Tutorial 6: Context-Free Tutorial

---

1. Solve the following problems. Assume the alphabet to be $\Sigma = \{a, b, c\}$

   (a) Consider the language of non-palindromes (words that are not palindromes). A palindrome is a word that spells the same way forward and backwards. For example, 'abcba' is a palindrome but 'abbaa' is not). Construct a PDA and a CFG for the language.

   (b) Consider the language $\{w \mid w \neq uu \text{ for any } u \in \Sigma^*\}$. Construct a PDA and a CFG for the language.

Solution:

1. The NPDA keeps pushing letters on the stack up to a point which is our guess of the halfway point (state Q_0). Then we keep popping letters from the stack as long as they match the scanned letter (state Q_1), until we spot a mismatch which is when we move to state Q_2, in which we pop off letters irrespective of a match. We accept the word if we are in state Q_2 at the end, with an empty stack. Formally, the NPDA is $(\{Q_0, Q_1, Q_2\}, \Sigma = \{a, b, c\}, Q = \{a, b, c, \perp\}, \delta, Q_0, \perp, \{Q_2\})$, with $\delta =$

$$\begin{gathered}((Q_0, x, A), (Q_0, xA))\,\forall\, x \in \Sigma, A \in Q \\ ((Q_0, x, A), (Q_1, A))\,\forall\, x \in \Sigma \cup \{\epsilon\}, A \in Q \\ ((Q_1, x, x), (Q_1, \epsilon))\,\forall\, x \in \Sigma \\ ((Q_1, x, y), (Q_2, \epsilon))\,\forall\, x, y \in \Sigma \text{ st } x \neq y \\ ((Q_2, x, A), (Q_2, \epsilon))\,\forall\, x \in \Sigma, A \in Q.\end{gathered}$$

   The CFG is:

$$\begin{aligned} S &\to aSa|bSb|cSc \\ S &\to xS'y \text{ for } x \neq y \\ S' &\to xS'y \text{ for all } x, y \\ S' &\to a|b|c|\epsilon. \end{aligned}$$

2. Let $w_{i..j}$ denote the word formed by the i^{th} to j^{th} position (both inclusive) of w, i.e it is a continuous substring of w. Let w_i denote the i^{th} letter of w. (We count from 1).

   Note that odd length words are immediately a part of this language, since they cannot be of the form uu for any word u. These odd words can be divided into three categories: Those which have a a at the center, those which have a b at the center, and those which have a c at the center. The reason for this division will become apparent later. The following CFG captures these words:

$$\begin{gathered} S_{odd} \to A \mid B \mid C \\ A \to a \mid aAb \mid bAc \mid cAa \mid bAa \mid aAc \mid cAb \\ B \to b \mid aBb \mid bBc \mid cBa \mid bBa \mid aBc \mid cBb \\ C \to c \mid aCb \mid bCc \mid cCa \mid bCa \mid aCc \mid cCb \end{gathered}$$

   For even length word in this language, suppose the length of the word is $2n$. Then, there exists a position $k \leq n$ such that the letter at position k and position $n + k$ are different. (otherwise the two halves of w are the same word, and so w = uu). Now

consider the substring $w_{1..2k-1}$ and $w_{2k..2n}$. Both of these are of odd length, and the center letter of the first one is w_k and that of the second word is w_{n+k}. Thus, **every** even word of this language can be written as the concatenation of two odd words with different centers, and every concatenation of two odd words with different centers belongs to this language. Therefore, here is the complete CFG for our language:

$$S \to A \mid B \mid C \mid AB \mid BC \mid CA \mid BA \mid AC \mid CB$$
$$A \to a \mid aAb \mid bAc \mid cAa \mid bAa \mid aAc \mid cAb$$
$$B \to b \mid aBb \mid bBc \mid cBa \mid bBa \mid aBc \mid cBb$$
$$C \to c \mid aCb \mid bCc \mid cCa \mid bCa \mid aCc \mid cCb$$

In the PDA, for odd words we simply accept them. For even words, we make two non-deterministic guesses: the center of the first odd part and the center of the second odd part, and check that they have different centers. Here is a description in words:

(a) First, make a nondeterministic guess (without scanning anything) whether we expect to see an odd word or an even word.

(b) In the odd case, just transition to a 2-state automata that keeps track of whether even or odd number of letters have been seen. No need of the stack here.

(c) For the even case, keep scanning the word and pushing it on the stack. At some point, make a guess for a certain letter to be the center of the first odd word. Remember this letter (using the state, by having one state for each letter). Now, keep popping off the stack until it becomes empty. This marks the end of the first odd word. Now, once again start pushing the scanned letters onto the stack, and at some point make a nondeterministic guess for the center of the second odd word (only if it is different from the first center). Finally, pop the letters off the stack until the stack becomes empty, and then accept.

2. Determine if the following languages are context-free or not. If yes, provide a CFG and PDA for the same, else prove, using the Pumping Lemma, that they are not Context Free

(a) $L_1 = \{w \mid w = uu \text{ for any } u \in \Sigma^*\}$

(b) $L_2 = \{0^p | \text{p is prime}\}$

For more on L_2, look at the takeaway problems

Solution: For both sub-parts, the languages are not context-free. To show this, we play the game between the believer and the adversary, as required by the Pumping Lemma for context-free languages. Specifically, the believer thinks the language is a CFL and chooses a constant p (> 0) that is at least as large as 2^k, where k is the number of non-terminal symbols in the supposed Chomsky Normal Form grammar for L). The subsequent reasoning for the two parts is given separately below.

1. The adversary chooses $w = 0^p1^p0^p1^p \in L$. Next, the believer decomposes w as $u.v.x.y.z$ with $|v.x.y| \leq p$ and $|x| \geq 1$. Notice that this implies $v.x.y$ must be either of the form 0^i1^j, where $i > 0, j \geq 0$ or of the form 1^i0^j, where $i > 0, j \geq 0$. In both cases, if the adversary pumps v and y twice, the string $u.v^2.x.y^2.z \notin L$ (try to reason why).

2. The adversary chooses $w = 0^m$, where m is the smallest prime larger than or equal to p. Next, the believer decomposes w as $u.v.x.y.z$ where $|v.x.y| \leq p$ and $|x| \geq 1$.

Let $|v.x.y| = n$. The adversary can now pump v and y exactly $1 + m.n$ times, so that the resulting word is $0^{m+m.n}$ or $0^{m.(1+n)}$. Clearly, $m.(1+n)$ is not prime, and hence $0^{m.(1+n)} \notin L$.

3. **Deterministic Context-Free Languages**
We know that Context-Free Languages (CFL) are accepted by Push-Down Automata (NPDA), where we had allowed non-determinism in PDA transitions. In this question, we will explore Deterministic Push-Down Automata (DPDA). Recall that a (not necessarily deterministic) PDA M can be defined as a 7-tuple:

$$M = (Q, \Sigma, \Gamma, q_0, Z_0, A, \delta)$$

h where Q is a finite set of states. Σ is a finite set of input symbols. Γ is a finite set of stack symbols. $q_0 \in Q$ is the start state. $Z_0 \in \Gamma$ is the starting stack symbol. $A \subseteq Q$, where A is the set of accepting, or final, states δ is a transition function, where $\delta : (Q \times (\Sigma \cup \varepsilon) \times \Gamma) \to \mathcal{P}(Q \times \Gamma^*)$ Here, $\mathcal{P}(X)$ is the power set of a set X, and ε denotes the empty string. We say that M is a deterministic PDA (or DPDA) if it satisfies both the following conditions:

For any $q \in Q, a \in \Sigma \cup \varepsilon, x \in \Gamma$, the set $\delta(q, a, x)$ has at most one element.

For any $q \in Q, x \in \Gamma$, if $\delta(q, \varepsilon, x) \neq \emptyset$, then $\delta(q, a, x) = \emptyset$ for every $a \in \Sigma$.

We call the languages accepted by DPDAs as DCFLs (Deterministic Context Free Languages). We have studied in class that PDAs can accept by *empty stack* or by *final state*, and that these provide equivalent accepting power. Interestingly, this is not so for DPDAs, so we need to make up our mind about which acceptance criterion to use. For purposes of this question, we will use acceptance by *final state*. We investigate acceptance by empty stack in a takeaway question at the end of this tutorial.

1. Consider the language of balanced parentheses. A string of parentheses is balanced if the number of opening parentheses in any proper prefix is at least as much as the number of closing parenthesis in the same prefix. Also, the total number of opening and closing parenthesis in the entire string must be equal.

   Draw a DPDA (accepting by final state) that accepts the language of balanced parentheses strings.
2. Construct a deterministic PDA for the complement of the above language. Does this give you an idea why DCFLs (recognized by DPDAs by final state) are closed under complementation?

Solution: Please see the solution posted by us on Piazza. The DPDA shown there accepts the complement of the language of balanced parentheses. If you flip the accepting/non-accepting status of the states of the DPDA, you get the DPDA accepting the language of balance parentheses.

4. **Takeaway: The Curious Case of the Unary Alphabet** Prove that any language over a unary alphabet (the alphabet has exactly one element) is context-free if and only if it is regular.

Solution: In the interests of time, we'd like to point interested students to a nice exposition on this problem here.

5. **Takeaway: Expressions in Intermediate Code**
Consider the following context-free grammar for expressions in some (familiar) programming languages, where $\langle$expr$\rangle$ is the start symbol of the grammar.

$\langle expr \rangle$ ::= $\langle term \rangle$ '+' $\langle term \rangle$
| $\langle term \rangle$

$\langle term \rangle$ ::= $\langle factor \rangle$ '*' $\langle factor \rangle$
| $\langle factor \rangle$

$\langle factor \rangle$ ::= '(' $\langle expr \rangle$ ')'
| $\langle number \rangle$

$\langle number \rangle$ ::= [0-1]+

Now, though expressions can be a sum of as many terms, it is essential, during an intermediate step of compilation, that every expression must be a sum of at most two terms and every term must be a product of at most two terms. Draw a Deterministic PDA (definition provided in an earlier question) to recognise strings that are of the form stated above. Note that the alphabet is $\Sigma = \{0, 1, (,), *, +\}$.

Solution: This can be obtained directly from the CFG to PDA construction studied in class. Remember that the moves of such a PDA on a given input string effectively mimic a leftmost derivation of the string using the given grammer. Since there is a unique (deterministic) parse tree for a string in the given grammar, the leftmost derivation of the string is also unique. Hence the PDA obtained by transforming the CFG has a unique sequence of moves on a given string. Indeed, as can be seen by constructing the PDA, it is a DPDA.

6. **Takeaway: Null-stack DPDAs**
In Problem 3, we used acceptance by final state for a DPDA. Let's see what happens if we now allow a DPDA to accept by emptying its stack (regardless of which state it is in, when the stack becomes empty). We will call such a DPDA a *null-stack DPDA*, i.e. it's a DPDA just like we had earlier, but it accepts by emptying its stack.

   1. Prove that no null-stack DPDA can accept the language of balanced parentheses. Recall that a string of parentheses is balanced if the number of opening parenthesis in any prefix of the string is at least as much as the number of closing parenthesis in the same prefix, and the total number of opening and closing parenthesis ar equal.
   [Hint:] Can a null-stack DPDA accept two strings u and $u.v$, where one is a proper prefix of the other?
   Note: This is quite damaging news in the DPDA world, since we saw in Problem 3 that the language of balanced parentheses can be accepted by a DPDA accepting by final state. The above proof should now convince you that unlike normal PDAs, acceptance by final state and acceptance by empty stack are not equally powerful in the DPDA world.
   2. A string is said to be *minimally balanced parentheses* if the number of opening parenthesis in any proper prefix is strictly more than the number of closing parenthesis in the same prefix, and the total number of opening and closing parenthesis in the entire string are equal. Thus, $(()())$ is a minimally balanced parentheses string, but $()()$ and ε are not. Any string of balanced parentheses can be written as either ε or a concatenation of a finite number of minimally balanced parentheses strings. Show that for every given value of $k > 0$, we can construct a null-stack DPDA that accepts the language of balanced parentheses strings containing exactly k minimal valid parentheses substrings. Can we construct a null-stack DPDA if we want to accept the language of balanced parentheses containing upto (instead of exactly) k minimally valid parentheses substrings?

Solution: Suppose there was a null-stack DPDA for recognizing the language of all balanced parentheses. Then both $()$ and $()()$ must be accepted by the DPDA. However, since the DPDA accepts by empty stack, and since it has a unique sequence of moves on reading an input, its stack must necessarily become empty after reading $()$. Since a DPDA with empty stack can't make any further moves (recall each move of a PDA requires looking up the symbol at the top of the stack), the null-stack DPDA will get stuck after reading the prefix $()$ of the string $()()$. Hence, it cannot accept $()()$. But then, this null-stack DPDA doesn't recognize the set of all balanced parentheses strings.

For the second part of the question, first construct a null-stack DPDA accepting a minimally balanced parentheses string. This should be straightforward, and is left as an exercise. Now take k such DPDAs, say $P_1, \ldots P_k$. Consider their stack alphabets as disjoint, say $\Gamma_1, \ldots \Gamma_k$. Let $X_{0,1}, \ldots X_{0,k}$ denote the initial symbol on the stack for $P_1, \ldots P_k$ respectively. Now create a new DPDA P whose stack alphabet is $\Gamma_1 \cup \cdots \Gamma_k \cup \{Z\}$, where Z is a new bottom of stack marker symbol that is not in $\Gamma_1 \cup \cdots \Gamma_k$. The new DPDA P uses Z as the initial symbol in its stack. It also has a new start state. The DPDA P works as follows:

From the start state of P, there is only a single transition that consumes ε, pops Z and pushes $X_{0,1}Z$ into the stack. It then transitions to the start state of P_1. From every state of P_i, we now add a transition that consumes ε and on seeing Z as the top of the stack, pops Z, pushes $X_{0,i+1}Z$ into the stack and transitions to the start state of P_{i+1}. Finally, from every state in P_k, we add a transition that consumes ε, pops Z from the stack and doesn't push anything, thereby emptying the stack.

We leave it for you to complete the reasoning that this accepts all and only strings that are concatenations of k minimal valid parentheses substrings.

Clearly, we can't construct a null-stack DPDA that accepts both $()$ and $()()$ – we've reasoned earlier about this. So we can't construct a null-stack DPDA that accepts the language of balanced parentheses containing upto (instead of exactly) k minimally valid parentheses substrings.