

# CMSC 330: Organization of Programming Languages

---

## Theory of Regular Expressions

## Last Lecture

---

- ▶ Ruby language
  - Regular expressions
  - Arrays
  - Code blocks
  - Hash
  - File
  - Exceptions

CMSC 330

2

## Introduction

---

- ▶ That's it for the basics of Ruby
  - If you need other material for your project, come to office hours or check out the documentation
- ▶ Next up: How do regular expressions (REs) really work?
  - Mixture of a very practical tool (string matching with REs) and some nice theory
  - A great computer science result

CMSC 330

3

## A Few Questions About REs

---

- ▶ What does a regular expression represent?
  - Just a set of strings
- ▶ What are the basic components of REs?
  - E.g., we saw that  $e+$  is the same as  $ee^*$
- ▶ How are REs implemented?
  - We'll see how to build a structure to parse REs

CMSC 330

4

## Definition: Alphabet

---

- ▶ An **alphabet** is a **finite** set of symbols
  - Usually denoted  $\Sigma$
- ▶ Example alphabets:
  - Binary:  $\Sigma = \{0,1\}$
  - Decimal:  $\Sigma = \{0,1,2,3,4,5,6,7,8,9\}$
  - Alphanumeric:  $\Sigma = \{0-9,a-z,A-Z\}$

CMSC 330

5

## Definition: String

---

- ▶ A **string** is a finite sequence of symbols from  $\Sigma$ 
  - $\epsilon$  is the empty string (" " in Ruby)
  - $|s|$  is the length of string  $s$ 
    - >  $|\text{Hello}| = 5, |\epsilon| = 0$
  - Note:  $\emptyset$  is the empty set (with 0 elements);  $\emptyset \neq \{\epsilon\}$
- ▶ Example strings:
  - $0101 \in \Sigma = \{0,1\}$  (binary)
  - $0101 \in \Sigma = \text{decimal}$
  - $0101 \in \Sigma = \text{alphanumeric}$

CMSC 330

6

## Definition: Concatenation

- Concatenation is indicated by juxtaposition
  - If  $s_1 = \text{super}$  and  $s_2 = \text{hero}$ , then  $s_1s_2 = \text{superhero}$
  - Sometimes also written  $s_1 \cdot s_2$
  - For any string  $s$ , we have  $s\epsilon = \epsilon s = s$
  - You **can** concatenate strings from different alphabets, then the new alphabet is the union of the originals:
    - > If  $s_1 = \text{super} \in \Sigma_1 = \{s,u,p,e,r\}$  and  $s_2 = \text{hero} \in \Sigma_2 = \{h,e,r,o\}$ , then  $s_1s_2 = \text{superhero} \in \Sigma_3 = \{e,h,o,p,r,s,u\}$



CMSC 330

7

## Definition: Language

- A **language** is a set of strings over an alphabet
- Example: The set of phone numbers over the alphabet  $\Sigma = \{0, 1, 2, 3, 4, 5, 6, 7, 9, (, ), -\}$ 
  - Give an example element of this language (123) 456-7890
  - Are all strings over the alphabet in the language? **No**
  - Is there a Ruby regular expression for this language?  
`/\(\d{3,3}\)\ \d{3,3}-\d{4,4}/`
- Example: The set of all strings over  $\Sigma$ 
  - Often written  $\Sigma^*$

CMSC 330

8

## Definition: Language (cont.)

- Example: The set of strings of length 0 over the alphabet  $\Sigma = \{a, b, c\}$ 
  - $\{s \mid s \in \Sigma^* \text{ and } |s| = 0\} = \{\epsilon\} \neq \emptyset$
- Example: The set of all valid Ruby programs
  - Is there a Ruby regular expression for this language?  
**No.** Matching (an arbitrary number of) brackets so that they are balanced is impossible.  $\{\{\{ \dots \}\}\}$
- Can REs represent all possible languages?
  - The answer turns out to be no!
  - The languages represented by regular expressions are called, appropriately, the **regular languages**

CMSC 330

9

## Operations on Languages

- Let  $\Sigma$  be an alphabet and let  $L, L_1, L_2$  be languages over  $\Sigma$
- Concatenation  $L_1L_2$  is defined as
  - $L_1L_2 = \{xy \mid x \in L_1 \text{ and } y \in L_2\}$
  - Example:  $L_1 = \{\text{"hi"}, \text{"bye"}\}$ ,  $L_2 = \{\text{"1"}, \text{"2"}\}$ 
    - $L_1L_2 = \{\text{"hi1"}, \text{"hi2"}, \text{"bye1"}, \text{"bye2"}\}$
- Union is defined as
  - $L_1 \cup L_2 = \{x \mid x \in L_1 \text{ or } x \in L_2\}$
  - Example:  $L_1 = \{\text{"hi"}, \text{"bye"}\}$ ,  $L_2 = \{\text{"1"}, \text{"2"}\}$ 
    - $L_1 \cup L_2 = \{\text{"hi"}, \text{"bye"}, \text{"1"}, \text{"2"}\}$

CMSC 330

10

## Operations on Languages (cont.)

- Define  $L^n$  inductively as
  - $L^0 = \{\epsilon\}$
  - $L^n = LL^{n-1}$  for  $n > 0$
- In other words,
  - $L^1 = LL^0 = L\{\epsilon\} = L$
  - $L^2 = LL^1 = LL$
  - $L^3 = LL^2 = LLL$
  - ...

CMSC 330

11

## Examples of $L^n$

- Let  $L = \{a, b, c\}$
- Then
  - $L^0 = \{\epsilon\}$
  - $L^1 = \{a, b, c\}$
  - $L^2 = \{aa, ab, ac, ba, bb, bc, ca, cb, cc\}$

CMSC 330

12

## Operations on Languages (cont.)

- ▶ **Kleene closure** is defined as

$$L^* = \bigcup_{i \in [0..*]} L^i$$

- ▶ In other words...

$L^*$  is the language (set of all strings) formed by concatenating together zero or more strings from  $L$

CMSC 330

13

## Definition: Regular Expressions

- ▶ Given an alphabet  $\Sigma$ , the **regular expressions** over  $\Sigma$  are defined inductively as

regular expression	denotes language
$\emptyset$	$\emptyset$
$\epsilon$	$\{\epsilon\}$
each element $\sigma \in \Sigma$	$\{\sigma\}$

**Constants**

CMSC 330

14

## Definition: Regular Expressions (cont.)

- ▶ Let  $A$  and  $B$  be regular expressions denoting languages  $L_A$  and  $L_B$ , respectively

regular expression	denotes language
$AB$	$L_A L_B$
$(A B)$	$L_A \cup L_B$
$A^*$	$L_A^*$

**Operations**

- ▶ There are no other regular expressions over  $\Sigma$

CMSC 330

15

## Precedence

- ▶ Order in which operators are applied

- In arithmetic
  - > Multiplication  $\times$  > addition  $+$
  - >  $2 \times 3 + 4 = (2 \times 3) + 4 = 10$
- In regular expressions
  - > Kleene closure  $*$  > concatenation > union  $|$
  - >  $ab|c = (a b) | c = \{“ab”, “c”\}$
  - >  $ab^* = a (b^*) = \{“a”, “ab”, “abb”...\}$
  - >  $a|b^* = a | (b^*) = \{“a”, “”, “b”, “bb”, “bbb”...\}$
- Can change order using parentheses  $()$ 
  - > E.g.,  $a(b|c)$ ,  $(ab)^*$ ,  $(a|b)^*$

CMSC 330

16

## The Language Denoted by an RE

- ▶ For a regular expression  $e$ , we will write  $[[e]]$  to mean the language denoted by  $e$ 
  - $[[a]] = \{a\}$
  - $[[a|b]] = \{a, b\}$
- ▶ If  $s \in [[RE]]$ , we say that  $RE$  *accepts*, *describes*, or *recognizes*  $s$

CMSC 330

17

## Example 1

- ▶ All strings over  $\Sigma = \{a, b, c\}$  such that all the  $a$ 's are first, the  $b$ 's are next, and the  $c$ 's last
    - Example:  $aaabbbbccc$  but not  $abcb$
  - ▶ Regexp:  $a^*b^*c^*$ 
    - This is a valid regexp because:
      - >  $a$  is a regexp ( $[[a]] = \{a\}$ )
      - >  $a^*$  is a regexp ( $[[a^*]] = \{\epsilon, a, aa, \dots\}$ )
      - > Similarly for  $b^*$  and  $c^*$
      - > So  $a^*b^*c^*$  is a regular expression
- (Remember that we need to check this way because regular expressions are defined inductively.)

CMSC 330

18

## Which Strings Does $a^*b^*c^*$ Recognize?

aabbcc

Yes;  $aa \in [[a^*]]$ ,  $bb \in [[b^*]]$ , and  $cc \in [[c^*]]$ , so entire string is in  $[[a^*b^*c^*]]$

abb

Yes,  $abb = abbc$ , and  $\epsilon \in [[c^*]]$

ac

Yes

$\epsilon$

Yes

aacbc

No

abcd

No -- outside the language

CMSC 330

19

## Example 2

- ▶ All strings over  $\Sigma = \{a, b, c\}$
- ▶ Regexp:  $(a|b|c)^*$
- ▶ Other regular expressions for the same language?
  - $(c|b|a)^*$
  - $(a^*|b^*|c^*)^*$
  - $(a^*b^*c^*)^*$
  - $((a|b|c)^*|abc)$
  - etc.

CMSC 330

20

## Example 3

- ▶ All whole numbers containing the substring 330
- ▶ Regular expression:  $(0|1|\dots|9)^*330(0|1|\dots|9)^*$
- ▶ What if we want to get rid of leading 0's?
  - ▶  $((1|\dots|9)(0|1|\dots|9)^*330(0|1|\dots|9)^* | 330(0|1|\dots|9)^*)^*$
- ▶ Any other solutions?
- ▶ Challenge: What about all whole numbers **not** containing the substring 330?
  - Is it recognized by a regexp? Yes. We'll see how to find it later...

CMSC 330

21

## Example 4

- ▶ What is the English description for the language that  $(10|0)^*(10|1)^*$  denotes?
  - $(10|0)^*$ 
    - > 0 may appear anywhere
    - > 1 must always be followed by 0
  - $(10|1)^*$ 
    - > 1 may appear anywhere
    - > 0 must always be preceded by 1
  - Put together, all strings of 0's and 1's where every pair of adjacent 0's precedes any pair of adjacent 1's
    - > i.e., no 00 may appear after 11

CMSC 330

22

## What Strings are in $(10|0)^*(10|1)^*$ ?

00101000 110111101

First part in  $[[ (10|0)^* ]]$

Second part in  $[[ (10|1)^* ]]$

Notice that 0010 also in  $[[ (10|0)^* ]]$

But remainder of string is not in  $[[ (10|1)^* ]]$

0010101

Yes

101

Yes

011001

No

CMSC 330

23

## Example 5

- ▶ What language does this regular expression recognize?
  - $((1|\epsilon)(0|1|\dots|9) | (2(0|1|2|3))) : (0|1|\dots|5)(0|1|\dots|9)$
- ▶ All valid times written in 24-hour format
  - 10:17
  - 23:59
  - 0:45
  - 8:30

CMSC 330

24

## Two More Examples

- ▶  $(000|00|1)^*$ 
  - Any string of 0's and 1's with no single 0's
- ▶  $(00|0000)^*$ 
  - Strings with an even number of 0's
  - Notice that some strings can be accepted more than one way
    - >  $000000 = 00-00-00 = 00-0000 = 0000-00$
  - How else could we express this language?
    - >  $(00)^*$
    - >  $(00|000000)^*$
    - >  $(00|0000|000000)^*$
    - > etc...

CMSC 330

25

## Regular Languages

- ▶ The languages that can be described using regular expressions are the **regular languages** or **regular sets**
- ▶ Not all languages are regular
  - Examples (without proof):
    - > The set of palindromes over  $\Sigma$ 
      - reads the same backward or forward
    - >  $\{a^n b^n \mid n > 0\}$  ( $a^n$  = sequence of  $n$  a's)
- ▶ Almost all programming languages are not regular
  - But aspects of them sometimes are (e.g., identifiers)
  - Regular expressions are commonly used in parsing tools

CMSC 330

26

## Ruby Regular Expressions

- ▶ Almost all of the features we've seen for Ruby REs can be reduced to this formal definition
  - $/\text{Ruby}/$  – concatenation of single-character REs
  - $/(Ruby|Regular)/$  – union
  - $/(Ruby)^*/$  – Kleene closure
  - $/(Ruby)^+ /$  – same as  $(Ruby)(Ruby)^*$
  - $/(Ruby)? /$  – same as  $(\epsilon|(Ruby))$  ( $//$  is  $\epsilon$ )
  - $/[a-z]/$  – same as  $(a|b|c|\dots|z)$
  - $/[^0-9]/$  – same as  $(a|b|c|\dots)$  for  $a,b,c,\dots \in \Sigma - \{0..9\}$
  - $^, \$$  – correspond to extra characters in alphabet

CMSC 330

27

## Summary

- ▶ Languages
  - Sets of strings
  - Operations on languages
- ▶ Regular expressions
  - Constants
  - Operators
  - Precedence

CMSC 330

28