CONTEXT-FREE GRAMMAR

INTRODUCTION

- Context-free grammars are a more powerful method of describing languages.
- There grammars can describe certain features that have a recursive structure, which makes them useful in a variety of applications
- An important application of context-free grammars occurs in the specification and compilation of programming languages. Most compilers and interpreters contain a component called a parser that extracts the meaning of a program prior to generating the compiled code or performing the interpreted execution

INTRODUCTION

- The following is an example of a context-free grammar, which we call G_1.
  - A -> 0A1
  - A -> B
  - B -> #
- A grammar consists of a collection of substitution rules, also called ***productions***.
- Each rule appears as a line in the grammar, comprising a ***symbol*** and a ***string*** separated by ***an arrow***.
- The symbol is called a ***variable***. The string consists of ***variables*** and other symbols called ***terminals***.

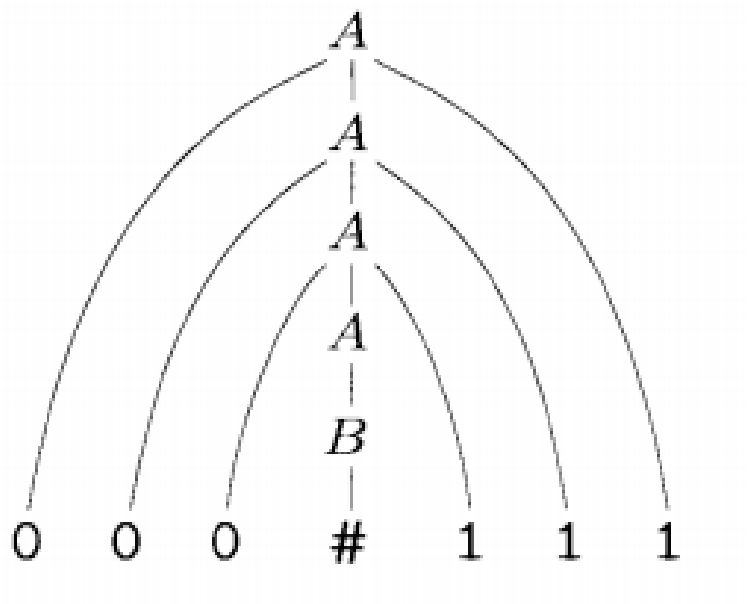
INTRODUCTION

- You use a grammar to describe a language by generating each string of that language in the following manner.
    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
- For example, grammar G_1 generates the string 000#111.
- The sequence of substitutions to obtain a string is called a ***derivation***.
- A derivation of string 000#111 in grammar G_1 is

    A => 0A1 => 00A11 => 000A111 => 000B111 => 000#111

INTRODUCTION

- You may also represent the same information pictorially with a parse tree. An example of a parse tree is shown in figure 1 below



- All strings generated in this way constitute the language of the grammar.
- We write $L(G_1)$ for the language of grammar G_1

INTRODUCTION

- Some experimentation with the grammar G_1 shows us that

$$L(G_1) \text{ is } \{0^n \# 1^n \mid n \geq 0\}$$

- Any language that can be generated by some context-free grammar is called a context-free language (CFL).
- For convenience when presenting a context-free grammar, we abbreviate several rules with the same left-hand variable, such as A →0A0 | B 0141 and A → B, into a single line →0A0 | B , using the symbol "|" as an "or."

INTRODUCTION

- The following is a second example of a context-free grammar, called G_2, which describes a fragment of the English language.

⟨SENTENCE⟩ → ⟨NOUN-PHRASE⟩⟨VERB-PHRASE⟩

⟨NOUN-PHRASE⟩ → ⟨CMPLX-NOUN⟩ | ⟨CMPLX-NOUN⟩⟨PREP-PHRASE⟩

⟨VERB-PHRASE⟩ → ⟨CMPLX-VERB⟩ | ⟨CMPLX-VERB⟩⟨PREP-PHRASE⟩

⟨PREP-PHRASE⟩ → ⟨PREP⟩⟨CMPLX-NOUN⟩

⟨CMPLX-NOUN⟩ → ⟨ARTICLE⟩⟨NOUN⟩

⟨CMPLX-VERB⟩ → ⟨VERB⟩ | ⟨VERB⟩⟨NOUN-PHRASE⟩

⟨ARTICLE⟩ → `a` | `the`

⟨NOUN⟩ → `boy` | `girl` | `flower`

⟨VERB⟩ → `touches` | `likes` | `sees`

⟨PREP⟩ → `with`

INTRODUCTION

- Grammar G_2 has
- 10 variables (the capitalized grammatical terms written inside brackets);
- 27 terminals (the standard English alphabet plus a space character) and
- 18 rules.
- Strings in $L(G_2)$ include

a boy sees

the boy sees a flower

a girl with a flower likes the boy

INTRODUCTION

- Each of the above strings has a derivation in grammar G_2.
- The following is a derivation of the first string on this list.

⟨SENTENCE⟩ ⇒ ⟨NOUN-PHRASE⟩⟨VERB-PHRASE⟩
⇒ ⟨CMPLX-NOUN⟩⟨VERB-PHRASE⟩
⇒ ⟨ARTICLE⟩⟨NOUN⟩⟨VERB-PHRASE⟩
⇒ a ⟨NOUN⟩⟨VERB-PHRASE⟩
⇒ a boy ⟨VERB-PHRASE⟩
⇒ a boy ⟨CMPLX-VERB⟩
⇒ a boy ⟨VERB⟩
⇒ a boy sees

FORMAL DEFINITION OF CFG

A ***context-free grammar*** is a 4-tuple (V, Σ, R, S), where

1. V is a finite set called the ***variables***,
2. Σ is a finite set, disjoint from V, called the ***terminals***,
3. R is a finite set of ***rules***, with each rule being a variable and a string of variables and terminals, and
4. $S \in V$ is the start variable.

If u, v, and w are strings of variables and terminals, and $A \to w$ is a rule of the grammar, we say that uAv ***yields*** uwv, written $uAv \Rightarrow uwv$. Say that u ***derives*** v, written $u \overset{*}{\Rightarrow} v$, if $u = v$ or if a sequence $u_1, u_2, \ldots, u_k$ exists for $k \geq 0$ and

$$u \Rightarrow u_1 \Rightarrow u_2 \Rightarrow \ldots \Rightarrow u_k \Rightarrow v.$$

FORMAL DEFINITION OF CFG

The ***language of the grammar*** is $\{w \in \Sigma^* \mid S \stackrel{*}{\Rightarrow} w\}$.

In grammar G_1, $V = \{A, B\}$, $\Sigma = \{0, 1, \#\}$, $S = A$, and R is the collection of the three rules appearing on page 100. In grammar G_2,

$$V = \{\langle \text{SENTENCE} \rangle, \langle \text{NOUN-PHRASE} \rangle, \langle \text{VERB-PHRASE} \rangle,$$
$$\langle \text{PREP-PHRASE} \rangle, \langle \text{CMPLX-NOUN} \rangle, \langle \text{CMPLX-VERB} \rangle,$$
$$\langle \text{ARTICLE} \rangle, \langle \text{NOUN} \rangle, \langle \text{VERB} \rangle, \langle \text{PREP} \rangle\},$$

and $\Sigma = \{\text{a}, \text{b}, \text{c}, \ldots, \text{z}, \text{“ ”}\}$. The symbol “ ” is the blank symbol, placed invisibly after each word (a, boy, etc.), so the words won't run together.

Example 1

Consider grammar $G_3 = (\{S\}, \{\mathtt{a}, \mathtt{b}\}, R, S)$. The set of rules, R, is

$$S \rightarrow \mathtt{a}S\mathtt{b} \mid SS \mid \varepsilon.$$

This grammar generates strings such as abab, aaabbb, and aababb. You can see more easily what this language is if you think of a as a left parenthesis "(" and b as a right parenthesis ")". Viewed in this way, $L(G_3)$ is the language of all strings of properly nested parentheses.

Example 2

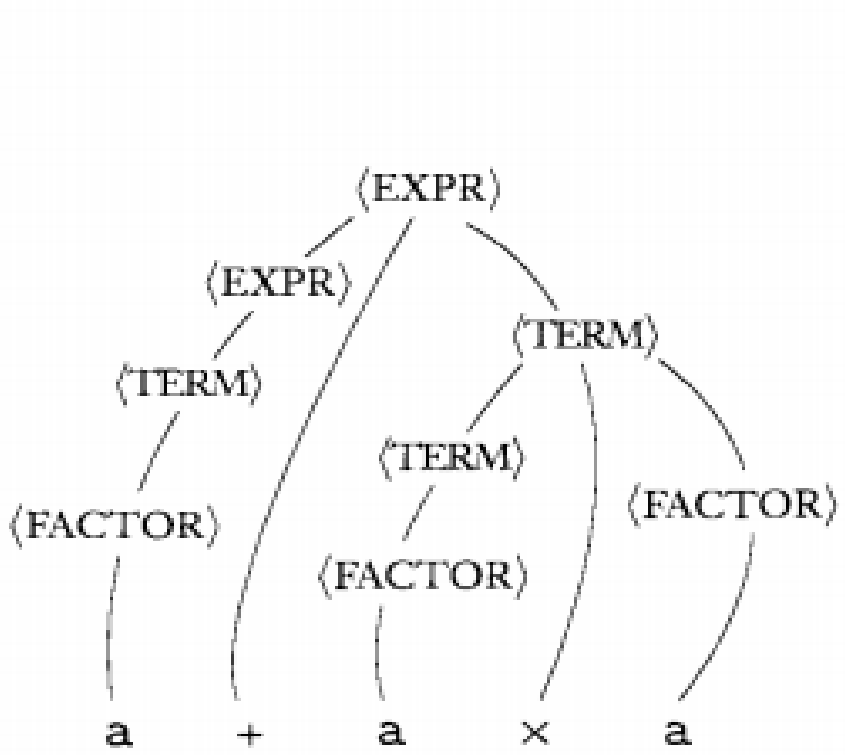
Consider grammar $G_4 = (V, \Sigma, R, \langle \text{EXPR} \rangle)$.

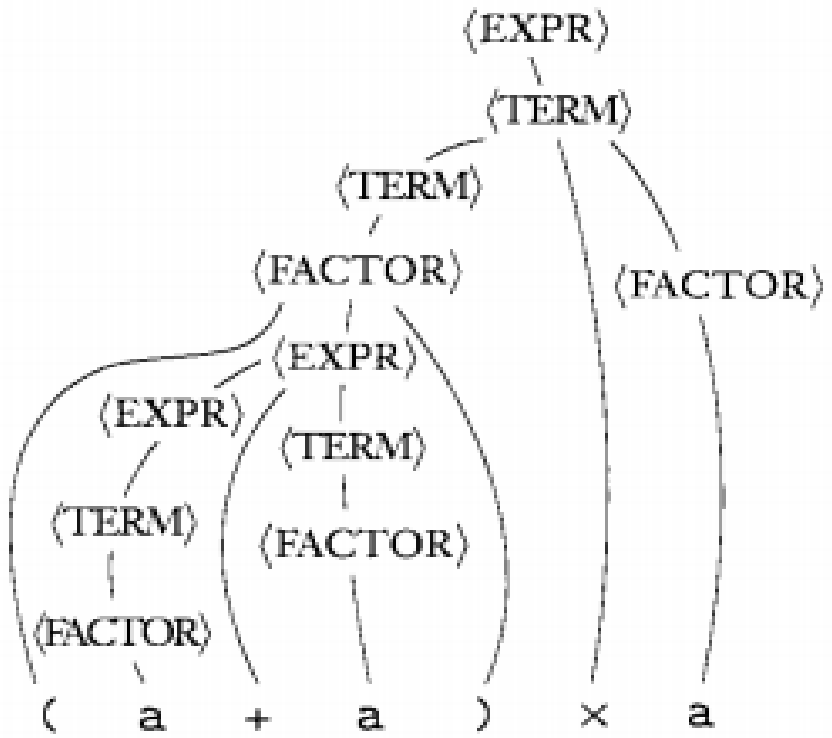
V is $\{\langle \text{EXPR} \rangle, \langle \text{TERM} \rangle, \langle \text{FACTOR} \rangle\}$ and Σ is {a, +, x, (,)}. The rules are

$$\begin{aligned} \langle \text{EXPR} \rangle &\rightarrow \langle \text{EXPR} \rangle + \langle \text{TERM} \rangle \mid \langle \text{TERM} \rangle \\ \langle \text{TERM} \rangle &\rightarrow \langle \text{TERM} \rangle \text{x} \langle \text{FACTOR} \rangle \mid \langle \text{FACTOR} \rangle \\ \langle \text{FACTOR} \rangle &\rightarrow (\langle \text{EXPR} \rangle) \mid \text{a} \end{aligned}$$

The two strings a+axa and (a+a)xa can be generated with grammar G_4. The parse trees are shown in the following figure.

Example 2





Parse tress for the strings a + a x a and (a + a) x a

Example 2

- Grammar G_4 of example 2 describes a fragment of a programming language concerned with arithmetic expressions.
- Observe how the parse trees "group" the operations.
- The tree for a + a x a groups the x operator and its operands (the second two a's) as one operand of the + operator.
- In the tree for (a + a) x a, the grouping is reversed.
- These groupings fit the standard precedence of multiplication before addition and the use of parentheses to override the standard precedence.
- Grammar G_4 is designed to capture these precedence relations

DESIGNING CONTEXT-FREE GRAMMARS

- Grammar G_4 of example 2 describes a fragment of a programming language concerned with arithmetic expressions.
- Observe how the parse trees "group" the operations.
- The tree for a + a x a groups the x operator and its operands (the second two a's) as one operand of the + operator.
- In the tree for (a + a) x a, the grouping is reversed.
- These groupings fit the standard precedence of multiplication before addition and the use of parentheses to override the standard precedence.
- Grammar G_4 is designed to capture these precedence relations

AMBIGUITY

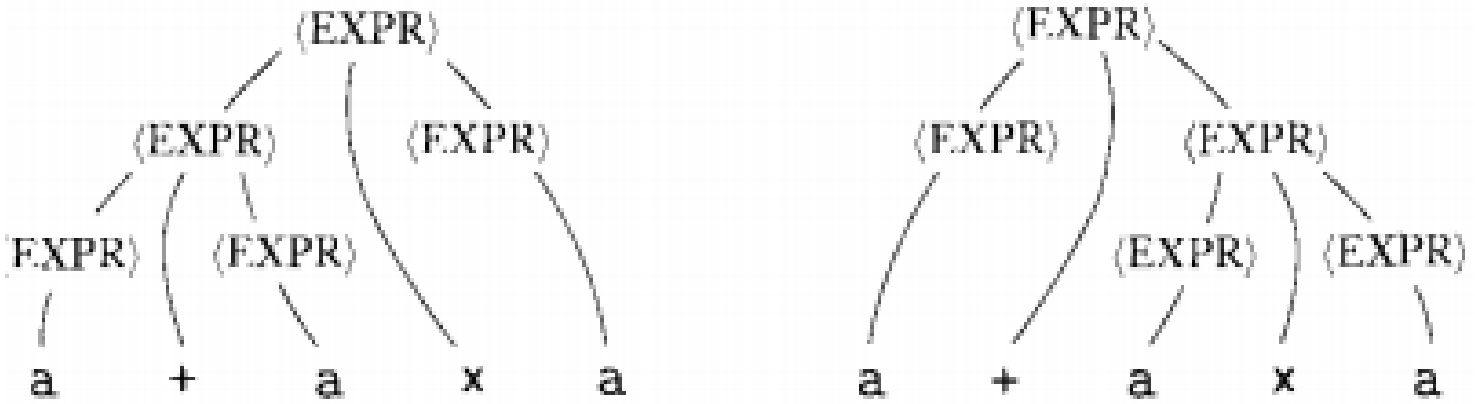
- Sometimes a grammar can generate the same string in several different ways.
- Such a string will have several different parse trees and thus several different meanings.
- This result may be undesirable for certain applications, such as programming languages, where a given program should have a unique interpretation.
- If a grammar generates the same string in several different ways, we say that the string is derived ambiguously in that grammar.
- If a grammar generates some string ambiguously we say that the grammar is ambiguous.

AMBIGUITY

- For example, consider grammar G5:

$$\langle \text{EXPR} \rangle \longrightarrow \langle \text{EXPR} \rangle + \langle \text{EXPR} \rangle \mid \langle \text{EXPR} \rangle \times \langle \text{EXPR} \rangle \mid (\langle \text{EXPR} \rangle) \mid \mathtt{a}$$

- This grammar generates the string a + a x a ambiguously. The following figure shows the two different parse trees.



AMBIGUITY

- This grammar doesn't capture the usual precedence relations and so may group the + before the x or vice versa.
- In contrast grammar $\mathbf{G_4}$ generates exactly the same language, but every generated string has a unique parse tree. Hence $\mathbf{G_4}$ is unambiguous, whereas $\mathbf{G_5}$ is ambiguous.
- Grammar $\mathbf{G_2}$ is another example of an ambiguous grammar. The sentence ***the girl touches the boy with the flower*** has two different derivations.
- **Example**: *give the two parse trees that can generate the above string and observe their correspondence with the two different ways to read that sentence.*

AMBIGUITY

Definition 7

A string w is derived ambiguously in context-free grammar *G* if it has two or more different leftmost derivations. Grammar *G* is ambiguous if it generates some string ambiguously.

- Sometimes when we have an ambiguous grammar we can find an unambiguous grammar that generates the same language.
- Some context-free languages, however, can be generated only by ambiguous grammars. Such languages are called inherently ambiguous

CHOMSKY NORMAL FORM

Chomsky Normal Form (CNF) is a standard form for context-free grammars used in formal language theory

A context-free grammar is in CNF if every rule is of the form

A -> BC

A -> a

Where a is any terminal and A, B, and C are any variables - except that B and C may not be the start variable.

In addition, we permit the rule S -> ε, where S is the start variable. This rule is allowed only if the empty string is part of the language generated by the grammar.

CONVERTING A CFG TO CNF

THEOREM

Any context-free language is generated by a context-free grammar in Chomsky normal form

PROOF IDEA

Conversion from CFG to CNF has several stages wherein rules that violate the conditions are replaced with equivalent ones that are satisfactory.

First, we add a new start variable. Then, we eliminate all ε rules of the form $A \rightarrow \varepsilon$. We also eliminate all unit rules of the form $A \rightarrow B$. In both cases we patch up the grammar to be sure that it still generates the same language. Finally, we convert the remaining rules into the proper form

CONVERTING A CFG TO CNF

PROOF:

- First, we add a new start variable So and the rule So → S, where S was the original start variable. This change guarantees that the start variable doesn't occur on the right-hand side of a rule.

- Second, we take care of all ε rules. We remove an ε -rule A → ε, where A is not the start variable. Then for each occurrence of an A on the right-hand side of a rule, we add a new rule with that occurrence deleted. In other words, if $R \rightarrow uAv$ is a rule in which u, and v are strings of variables and terminals, we add rule $R \rightarrow uv$.

CONVERTING A CFG TO CNF

PROOF:

We do so for each occurrence of an A, so the rule $R \rightarrow uAvAw$ causes us to add $R \rightarrow uvAw$, $R \rightarrow uAvw$ Avw, and $R \rightarrow uvw$. If we have the rule R → A, we add $R \rightarrow \varepsilon$ unless we had previously removed the rule R → ε . We repeat these steps until we eliminate all rules not involving the start variable.

CONVERTING A CFG TO CNF

PROOF:

Third, we handle all unit rules. We remove a unit rule $A \rightarrow B$.

Then, whenever a rule $B \rightarrow u$ appears, we add the rule $A \rightarrow u$ unless this was a unit rule previously removed.

As before, u is a string of variables and terminals. We repeat these steps until we eliminate all unit rules.

CONVERTING A CFG TO CNF

PROOF:

Finally, we convert all remaining rules into the proper form.

We replace each rule $A \rightarrow u_1 u_2 \text{----} u_k$, where $k \geq 3$ and each u_i is a variable or terminal symbol, with the rules $A \rightarrow u_1 A_1$, $A_1 \rightarrow u_2 A_2$, $A_2 \rightarrow u_3 A_3$,…, and $A_{k-2} \rightarrow u_{k-1} u_k$.

The A_i's are new variables. If k=2, we replace any terminal in the preceding rule(s) with the new variable U_i and add the rule $U_i \rightarrow u_i$.

CONVERTING A CFG TO CNF

Example:

Consider a context-free grammar with the following rules:

1. $S \rightarrow ASA | aB$
2. $A \rightarrow B | S$
3. $B \rightarrow b | \epsilon$

a) New Start Symbol

Add $S_0 \rightarrow S$.

CONVERTING A CFG TO CNF

Example:

b) Remove ε-Productions:

- Identify nullable non-terminals (e.g., $B \rightarrow \epsilon$).
- Modify the rules to account for these nullable non-terminals.
- Example: From $S \rightarrow ASA$, consider A could be ϵ, leading to $S \rightarrow SA|AS|S$.

CONVERTING A CFG TO CNF

Example:

c) Remove Unit Productions

- Eliminate productions like $A \rightarrow B$
- Replace them with the rules of B (e.g., $A \rightarrow b$ and $S \rightarrow S$).

d) Remove Useless Symbols

- Ensure all non-terminals contribute to terminal strings.

e) Convert to Binary Form

- For rules longer than 2 symbols, introduce new non-terminals.
- Example: Convert $S \rightarrow ASA$ into $S \rightarrow AX$ and $X \rightarrow SA$.

USE CASES OF CNF

Chomsky Normal Form is particularly useful for algorithmic applications, such as the CYK (Cocke-Younger-Kasami) algorithm, which determines whether a given string belongs to the language generated by a grammar.