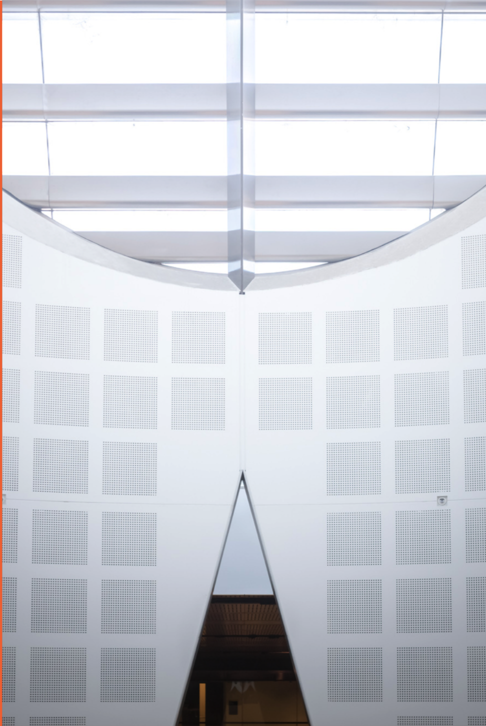
COMP2022|2922 Models of Computation

Context-free Grammars

Sasha Rubin

September 18, 2022

THE UNIVERSITY OF SYDNEY



Agenda

Context-free grammars

1. Syntax, semantics
2. Derivations
3. Parse trees
4. Ambiguity
5. Why are they called context-free grammars?

Limitations of Regular Expressions

- We saw that regular expressions are useful for basic pattern matching, e.g., recognising keywords.
- But they are limited.
- The basic difficulty is handling arbitrary nesting.
  - e.g., $1 + (1 + 1)$ and $1 + (1 + (1 + 1))$ and …
  - needed by parsers

Context-free grammars in a nutshell

A grammar is a set of rules which generates a language.

- The rules are used to derive strings (in contrast, regular expressions are used to match strings).
- The rules are a recursive description of the strings.
- Grammars naturally describe the hierarchical structure of most programming languages.
- Grammars also form the basis for translating between different representations of programs, see Tutorial.

Context-free grammars

Program Syntax

```
statements:  statement+
statement :  compound_stmt | simple_stmt
```

Document Description Definition

```
<!ELEMENT NEWSPAPER (ARTICLE+)>
<!ELEMENT ARTICLE (STORY | ADVERT) >
```

Our Syntax

$$S \to TS$$
$$T \to c \mid d$$

Context-free grammars: Example

$$S \to aSb$$
$$S \to T$$
$$T \to c$$

To generate/derive a string:

1. Write down the start variable. It is the variable on the left-hand side of the top rule, unless specified otherwise.
2. Find a variable that is written down and a rule that starts with that variable. Replace the written down variable with the right-hand side of that rule.
3. Repeat step 2 until no variables remain.

Context-free grammars

A context-free grammar consists of four items:

1. Variables, aka non-terminals: $A, B, C, \ldots$
2. Terminals: $a, b, c, \ldots, 0, 1, 2, \cdots, +, -, (,), \cdots$
3. Rules: $A \to u$ where u is a string of variables and terminals.
4. Start variable: usually S, or the first one listed.

$$(V, \Sigma, R, S)$$

Context-free grammars: Example

- Variables S, T
- Terminals a, b, c
- Start variable S
- Rules:

$$S \to aSb \tag{1}$$

$$S \to T \tag{2}$$

$$T \to c \tag{3}$$

Context-free grammars: Example

- Variables E
- Terminals $a, b, c, -$
- Start variable E
- Rules:

$$E \to E - E \tag{1}$$

$$E \to a \tag{2}$$

$$E \to b \tag{3}$$

$$E \to c \tag{4}$$

Example derivations:

- One step of a derivation is written $\Rightarrow$
  - read "yields"
- Zero or more steps are written $\Rightarrow^*$.
  - read "derives"
- The set of strings over Σ that are derived from the start variable is called the language generated by G, denoted $L(G)$.

$$L(G) = \{u \in \Sigma^* : S \Rightarrow^* u\}$$

Language of a CFG

What is the language generated by the following grammar?

$$
\begin{aligned}
S &\to aSb \\
S &\to T \\
T &\to c
\end{aligned}
$$

Vote now! (on mentimeter)

1. All strings over alphabet $\{a, b, c, S, T\}$.
2. All strings over alphabet $\{a, b, c\}$ that match the regular expression a^*cb^*
3. All strings over alphabet $\{a, b, c\}$ of the form a^ncb^n where $n \geq 0$.

Language of a CFG

What is the language generated by the following grammar?

$$
\begin{aligned}
E &\to E + E \\
E &\to 0 \\
E &\to 1
\end{aligned}
$$

Vote now! (on mentimeter)

1. All strings over the alphabet $\{0, 1, +\}$ that represent arithmetic expressions using the symbols for addition and the numbers 0 and 1.
2. All natural numbers.
3. All binary strings over the alphabet $\{0, 1\}$.

Shorthand notation

A variable can have many rules:

$$S \to aSb$$
$$S \to T$$

They can be written together:

$$S \to aSb \mid T$$

Designing CFGs

1. Variables generate substrings with similar properties.
   - Think of the variables as storing information, or as having meaning.
2. Think recursively.
   - How can a string in the language be built from smaller strings in the language?
   - Make sure you cover all cases.

Designing CFGs

Design a grammar that generates the language of binary strings of the form $0^n 1^m 0^n$ for $n, m \geq 0$.

Variables generate substrings with similar properties

$$
\begin{aligned}
S &\to 0S0 \mid X \\
X &\to 1X \mid \epsilon
\end{aligned}
$$

– The variable X generates the language $L(1^*)$.

Designing CFGs

Design a grammar that generates the language of binary strings that are *palindromes*, i.e., reads the same forwards as backwards.

Think recursively

1. Base case: 0, 1, and ϵ are palindromes.
2. Recursive case: if u is a palindrome, then $0u0$ and $1u1$ are palindromes.
   Why are there no other cases?

Here is a grammar:

$$S \to 0 \mid 1 \mid \epsilon$$
$$S \to 0S0 \mid 1S1$$

Designing CFGs

Design a grammar that generates the language of binary strings with the same number of 0's and 1's.

Think recursively

1. Base case: ϵ has the same number of 0's and 1's, i.e., none.
2. Recursive case: if u, v has the same number of 0's and 1's, then so do $0u1v$ and $1u0v$.
   Why are there no other cases?

Here is a grammar:

$$S \to \epsilon$$
$$S \to 0S1S \mid 1S0S$$

Language of a CFG

The tutorial asks you to give a grammar for the set of strings over terminal symbols (and) in which the parentheses are well-balanced.

This is probably the single most important example of a CFG since, e.g., arbitrary expressions, programming languages, usually require balanced parentheses.

Context-Free Languages

Definition

A language is context-free if it is generated by a CFG.

Easy facts.

- The union of two CFL is also context-free.

  Why? Just add a new rule $S \to S_1 \mid S_2$ where S_i is the start symbol of grammar i.
- The concatenation of two CFL is also context-free

  Why? Just add a new rule $S \to S_1 S_2$
- The star closure of a CFL is also context-free

  Why? Just add a new rule $S \to S S_1 \mid \epsilon$

Context-Free Languages

1. Directly design a grammar for it.
2. Write it as a union, concatenation or star of other context-free languages.

Designing context-free grammars

Show that $L(a^* \cup b^*)$ is context-free.

- Here is a grammar for $L(a^*)$:

$$S_1 \to S_1 a \mid \epsilon$$

- Here is a grammar for $L(b^*)$:

$$S_2 \to S_2 b \mid \epsilon$$

- So here is a grammar for $L(a^* \cup b^*) = L(a^*) \cup L(b^*)$:

$$\boxed{\begin{aligned} S &\to S_1 \mid S_2 \\ S_1 &\to S_1 a \mid \epsilon \\ S_2 &\to S_2 b \mid \epsilon \end{aligned}}$$

Designing context-free grammars

Show that $L(a^*b^*)$ is context-free.

- Here is a grammar for $L(a^*)$:

$$S_1 \to S_1 a \mid \epsilon$$

- Here is a grammar for $L(b^*)$:

$$S_2 \to S_2 b \mid \epsilon$$

- So here is a grammar for $L(a^*b^*) = L(a^*)L(b^*)$:

$$\boxed{\begin{aligned} S &\to S_1 S_2 \\ S_1 &\to S_1 a \mid \epsilon \\ S_2 &\to S_2 b \mid \epsilon \end{aligned}}$$

Designing context-free grammars

Show that $L((aa \,|\, bb)^*)$ is context-free.

Note that this is the language $L(aa \,|\, bb)^*$.

- Here is a grammar for $L(aa \,|\, bb)$:

$$S_1 \to aa \mid bb$$

- So here is a grammar for $L(aa \,|\, bb)^*$:

$$\boxed{\begin{aligned} S &\to S_1 S \mid \epsilon \\ S_1 &\to aa \mid bb \end{aligned}}$$

Regular expressions and Context-free Grammars

- Note that although CFGs have $|$, this is shorthand for multiple rules.
- Note that CFGs do not mention $*$, but we have seen how we can simulate/mimic it.

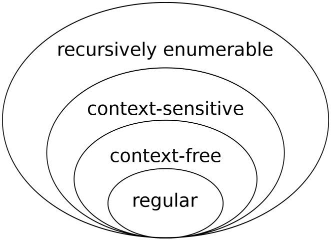
- We already know that there is a context-free language that is not regular.
- In the tutorial you will show how to convert a regular expression R into a CFG G such that $L(R) = L(G)$.
- *i.e.,* CFGs are strictly more expressive than regular expressions

Why are they called "context-free"?

The Chomsky Hierarchy consists of 4 classes of grammars, depending on the type of production rules that they allow:

Type 0 (recursively enumerable)	$z \to v$
Type 1 (context-sensitive)	$uAv \to uzv$
Type 2 (context-free)	$A \to u$
Type 3 (regular)	$A \to aB$ and $A \to a$

– u, v, z string of variables and terminals, z not empty.



Good to know

- $\{ww : w \in \{0,1\}^*\}$ is not context-free (the proof uses a pumping argument, see Sipser Chapter 2.3)
- Let's see that it is context-sensitive ($uAV \to uzv$)

$$
\begin{aligned}
S &\to aAS \mid bBS \mid T \\
Aa &\to aA \\
Ab &\to bA \\
Ba &\to aB \\
Bb &\to bB \\
AT &\to Ta \\
BT &\to Tb \\
T &\to \epsilon
\end{aligned}
$$

Derive $aabaab$:

$$
\begin{aligned}
&S \Rightarrow aAS \Rightarrow aAaAS \Rightarrow aAaAbBS \Rightarrow aAaAbBT \\
&\Rightarrow aAabABT \Rightarrow aaAbABT \Rightarrow aabAABT \\
&\Rightarrow aabAATb \Rightarrow aabATab \Rightarrow aabTaab \Rightarrow aabaab.
\end{aligned}
$$

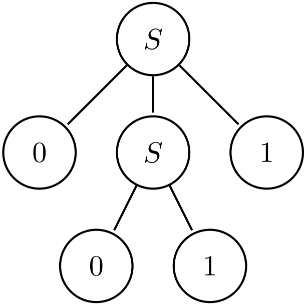
Parsing

The problem of *parsing* is determining *how* the grammar generates a given string.

Parse Tree

A parse tree (aka derivation tree) is a tree labeled by variables and terminal symbols of the CFG

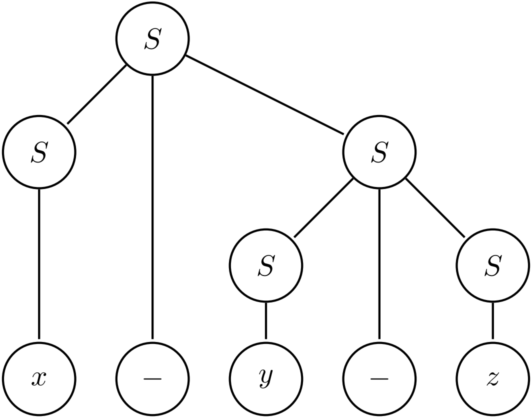
- the root is labeled by the start variable
- each interior node is labeled by a variable
- each leaf node is labeled by a terminal or ϵ
- the children of a node labeled X are labeled by the right hand side of a rule $X \to u$, in order.



- Example parse tree for 0011 in $S \to 0S1 \mid 01$
- A traversal of the leaf nodes retrieves the string

Parse Tree

The parse tree gives the "meaning" of a string.



$$S \to S - S$$
$$S \to x \mid y \mid z$$

This parse tree says that the expression means "$x - (y - z)$" rather than "$((x - y) - z)$".

Natural Language Processing (NLP)

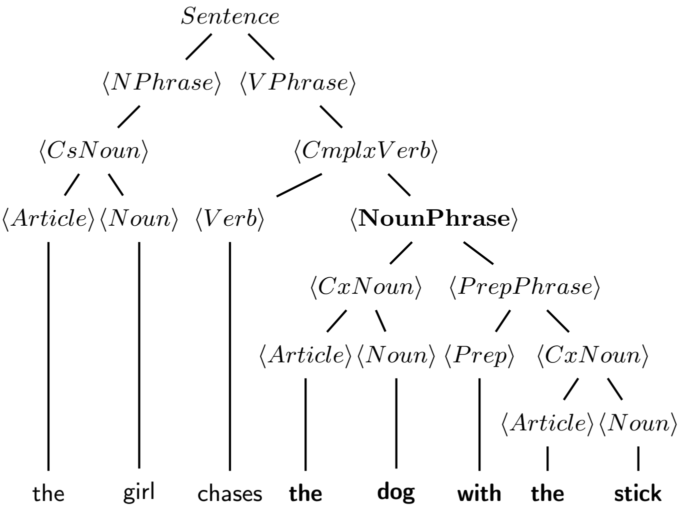
$$\begin{aligned}
\langle Sentence\rangle &\to \langle NounPhrase\rangle\ \langle VerbPhrase\rangle \\
\langle NounPhrase\rangle &\to \langle ComplexNoun\rangle \\
\langle NounPhrase\rangle &\to \langle ComplexNoun\rangle\ \langle PrepPhrase\rangle \\
\langle VerbPhrase\rangle &\to \langle ComplexVerb\rangle \mid \langle ComplexVerb\rangle\ \langle PrepPhrase\rangle \\
\langle PrepPhrase\rangle &\to \langle Prep\rangle\ \langle ComplexNoun\rangle \\
\langle ComplexNoun\rangle &\to \langle Article\rangle\ \langle Noun\rangle \\
\langle ComplexVerb\rangle &\to \langle Verb\rangle \mid \langle Verb\rangle\ \langle NounPhrase\rangle \\
\langle Article\rangle &\to \text{a} \mid \text{the} \\
\langle Noun\rangle &\to \text{girl} \mid \text{dog} \mid \text{stick} \mid \text{ball} \\
\langle Verb\rangle &\to \text{chases} \mid \text{sees} \\
\langle Prep\rangle &\to \text{with}
\end{aligned}$$

- Terminals are the lower-case English alphabet
- For variables, we may use the notation $\langle Noun\rangle$ instead of simply N. This is only for readability.

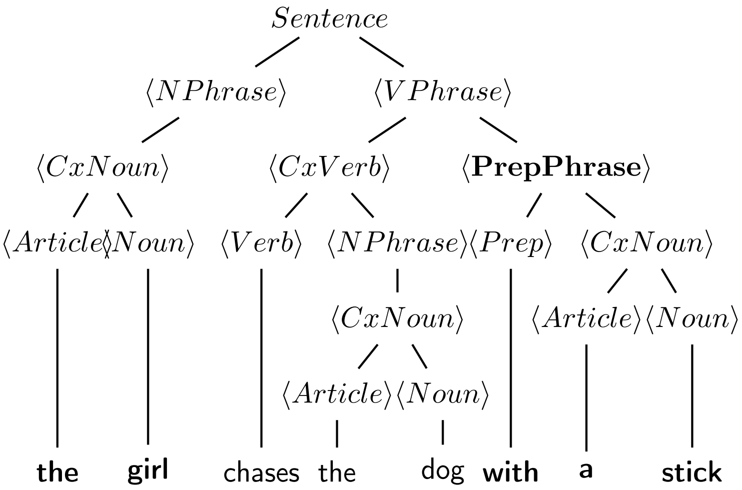
Ambiguity

– The string "the girl chases the dog with the stick" can be derived in this grammar.
– But it has (at least) two parse-trees depending on who has the stick!
– ... the dog vs the girl

First parse tree



Second parse tree



Ambiguous grammars

Definition

- A string is ambiguous on a given grammar if it has at least two different parse trees.
- A grammar is ambiguous if it derives at least one ambiguous string.

So, the previous two grammars are ambiguous.

Ambiguous strings

Is there a way to see if a string is ambiguous without drawing parse trees?

- A derivation is called leftmost if it always derives the leftmost symbol first.
- Each parse tree corresponds to one leftmost derivation.
- So, a string is ambiguous if it has at least two leftmost derivations.
- The same two statements hold with "rightmost" instead of "leftmost"

Ambiguous strings

- Ambiguity = several meanings for the same sentence.
- "The girl chases the dog with a stick"
- Who has the stick?

"The girl chases the dog with a stick" has *two leftmost derivations*.

$$
\begin{aligned}
\langle Sentence \rangle &\Rightarrow^* \text{the girl } \langle VerbPhrase \rangle \\
&\Rightarrow \text{ the girl } \langle Verb \rangle \langle NounPhrase \rangle \\
&\Rightarrow^* \text{the girl chases the dog with a stick}
\end{aligned}
$$

$$
\begin{aligned}
\langle Sentence \rangle &\Rightarrow^* \text{the girl } \langle VerbPhrase \rangle \\
&\Rightarrow \text{ the girl } \langle ComplexVerb \rangle \ \langle PrepPhrase \rangle \\
&\Rightarrow^* \text{the girl chases the dog with a stick}
\end{aligned}
$$

Is this grammar ambiguous?

$$E \to E - E$$
$$E \to a \mid b \mid c$$

Rightmost derivations of $a - b - c$:

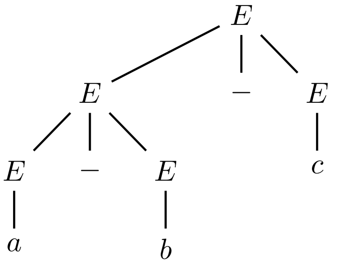
$$\begin{aligned} E &\Rightarrow E - E \\ &\Rightarrow E - c \\ &\Rightarrow E - E - c \\ &\Rightarrow E - b - c \\ &\Rightarrow a - b - c \end{aligned}$$

$$\begin{aligned} E &\Rightarrow E - E \\ &\Rightarrow E - E - E \\ &\Rightarrow E - E - c \\ &\Rightarrow E - b - c \\ &\Rightarrow a - b - c \end{aligned}$$

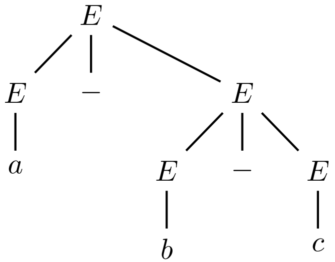
Is this grammar ambiguous?

$$E \rightarrow E - E$$
$$E \rightarrow a \mid b \mid c$$

Rightmost derivations of $a - b - c$:



i.e. "$(a - b) - c$"



i.e. "$a - (b - c)$"

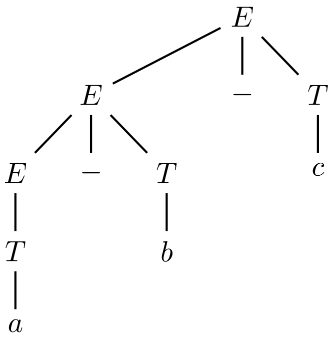
Removing ambiguity

- Suppose we want $a - b - c$ to always mean $(a - b) - c$?
- Introduce a new nonterminal symbol T:

$$
\begin{aligned}
E &\to E - T \mid T \\
T &\to a \mid b \mid c
\end{aligned}
$$

- Now the only rightmost derivation of $a - b - c$ is:

$$
\begin{aligned}
E &\Rightarrow E - T \\
&\Rightarrow E - c \\
&\Rightarrow E - T - c \\
&\Rightarrow E - b - c \\
&\Rightarrow T - b - c \\
&\Rightarrow a - b - c
\end{aligned}
$$



Next week

Next week we study a classic parsing algorithm:

- Input is a grammar G and a string u over the alphabet.
- Output is a derivation of u in G, or "u is not derivable in G".