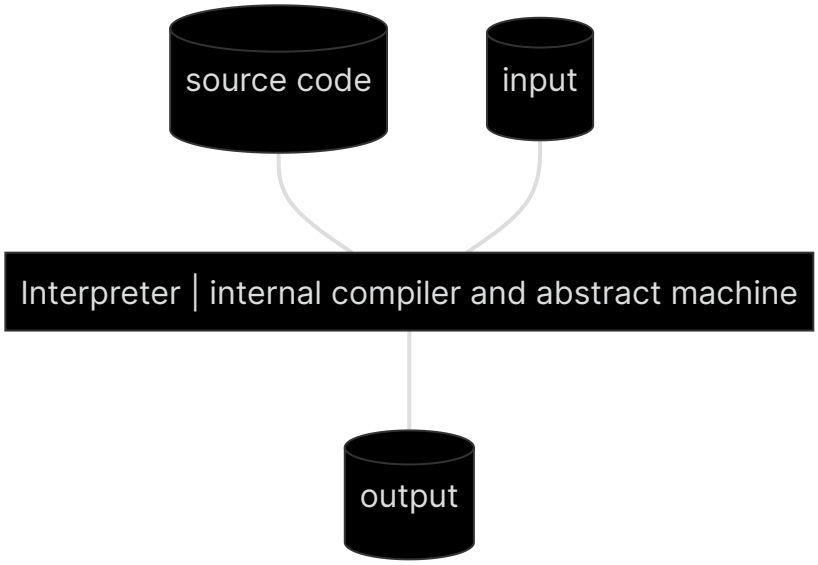
- Short Answer
- Derive a REGEX, FSA, or Grammar for a language
- Derive a language from FSA, Grammar, or REGEX
- Apply definitions covered in topcis below
- Classify grammars in Chomsky Hierarchy
- Apply set theory operations for reasoning about languages

Topics

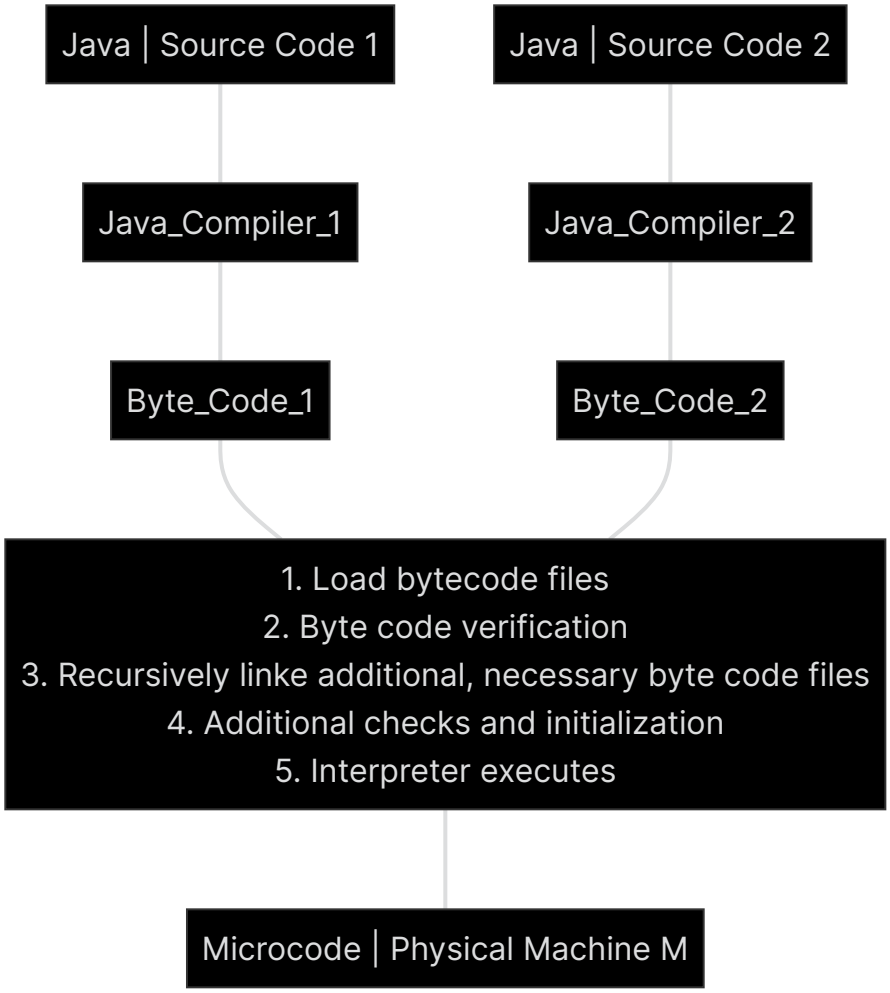
- **Be able to define and/or explain the concepts of compiler, interpreter, hybrid compiler-interpreter, linkers, virtual machines, source code, object code, target code, executable code, static library; know how these terms are related and be ready to give examples of languages and their development environments**
    - **Compiler**:
        - *A translator that translates the source code into code expressed in another language called Target Code.
        - *Generate very low-level code for the most part*
        - *Is a problem itself and must be written in a language*
        - *Can have multiple target languages assigned to one source code*



- - **Cross-compilers**:
    - *Compilers that generate code for a machine that is not the same as the current machine being worked on ie.* $M_1 \neq M_2$
  - **Interpreter**:
    - Also includes a scanner and a parser.
    - The input language is not translated to a language external to the interpreter code.
    - An AST is generated to emulate semantics of the construct
    - Often implemented in a Read-Evaluate-Print-Loop (REPL) environment. bash or Powershell operate this way
    - Code execution often slower than compilation



- **Hybrid compiler-interpreter**
    - Target code resembles that of compilers but is instead interpretetd by a virtual machine.
    - Target code is called byte-code and the most widely-used language with such a scheme is Java.
    - Java's byte code is interpreted by the Java virtual machine or JVM.
    - Python also has a similar build process as Java and uses the Python Virtual Machine as well as it's own associated byte-code.



- **Linker**
    - Used to resolve object files with unresolved operand addresses.
    - The final phase of building a program
- **Virtual machine**
    - Abstract software-based machine that interptrets target code generated by hybrid compiler-interpreter
- **Source code**
    - *Code that is to be translated*
- **Object code**
    - *Stored in object files*
    - *Target code for a machine that has unresolved operand addresses (such as location of function calls)*
- **Target code**

        - *The code that has been translated from another source code. Can be expressed in another language compared to the source code. The language of the target code is called the target language*
    - **Executable code**
    - **Static library**

- # Know the major phases and data structures/models used in compilers:
    - **Lexical analysis/scanner: (token stream)**
        - *Involves reading the source code character by character, eliminating whitespace (space, tab, newline characters), removing comments and recognizing lexemes.*
        -
    - **Parser/syntax analyzer/parse: parse tree**
        - *Parser analyzes grammatical structure of source code and, if syntax is correct, produces a parse tree representation of syntactical structure.*
        -
    - **Semantic analyzer and intermediate code generation: abstract syntax tree**
        - *AST produced by syntax analysis is traversed.*
        -
    - **Machine independent code optimization (optional): (e.g. 3-address instructions)**
    - **Code generator target code generator (e.g. assembler)**
    - **Modified target code generator**
- # Know how the Simple_PL1 scanner is designed and how tokens and lexemes are represented and communicated to the parser. Know the difference between a lexeme and a token.
    - **Lexeme**
        - *Lexemes are meaningful strings that are treated as "atomic" in a language like* `sum`, `<=`, *or* `for`
- # Know definitions related to Chomsky grammars, such as definitions of the terms: Kleene closure, languages, productions, derives, sentential forms, sentences, non-terminals, terminals

- **Kleene closure**
  -
- **Languages**
  - *A language, L(G), generated by a grammar G, is defined as the set of sentences:*

$$L(G) = a \in V_T^* | S \implies {}^+\alpha$$

- **Productions**
  - *Not derivations, denoted with the $\rightarrow$ symbol to show the base cases for which a grammar can produce. Ex;*
    - *Assume grammar with following productions:*

> S $\rightarrow$ AB
> A $\rightarrow$ a
> B $\rightarrow$ b

    - *A $\rightarrow$ a is a production just as B $\rightarrow$ b is one*
- **Derives**
  - *If $\alpha \rightarrow \beta$ is a production and $\omega\alpha\phi$ is a string where $\omega, \phi \in V^*$* and $\alpha \in V^+$, then $\omega\alpha\phi \implies \omega\beta\phi$ is an immediate or direct derivation*
    - *Denoted with $\implies$ symbol, not production, shows the derived cases from a production*
    - *Assume the following grammar:*

> S $\rightarrow$ AB
> A $\rightarrow$ a
> B $\rightarrow$ b

      - *We could show a derivation as follows:*

$$S \implies AB \implies aB \implies ab$$

        - *Or:*

$$S \implies AB \implies Ab \implies ab$$

        - *Both are valid derivations*
- **Sentential forms**
  - *A string α derivable from the start symbol, S*
- **Sentences**
- **Non-terminals**
  - *V_N is the nonempty finite set of symbols called non-terminals*
  - *In our cases, non-terminals have been lowercase, or non-uppercase letters*
- **Terminals**
  - *V_T is the nonempty finite set of symbols called terminals where $V_N \cap V_T = \emptyset$*

- *In our cases, terminals have been upper-case letters*

• Be able to classify the most restrictive grammar class according to the form of production rules.

- **Phrase-Structured Grammar (type 0):**
  - *Productions of the form* $\alpha \rightarrow \beta$
  - *Where* $\alpha \in V^+$ *and $\beta \in V^$.**
  - *Also known as unrestricted*
  - Example of Phrase-Structured Grammar:

$$\begin{array}{lrcl} G3:(PSG) & S & \rightarrow & Sc \mid abS \\ & abSc & \rightarrow & abbAcc \\ & bA & \rightarrow & b \end{array}$$

    -
    - There are multiple symbols on the left side for the production abSc $\rightarrow$ abbAcc therefore the grammar cannot be Regular or Context-Free. The production bA $\rightarrow$ b has $|\beta| < |\alpha|$ therefore it cannot be Context-Sensitive which leaves the only option being Phrase-Structure Grammar
- **Context-Sensitive Grammar (type 1):**
  - Productions of the form $\alpha \rightarrow \beta$ where $\alpha \in V^+, \alpha \neq \beta, and |\beta| \geq |\alpha|$
  - Example of Context-Sensitive Grammar:

$$\begin{array}{lrcl} G6:(CSG) & S & \rightarrow & cE \mid cS \mid dS \mid b \\ & cE & \rightarrow & cdS \mid cd \end{array}$$

    -
    - There are 2 symbols on the left with the productions cE $\rightarrow$ cdS | cd therefore the grammar cannot be Context-Free or Regular.
- **Context-Free Grammar (type 2):**
  - Productions of the form: $A \rightarrow \beta$ where $A \in V_N$ and $\beta \in V^+$.
  - Example of Context-Free Grammar:
    - $$G1:(CFG) \qquad S \rightarrow aSb \mid ab$$
    - The production S $\rightarrow$ aSb does not follow the form of either a right linear or left linear grammar thefore the grammar cannot be Regular.
    - S $\in V_N$, aSb $\in V^+$, ab $\in V^+$ also follows the form of Context-Free Grammar therefore the grammar can be considered such
- **Regular (or linear) Grammar (type 3):**
  - 2 production forms where left linear is:$A \rightarrow Ba$ or $A \rightarrow a$ where $A, B \in V_N$ and $a \in V_T$
  - Right linear is: $A \rightarrow aB$ or $A \rightarrow a$ where $A, B \in V_N$ and $\alpha \in V_T$
  - Example of Regular Grammar

$$\begin{array}{lrcl} G5:(RRG) & S & \rightarrow & aU \mid bV \\ & U & \rightarrow & bS \mid b \\ & V & \rightarrow & aS \mid a \end{array}$$

    -

- $S, U, V \in V_N$ and $a, b \in V_T$ therefore all the productions follow the form of a regular grammar. The productions also follow the form of a right linear grammar

- **Know the difference between the classification of Chomsky languages and classifications of Chomsky grammars and the relationship between the languages and grammars.**
- **Given grammar, G, describe L(G) using set notation and exponent notation.**
  - G1: $G1 : (CFG) \qquad S \rightarrow aSb \mid ab$, to describe L(G1) we can simply produce a ton of derivations for now and observe the patterns
  - $$S \implies aSb \implies aaSbb \implies aaaSbbb \implies aaaabbbb$$
  - Based off of these derivations, we can see that the number of a's stays equal to the number of b's, therefore with set notation we can say that: $\{a^n b^n | n > 0\}$
- **Be able to construct languages (remember: they are sets of strings) using set union, set intersection and set difference; and subset-of and member-of predicates.**
- **Know how to express grammars in BNF, EBNF, and Syntax flow diagrams.**
  - **BNF**:
    - *Backus-Naur Form, notation for expressing grammars*
    - **
- **Given a description of a Regular Language (RL) via English, or Regular Expression (RegEx), create an NFA or DFA M such that RL = L(M).**

- **Given a DFA or NFA M or English description of an RL give a RegEx of that RL**
- **Given a RegEx denoting the language LregEx, give a DFA, M, such that LregEx = L(M).**
- **Be able to show a sentence is ambiguous or a grammar is ambiguous. Define what inherently ambiguous means.**
    - An ambiguous sentence is a sentence that has two or more distinct leftmost/rightmost derivations with respect to a grammar.
    - An ambiguous grammar is a grammar that generates at least one ambiguous sentence.
    - An language is inherently ambiguous iff there are no unambiguous grammars that generate the language.
- **Show how a grammar can be used for syntax-directed translation where the parse trees correspond to easier synthesis of target code according to the semantics of the language, such as precedence or associativity of arithmetic operators, control constructs (while, if-then-else, …, etc.).**
    -
- **Know how to create and read (use) a context free grammar expressed with BNF or EBNF notation.**
    -
- **Be able to draw parse trees for leftmost and rightmost derivations.**
    - Rightmost just involves deriving the right most term first and then working from right to left

- Leftmost just involves deriving the left most term first and then working from left to right

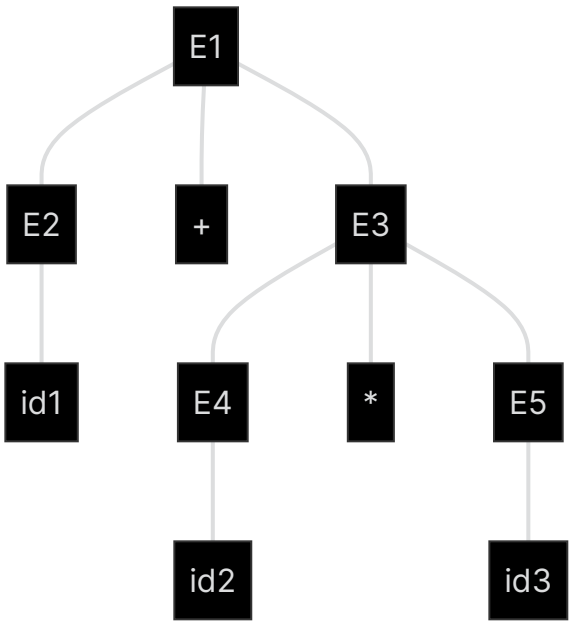
$$G =< \{E\}, \{*, +, id\}, P, E > \textit{ where } P \textit{ is:}$$

$$E \rightarrow E + E \mid E * E \mid (E) \mid id$$

- Given a leftmost derivation could be

$$E \implies E + E \implies id + E \implies id + E * E \implies id + id * E \implies id + id * id$$

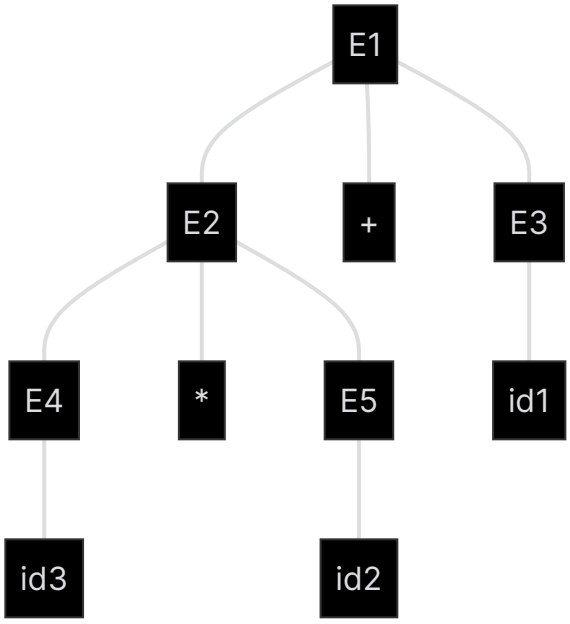
the parse tree could then be taken from this directly where we put the root as the starting node and from there we can follow the progression very linearly where E goes to E, +, and E, each E goes to it's respective terms.



the rightmost derivation would just be taking the rightmost term and deriving it just as we would with a leftmost derivation. This same derivation can be expressed as

$$E \implies E + E \implies E + id \implies E * E + id \implies E * id + id \implies id * id + id$$

and the parse tree would look something like this:


E1
E2
+
E3
E4
*
E5
id1
id3
id2