Lecture 2: Static Analysis Principles – Lexical and Syntactical Language Analysis

Passive Testing Techniques for Communication Protocols

Dr. Jorge López, PhD.
jorgelopezcoronado[at]gmail.com

National Research Tomsk State University

February 15, 2016

Acknowledgments

Some content was taken from: Saman Amarasinghe, and Martin Rinard. 6.035 Computer Language Engineering, Spring 2010. (Massachusetts Institute of Technology: MIT OpenCourseWare), http://ocw.mit.edu. License: Creative Commons BY-NC-SA

Outline

Goal & Motivation

Lexical Analysis using Finite State Automata / Regular Expressions

Syntactical Analysis with Context Free Grammars

STATIC ANALYSIS

```
/*Test equal distribution of random number generation algorithm*/
#include <stdio.h>
#include <stdlib.h>
#define NUM 30
#define MEM_SIZE 512*1024 //512MB
int main()
{
        long i = 0 , j;
        short acc = 0;
        if(!numbers)
        {
                printf(``Can't allocate memory\n'');
                exit(-1);
        }
        while (1)
        {
                numbers[i] = rand() % NUM ;  //random numbers from 0 - NUM
                acc = 0;
                for (j = 0; j < i; j++)
                        acc += numbers[j];
                printf(``New average: %ld\n'',  acc/++i); //should converge to NUM/2
        }
}
```

Do you see any problems with this code?

Static Analysis (cont.)

```
/*Test equal distribution of random number generation algorithm*/
#include <stdio.h>
#include <stdlib.h>
#define NUM 30
#define MEM_SIZE 512*1024 //512MB
int main()
{
        long i = 0 , j;
        short acc = 0;
        if(!numbers)
        {
                printf(``Can't allocate memory\n'');
                exit(-1);
        }
        while (1)
        {
                numbers[i] = rand() % NUM ;  //random numbers from 0 - NUM
                acc = 0;
                for (j = 0; j < i; j++)
                        acc += numbers[j];
                printf(``New average: %ld\n'',  acc/++i); //should converge to NUM/2
        }
}
```

See how hard it is? :)

STATIC ANALYSIS PRINCIPLES

Source Code is?..

Static Analysis Principles

Source Code is?..

- A text representation of “computer instructions” in a specific **programming language**

Static Analysis Principles

Source Code is?..

- A text representation of "computer instructions" in a specific **programming language**
- Not text. Nor machine code

Static Analysis Principles

Source Code is?..

- A text representation of "computer instructions" in a specific **programming language**
- Not text. Nor machine code
- A description of a system

Static Analysis Principles

Source Code is?..

- A text representation of “computer instructions” in a specific **programming language**
- Not text. Nor machine code
- A description of a system

Static Analysis is?..

Static Analysis Principles

Source Code is?..

- A text representation of “computer instructions” in a specific **programming language**
- Not text. Nor machine code
- A description of a system

Static Analysis is?..

- The analysis (performed by a program) of code without executing the program

Static Analysis Principles

Source Code is?..

- A text representation of "computer instructions" in a specific **programming language**
- Not text. Nor machine code
- A description of a system

Static Analysis is?..

- The analysis (performed by a program) of code without executing the program
- Not looking for lexical, syntactical, or *type* errors that a compiler finds, i.e., the code is assumed to be compilable

Static Analysis Principles (cont.)

How do we analyze source code then?

Static Analysis Principles (cont.)

How do we analyze source code then?

- Can we treat it as text?

Static Analysis Principles (cont.)

How do we analyze source code then?

- Can we treat it as text?
  - No. This would be terribly^n hard

Static Analysis Principles (cont.)

How do we analyze source code then?

- Can we treat it as text?
  - No. This would be terribly^n hard
- We need to manipulate it with a structure

Static Analysis Principles (cont.)

How do we analyze source code then?

- Can we treat it as text?
  - No. This would be terribly^n hard
- We need to manipulate it with a structure
  - A famous structure to manipulate source code is an **Abstract Syntax Tree (AST)**

Static Analysis Principles (cont.)

How do we analyze source code then?

- Can we treat it as text?
  - No. This would be terribly^n hard
- We need to manipulate it with a structure
  - A famous structure to manipulate source code is an **Abstract Syntax Tree (AST)**
  - To build an AST, we need to parse the code

Static Analysis Principles (cont.)

How do we analyze source code then?

- Can we treat it as text?
  - No. This would be terribly^n hard
- We need to manipulate it with a structure
  - A famous structure to manipulate source code is an **Abstract Syntax Tree (AST)**
  - To build an AST, we need to parse the code
  - To parse the code, we need to tokenize

Static Analysis Principles (cont.)

How do we analyze source code then?

- Can we treat it as text?
  - No. This would be terribly^n hard
- We need to manipulate it with a structure
  - A famous structure to manipulate source code is an **Abstract Syntax Tree (AST)**
  - To build an AST, we need to parse the code
  - To parse the code, we need to tokenize
  - Wait, what?

Static Analysis Principles (cont.)

How do we analyze source code then?

- Can we treat it as text?
  - No. This would be terribly^n hard
- We need to manipulate it with a structure
  - A famous structure to manipulate source code is an **Abstract Syntax Tree (AST)**
  - To build an AST, we need to parse the code
  - To parse the code, we need to tokenize
  - Wait, what?

How to structure code: A natural language comparison

Static Analysis Principles (cont.)

How do we analyze source code then?

- Can we treat it as text?
  - No. This would be terribly^n hard
- We need to manipulate it with a structure
  - A famous structure to manipulate source code is an **Abstract Syntax Tree (AST)**
  - To build an AST, we need to parse the code
  - To parse the code, we need to tokenize
  - Wait, what?

How to structure code: A natural language comparison

- A language can produce sentences. Sentences are structured; composed by words

Static Analysis Principles (cont.)

How do we analyze source code then?

- Can we treat it as text?
    - No. This would be terribly^n hard
- We need to manipulate it with a structure
    - A famous structure to manipulate source code is an **Abstract Syntax Tree (AST)**
    - To build an AST, we need to parse the code
    - To parse the code, we need to tokenize
    - Wait, what?

How to structure code: A natural language comparison

- A language can produce sentences. Sentences are structured; composed by words
- How to recognize allowed words in a language?

Lexical Analysis

using Deterministic Finite Automata (DFA) and Regular Expressions (RE)

Lexical Analysis

DFA and RE equivalence

Lexical Analysis

DFA and RE equivalence

- A DFA can recognize a language

LEXICAL ANALYSIS

DFA and RE equivalence

- A DFA can recognize a language
- A RE can describe how to generate a language

Lexical Analysis

DFA and RE equivalence

- A DFA can recognize a language
- A RE can describe how to generate a language
- The language of a DFA is the set of all strings that the automaton can accept while the language of a RE is the set of all strings that the RE can generate

Lexical Analysis

DFA and RE equivalence

- A DFA can recognize a language
- A RE can describe how to generate a language
- The language of a DFA is the set of all strings that the automaton can accept while the language of a RE is the set of all strings that the RE can generate
- There is a way to go from one to the other, describing the same language

Lexical Analysis

DFA and RE equivalence

- A DFA can recognize a language
- A RE can describe how to generate a language
- The language of a DFA is the set of all strings that the automaton can accept while the language of a RE is the set of all strings that the RE can generate
- There is a way to go from one to the other, describing the same language

Let's start with regular expressions…

Regular expressions

Given a finite alphabet A, a regular expression is:

Regular expressions

Given a finite alphabet A, a regular expression is:

- ε — The empty "string"

Regular expressions

Given a finite alphabet A, a regular expression is:

- ε — The empty "string"
- $a \in A$ – A letter form the alphabet

Regular expressions

Given a finite alphabet A, a regular expression is:

- ε — The empty "string"
- $a \in A$ – A letter form the alphabet
- $r_1 r_2$ – RE r_1 followed by the RE r_2

Regular expressions

Given a finite alphabet A, a regular expression is:

- ε — The empty "string"
- $a \in A$ – A letter form the alphabet
- $r_1 r_2$ – RE r_1 followed by the RE r_2
- $r_1 | r_2$ – either RE r_1 or RE r_2 (choice)

Regular expressions

Given a finite alphabet A, a regular expression is:

- ε — The empty "string"
- $a \in A$ – A letter form the alphabet
- $r_1 r_2$ – RE r_1 followed by the RE r_2
- $r_1 | r_2$ – either RE r_1 or RE r_2 (choice)
- r^* – Kleene star = $\varepsilon | r | rr | rrr | \ldots$

Regular expressions

Given a finite alphabet A, a regular expression is:

- ε — The empty "string"
- $a \in A$ – A letter form the alphabet
- $r_1 r_2$ – RE r_1 followed by the RE r_2
- $r_1 | r_2$ – either RE r_1 or RE r_2 (choice)
- r^* – Kleene star = $\varepsilon | r | rr | rrr | \ldots$
- (r) – grouping / precedence, precedence defines first Kleene star, then concatenation, then choice (in case no parenthesis is specified)

Regular expressions

Given a finite alphabet A, a regular expression is:

- ε — The empty "string"
- $a \in A$ – A letter form the alphabet
- $r_1 r_2$ – RE r_1 followed by the RE r_2
- $r_1 | r_2$ – either RE r_1 or RE r_2 (choice)
- r^* – Kleene star = $\varepsilon | r | rr | rrr | \ldots$
- (r) – grouping / precedence, precedence defines first Kleene star, then concatenation, then choice (in case no parenthesis is specified)

Example

Over the alphabet $\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$,
$(1|2|3|4|5|6|7|8|9)(0|1|2|3|4|5|6|7|8|9)^*$

Regular expressions (cont.)

Generating from $(1|2|3|4|5|6|7|8|9)(0|1|2|3|4|5|6|7|8|9)^*$

Regular expressions (cont.)

Generating from $(1|2|3|4|5|6|7|8|9)(0|1|2|3|4|5|6|7|8|9)^*$

- $(1|2|3|4|5|6|7|8|9) \mapsto 8$
  $8(0|1|2|3|4|5|6|7|8|9)^*$

Regular expressions (cont.)

Generating from $(1|2|3|4|5|6|7|8|9)(0|1|2|3|4|5|6|7|8|9)^*$

- $(1|2|3|4|5|6|7|8|9) \mapsto 8$
  $8(0|1|2|3|4|5|6|7|8|9)^*$
- $(0|1|2|3|4|5|6|7|8|9)^* \mapsto 9(0|1|2|3|4|5|6|7|8|9)^*$
  $89(0|1|2|3|4|5|6|7|8|9)^*$

Regular expressions (cont.)

Generating from $(1|2|3|4|5|6|7|8|9)(0|1|2|3|4|5|6|7|8|9)^*$

- $(1|2|3|4|5|6|7|8|9) \mapsto 8$
  $8(0|1|2|3|4|5|6|7|8|9)^*$
- $(0|1|2|3|4|5|6|7|8|9)^* \mapsto 9(0|1|2|3|4|5|6|7|8|9)^*$
  $89(0|1|2|3|4|5|6|7|8|9)^*$
- $(0|1|2|3|4|5|6|7|8|9)^* \mapsto \varepsilon$
  89

Regular expressions (cont.)

Generating from $(1|2|3|4|5|6|7|8|9)(0|1|2|3|4|5|6|7|8|9)^*$

- $(1|2|3|4|5|6|7|8|9) \mapsto 8$
  $8(0|1|2|3|4|5|6|7|8|9)^*$
- $(0|1|2|3|4|5|6|7|8|9)^* \mapsto 9(0|1|2|3|4|5|6|7|8|9)^*$
  $89(0|1|2|3|4|5|6|7|8|9)^*$
- $(0|1|2|3|4|5|6|7|8|9)^* \mapsto \varepsilon$
  89

Remarks on generation

Regular expressions (cont.)

Generating from $(1|2|3|4|5|6|7|8|9)(0|1|2|3|4|5|6|7|8|9)^*$

- $(1|2|3|4|5|6|7|8|9) \mapsto 8$
  $8(0|1|2|3|4|5|6|7|8|9)^*$
- $(0|1|2|3|4|5|6|7|8|9)^* \mapsto 9(0|1|2|3|4|5|6|7|8|9)^*$
  $89(0|1|2|3|4|5|6|7|8|9)^*$
- $(0|1|2|3|4|5|6|7|8|9)^* \mapsto \varepsilon$
  89

Remarks on generation

- Can be non-deterministic, applying different rules will produce different results

Regular expressions (cont.)

Generating from $(1|2|3|4|5|6|7|8|9)(0|1|2|3|4|5|6|7|8|9)^*$

- $(1|2|3|4|5|6|7|8|9) \mapsto 8$
  $8(0|1|2|3|4|5|6|7|8|9)^*$
- $(0|1|2|3|4|5|6|7|8|9)^* \mapsto 9(0|1|2|3|4|5|6|7|8|9)^*$
  $89(0|1|2|3|4|5|6|7|8|9)^*$
- $(0|1|2|3|4|5|6|7|8|9)^* \mapsto \varepsilon$
  89

Remarks on generation

- Can be non-deterministic, applying different rules will produce different results
- Our goal is to describe language with a regular expression

EXERCISES WITH REs

What language the RE describes?

Exercises with REs

What language the RE describes?

- $A = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$, RE = $(1|2|3|4|5|6|7|8|9)(0|1|2|3|4|5|6|7|8|9)^*$

Exercises with REs

What language the RE describes?

- $A = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$, RE = $(1|2|3|4|5|6|7|8|9)(0|1|2|3|4|5|6|7|8|9)^*$
  - $\mathbb{Z}^+$

Exercises with REs

What language the RE describes?

- $A = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$, RE = $(1|2|3|4|5|6|7|8|9)(0|1|2|3|4|5|6|7|8|9)^*$
  - $\mathbb{Z}^+$
- $A = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$, RE = $(0|1|2|3|4|5|6|7|8|9)^*$

Exercises with REs

What language the RE describes?

- $A = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$, RE = $(1|2|3|4|5|6|7|8|9)(0|1|2|3|4|5|6|7|8|9)^*$
  - $\mathbb{Z}^+$
- $A = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$, RE = $(0|1|2|3|4|5|6|7|8|9)^*$
  - $\mathbb{Z}^+ \cup \{0\}|\varepsilon$

Exercises with REs

What language the RE describes?

- $A = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$, RE = $(1|2|3|4|5|6|7|8|9)(0|1|2|3|4|5|6|7|8|9)^*$
  - $\mathbb{Z}^+$
- $A = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$, RE = $(0|1|2|3|4|5|6|7|8|9)^*$
  - $\mathbb{Z}^+ \cup \{0\}|\varepsilon$

Create the RE such that…

Exercises with REs

What language the RE describes?

- $A = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$, RE = $(1|2|3|4|5|6|7|8|9)(0|1|2|3|4|5|6|7|8|9)^*$
  - $\mathbb{Z}^+$
- $A = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$, RE = $(0|1|2|3|4|5|6|7|8|9)^*$
  - $\mathbb{Z}^+ \cup \{0\}|\varepsilon$

Create the RE such that…

- It generates valid Binary Code Decimal (BCD) strings (In case you don't know, binary strings of length 4, representing digits, example: 1001 1000 0011 = 983, note that 1100 0000 0101 is not valid)

EXERCISES WITH REs

What language the RE describes?

- $A = \{0,1,2,3,4,5,6,7,8,9\}$, RE = $(1|2|3|4|5|6|7|8|9)(0|1|2|3|4|5|6|7|8|9)^*$
  - $\mathbb{Z}^+$
- $A = \{0,1,2,3,4,5,6,7,8,9\}$, RE = $(0|1|2|3|4|5|6|7|8|9)^*$
  - $\mathbb{Z}^+ \cup \{0\} | \varepsilon$

Create the RE such that...

- It generates valid Binary Code Decimal (BCD) strings (In case you don't know, binary strings of length 4, representing digits, example: 1001 1000 0011 = 983, note that 1100 0000 0101 is not valid)
  - $A = \{0, 1, \text{“ ”}\}$, RE = $100(1|0)|0(1|0)(1|0)(1|0)\text{“ ”}(100(1|0)|0(1|0)(1|0)(1|0)\text{“ ”})^*$

Short RE notations

For convenience…

Short RE notations

For convenience…

- Dot (.) represents any character of A
  - e.g., $A = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$, the RE $(0|1|2|3|4|5|6|7|8|9)^* \mapsto .^*$

Short RE notations

For convenience…

- Dot (.) represents any character of A
  - e.g., $A = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$, the RE $(0|1|2|3|4|5|6|7|8|9)^* \mapsto .^*$
- ? represents 0 or 1 occurrences of the preceding RE
  - e.g., for $A = \{0, 1\}$, the RE $(1|\varepsilon)(01)^* \mapsto 1?(01)^*$

SHORT RE NOTATIONS

For convenience…

- Dot (.) represents any character of A
  - e.g., $A = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$, the RE $(0|1|2|3|4|5|6|7|8|9)^* \mapsto .^*$
- ? represents 0 or 1 occurrences of the preceding RE
  - e.g., for $A = \{0, 1\}$, the RE $(1|\varepsilon)(01)^* \mapsto 1?(01)^*$
- $^+$ represents at least 1 occurrence of the preceding RE
  - e.g., for $A = \{a, b\}$, RE = $(a|b)(a|b)^* \mapsto .^+$

SHORT RE NOTATIONS

For convenience...

- Dot (.) represents any character of A
  - e.g., $A = \{0,1,2,3,4,5,6,7,8,9\}$, the RE $(0|1|2|3|4|5|6|7|8|9)^* \mapsto .^*$
- ? represents 0 or 1 occurrences of the preceding RE
  - e.g., for $A = \{0,1\}$, the RE $(1|\varepsilon)(01)^* \mapsto 1?(01)^*$
- $^+$ represents at least 1 occurrence of the preceding RE
  - e.g., for $A = \{a,b\}$, RE = $(a|b)(a|b)^* \mapsto .^+$
- $\{n\}$ represents exactly n occurrences of the preceding RE
  - e.g., for $A = \{a,b\}$, RE = $aaa(a|b)(a|b) \mapsto a\{3\}.\{2\}$

SHORT RE NOTATIONS

For convenience…

- Dot (.) represents any character of A
  - e.g., $A = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$, the RE $(0|1|2|3|4|5|6|7|8|9)^* \mapsto .^*$
- ? represents 0 or 1 occurrences of the preceding RE
  - e.g., for $A = \{0, 1\}$, the RE $(1|\varepsilon)(01)^* \mapsto 1?(01)^*$
- $^+$ represents at least 1 occurrence of the preceding RE
  - e.g., for $A = \{a, b\}$, RE = $(a|b)(a|b)^* \mapsto .^+$
- $\{n\}$ represents exactly n occurrences of the preceding RE
  - e.g., for $A = \{a, b\}$, RE = $aaa(a|b)(a|b) \mapsto a\{3\}.\{2\}$
- $\{m,\}$ represents at least m occurrences of the preceding RE
  - e.g., for $A = \{a, b\}$, RE = $aaa(a*)b \mapsto a\{3,\}b$

Short RE notations

For convenience...

- Dot (.) represents any character of A
  - e.g., $A = \{0,1,2,3,4,5,6,7,8,9\}$, the RE $(0|1|2|3|4|5|6|7|8|9)^* \mapsto .^*$
- ? represents 0 or 1 occurrences of the preceding RE
  - e.g., for $A = \{0,1\}$, the RE $(1|\varepsilon)(01)^* \mapsto 1?(01)^*$
- $^+$ represents at least 1 occurrence of the preceding RE
  - e.g., for $A = \{a,b\}$, RE = $(a|b)(a|b)^* \mapsto .^+$
- $\{n\}$ represents exactly n occurrences of the preceding RE
  - e.g., for $A = \{a,b\}$, RE = $aaa(a|b)(a|b) \mapsto a\{3\}.\{2\}$
- $\{m,\}$ represents at least m occurrences of the preceding RE
  - e.g., for $A = \{a,b\}$, RE = $aaa(a*)b \mapsto a\{3,\}b$
- $\{m,n\}$ represents at least m, and at most n occurrences of the preceding RE
  - e.g., for $A = \{a,b\}$, RE = $(a|aa|aaa)b \mapsto a\{1,3\}b$

Short RE notations (cont.)

For convenience…

SHORT RE NOTATIONS (CONT.)

For convenience…

- $[a_1 a_2 \ldots a_n]$ represents choice between all a_i
  - e.g., $A = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$, the RE $(0|1|2|3|4|5|6|7|8|9) \mapsto [0123456789]$ (note it is different from 0123456789 which is concatenation)

SHORT RE NOTATIONS (CONT.)

For convenience...

- $[a_1 a_2 ... a_n]$ represents choice between all a_i
  - e.g., $A = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$, the RE $(0|1|2|3|4|5|6|7|8|9) \mapsto [0123456789]$ (note it is different from 0123456789 which is concatenation)
- $[a_1 - a_n]$ represents choice between a range. Note that it only makes sense for ranges
  - e.g., $A = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$, the RE $(0|1|2|3|4|5|6|7|8|9) \mapsto [0 - 9]$ (note it is different from $0| - |9$, $-$ has a special meaning inside brackets, useful for REs like $[a - z]$ or $[a - zA - Z]$, the later combined with the previous short notation)

SHORT RE NOTATIONS (CONT.)

For convenience…

- $[a_1 a_2 ... a_n]$ represents choice between all a_i
  - e.g., $A = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$, the RE $(0|1|2|3|4|5|6|7|8|9) \mapsto [0123456789]$ (note it is different from 0123456789 which is concatenation)
- $[a_1 - a_n]$ represents choice between a range. Note that it only makes sense for ranges
  - e.g., $A = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$, the RE $(0|1|2|3|4|5|6|7|8|9) \mapsto [0 - 9]$ (note it is different from $0| - |9$, $-$ has a special meaning inside brackets, useful for REs like $[a - z]$ or $[a - zA - Z]$, the later combined with the previous short notation)
- $[\hat{}r]$ negation of r
  - e.g., $A = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$, the RE $(1|2|3|4|5|6|7|8|9) \mapsto [\hat{}0]$ (note it is different from $0|\hat{}$ by using the "short OR")

Short RE notations – Final Remarks

In practice (not formal)...

Short RE notations – Final Remarks

In practice (not formal)...

- The alphabet is usually omitted and assumed from the symbols appearing in the RE
  - e.g., the RE $[0-9]$ is assumed to have the alphabet $A = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$

Short RE notations – Final Remarks

In practice (not formal)…

- The alphabet is usually omitted and assumed from the symbols appearing in the RE
  - e.g., the RE $[0-9]$ is assumed to have the alphabet $A = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$
- REs are used for **pattern matching**, which means if a part of a string can be generated by the RE (or accepted by the equivalent automaton, more on this later), the string matches the pattern described by the RE. This leads to some notations, as: ^ to denote the beginning of the string, and \$ for the end of the string.
  - e.g., the RE $\hat{}a.*b\$$ means anything that starts with a, and finishes with b

Exercises with short RE notations

What language the RE describes?

Exercises with short RE notations

What language the RE describes?

- $[0-9]\{4\}-(0[1-9]|1[0-2])-(0[1-9]|(1|2)[0-9]|3(0|1)?$

Exercises with short RE notations

What language the RE describes?

- $[0-9]\{4\}-(0[1-9]|1[0-2])-(0[1-9]|(1|2)[0-9]|3(0|1))$?
  - Dates in the format yyyy-mm-dd (not well specified since it accepts 2016-02-31, for example)

EXERCISES WITH SHORT RE NOTATIONS

What language the RE describes?

- $[0-9]\{4\}-(0[1-9]|1[0-2])-(0[1-9]|(1|2)[0-9]|3(0|1)?$
  - Dates in the format yyyy-mm-dd (not well specified since it accepts 2016-02-31, for example)
- $bb|[\hat{}(2b)]$

Exercises with short RE notations

What language the RE describes?

- $[0-9]\{4\}-(0[1-9]|1[0-2])-(0[1-9]|(1|2)[0-9]|3(0|1)?$
  - Dates in the format yyyy-mm-dd (not well specified since it accepts 2016-02-31, for example)
- $bb|[\hat{}(2b)]$
  - Two 'b' or not '2b', that's the question... Actually, a bad example, it accepts anything that is not '2b', but it's a classical :)

Exercises with short RE notations

What language the RE describes?

- $[0-9]\{4\}-(0[1-9]|1[0-2])-(0[1-9]|(1|2)[0-9]|3(0|1)?$
  - Dates in the format yyyy-mm-dd (not well specified since it accepts 2016-02-31, for example)
- $bb|[\hat{}(2b)]$
  - Two 'b' or not '2b', that's the question… Actually, a bad example, it accepts anything that is not '2b', but it's a classical :)

Create the RE with short notations such that…

Exercises with short RE notations

What language the RE describes?

- $[0-9]\{4\}-(0[1-9]|1[0-2])-(0[1-9]|(1|2)[0-9]|3(0|1)?$
  - Dates in the format yyyy-mm-dd (not well specified since it accepts 2016-02-31, for example)
- $bb|[\hat{}(2b)]$
  - Two ‘b’ or not ‘2b’, that’s the question… Actually, a bad example, it accepts anything that is not ‘2b’, but it’s a classical :)

Create the RE with short notations such that…

- It generates non-limited floating point numbers (assume “.” can be specified as \.)

Exercises with short RE notations

What language the RE describes?

- $[0-9]\{4\}-(0[1-9]||1[0-2])-(0[1-9]||(1|2)[0-9]||3(0|1)?$
  - Dates in the format yyyy-mm-dd (not well specified since it accepts 2016-02-31, for example)
- $bb|[\hat{}(2b)]$
  - Two 'b' or not '2b', that's the question... Actually, a bad example, it accepts anything that is not '2b', but it's a classical :)

Create the RE with short notations such that...

- It generates non-limited floating point numbers (assume "." can be specified as \.)
  - $(-)?[0-9]^+(\backslash.[0-9]^+)?$

Converting REs to Non-deterministic Finite Automata (NFA)

Assumption: We can convert any RE r to a NFA with one initial state and one final state

Converting REs to Non-deterministic Finite Automata (NFA)

Assumption: We can convert any RE r to a NFA with one initial state and one final state

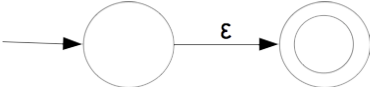
- Then, let's show it for each RE construction

Converting REs to Non-deterministic Finite Automata (NFA)

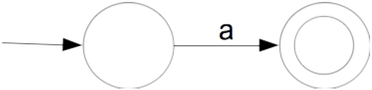
Assumption: We can convert any RE r to a NFA with one initial state and one final state

- Then, let's show it for each RE construction

ε

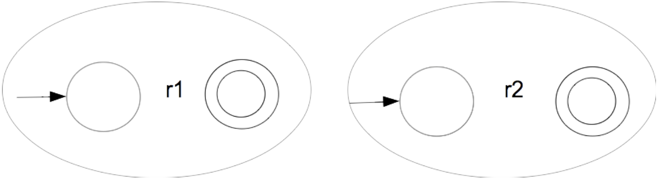


$a \in A$



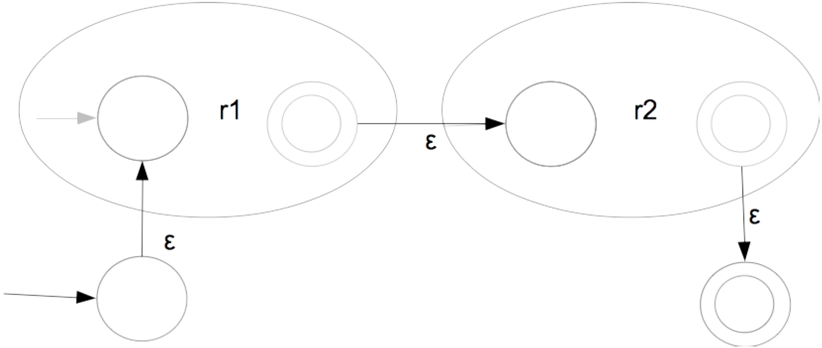
Converting REs to NFA (cont.)

Sequence, $r1r2$



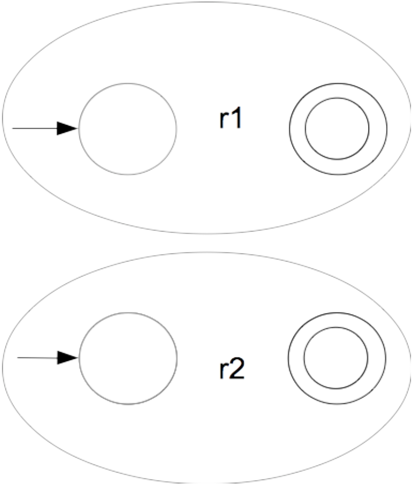
CONVERTING REs TO NFA (CONT.)

Sequence, $r1r2$



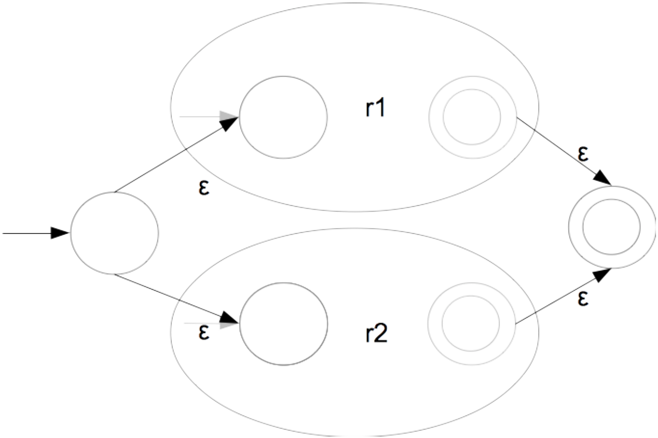
Converting REs to NFA (cont. 2)

Choice, $r1|r2$



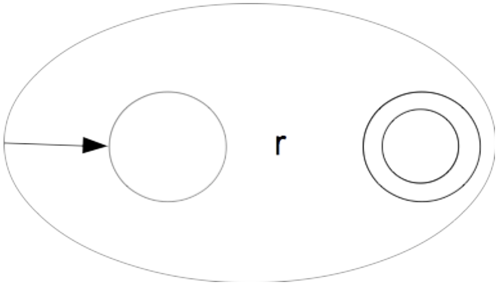
CONVERTING REs TO NFA (CONT. 2)

Choice, $r1|r2$



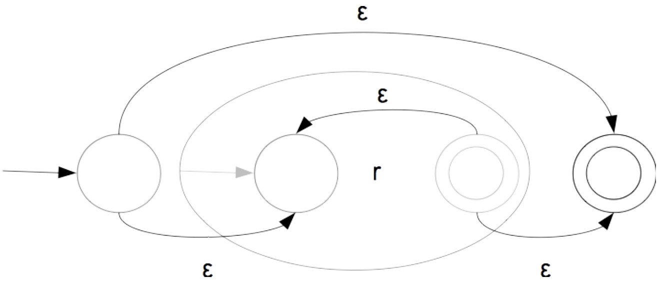
Converting REs to NFA (cont. t3)

Kleene Star, r^*



Converting REs to NFA (cont. t3)

Kleene Star, r^*



FROM NFA TO DFA

Algorithm (in case you forgot (:)

From NFA to DFA

Algorithm (in case you forgot (:)

- The DFA has a state for a subset of NFA states

FROM NFA TO DFA

Algorithm (in case you forgot (:)

- The DFA has a state for a subset of NFA states
  - The initial DFA state is the subset of NFA states that can be reached by the initial NFA state by following ε transitions

From NFA to DFA

Algorithm (in case you forgot (:)

- The DFA has a state for a subset of NFA states
  - The initial DFA state is the subset of NFA states that can be reached by the initial NFA state by following ε transitions
  - The DFA state is considered final if it contains a NFA state

From NFA to DFA

Algorithm (in case you forgot (:)

- The DFA has a state for a subset of NFA states
  - The initial DFA state is the subset of NFA states that can be reached by the initial NFA state by following ε transitions
  - The DFA state is considered final if it contains a NFA state
- Starting from the initially computer DFA state, for all D of the DFA, and $a \in A$:

FROM NFA TO DFA

Algorithm (in case you forgot (:)

- The DFA has a state for a subset of NFA states
  - The initial DFA state is the subset of NFA states that can be reached by the initial NFA state by following ε transitions
  - The DFA state is considered final if it contains a NFA state
- Starting from the initially computer DFA state, for all D of the DFA, and $a \in A$:
  - Set $S = \emptyset$

From NFA to DFA

Algorithm (in case you forgot (:)

- The DFA has a state for a subset of NFA states
  - The initial DFA state is the subset of NFA states that can be reached by the initial NFA state by following ε transitions
  - The DFA state is considered final if it contains a NFA state
- Starting from the initially computer DFA state, for all D of the DFA, and $a \in A$:
  - Set $S = \emptyset$
  - Set N to the NFA original states of D

FROM NFA TO DFA

Algorithm (in case you forgot (:)

- The DFA has a state for a subset of NFA states
  - The initial DFA state is the subset of NFA states that can be reached by the initial NFA state by following ε transitions
  - The DFA state is considered final if it contains a NFA state
- Starting from the initially computer DFA state, for all D of the DFA, and $a \in A$:
  - Set $S = \emptyset$
  - Set N to the NFA original states of D
    - Compute the set N' that the NFA might be after matching a

From NFA to DFA

Algorithm (in case you forgot (:)

- The DFA has a state for a subset of NFA states
  - The initial DFA state is the subset of NFA states that can be reached by the initial NFA state by following ε transitions
  - The DFA state is considered final if it contains a NFA state
- Starting from the initially computer DFA state, for all D of the DFA, and $a \in A$:
  - Set $S = \emptyset$
  - Set N to the NFA original states of D
    - Compute the set N' that the NFA might be after matching a
    - Set $S = S \cup N'$

From NFA to DFA

Algorithm (in case you forgot (:)

- The DFA has a state for a subset of NFA states
  - The initial DFA state is the subset of NFA states that can be reached by the initial NFA state by following ε transitions
  - The DFA state is considered final if it contains a NFA state
- Starting from the initially computer DFA state, for all D of the DFA, and $a \in A$:
  - Set $S = \emptyset$
  - Set N to the NFA original states of D
    - Compute the set N' that the NFA might be after matching a
    - Set $S = S \cup N'$
  - if S is nonempty, there is a transition with a from D to the DFA state with S NFA states in it.

From NFA to DFA

Algorithm (in case you forgot (:)

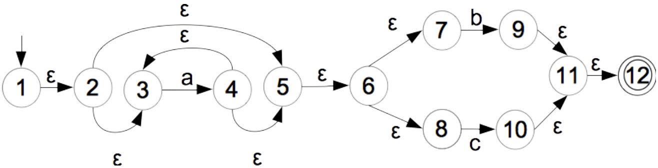
- The DFA has a state for a subset of NFA states
  - The initial DFA state is the subset of NFA states that can be reached by the initial NFA state by following ε transitions
  - The DFA state is considered final if it contains a NFA state
- Starting from the initially computer DFA state, for all D of the DFA, and $a \in A$:
  - Set $S = \emptyset$
  - Set N to the NFA original states of D
    - Compute the set N' that the NFA might be after matching a
    - Set $S = S \cup N'$
  - if S is nonempty, there is a transition with a from D to the DFA state with S NFA states in it.
  - Otherwise, there is no transition

NFA TO DFA EXAMPLE

Consider the NFA obtained from $a^*(b|c)$

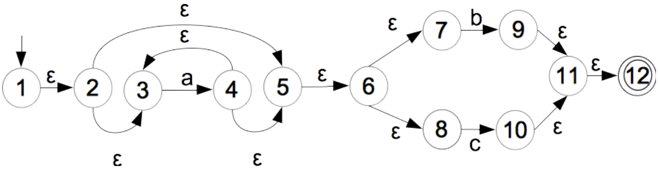
NFA TO DFA EXAMPLE

Consider the NFA obtained from $a^*(b|c)$



NFA TO DFA EXAMPLE

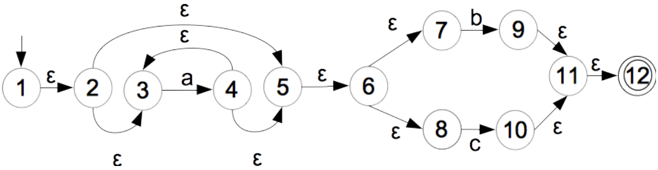
Consider the NFA obtained from $a^*(b|c)$



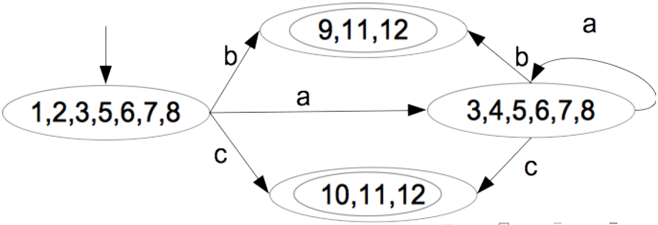
The equivalent DFA

NFA to DFA example

Consider the NFA obtained from $a^*(b|c)$



The equivalent DFA

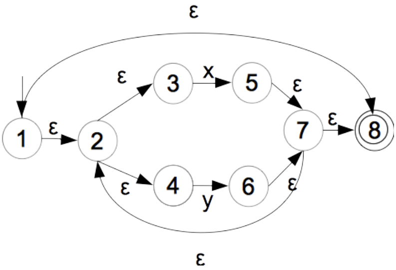


(simple) NFA to DFA exercise

Consider the NFA

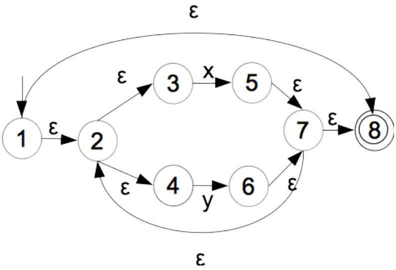
(simple) NFA to DFA exercise

Consider the NFA



(simple) NFA to DFA exercise

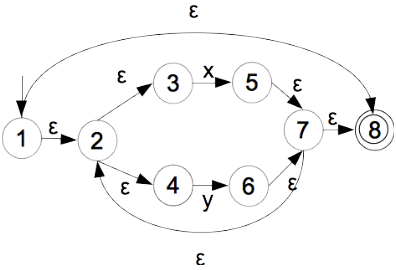
Consider the NFA



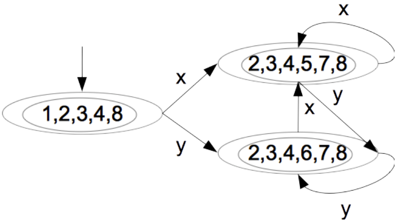
The equivalent DFA $(x|y)^*$

(SIMPLE) NFA TO DFA EXERCISE

Consider the NFA



The equivalent DFA $(x|y)^*$



Lexical Analysis — Final Remarks

Lexical Analysis — Final Remarks

- Non-determinism can be eliminated, this is important for the implementations recognizing the inputs

Lexical Analysis — Final Remarks

- Non-determinism can be eliminated, this is important for the implementations recognizing the inputs
- The important REs for the source code include keywords (if, while, for, etc.), numbers (floats, ints, etc.) identifiers (variable names, etc.), each category is specified with a RE.

Lexical Analysis — Final Remarks

- Non-determinism can be eliminated, this is important for the implementations recognizing the inputs
- The important REs for the source code include keywords (if, while, for, etc.), numbers (floats, ints, etc.) identifiers (variable names, etc.), each category is specified with a RE.
- When the source code is passed to the lexer (the program performing the lexical analysis), it returns a **tokenized** string. For example:

```
if(j + 2) return 10;
```

Yields:

```
keyword(if) l_paren identifier(j) add_op num(2) r_paren keyword(return) num(10) sc
```

Back to the big picture...

We can specify the words of a language

Back to the big picture...

We can specify the words of a language

- Are RE applicable for specifying the structure of the sentence?

Back to the big picture...

We can specify the words of a language

- Are RE applicable for specifying the structure of the sentence?
  - No. OK, why?

BACK TO THE BIG PICTURE...

We can specify the words of a language

- Are RE applicable for specifying the structure of the sentence?
  - No. OK, why?
  - Nested constructs required, nested if-else, nested expressions, etc., a simple example, build a RE that matches the same number of left parenthesis to the right one. For example: ((())), (), (((((())))))

Back to the big picture...

We can specify the words of a language

- Are RE applicable for specifying the structure of the sentence?
  - No. OK, why?
  - Nested constructs required, nested if-else, nested expressions, etc., a simple example, build a RE that matches the same number of left parenthesis to the right one. For example: ((())), (), ((((()))))

Solution:

Back to the big picture...

We can specify the words of a language

- Are RE applicable for specifying the structure of the sentence?
  - No. OK, why?
  - Nested constructs required, nested if-else, nested expressions, etc., a simple example, build a RE that matches the same number of left parenthesis to the right one. For example: ((())), (), (((((())))))

Solution:

Syntactical analysis with a **Context-Free Grammar (CFG)**

Syntactical Analysis

using Context-Free Grammars (CFG)

Before we dive into CFG...

Before we dive into CFG...

- We want to **recognize** the source code as a structured object

Before we dive into CFG...

- We want to **recognize** the source code as a structured object
- As a DFA (or NFA) recognizes a regular language, a Push Down Automaton (PDA) recognizes a non-regular language

Before we dive into CFG...

- We want to **recognize** the source code as a structured object
- As a DFA (or NFA) recognizes a regular language, a Push Down Automaton (PDA) recognizes a non-regular language
- To describe a push down automaton, it can be convenient to do it through a CFG, as it is more convenient to describe a DFA with a very elegant and simple RE (instead of state transition, etc)

CONTEXT-FREE GRAMMAR (CFG)

A CFG is a 4-tuple (T, N, S, P) such that

Context-Free Grammar (CFG)

A CFG is a 4-tuple (T, N, S, P) such that

- T is a finite set of terminal symbols (constants or tokenized strings described by regular expressions)

Context-Free Grammar (CFG)

A CFG is a 4-tuple (T, N, S, P) such that

- T is a finite set of terminal symbols (constants or tokenized strings described by regular expressions)
- N is a finite set of non-terminal symbols

CONTEXT-FREE GRAMMAR (CFG)

A CFG is a 4-tuple (T, N, S, P) such that

- T is a finite set of terminal symbols (constants or tokenized strings described by regular expressions)
- N is a finite set of non-terminal symbols
- $S \in N$ is a start non-terminal symbol

Context-Free Grammar (CFG)

A CFG is a 4-tuple (T, N, S, P) such that

- T is a finite set of terminal symbols (constants or tokenized strings described by regular expressions)
- N is a finite set of non-terminal symbols
- $S \in N$ is a start non-terminal symbol
- P is a finite set of production rules, a relation from N to $(T \cup N)^*$. The symbol * denotes the Kleene star (ε is also allowed!)

Context-Free Grammar (CFG)

A CFG is a 4-tuple (T, N, S, P) such that

- T is a finite set of terminal symbols (constants or tokenized strings described by regular expressions)
- N is a finite set of non-terminal symbols
- $S \in N$ is a start non-terminal symbol
- P is a finite set of production rules, a relation from N to $(T \cup N)^*$. The symbol * denotes the Kleene star (ε is also allowed!)
- $p \in P$ has the notation $N \mapsto \alpha$, where $\alpha \in (T \cup N)^*$. This means, in the left hand side of the production rule there is only one non-terminal (which makes it context-free), and on the right hand a sequence of terminal and non-terminal symbols

CFG Example

Consider the following REs:

CFG Example

Consider the following REs:

- $op = +|-|/|*$
- $int = [0-9]^{+}$
- $opar = ($
- $cpar =)$

CFG Example

Consider the following REs:

- $op = +|-|/|*$
- $int = [0-9]^+$
- $opar = ($
- $cpar =)$

Consider the following production rules

CFG Example

Consider the following REs:

- $op = +|-|/|*$
- $int = [0-9]^+$
- $opar = ($
- $cpar =)$

Consider the following production rules

- $Start \mapsto Expr$
- $Expr \mapsto Expr\ op\ Expr$
- $Expr \mapsto int$
- $Expr \mapsto opar\ Expr\ cpar$

Using the CFG to generate “a sentence“

the (informal) algorithm

Using the CFG to generate “a sentence“

the (informal) algorithm

- Begin with the start symbol (Start in the previous CFG)

Using the CFG to generate “a sentence“

the (informal) algorithm

- Begin with the start symbol (Start in the previous CFG)
- Loop until non-terminal symbols appear

Using the CFG to generate "a sentence"

the (informal) algorithm

- Begin with the start symbol (Start in the previous CFG)
- Loop until non-terminal symbols appear
  - Choose a non-terminal from the current Right Hand Side (RHS) of the production rule

USING THE CFG TO GENERATE “A SENTENCE“

the (informal) algorithm

- Begin with the start symbol (Start in the previous CFG)
- Loop until non-terminal symbols appear
  - Choose a non-terminal from the current Right Hand Side (RHS) of the production rule
  - Choose a production rule from the previously selected non-terminal symbol

Using the CFG to generate "a sentence"

the (informal) algorithm

- Begin with the start symbol (Start in the previous CFG)
- Loop until non-terminal symbols appear
  - Choose a non-terminal from the current Right Hand Side (RHS) of the production rule
  - Choose a production rule from the previously selected non-terminal symbol
  - Replace the chosen non-terminal symbol by substituting Left Hand Side (LHS) non-terminal symbol for the RHS sequence in the chosen production rule

USING THE CFG TO GENERATE "A SENTENCE"

the (informal) algorithm

- Begin with the start symbol (Start in the previous CFG)
- Loop until non-terminal symbols appear
  - Choose a non-terminal from the current Right Hand Side (RHS) of the production rule
  - Choose a production rule from the previously selected non-terminal symbol
  - Replace the chosen non-terminal symbol by substituting Left Hand Side (LHS) non-terminal symbol for the RHS sequence in the chosen production rule
- Generate the necessary strings from the token REs

Using the CFG to generate "a sentence"

the (informal) algorithm

- Begin with the start symbol (Start in the previous CFG)
- Loop until non-terminal symbols appear
  - Choose a non-terminal from the current Right Hand Side (RHS) of the production rule
  - Choose a production rule from the previously selected non-terminal symbol
  - Replace the chosen non-terminal symbol by substituting Left Hand Side (LHS) non-terminal symbol for the RHS sequence in the chosen production rule
- Generate the necessary strings from the token REs

The obtained string belongs the CFG language!

Using the CFG to generate "a sentence"

the (informal) algorithm

- Begin with the start symbol (Start in the previous CFG)
- Loop until non-terminal symbols appear
  - Choose a non-terminal from the current Right Hand Side (RHS) of the production rule
  - Choose a production rule from the previously selected non-terminal symbol
  - Replace the chosen non-terminal symbol by substituting Left Hand Side (LHS) non-terminal symbol for the RHS sequence in the chosen production rule
- Generate the necessary strings from the token REs

The obtained string belongs the CFG language!

- Important note: different choices produce different "sentences"

Example CFG, generating a string of the CFG

- $op = +| - |/|*$
- $int = [0 - 9]^+$
- $opar = ($
- $cpar =)$

1. $Start \mapsto Expr$
2. $Expr \mapsto Expr\ op\ Expr$
3. $Expr \mapsto int$
4. $Expr \mapsto opar\ Expr\ cpar$

Example CFG, generating a string of the CFG

Let's derive…

- $op = +|-|/|*$
- $int = [0-9]^+$
- $opar = ($
- $cpar =)$

1. $Start \mapsto Expr$
2. $Expr \mapsto Expr\ op\ Expr$
3. $Expr \mapsto int$
4. $Expr \mapsto opar\ Expr\ cpar$

Example CFG, generating a string of the CFG

Let's derive...

- *Start*

- $op = +|-|/|*$
- $int = [0-9]^+$
- $opar = ($
- $cpar =)$

1. $Start \mapsto Expr$
2. $Expr \mapsto Expr\ op\ Expr$
3. $Expr \mapsto int$
4. $Expr \mapsto opar\ Expr\ cpar$

EXAMPLE CFG, GENERATING A STRING OF THE CFG

- $op = +|-|/|*$
- $int = [0-9]^+$
- $opar = ($
- $cpar =)$

Let's derive...

- *Start*
- *Expr* (1)

1. $Start \mapsto Expr$
2. $Expr \mapsto Expr\ op\ Expr$
3. $Expr \mapsto int$
4. $Expr \mapsto opar\ Expr\ cpar$

EXAMPLE CFG, GENERATING A STRING OF THE CFG

- $op = +|-|/|*$
- $int = [0-9]^+$
- $opar = ($
- $cpar =)$

Let's derive...

- $Start$
- $Expr$ (1)
- $Expr\ op\ Expr$ (2)

1. $Start \mapsto Expr$
2. $Expr \mapsto Expr\ op\ Expr$
3. $Expr \mapsto int$
4. $Expr \mapsto opar\ Expr\ cpar$

Example CFG, generating a string of the CFG

- $op = +|-|/|*$
- $int = [0-9]^+$
- $opar = ($
- $cpar =)$

Let's derive...

- $Start$
- $Expr$ (1)
- $Expr\ op\ Expr$ (2)
- $int\ op\ Expr$ (3)

1. $Start \mapsto Expr$
2. $Expr \mapsto Expr\ op\ Expr$
3. $Expr \mapsto int$
4. $Expr \mapsto opar\ Expr\ cpar$

Example CFG, generating a string of the CFG

Let's derive...

- $op = +|-|/|*$
- $int = [0-9]^+$
- $opar = ($
- $cpar =)$

- *Start*
- *Expr* (1)
- *Expr op Expr* (2)
- *int op Expr* (3)
- *int op opar Expr cpar* (4)

1. $Start \mapsto Expr$
2. $Expr \mapsto Expr\ op\ Expr$
3. $Expr \mapsto int$
4. $Expr \mapsto opar\ Expr\ cpar$

EXAMPLE CFG, GENERATING A STRING OF THE CFG

- $op = +|-|/|*$
- $int = [0-9]^+$
- $opar = ($
- $cpar =)$

1. $Start \mapsto Expr$
2. $Expr \mapsto Expr\ op\ Expr$
3. $Expr \mapsto int$
4. $Expr \mapsto opar\ Expr\ cpar$

Let's derive...

- $Start$
- $Expr$ (1)
- $Expr\ op\ Expr$ (2)
- $int\ op\ Expr$ (3)
- $int\ op\ opar\ Expr\ cpar$ (4)
- $int\ op\ opar\ Expr\ op\ Expr\ cpar$ (2)

Example CFG, generating a string of the CFG

- $op = +|-|/|*$
- $int = [0-9]^+$
- $opar = ($
- $cpar =)$

1. $Start \mapsto Expr$
2. $Expr \mapsto Expr\ op\ Expr$
3. $Expr \mapsto int$
4. $Expr \mapsto opar\ Expr\ cpar$

Let's derive…

- $Start$
- $Expr$ (1)
- $Expr\ op\ Expr$ (2)
- $int\ op\ Expr$ (3)
- $int\ op\ opar\ Expr\ cpar$ (4)
- $int\ op\ opar\ Expr\ op\ Expr\ cpar$ (2)
- $int\ op\ opar\ int\ op\ Expr\ cpar$ (3)

Example CFG, generating a string of the CFG

- $op = +|-|/|*$
- $int = [0-9]^+$
- $opar = ($
- $cpar =)$

1. $Start \mapsto Expr$
2. $Expr \mapsto Expr\ op\ Expr$
3. $Expr \mapsto int$
4. $Expr \mapsto opar\ Expr\ cpar$

Let's derive…

- $Start$
- $Expr$ (1)
- $Expr\ op\ Expr$ (2)
- $int\ op\ Expr$ (3)
- $int\ op\ opar\ Expr\ cpar$ (4)
- $int\ op\ opar\ Expr\ op\ Expr\ cpar$ (2)
- $int\ op\ opar\ int\ op\ Expr\ cpar$ (3)
- $int\ op\ opar\ int\ op\ int\ cpar$ (3)

Example CFG, generating a string of the CFG

- $op = +|-|/|*$
- $int = [0-9]^+$
- $opar = ($
- $cpar =)$

1. $Start \mapsto Expr$
2. $Expr \mapsto Expr\ op\ Expr$
3. $Expr \mapsto int$
4. $Expr \mapsto opar\ Expr\ cpar$

Let's derive...

- $Start$
- $Expr$ (1)
- $Expr\ op\ Expr$ (2)
- $int\ op\ Expr$ (3)
- $int\ op\ opar\ Expr\ cpar$ (4)
- $int\ op\ opar\ Expr\ op\ Expr\ cpar$ (2)
- $int\ op\ opar\ int\ op\ Expr\ cpar$ (3)
- $int\ op\ opar\ int\ op\ int\ cpar$ (3)
- 10 / (7 – 5)

Example CFG, generating a string of the CFG

- $op = +|-|/|*$
- $int = [0-9]^+$
- $opar = ($
- $cpar =)$

1. $Start \mapsto Expr$
2. $Expr \mapsto Expr\ op\ Expr$
3. $Expr \mapsto int$
4. $Expr \mapsto opar\ Expr\ cpar$

Let's derive…

- $Start$
- $Expr$ (1)
- $Expr\ op\ Expr$ (2)
- $int\ op\ Expr$ (3)
- $int\ op\ opar\ Expr\ cpar$ (4)
- $int\ op\ opar\ Expr\ op\ Expr\ cpar$ (2)
- $int\ op\ opar\ int\ op\ Expr\ cpar$ (3)
- $int\ op\ opar\ int\ op\ int\ cpar$ (3)
- 10 / (7 – 5)

Please, derive one yourselves

Constructing a CFG

Let's assume we want to describe the non-regular language for a for HTML tables (looks like this):

Constructing a CFG

Let's assume we want to describe the non-regular language for a for HTML tables (looks like this):

```
<table style="width:100%">
  <tr>
    <th>Name</th>
    <th>Lastname</th>
    <th>Score</th>
  </tr>
  <tr>
    <td>Eve</td>
    <td>Jackson-Renard</td>
    <td>94</td>
  </tr>
  <tr>
    <td>Vladimir</td>
    <td>López</td>
    <td>80</td>
  </tr>
</table>
```

Constructing a CFG

Let's assume we want to describe the non-regular language for a for HTML tables (looks like this):

```
<table style="width:100%">
  <tr>
    <th>Name</th>
    <th>Lastname</th>
    <th>Score</th>
  </tr>
  <tr>
    <td>Eve</td>
    <td>Jackson-Renard</td>
    <td>94</td>
  </tr>
  <tr>
    <td>Vladimir</td>
    <td>López</td>
    <td>80</td>
  </tr>
</table>
```

Produces (close-enough):

Name	Lastname	Score
Eve	Jackson-Renard	94
Vladimir	López	80

Constructing a CFG (cont.)

Let's assume we want to describe the non-regular language for a for HTML tables (looks like this):

```
<table style="width:100%">
  <tr>
    <th>Name</th>
    <th>Lastname</th>
    <th>Score</th>
  </tr>
  <tr>
    <td>Eve</td>
    <td>Jackson-Renard</td>
    <td>94</td>
  </tr>
  <tr>
    <td>Vladimir</td>
    <td>Lopez</td>
    <td>80</td>
  </tr>
</table>
```

Constructing a CFG (cont.)

Let's assume we want to describe the non-regular language for a for HTML tables (looks like this):

```
<table style="width:100%">
  <tr>
    <th>Name</th>
    <th>Lastname</th>
    <th>Score</th>
  </tr>
  <tr>
    <td>Eve</td>
    <td>Jackson-Renard</td>
    <td>94</td>
  </tr>
  <tr>
    <td>Vladimir</td>
    <td>Lopez</td>
    <td>80</td>
  </tr>
</table>
```

REs:

- table attributes (class, border, etc.)

Constructing a CFG (cont.)

Let's assume we want to describe the non-regular language for a for HTML tables (looks like this):

```
<table style="width:100%">
  <tr>
    <th>Name</th>
    <th>Lastname</th>
    <th>Score</th>
  </tr>
  <tr>
    <td>Eve</td>
    <td>Jackson-Renard</td>
    <td>94</td>
  </tr>
  <tr>
    <td>Vladimir</td>
    <td>Lopez</td>
    <td>80</td>
  </tr>
</table>
```

REs:

- $attributes = style|class|border|\ldots$

Constructing a CFG (cont.)

Let's assume we want to describe the non-regular language for a for HTML tables (looks like this):

```
<table style="width:100%">
  <tr>
    <th>Name</th>
    <th>Lastname</th>
    <th>Score</th>
  </tr>
  <tr>
    <td>Eve</td>
    <td>Jackson-Renard</td>
    <td>94</td>
  </tr>
  <tr>
    <td>Vladimir</td>
    <td>Lopez</td>
    <td>80</td>
  </tr>
</table>
```

REs:

- $attributes = style|class|border|\ldots$
- values of the attributes

Constructing a CFG (cont.)

Let's assume we want to describe the non-regular language for a for HTML tables (looks like this):

```
<table style="width:100%">
  <tr>
    <th>Name</th>
    <th>Lastname</th>
    <th>Score</th>
  </tr>
  <tr>
    <td>Eve</td>
    <td>Jackson-Renard</td>
    <td>94</td>
  </tr>
  <tr>
    <td>Vladimir</td>
    <td>Lopez</td>
    <td>80</td>
  </tr>
</table>
```

REs:

- $attributes = style|class|border|\ldots$
- $values = \backslash''[\hat{}\ \backslash\ '']^*\ \backslash\ ''$

Constructing a CFG (cont.)

Let's assume we want to describe the non-regular language for a for HTML tables (looks like this):

```
<table style="width:100%">
  <tr>
    <th>Name</th>
    <th>Lastname</th>
    <th>Score</th>
  </tr>
  <tr>
    <td>Eve</td>
    <td>Jackson-Renard</td>
    <td>94</td>
  </tr>
  <tr>
    <td>Vladimir</td>
    <td>Lopez</td>
    <td>80</td>
  </tr>
</table>
```

REs:

- $attributes = style|class|border|\ldots$
- $values = \backslash''[\hat{}\ \backslash'']^* \backslash''$
- data

Constructing a CFG (cont.)

Let's assume we want to describe the non-regular language for a for HTML tables (looks like this):

```
<table style="width:100%">
  <tr>
    <th>Name</th>
    <th>Lastname</th>
    <th>Score</th>
  </tr>
  <tr>
    <td>Eve</td>
    <td>Jackson-Renard</td>
    <td>94</td>
  </tr>
  <tr>
    <td>Vladimir</td>
    <td>Lopez</td>
    <td>80</td>
  </tr>
</table>
```

REs:

- $attributes = style|class|border|\ldots$
- $values = \backslash''[\hat{}\ \backslash\ '']^*\ \backslash\ ''$
- $data = [a - zA - Z0 - 9]^*\ldots$

Constructing a CFG (cont.++)

Let's assume we want to describe the non-regular language for a for HTML tables (looks like this):

```
<table style="width:100%">
  <tr>
    <th>Name</th>
    <th>Lastname</th>
    <th>Score</th>
  </tr>
  <tr>
    <td>Eve</td>
    <td>Jackson-Renard</td>
    <td>94</td>
  </tr>
  <tr>
    <td>Vladimir</td>
    <td>Lopez</td>
    <td>80</td>
  </tr>
</table>
```

Constructing a CFG (cont.++)

Let's assume we want to describe the non-regular language for a for HTML tables (looks like this):

```
<table style="width:100%">
  <tr>
    <th>Name</th>
    <th>Lastname</th>
    <th>Score</th>
  </tr>
  <tr>
    <td>Eve</td>
    <td>Jackson-Renard</td>
    <td>94</td>
  </tr>
  <tr>
    <td>Vladimir</td>
    <td>Lopez</td>
    <td>80</td>
  </tr>
</table>
```

Production rules:

Constructing a CFG (cont.++)

Let's assume we want to describe the non-regular language for a for HTML tables (looks like this):

```
<table style="width:100%">
  <tr>
    <th>Name</th>
    <th>Lastname</th>
    <th>Score</th>
  </tr>
  <tr>
    <td>Eve</td>
    <td>Jackson-Renard</td>
    <td>94</td>
  </tr>
  <tr>
    <td>Vladimir</td>
    <td>Lopez</td>
    <td>80</td>
  </tr>
</table>
```

Production rules:

1. $S \mapsto < table\ A1 > S2 < /table >$

CONSTRUCTING A CFG (CONT.++)

Let's assume we want to describe the non-regular language for a for HTML tables (looks like this):

```
<table style="width:100%">
  <tr>
    <th>Name</th>
    <th>Lastname</th>
    <th>Score</th>
  </tr>
  <tr>
    <td>Eve</td>
    <td>Jackson-Renard</td>
    <td>94</td>
  </tr>
  <tr>
    <td>Vladimir</td>
    <td>Lopez</td>
    <td>80</td>
  </tr>
</table>
```

Production rules:

1. $S \mapsto < table\ A1 > S2 < /table >$
2. $S2 \mapsto < tr > S3 < /tr >$

CONSTRUCTING A CFG (CONT.++)

Let's assume we want to describe the non-regular language for a for HTML tables (looks like this):

```
<table style="width:100%">
  <tr>
    <th>Name</th>
    <th>Lastname</th>
    <th>Score</th>
  </tr>
  <tr>
    <td>Eve</td>
    <td>Jackson-Renard</td>
    <td>94</td>
  </tr>
  <tr>
    <td>Vladimir</td>
    <td>Lopez</td>
    <td>80</td>
  </tr>
</table>
```

Production rules:

1. $S \mapsto < table\ A1 > S2 < /table >$
2. $S2 \mapsto < tr > S3 < /tr >$
3. $S2 \mapsto S2\ S2$

Constructing a CFG (cont.++)

Let's assume we want to describe the non-regular language for a for HTML tables (looks like this):

```
<table style="width:100%">
  <tr>
    <th>Name</th>
    <th>Lastname</th>
    <th>Score</th>
  </tr>
  <tr>
    <td>Eve</td>
    <td>Jackson-Renard</td>
    <td>94</td>
  </tr>
  <tr>
    <td>Vladimir</td>
    <td>Lopez</td>
    <td>80</td>
  </tr>
</table>
```

Production rules:

1. $S \mapsto < table\ A1 > S2 < /table >$
2. $S2 \mapsto < tr > S3 < /tr >$
3. $S2 \mapsto S2\ S2$
4. $S3 \mapsto < th > S4 < /th >$

Constructing a CFG (cont.++)

Let's assume we want to describe the non-regular language for a for HTML tables (looks like this):

```
<table style="width:100%">
  <tr>
    <th>Name</th>
    <th>Lastname</th>
    <th>Score</th>
  </tr>
  <tr>
    <td>Eve</td>
    <td>Jackson-Renard</td>
    <td>94</td>
  </tr>
  <tr>
    <td>Vladimir</td>
    <td>Lopez</td>
    <td>80</td>
  </tr>
</table>
```

Production rules:

1. $S \mapsto < table\ A1 > S2 < /table >$
2. $S2 \mapsto < tr > S3 < /tr >$
3. $S2 \mapsto S2\ S2$
4. $S3 \mapsto < th > S4 < /th >$
5. $S3 \mapsto < td > S4 < /td >$

Constructing a CFG (cont.++)

Let's assume we want to describe the non-regular language for a for HTML tables (looks like this):

```
<table style="width:100%">
  <tr>
    <th>Name</th>
    <th>Lastname</th>
    <th>Score</th>
  </tr>
  <tr>
    <td>Eve</td>
    <td>Jackson-Renard</td>
    <td>94</td>
  </tr>
  <tr>
    <td>Vladimir</td>
    <td>Lopez</td>
    <td>80</td>
  </tr>
</table>
```

Production rules:

1. $S \mapsto < table\ A1 > S2 < /table >$
2. $S2 \mapsto < tr > S3 < /tr >$
3. $S2 \mapsto S2\ S2$
4. $S3 \mapsto < th > S4 < /th >$
5. $S3 \mapsto < td > S4 < /td >$
6. $S3 \mapsto S3\ S3$

Constructing a CFG (cont.++)

Let's assume we want to describe the non-regular language for a for HTML tables (looks like this):

```
<table style="width:100%">
  <tr>
    <th>Name</th>
    <th>Lastname</th>
    <th>Score</th>
  </tr>
  <tr>
    <td>Eve</td>
    <td>Jackson-Renard</td>
    <td>94</td>
  </tr>
  <tr>
    <td>Vladimir</td>
    <td>Lopez</td>
    <td>80</td>
  </tr>
</table>
```

Production rules:

1. $S \mapsto < table\ A1 > S2 < /table >$
2. $S2 \mapsto < tr > S3 < /tr >$
3. $S2 \mapsto S2\ S2$
4. $S3 \mapsto < th > S4 < /th >$
5. $S3 \mapsto < td > S4 < /td >$
6. $S3 \mapsto S3\ S3$
7. $S4 \mapsto data$

CONSTRUCTING A CFG (CONT.++)

Let's assume we want to describe the non-regular language for a for HTML tables (looks like this):

```
<table style="width:100%">
  <tr>
    <th>Name</th>
    <th>Lastname</th>
    <th>Score</th>
  </tr>
  <tr>
    <td>Eve</td>
    <td>Jackson-Renard</td>
    <td>94</td>
  </tr>
  <tr>
    <td>Vladimir</td>
    <td>Lopez</td>
    <td>80</td>
  </tr>
</table>
```

Production rules:

1. $S \mapsto < table\ A1 > S2 < /table >$
2. $S2 \mapsto < tr > S3 < /tr >$
3. $S2 \mapsto S2\ S2$
4. $S3 \mapsto < th > S4 < /th >$
5. $S3 \mapsto < td > S4 < /td >$
6. $S3 \mapsto S3\ S3$
7. $S4 \mapsto data$
8. $S4 \mapsto S2$ //(assume no table def needed)

Constructing a CFG (cont.++)

Let's assume we want to describe the non-regular language for a for HTML tables (looks like this):

```
<table style="width:100%">
  <tr>
    <th>Name</th>
    <th>Lastname</th>
    <th>Score</th>
  </tr>
  <tr>
    <td>Eve</td>
    <td>Jackson-Renard</td>
    <td>94</td>
  </tr>
  <tr>
    <td>Vladimir</td>
    <td>Lopez</td>
    <td>80</td>
  </tr>
</table>
```

Production rules:

1. $S \mapsto < table\ A1 > S2 < /table >$
2. $S2 \mapsto < tr > S3 < /tr >$
3. $S2 \mapsto S2\ S2$
4. $S3 \mapsto < th > S4 < /th >$
5. $S3 \mapsto < td > S4 < /td >$
6. $S3 \mapsto S3\ S3$
7. $S4 \mapsto data$
8. $S4 \mapsto S2$ //(assume no table def needed)
9. $A1 \mapsto attributes = values$

CONSTRUCTING A CFG (CONT.++)

Let's assume we want to describe the non-regular language for a for HTML tables (looks like this):

```
<table style="width:100%">
  <tr>
    <th>Name</th>
    <th>Lastname</th>
    <th>Score</th>
  </tr>
  <tr>
    <td>Eve</td>
    <td>Jackson-Renard</td>
    <td>94</td>
  </tr>
  <tr>
    <td>Vladimir</td>
    <td>Lopez</td>
    <td>80</td>
  </tr>
</table>
```

Production rules:

1. $S \mapsto < table\ A1 > S2 < /table >$
2. $S2 \mapsto < tr > S3 < /tr >$
3. $S2 \mapsto S2\ S2$
4. $S3 \mapsto < th > S4 < /th >$
5. $S3 \mapsto < td > S4 < /td >$
6. $S3 \mapsto S3\ S3$
7. $S4 \mapsto data$
8. $S4 \mapsto S2$ //(assume no table def needed)
9. $A1 \mapsto attributes = values$
10. $A1 \mapsto A1\ A1$

Exercises – Describing languages through CFG

Please, describe the grammar for the language:

Exercises – Describing languages through CFG

Please, describe the grammar for the language:

- Balanced parenthesis (at least one pair), e.g,(()())((()))(())

Exercises – Describing languages through CFG

Please, describe the grammar for the language:

- Balanced parenthesis (at least one pair), e.g,(()())((()))(())
  - $P \mapsto PP$
  - $P \mapsto (P)$
  - $P \mapsto ()$ //E-Z, right?

EXERCISES – DESCRIBING LANGUAGES THROUGH CFG

Please, describe the grammar for the language:

- Balanced parenthesis (at least one pair), e.g,(()())((()))(())
  - $P \mapsto PP$
  - $P \mapsto (P)$
  - $P \mapsto ()$ //E-Z, right?
- The non-regular language $\{a^m b^n a^m : m \geq 1, n \geq 0\}$, e.g. aaabaaa, abbba, aabbaa, aaaa

Exercises – Describing languages through CFG

Please, describe the grammar for the language:

- Balanced parenthesis (at least one pair), e.g,(()())((()))(())
  - $P \mapsto PP$
  - $P \mapsto (P)$
  - $P \mapsto ()$ //E-Z, right?
- The non-regular language $\{a^m b^n a^m : m \geq 1, n \geq 0\}$, e.g. aaabaaa, abbba, aabbaa, aaaa
  - $S \mapsto aSa|aS_1a$ // Note the "OR"(|) notation
  - $S_1 \mapsto b|bS_1|\varepsilon$ //E-Zer, right?

EXERCISES – DESCRIBING LANGUAGES THROUGH CFG

Please, describe the grammar for the language:

- Balanced parenthesis (at least one pair), e.g,(()())((()))(())
  - $P \mapsto PP$
  - $P \mapsto (P)$
  - $P \mapsto ()$ //E-Z, right?
- The non-regular language $\{a^m b^n a^m : m \geq 1, n \geq 0\}$, e.g. aaabaaa, abbba, aabbaa, aaaa
  - $S \mapsto aSa|aS_1a$ // Note the "OR"(|) notation
  - $S_1 \mapsto b|bS_1|\varepsilon$ //E-Zer, right?
- Last one, balanced brackets [] and (), including ε quite simple for you now…

Exercises – Describing languages through CFG

Please, describe the grammar for the language:

- Balanced parenthesis (at least one pair), e.g,(()())((()))(())
  - $P \mapsto PP$
  - $P \mapsto (P)$
  - $P \mapsto ()$ //E-Z, right?
- The non-regular language $\{a^m b^n a^m : m \geq 1, n \geq 0\}$, e.g. aaabaaa, abbba, aabbaa, aaaa
  - $S \mapsto aSa | aS_1 a$ // Note the "OR"(|) notation
  - $S_1 \mapsto b | bS_1 | \varepsilon$ //E-Zer, right?
- Last one, balanced brackets [] and (), including ε quite simple for you now…
  - $S \mapsto SS | (S) | [S] | () | [] | \varepsilon$

Grammars...

Every regular grammar is a CFG

Grammars...

Every regular grammar is a CFG

- The grammar generated by: $(a|b)^*$ can be generated by $S \mapsto aS|bS|\varepsilon$

Grammars...

Every regular grammar is a CFG

- The grammar generated by: $(a|b)^*$ can be generated by $S \mapsto aS|bS|\varepsilon$

Not every CFG is a regular grammar (canonical balanced parenthesis example)

Grammars...

Every regular grammar is a CFG

- The grammar generated by: $(a|b)^*$ can be generated by $S \mapsto aS|bS|\varepsilon$

Not every CFG is a regular grammar (canonical balanced parenthesis example)

A hierarchical power of grammars, and corresponding automata

GRAMMARS...

Every regular grammar is a CFG

- The grammar generated by: $(a|b)^*$ can be generated by $S \mapsto aS|bS|\varepsilon$

Not every CFG is a regular grammar (canonical balanced parenthesis example)

A hierarchical power of grammars, and corresponding automata

- Regular Grammars & Finite State Automata
- Context-Free Grammars & Push Down Automata
- Context-Sensitive Grammars & Turing Machines
- I mean, just in case you forgot :)

You are here ↓

What we know

You are here ↓

What we know

- How to describe context-free languages using CFGs + REs

You are here ↓

What we know

- How to describe context-free languages using CFGs + REs
- The generative vs. the recognition approach of grammars and automata

You are here ↓

What we know

- How to describe context-free languages using CFGs + REs
- The generative vs. the recognition approach of grammars and automata

What we do not know (yet)

You are here ↓

What we know

- How to describe context-free languages using CFGs + REs
- The generative vs. the recognition approach of grammars and automata

What we do not know (yet)

- How to obtain the structure to analyze the code (which was the final goal) :(

You are here ↓

What we know

- How to describe context-free languages using CFGs + REs
- The generative vs. the recognition approach of grammars and automata

What we do not know (yet)

- How to obtain the structure to analyze the code (which was the final goal) :(
- To obtain such structure, we focus on the grammar derivation, grammar ambiguities are harmful, and we will see how to correct them

You are here ↓

What we know

- How to describe context-free languages using CFGs + REs
- The generative vs. the recognition approach of grammars and automata

What we do not know (yet)

- How to obtain the structure to analyze the code (which was the final goal) :(
- To obtain such structure, we focus on the grammar derivation, grammar ambiguities are harmful, and we will see how to correct them
- But, do not worry, next Tuesday(?) you'll know...