Art
All

CST8152 Compilers

Algonquin College

Computer Engineering Technology

CST8152 Compilers

Fall, 2023

Art All

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

Prof. Paulo Sousa

Algonquin College

Computer Engineering Technology

CST8152 Compilers

Fall, 2023

Art All

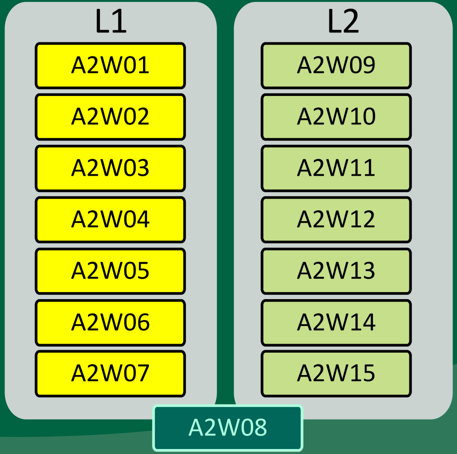
Revision

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

Prof. Paulo Sousa

Art All: Inside Compilers

- *General View*
- *Hybrid Compilation*



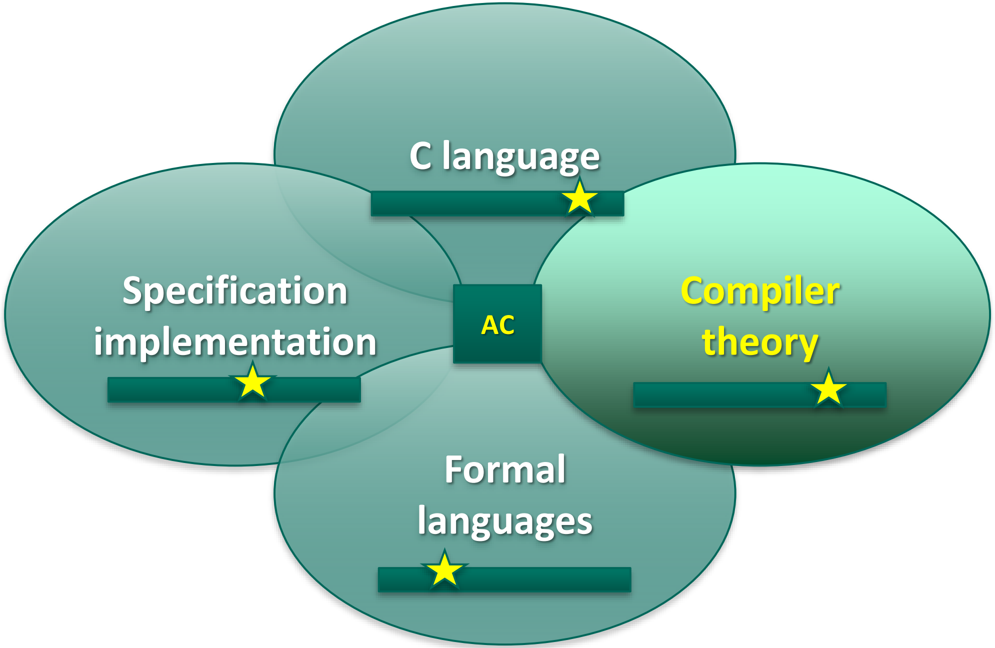
ALGONQUIN COLLEGE

Compilers – Lect. All

General View

ALGONQUIN COLLEGE

Let's start…


C language
Specification implementation
AC
Compiler theory
Formal languages

1.1. Initial Concepts

- A **Compiler** is a program that runs on some computer architecture under some operating system and transforms (translates) an input program (source program) written in some programming language into an output program (target program) expressed in different programming language.
- A **Programming Language** is a notational system for describing computations in machine-readable and human-readable form.

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

1.1. Initial Concepts

- **Computation** in general is any process that can be carried by a computer. Programming Languages must provide two types of abstractions:
  - **Data abstractions** and
  - **Control abstractions**.

Note:

- Niklaus Wirth (Pascal language creator) defined a Programming language as: **Data structures + Algorithms.**
- Decades after, **OO** defined entities as composed by **properties** and **methods**.

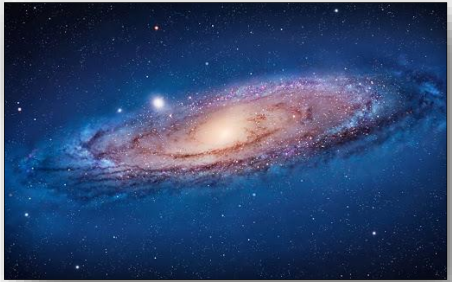
Source: https://towardsdatascience.com/top-10-in-demand-programming-languages-to-learn-in-2020

1.4. DSL (Domain Specific Languages)

Definition:

A **domain-specific language** (DSL) is a computer language specialized to a particular application domain.

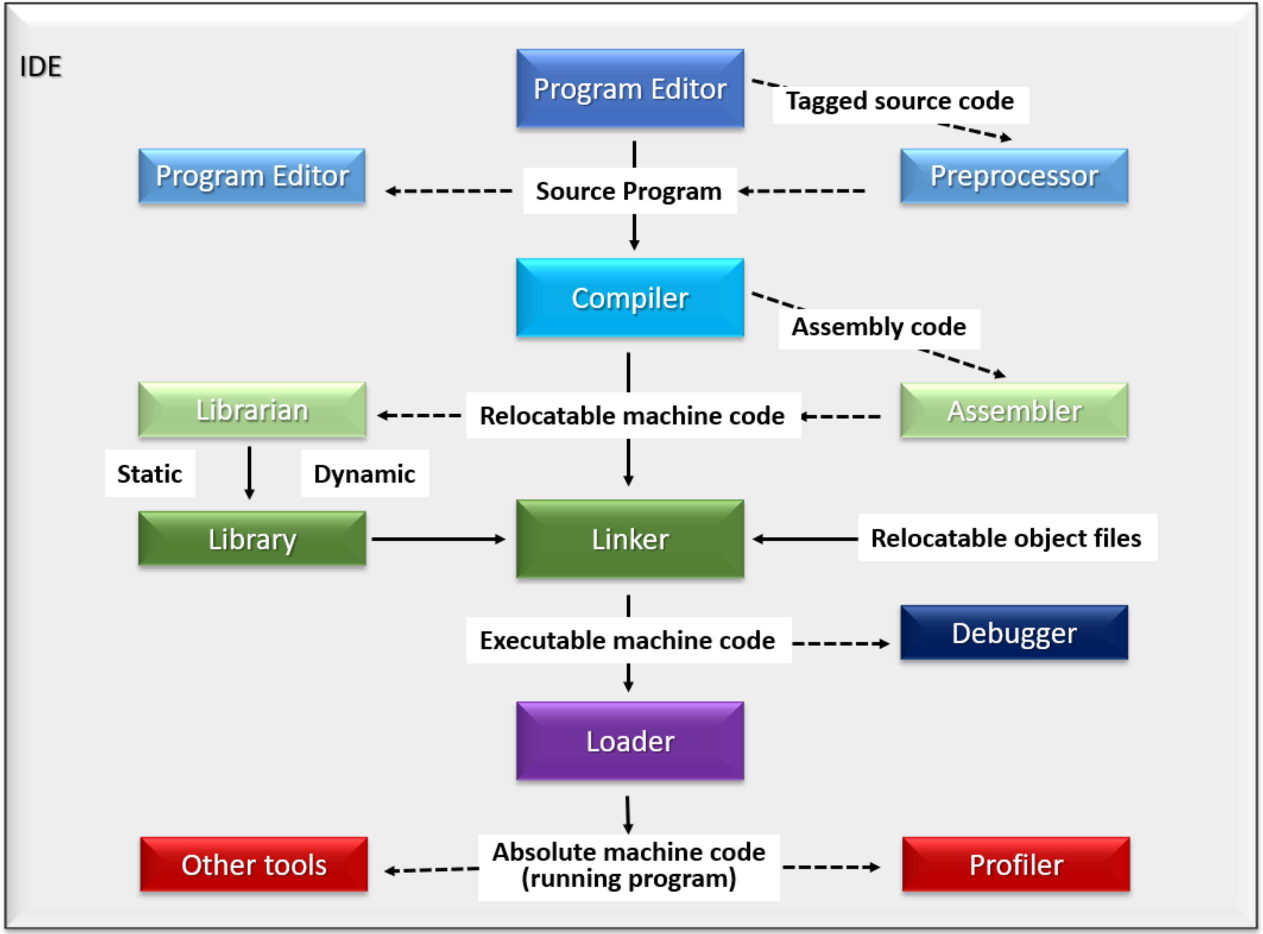
- **Note:**
  - This is in contrast to a general-purpose language (GPL), which is broadly applicable across domains.
- **Design Goals:**
  - Domain-specific languages are less comprehensive.
  - Domain-specific languages are much more expressive in their domain.
  - Domain-specific languages should exhibit minimal redundancy.



Examples:

OS Shells, Wiki environments, OpenGL, Markup Languages…

4.1 - Review



IDE
Program Editor
Tagged source code
Program Editor
Source Program
Preprocessor
Compiler
Assembly code
Librarian
Relocatable machine code
Assembler
Static
Dynamic
Library
Linker
Relocatable object files
Executable machine code
Debugger
Loader
Other tools
Absolute machine code
(running program)
Profiler

Resume

SINGLE PASS COMPILER

VERSUS

MULTIPASS COMPILER

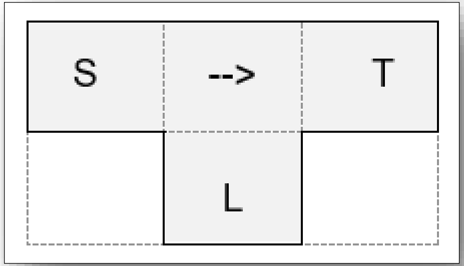
SINGLE PASS COMPILER	MULTIPASS COMPILER
A type of compiler that passes through the parts of each compilation unit only once, immediately translating each code section into its final machine code	A type of compiler that processes the source code or abstract syntax tree of a program several times
Faster than multipass compiler	Slower as each pass reads and writes an intermediate file
Called a narrow compiler	Called a wide compiler
Has a limited scope	Has a great scope
There is no code optimization	There is code optimization
There is no intermediate code generation	There is intermediate code generation
Takes a minimum time to compile	Takes some time to compile
Memory consumption is lower	Memory consumption is higher
Used to implement programming languages such as Pascal	Used to implement programming languages such as Java

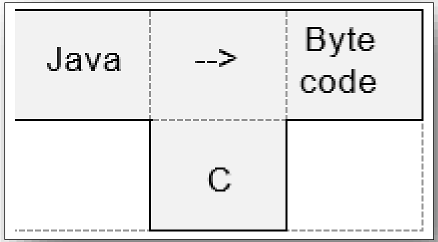
Visit www.PEDIAA.com

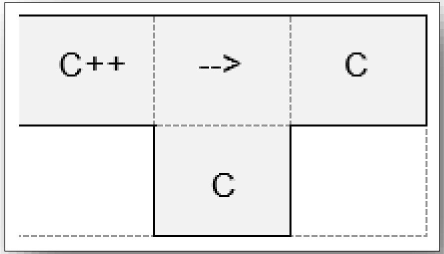
11

Current Development

- **Diagrams:**







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

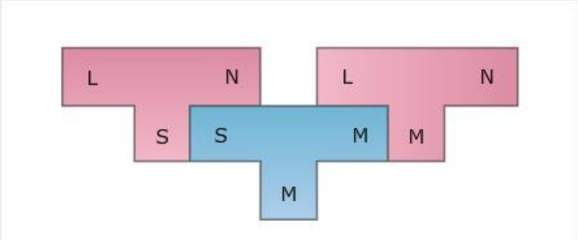
12

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.



Cross Compiler

Hybrid Compilation

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

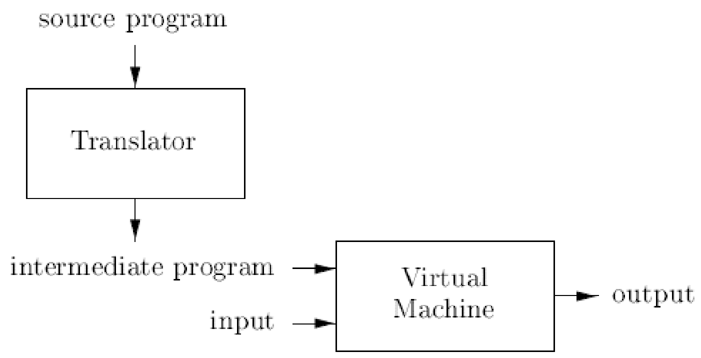


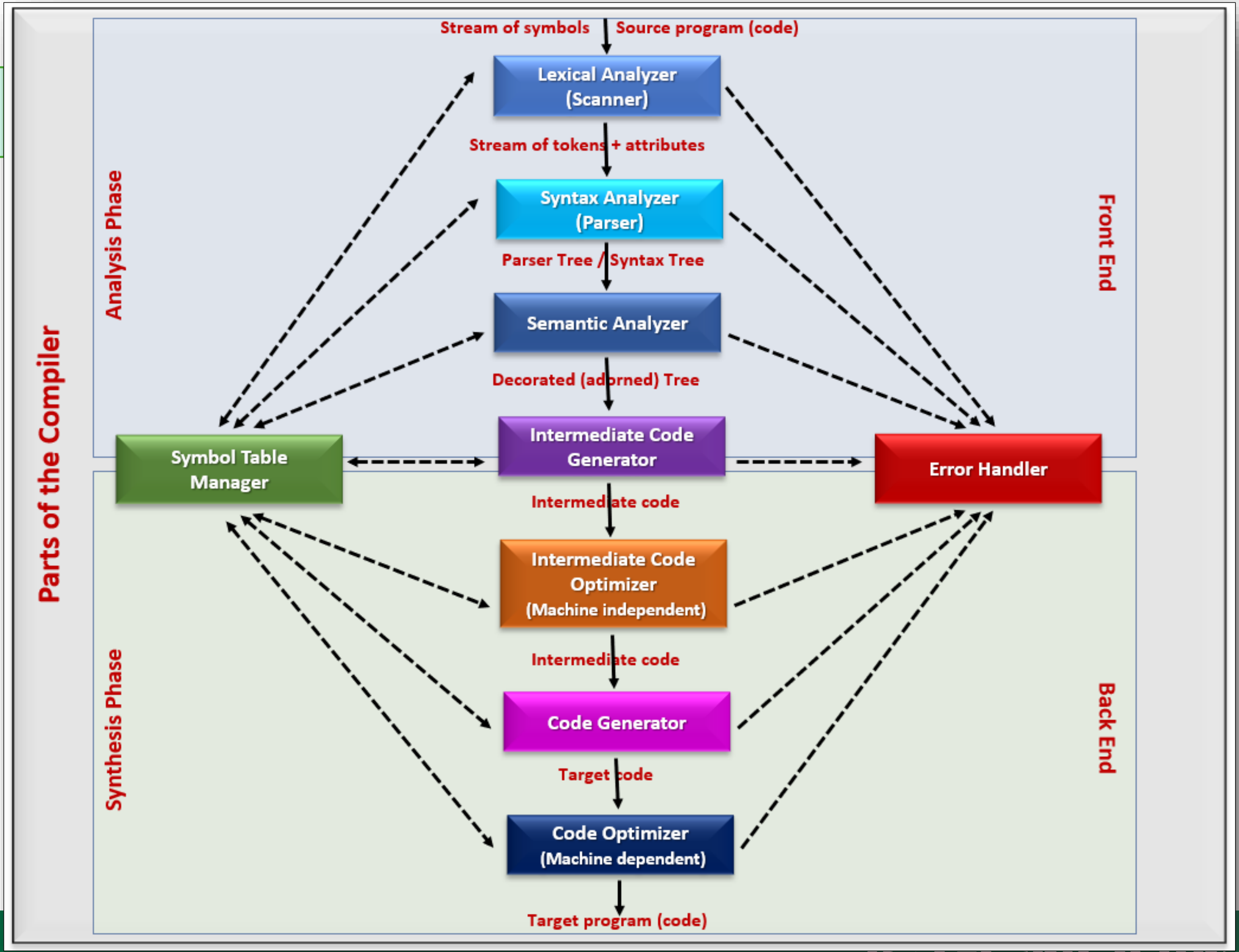
Figure 1.4: A hybrid compiler

Example:

- Java Code:

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

4.1 - Review



Parts of the Compiler
Analysis Phase
Synthesis Phase
Front End
Back End
Stream of symbols
Source program (code)
Lexical Analyzer (Scanner)
Stream of tokens + attributes
Syntax Analyzer (Parser)
Parser Tree / Syntax Tree
Semantic Analyzer
Decorated (adorned) Tree
Intermediate Code Generator
Symbol Table Manager
Error Handler
Intermediate code
Intermediate Code Optimizer (Machine independent)
Intermediate code
Code Generator
Target code
Code Optimizer (Machine dependent)
Target program (code)

2.3. General View

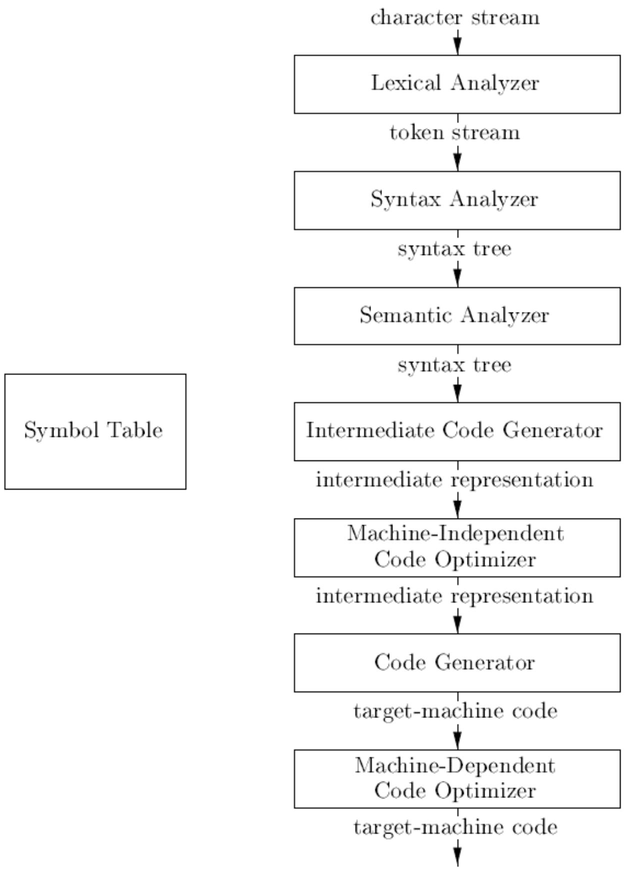
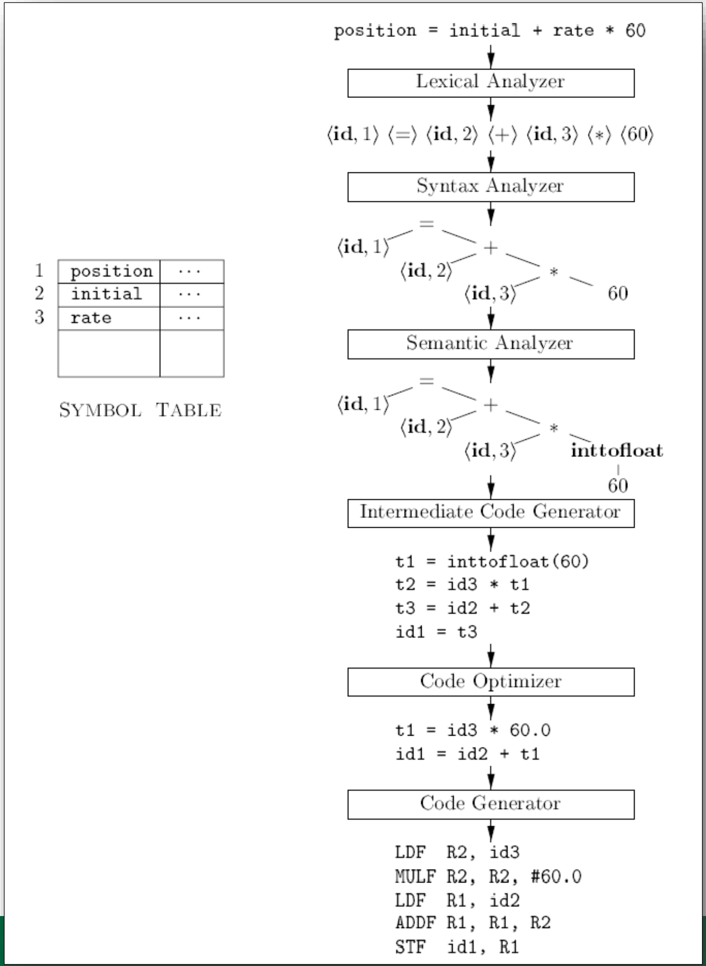


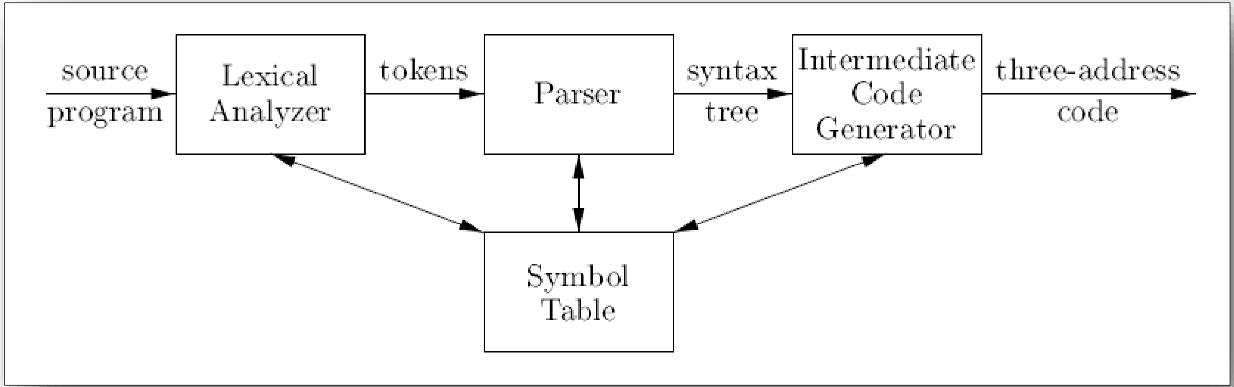
Figure 1.6: Phases of a compiler

2.3. General View



More about front-end compiler

- **Diagram:**



- Note that ST (Symbol Table) is always used and updated.

More about front-end compiler

- See the buffer…

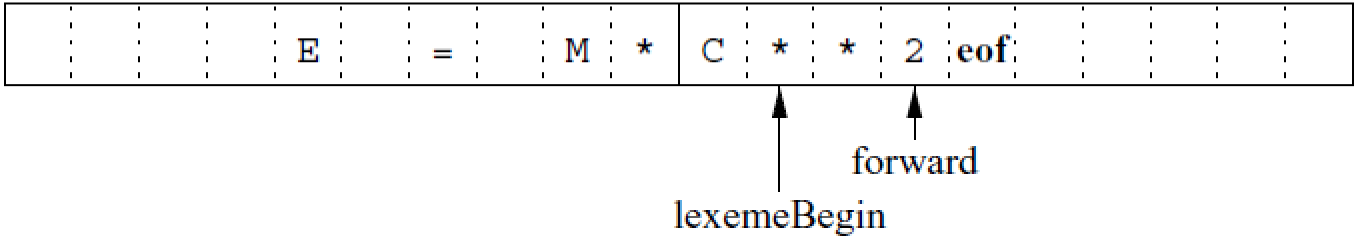


Figure 3.3: Using a pair of input buffers

ALGONQUIN COLLEGE

More about front-end compiler

- **Connecting concepts…**

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

ALGONQUIN COLLEGE

Compilers – Art. 4

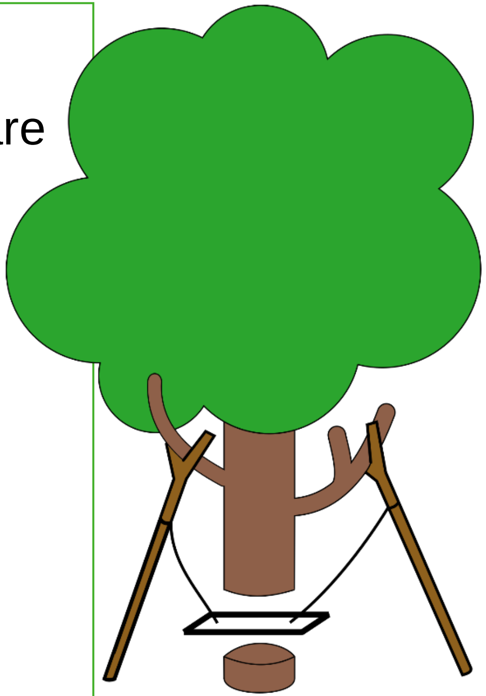
General Purpose Languages

4.2. The "ontological" problem

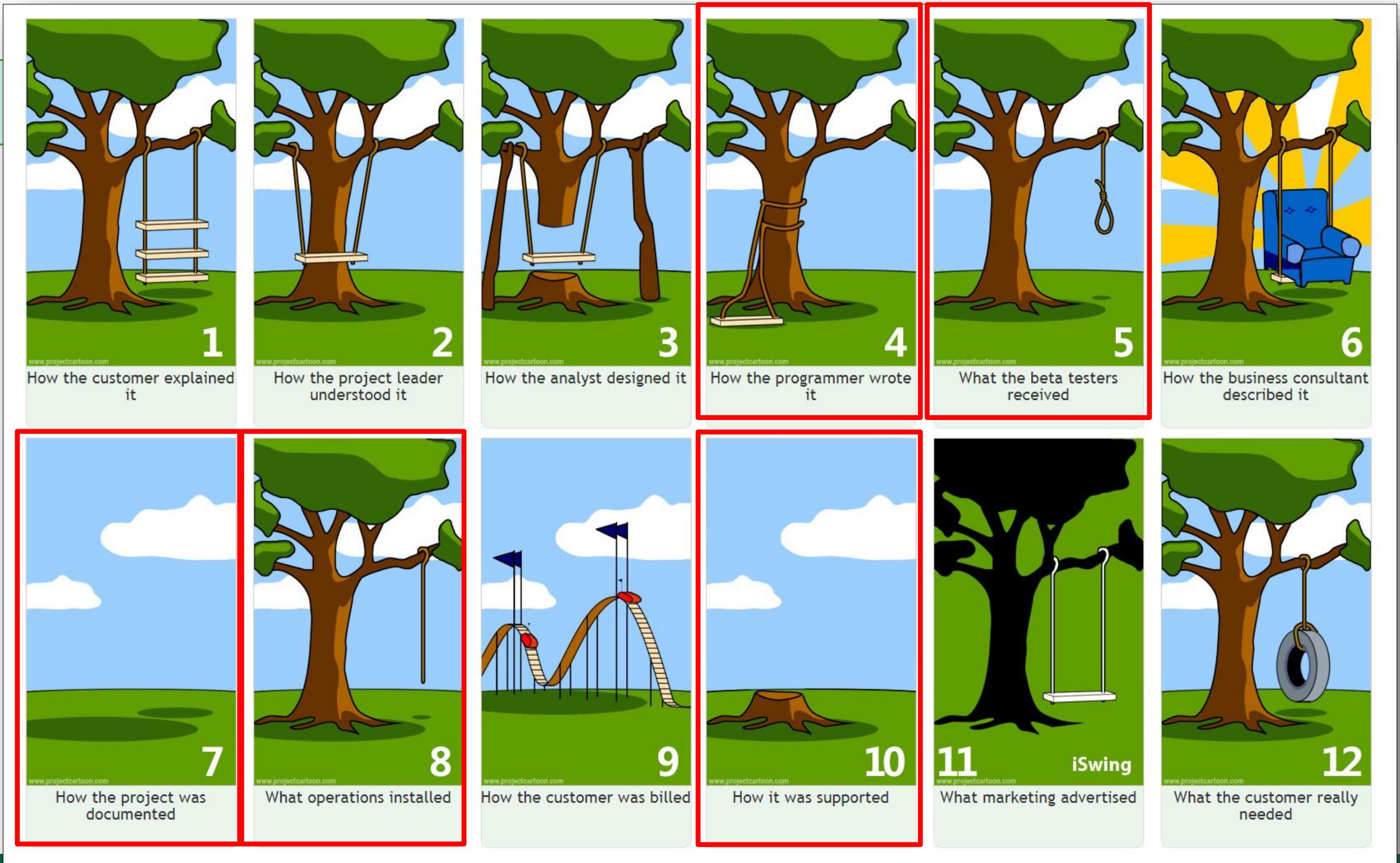
GPL (General Purpose Languages)

- Should be able to create "artefacts" for software development;
- But software needs to attend business needs;
- Business are domain-specific

Note: Part of the objective of SE (Software Engineering) is decrease the "gap" between the idea and the implementation…



I know!



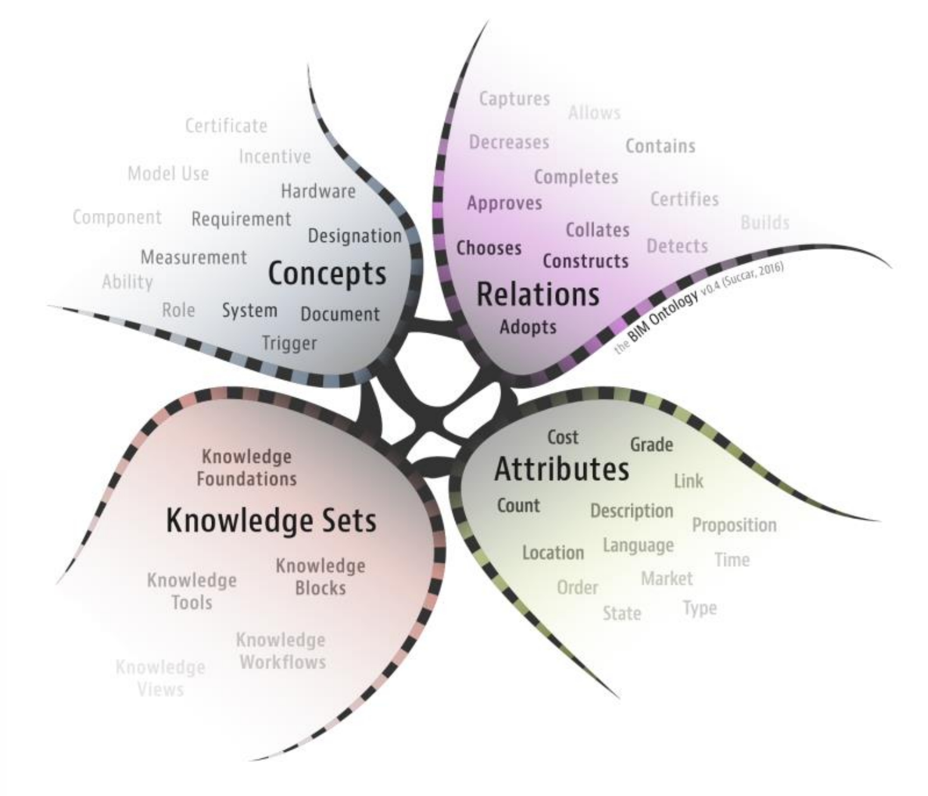
1
How the customer explained it
2
How the project leader understood it
3
How the analyst designed it
4
How the programmer wrote it
5
What the beta testers received
6
How the business consultant described it
7
How the project was documented
8
What operations installed
9
How the customer was billed
10
How it was supported
11
iSwing
What marketing advertised
12
What the customer really needed

4.2. The "ontological" idea

How to explain languages…

- For a "normal" people;

Note: Even OO is not a real "paradigm" to users, but for programmers.

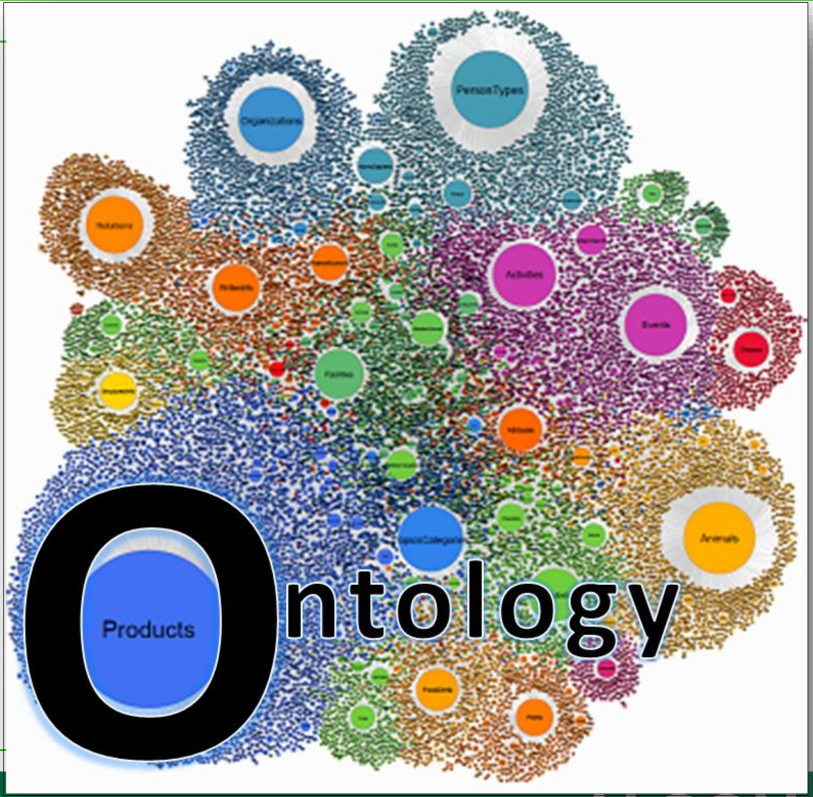


4.2. The “ontological” idea

How to explain languages…

Note: Is it possible to create high-level languages?

Imagine the complexity of concepts and associations required to implement “ontological” languages…

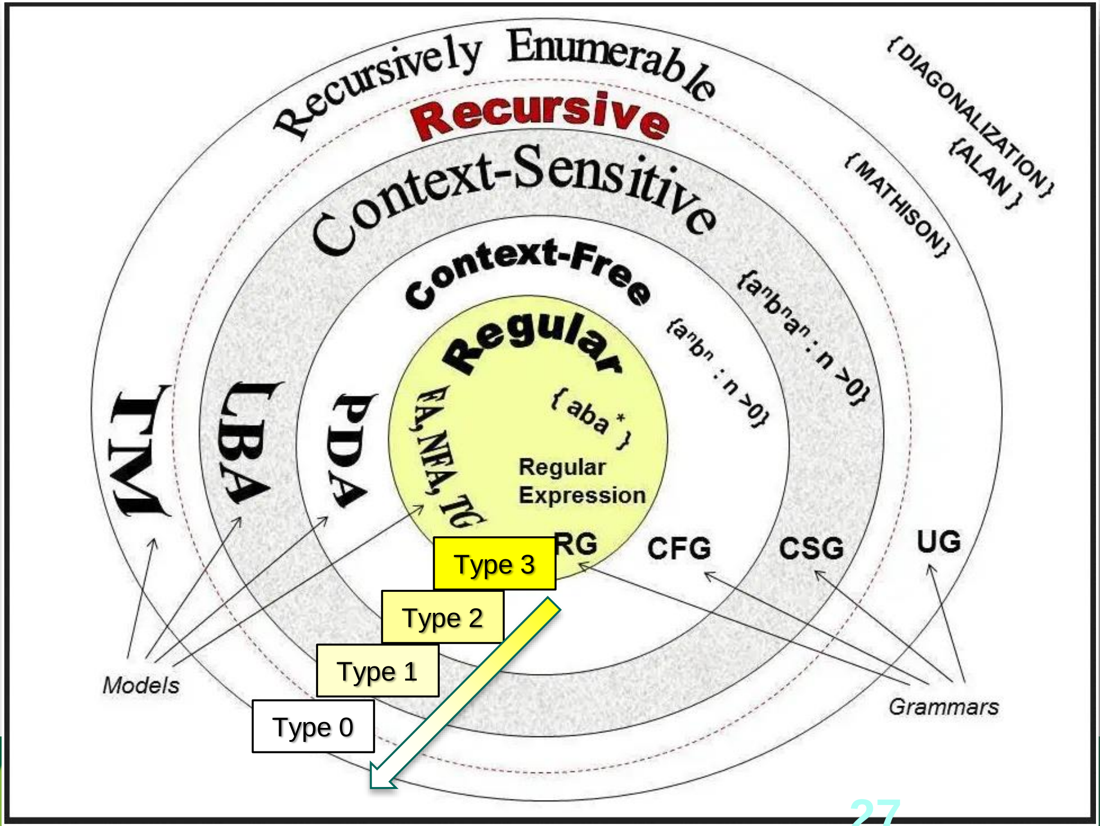


https://www.enterrasolutions.com/wp-content/uploads/2015/03/Ontology-01.png

Compilers – Art. 5

Models

General Models (take a breath…)



Think about this [1]:

- What is the best model for PL?

Source:
https://i2.wp.com/www.theoryofcomputation.co/wp-content/uploads/2018/09/Chomsky_Hierarchy.jpg

Initial Concepts

Alonzo Church Idea

The lambda calculus (also written as λ-calculus, where lambda is the name of the Greek letter λ) was created by Alonzo Church in the early 1930s to study which functions are computable.

- In addition to being a concise yet powerful model in **computability theory**, the lambda calculus is also the simplest functional programming language.
- So much so that the lambda calculus looks like a “toy” language, even though it is (provably!) as powerful as any of the programming languages being used today, such as JavaScript, Java, C++, etc.

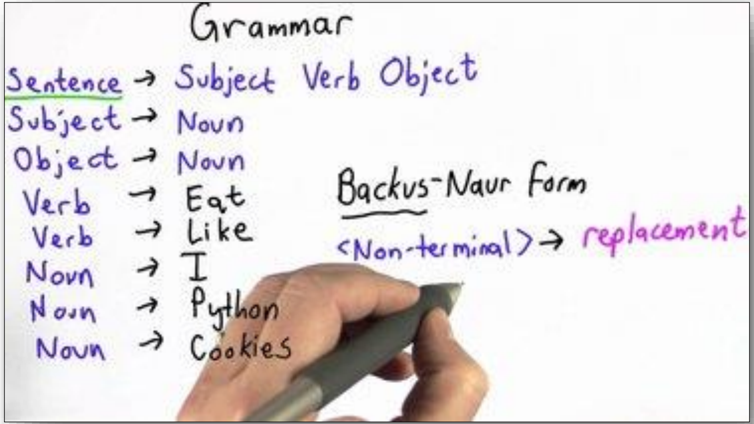
https://opendsa-server.cs.vt.edu/OpenDSA/Books/PL/html/Syntax.html

Grammars (2)

- **BNF:**

The following metalanguage is that of **Noam Chomsky** in a notation proposed by **John Backus**. Later we will use a more compact form called **Backus-Naur Form** (**BNF**).

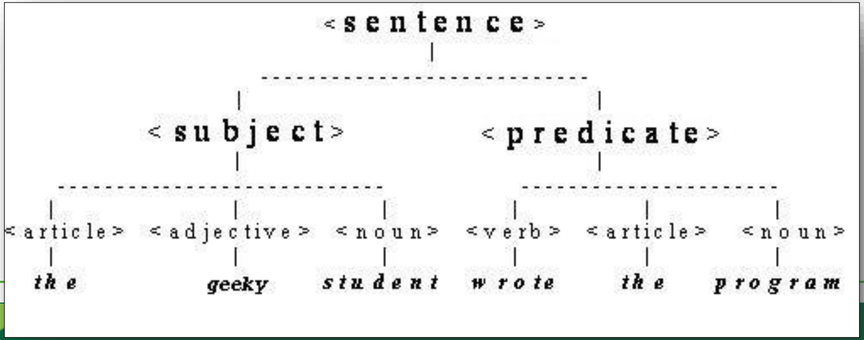
Model:



Grammars (3)

- **Example:**

```
<sentence>       ::= <subject> <predicate>
<subject>        ::= <article> <adjective> <noun>
<predicate>      ::= <verb> <direct object>
<direct object>  ::= <article> <noun>
<article>        ::= the
<adjective>      ::= geeky
<verb>           ::= wrote
<noun>           ::= student
<noun>           ::= program
```



Metalanguage:

- The **metasymbol** ::= means "*has the form of*" or "*may be composed of*".
- A single rule is called a **production** since what is on the left-hand side can produce a more detailed string on the right-hand side.
- The **metasymbols enclosed** in <> (<sentence> etc.) are called **non-terminals** since they will be replaced by applying the production.
- The symbols in bold (the, geeky, etc.) are called **terminals** since they terminate the syntax (they are the leaf nodes).

Languages (1)

- **Remember:**

1. **lexical analysis (the scanner)** - constructing the string of tokens (terminal symbols) from the string of characters
2. **syntactic analysis (the parser)** – validating (recognizing) the token string against the grammar.

- **Example:**

"The geeky student wrote the program"

token	attribute - lexeme
ARTICLE	*the*
ADJECTIVE	*geeky*
NOUN	*student*
VERB	*wrote*
ARTICLE	*the*
NOUN	*program*

ARTICLE ADJECTIVE NOUN VERB ARTICLE NOUN

Top Down Parsing

Definition:

- In this we begin with the <sentence> nonterminal, called the **start symbol**, and by successive applications (**derivations**) of the productions attempt to arrive at the sentence.
- At each stage one token is consumed.

Example:

- We begin at the start nonterminal <sentence>.
- We find using rule 1. that a sentence starts with a `<subject>` that starts with an `<article>`.
- Now that **ARTICLE** has been recognized, it is consumed, the token string is shorter and we are now attempting to recognize **ADJECTIVE.**
- ***Continue this process until the end*.**

Top Down Parsing

Note:

- Humans express themselves recursively in written, spoken and programming languages.
- Useful programming languages are also recursive and can use recursive algorithms to implement the parser (**recursive descent parser**).

- However, recursion will cause problems in the code implementation of the grammar that we will need to solve.
- ***For this, it is necessary use formal languages…***

Calc Language (1)

Informal Specification of the Calc Language

- The Calc language is a language for programming simple calculations with whole numbers.
  - The Calc program is a sequence of ***assignment statements***.
  - An assignment statement consists of an identifier representing a variable followed by an **=** symbol, followed by an ***arithmetic expression.***
  - An ***arithmetic expression*** is an infix expression constructed from ***variables***, ***integer literals***, and the ***operators*** *plus* (**+**), *minus* (**-**), *multiplication* (*****), and *division* (**/**).
- The arithmetic expression always evaluates to an integer value which is assigned to the variable on the left side of the = sign.
- If a variable is used in an expression, the variable must have a previously assigned value.
- Before termination, the program automatically prints the values of all variables introduced in the program.

Example of a Calc program:

```
a = 1 * 5
b = a / 3
c = a + b * c / d
```

Output:

```
a -> 5
b -> 1
c -> 6
```

Calc Language (2)

Lexical Grammar for the Calc Language

```
<vid>         -> <letters>
<letters>      -> <letter> | <letters><letter>
<letter>      -> a | b | … | z
<num>         -> <digits>
<digits>      -> <digit> | <digits> <digit>
<digit>       -> 0 | 1 | 2 | … | 9
<operator> -> + | - | * | / | =
```

Syntactical Grammar for the Calc Language (Calc Syntax)

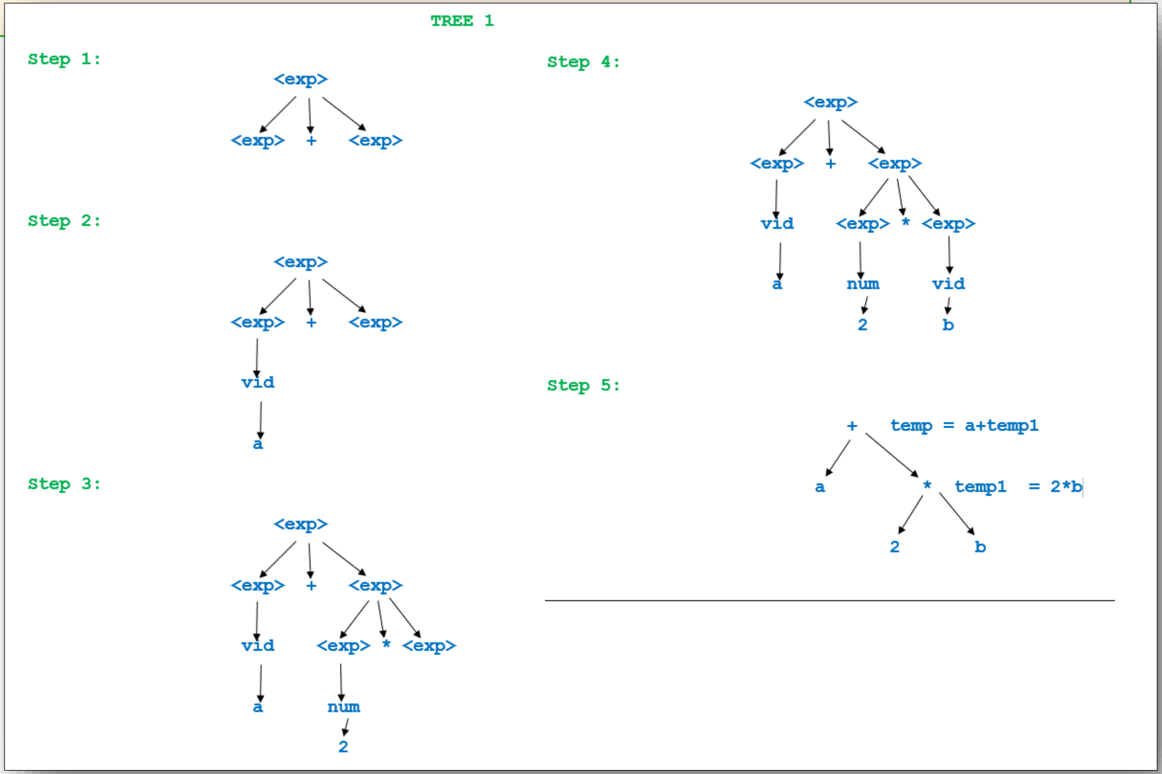
```
<program>    -> <statements>
<statements> -> <statement> |
                <statements> <statement>
<statement>  -> <assignment>
<assignment> -> vid = <expression>
<expression> -> vid
     | num
     | <expression> <operator> <expression>
```

Parse Tree (1)

- Grammar 1: Ambiguous

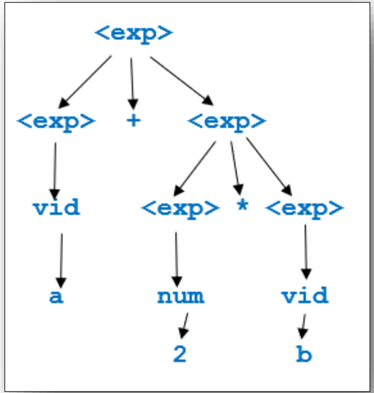
```
<exp> ->
        vid (1)
      | num (2)
      | <exp> + <exp> (3)
      | <exp> * <exp> (4)
      | <exp> - <exp> (5)
      | <exp> / <exp> (6)
```

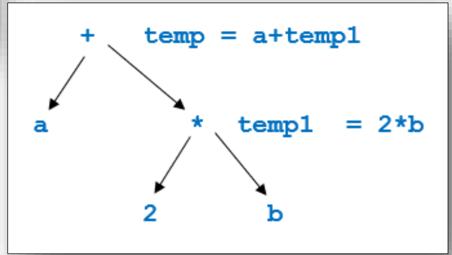
- **Example: a + 2 * b**



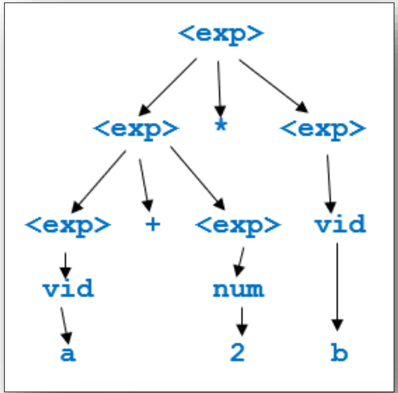
Parse Tree (2)

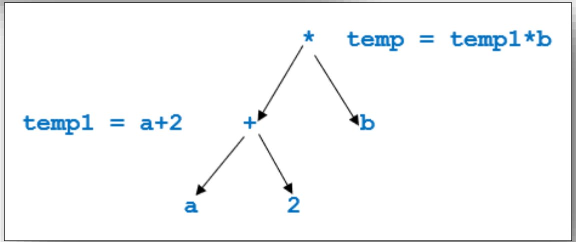
Tree1:





Tree2:





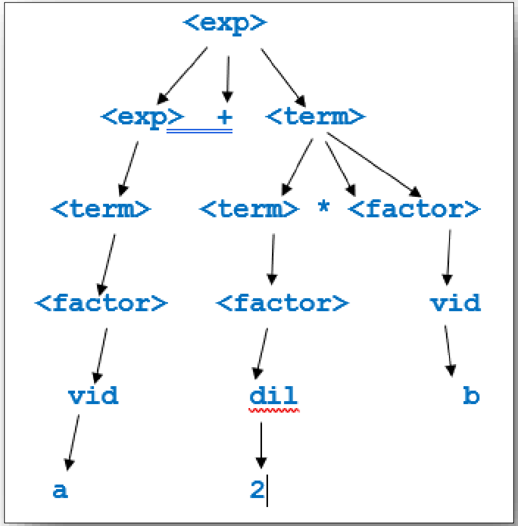
Parse Tree (3)

- Grammar 1: No-Ambiguous

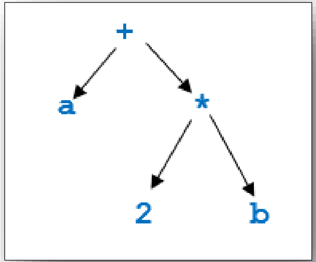
```
<exp> -> <exp> + <term>
        | <exp> - <term>
        | <term>

<term> ->  <term> * <factor>
          | <term> / <factor>
          | <factor>

<factor> -> vid | dil | (<exp>)
```

- **Example: a + 2 * b**

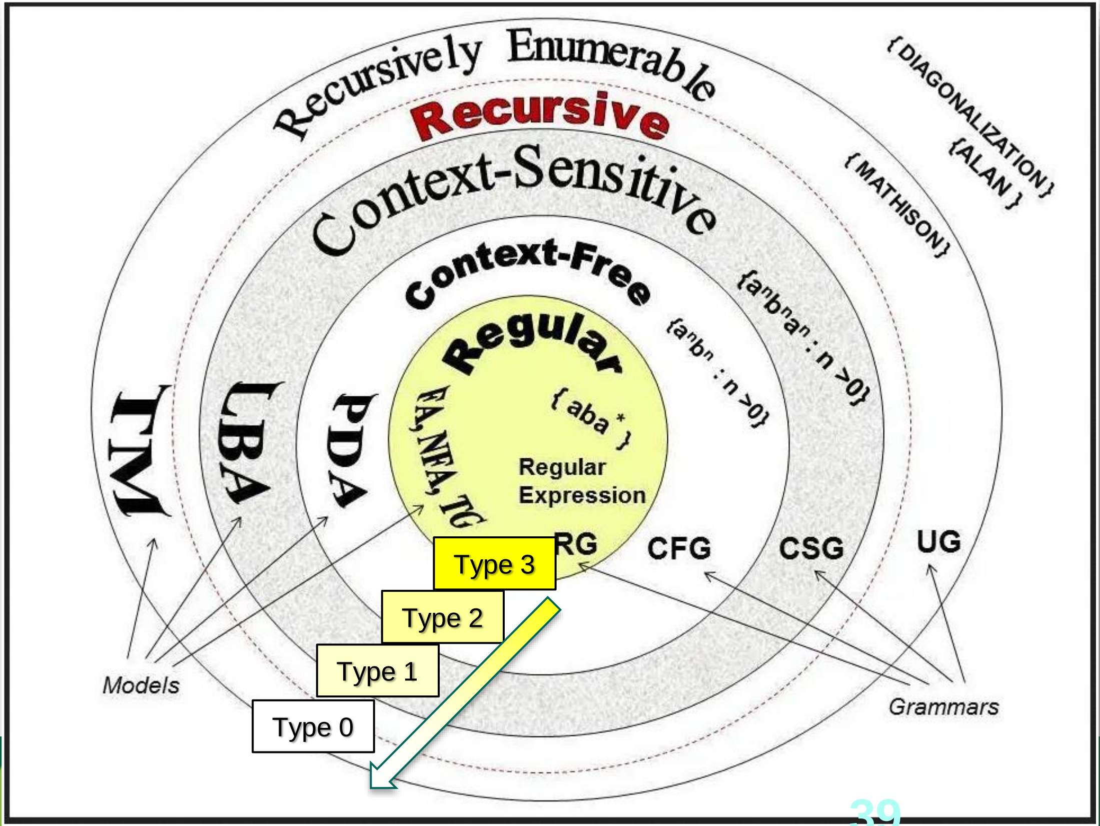
Tree:



Syntax tree:



General Models (take a breath…)



Think about this [1]:

- What is the best model for PL?

Source: https://i2.wp.com/www.theoryofcomputation.co/wp-content/uploads/2018/09/Chomsky_Hierarchy.jpg

Universal Concepts

Alphabet:

- An **alphabet** ∑ is a finite, non-empty, set of symbols. For example:
- The binary alphabet is {0, 1}
- The decimal alphabet is {0,1,2,3,4,5,6,7,8,9}
- **Note**: The metasymbols { , and } used here that are not in the alphabet.

*** IMPLEMENTATION NOTE ***

- For the scanner, the alphabet may be characters in the **ASCII** character set.
- For the parser the alphabet is the set of tokens produced by the scanner.
- Ex. Important sets: *keywords*

SOFIA Keywords: { "DO", "ELSE", "FALSE", "IF", "INPUT", "OUTPUT", "THEN", "TRUE", "WHILE"}

Understanding the Kleene Theorem

- **Main Idea:**

Theorem

The language that can be defined by any of these three methods

1. Regular Expressions (or Regular Grammar)

   or

2. Transition graph (transition or state diagrams)

   or

3. Finite Automaton (Finite State Machine)

Prof. Kleene

Source: Wikipedia

The language that can be defined by:

1. Regular Expressions (compact language);
2. Regular Grammar (syntax production rules);
3. Finite Automaton (DFA / NFA);
4. Transition graph (transition / state diagrams);
5. Lambda calculus (math definition)

Operations in Languages

Concatenation of sets

- The concatenation of two sets A and B is defined by:

AB = { xy | x in A and y in B }

which reads "*the set of strings xy such that x is in A and y is in B*".

- For example.
- If A = {a,b} and B = {c,d} then AB = { ac, ad, bc, bd }

Powers of sets

- The power of a set A: The repetition of A several times.

A^4 = { x | 4-symbol string }

which reads "*the set of strings with four symbols*".

- This is just repeated:

A^0 = { ε }, A^1 = A, A^2 = AA, A^3 = AAA, ...

- Note that A^0 = { ε } (for any set)

Operations in Languages

Union of sets

- The union of two sets A and B is defined by:

$$A \cup B = \{ x \mid x \text{ in } A \text{ or } x \text{ in } B \}$$

which reads "*the set of strings x such that x is in A or x is in B*".

- For example.
- If A = {a,b} and B = {c,d} then AUB = { a, b, c, d }

Kleene closure

- The Kleene closure of a set A is the * operator defined as the set of all strings including the empty string:

$$A^* = \bigcup_{i=0}^{\infty} A^i$$

- It is the union of all powers of A.

$A^* = A^0 + A^1 + A^2 + A^3 + \ldots$

Remember Kleene Theorem

- **TIP**: A regular expression can be used to construct a **Deterministic Finite Automaton (DFA)** which therefore can recognize strings (words) of the grammar, which is the purpose of the Scanner.
- The sets of strings defined by regular expression are termed **regular sets**.

To define the RE (as any expression notation) use operands and operations.

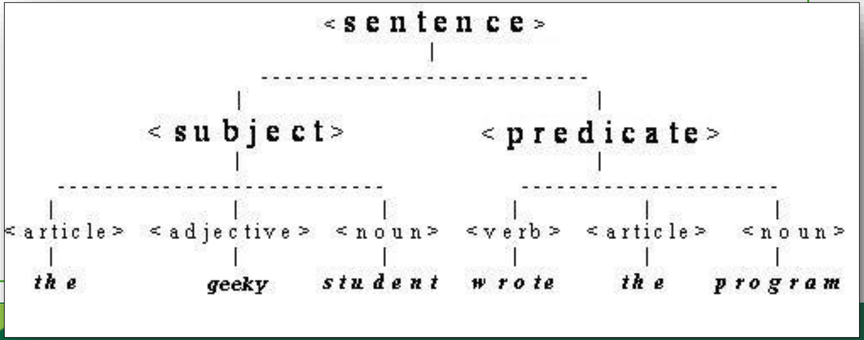
- The **operands** are alphabet symbols or strings defined by regular expressions (regular definitions).
- The standard operations are **catenation** (concatenation) , union or **alternation** (|), and **recursion** or Kleene closure (*)
- Regular expressions use the **metasymbols** |, (,), {, } , [,], * , + (and others ?, ^) to define its operations.

Grammars (1)

- **Informal Definition:**

Who Does What To Whom How Where When

"The geeky student wrote the program".



Metalanguage:

- Note the use of meta-symbols like **<subject>** enclosed in <> to denote the syntactic entities;
- These constitute a special language that describes a language.
- Being a language about a language, it is a **metalanguage**.

Remember Kleene Theorem

- **TIP**: A regular expression can be used to construct a **Deterministic Finite Automaton (DFA)** which therefore can recognize strings (words) of the grammar, which is the purpose of the Scanner.
- The sets of strings defined by regular expression are termed **regular sets**.

To define the RE (as any expression notation) use operands and operations.

- The **operands** are alphabet symbols or strings defined by regular expressions (regular definitions).
- The standard operations are **catenation** (concatenation) , union or **alternation** (|), and **recursion** or Kleene closure (*)
- Regular expressions use the **metasymbols** |, (,), {, } , [,], * , + (and others ?, ^) to define its operations.

Remember Kleene Theorem

- **Model 2:** **Grammar Formalization**

$$G = \{V_T, V_N, P, S\}$$

- Where:

1. V_T = Terminals;
2. V_N = Non-Terminals;
3. P = Production rules: $\{A \rightarrow X_1X_2\ldots X_N\}$;
4. S = Start symbol.

S
/   \
Sb   a

Example:

1. G1: {{a,b}, {S}, P = {S →Sb|a}, S}

The infinite set of words can be given by:

a, ab, abb, abbb, …

The corresponding RE1: ab*

Note: The grammar G1 and RE1 are equivalent!

Remember Kleene Theorem

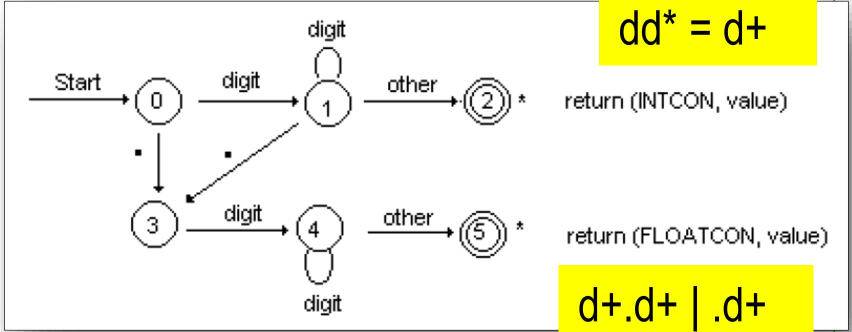
Other = ^digit && ^.
Σ=(digit,., other)

Q = set of states = {q0…q5}

Qf = {q2, q5}

- **Model 3: Automaton:**

1. Functional way to define the evolution of an acceptable string in a language.
2. Can be visual such as:



dd* = d+

d+.d+ | .d+

q1 = δ(q0, digit)

q3 = δ(q0, .), δ(q1, .)

Finite Automaton:

1. Mathematical representation of transitions that transform inputs in outputs that describes a language.

$$L(G) = A(\Sigma, Q, q0, Qf, \Delta)$$

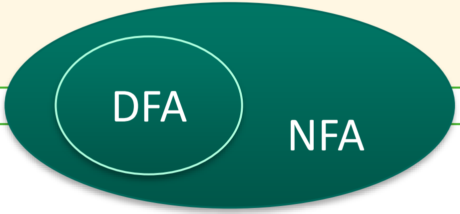
2. This notation includes the alphabet (Σ) that, starting in a state (q0) can perform words in the end (Qf), by productions (Δ) between states (Q).

Note: Empty string (ε) is also acceptable.

L(G) = Language from a Grammar (RE)

Formalization [1]

- **FA:**
- $FA = (\Sigma, Q, q0, Qf, \Delta)$
- Where:
  - Σ = Alphabet = {a0, a1, …, an}
  - Q = Set of states = {q0, q1, …, qm}
  - q0 = Initial state;
  - Qf = Final states;
  - $\Delta = P$ = Production rules: $\{q_y = \delta(q_x, a_m), \ldots, q_z = \delta(qy, an)\}$



NFA vs DFA

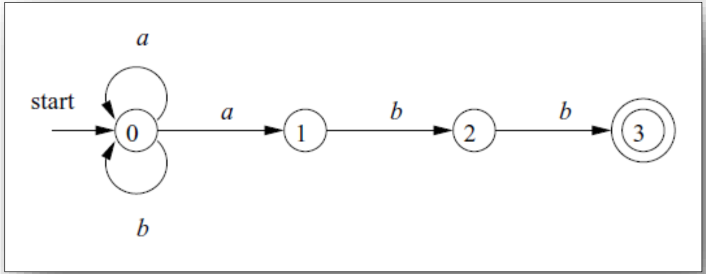
1. DFA: Deterministic, once you are in a state and read a symbol, you know exactly where to go.

2. NFA: Indeterministic because:

- It is possible to have more than one transition when read a symbol;
- Null (Epson) transitions: you can go to another state reading nothing.

But, what about NFA?

NFA:



(a|b)*abb

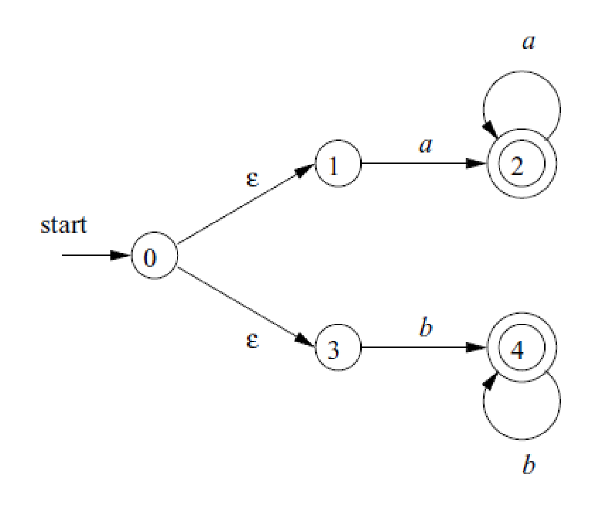
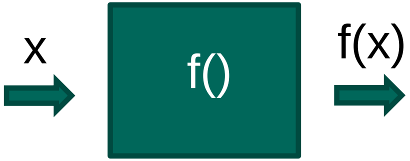


Figure 3.26: NFA accepting $\mathbf{aa}^*|\mathbf{bb}^*$

Lambda Calculus (1)

Model:



Abstraction for functions (no internal state is important).

We just have:

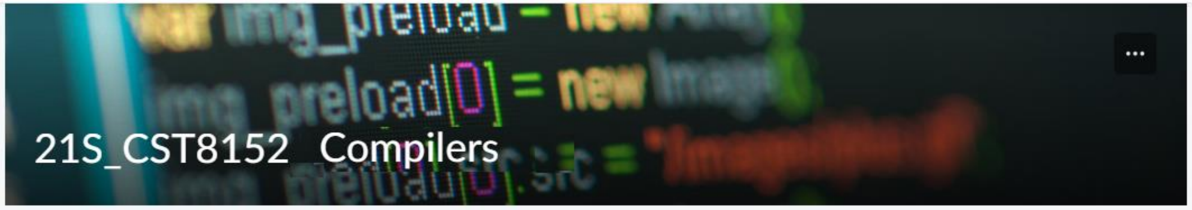
- Variables
- Functions (how to define/apply)

We do not have:

- Datatypes
- Controls

Several definitions are functions:

- Constants
- Operations
- Expressions.



Compilers – Art. 1-5

Implementations

Example (3)

DFA (analogous to NFA):

```
s = s0;
c = nextChar();
while ( c != eof ) {
        s = move(s, c);
        c = nextChar();
}
if ( s is in F ) return "yes";
else return "no";
```

Figure 3.27: Simulating a DFA

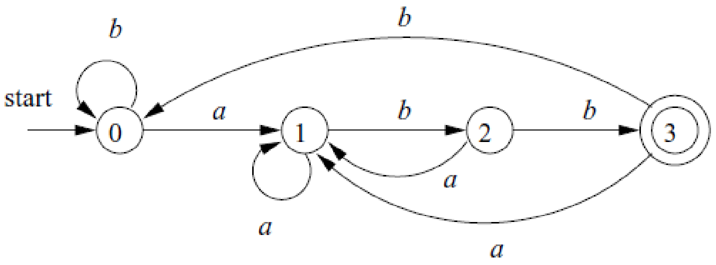
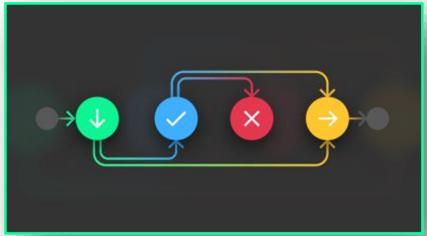


Figure 3.28: DFA accepting $(\mathbf{a}|\mathbf{b})^*\mathbf{abb}$

Implementing Transition Tables

- The NFA implementation is a **loop** in which a character is taken from the input, the next_state() function is called and the type of the returned state is tested.
- If the state is not-accepting the loop **continues**.

- If the state is accepting the loop is **terminated** and the recognized string (lexeme) is processed by the corresponding accepting function.
- Usually an array of pointers to functions is used to call an accepting function using the accepting state number as an index.



Check the Examples

Language: **Mold:**

1. You can see different elements such as comments, keywords, identifiers, methods, constants and separators.

2. Most languages must define rules to recognize these elements (ex: <id,1>, etc.)

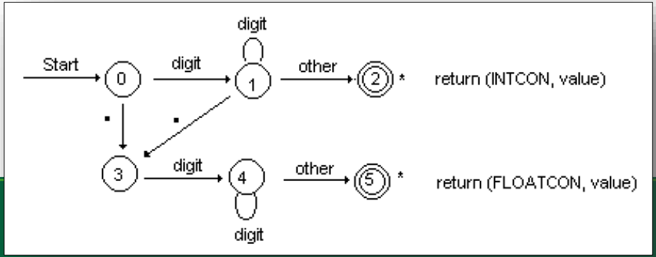
3. Different strategies can be used.

```
# Mold Example (Volume of a sphere) #
main& {
  data {
    real PI%, r%, Vol%;
  }
  code {
    PI% = 3.14;
    input&(r%);
    Vol% = 4.0 / 3.0 * PI% * (r% * r% * r%);
    print&(Vol%);
  }
}
```

Manual Implementation

- **Basic Idea:**
  - A transition diagram can be implemented manually, with states being represented by variables, and transitions being decided on by if statements.

Note: This procedure can take time and requires careful observation about states.



Basic code:

```
token next_token() {
  while (1) {
    switch(state)
    case 0:
      c = nextchar();
      if (c == ' ') {  // Skip blanks
        lexeme_start++;
        state = 0;
      } else if (isdigit(c)) state = 1;
      else if (c == '.') state = 3;
    break;
//...
```

```
//...
    case 1:
      c = nextchar();
      if (isdigit(c)) state = 1;
      else if (digit == '.') state=3;
      else state= 2;
     break;
     case 2:
       retract(1); // Backtrack in input
       store_the_lexeme();
       return(INTCON);
    // Cases 3, 4, and 5 would
    // also be implemented here...
```

Better implementation

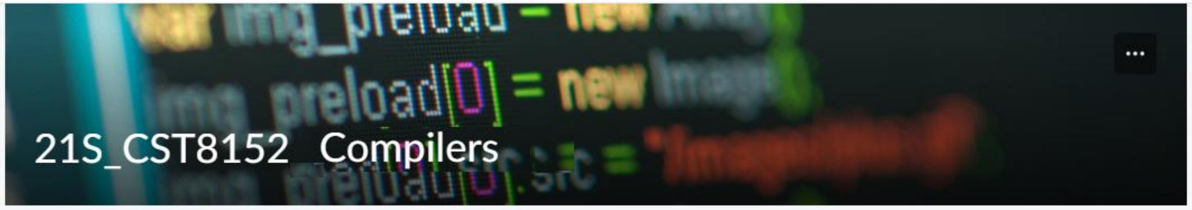
- **Using TT (Arrays)**
- A two-dimensional array can be used to implement a transition table.
- A table consists of rows representing **states** and columns representing character **classes (kinds of Token)**.

Class Value (column)	
Digit	0
.	1
Other	2

```
#define ES 6
#define IS -1
int Table[6][3] = {
/* State 0 */ { 1, 3, ES},
/* State 1 */ { 1, 3, 2 },
/* State 2 */ { IS,IS,IS},
...
/* State ES */ { IS, IS,
IS};
```

Basic code:

```
int next_state(int currentState, char ch) {
  int column = get_column(ch)
  return Table[currentState][column];'
}

// Given a character, find what class it
// belongs to, to be used for table indexing
int get_column(char ch) {
  if isdigit(ch) return(0);
  else if (ch = '.') return 1;
  else return 2;
}
```

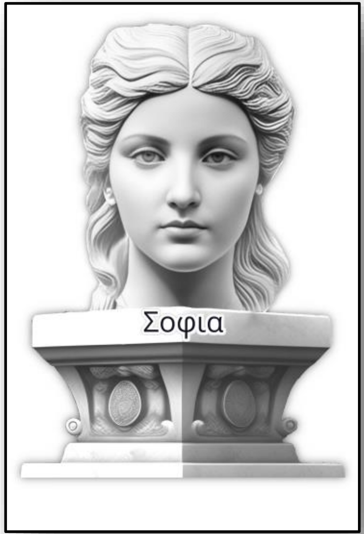


Compilers – Art. 1-5

Activities

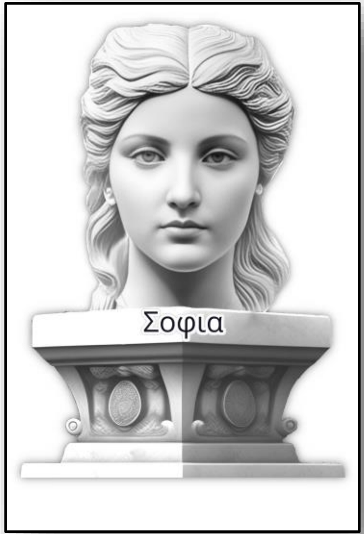
Quiz 1

- In this first Team Quiz, answer:
  - **Question 1**: **Which language** you consider the “better” one. Explain why (at least 2 reasons).
  - **Question 2**: If you are supposed to implement **THIS** language, what could be the challenges to do it (at least 2 ideas)?



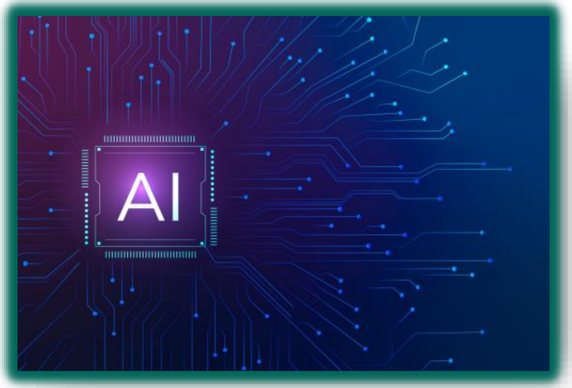
Quiz 2

- Considering the Compilation process:
  - **Question 1**: Considering the features of your preferred **IDE**, during compilation process, which one you consider the better feature? Give one example of this.
  - **Question 2**: Which **new feature** (that does not exist) you would like to include in your IDE? Why?



Quiz 3

- About bootstrapping (when a language can be built by itself)?
  - **Question 1**: If you would create a new AI language, composed by several others, what you would select?
    - For instance: for backend / frontend / data processing / etc.

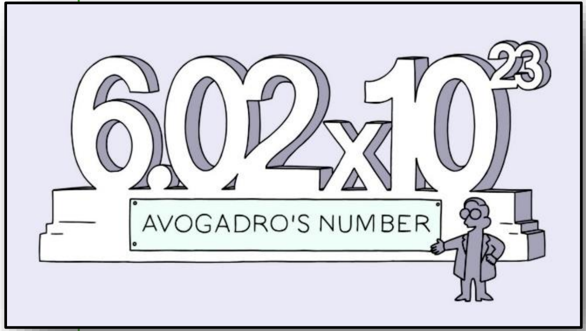


Quiz 4

- How to use lambda to define:
  - **String operators**: Define:
    - Concatenation, find(Substring).
  - **Arithmetic operators**: Define:
    - Multiplication by addition;
    - Power by multiplication.

Quiz 5

- PART 1: You need to define a **regular expression** for a **scientific notation number.**
  - **Examples:**
    - **Avogrado Number: 6.023e+23.**
    - **Electron charge: -1.602e-19.**
  - **Define:**
    - **Classes to be used**
    - **Express the RE for numbers.**
- PART2: **Define:**
  - **The AUTOMATON to be used.**



ALGONQUIN COLLEGE

Compilers – Art. All

Thank you for your attention!

ALGONQUIN COLLEGE