

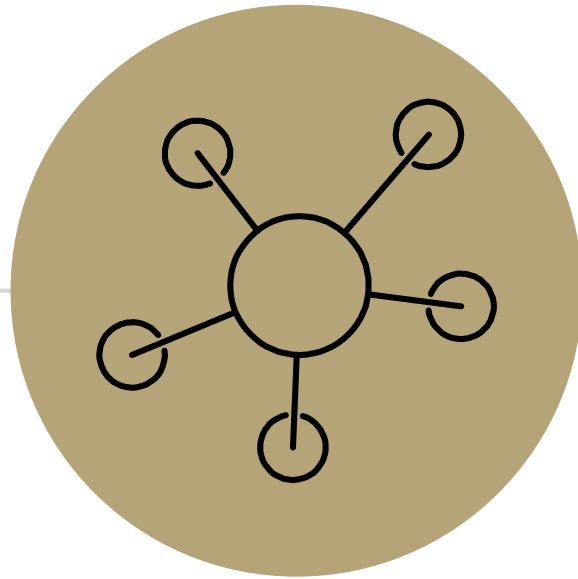
Warm up:  
How would you search through a document  
for all email addresses inside?  
(e.g. with control-f)

What if it was a document full of tweets?



# Regular Expressions (for real this time)

CSE 311 Winter 2022  
Lecture 20



Part 3 of the course!

# Course Outline

Symbolic Logic (training wheels; lectures 1-8)

Just make arguments in mechanical ways.

Set Theory/Arithmetic (bike in your backyard; lectures 9-19)

Models of computation (biking in your neighborhood; lectures 19-30)

Still make and communicate rigorous arguments

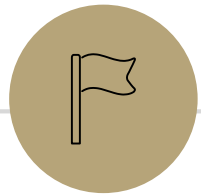
But now with objects you haven't used before.

- A first taste of how we can argue rigorously about computers.

This week: regular expressions and context free grammars – understand these “simpler computers”

After Thanksgiving: what these simple computers can do

Last week of class: what simple computers (and normal ones) can't do.



# Regular Expressions

---

# Regular Expressions

I have a giant text document. And I want to find all the email addresses inside. What does an email address look like?

[some letters and numbers] @ [more letters] . [com, net, or edu]

We want to ctrl-f for a **pattern of strings** rather than a single string

# Languages

A set of strings is called a **language**.

$\Sigma^*$  is a language

“the set of all binary strings of even length” is a language.

“the set of all palindromes” is a language.

“the set of all English words” is a language.

“the set of all strings matching a given **pattern**” is a language.

# Regular Expressions

Every pattern automatically gives you a language .  
The set of all strings that match that pattern.

We'll formalize "patterns" via "regular expressions"

$\epsilon$  is a regular expression. The empty string itself matches the pattern (and nothing else does).

$a$  is a regular expression, for any  $a \in \Sigma$  (i.e. any character). The character itself matching this pattern.

$\emptyset$  is a regular expression. No strings match this pattern.

# Regular Expressions

## Basis:

$\varepsilon$  is a regular expression. The empty string itself matches the pattern (and nothing else does).

$\emptyset$  is a regular expression. No strings match this pattern.

$a$  is a regular expression, for any  $a \in \Sigma$  (i.e. any character). The character itself matching this pattern.

## Recursive

If  $A, B$  are regular expressions then  $(A \cup B)$  is a regular expression  
matched by any string that matches  $A$  or that matches  $B$  [or both]).

If  $A, B$  are regular expressions then  $AB$  is a regular expression.  
matched by any string  $x$  such that  $x = yz$ ,  $y$  matches  $A$  and  $z$  matches  $B$ .

If  $A$  is a regular expression, then  $A^*$  is a regular expression.  
matched by any string that can be divided into 0 or more strings that match  $A$ .

# Regular Expressions

$(a \cup bc)$

$0(0 \cup 1)1$

$0^*$

$(0 \cup 1)^*$

# Regular Expressions

$(a \cup bc)$

Corresponds to  $\{a, bc\}$

$0(0 \cup 1)1$

Corresponds to  $\{001, 011\}$

all length three strings that start with a 0 and end in a 1.

$0^*$

Corresponds to  $\{\epsilon, 0, 00, 000, 0000, \dots\}$

$(0 \cup 1)^*$

Corresponds to the set of all binary strings.

# More Examples

$(0^*1^*)^*$

$0^*1^*$

$(0 \cup 1)^*(00 \cup 11)^*(0 \cup 1)^*$

$(00 \cup 11)^*$

[Pollev.com/uwcse311](https://pollev.com/uwcse311)

# More Examples

$(0^*1^*)^*$

All binary strings

$0^*1^*$

All binary strings with any 0's coming before all 1's

$(0 \cup 1)^*(00 \cup 11)^*(0 \cup 1)^*$

This is all binary strings again. Not a "good" representation, but valid.

$(00 \cup 11)^*$

All binary strings where 0s and 1s come in pairs

# More Practice

You can also go the other way

Write a regular expression for “the set of all binary strings of odd length”

Write a regular expression for “the set of all binary strings with at most two ones”

Write a regular expression for “strings that don’t contain 00”

# More Practice

You can also go the other way

Write a regular expression for “the set of all binary strings of odd length”

$(0 \cup 1)(00 \cup 01 \cup 10 \cup 11)^*$

Write a regular expression for “the set of all binary strings with at most two ones”

$0^*(1 \cup \epsilon)0^*(1 \cup \epsilon)0^*$

Write a regular expression for “strings that don’t contain 00”

$(01 \cup 1)^*(0 \cup \epsilon)$  (key idea: all 0s followed by 1 or end of the string)

# Practical Advice

Check  $\varepsilon$  and 1 character strings to make sure they're excluded or included (easy to miss those edge cases).

If you can break into pieces, that usually helps.

"nots" are hard (there's no "not" in standard regular expressions)

But you can negate things, usually by negating at a low-level. E.g. to have binary strings without 00, your building blocks are 1's and 0's followed by a 1

$(01 \cup 1)^*(0 \cup \varepsilon)$  then make adjustments for edge cases (like ending in 0)

Remember  $*$  allows for 0 copies! To say "at least one copy" use  $AA^*$ .

# Regular Expressions In Practice

EXTREMELY useful. Used to define valid "tokens" (like legal variable names or all known keywords when writing compilers/languages)

Used in `grep` to actually search through documents.

```
Pattern p = Pattern.compile("a*b");
Matcher m = p.matcher("aaaaab");
boolean b = m.matches();
```

`^` start of string

`$` end of string

`[01]` a 0 or a 1

`[0-9]` any single digit

`\.` period    `\,` comma    `\-` minus

`.` any single character

`ab` a followed by b                    **(AB)**

`(a|b)` a or b    **(A ∪ B)**

`a?` zero or one of a                    **(A ∪ ε)**

`a*` zero or more of a                    **A\***

`a+` one or more of a                    **AA\***

e.g. `^[\\-+]?[0-9]*(\\.|\\,)?[0-9]+$`

General form of decimal number e.g. 9.12 or -9,8 (Europe)

# Regular Expressions In Practice

When you only have ASCII characters (say in a programming language)

| usually takes the place of  $\cup$

? (and perhaps creative rewriting) take the place of  $\varepsilon$ .

E.g.  $(0 \cup \varepsilon)(1 \cup 10)^*$  is  $0?(1|10)^*$

# A Final Vocabulary Note

Not everything can be represented as a regular expression.

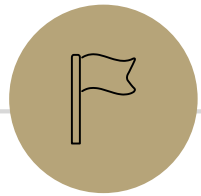
E.g. “the set of all palindromes” is not the language of any regular expression.

Some programming languages define features in their “regexes” that can’t be represented by our definition of regular expressions.

Things like “match this pattern, then have exactly that **substring** appear later.

So before you say “ah, you can’t do that with regular expressions, I learned it in 311!” you should make sure you know whether your language is calling a more powerful object “regular expressions”.

But the more “fancy features” beyond regular expressions you use, the slower the checking algorithms run, (and the harder it is to force the expressions to fit into the framework) so this is still very useful theory.



# Context Free Grammars

# What Can't Regular Expressions Do?

Some easy things

Things where you could say whether a string matches with just a loop

$\{0^k 1^k : k \geq 0\}$

The set of all palindromes.

And some harder things

Expressions with matched parentheses

Properly formed arithmetic expressions

Context Free Grammars can solve all of these problems!

# Context Free Grammars

A context free grammar (CFG) is a finite set of production rules over:

An alphabet  $\Sigma$  of "terminal symbols"

A finite set  $V$  of "nonterminal symbols"

A start symbol (one of the elements of  $V$ ) usually denoted  $S$ .

A production rule for a nonterminal  $A \in V$  takes the form

$$A \rightarrow w_1 | w_2 | \cdots | w_k$$

Where each  $w_i \in (V \cup \Sigma)^*$  is a string of nonterminals and terminals.

# Context Free Grammars

We think of context free grammars as **generating** strings.

1. Start from the start symbol  $S$ .
2. Choose a nonterminal in the string, and a production rule  $A \rightarrow w_1 | w_2 | \dots | w_k$  replace that copy of the nonterminal with  $w_i$ .
3. If no nonterminals remain, you're done! Otherwise, goto step 2.

A string is in the language of the CFG iff it can be generated starting from  $S$ .

# Examples

$$S \rightarrow 0S0|1S1|0|1|\varepsilon$$

$$S \rightarrow 0S|S1|\varepsilon$$

$$S \rightarrow (S)|SS|\varepsilon$$

# Arithmetic

$E \rightarrow E + E | E * E | (E) | x | y | z | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9$

Generate  $(2 * x) + y$

Generate  $2 + 3 * 4$  in two different ways

[Pollev.com/uwcse311](https://pollev.com/uwcse311)

# Arithmetic

$E \rightarrow E + E | E * E | (E) | x | y | z | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9$

Generate  $(2 * x) + y$

$E \Rightarrow E + E \Rightarrow (E) + E \Rightarrow (E * E) + E \Rightarrow (2 * E) + E \Rightarrow (2 * x) + E \Rightarrow (2 * x) + y$

Generate  $2 + 3 * 4$  in two different ways

$E \Rightarrow E + E \Rightarrow E + E * E \Rightarrow 2 + E * E \Rightarrow 2 + 3 * E \Rightarrow 2 + 3 * 4$

$E \Rightarrow E * E \Rightarrow E + E * E \Rightarrow 2 + E * E \Rightarrow 2 + 3 * E \Rightarrow 2 + 3 * 4$

# Parse Trees

Suppose a context free grammar  $G$  generates a string  $x$

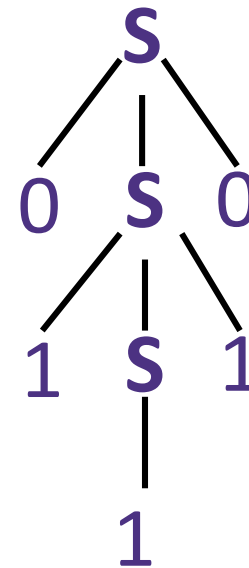
A parse tree of  $x$  for  $G$  has

Rooted at  $S$  (start symbol)

Children of every  $A$  node are labeled with the characters of  $w$  for some  $A \rightarrow w$

Reading the leaves from left to right gives  $x$ .

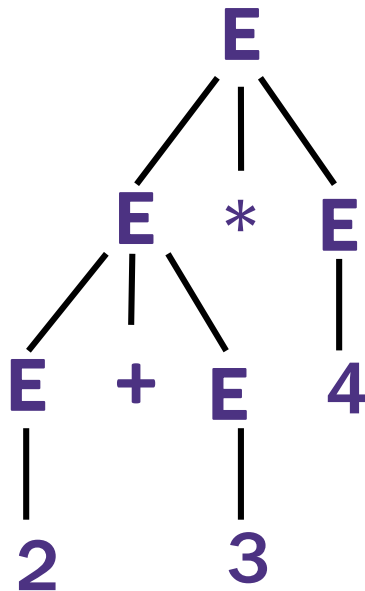
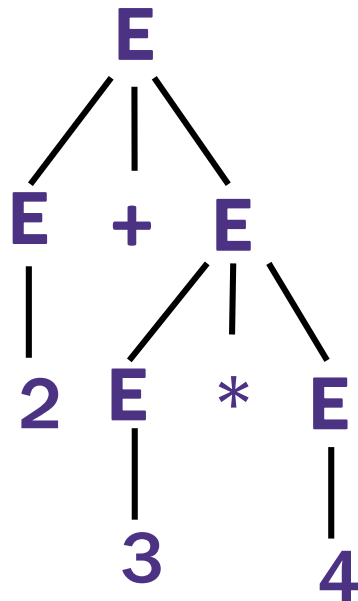
$$S \rightarrow 0S0 \mid 1S1 \mid 0 \mid 1 \mid \varepsilon$$



# Back to the arithmetic

$E \rightarrow E + E | E * E | (E) | x | y | z | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9$

Two parse trees for  $2 + 3 * 4$



# How do we encode order of operations

If we want to keep "in order" we want there to be only one possible parse tree.

Differentiate between "things to add" and "things to multiply"

Only introduce a \* sign after you've eliminated the possibility of introducing another + sign in that area.

$$E \rightarrow T | E + T$$

$$T \rightarrow F | T * F$$

$$F \rightarrow (E) | N$$

$$N \rightarrow x | y | z | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9$$

