# Computer science problems.

**About the problems.** The theme of this year's problems is context free grammars and probabilistic context free grammars. Context free grammars are used in a variety of applications, including compilers, bioinformatics, and natural language processing.

**What you need to do.** For these problems we ask you to write a program (or programs), as well as write some "paper-and-pencil" solutions (use any text editor that you see fit, or scan an actual handwritten solution; convert the result to pdf format if possible).

You may use any programming language you want for your programs, as long as its full implementation is available at no cost and with an easy installation for both a Mac and Windows (free trial versions do not qualify). It is best to implement each problem as a separate function so that we can run them separately. We will be looking for the following in your submissions:

- Correct code that we can run. You need to send us all your code files, including the header files for languages like C++. If you are using standard libraries, make sure to include all "import" statements, as required by the language you are using. Make sure to send the files under the correct names, including the file extension (.java, .c, etc). Make sure that the file names do not contain any identifying information about you, such as your first or last name.

- Test data for your code that you have used (you can write it in comment or in a separate file). Make sure to test your code well – you don't want it to fail our tests!

- Code documentation and instructions. **Important: do not include your name in comments or in any file names.** If you are submitting your answers to non-code problems in a separate file, also make sure that it does not have your name in the contents or in the file name. The only place where you specify your name is the zip file with your solutions which must be of the form `yourlastname-CS-solution.zip` (replace `yourlastname` by your actual last name). **Make sure that you use zip compression, and not any other one, such as tar**. In the beginning of each file specify, in comments:

  1. Problem number(s) in the file. If you have a file with "helper" functions, mark it as such.
  2. The *programming language*, including the *version* (Java 1.11, for instance), the *development framework* (such as Visual Studio) that you used, unless you were using just a plaintext editor (notepad, emacs, etc), and the *platform* (such as Windows, Mac, Linux)
  3. Instructions for running your program (how to call individual functions, pass the input (if any), etc), either in comments in your program file or as a separate file, clearly named. Your program may get

input from the user (i.e. it asks to enter some data and then reads it) or you may store the data in specific variables within your program. You need to clearly explain how to input or set the data.

4. Some of your code may be commented out if it is not used in the final run of your program. Make sure it is clear what needs to be uncommented to run code for each of the problems.
5. All of your test data.
6. If you were using sources other than the ones listed here (i.e. textbooks, online resources, etc) for ideas for your solutions, please clearly credit these contributions. This is a courtesy to work of others and a part of ethics code for scholars.
7. Make sure that you clearly specify where input files are supposed to be located, provide an example input file and an example of how the file name would be specified in the input. Use relative paths (from the top of the project or from the executable), not absolute paths. **All input files must have an extension `.in`, all output files an extension `.out`.**

- Clear, understandable, and well-organized code. This includes:

1. Clear separation between problems; comments that help find individual problems and explain how to run the corresponding functions.
2. Breaking down code into functions that are clearly named and described (in comments), using meaningful names for variables and function parameters. Your code should be as self-explanatory as possible. While using comments helps, naming a variable `average` is better that naming it `x` and writing a comment "x represents the average".
3. Minimization of code repetition. Rather than using a copy-paste approach, use functions for repeated code and reuse these functions.
4. Using well-chosen storage structures (use an array or a list instead of ten variables, for instance) and well-chosen programming constructs (use loops or recursion when you can, rather than repeated code).
5. While we are not asking for the fastest program (it's better to make it more readable), you should avoid unnecessary overhead.

**Background and problems.**

We start by briefly defining the key terms. For more details and discussion see [3] or `https://en.wikipedia.org/wiki/Context-free_grammar` or any other resources on CFGs.

**Definition 1.** A context-free grammar (CFG) is a set of 4 elements: $\{V, \Sigma, R, S\}$, where:

- $V$ is a finite set of variables (called *non-terminals* or *variables*). We use upper-case letters for elements of $V$.

- $\Sigma$ (called *the alphabet*) is a finite set of symbols (called *terminals*). We use lower-case letters for elements of $\Sigma$.
- $R$ is a set of *rules* of the form $V_1 \to w$ where the left-hand side $V_1$ is an element of $V$, and the right-hand side $w$ is a finite string of terminals and non-terminals. $w$ may be an empty string, denoted as $\varepsilon$.
- $S$ is an element of $V$ called the *start variable* or *start symbol*.

**Definition 2.** A string of terminals $w$ is derived from a grammar $G = \{V, \Sigma, R, S\}$ if there is a sequence $w_0 \to w_1 \to w_2 \dots w_n$ such that $w_0 = S, w_n = w$ and in each step $w_i \to w_{i+1}$ is performed by replacing a non-terminal $V'$ in $w_i$ by a string $w'$ such that $V' \to w'$ is a member of $R$. This sequence is called a *derivation* of $w$ in $G$. Note that $w$ consists only of terminals, and therefore no further derivation steps can be performed.

A derivation in which in each $w_i$ the leftmost non-terminal is replaced is called a *leftmost derivation*.

The set of all strings that are derived from a grammar $G$ are called its *language* and is denoted $L(G)$. We say that $G$ generates $L(G)$.

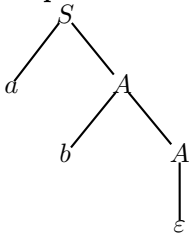
**Example 1.** Consider the following grammar: $V = \{S, A\}, \Sigma = \{a, b\}, R = \{S \to aA, A \to \varepsilon, A \to aA, A \to bA\}$, and the start variable is $S$. We will be using a slightly different way of writing grammars in which we don't explicitly specify $V$ or $\Sigma$ since they may be determined from the rules, we $S$ as the start variable (unless explicitly stated otherwise), and combine rules for the same non-terminal into one string using | as a separator. In this format the same grammar will be written as:

$$S \to aA$$
$$A \to \varepsilon \mid aA \mid bA$$

The string $ab$ is in $L(G)$ and has a derivation $S \to aA \to abA \to ab$. The last step uses the rule $A \to \varepsilon$. It is easy to observe that any string of $a$ and $b$ that starts with $a$ is derivable in $G$, and no other strings are, so $L(G)$ is the set of all strings over $a, b$ that start with $a$.

Another way of representing a derivation of a string in a grammar is a parse tree. A parse tree for string $w \in L(G)$ is a tree that's rooted in $S$ and represents a rule-based substitution $V' \to w'$ by positioning symbols of $w'$ in order as children of $V'$. The derived string is all the terminals at the leaves of the tree in left-to-right order.

**Example 2.** The parse tree for the derivation in Example 1 is as follows:



**Problem 1.** Consider the following CFG:

$$S \to \varepsilon \mid aS \mid A$$
$$A \to \varepsilon \mid aAb$$

**Question 1.** Construct derivations and draw parse trees for the following strings: $\varepsilon, a, aab, aabb$.
**Question 2.** What is the language of this CFG?

Note that there may be multiple derivations corresponding to the same parse tree. For instance, consider the grammar $G_1$:

$$S \to AB$$
$$A \to \varepsilon \mid aA \qquad\qquad B \to \varepsilon \mid bB$$

The following two derivations both derive $ab$:

- $S \to AB \to aAB \to aB \to abB \to ab$
- $S \to AB \to AbB \to Ab \to aAb \to ab$

However, there is only one leftmost derivation for $ab$ and only one parse tree. In general, each parse tree corresponds to only one leftmost derivation, and each leftmost derivation corresponds to a unique parse tree.

**Problem 2. Question 1.** Are there other derivations for $ab$ in $G_1$? If yes, show one, otherwise explain why there are no other ones.
**Question 2.** What is the leftmost derivation for $ab$ in $G_1$?
**Question 3.** Draw a parse tree for $ab$ in $G_1$.
**Question 4.** What is the language of $G_1$

**Definition 3.** Given a CFG $G$, if at least one string $w \in L(G)$ has two different parse trees (or equivalently, two different leftmost derivations) then $G$ is called an *ambiguous* grammar.

**Example 3.** Consider the following grammar:

$$S \to aA$$
$$A \to \varepsilon \mid aA \mid bA \mid Aa \mid Ab$$

**Problem 3.** The problem refers to the grammar defined in Example 3

- Find a string derived in $G$ that has at least two different leftmost derivations. Show either the derivations or the parse trees.
- Explain why this grammar defines the same language as the one given in Example 1.
- Explain why the grammar in Example 1 is not ambiguous.

The CFG in Example 3 is ambiguous, but it has an equivalent (i.e. defining the exact same language) unambiguous CFG. However, some ambiguous CFGs don't have an equivalent unambiguous CFG. These grammars are called *inherently ambiguous.*

One common application of CFGs is parsing expressions in a programming language. In this case we would like to eliminate ambiguity. For instance, an expression $2 + 3 \times 4 - 5$ should be parsed as if it has the set of parentheses $(2 + (3 \times 4)) - 5$, otherwise the result would be not what we would expect. [3], as well as a large number of compilers textbooks, discuss setting correct *precedence* of operations (which operator must be executed first in the absence of parentheses) and *associativity* (if there is an ambiguity between two operators of the same precedence, should the expression be evaluated from left to right or from right to left). This online resource `http://www.cs.ecu.edu/karl/5220/spr16/Notes/CFG/precedence.html` has a concise informative overview of the problem and the solutions. More information is available in books on compiler design, e.g. [1].

**Problem 4.** Design an unambiguous CFG for a language that consists only of variable names, operators `+, -, =` (assignment statement, as in C or Java), and parentheses. The CFG should be unambiguous with the following ways of determining order of operations:

1. `+` and `-` have the same precedence, and `=` has a lower precedence than `+` and `-`. For instance, in $a = b + c$ the subexpression $b + c$ is evaluated first, and its result is assigned to $a$. Note that although the expression $a+b = c$ doesn't make sense in most programming languages, for simplicity we consider it to be a valid expression and parse it as if parentheses were placed as $(a + b) = c$.

2. `+` and `-` are left-associative, so $a + b - c$ is evaluated left to right, as in $(a + b) - c$, and $a - b - c$ is evaluated as $(a - b) - c$.

3. `=` is right-associative, so $a = b = c$ is evaluated as $a = (b = c)$ (in many programming languages the value, or the "result", of an assignment statement is the assigned value, so the expression $a = b = c$ assigns the value of $c$ to both $a$ and $b$).

4. Parentheses may surround any valid subexpression. They overwrite the default order of operations and can be nested arbitrarily. For instance, the following is a valid expression: $a = (b + (c = d))$. In this case $d$ is assigned to $c$, then the sum of $b$ and the result of the assignment is assigned to $a$. Note that the outer set of parentheses is not needed, but that's not an error. Surrounding a single variable or an expression in parentheses by a set of parentheses is allowed.

Use a non-terminal $V$ to denote a variable, i.e. $V \to a \mid b \mid c \ldots$. `+, -, =` and both parentheses are terminals, i.e. a part of $\Sigma$. Write your CFG as a set of rules. Show parse trees or leftmost derivations for all expressions in this problem and at least 2 more that show different cases.

**Definition 4** (Chomsky normal form)**.** A CFG is said to be in *Chomsky normal form (CNF)* if all the rules in it have one of the following forms:

- $V_1 \to a_1$, where $V_1 \in V, a_1 \in \Sigma$.
- $V_1 \to V_2V_3$, where $V_1, V_2, V_3 \in V$, $V_2 \neq S$, $V_3 \neq S$.
- $S \to \varepsilon$.

The advantage of Chomsky normal form is that it provides an upper bound on the length of derivation of a string: if $w$ is a string of terminals of the length $n > 0$ then the length of a derivation for $w$ in $G$ is $2n - 1$. This allows a naive way of checking whether $w$ is in $L(G)$: run all derivations of $2n - 1$ steps and check if $w$ is one of the resulting strings (see [3] or online resources for details).

**Problem 5. Question 1.** Given a grammar $G$ in CNF, how do you determine whether $\varepsilon$ is in $L(G)$?
**Question 2.** Prove the above claim: if $w$ is a string of terminals of the length $n > 0$ then the length of a derivation for $w$ in $G$ is $2n - 1$.
**Question 3.** What is the running time of the naive way of checking if $w$ of length $n$ is in $L(G)$ in the worst case (clearly state what situation would be the worst case), assuming that each derivation step takes a constant time? Give the result as a function of $n$, justify your answer. Ignore constants and lower terms.

The *Cocke, Younger and Kasami* (usually abbreviated as *CYK*, sometimes as CKY) *algorithm* is a bottom-up algorithm that builds all parse trees (equivalently, all leftmost derivations) for a string in $O(n^3)$ time. The grammar is given in CNF. The first pass of the algorithm is to determine non-terminals that produce each of the terminals of a string, the next one is to determine which non-terminals produce pairs of neighboring non-termimals produced by the first pass, etc. At the end the algorithm determines whether the result is derived from $S$ (the start variable) or not. For more details see [3] or `https://en.wikipedia.org/wiki/CYK_algorithm`.

**Problem 6.** Write a program that reads a file name (see above instructions for how to specify a file name) and a string. The file name gives a CFG $G$ in the following format:

```
S:aA
A:$
A:aA
A:bA
```

This is the CFG from Example 1. Each rule is on its own line, the left hand side followed by `:`, followed by the right hand side of the rule. All non-terminals are capital letters, all terminals are lower-case. The `$` denotes $\varepsilon$. Your program will output `Yes` if the string is in $L(G)$ and `No` if it's not. Input `$` denotes the empty string. Rules may be given in any order; the start variable is always $S$.

You may use a naive method described above (trying all derivation of length $2n-1$), but it's preferable to use CYK algorithm. Even though its implementation may be available in a library, you need to develop your own. Note that the grammar might not be in CNF, so you need to convert it first. The conversion algorithm is described in [3].

**Definition 5.** Given an alphabet $\Sigma$ (assumes to have its own order which we referred to as alphabetical) and a set $L \subseteq \Sigma$, *short-lex* (sometimes mistakenly called *lexicographic*) order is an order in which a string $w_1$ is before a string $w_2$ if the length of $w_1$ is less than the length of $w_2$ or $w_1$ and $w_2$ have the same length and the first different symbol in the the two strings is alphabetically smaller in $w_1$ than in $w_2$. For instance, if the alphabet is $\{a, b, c\}$ (with the usual alphabetical ordering) then $ab$ is before $aaa$ (because it's shorter), and $aab$ is before $aba$ because at the position of their first difference (the second symbol) $aab$ has $a$, whereas $aba$ has $b$ which is later in the alphabet.
An empty string $\varepsilon$ is the first string in any set in which it appears.

**Problem 7.** Write a program that reads an input file name, an output file name, and a positive integer $n$. The input file contains the description of a CFG, as in Problem 6. The program outputs the first $n$ strings in $L(G)$ in short-lex order (see Definition 5) into the output file, 10 per line (the last line may have fewer than 10; make sure to add the end-of-line marker at the end of the last line and close the file). You may assume that $n \leq 1000$.

**Definition 6.** A *probabilistic context free grammar (PCFG)* is a CFG with a probability $p$ ($0 \leq p \leq 1$) assigned to every rule in $R$. For any non-terminal $V_1 \in V$ the sum of all rules in $R$ for which $V_1$ is the left-hand side equals 1.

Probabilities attached to rules indicates how likely the rule would be chosen when generating strings. Thus, PCFGs generate distributions of parse trees and of strings of terminals. Note that since the same string may have multiple parse trees (because the underlying CFG may be ambiguous), a parse tree may have a different probability among all trees than the string it generates has among all strings.

Certain conditions need to be satisfied to make sure that most probably derivations are likely to terminate. We assume that only such grammars are considered in all problems and questions.

One of the most common applications of PCFGs is to describe natural languages where interpretation of a sentence may depend on its structure. For instance, assigning likely the most meaning to a grammatically ambiguous sentence *Time flies like an arrow* depends on what the most likely parsing of it is.

If you are curious about the details and connections to other statistical models, see a foundational work on PCFG [2].

**Example 4.** Consider the following PCFG, where each rule is followed by its

probability:

$$S \to A\ 0.3 \mid B\ 0.7$$
$$A \to aA\ 0.3 \mid bA\ 03 \mid a\ 04$$
$$B \to aB\ 0.3 \mid bB\ 03 \mid b\ 04$$

This language contains all strings over the alphabet $\{a, b\}$ except $\varepsilon$. However, strings that end with $a$ appear 30% of the time, and those that end with $b$ appear 70% of the time. Note that in this PCFG each string has a unique parse tree.

**Problem 8.** Given the PCFG in Example 4, what is the probability of a parse tree of the string $a$? Of the string $aa$? Of the string $abb$? Show your computations.

**Problem 9.** Develop a PCFG for a language of all strings over $\{a, b\}$ in which groups of repetition of the same symbol in a row even number of times (2 and larger) appear 30% of the time groups of odd number of repetitions of the same symbol. Within each group the probability of a sequence of length $n$ is twice higher than of all lengths higher than $n$ combined. The probability of a string with $n$ groups of repetition is also twice higher than of all of the strings with the larger number of groups combined. The empty string doesn't appear in the language.

**Problem 10.** Write a program that reads an input file name, an output file name, and a number $n$. The file contains description of a PCFG in the following format:

```
S:A 0.3
S:B 0.7
A:aA 0.3
A:bA 0.3
B:aB 0.3
B:bB 0.3
A:a 0.4
B:b 0.4
```

Rules are one per line in the same format as in Problem 6, but additionally each rule is followed by its probability. You may assume that probabilities are valid, i.e. add up to 1 for each non-terminal.

The program generates $n$ strings in the language (with possible repetitions), strings are listed in any order. Strings are printed to the output file 10 per line, as in Problem 7.

The CYK algorithm directly generalizes to PCFGs: in addition to all possible parse trees it also computes the probabilities of each subtree rooted in each encountered non-terminal and eventually of the entire parse trees. An example is given here `https://courses.cs.washington.edu/courses/cse590a/09wi/pcfg.pdf`. Note that the grammar must be in CNF. If it is given in a different from, probabilities of rules must be recomputed when converting into CNF.

**Problem 11.** Write a program that reads an input file name and a string. The file contains PCFG rules, as in Problem 10. **The grammar is given in Chomsky normal form.** The program outputs all possible parse trees (in any format) or leftmost derivations for the given string, with their probabilities, starting with the most probably one. If the string is not in the language, it outputs "Not in the language".

Try your program on the following grammar and the strings *aab, abab, abb.*

```
S:AB 1.0
A:a 0.6
A:AA 0.2
A:AB 0.2
B:b 0.5
B:AB 0.5
```

**Problem 12.** This problem is the opposite of the previous one: you are given a thousand strings generated by a PCFG. However, the PCFG is given without the probabilities. Based on the distribution of strings, approximate the probabilities. The grammar is:

```
S:NM
S:MN
A:a
B:b
N:AP
N:BR
P:NA
R:NB
N:a
N:b
M:AM
M:BM
M:a
M:b
```

The output is given here: `problem12.out`.

You may use any methods you'd like. However, more general methods will earn higher points. You may assume that any grammar given to you is in CNF.

# References


[1] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools (2Nd Edition)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2006.

[2] Zhiyi Chi. Statistical properties of probabilistic context-free grammars. *Comput. Linguist.*, 25(1):131–160, March 1999.

[3] Michael Sipser. *Introduction to the Theory of Computation*. Cengage Learning, Boston, MA, third edition, 2012.