Context Free Grammars

CSE 311 Autumn 2025
Lecture 23

Regular Expressions

Basis:

ε 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.

What **can’t** regular expressions do? (1)

Can you write a regular expression for all binary palindromes?

Can you write a regular expression for all binary palindromes of length at most 100?

What **can't** regular expressions do? (1 answers)

Can you write a regular expression for all binary palindromes?

No! There is no such regular expression (we'll prove it in a few weeks).

Can you write a regular expression for all binary palindromes of length at most 100?

Yes! It'll probably take you a while to write it though…

…there are a finite number of strings satisfying this description, just list them all and ∪ them together.

What **can't** regular expressions do? (2)

Can you write a regular expression for all strings of the form $0^k 1^k$?
i.e., same number of 0's and 1's, all 0's coming first

Can you write a regular expression for $\{0^k 1^k : k \leq 100\}$
i.e., same restrictions as above, but also only at most 200 characters total.

What **can't** regular expressions do? (2 answers)

Can you write a regular expression for all strings of the form $0^k 1^k$?

i.e., same number of 0's and 1's, all 0's coming first

No! There is no such regular expression (we'll prove it in a few weeks).

Can you write a regular expression for $\{0^k 1^k : k \leq 100\}$

i.e., same restrictions as above, but also only at ~~most 200~~ characters total.

Yes! It'll probably take you a while to write it though…

…there are a finite number of strings satisfying this description, just list them all and ∪ them together.

A 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.

Where Are We?

We're working our way up to proving theorems about computation

A statement like "there is no Java program that solves this problem"

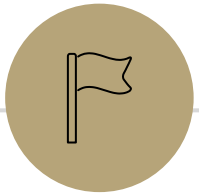
Before we get there, we've got a grab bag of background topics.

The mathematical model for programs is defining a language, so we're seeing different tools computer scientists use to do that.

The set of all inputs that will cause the program to return true.

Last time: Regular expressions: a common practical tool

Today: CFGs: a key to defining a programming language



Context Free Grammars

Another way of defining a language

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 (definitions)

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 (process)

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

Examples

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

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

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

The alphabet here is {(,)} i.e. parentheses are the characters.

$S \to AB$

$A \to 0A1 | \varepsilon$

$B \to 1B0 | \varepsilon$

$S \to 0S0 \to 01S10 \to 0101$

$S \to 0S0 \to 00S00 \to 0000$

$S \to SS$

Examples (with answers)

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

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

$S \to AB$

$A \to 0A1|\varepsilon$

$B \to 1B0|\varepsilon$

$\{0^j 1^{j+k} 0^k : j, k \geq 0\}$

Arithmetic CFG

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

Generate $(2 * x) + y$

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

$E \Rightarrow E + E$

$E \to E * E$

$E + E * E$

$\downarrow$

$2 + E * E$

$\downarrow$

$2 + 3 * E$

$\downarrow$

$2 + 3 * 4$

Arithmetic CFG (with answers)

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

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

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

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

Multiple ways of generating strings

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

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

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

What did we mean by these being different? They represent different meanings mathematically.

One says "you're adding together two numbers: 2 and (whatever 3*4 is)"

The other says "you're multiplying two numbers: (whatever 2+3 is) and 4"

Those have different meanings!

Parse Trees—remember where parentheses go

Suppose a context free grammar G generates a string x

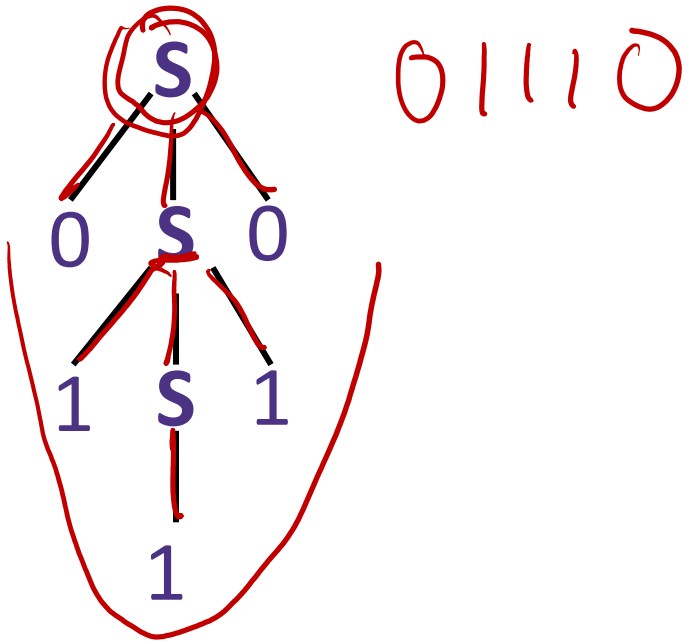
A parse tree of x for G has

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.

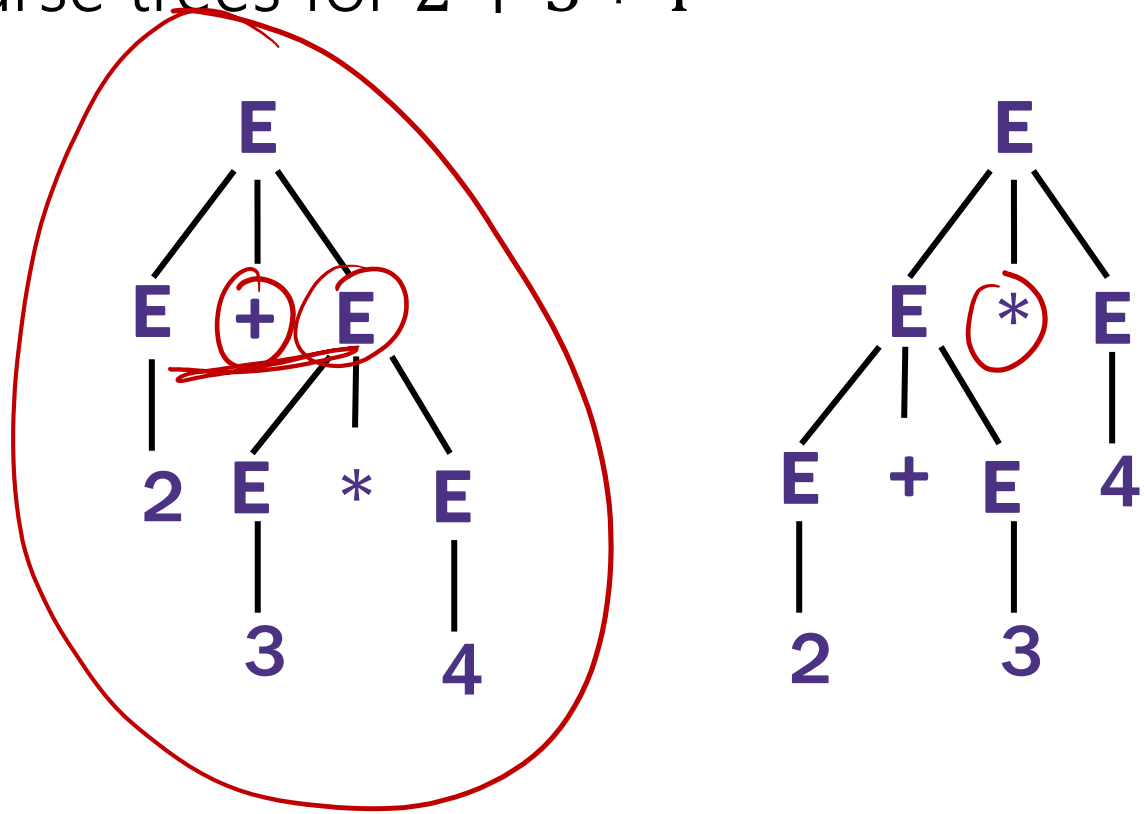
$S \to 0S0|1S1|0|1|\varepsilon$



Back to the arithmetic

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

Two parse trees for $2 + 3 * 4$



Why do we care about parsing?

$2 + 3 * 4$ can only mean one thing!

If I write these symbols in a program, we need to make sure we know which one to do.

The first grammar we saw was “ambiguous” it allows the same string to “mean” two different things.

Sometimes you can fix that!

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”

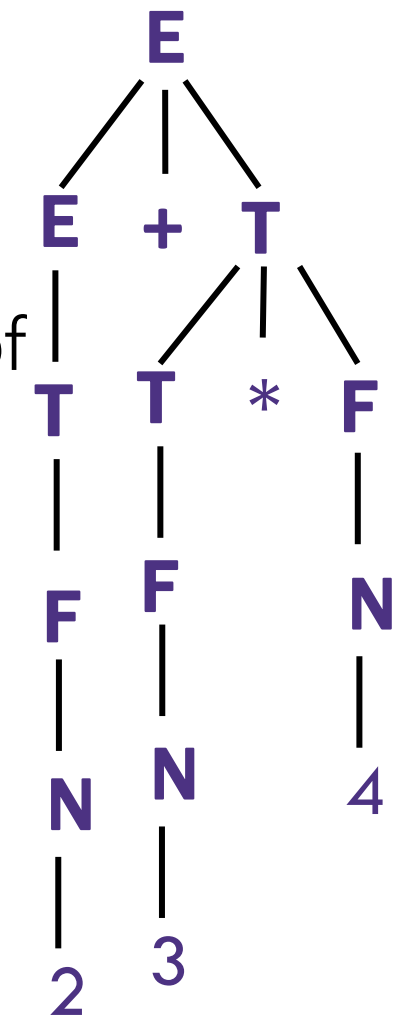
Only introduce a * sign after you’ve eliminated the possibility of introducing another + sign in that area.

$E \rightarrow T|E + T$

$T \rightarrow F|T * F$

$F \rightarrow (E)|N$

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



How do Computer Scientists use CFGs?

Most programming languages define valid programs as "strings that fit a CFG" That makes sure Java breaks down math expressions correctly! And also code like this:

```
if(i>0)
   if(j>0)
      System.out.println("hi");
   else
      System.out.println("bye");
```

```
if(i>0)
   if(j>0)
      System.out.println("hi");
else
   System.out.println("bye");
```

The else could be attached to either "if"! Java needs a rule to decide which it goes with. Java's convention makes the one on the left the intuitive whitespace.
(You as a programmer should put braces so the humans reading your code don't have to wonder!)

CFGs in practice

Used to define programming languages.

Often written in Backus-Naur Form – just different notation

Variables are <names-in-brackets> (or sometimes without)

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

→ is replaced with ::= or :

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

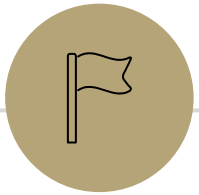
```
statement:
  ((identifier | "case" constant-expression | "default") ":")*
  (expression? ";" |
   block |
   "if" "(" expression ")" statement |
   "if" "(" expression ")" statement "else" statement |
   "switch" "(" expression ")" statement |
   "while" "(" expression ")" statement |
   "do" statement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

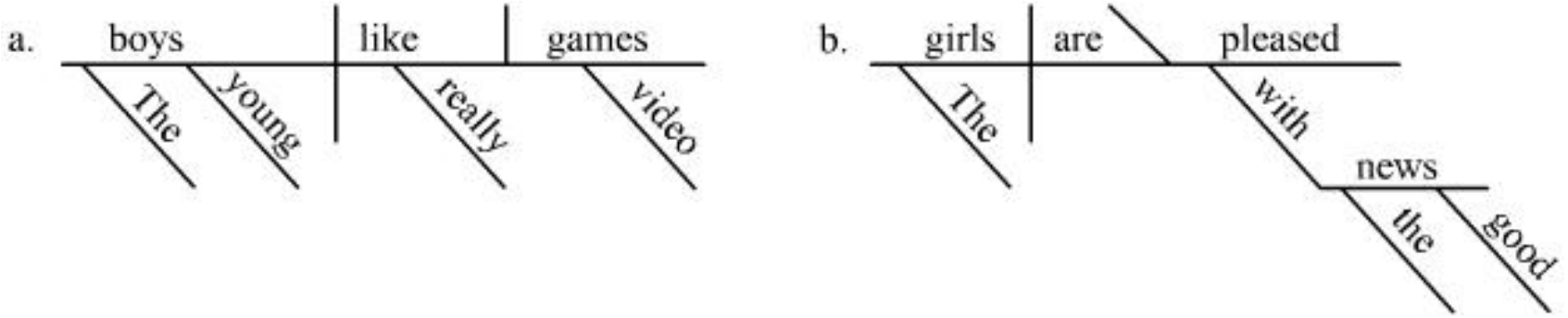


Some fun CFG applications

If we have time

Parse Trees (sentence diagramming)

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 (CFG for sentences)

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

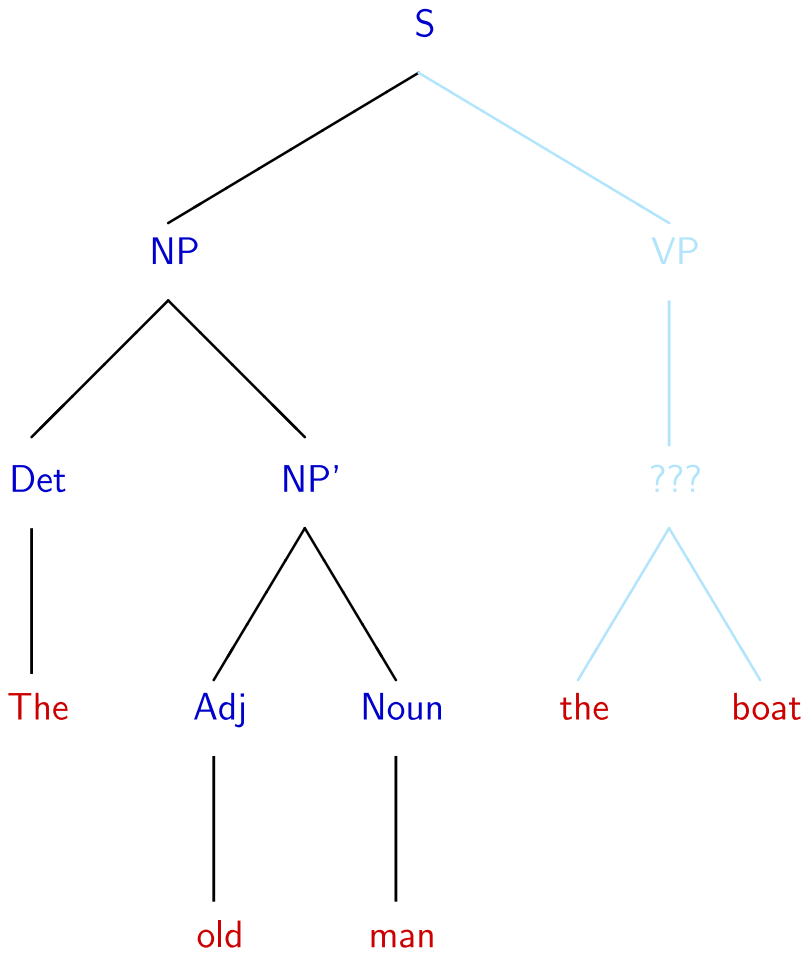
<noun phrase>::=<determiner><adjective><noun>

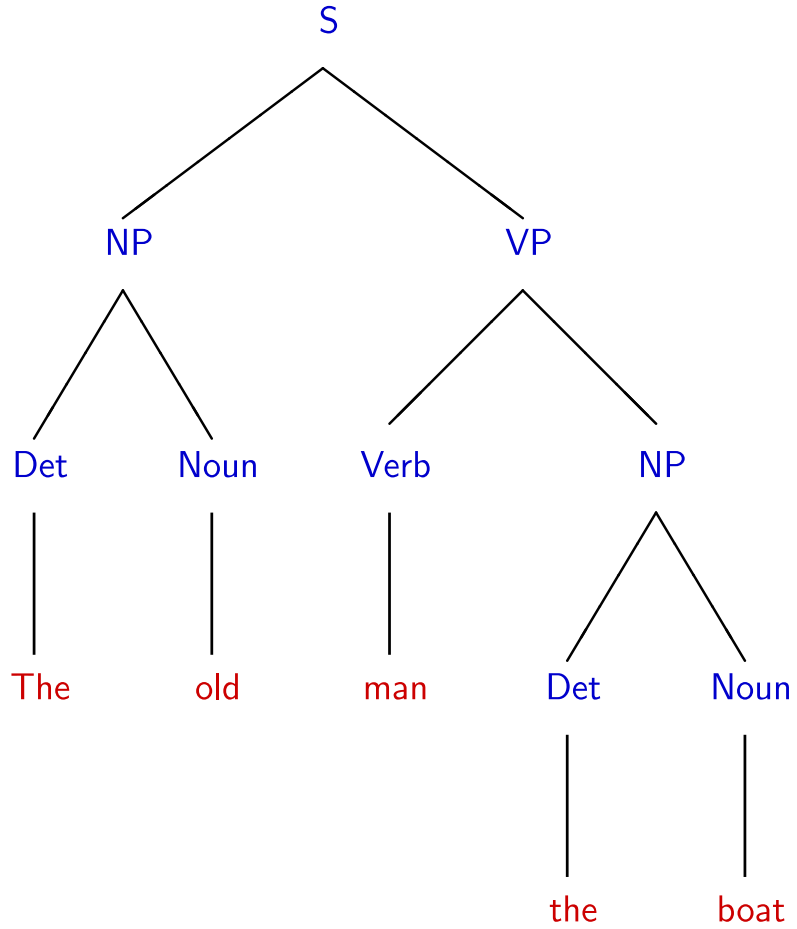
<verb phrase>::=<verb><adverb>|<verb><object>

<object>::=<noun phrase>

The old man the boat.

The old man the boat





English is ambiguous

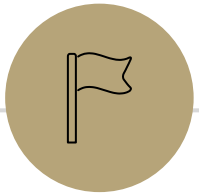
(Most of 'standard') English can be represented as a context free grammar.

It's not perfect (ask Robbie details later).

The grammar is ambiguous! That is, there are sentences which have multiple valid parsings (multiple meanings).

Can you find multiple meanings of this sentence:
"Place these 3 exercise balls on the mat at the top of the hill."

See this video



The Important Takeaways

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”

Next: (mathematical representations of) Tiny computers! And how they relate to regular expressions and CFGs.