

# PROGRAMMING LANGUAGES FOUNDATIONS

## Computer Science III

Author: Eng. Carlos Andrés Sierra, M.Sc.  
[cavirguezs@udistrital.edu.co](mailto:cavirguezs@udistrital.edu.co)

Lecturer  
Computer Engineering  
School of Engineering  
Universidad Distrital Francisco José de Caldas

2025-I



# Outline

1 History of the Computation

2 Programming Languages

3 Finite-State Machines

4 Generative Grammars



# Outline

1 History of the Computation

2 Programming Languages

3 Finite-State Machines

4 Generative Grammars



# Babbage Machine

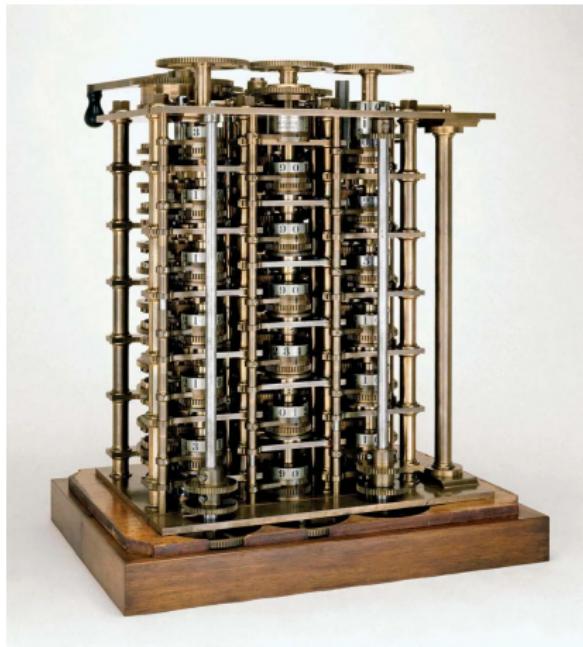


Figure: Analytical Machine



# Ada Lovelace

- Ada Lovelace (1815 — 1852) was an English mathematician and writer.
- She is known for her work on Charles Babbage's early mechanical general-purpose computer, the Analytical Engine.
- She was the first to recognize that the machine had applications beyond pure calculation, and to have published the first algorithm intended to be carried out by such a machine.



Figure: Ada Lovelace

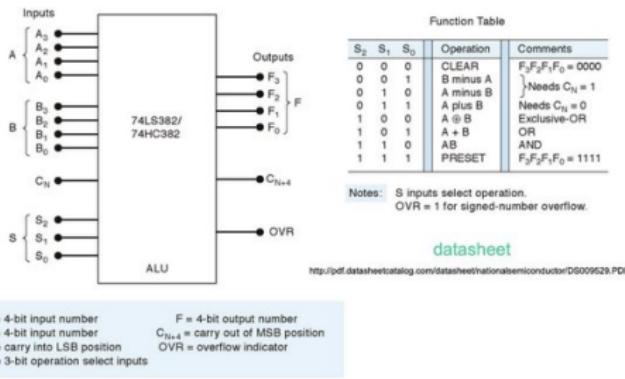


# Physical Binary Language

- **Binary** is a base-2 number system. It uses only two symbols: typically **0** (zero) and **1** (one).
- The **bit** is the basic unit of information in computing and digital communications.

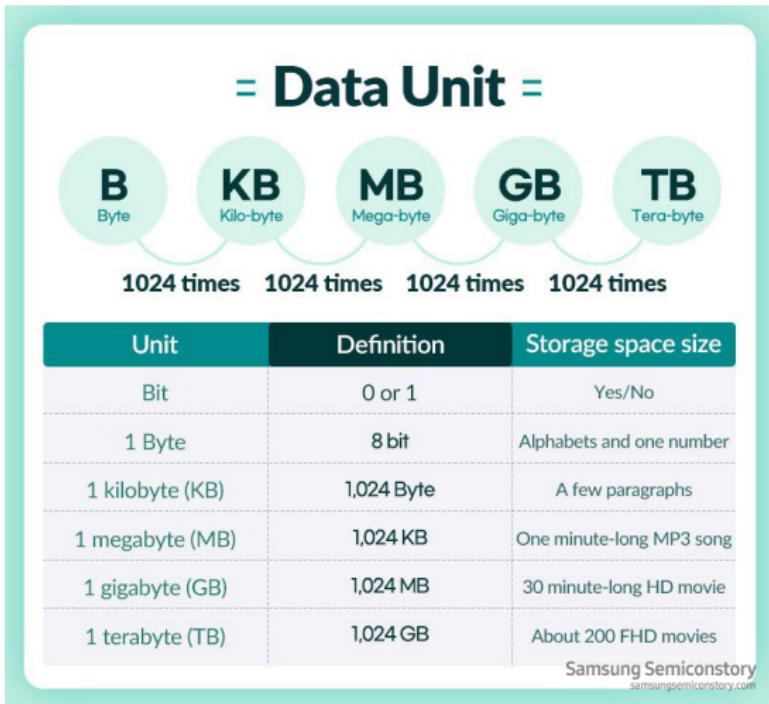
## Arithmetic Logic Unit

- Examine the functionality of this 74LS382 ALU chip



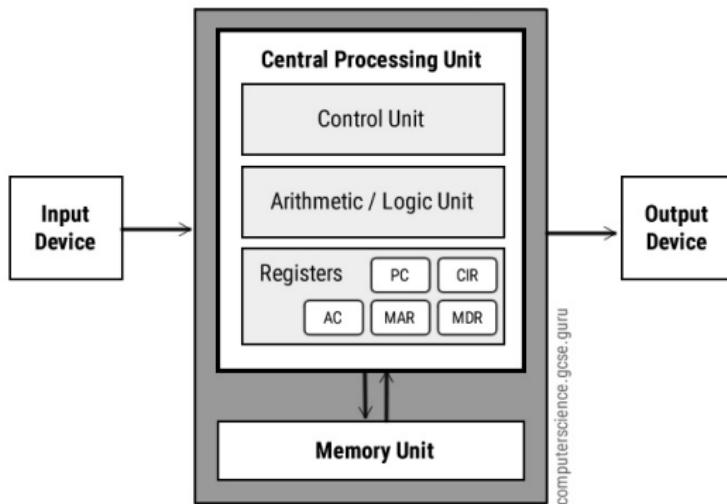
ECE331: MIPS Instructions-I 13

# Bits and Bytes



# Von Neumann Architecture

- The **Von Neumann Architecture** is a computer architecture based on the stored-program computer concept.
- The **design** is based on the concept of an instruction set.
- The **program** and **data** are both stored in the same memory unit.



computerscience gse.guru



# Bit Operations

- **Bitwise operations** are **operations** that directly **manipulate bits**.
- They are used in **low-level programming** for performing calculations, file processing, and data compression.



# Memory and Bit Storage



# Outline

1 History of the Computation

2 Programming Languages

3 Finite-State Machines

4 Generative Grammars



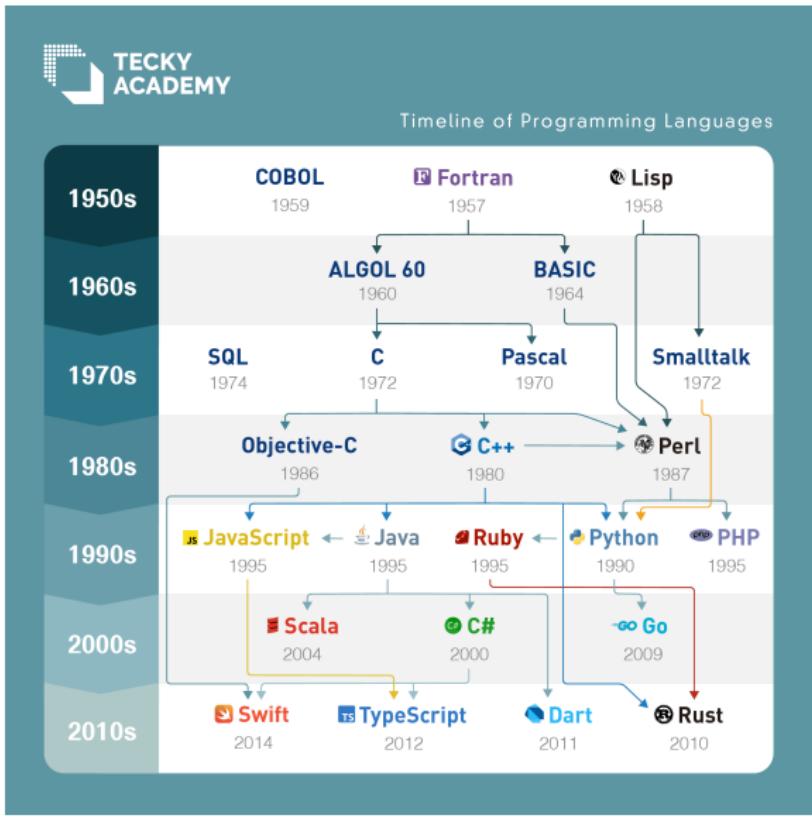
# Assembly Programming Language

```
1 ; Example of a basic conditional structure in x86  
2  
3 cmp eax, 10      ; Compare the value in eax with 10  
4 je equal_label ; Jump to equal_label if eax is equal to  
5 10  
6  
7 ; Code for not equal case  
8 jmp end_label   ; Jump to end_label to avoid executing the  
9 equal case code  
10  
11 equal_label:  
12 ; Code for equal case  
13  
14 end_label:  
15 ; Continue execution
```

Listing 1: Basic Conditional Structure in Assembly

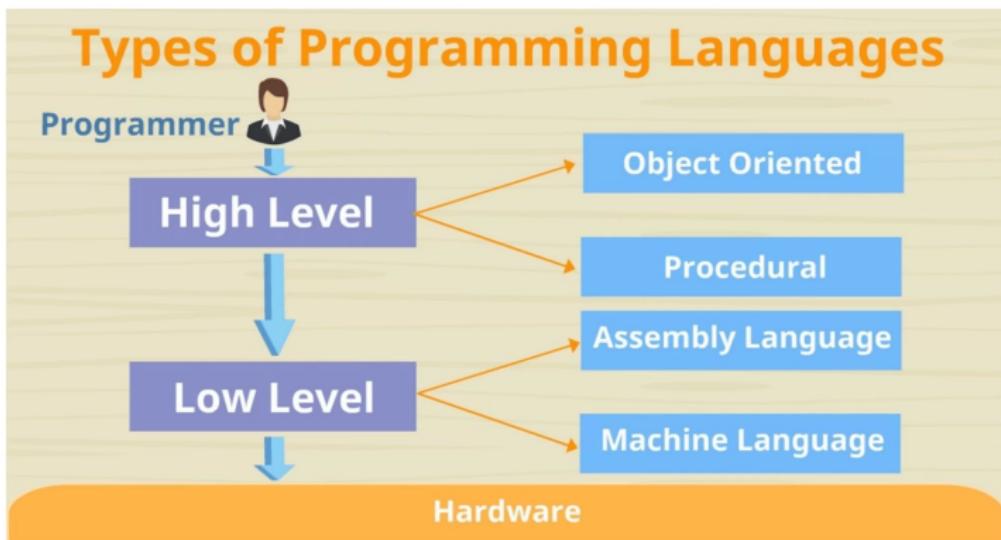


# History of Programming Languages



# High-Level Programming Languages I

## High-Level vs Low-Level



inprogrammer



# High-Level Programming Languages II

## Purpose of High-Level Languages

### Ease of Use

- Simplifying programming
- Minimizing learning curve
- Enhancing productivity
- Automated memory management
- Clear syntax
- Readability and maintainability

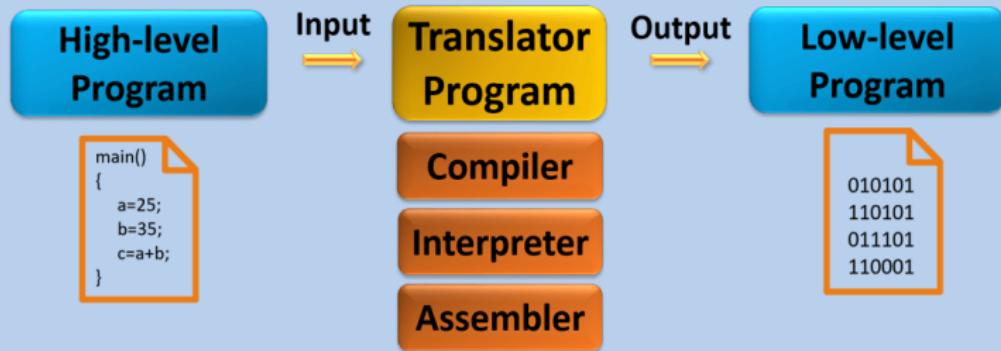


### Portability Across Systems

- Cross-platform compatibility
- Utilization of compilers and interpreters
- Seamless execution on various platforms
- Reduction of platform-specific modifications
- Enhanced flexibility across environments



# Translation Process



# Efficiency and Readability

- **Efficiency** is the ability to avoid **wasting materials, energy, efforts, money, and time** in doing something or in producing a desired result.
- **Readability** is the ease with which a **human reader** can understand the purpose, control flow, and operation of source code.



# Efficiency and Readability

- **Efficiency** is the ability to avoid wasting materials, energy, efforts, money, and time in doing something or in producing a desired result.
- **Readability** is the ease with which a human reader can understand the purpose, control flow, and operation of source code.



# Programming Languages Levels

32-bit (4-byte) ADD instruction:

100000	00101	00010	00011	000000000000
<b>opcode</b>	<b>rc</b>	<b>ra</b>	<b>rb</b>	<b>(unused)</b>

Could be something like: Reg[4] <- Reg[2] + Reg[3]

In assembler:

```
1     ADD(R2, R3, R4)
2
```

In any high-level language like C:

```
1     a = b + c;
2
```



# Interpretation

- **Interpretation** is the process of **executing** a program in a high-level language by another program.
- Interpretation is an effective **implementation strategy** when performing a computation once or when exploring.
- There is a special program called an **interpreter** that reads a high-level program and executes it.
- **Model of Interpretation:**
  - Start with some hard-to-program machine, say  $M_1$ .
  - Write a program  $P_1$  for  $M_1$  that mimics the operation of another easier machine  $M_2$ .
  - Result,  $P_1$  is an interpreter for  $M_2$ , it means, a virtual  $M_2$ .
- **Advantages:**
  - Portability.
  - Flexibility.
  - Ease of debugging.



# Interpretation

- **Interpretation** is the process of **executing** a program in a high-level language by **another program**.
- **Interpretation** is an effective **implementation strategy** when performing a computation once or when exploring.
- There is a special program called an **interpreter** that reads a high-level program and executes it.
- **Model of Interpretation:**
  - Start with some **hard-to-program** machine, say  $M_1$ .
  - Write a program  $P_1$  for  $M_1$  that mimics the operation of another easier machine  $M_2$ .
  - Result,  $P_1$  is an **interpreter** for  $M_2$ , it means, a **virtual  $M_2$** .
- **Advantages:**
  - Portability.
  - Flexibility.
  - Ease of debugging.



# Interpretation

- **Interpretation** is the process of **executing** a program in a high-level language by **another program**.
- **Interpretation** is an effective **implementation strategy** when performing a computation once or when exploring.
- There is a special program called an **interpreter** that reads a **high-level program** and executes it.
- **Model of Interpretation:**
  - Start with some **hard-to-program** machine, say  $M_1$ .
  - Write a program  $P_1$  for  $M_1$  that mimics the operation of another easier machine  $M_2$ .
  - Result,  $P_1$  is an **interpreter** for  $M_2$ , it means, a **virtual  $M_2$** .
- **Advantages:**
  - Portability.
  - Flexibility.
  - Ease of debugging.



# Interpretation

- **Interpretation** is the process of **executing** a program in a high-level language by **another program**.
- **Interpretation** is an effective **implementation strategy** when performing a computation once or when exploring.
- There is a special program called an **interpreter** that reads a high-level program and executes it.
- **Model of Interpretation:**
  - Start with some **hard-to-program** machine, say  $M_1$ .
  - Write a program  $P_1$  for  $M_1$  that mimics the operation of another easier machine  $M_2$ .
  - Result,  $P_1$  is an **interpreter** for  $M_2$ , it means, a **virtual**  $M_2$ .
- **Advantages:**
  - Portability.
  - Flexibility.
  - Ease of debugging.



# Interpretation

- **Interpretation** is the process of **executing** a program in a high-level language by **another program**.
- **Interpretation** is an effective **implementation strategy** when performing a computation once or when exploring.
- There is a special program called an **interpreter** that reads a high-level program and executes it.
- **Model of Interpretation:**
  - Start with some **hard-to-program** machine, say  $M_1$ .
  - Write a program  $P_1$  for  $M_1$  that mimics the operation of another easier machine  $M_2$ .
  - Result,  $P_1$  is an **interpreter** for  $M_2$ , it means, a **virtual**  $M_2$ .
- **Advantages:**
  - Portability.
  - Flexibility.
  - Ease of debugging.



# Compilation

- **Compilation** is the process of **translating** a program in a **high-level language** into a **low-level language**.
- **Compilation** is an effective **implementation strategy** when performing a computation many times.
- There is a special program called a **compiler** that reads a **high-level program** and translates it.
- **Model of Compilation:**
  - Start with some **hard-to-program** machine, say  $M_1$ .
  - Write a program  $P_2$  for  $M_1$  that translates a program in a **high-level language** into a program in a **low-level language**.
  - Result,  $P_2$  is a **compiler** for  $M_1$ .
- **Advantages:**
  - **Fast Execution.**
  - **Efficiency.**
  - **Portability.**



# Compilation

- **Compilation** is the process of **translating** a program in a **high-level language** into a **low-level language**.
- **Compilation** is an effective **implementation strategy** when performing a computation many times.
- There is a special program called a **compiler** that reads a **high-level program** and translates it.
- **Model of Compilation:**
  - Start with some **hard-to-program** machine, say  $M_1$ .
  - Write a program  $P_2$  for  $M_1$  that translates a program in a **high-level language** into a program in a **low-level language**.
  - Result,  $P_2$  is a **compiler** for  $M_1$ .
- **Advantages:**
  - Fast Execution.
  - Efficiency.
  - Portability.



# Compilation

- **Compilation** is the process of **translating** a program in a **high-level language** into a **low-level language**.
- **Compilation** is an effective **implementation strategy** when performing a computation many times.
- There is a special program called a **compiler** that reads a **high-level program** and translates it.
- **Model of Compilation:**
  - Start with some **hard-to-program** machine, say  $M_1$ .
  - Write a program  $P_2$  for  $M_1$  that translates a program in a **high-level language** into a program in a **low-level language**.
  - Result,  $P_2$  is a **compiler** for  $M_1$ .
- **Advantages:**
  - **Fast Execution.**
  - **Efficiency.**
  - **Portability.**



# Compilation

- **Compilation** is the process of **translating** a program in a **high-level language** into a **low-level language**.
- **Compilation** is an effective **implementation strategy** when performing a computation many times.
- There is a special program called a **compiler** that reads a **high-level program** and translates it.
- **Model of Compilation:**
  - Start with some **hard-to-program** machine, say  $M_1$ .
  - Write a program  $P_2$  for  $M_1$  that translates a program in a **high-level language** into a program in a **low-level language**.
  - Result,  $P_2$  is a **compiler** for  $M_1$ .
- **Advantages:**
  - Fast Execution.
  - Efficiency.
  - Portability.



# Compilation

- **Compilation** is the process of **translating** a program in a **high-level language** into a **low-level language**.
- **Compilation** is an effective **implementation strategy** when performing a computation many times.
- There is a special program called a **compiler** that reads a **high-level program** and translates it.
- **Model of Compilation:**
  - Start with some **hard-to-program** machine, say  $M_1$ .
  - Write a program  $P_2$  for  $M_1$  that translates a program in a **high-level language** into a program in a **low-level language**.
  - Result,  $P_2$  is a **compiler** for  $M_1$ .
- **Advantages:**
  - **Fast Execution.**
  - **Efficiency.**
  - **Portability.**



# Interpretation Vs. Compilation

Characteristics differences:

	<b>Compilation</b>	<b>Interpretation</b>
How does it treat input $x + 2$ ?	Generate a program that computes $x + 2$	Computes $x + 2$
When it happens?	Before Execution	During Execution
What it complicates/slow?	Program Development	Program Execution
Decisions made at	Compile Time	Run Time



# Outline

1 History of the Computation

2 Programming Languages

3 Finite-State Machines

4 Generative Grammars



# Finite-State Machines

- A **finite-state machine** (FSM) is a **mathematical model** of computation.
- It is an **abstract machine** that can be in **exactly one of a finite number of states** at any given time.
- The FSM can change from one state to another in response to some **external inputs**.
- The FSM is defined by a **list of states**, a **list of inputs**, a **list of transitions**, and a **list of outputs**.



# Finite-State Machines

- A **finite-state machine** (FSM) is a **mathematical model** of computation.
- It is an **abstract machine** that can be in **exactly one of a finite number of states** at any given time.
- The FSM can change from one state to another in response to some **external inputs**.
- The FSM is defined by a **list of states**, a **list of inputs**, a **list of transitions**, and a **list of outputs**.



# Finite-State Machines

- A **finite-state machine** (FSM) is a **mathematical model** of computation.
- It is an **abstract machine** that can be in **exactly one of a finite number of states** at any given time.
- The FSM can change from one state to another in response to some **external inputs**.
- The FSM is defined by a **list of states**, a **list of inputs**, a **list of transitions**, and a **list of outputs**.



# Finite-State Machine Drawing



# Alonzo Church

- **Alonzo Church** (1903 — 1995) was an American mathematician and logician.
- He is best known for the **Lambda Calculus**, which he developed in the 1930s.
- The **Lambda Calculus** is a formal system in mathematical logic for expressing computation based on function abstraction and application.

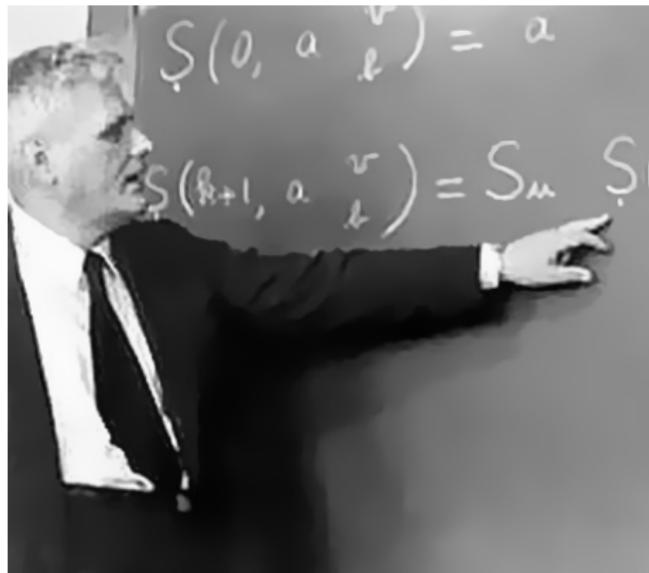


Figure: Alonzo Church



# Regular Expressions

A **regular expression** is a sequence of characters that define a search pattern.

- **Regular expressions** are used in search engines, search and replace dialogs of word processors, and in text processing utilities.
- A **regular expression** is a pattern that is used to match character combinations in strings.
- The **pattern** describes one or more **strings** to match.



# Regular Expressions

A **regular expression** is a sequence of characters that define a search pattern.

- **Regular expressions** are used in search engines, search and replace dialogs of word processors, and in text processing utilities.
- A **regular expression** is a **pattern** that is used to **match character combinations** in **strings**.
- The **pattern** describes one or more **strings** to match.



# Finite Automata: Concatenation

The **concatenation** of two regular expressions  $R_1$  and  $R_2$  is a regular expression that matches the **concatenation** of strings that are matched by  $R_1$  followed by  $R_2$ .



# Finite Automata: Union

The **union** of two regular expressions  $R_1$  and  $R_2$  is a regular expression that matches the **union** of strings that are matched by  $R_1$  **or**  $R_2$ .



# Finite Automata: Kleene's Star

The **Kleene's star** of a regular expression  $R$  is a regular expression that matches the **concatenation** of zero or more strings that are matched by  $R$ .



# Finite Automata: $\lambda$ -transition

The  **$\lambda$ -transition** is a [transition](#) that can be taken without consuming any input.



# Strings Processing



# Alan Turing

- **Alan Turing** (1912 — 1954) was an English mathematician, computer scientist, logician, cryptanalyst, philosopher, and theoretical biologist.
- He is widely considered to be the **father of theoretical computer science** and **artificial intelligence**.
- He was highly influential in the development of **theoretical computer science**, providing a formalization of the concepts of **algorithm** and **computation** with the **Turing machine**.



Figure: Alan Turing



# Turing Machine



# Universal Turing Machine



# Regular Expression & Finite Automata: *Example i*

Be  $L$  a language over the alphabet  $\Sigma = \{0, 1\}$ , such that  $L$  is the set of all strings that contain an even number of 1s. The regular expression for  $L$  is:



# Regular Expression & Finite Automata: *Example ii*

Be  $L$  a language over the alphabet  $\Sigma = \{0, 1\}$ , such that  $L$  is the set of all strings that contain the substring 01. The regular expression for  $L$  is:



## Regular Expression & Finite Automata: *Example iii*

Be  $L$  a language over the alphabet  $\Sigma = \{a, c, c\}$ , such that  $L$  is the set of all strings that contain the substring  $acc$ . The regular expression for  $L$  is:



# Regular Expression & Finite Automata: *Example iv*

Be  $L$  a language over the alphabet  $\Sigma = \{a, b, c\}$ , such that  $L$  is the set of all strings that start with the substring  $abc$ . The regular expression for  $L$  is:



# Regular Expression & Finite Automata: *Example v*

Be  $L$  a language over the alphabet  $\Sigma = \{a, b, c\}$ , such that  $L$  is the set of all strings that end with the substring  $abc$ . The regular expression for  $L$  is:



# Regular Expression & Finite Automata: *Example vi*

Be  $L$  a language over the alphabet  $\Sigma = \{a, b, c\}$ , such that  $L$  is the set of all strings that start with the substring  $ab$ , contain just two  $c$ 's and end with the substring  $ba$ . The regular expression for  $L$  is:



## Regular Expression & Finite Automata: *Example vii*

Be  $L$  a language over the alphabet  $\Sigma = \{a, b, c\}$ , such that  $L$  is the set of all strings that start with any number of  $a$ 's (could be 0), followed by any number of  $b$ 's, and end with any number of  $c$ 's. The regular expression for  $L$  is:



## Regular Expression & Finite Automata: *Example viii*

Be  $L$  a language over the alphabet  $\Sigma = \{a, b, c\}$ , such that  $L$  is the set of all strings that contain the substring  $abc$  or  $bac$ . The regular expression for  $L$  is:



# Regular Expression & Finite Automata: *Example ix*

Be  $L$  a language over the alphabet  $\Sigma = \{0, 1\}$ , such that  $L$  is the set of all strings that contain an odd number of 1s. The regular expression for  $L$  is:



# Regular Expression & Finite Automata: *Example x*

Be  $L$  a language over the alphabet  $\Sigma = \{0, 1\}$ , such that  $L$  is the set of all strings that contain an even number of 0s or an odd number of 1s. The regular expression for  $L$  is:



# Outline

1 History of the Computation

2 Programming Languages

3 Finite-State Machines

4 Generative Grammars



# Noam Chomsky

- Noam Chomsky (1928 — ) is an American linguist, philosopher, cognitive scientist, historian, social critic, and political activist.
- He is considered the **father of modern linguistics**.
- He introduced the **Chomsky hierarchy**, a classification of formal languages.



Figure: Noam Chomsky



# Natural Processing Language

- **Natural Language Processing** (NLP) is a subfield of linguistics, computer science, information engineering, and artificial intelligence concerned with the interactions between computers and human languages.
- NLP is used to apply algorithms to text and speech.
- NLP is used to understand the meaning of text and speech.
- NLP is used to generate human language text.



# Natural Processing Language

- **Natural Language Processing** (NLP) is a subfield of linguistics, computer science, information engineering, and artificial intelligence concerned with the interactions between computers and human languages.
- **NLP** is used to apply algorithms to text and speech.
- **NLP** is used to understand the meaning of text and speech.
- **NLP** is used to generate human language text.



# Natural Processing Language

- **Natural Language Processing** (NLP) is a subfield of linguistics, computer science, information engineering, and artificial intelligence concerned with the interactions between computers and human languages.
- **NLP** is used to apply algorithms to text and speech.
- **NLP** is used to understand the meaning of text and speech.
- **NLP** is used to generate human language text.



# Formal Languages

- A **formal language** is a set of **strings** of **symbols**.
- The **alphabet** of a formal language is the set of **symbols** that can be used to form **strings**.
- A **formal grammar** is a set of **rules** for generating strings in a **formal language**.
- A **generative grammar** is a formal system for describing the structure of strings in a formal language.



# Formal Languages

- A **formal language** is a set of **strings** of **symbols**.
- The **alphabet** of a formal language is the set of **symbols** that can be used to form **strings**.
- A **formal grammar** is a set of **rules** for **generating** strings in a **formal language**.
- A **generative grammar** is a **formal system** for **describing** the **structure** of strings in a **formal language**.



# Grammars Foundations

- A **grammar** is a set of **rules** for generating strings in a **formal language**.
- A **grammar** is a **formal system** for **describing** the **structure** of **strings** in a **formal language**.
- A **generative grammar** is a set of rules for generating strings in a **formal language**.



# Grammars Foundations

- A **grammar** is a set of **rules** for generating strings in a **formal language**.
- A **grammar** is a **formal system** for **describing** the **structure** of **strings** in a **formal language**.
- A **generative grammar** is a **set of rules** for generating strings in a **formal language**.



# Chomsky Hierarchy

- The **Chomsky hierarchy** is a classification of formal languages.
- The **Chomsky hierarchy** is named after the linguist and cognitive scientist **Noam Chomsky**.
- The **Chomsky hierarchy** consists of four types of formal grammars:
  - Type 0: Unrestricted grammars.
  - Type 1: Context-sensitive grammars.
  - Type 2: Context-free grammars.
  - Type 3: Regular grammars.



# Chomsky Hierarchy

- The **Chomsky hierarchy** is a classification of formal languages.
- The **Chomsky hierarchy** is named after the linguist and cognitive scientist **Noam Chomsky**.
- The **Chomsky hierarchy** consists of four types of formal grammars:
  - Type 0: Unrestricted grammars.
  - Type 1: Context-sensitive grammars.
  - Type 2: Context-free grammars.
  - Type 3: Regular grammars.



# Context-Free Grammars

- A **context-free grammar** is a **formal grammar** and **generative grammar** in which every **production rule** is of the form:

$$A \rightarrow \alpha \quad (1)$$

- Where  $A$  is a **nonterminal symbol** and  $\alpha$  is a string of **terminals and nonterminals**.
- A **context-free grammar** is a generative grammar that can generate a **context-free language**.



# Context-Free Grammars

- A **context-free grammar** is a **formal grammar** and **generative grammar** in which every **production rule** is of the form:

$$A \rightarrow \alpha \quad (1)$$

- Where  $A$  is a **nonterminal symbol** and  $\alpha$  is a string of **terminals** and **nonterminals**.
- A **context-free grammar** is a generative grammar that can generate a context-free language.



# Context-Free Grammars

- A **context-free grammar** is a **formal grammar** and **generative grammar** in which every **production rule** is of the form:

$$A \rightarrow \alpha \quad (1)$$

- Where  $A$  is a **nonterminal symbol** and  $\alpha$  is a string of **terminals** and **nonterminals**.
- A **context-free grammar** is a **generative grammar** that can generate a **context-free language**.



# Derivation Trees

A **derivation tree** is a **tree** that represents the **sequence of production rules** used to **generate** a **string** in a **formal language**.



# Equivalence between Grammars and Finite Automatas

A **context-free grammar** can be **converted** into a **finite automaton** and a **finite automaton** can be **converted** into a **context-free grammar**.



# Equivalence Between Grammars and Regular Expressions

A **regular expression** can be converted into a **finite automaton** and a **finite automaton** can be converted into a **regular expression**.



# Free-Context Grammar: *Example i*

Be  $L$  a language over the alphabet  $\Sigma = \{a, b\}$ , such that  $L$  is the set of all strings that contain the substring  $ab$ . The context-free grammar for  $L$  is:



## Free-Context Grammar: *Example ii*

Be  $L$  a language over the alphabet  $\Sigma = \{a, b\}$ , such that  $L$  is the set of all strings that contain the substring  $ab$  or  $ba$ . The context-free grammar for  $L$  is:



## Free-Context Grammar: *Example iii*

Be  $L$  a language over the alphabet  $\Sigma = \{0, 1\}$ , such that  $L$  is the set of all strings that contain an even number of 0s. The context-free grammar for  $L$  is:



## Free-Context Grammar: *Example iv*

Be  $L$  a language over the alphabet  $\Sigma = \{0, 1\}$ , such that  $L = 0^i 1^j 0^i$ , where  $i, j \geq 0$ . The context-free grammar for  $L$  is:



# Free-Context Grammar: *Example v*

Be  $L$  a language over the alphabet  $\Sigma = \{a, b, c\}$ , such that  $L = a^i b^j c^{i+j}$  where  $i, j \geq 1$ . The context-free grammar for  $L$  is:



# Outline

1 History of the Computation

2 Programming Languages

3 Finite-State Machines

4 Generative Grammars



# Thanks!

## Questions?



Repo: <https://github.com/EngAndres/ud-public/tree/main/courses/computer-science-iii>

