Theory of Computation
Problem Set 7
Universidad Politecnica de San Luis Potosi

Please start solving these problems immediately, don't procrastinate, and work in study groups.

Please do not simply copy answers that you do not fully understand;

Advice: Please try to solve the easier problems first (where the meta-problem here is to figure out which are the easier ones). Don't spend too long on any single problem without also attempting (in parallel) to solve other problems as well. This way, solutions to the easier problems (at least easier for you) will reveal themselves much sooner (think about this as a "hedging strategy" or "dovetailing strategy").

Problem One: Designing CFGs (24 Points)

Below are a list of alphabets and languages over those alphabets. For each language, design a context-free grammar that generates that language.

i. For the alphabet $\Sigma = \{$ `0`, `1`, `2` $\}$, write a CFG for the language $L = \{ w \in \Sigma^* \mid w$ contains `00` as a substring $\}$

ii. For the alphabet $\Sigma = \{$ `0`, `1`, `2` $\}$, write a CFG for the language $L = \{ w \in \Sigma^* \mid w$ does not contain both `0` and `1`. $\}$

iii. For the alphabet $\Sigma = \{$ `0`, `1`, `2` $\}$, write a CFG for $L = \{ \texttt{0}^i\texttt{1}^j\texttt{2}^k \mid i, j, k \in \mathbb{N} \wedge (i = j \vee i = k) \}$

iv. Suppose that you want to write a context-free grammar that describes function prototypes in C or C++. Let $\Sigma = \{$ `int`, `double`, `(`, `)`, `","`, `name`, `;`$\}$, where `name` is a symbol that represents the name of some function or variable. Let $L = \{ w \in \Sigma^* \mid w$ is a valid function prototype $\}$. So, for example, the following would all be valid function prototypes:

```
int name();

double name(int name);

double name(double name, int name, double name);
```

Assuming that each argument to a function must have a name (though in C and C++ names are optional), write a CFG that generates L. Functions may take any number of arguments.

You don't need to worry about generating spaces in-between the symbols you produce. Typically, a compiler would handle that in a separate step.

v. For the alphabet $\Sigma = \{$`0`, `1`$\}$, write a CFG for the language $\overline{PAL} = \{ w \in \Sigma^* \mid w$ is ***not*** a palindrome $\}$. That is, w is not the same when read forwards and backwards. Thus `001` $\in \overline{PAL}$ and `100101` $\in \overline{PAL}$, but `101` $\notin \overline{PAL}$ and `11` $\notin \overline{PAL}$.

Problem Two: The Complexity of Addition (12 Points)

On the previous problem set, we began addressing the question

How hard is it to add two numbers?

We will now directly answer that question.

Consider the language $ADD = \{\texttt{1}^m\texttt{+}\texttt{1}^n\texttt{=}\texttt{1}^{m+n} \mid m, n \in \mathbb{N}\}$ over the alphabet $\Sigma = \{$`1`, `+` , `=`$\}$. That is, *ADD* consists of strings encoding two unary numbers and their sum. As you proved on the previous problem set, *ADD* is not regular.

i. Write a context-free grammar for *ADD*. This proves that *ADD* is context-free.

ii. Design a **deterministic** PDA that recognizes *ADD* (recall that a DPDA is a PDA in which for any combination of a state, input symbol, and stack symbol there is at most one transition that can be followed, including ε-transitions). This proves that *ADD* is not only a context-free language, but also a deterministic context-free language.

Problem Three: The Complexity of Pet Ownership (16 Points)

On the previous problem set, you designed a DFA for taking your dog on a walk with a leash. In this problem, you will see what happens when you take off your dog's leash.

Let $\Sigma = \{\texttt{Y}, \texttt{D}\}$, where **Y** represents you moving one unit forward and **D** represents your dog moving one step forward. Then a string of **Y**s and **D**s represents you and your dog going for a walk. For example, in the string **YYDY**, you end up two steps ahead of your dog, while in the string **DDDDYY**, your dog takes off and ends up two steps ahead of you.

If we consider the language of strings representing walks where you and your dog end up at the same location, we get the language $DOGWALK = \{ w \in \Sigma^* \mid w$ has the same number of **Y**s and **D**s $\}$.

i. Write a context-free grammar that generates *DOGWALK*. This proves that *DOGWALK* is context-free.

ii. Design a **deterministic** PDA that recognizes *DOGWALK*. This proves that *DOGWALK* is not only a context-free language, but also as deterministic context-free language.

Problem Four: Uncertainty about Ambiguity (16 Points)

In this question, you'll explore some properties of ambiguous grammars. Consider the language

$$GE_{01} = \{ \texttt{0}^n\texttt{1}^m \mid n \geq m \}$$

Here is one possible context-free grammar for GE_{01}:

$$S \rightarrow \texttt{0}S \mid \texttt{0}S\texttt{1} \mid \varepsilon$$

You may want to play around with this grammar a bit before answering these questions.

i. Show that this grammar is ambiguous by providing a string in GE_{01} and two different parse trees for that string.

ii. Rewrite this grammar so that it is unambiguous. Explain, but do not formally prove, why your new grammar is unambiguous.

iii. Prove or disprove: If a grammar G is ambiguous, then there is no DPDA that accepts $\mathscr{L}(G)$.

Problem Five: Shrinking PDAs (20 Points)

As you saw in lecture, any CFG can be converted into a PDA with just three states, meaning that the full power of the context-free languages can be expressed by three-state PDAs. This question asks you to see just how few states PDAs can have while retaining their expressive power.

i. Show how to modify the construction from lecture that turns CFGs into PDAs so that the generated PDA has only *two* states instead of three. You should briefly describe how your construction works, but you don't need to formally prove that it is correct.

ii. Prove that for any PDA P, there is a PDA P' such that $\mathscr{L}(P) = \mathscr{L}(P')$ and P' has only two states.

iii. Your result from (ii) shows that any PDA can be converted into an equivalent PDA with just two states. However, it is not always possible to convert a PDA into an equivalent PDA with just one state. Find a context-free language that does not have a one-state PDA, then prove that it does not have one.

We will cover the material necessary to solve these problems in Monday's lecture.

Problem Six: The Complexity of Exponentiation (12 Points)

On the previous problem set, we began addressing the question

How hard is it to check whether a number is a perfect power of two?

A number is a power of two if it can be written as 2^n for some natural number n. Consider the language *POWER2* = $\{ 1^{2^n} \mid n \in \mathbb{N} \}$ over the alphabet $\Sigma = \{\texttt{1}\}$. That is, *POWER2* contains all strings whose lengths are a power of two. For example, the smallest strings in *POWER2* are **1**, **11**, and **1111**.

On the previous problem set, you proved that *POWER2* is not regular using the pumping lemma for *regular* languages. Now, using the pumping lemma for context-free languages, prove that it is not context-free either. *(Hint: As with last time, you may want to use the fact that $n < 2^n$ for all $n \in \mathbb{N}$)*

Problem Seven: The Complexity of String Searching (20 Points)

On the previous problem set, we began addressing the question

How hard is it to search a string for a substring?

Given a string to search for (called the **pattern**) and a string in which the search should be conducted (called the **text**), we want to determine whether the pattern appears in the text. To encode this as a language problem, we let $\Sigma = \{\texttt{0}, \texttt{1}, \texttt{?}\}$ and encoded questions of the form "does pattern string *p* appear in text *t*" as the string *p*`?`*t*. For example:

"Does **0110** appear in **1110110** ?"	would be encoded as	**0110?1110110**
"Does **11** appear in **0001** ?"	would be encoded as	**11?0001**
"Does ε appear in **1100** ?"	would be encoded as	**?1100**

Let the language *SEARCH* = $\{ p\texttt{?}t \mid p, t \in \{\texttt{0}, \texttt{1}\}^* \text{ and } p \text{ is a substring of } t \}$. On the last problem set, you proved that *SEARCH* is not regular using the pumping lemma for *regular* languages. Now, using the pumping lemma for context-free languages, prove that *SEARCH* is not context-free either.

As a hint, the pattern and text string you show cannot be pumped must use both **0**s and **1**s. In fact, if you restrict the pattern and text strings to strings consisting solely of **0**s, you get the language

$$LE = \{ \texttt{0}^n\texttt{?}\texttt{0}^m \mid n, m \in \mathbb{N} \wedge n \leq m \}$$

which *is* context-free.

Problem Eight: Course Feedback (5 Points)

We want this course to be as good as it can be, and we'd really appreciate your feedback on how we're doing. For a free five points, please answer the following questions. We'll give you full credit no matter what you write (as long as you write something!), but we'd appreciate it if you're honest.

i. How hard did you find this problem set? How long did it take you to finish? Does that seem unreasonably difficult or time-consuming for a five-unit class?

ii. Did you attend Monday's problem session? If so, did you find it useful?

iii. How is the pace of this course so far? Too slow? Too fast? Just right?

iv. Is there anything in particular we could do better? Is there anything in particular that you think we're doing well?