

Context Free Grammars

CSE 311 Autumn 2023
Lecture 21

More Regular Expression 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 \cup

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

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.

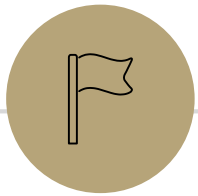
E.g. “the set of all palindromes” is not the language of any regular expression.

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

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

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

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

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

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

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

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

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

$$S \rightarrow AB$$

$$A \rightarrow 0A1|\varepsilon$$

$$B \rightarrow 1B0|\varepsilon$$

Examples

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

The set of all binary palindromes

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

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

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

Balanced parentheses

$$S \rightarrow AB$$

$$A \rightarrow 0A1|\varepsilon$$

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

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

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

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

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

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

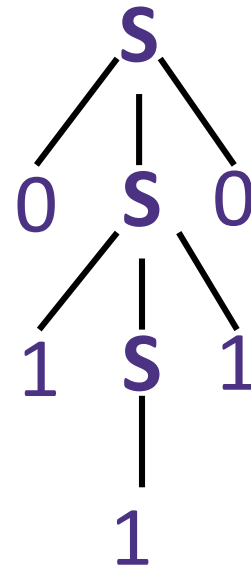
A parse tree of x for G has

Rooted at S (start symbol)

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

Reading the leaves from left to right gives x .

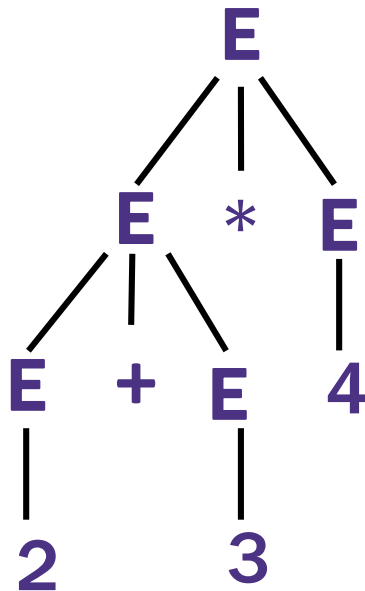
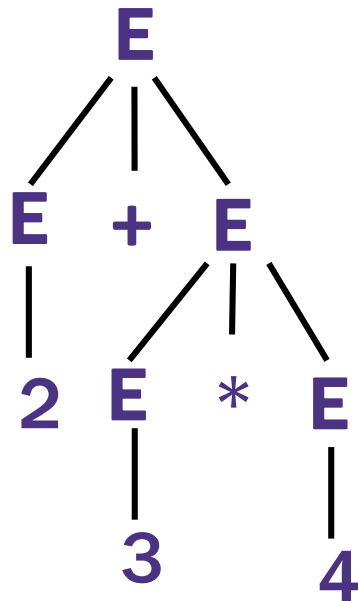
$$S \rightarrow 0S0 \mid 1S1 \mid 0 \mid 1 \mid \varepsilon$$



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 $2 + 3 * 4$



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"

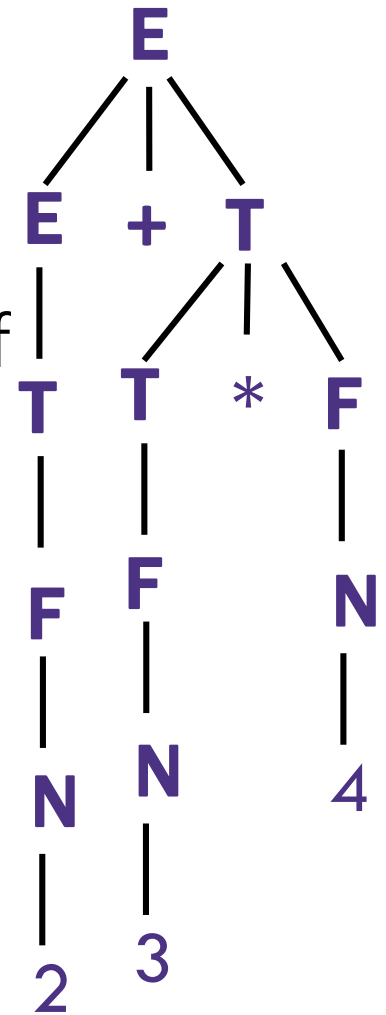
Only introduce a * sign after you've eliminated the possibility of introducing another + sign in that area.

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

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

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

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



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 (k>1)
    System.out.println("bye");
```

```
if (i>0)
    if (j>0)
        System.out.println("hi");
else (k>1)
    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 ::=)

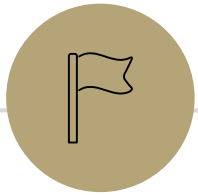
```
statement:
  ((identifier | "case" constant-expression | "default") ":")*
  (expression? ";" |
  block |
  "if" "(" expression ")" statement |
  "if" "(" expression ")" statement "else" statement |
  "switch" "(" expression ")" statement |
  "while" "(" expression ")" statement |
  "do" statement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

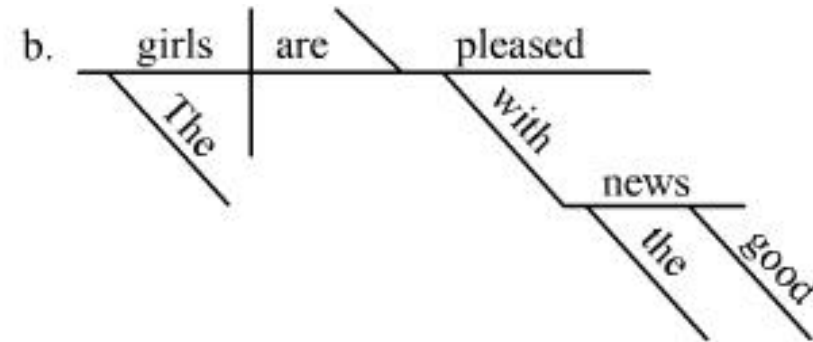
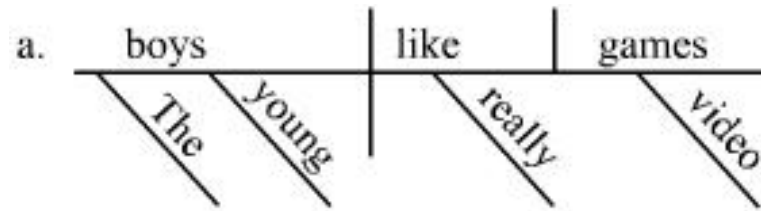


Some fun CFG applications

If we have time

Parse Trees

Remember diagramming sentences in middle school?



$\langle \text{sentence} \rangle ::= \langle \text{noun phrase} \rangle \langle \text{verb phrase} \rangle$

$\langle \text{noun phrase} \rangle ::= \langle \text{determiner} \rangle \langle \text{adjective} \rangle \langle \text{noun} \rangle$

$\langle \text{verb phrase} \rangle ::= \langle \text{verb} \rangle \langle \text{adverb} \rangle | \langle \text{verb} \rangle \langle \text{object} \rangle$

$\langle \text{object} \rangle ::= \langle \text{noun phrase} \rangle$

Parse Trees

$\langle \text{sentence} \rangle ::= \langle \text{noun phrase} \rangle \langle \text{verb phrase} \rangle$

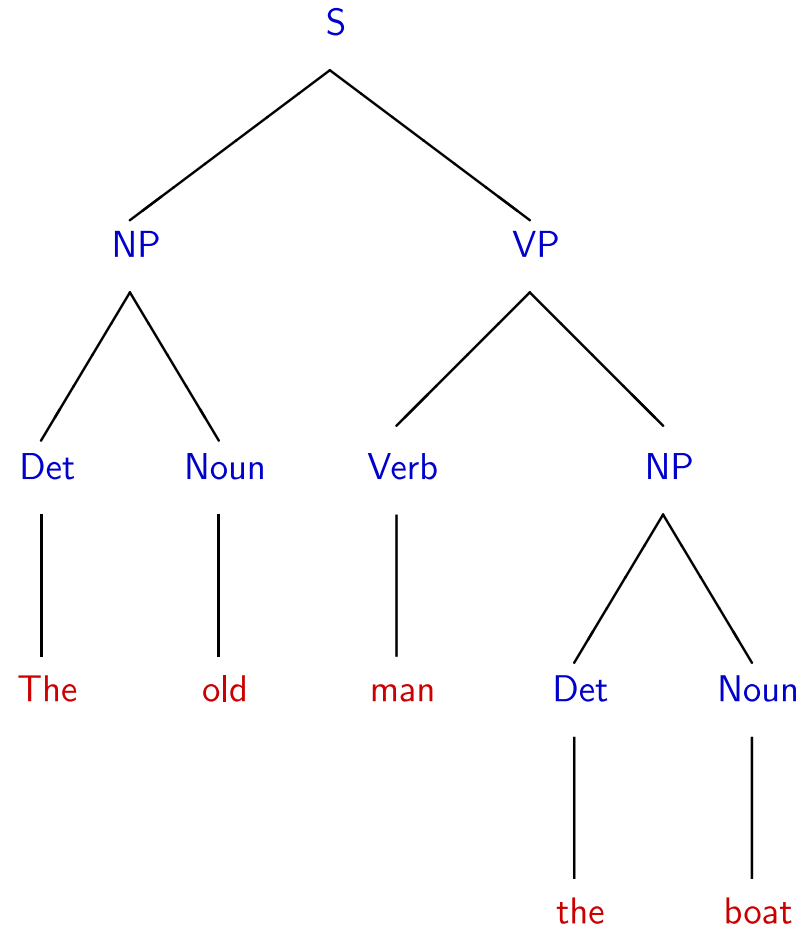
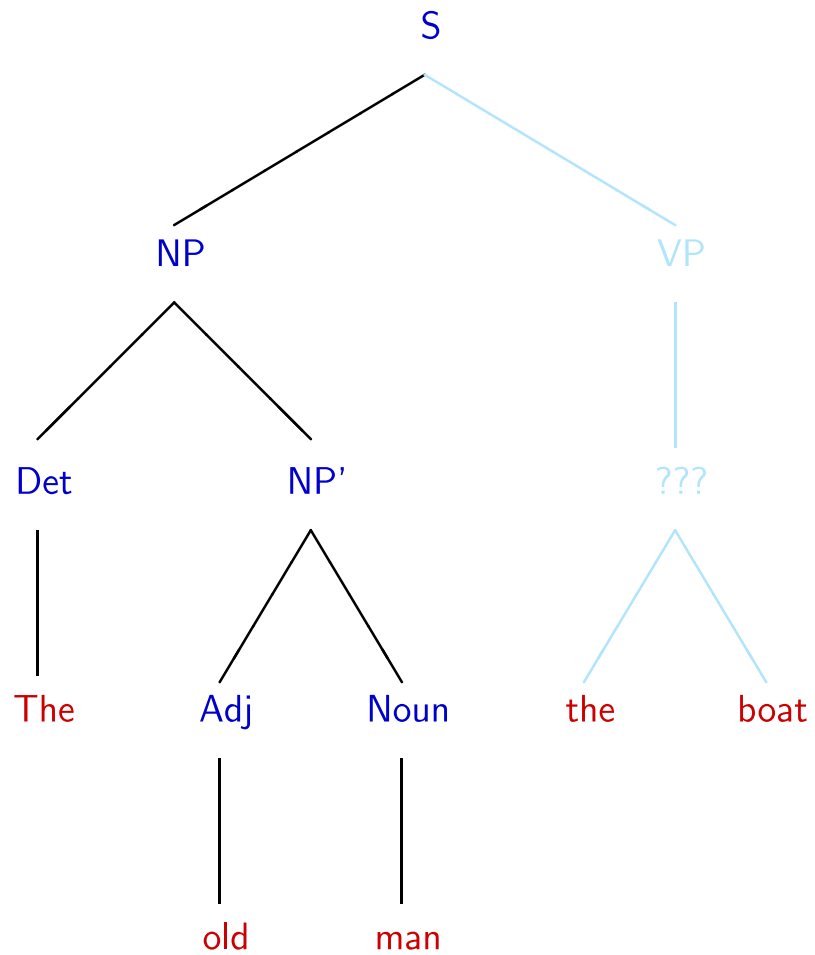
$\langle \text{noun phrase} \rangle ::= \langle \text{determiner} \rangle \langle \text{adjective} \rangle \langle \text{noun} \rangle$

$\langle \text{verb phrase} \rangle ::= \langle \text{verb} \rangle \langle \text{adverb} \rangle \mid \langle \text{verb} \rangle \langle \text{object} \rangle$

$\langle \text{object} \rangle ::= \langle \text{noun phrase} \rangle$

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!

$\{0^k 1^j 2^k 3^j \mid j, k \geq 0\}$ 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 Week, Two more tools for our toolbox (relations, graphs)

After that, (mathematical representations of) Tiny computers! And how they relate to regular expressions and CFGs.