Complementary Task 2:
Algorithms for Context-Free Grammars

1 General Instructions Applying to All Tasks

Each of the complementary tasks of the course results in a written report that will be graded. As the language of this course is English, your report must be written in English, too. Each task is divided into four individual steps that should roughly correspond to sections of your report. Each step has its own deadline at which you must to hand in a draft of your report (or, in step 4, the final version). The drafts corresponding to steps 1–3 must each receive a pass grade in order to continue. The pass grade merely signifies that you have worked on the task according to its specification and have made reasonable progress. As a consequence, a fail grade cannot be rectified by handing in a new version. On each of the drafts handed in in steps 1–3 you will receive comments that must be taken into account when going ahead. The final report will be graded according to the scale fail, 3, 4, 5.

The following are the deadlines. – Make sure to keep them!

Step 1 2014-09-21, 23:59
Step 2 2014-10-05, 23:59
Step 3 2014-10-19, 23:59
Step 4 2014-11-02, 23:59

Please submit your drafts as well as the final report here. If implementations are required (i.e., for tasks 1 and 2), make sure that the files are organized in a reasonable way, and that it does not require a wizard to make the implementation work outside your personal environment.

2 The Task

From previous courses, you should already have at least some basic knowledge about context-free grammars (CFGs, for short). If not, this task may not be the right one for you, so please think twice before choosing it.

One of the nice things regarding CFGs is that there are efficient algorithms that analyze and modify CFGs. One may say that CFGs are among the most general systems known in Computer Science that, for many interesting questions, still admit efficient algorithms. Going beyond CFGs, one mostly encounters undecidable or, at the very least, computationally intractable problems. (In fact, even some of the basic questions for CFGs are undecidable, which is why finite automata are of interest as an even simpler model, but in this task we will focus on the decidable problems.)

Your task is to make yourself familiar with some of the most useful algorithms for CFGs, describe them in your own words, implement them, and compare the efficiency
of your implementation with the theoretical prediction. The task must not be confused
with a programming exercise. Of course, your implementations should work well,
but the important thing is that you understand the algorithms, present them in an
appropriate way in your report, and discuss your experimental findings in a convincing
manner. You should also include a brief description of how to use your implementation,
making it possible for others to reproduce your results.

You may choose any reasonably well-known programming language. Not only impera-
tive (including object-oriented) languages may be of interest, but also functional ones.
In fact, since efficient algorithms are usually presented in an imperative style, it could
be interesting to see what happens in terms of efficiency if they are implemented in
a functional style. Together with the final report, you must submit a version of im-
plemented algorithms that runs under Unix/Linux, even if you choose to work under
another operating system (which is fine).

Each step except the last one follows the same pattern:

1. Search the literature to find a good exposition of the algorithms in question.
   It should not only consist of the algorithm itself, but also include correctness
   arguments and preferably an efficiency analysis. Read and understand. Make
   sure to use reliable scientific literature (textbooks or articles) rather than web
   resources. For example, various good textbooks are available at the university
   library. If you find a good one that suits you, a single textbook may be sufficient
   for the whole task, but I recommend you to check a few.

2. Implement the algorithms in a faithful way, i.e., try to make sure to preserve
   their efficiency, and conduct experiments to (a) be reasonably sure that your
   implementation is correct and (b) compare the efficiency of selected algorithms
   (specifically mentioned below) with the theoretical prediction.
   Please invest some time and thinking in both the correctness tests and the effi-
   ciency experiments. Concerning correctness, it is important that the tests cover
   all cases, even the odd ones, rather than just grammars that are built in a “nice”
   way. The efficiency experiments are even more demanding, because there you
   have to come up with one or more sequences of larger and larger context-free
   grammars that provide meaningful test cases, i.e., preferably test cases which
   exhibit the worst case behavior of the algorithm under examination.

3. Extend your report by a corresponding section. It should introduce the no-
tions and algorithms in question, briefly discuss their correctness and efficiency
   characteristics, and the results of your experiments. Some words about your
   implementation may also be in order, but this depends on how straightforwardly
   the abstract algorithms could be turned into concrete programs. Do not forget
   to add appropriate references, telling the reader where you got your information
   from.

To make input and output reasonably simple, it is recommended to use uppercase let-
ters followed by zero or more digits as nonterminals and lowercase letters as terminals.
With the additional convention that the initial nonterminal is always $S$, a context-free
grammar can thus be denoted as a list of rules. For instance,

$$
S \rightarrow SS, \quad S \rightarrow A, \quad A \rightarrow bAb, \quad A \rightarrow aAA, \quad A \rightarrow \epsilon, \quad S1 \rightarrow S
$$

would denote the CFG $(N,T,P,S)$ with $N = \{S,S_1,A\}$, $T = \{a,b\}$, and $P = \{S \rightarrow SS, \ S \rightarrow A, \ A \rightarrow aAA, \ A \rightarrow bAb, \ A \rightarrow \epsilon, \ S_1 \rightarrow S\}$ (where $\epsilon$ denotes the empty
string).
Step 1: The Chomsky normal form and the CYK algorithm

Recall CFGs (if necessary), and the Chomsky normal form for CFGs. One of the most basic problems for CFGs is the membership problem, i.e., given a CFG $G$ and a string $w$, is $w$ an element of $L(G)$? A polynomial algorithm that solves this problem is the Cocke-Younger-Kasami algorithm (CYK algorithm). Acquaint yourself with this algorithm, which is a typical dynamic programming algorithm, and implement it. Note that the algorithm requires $G$ to be in Chomsky normal form. For a fixed CFG $G$, the algorithm should run in cubic time. Choose a suitable CFG $G$ and a sequence of strings $w_1, w_2, \ldots$ to check whether this seems to be the case.

Note that, while the section of the report corresponding to this step should focus on the Chomsky normal form and the CYK algorithm, it is a good idea to establish some basic notation and terminology. It is recommended that you start by making notes about your conventions in a separate section. This part does not need to be well worked out yet, and can evolve over time. It is mostly valuable for yourself, because it helps you to avoid inconsistencies. In Step 4, your notes will be used to write the preliminary part of your report.

Step 2: Turning a CFG into Chomsky Normal Form

As seen in the previous step, it is useful if a CFG is in Chomsky normal form. In the second step, the algorithm that turns an arbitrary CFG into an equivalent CFG in Chomsky normal form will be studied.

To start with, one usually pre-processes the given CFG by applying two very basic algorithms, namely those which remove unproductive nonterminals (those from which no terminal string can be derived) and unreachable nonterminals (those which cannot be reached from the initial nonterminal, such as $S_1$ in the example above). Both can easily be implemented in such a way that they run in linear time. Executed one after another (in the right order!), they yield a linear algorithm for the removal of so-called useless nonterminals. (Be aware of the possibility that the initial nonterminal may be useless. Handle this case in an appropriate way. Obviously, the initial nonterminal is the only useless nonterminal that cannot be removed from $G$.)

Going on, conversion into Chomsky normal form now requires the removal of $\epsilon$-productions (those with an empty right-hand side) and chain productions (those of the form $A \rightarrow B$). These are also easy polynomial algorithms (though they do not, in general, run in linear time). Finally, Chomsky normal form is obtained by introducing new nonterminals and “splitting up” the rules into rules of the types $A \rightarrow BC$ and $A \rightarrow a$.

No running times of your implementation need to be checked experimentally, but make sure to conduct a few tests to convince yourself about the correctness of your implementation.

Step 3: Deciding Emptiness and Finiteness

This step focuses on algorithms that take as input a CFG $G$ and check whether $L(G)$ is empty and whether it is finite. The first algorithm is a trivial application of the algorithm that detects productive nonterminals. Thus, it runs in linear time. The second is based on the idea to discover productive cycles. Provided that useless nonterminals, $\epsilon$-productions, and chain productions have been removed — procedures that you already have — everything needed is to detect whether the grammar contains cycles. Hence, even this one should run in linear time if implemented appropriately. After implementing the algorithms, check in an experimental way whether the running time
is as expected (and, of course, whether the results are correct).

**Step 4: The Final Touch**

If necessary, tidy up your implementation to a reasonable extent, add comments to the code, etc. While I do not intend to study the implementations in detail, I might want to check one thing or the other, and in those cases the code must be accessible.

Add a short appendix to your report, which contains instructions about how to install and use your implementation of the algorithms. Moreover, extend your report by an introduction that explains what the document is about and provides a short general introduction to CFGs and the notation and terminology used. In addition to this, the introduction should summarize the structure and contents of the remainder of your report.

Finally, write a concluding section that summarizes your work and mentions algorithms for CFGs that have not been implemented. (You should easily be able to find a few in the literature.) Moreover, to round things off, add one or two paragraphs about undecidable problems for CFGs.

*I hope that this task has been interesting and inspiring to work with. Please do not hesitate to give me any type of feedback.*