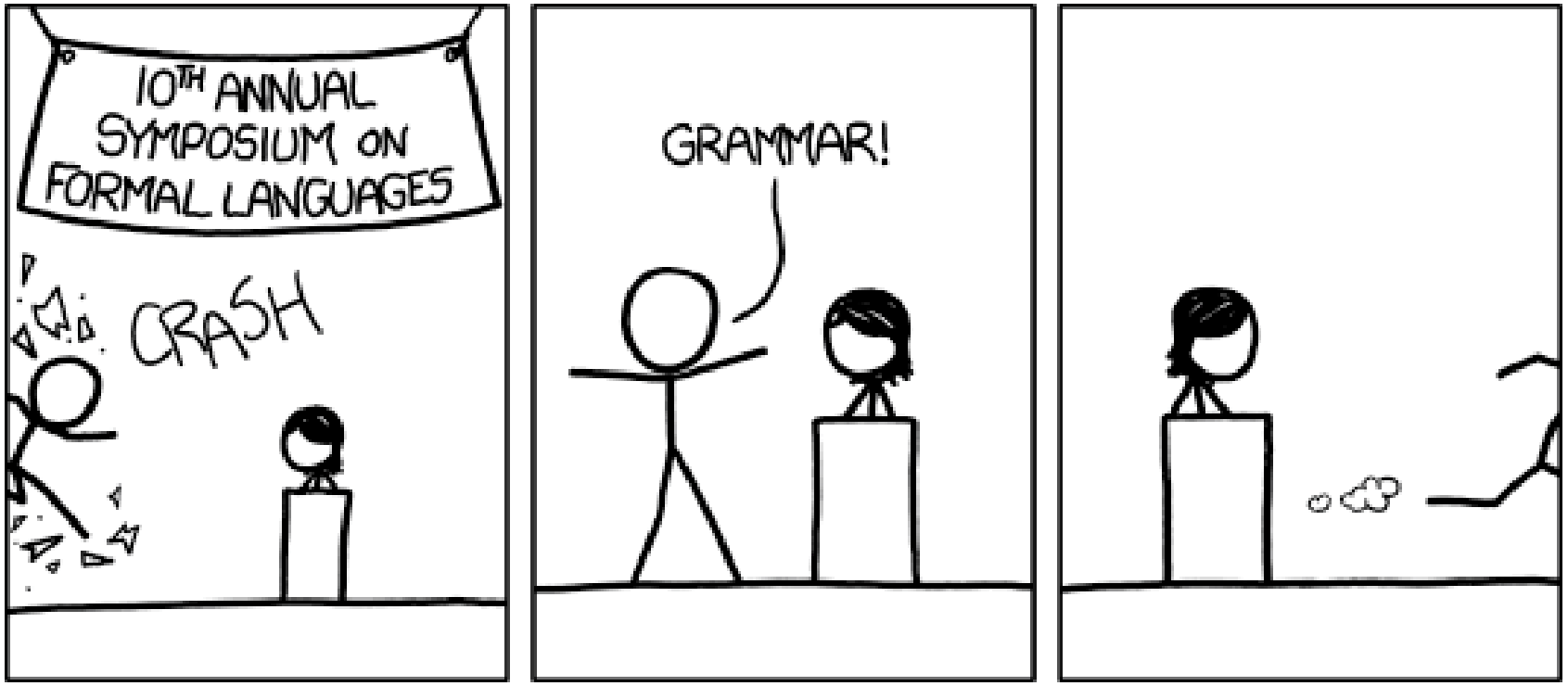
Warm-up
Write a regular expression for "the set of all binary strings which represent binary numbers congruent to 1 (mod 4) (make sure the representation is "nice" e.g. 0001 is not in the language)

What about congruent to 0 mod 4?



[Audience looks around] "What just happened?" "There must be some context we're missing."
xkcd.com/1090

Context Free Grammars

CSE 311 Autumn 20
Lecture 21

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 ε 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 ∪

? (and perhaps creative rewriting) take the place of ε.

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

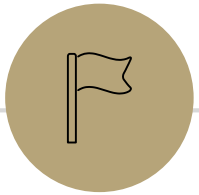
A Final Vocabulary Note

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
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 Σ 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

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

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 $A \to w_1|w_2|\ldots|w_k$ 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.

Notation: $xAy \Rightarrow xwy$ is rewriting A with w.

Examples

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

the language "the set of all palindromes"

$S \to 0S|S1|\varepsilon$

0^*1^*

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

balanced parentheses

$S \to AB$

$A \to 0A1|\varepsilon$

$B \to 1B0|\varepsilon$

Examples

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

The set of all binary palindromes

$S \to 0S|S1|\varepsilon$

The set of all strings with any 0's coming before any 1's (i.e. 0^*1^*)

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

Balanced parentheses

$S \to AB$

$A \to 0A1|\varepsilon$

$B \to 1B0|\varepsilon$ $\quad \{0^j 1^{j+k} 0^k : j, k \geq 0\}$

Arithmetic

$\Sigma = \{+, *, (,), x, y, z, 0, 1, \ldots, 9\}$

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

Generate $(2 * x) + y$

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

Fill out the poll everywhere for Activity Credit!

Go to pollev.com/cse311 and login with your UW identity
Or text cse311 to 22333

Arithmetic

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

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

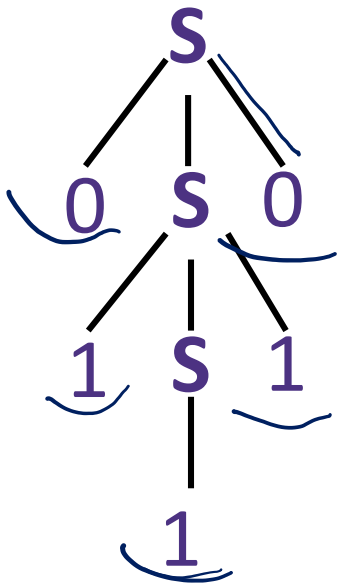
A parse tree of x for G has

Rooted at S (start symbol)

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

Reading the leaves from left to right gives x.

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

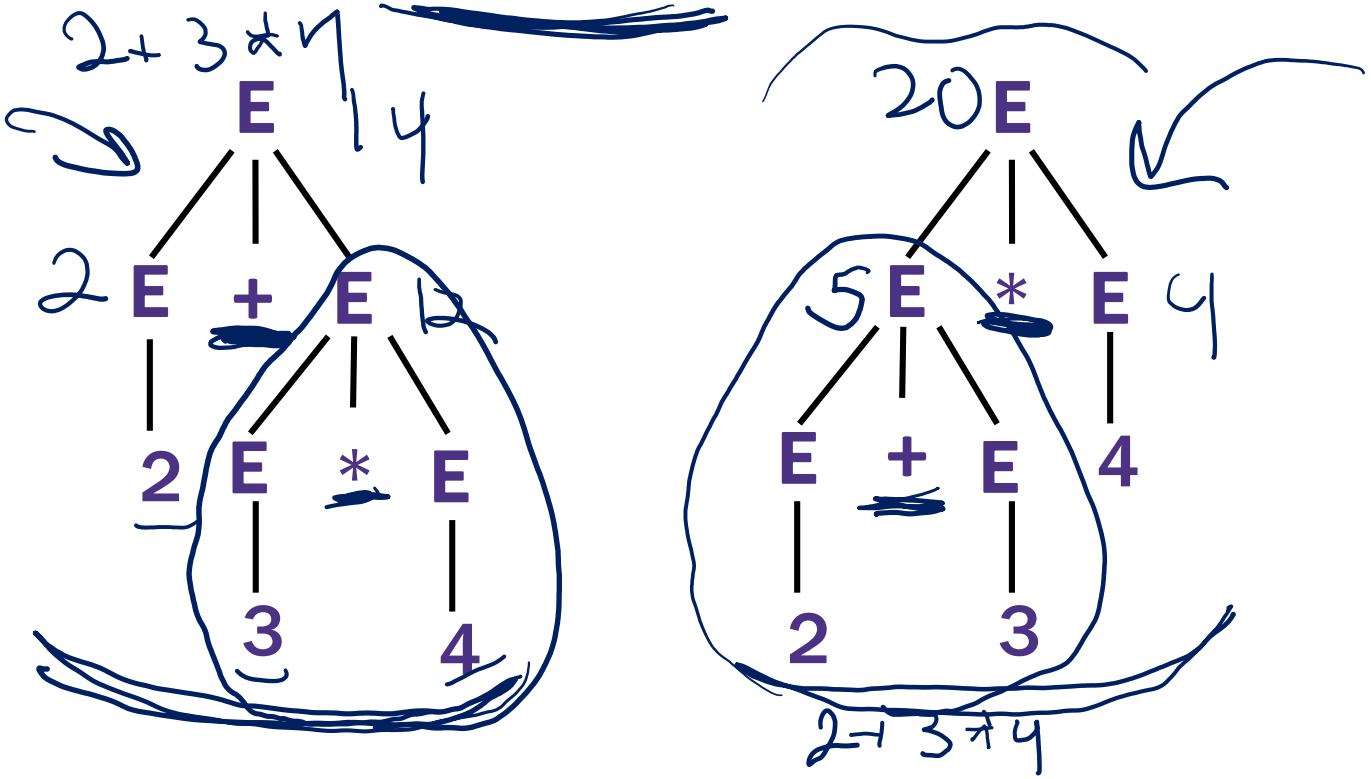


Back to the arithmetic

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

↳ ambiguity

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”

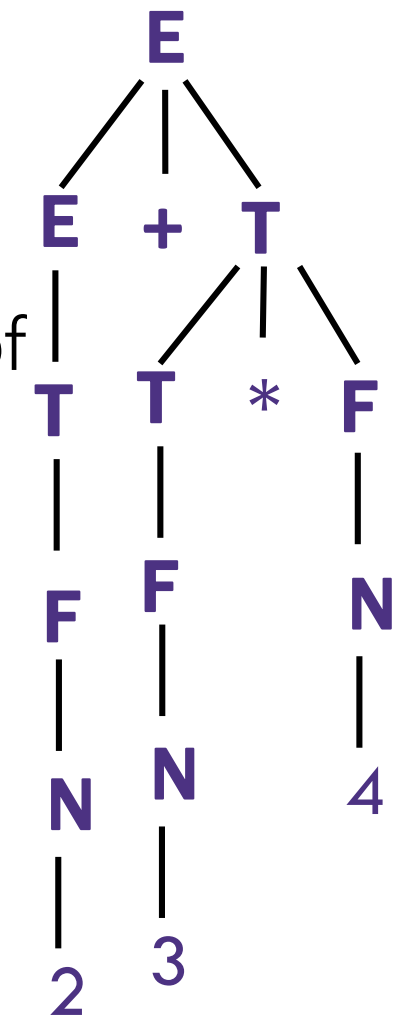
Only introduce a * sign after you’ve eliminated the possibility of introducing another + sign in that area.

$E \rightarrow T|E + T$

$T \rightarrow F|T * F$

$F \rightarrow (E)|N$

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



CNFs in practice

Used to define programming languages.

Often written in Backus-Naur Form – just different notation

Variables are <names-in-brackets>

like <if-then-else-statement>, <condition>, <identifier>

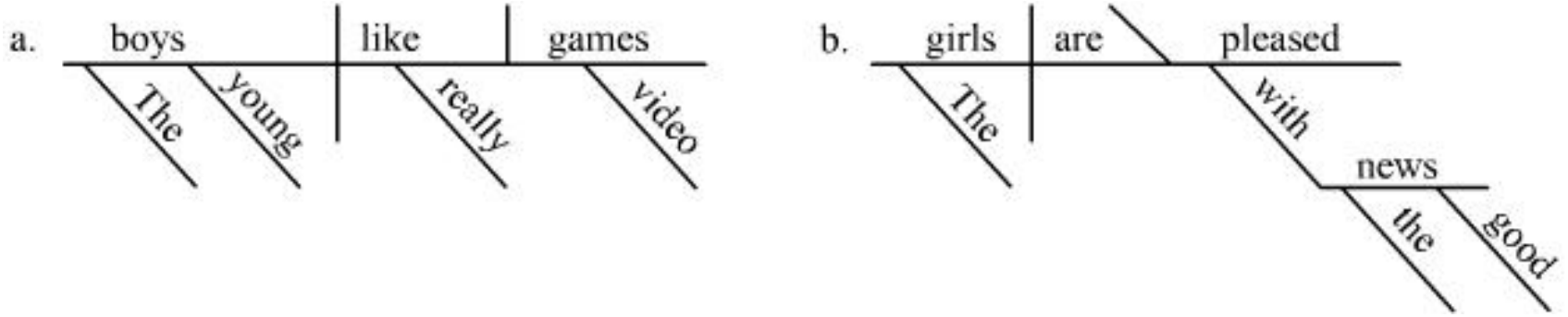
→ is replaced with ::= or :

BNF for C (no <...> and uses : instead of ::=)

```
statement:
  ((identifier | "case" constant-expression | "default") ":")*
  (expression? ";" |
   block |
   "if" "(" expression ")" statement |
   "if" "(" expression ")" statement "else" statement |
   "switch" "(" expression ")" statement |
   "while" "(" expression ")" statement |
   "do" statement "while" "(" expression ")" ";" |
   "for" "(" expression? ";" expression? ";" expression? ")" statement |
   "goto" identifier ";" |
   "continue" ";" |
   "break" ";" |
   "return" expression? ";"
  )

block: "{" declaration* statement* "}"

expression:
  assignment-expression%

assignment-expression: (
    unary-expression (
      "=" | "*=" | "/=" | "%=" | "+=" | "-=" | "<<=" | ">>=" | "&=" |
      "^=" | "|="
    )
  )* conditional-expression

conditional-expression:
  logical-OR-expression ( "?" expression ":" conditional-expression )?
```

Parse Trees

Remember diagramming sentences in middle school?



<sentence>::=<noun phrase><verb phrase>

<noun phrase>::=<determiner><adjective><noun>

<verb phrase>::=<verb><adverb>|<verb><object>

<object>::=<noun phrase>

Parse Trees

<sentence>::=<noun phrase><verb phrase>

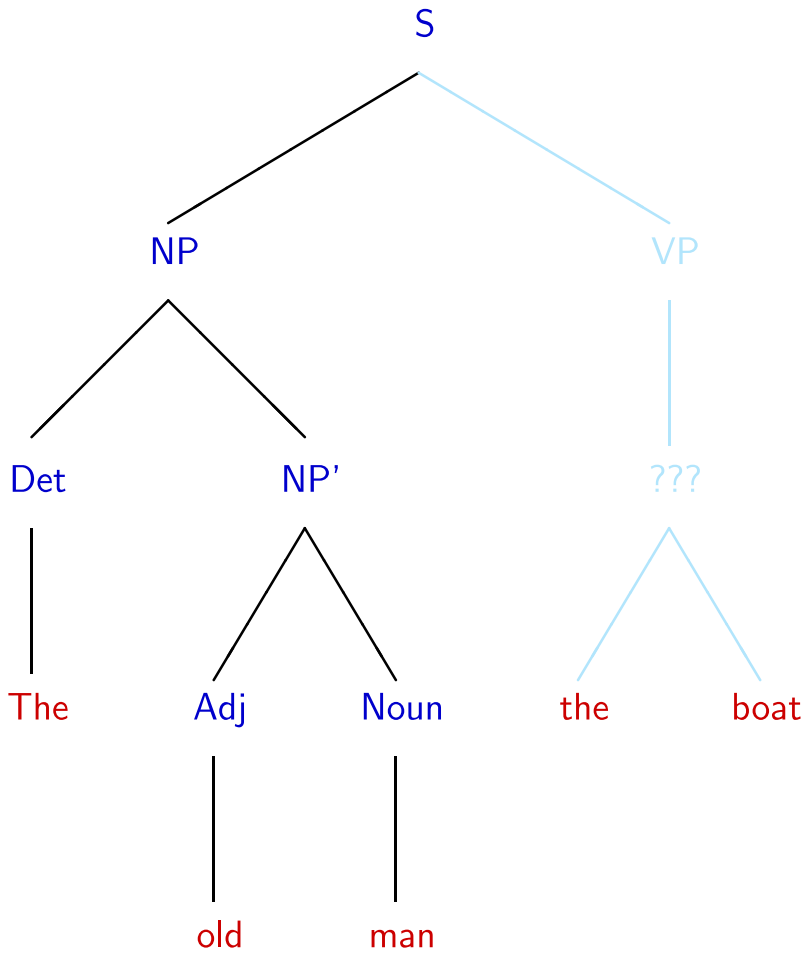
<noun phrase>::=<determiner><adjective><noun>

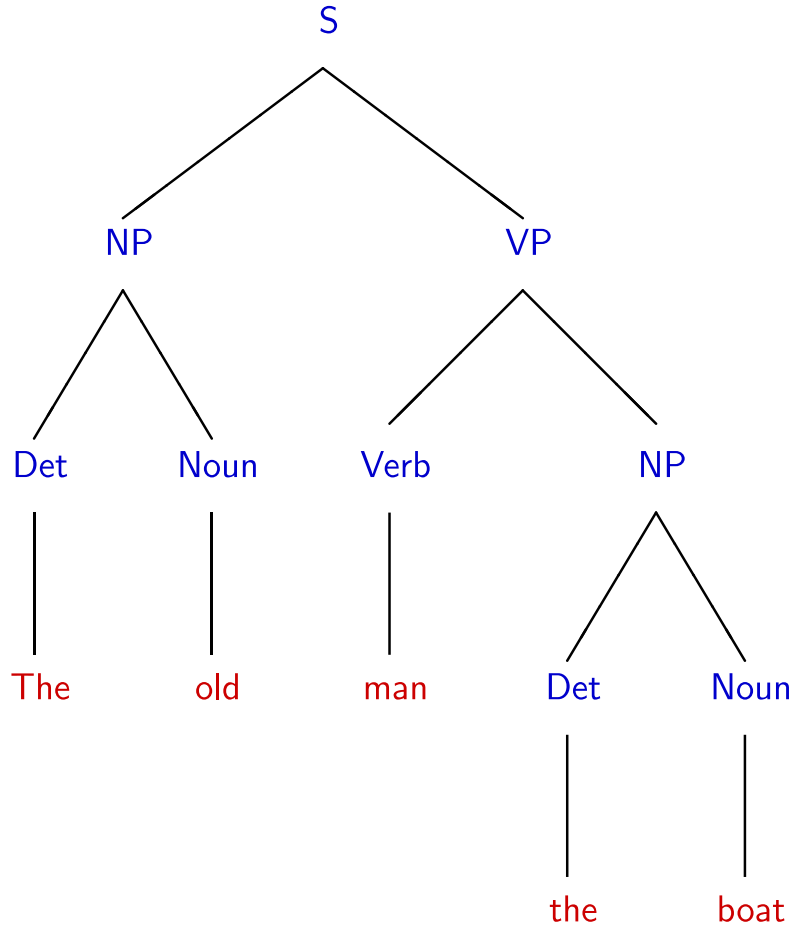
<verb phrase>::=<verb><adverb>|<verb><object>

<object>::=<noun phrase>

The old man the boat.

The old man the boat





Power of Context Free Languages

There are languages CFGs can express that regular expressions can't

e.g. palindromes

What about vice versa – is there a language that a regular expression can represent that a CFG can't?

No!

Are there languages even CFGs cannot represent?

Yes!

$\{0^k 1^j 2^k 3^j | j, k \geq 0\}$ cannot be written with a context free grammar.

Takeaways

CFGs and regular expressions gave us ways of succinctly representing sets of strings

- Regular expressions super useful for representing things you need to search for
- CFGs represent complicated languages like “java code with valid syntax”

After Thanksgiving, we’ll talk about how each of these are “equivalent to weaker computers.”

Next time: Two more tools for our toolbox.