Q1: How can you tell if a program is written in C++ / Python / Java / Racket?

Q2: What makes them different?

Finite Automatons

Descriptions of a language

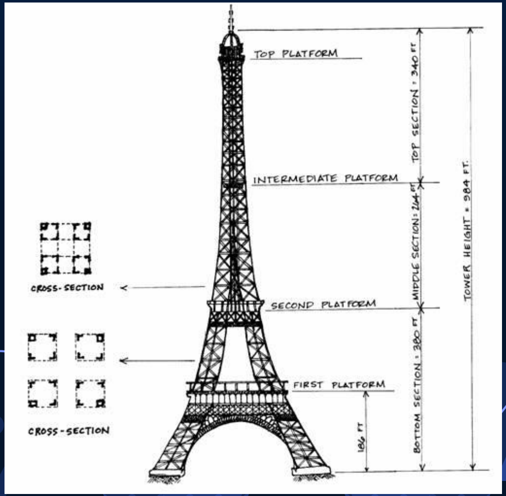
Parts of a language description:

1. **Lexical**

The vocabulary

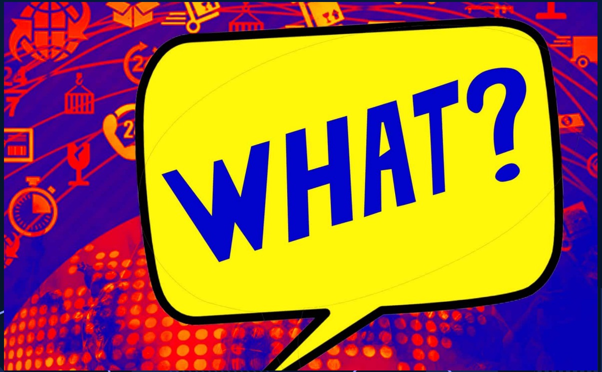
1. **Syntax**

The structure



1. **Semantic**

The meaning

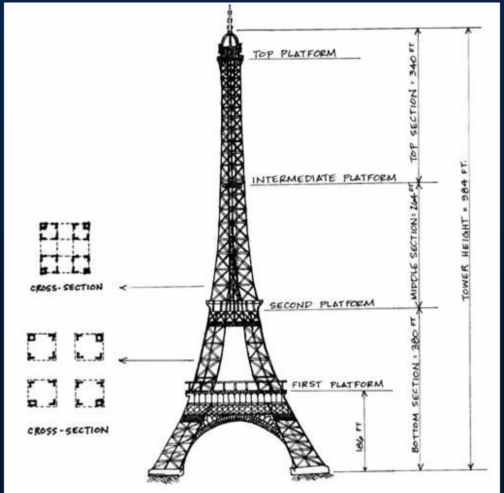


Lexicon

- The structure of the words (tokens) available in a language. For example: tokens used in C for conditionals are if and else.
- Other tokens commonly used in programming languages are:
  - Reserved words
  - Identifiers
  - Operators
  - Punctuation symbols (dot, comma, brackets)

Syntax

- The grammar or structure of a language.
- A description of how tokens can be combined to form valid expressions.
- All computer languages now use a formal definition to describe their grammar. The most common representation is through Context Free Grammars.



The grammar for the C++ Conditional is:

The token if

Followed by an expression enclosed in parenthesis

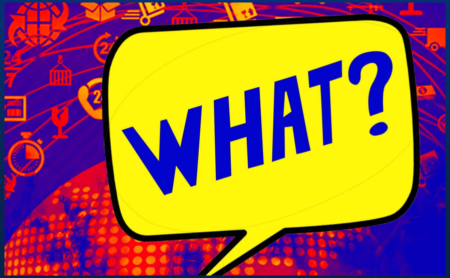
```
std::string animal;
if (animal == "cow")
{
    std::cout << "Mooo!" << std::endl;
}
else
{
    std::cout << "So my cow was stolen" << std::endl;
}
```

Followed by a block of code

Optionally followed by an alternative clause: the token else

Followed by a block of code

Semantic



- The meaning of expressions in a language.
- More difficult to formally describe, since the meaning can be interpreted in different ways.
- In the case of code, meaning would be represented by the effects of the program, but this can become very complex when considering all possible interactions.
- There is no standard description for semantics.

Semantic example:

- The semantic for the C conditional is (adapted from Kernighan and Ritchie [1988]):
    - The expression of the if statement is evaluated, and if it is different from 0, the next block of code is executed.
    - If there is an else statement, and the expression of the if evaluated to 0, then the code block of the else is executed
- It is difficult to cover all possible cases related to the meaning. For example, there is no description for the case the expression is 0 and there is no else clause

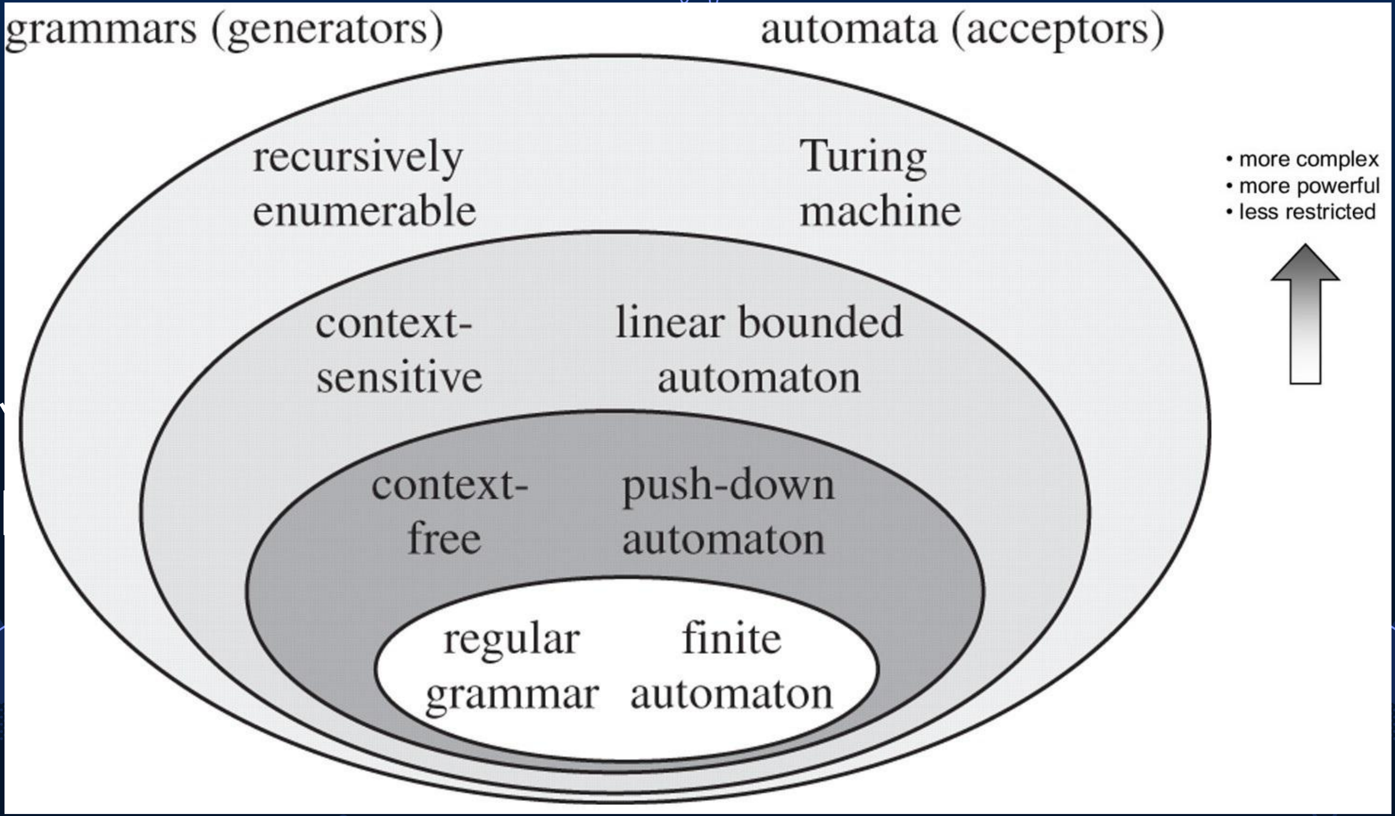
How to formally define lexical and grammatical rules?

Chomsky Hierarchy

Chomsky Hierarchy

- Languages are described in terms of syntax and semantics.
- Sometimes the context also affects the semantics.
- Linguist Noam Chomsky created a categorization of languages relative to their complexity.
- These languages can be represented in different ways.

Chomsky Hierarchy



Chomsky Hierarchy

- Type-3: Regular Grammar - most restrictive of the set, they generate regular languages. They must have a single non-terminal on the left-hand-side and a right-hand-side consisting of a single terminal or single terminal followed by a single non-terminal.

- Type-2: Context-Free Grammar - generate context-free languages, a category of immense interest to NLP practitioners. Here all rules take the form $A \longrightarrow \beta$, where A is a single non-terminal symbol and β is a string of symbols.

Chomsky Hierarchy

- Type-1: Context-Sensitive Grammar - the highest programmable level, they generate context-sensitive languages. They have rules of the form $\alpha A \beta \rightarrow \alpha \gamma \beta$ with A as a non-terminal and α, β, γ as strings of terminals and non-terminals. Strings α, β may be empty, but γ must be nonempty.

- Type-0: Recursively enumerable grammar - are too generic and unrestricted to describe the syntax of either programming or natural languages.

Practical applications

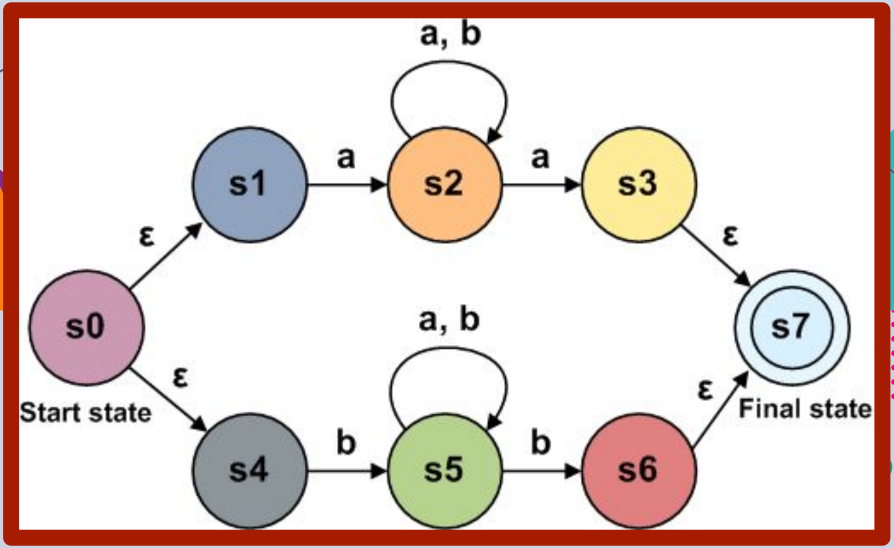
- Type-3: Regular Grammar - Identifying valid tokens in a language (regular expressions)
- Type-2: Context-Free Grammar - Identifying nested constructions, such as parentheses and brackets
- Type-1: Context-Sensitive Grammar - Awareness of conditions that can alter the interpretation
- Type-0: Recursively enumerable grammar - not programmable

When reading a program, Types 3 and 2 are used

1.

State machines to define language rules

Finite automatons



Finite Automatons

- Used to validate an input, as positive or negative
- Finite-state machines are used to determine if a string is accepted as part of a language
- It is not related to the hardware, but to the process used to convert input into output
- State machines can be deterministic if for a given input the output is always the same

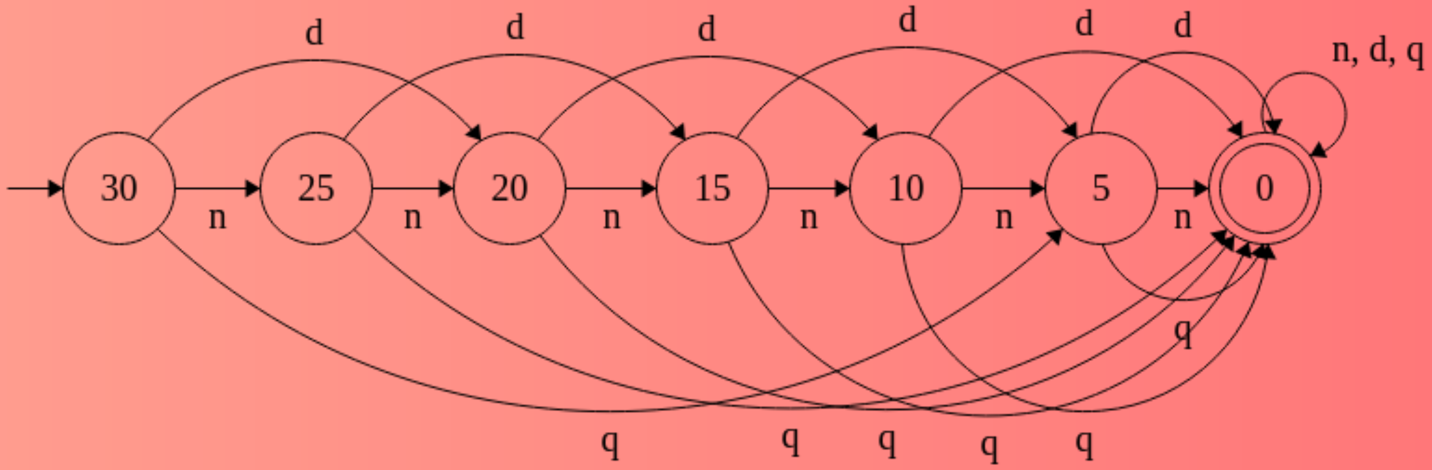
State Machine

- A directed graph, where inputs are used to select a path taken from one state to the next one
- The inputs are taken from a valid set
- If the final state is an accept, the input is correct

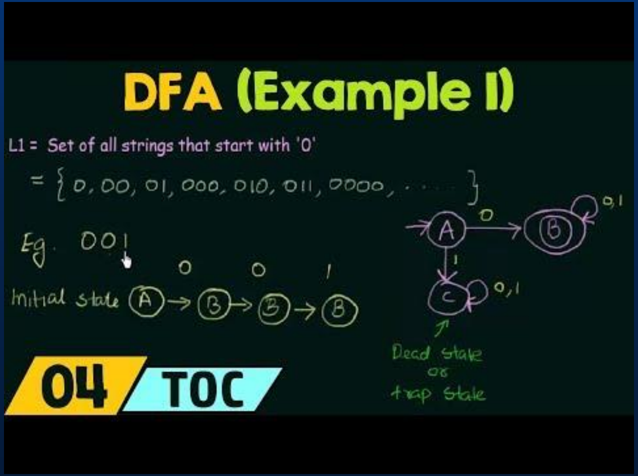
Example of a state machine

- An automatic machine that receives coins (nickel, dime, quarter) and will open the door when receiving 30 cents
- Any more coins inserted after 30 are not returned
- States are named after the amount left to pay

Example of a state machine



Deterministic Finite Automatons (DFA)



DFA

Mathematically described as a 5-tuple:

$$\mathbf{M} = (\mathbf{Q}, \mathbf{\Sigma}, \delta, q_0, \mathbf{F})$$

Where:

Q is the finite set of states

Σ is the finite alphabet

δ is a total function from **Q** x **Σ** to **Q**, known as the transition function

$q_0 \in \mathbf{Q}$ is the initial state

F is the subset of **Q** of accept states

Analogy

A DFA can be described as an abstract machine, with components such as those in a mechanical device:

- A single internal register
- A set of values for the register
- A tape
- A tape reader
- An instruction set

Deterministic

- Since δ is a total function, there is a exactly one instruction defined for each combination of symbol and state

$$\mathbf{Q} \times \mathbf{\Sigma} \text{ to } \mathbf{Q}$$

Language

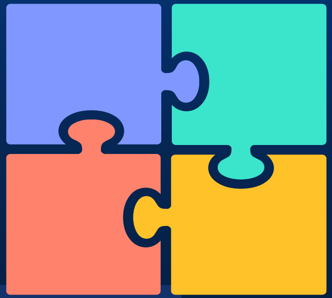
Given the definition of a DFA as

$$\mathbf{M} = (\mathbf{Q}, \boldsymbol{\Sigma}, \delta, q_0, \mathbf{F})$$

The language of **M**, denoted **L**(**M**) is the set of strings in $\boldsymbol{\Sigma}^*$ accepted by **M.**

A DFA is considered a language acceptor.

DFA representation



The example of the newspaper machine:

Q = {0, 5, 10, 15, 20, 25, 30}

Σ = {n, d, q}

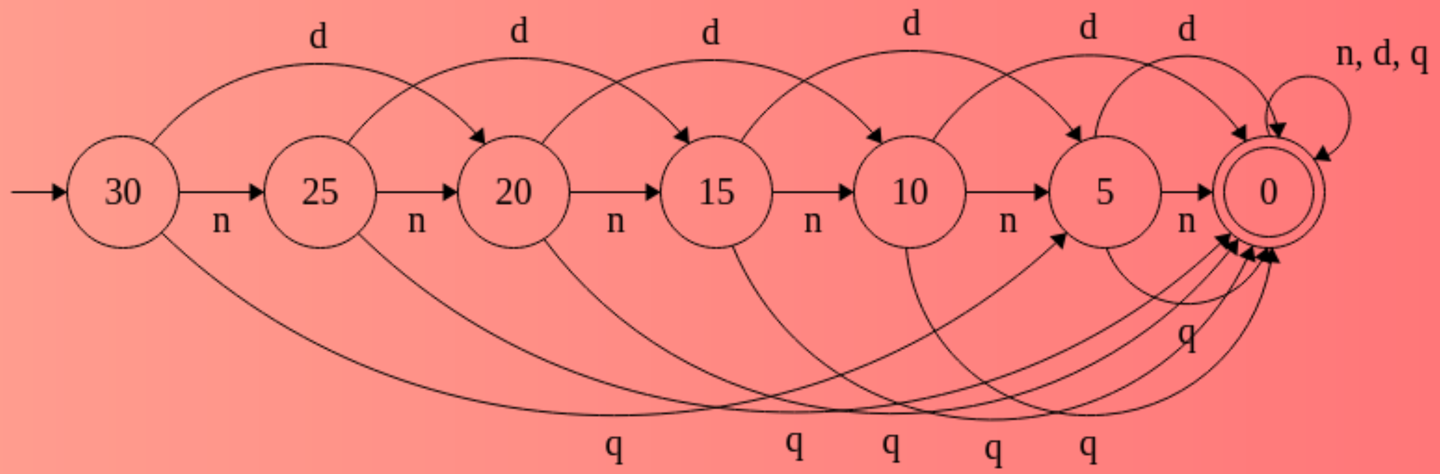
F = {0}

q_0 = 30

δ =

	n	d	q
0	0	0	0
5	0	0	0
10	5	0	0
15	10	5	0
20	15	10	0
25	20	15	0
30	25	20	5

Example of a state machine



Extended transition function

Represents the transitions followed by a string w

Represented by $\hat{\delta}$ as a function from **Q** x $\mathbf{\Sigma}^*$ to **Q**

The value of $\hat{\delta}$ is defined recursively:

I. Basis:

length(w) = 0. Then $w = \lambda$ and $\hat{\delta}(q_i, \lambda) = q_i$

length(w) = 1. Then $w = a$ for some a $\in \mathbf{\Sigma}$

and $\hat{\delta}(q_i, \mathrm{a}) = \delta(q_i, \mathrm{a})$

II. Recursive step:

Let w be a string of length n>1. Then $w = ua$ and $\hat{\delta}(q_i, ua) = \delta(\hat{\delta}(q_i, u), a)$

Example

What is the result of $\hat{\delta}(30, nndd)$

$w =$ nndd

$\hat{\delta}(30, nndd)$

$\delta(\ \hat{\delta}(30, nnd), d)$

$\delta(\ \delta(\ \hat{\delta}(30, nn), d), d)$

$\delta(\ \delta(\ \delta(\ \hat{\delta}(30, n), n), d), d)$

$\delta(\ \delta(\ \delta(\ \delta(30, n), n), d), d)$

$\delta(\ \delta(\ \delta(\ 25, n), d), d)$

$\delta(\ \delta(\ 20, d), d)$

$\delta(\ 10, d)$

0

String acceptance

A string w is accepted by **M** if $\delta\hat{}(q_0, w) \in \mathbf{F}$

With this notation, the language of a DFA **M** is the set:

$$\mathbf{L(M)} = \{w \mid \delta\hat{}(q_0, w) \in \mathbf{F}\}$$

State diagram

It is a directed graph representing a FA:

- Vertices represent states in **Q**, shown as circles
  - q_o is the start state, indicated by an arrow pointing to it
  - Accept states **F** are represented by a double circle
- Edges represent transitions between states, and are labeled with the symbol of **Σ** that triggers the transition.
- The FA accepts a word if there is a path from the start state to an accept state.

Examples

Using the alphabet **Σ** = {*a*, *b*}

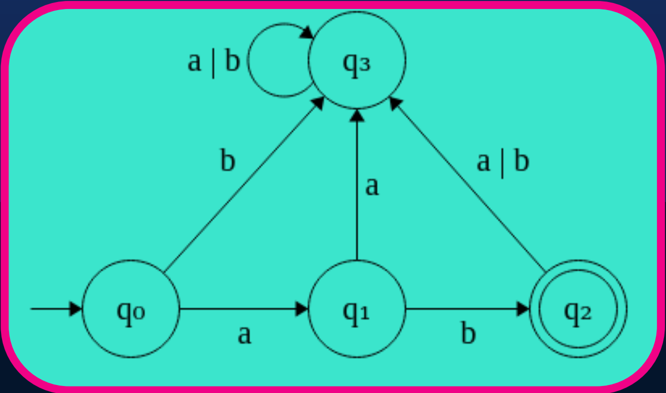
- Create a FA that accepts only the string *ab*
- Create a FA that accepts strings that start with *ba*
- Create a FA that accepts all strings that contain *bb*
- Create a FA that accepts all strings that start with *a* and end with *b*

Solutions

Using the alphabet $\boldsymbol{\Sigma} = \{a, b\}$

- Create a FA that accepts only the string *ab*

$M = (\{q_0, q_1, q_2, q_3\}, \{a, b\}, \delta, q_0, \{q_2\})$



$\delta =$

	a	b
q_0	q_1	q_3
q_1	q_3	q_2
q_2	q_3	q_3
q_3	q_3	q_3

Solutions

Using the alphabet $\boldsymbol{\Sigma} = \{a, b\}$

- Create a FA that accepts strings that start with *ba*

$M = (\{q_0, q_1, q_2, q_3\}, \{a, b\}, \delta, q_0, \{q_2\})$

$\delta =$

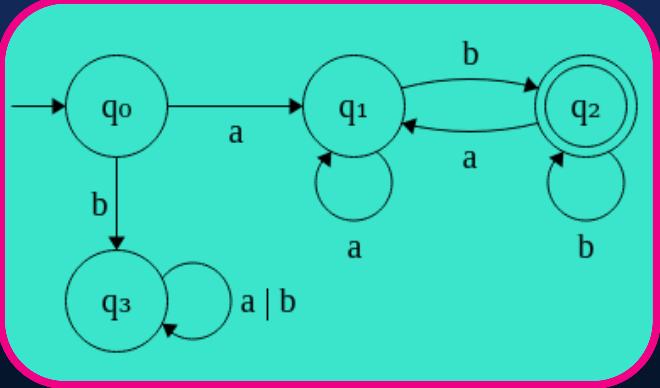
	a	b
q_0	q_3	q_1
q_1	q_2	q_3
q_2	q_2	q_2
q_3	q_3	q_3

Solutions

Using the alphabet **Σ** = {*a*, *b*}

- Create a FA that accepts all strings that start with *a* and end with *b*

$M = (\{q_0, q_1, q_2, q_3\}, \{a, b\}, \delta, q_0, \{q_2\})$



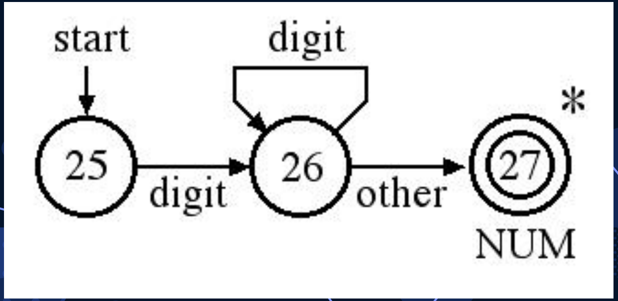
$\delta =$

	a	b
q_0	q_1	q_3
q_1	q_1	q_2
q_2	q_1	q_2
q_3	q_3	q_3

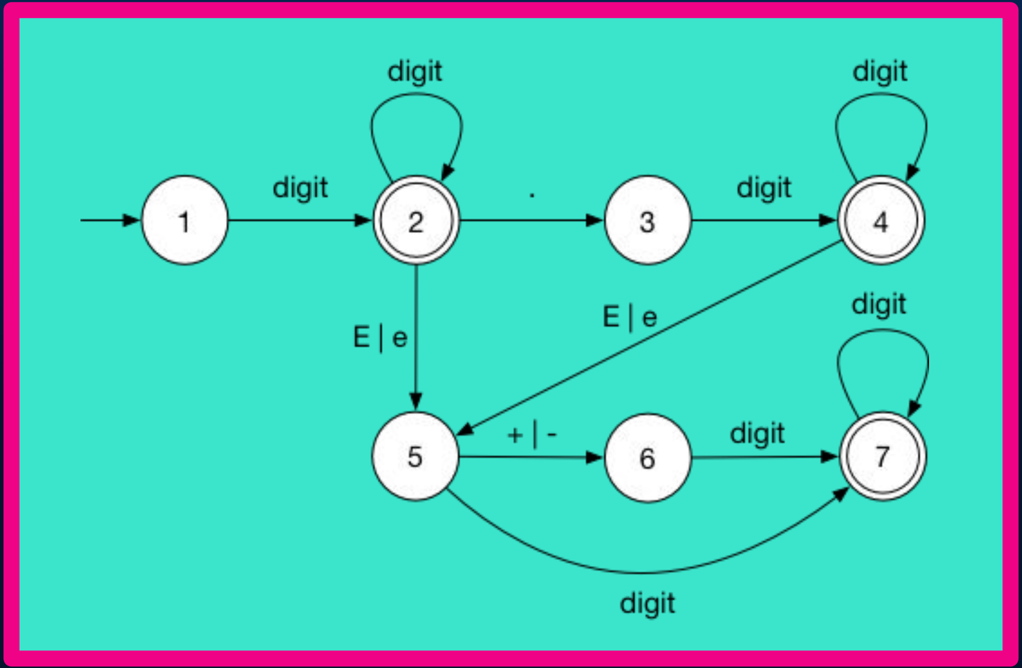
Real world application

Token recognition

- Compilers use FAs or regular expressions to identify valid tokens
- Example: identifying integer numbers
  - Regular expression: `{0,1,2,3,4,5,6,7,8,9}`$^+$, also represented as `[0-9]`$^+$.
  - Finite Automaton:



FA (incomplete) example: number token



https://hackernoon.com/lexical-analysis-861b8bfe4cb0

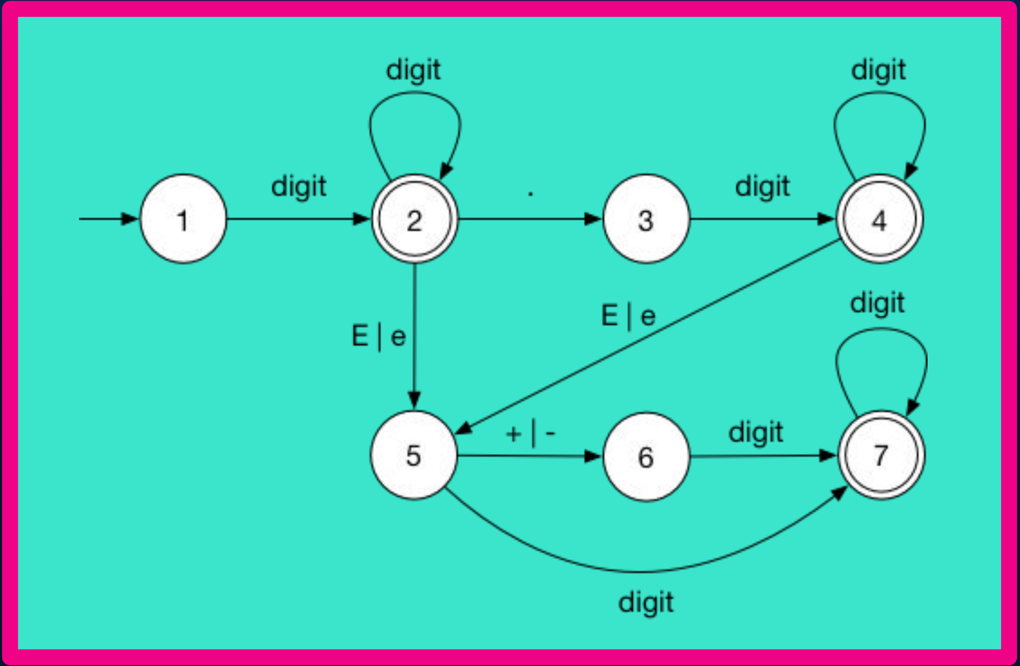
RE example: number token

RE:

\d+([.]\d+)?([eE][+-]?\d+)?

Non capturing groups:

\d+(?:[.]\d+)?(?:[eE][+-]?\d+)?



https://hackernoon.com/lexical-analysis-861b8bfe4cb0

Describing a way to detect tokens in a language

Conclusion

THANKS!

Do you have any questions?
Contact me at:
g.echeverria@tec.mx

References:

Nombre: Languages and machines: an introduction to the theory of computer science

Autor: Thomas Sudkamp

Edición: 3rd

Año: 2016

Editorial: Addison-Wesley

ISBN: 9780321322210

References:

https://devopedia.org/chomsky-hierarchy

https://regex101.com/

http://madebyevan.com/fsm/

https://hackernoon.com/lexical-analysis-861b8bfe4cb0

https://dev.to/mconner89/regular-expressions-grouping-and-string-methods-3ijn

DIAGRAMS AND INFOGRAPHICS