



# COMP261

## Parsing, 1 of 4

Marcus Fread

**Victoria**  
UNIVERSITY OF WELLINGTON

*Te Whare Wānanga  
o te Ūpoko o te Ika a Māui*













CAPITAL CITY UNIVERSITY

## a bit about me

- 1. brains are a mystery → AI & machine learning
- 2. doubt is important → uncertainty, probabilities
- 3. living things are collections → evolution of cooperation

(2018: @ Max Planck Institute looking into origins of money)

# what's coming up in the rest of COMP261?

1. grammars and parsing    
2. searching texts  
3. text compression  
4. using predictions  

And: two assignments.

# What makes these texts “structured”?

## SQL schema definition or query:

```
DELETE FROM DomesticStudentsFor2018  
WHERE mark = 'E';
```

## Java statement:

```
while ( A[k] != x ) { k++; }
```

## XML documents:

```
<html><head><title>My Web Page</title></head>  
<body><p> Thanks for viewing </p></body></html>
```

# Structured text

Text is “structured” if it can be described using a **grammar**:

- *a set of rules*
- the rules describe how to form strings from the language’s alphabet that are valid according to *the language’s syntax*.
- but a grammar does not describe the meaning of the strings or what can be done with them in whatever context – only their *form*.

# How to define a grammar?

- Specify a finite set of acceptable sentences..?
- But what about infinite languages?
- Instead: look for a finite recipe that describes all acceptable sentences

**A grammar is a finite description of a possibly infinite set of acceptable sentences**

Today we will look at the simplest grammars, or rather an equivalent representation we call regular expressions

ways of representing a language

#1: by a **regex** (regular expression)

there's the whole set of everything that's “legal”, e.g.

legal:           a, ab, aaa, aab, aaaaaaabbbbb, b, bbb, ....

not legal:       ba, aba, abba, bbbbbbbaaaaaa, ...

A very condensed way of saying this is a **regular expression**:

$a^*b^*$

Notice the ‘regex’ is finite even though the list might not be

# #1: pattern matching with “regular expressions”

- `abc` matches “abc”
- `a|e|i|o|u` matches any one out of a,e,i,o,u
- `.` dot just means any char
- `[0-9]` numerals
- `[a-z]` letters
- `\d` digit
- `\w` whitespace
- `*` any number of
- `+` at least one of
- `?` optional



# there are lots of great web resources

a couple of examples:

- <https://regexone.com/>  
(note in Java you have to use *two* \ 's )
- and ... <http://web.mit.edu/6.005/www/fa16/classes/17-regex-grammars/>

# regex in Java

Helpful: <https://regexone.com/references/java>

And here's the start of the relevant [java docs](#):

.	Any character (may or may not match line terminators)
\d	A digit: [0-9]
^	“not something”, e.g. [^0-9] means <i>not</i> a digit
\s	A whitespace character: [ \t\n\x0B\f\r]
\S	A non-whitespace character: [^\s]
XY	X followed by Y
X Y	Either X or Y
(X)	X, as a capturing group
X*	X, zero or more times
X+	X, one or more times
X{n}	X, exactly n times

## regex get converted → graphs

To actually do matches in bulk, javac turns a regex into a finite state machine (called an “*Acceptor*”).

Making the network isn't quite trivial - it's quite an expensive process.

---

If curious: here's someone talking through the process  
<https://www.youtube.com/watch?v=GwsU2LPs85U>

And here's someone converting regex to FSM:  
(unfortunately they use “+” operator in a non-standard way though!)  
[http://ivanzuzak.info/noam/webapps/fsm\\_simulator/](http://ivanzuzak.info/noam/webapps/fsm_simulator/)

## Java does it like this

```
boolean b = Pattern.matches("a*b", "aaaaab");
```

But this combines the generation of the *Acceptor* (for the first arg) and its use on the particular string (second arg).

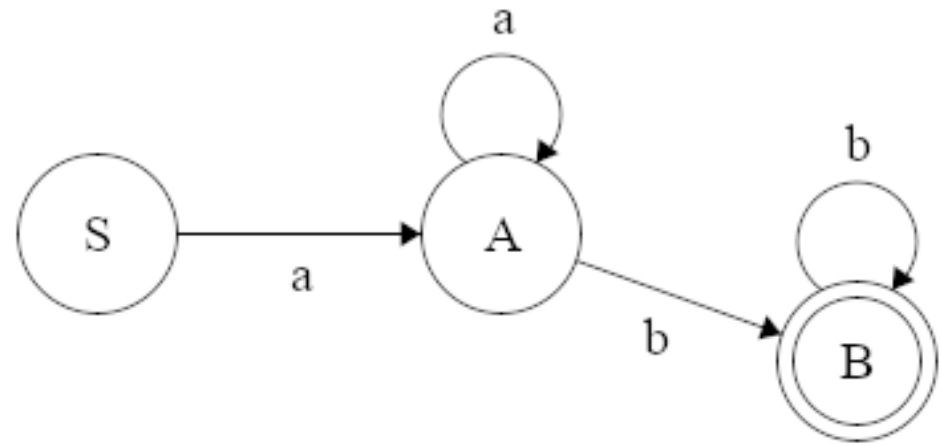
But it's expensive to build that graph ("Thompson's construction algorithm")

So if you want to match lots of strings it's better to do it like this (from the docs):

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

## eg. acceptors

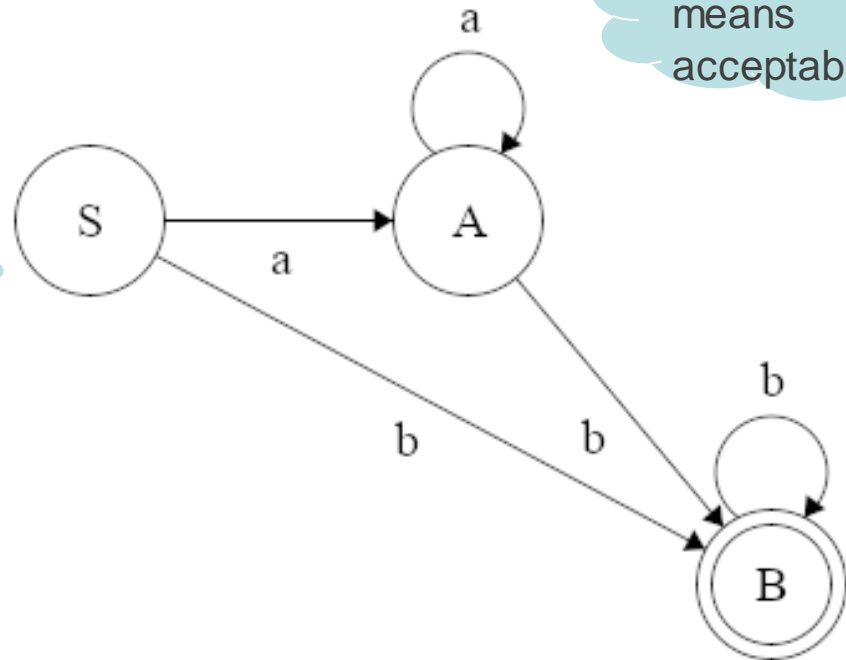
- $a^+b^+$
- $aa^*bb^*$



double circle  
means  
acceptable

- $a^*b^+$

optional: we  
could have just  
made A the start  
state...



## tomorrow: ways of representing a language #2: grammar

Rules that spell out what's possible, eg:

- $S \rightarrow aA$
- $A \rightarrow aA \mid bB$
- $B \rightarrow bB \mid \text{end}$

(this is same as the regex:  $a^+b^+$ )

Tomorrow we will see that regex's correspond to just the simplest grammars - the “regular” ones