COMP112 in Week 7, 2022

STRUCTURE

in text : regexs and grammars

Are these texts "structured"?

SQL schema definition or query:

```
DELETE FROM DomesticStudentsFor2022
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>
```

Whakatauki:

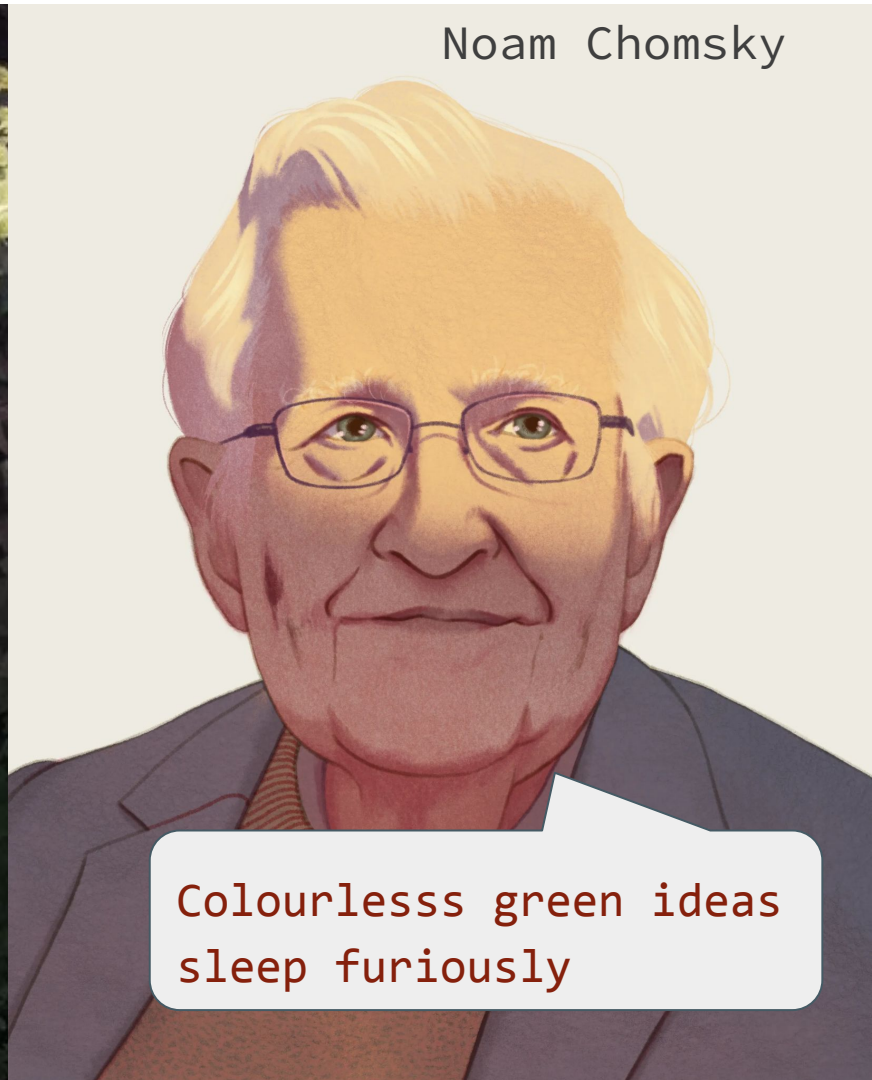
I orea te tuatara ka puta ki waho

(a problem is solved by continuing to find solutions)

Poem:

I've hunted near,
I've hunted far
I even looked inside my car.
I've lost my glasses,
I'm in need,
To have them now so I can read.

Anne Scott, 2014


Noam Chomsky
Colourlesss green ideas sleep furiously

How to define syntax?

- Specify a finite set of acceptable sentences..?
- Instead: look for a finite recipe that describes all acceptable sentences

So 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 (“regex”)

REGULAR EXPRESSIONS

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

legal: a, ab, aaa, aab, aaaaaaaabbbbb, b, bbb, ….
not legal: ba, aba, abba, bbbbbbaaaaaa, …

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

a*b*

➡ note the 'regex' is finite even though the list might not be.

Regex in Java

Here's the start of the relevant java docs:

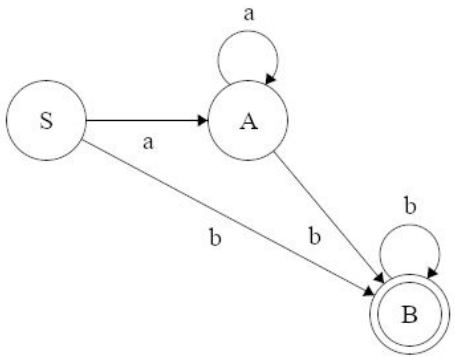
.	Any character
\d	A digit: [0-9]
^	"not something", e.g. [^0-9] means *not* a digit
\s	A whitespace character: [\t\n\x0B\f\r]
\S	A non-whitespace character: [^\s]
XY	X followed by Y
X|Y	Either X or Y
(X)	X, as a capturing group
X*	X, zero or more times
X+	X, one or more times
X{n}	X, exactly n times

helpful:
https://regexone.com/references/java

REGEX ⟶ AUTOMATA

the regex is just a code: javac turns it into a finite state machine

Wikipedia:
“A **finite-state machine** (**FSM**) or **finite-state automaton** is a mathematical model of computation.”



Constructing this machine isn’t always trivial.

Java does it like this

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

combines the generation of the FSM (for the first arg) and its use on the particular string (second arg).

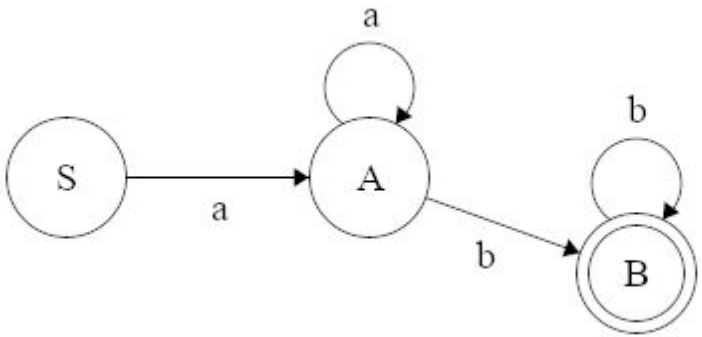
But because it's expensive to build that FSM, if you want to match lots of strings it's better to do it like this (from the docs):

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

WAYS OF REPRESENTING A REGULAR LANGUAGE #2:

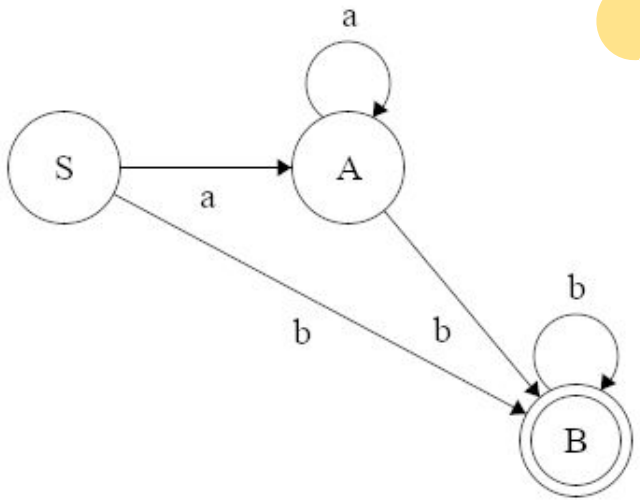
FINITE STATE MACHINE

- a+b+



double circle means acceptable

- aa*bb*
- a*b+



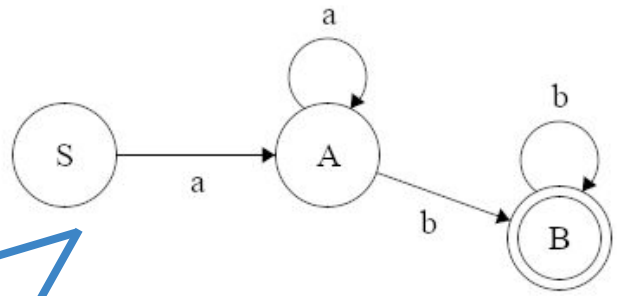
optional: we could have just made A the start state…

WAYS OF REPRESENTING A REGULAR LANGUAGE #3:

GRAMMAR

Rules that spell out what's possible, eg:

- S → aA
- A → aA | bB
- B → bB | end



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

regex's correspond to just the simplest grammars - the "regular" ones

regular grammars ⇔ regex (describe the same set of sequences)

each rule component is either

- a "terminal" (a,b…) OR
- a terminal and a non-terminal (A,B,...)

Example of a "context free" grammar

```
HTMLFILE □  "<html>"  [ HEAD ]  BODY "</html>"
HEAD     □  "<head>"  TITLE  "</head>"
TITLE    □  "<title>"  TEXT "</title>"
BODY     □  "<body>"  [ BODYTAG ]*  "</body>"
BODYTAG  □  H1TAG  |  PTAG  |  OLTAG  |  ULTAG
H1TAG    □  "<h1>"  TEXT  "</h1>"
PTAG     □  "<p>"  TEXT  "</p>"
OLTAG    □  "<ol>"  [ LITAG ]+  "</ol>"
ULTAG    □  "<ul>"  [ LITAG ]+  "</ul>"
LITAG    □  "<li>"  TEXT  "</li>"
TEXT     □  sequence of  characters
            other than < and >
```

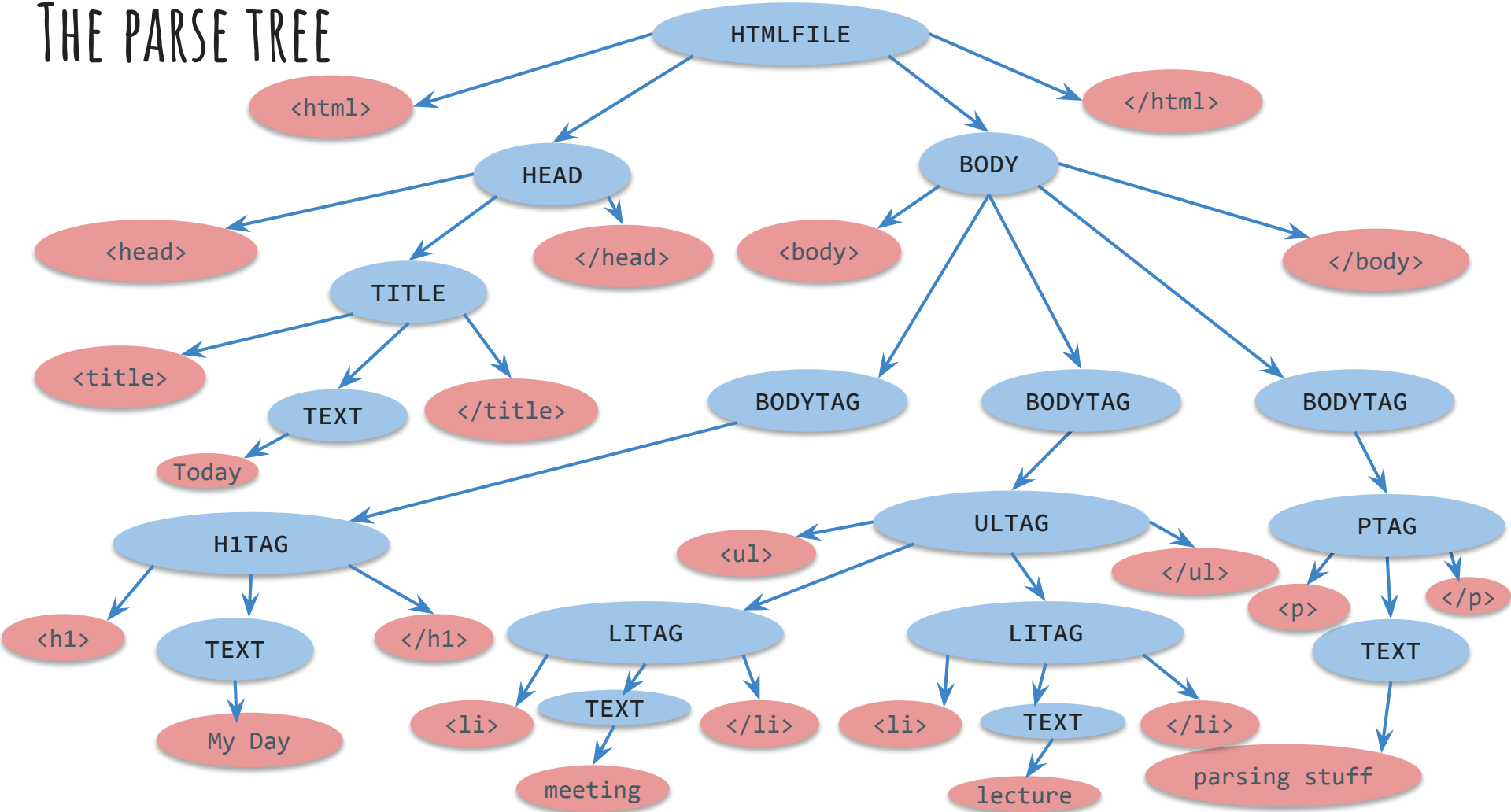
more than a "regular" grammar

regular grammars ⇔ regex (describe the same set of sequences)

each rule component is either

- a "terminal" (a,b…)
  OR
- a terminal and a non-terminal (A,B,...)

The parse tree



Context-Free grammars

we've gone beyond regex (== **regular** grammars) to what are called **context free** grammars (CFG).

Regular grammars can't do "nesting", but CFG can.

e.g. can you write a regex for a language that includes

(x) ((x)) (((x))) ((((x)))) ….

?

Specific cases are fine, but the general matching between number of brackets isn't. This CFG can do it though:

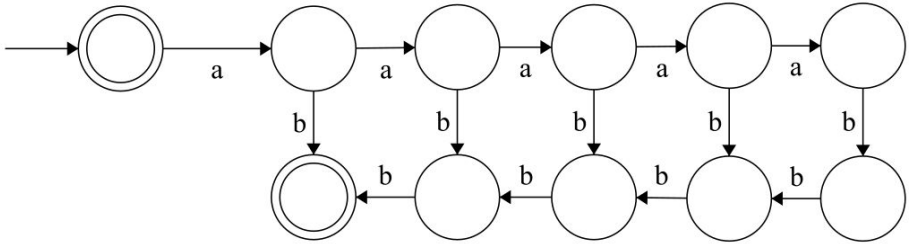
EXPR ::= "x" | "(" EXPR ")"

HOW ABOUT THIS?

can you write a regex for a language that includes (only)

ab aabb aaabbb aaaabbbb ...

?



regex / FSM / regular grammar

attempts (fail)

Intuitively, you'd need some kind of a "**stack**", right?

This Context Free Grammar can do it:

EXPR → "a" "b" | "a" EXPR "b"

THERE ARE LOTS OF GREAT WEB RESOURCES

just a couple of fun examples if you're interested:

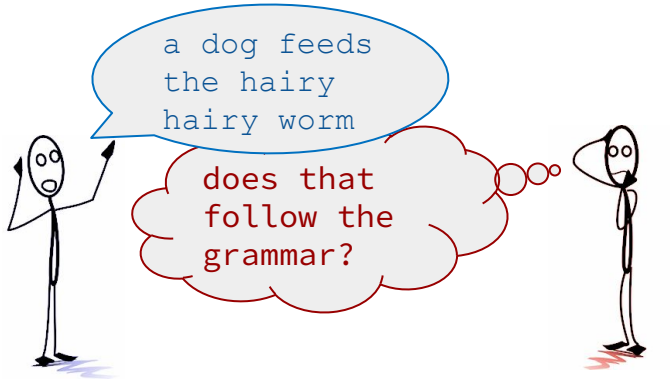
- https://regexone.com/
  (note in Java you have to use *two* \'s)

- and … http://web.mit.edu/6.005/www/fa16/classes/17-regex-grammars/

What Code shall we write tomorrow?

We will read in a file that contains a grammar, such as

```
START -> NP VP
NP    -> the NOUN | a NOUN
VP    -> VERB NP
NOUN  -> THING | ADJ NOUN
THING -> dog | cat | worm | ferret | door knob
VERB  -> eats | squashes | feeds | loves
ADJ   -> hairy | smooth
```



And then we will generate lots of strings from this grammar, like this:

the cat loves a dog

the hairy cat loves the door knob

the smooth hairy smooth hairy cat squashes the smooth ferret

the smooth ferret eats a hairy dog

a smooth dog eats the hairy door knob