

PROGRAMMING LANGUAGES FOUNDATIONS

Computer Science III

Author: Eng. Carlos Andrés Sierra, M.Sc.
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

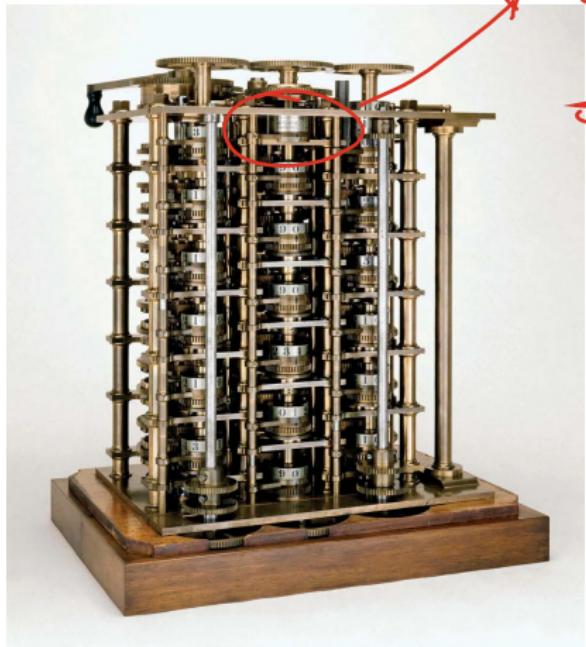


Figure: Analytical Machine

$$\begin{array}{c}
 8+5+7 \\
 \textcircled{L}+\textcircled{F}+\textcircled{3} \quad \textcircled{4}+\textcircled{5} \quad \textcircled{4}+\textcircled{7} \\
 \textcircled{3}+\textcircled{7} \quad 20
 \end{array}$$

- Charles Babbage (1791 — 1871) was an English mathematician, philosopher, inventor, and mechanical engineer.
- He originated the concept of a digital programmable computer.
- Considered the “father of the computer”. He creates the Analytical Engine.
- The **Analytical Engine** was a general-purpose mechanical computer.

fortune



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.



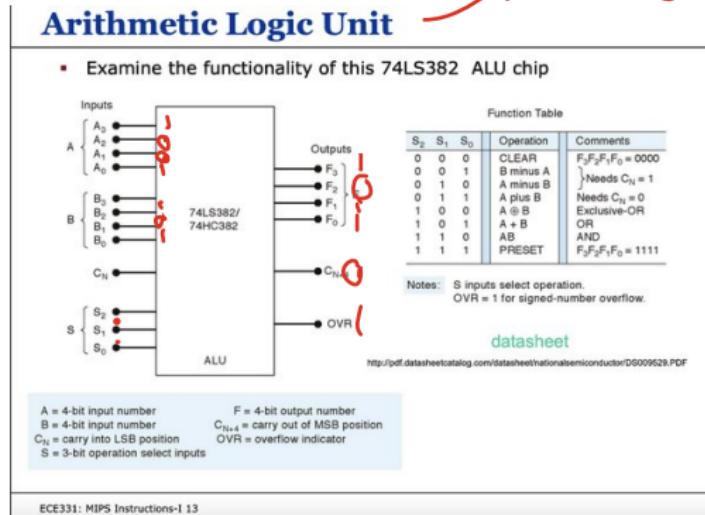
Figure: Ada Lovelace



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.

CLW



Bits and Bytes

$$2^{10} = 1024$$

= Data Unit =

Unit	Definition	Storage space size
Bit	0 or 1	Yes/No
1 Byte	8 bit	Alphabets and one number
1 kilobyte (KB)	1,024 Byte	A few paragraphs
1 megabyte (MB)	1,024 KB	One minute-long MP3 song
1 gigabyte (GB)	1,024 MB	30 minute-long HD movie
1 terabyte (TB)	1,024 GB	About 200 FHD movies

Samsung Semiconstorty
samsungsemiconductor.com

Peto

$$2^8 = 256$$

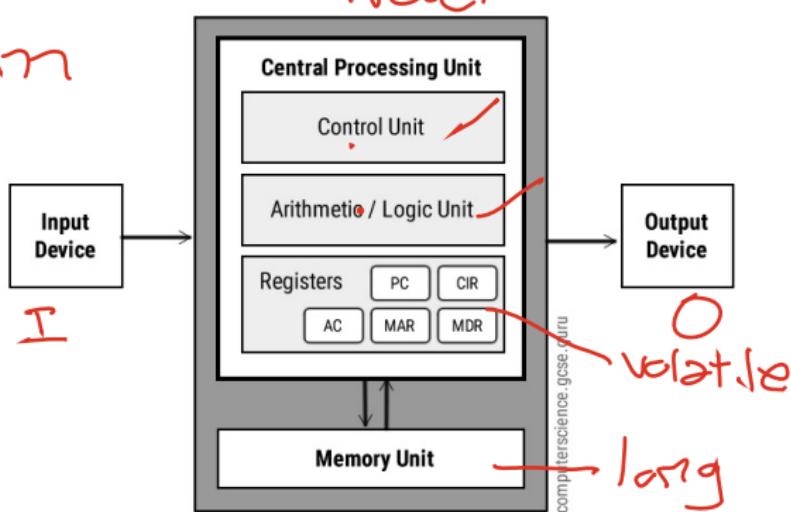
ASCII
0-255



Von Neumann Architecture

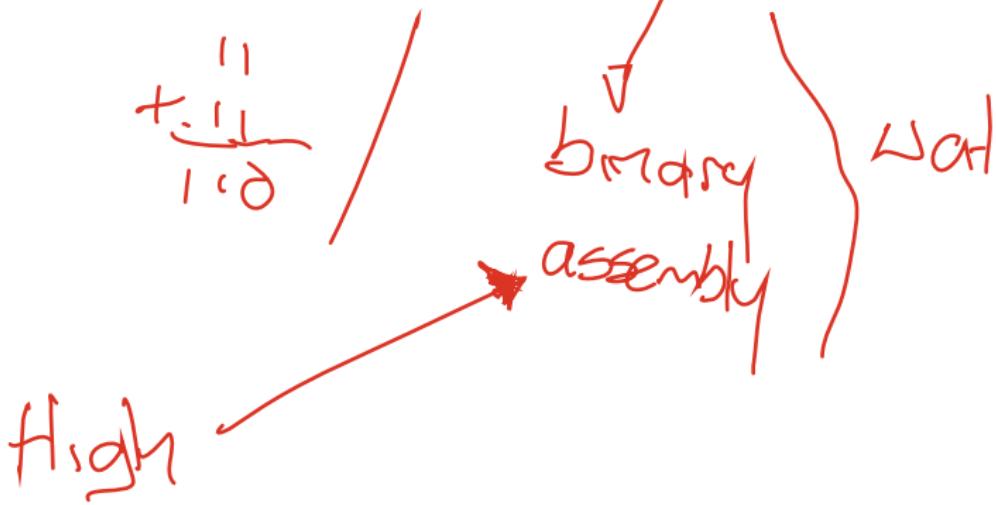
- The **Von Neumann Architecture** is a computer architecture based on the stored-program computer concept.
 - The design is based on the concept of an instruction set.
 - The program and data are both stored in the same memory unit.

System

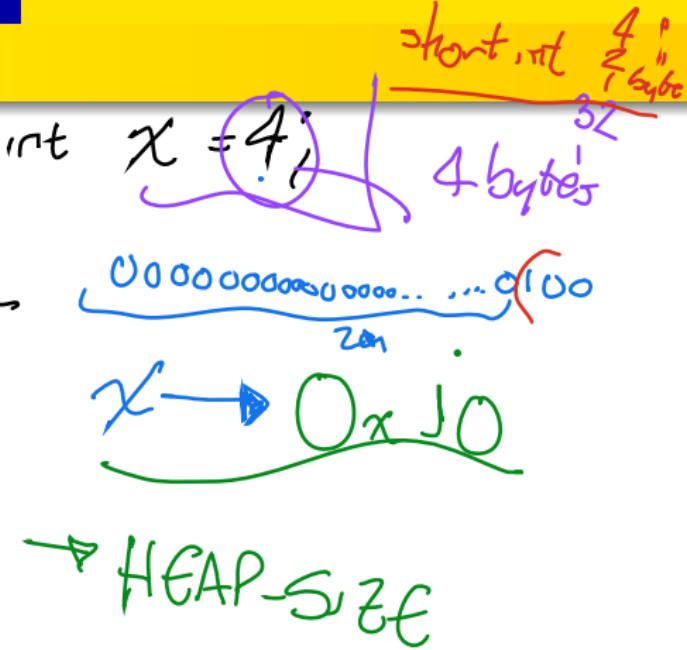
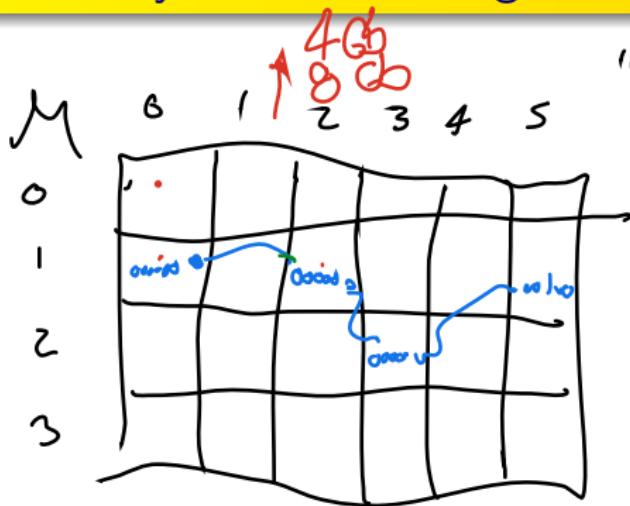


Bit Operations

- **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



Garbage Collector \rightarrow scheduled



Outline

1 History of the Computation

2 Programming Languages

3 Finite-State Machines

4 Generative Grammars



Assembly Programming Language

```
if eax==10){  
}  
  
—  
structure in x86 IF(cond){  
    /*  
    */  
    /*  
    */  
} else {  
}
```

```
1 ; Example of a basic conditional structure in x86
2 assembly
3
4
5
6
7
8
9
10
11
12
13
```

Annotations:

- Line 1: A red bracket covers the first two lines, with the text "if/cond" written above it.
- Line 2: The word "for" is circled in green.
- Line 3: The instruction "cmp eax, 10" is circled in green, with the text "Compare the value in eax with 10" written below it.
- Line 4: The instruction "je equal_label" is circled in green, with the text "Jump to equal_label if eax is equal to 10." written below it.
- Line 5: A red dashed circle surrounds the entire section from line 5 to line 7.
- Line 6: The text "Code for not equal case" is written above line 6.
- Line 7: The instruction "jmp end_label" is circled in green, with the text "Jump to end_label to avoid executing the equal case code" written below it.
- Line 9: The label "equal_label:" is circled in green.
- Line 10: The text "Code for equal case" is written above line 10.
- Line 12: The label "end_label:" is circled in green.
- Line 13: The text "Continue execution" is written above line 13.

Handwritten notes:

- Line 1: "if/cond" is written above the first two lines.
- Line 2: "else {" is written to the right of the "for" annotation.
- Line 3: "}" is written to the right of the "else" annotation.
- Line 6: "+ ultra speed" is written above the "Code for not equal case" annotation.
- Line 6: "- ugly" is written below the "Code for not equal case" annotation.
- Line 10: "Processor unit" is written above "Machine & b..".

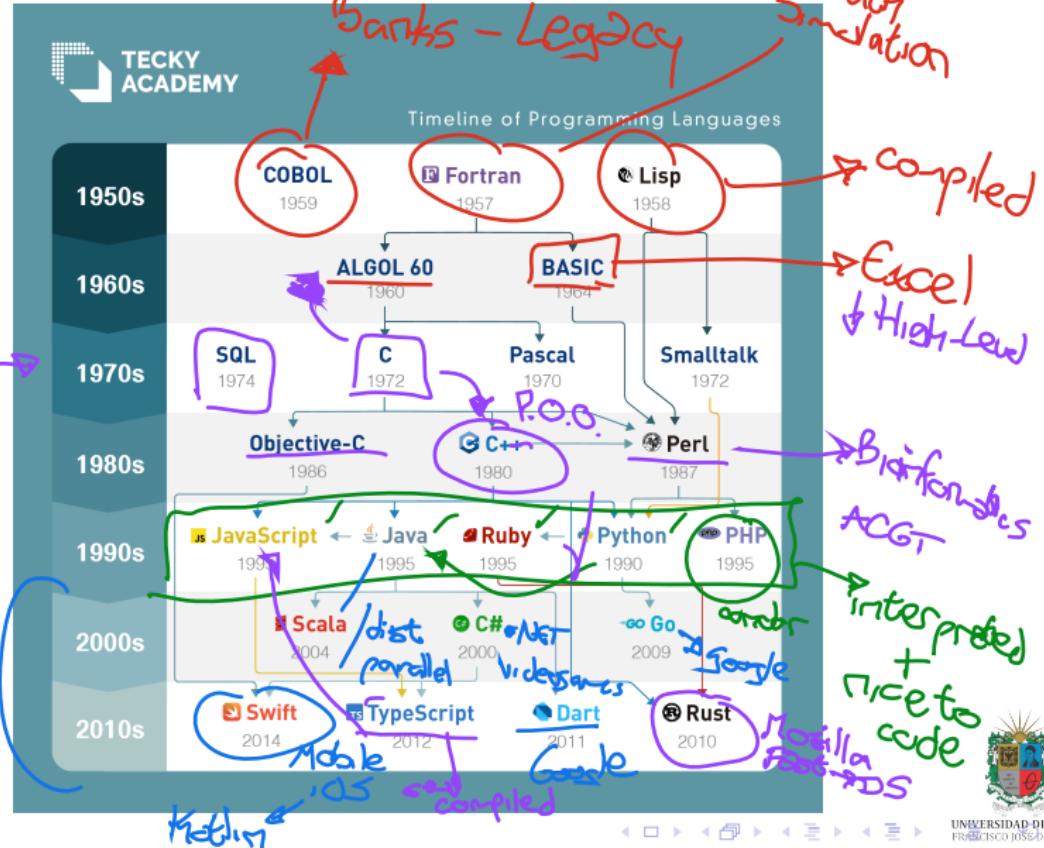
Listing 1: Basic Conditional Structure in Assembly



History of Programming Languages

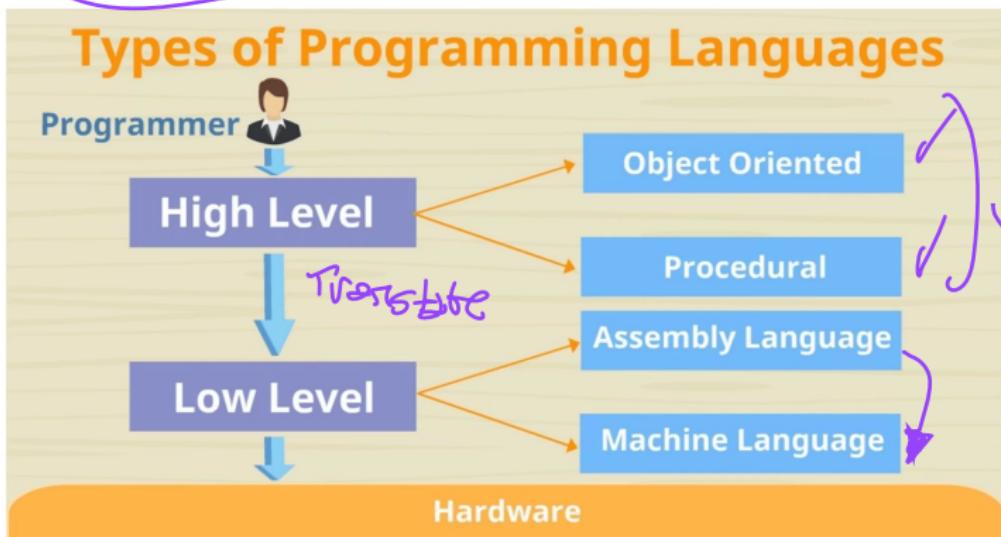
Mel
Fury

50 -
years
ago



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



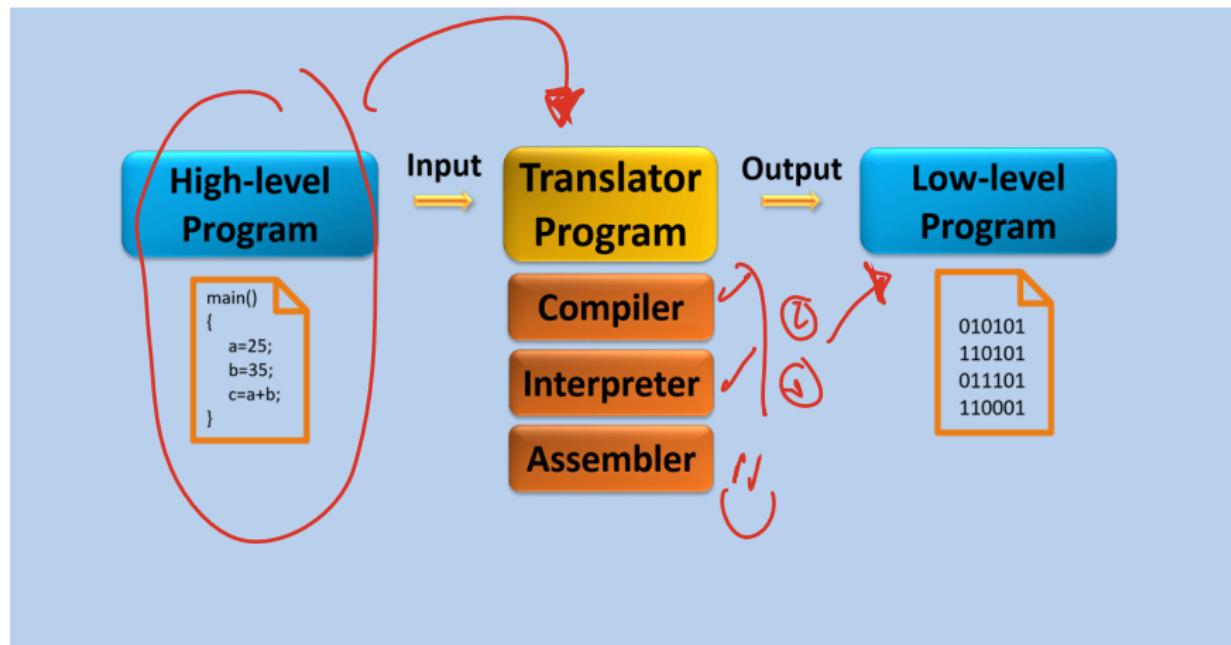
Portability Across Systems

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

Techopedia



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.

~ Cx29,

~ ~

~~ - -

~~

Assembly

Pqdn..

x6 = 5



Programming Languages Levels

32-bit (4-byte) ADD instruction:

100000	00101	00010	00011	000000000000
opcode	rc	ra	rb	(unused)

Could be something like: Reg[4] \leftarrow Reg[2] + Reg[3]

In assembler:

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

Low

In any high-level language like C:

High

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

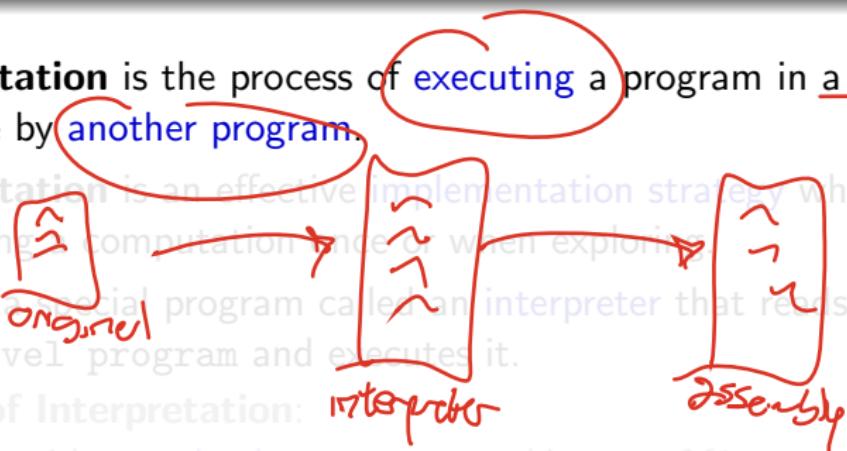


Interpretation

- **Interpretation** is the process of **executing** a program in a high-level language by another program.

- Interpretation is an effective implementation strategy when performing computation needed 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:

execute → reexecute
 - 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.**
- Pointers*
- 



Interpretation Vs. Compilation

Characteristics differences:

	Compilation	Interpretation
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/slowes?	Program Development	Program Execution
Decisions made at	Compile Time	Run Time



C.V.
spanish

official
translator

process

C.V.
english



Outline

1 History of the Computation

2 Programming Languages

3 Finite-State Machines

4 Generative Grammars



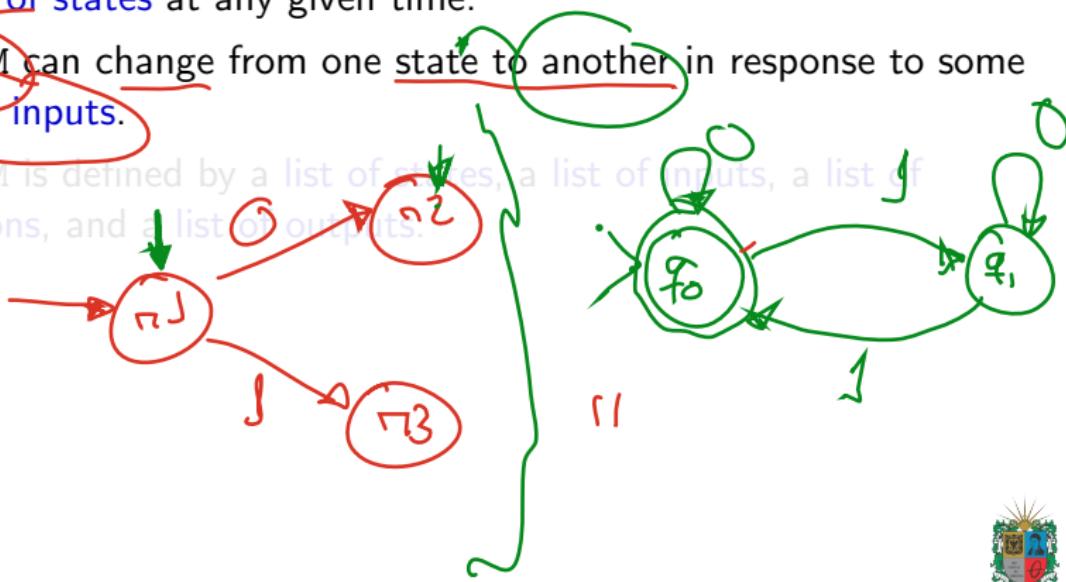
Finite-State Machines

- 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

$dco1000,$
 $q_0q_1q_1q_1q_1q_2$

- 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.

States $\rightarrow q_0, q_1, q_2$

Inputs $\rightarrow 0, 1$

$q_0, 0 \rightarrow q_0$

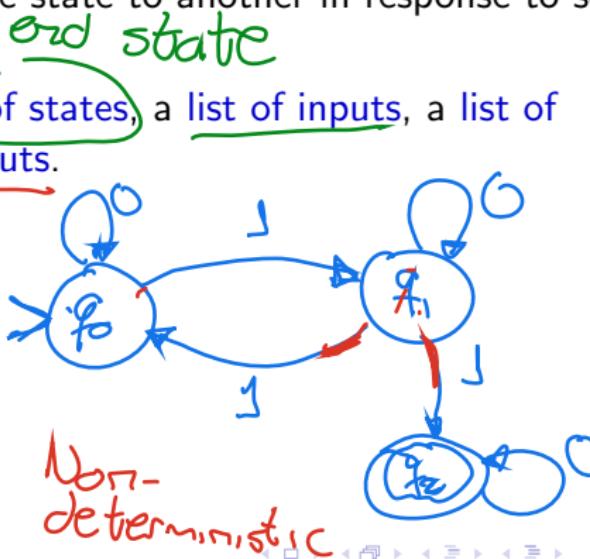
$q_0, 1 \rightarrow q_1$

$q_1, 0 \rightarrow q_1$

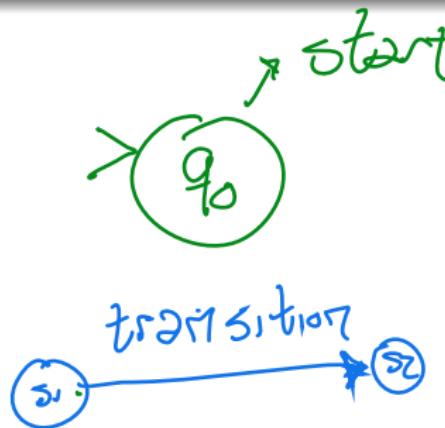
$q_1, 1 \rightarrow q_0$

$q_1, 1 \rightarrow q_2$

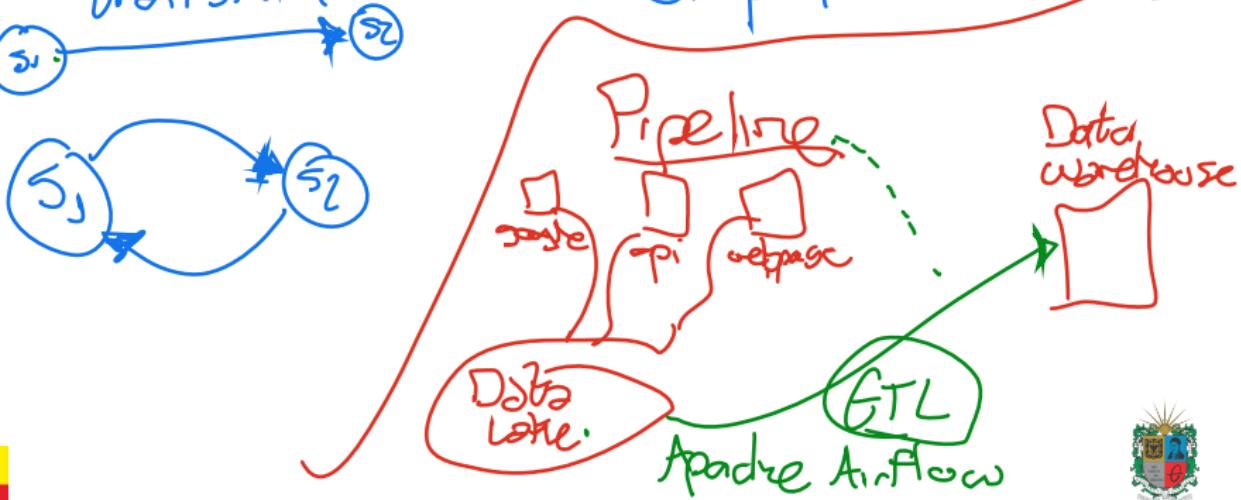
$q_2, 0 \rightarrow q_2$



Finite-State Machine Drawing



~~start, end~~



Alonzo Church

1921

- **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.

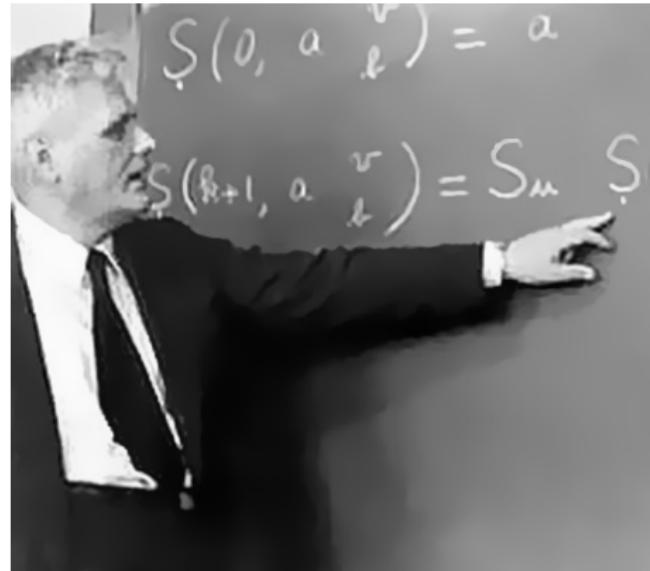


Figure: Alonzo Church



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.

$[a-z]^*$

Kleene's Star

X j@ua...

/ reperez@udistrital.edu.co

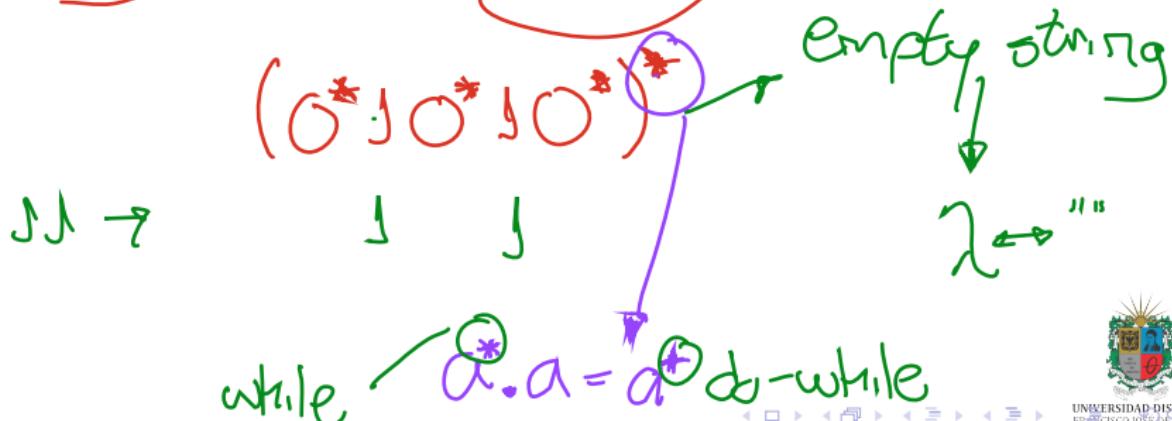
*.json



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

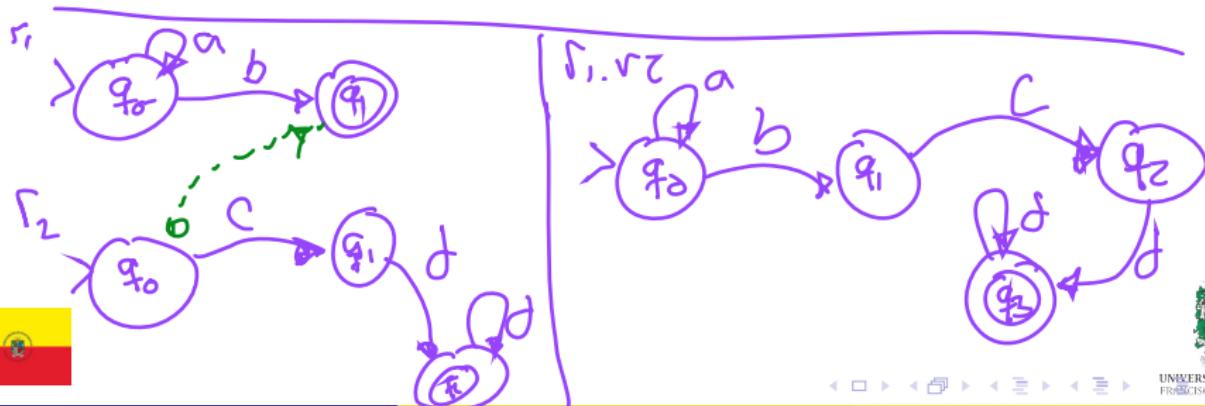
The **concatenation** of two regular expressions R_1 and R_2 is a regular expression that matches the **concatenation** of strings that are matched by R_1 followed by R_2 .

$$r_1 = a^* b$$

$$r_2 = c d^+$$

$$r_1 \cdot r_2 = a^* b c d^+$$

$$r_1 \cdot r_2 = a^* b c d^+$$

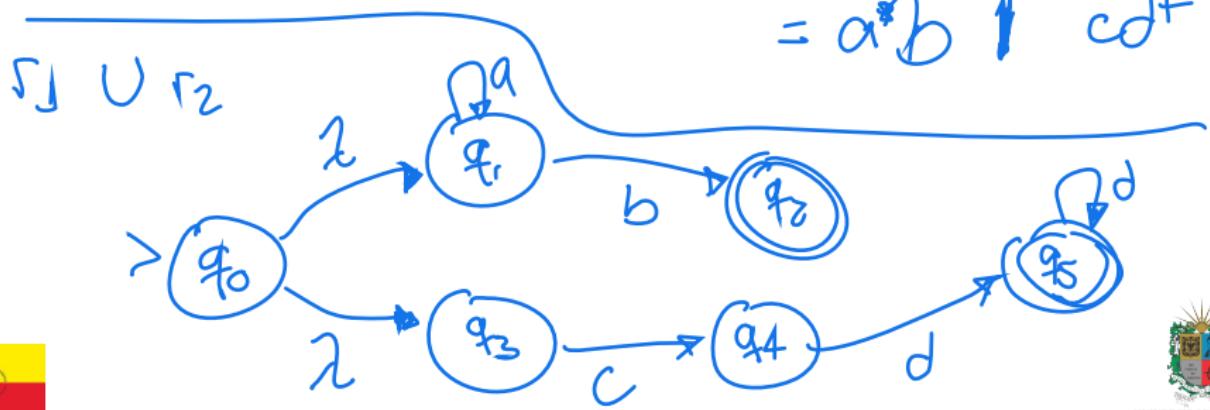


Finite Automata: Union

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 .

$$\begin{aligned} r_1 &= a^* b \\ r_2 &= c d^+ \end{aligned}$$

$$\begin{aligned} r_1 \cup r_2 &= a^* b \cup c d^+ \\ &= a^* b \text{ or } c d^+ \\ &= a^* b \uparrow c d^+ \end{aligned}$$



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 .

$$a^* = a$$

$$a^2 = a \cdot a$$

$$a^0 = \lambda$$

$$a^* = aad \dots a$$

$$a^+ = a \cdot a^*$$

$$a^+ = a^* \cdot a$$

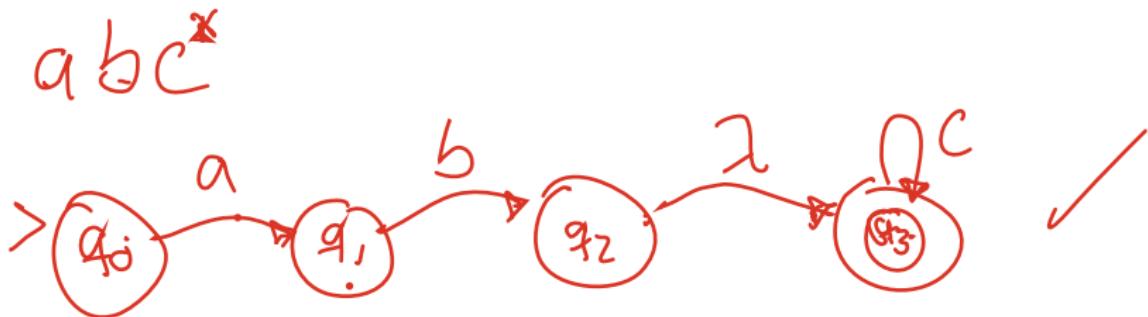
$$a \cdot \lambda = a$$

$$abc \cdot \lambda = abc$$

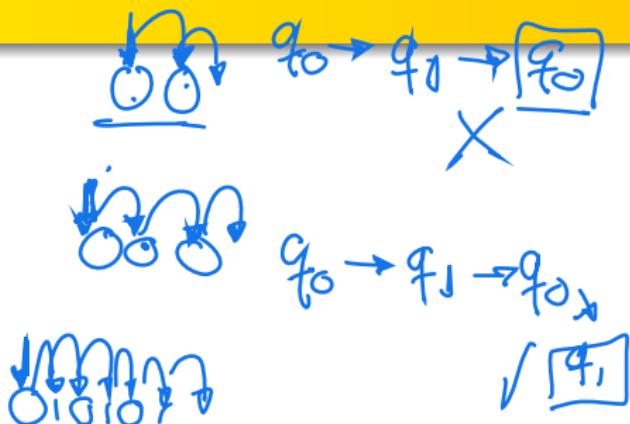
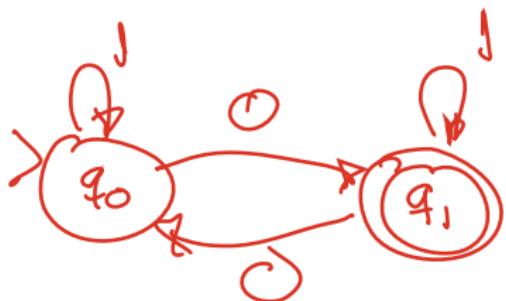


Finite Automata: λ -transition

The λ -transition is a **transition** that can be taken without consuming any input.



Strings Processing



$q_0 \rightarrow q_1 \rightarrow q_1, i = q_2 \rightarrow q_0 \rightarrow q_1 \rightarrow q_1$



Alan Turing

Characterizing

- **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.
Parallel



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 α 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 α 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 α 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>

