

## Problem Set 3 Comments

**Problem 1: Nondeterministic Pushdown Automata.** Draw a nondeterministic pushdown automaton that recognizes the language  $\{w0w^R \mid w \in \{0,1\}^*\}$ . The fewer states you use, the better.

**Answer:** This problem is very similar to Figure 2.19 in Sipser, except for the addition of the 0 in between  $w$  and  $w^R$ . We can accommodate that in the PDA by modifying the arrow (which was  $\epsilon, \epsilon \rightarrow \epsilon$ ) between states  $q_2$  and  $q_3$  to consume a 0:  $0, \epsilon \rightarrow \epsilon$ .

**Problem 2: Defining Regular Expressions.** Definition 1.52 provides a formal definition of a *regular expression*. Rewrite that definition as a context-free grammar. The set of terminals is implied by the definition:  $\Sigma = \{a, \epsilon, \emptyset, \cup, \circ, *, (, )\}$  (where  $a$  represents any symbol in the alphabet, and  $\epsilon$  means the epsilon symbol, not the empty string; if you need to represent the empty string in your grammar use  $\lambda$  to avoid confusion with the  $\epsilon$  symbol that can appear in a regular expression). You may use as many variables as you need to be clear, but should give them sensible names. Your grammar should be able to generate all possible regular expressions, but no strings that are not regular expressions.

**Answer:** We can produce the CFG directly from the definition.

$$\begin{aligned}
 R &\rightarrow a \text{ (for each } a \text{ in the alphabet } \Sigma) \\
 R &\rightarrow \epsilon \\
 R &\rightarrow \emptyset \\
 R &\rightarrow (R \cup R) \\
 R &\rightarrow (R \circ R) \\
 R &\rightarrow (R^*)
 \end{aligned}$$

**Problem 3: Context-Free Grammars.** Consider the grammar  $G$  below, which describes the same language as Problem 1d from PS1: ( $S$  is the start symbol, and 0 and 1 are terminals)

$$\begin{aligned}
 S &\rightarrow \epsilon \\
 S &\rightarrow S00 \\
 S &\rightarrow 11S \\
 S &\rightarrow 0S1 \\
 S &\rightarrow 10S
 \end{aligned}$$

- a. Show that the string 111000 can be produced by  $G$  by showing a derivation that produces it.

**Answer:**  $S \rightarrow 11S \rightarrow 1110S \rightarrow 1110S00 \rightarrow 111000$

- b. How many different derivations are there in  $G$  to produce 111000? (Support your answer with a clear argument.)

**Answer:** There are three possible derivations of 111000. The other two are:

$$\begin{aligned} S &\rightarrow S00 \rightarrow 11S00 \rightarrow 1110S00 \rightarrow 111000 \\ S &\rightarrow 11S \rightarrow 11S00 \rightarrow 1110S00 \rightarrow 111000 \end{aligned}$$

We know there are no more possible derivations since the string starts with 11 (which can only be produced using the  $S \rightarrow 11S$  rule). So, this rule must be used when there is an  $S$  at the left end of the string. We have two choices: using  $S \rightarrow S00$  first, and then using  $S \rightarrow 11S$ ; or, using  $S \rightarrow 11S$  first. Any other productions put a 0 to the left of the leftmost  $S$ , so cannot produce a string that starts with 11. For the next step after starting with  $S \rightarrow S00$ , there is only one choice that works: using  $S \rightarrow 11S$  to get 11S00. The other three options produce strings that either cannot end in 1000 ( $S \rightarrow S00$  produces  $S0000$  which ends in four 0s) or that cannot start with 111 ( $S \rightarrow 0S1$  and  $S \rightarrow 10S$ ) put a 0 in the first or second symbol, with no variables before it. So, the only choice is  $S \rightarrow 11S$ , and from 11S00 there is only one possible choice that works. We use a similar line of reasoning for the other option, starting with  $S \rightarrow 11S$ . For there we can eliminate the  $S \rightarrow 11S$  option which would produce 1111S since it starts with four 1s, and the  $S \rightarrow 0S1$  option, since it produces 110S1 which cannot produce a string that starts with 111. The other two options work, as shown in the derivations. After the second step, there is only one choice for each following step.

- c. What is the fewest number of rules that can be added to  $G$  to produce a grammar that describes the language of all even-length strings in  $\{0, 1\}^*$ ? (Your answer should include the rules to add.)

**Answer:** It is easy to see that adding these two rules can make the grammar cover all even-length strings:

$$\begin{aligned} S &\rightarrow 00S \\ S &\rightarrow 01S \end{aligned}$$

With these two rules, and the given rules, we have covered all four possible two-symbol sequences before an  $S$ .

But, it is also possible to cover all even-length strings by only adding just the first rule:

$$S \rightarrow 0S0$$

Proving that the resulting six rules can produce all even-length strings can be done using induction on the length of the string. The six rules (ordered by where  $S$  appears on the right side) are:

$$\begin{aligned} (1) S &\rightarrow \epsilon \\ (2) S &\rightarrow S00 \\ (3) S &\rightarrow 0S1 \end{aligned}$$

- (4)  $S \rightarrow 0S0$
- (5)  $S \rightarrow 10S$
- (6)  $S \rightarrow 11S$

**Basis.** The rules can produce all strings of length 2. There are four possible strings of length two: 00, 01, 10, 11. Here is how to produce them:  $S \rightarrow_4 S00 \rightarrow_1 00$ ;  $S \rightarrow_3 0S1 \rightarrow_1 01$ ;  $S \rightarrow_5 10S \rightarrow_1 10$ ;  $S \rightarrow_6 11S \rightarrow_1 11$ .

**Induction.** We need to prove that if the grammar can generate all even-length strings of length  $k$ , it can generate all even-length strings of length  $k + 2$ . We can prove this by showing all even-length strings of length  $k + 2$  can be generated using the rules, starting from all even-length strings of length  $k$ . Suppose  $s$  is some string of length  $k + 2$ . There are two cases to consider:

- (a)  $s$  starts with a 0. Then either  $s = 0t0$  or  $s = 0t1$  where  $t$  is some even-length string of length  $k$ . By the inductive hypothesis, we are assuming  $S \rightarrow^* t$  since we assumed the grammar can produce all even-length strings of length  $k$ . So, there is some derivation  $S \rightarrow_i \dots \rightarrow t$  that produces  $t$ . We can add  $S \rightarrow_4 0S0$  to the beginning of that derivation to produce  $0t0$ . We can add  $S \rightarrow_3 0S1$  to the beginning of that derivation to produce  $0t1$ . This covers all even-length strings that start with a 0.
- (b)  $s$  starts with a 1. Then either  $s = 10t$  or  $s = 11t$  where  $t$  is some even-length string of length  $k$ . Using the same reasoning as the previous case, we can produce both of these. The first, by adding  $S \rightarrow_5 10S$  to the beginning of the derivation that produces  $t$ , the by adding  $S \rightarrow_6 11S$  to the beginning of the derivation that produces  $t$ .

Note that no part of our proof used rule 2, so that rule is unnecessary. We have proved that rules 1, 3, 4, 5, and 6 by themselves can generate all possible even-length strings.

**Problem 4: Regular Grammars.** As discussed in Lecture 8, a *regular grammar* is a replacement grammar in which all rules have the form  $A \rightarrow aB$  or  $A \rightarrow a$  where  $A$  and  $B$  represent any variable and  $a$  represents a terminal. Prove that all regular languages can be recognized by a regular grammar.

**Answer:** We have to prove that for any regular language  $L$ , there is a regular grammar that produces  $L$ . Since  $L$  is regular, we know there is a DFA,  $M$ , that recognizes  $L$ . We will show that all regular languages can be recognized by a regular grammar by showing a construction that converts a DFA into a regular grammar.

The DFA,  $M = (Q, \Sigma, \delta, q_0, F)$  recognizes  $L$ . We produce a regular grammar  $G = (V, \Sigma, R, S)$  from  $M$  as follows:

- $V = Q$  (the variables in  $G$  are the names of the states in  $M$ )
- $\Sigma = \Sigma$  (the terminals in  $G$  are the symbols in  $M$ )

$$\begin{aligned}
S &= q_0 \text{ (the start variable for the grammar is the start state of } M) \\
R &= \{A \rightarrow aB \mid \text{for all } A \in Q, a \in \Sigma \text{ where } \delta(A, a) = B\} \\
&\quad \cup \{A \rightarrow a \mid \text{for all } A \in Q, B \in F, a \in \Sigma \text{ where } \delta(A, a) = B\}
\end{aligned}$$

$G$  is a regular grammar since all rules in  $R$  has the form  $A \rightarrow aB$  or  $A \rightarrow a$ . The tricky part is dealing with the accepting states, which should corresponding to completing a derivation. Since we defined the regular grammar to not include  $A \rightarrow \epsilon$  rules (in fact, a regular grammar can be defined to include such rules also, but they are not necessary), we need to make the corresponding grammar rule end the derivation (by not including any variables in the right side) instead of going into the accepting state ( $B$  in the definition of  $R$ ).

**Problem 5: Pumping Lemma for Context-Free Languages.** For each part, either argue that the language is context-free (ideally, by showing how a PDA could recognize it or a CFG could generate it) or use the pumping lemma to show it is not context free.

a.  $\{0^i 1^i 0^i\}$

**Answer:** This is very similar to Sipser's Example 2.36.  $\{0^i 1^i 0^i\}$  is **not** a context-free language. Prove by contradiction using the pumping lemma for context-free languages.

Assume  $\{0^i 1^i 0^i\}$  is a CFL with a pumping length  $p$ . Let  $s = 0^p 1^p 0^p$ . Consider all possible ways of dividing  $s = uvxyz$ . There are two cases to consider:

- Case 1. Both  $v$  and  $y$  contain only one type of character (each on is either all 0s or all 1s). Then  $uv^0xy^0z$  will have fewer of one or two characters but not all three since  $|vxy| \leq p$ , so it is not in the language as required by the pumping lemma.
- Case 2. One of  $v$  or  $y$  contains two different characters. Then  $uv^2xy^2z$  will contain some zeros between ones which is not in the language.

Since case 1 and case 2 cover all possible ways of splitting  $s$ , and both case 1 and 2 lead to contradictions, we have proven the language is not a CFL.

b.  $\{1^a + 1^b = 1^c \mid a \geq 0, b \geq 0, c \geq 0, a + b = c\}$

**Answer:**  $\{1^a + 1^b = 1^c \mid a \geq 0, b \geq 0, c \geq 0, a + b = c\}$  is a context-free language. We could construct a PDA to recognize it by pushing all the 1s before the  $=$ , and then popping them to count  $c$ . We could also define a CFG that generates this language:

$$\begin{aligned}
S &\rightarrow 1S1 \mid +M \\
M &\rightarrow 1M1 \mid =
\end{aligned}$$

c.  $\{0^i 1^j 2^k \mid i < j < k\}$  (hint: compare to Example 2.37)

**Answer:**  $\{0^i 1^j 2^k \mid i < j < k\}$  is **not** a context-free language. We prove by contradiction using the pumping lemma for CFLs.

Assume  $0^i 1^j 2^k | i < j < k$  is a CFL with pumping length  $p$ . Let  $s = 0^{(p-1)} 1^{p^2} 2^{(p+1)}$  and consider all possible ways of dividing  $s = uvxyz$ . There are five cases:

- Case 1. Both  $v$  and  $y$  contain only 0s. Then,  $uv^2xy^z$  would have more than  $\geq p$  0s which is not in the language.
- Case 2. Both  $v$  and  $y$  contain only 1s. Then,  $uv^0xy^0z$  would have  $\leq p - 1$  1s which is not in the language.
- Case 3. Both  $v$  and  $y$  contain only 2s. Then,  $uv^0xy^0z$  would have  $\geq p$  2s which is not in the language.
- Case 4. Both  $v$  and  $y$  contain 0s and 1s only. Then,  $uv^2xy^2z$  would have  $\geq p + 1$  1s which is not in the language.
- Case 5. Both  $v$  and  $y$  contain 1s and 2s only. Then,  $uv^0xy^0z$  would have  $\leq p - 1$  1s which is not in the language.

Since  $|vxy| \leq p$  is required by the pumping lemma, there is no way for  $v$  and  $y$  to contain all three symbols, so these five cases cover all possibilities. Thus, the language is not context-free.

**Problem 6: Parsing.** Below is a slightly simplified excerpt from the actual Java grammar specification (from [http://java.sun.com/docs/books/jls/second\\_edition/html/syntax.doc.html](http://java.sun.com/docs/books/jls/second_edition/html/syntax.doc.html), Chapter 18). I have changed the syntax to match the context-free grammar notation used in Sipser and the class.

<i>Expression</i>	→	<i>Expression1 OptAssignmentOperator</i>
<i>OptAssignmentOperator</i>	→	$\epsilon$   <i>AssignmentOperator Expression1</i>
<i>Expression1</i>	→	<i>Expression2 OptExpression1Rest</i>
<i>OptExpression1Rest</i>	→	$\epsilon$   <i>Expression1Rest</i>
<i>Expression1Rest</i>	→	? <i>Expression : Expression1</i>
<i>AssignmentOperator</i>	→	=
<i>Expression2</i>	→	<i>Expression3 OptExpression2Rest</i>
<i>OptExpression2Rest</i>	→	$\epsilon$   <i>Expression2Rest</i>
<i>Expression2Rest</i>	→	<i>InfixExpressionList</i>
<i>InfixExpressionList</i>	→	$\epsilon$   <i>InfixExpression InfixExpressionList</i>
<i>InfixExpression</i>	→	<i>InfixOp Expression3</i>
<i>InfixOp</i>	→	&&   ==   +
<i>Expression3</i>	→	<i>Primary SelectorList</i>
<i>Primary</i>	→	( <i>Expression</i> )   <b>Identifier</b>   <b>Literal</b>
<i>SelectorList</i>	→	$\epsilon$   <i>Selector SelectorList</i>
<i>Selector</i>	→	[ <i>Expression</i> ]   . <b>Identifier</b>

We use **Identifier** to mean any valid Java identifier (see Section 3.8 of the Java Language Specification for the grammar for Identifiers) and **Literal** to mean any numeric literal.

For the examples, assume any single alphabet letter is an **Identifier** and that all variables are declared with type `boolean`, and any number, `true`, and `false` are **Literals**. (The conditional expression,  $Expression_{pred} ? Expression_{consequent} : Expression_{alternate}$ , is evaluated by first evaluating  $Expression_{pred}$ , which must evaluate to a boolean. If it evaluates to `true`, then the value of the conditional expression is the value obtained by evaluating  $Expression_{consequent}$  (and  $Expression_{alternate}$  is not evaluated). If it evaluates to `false`, then the value of the conditional expression is the value obtained by evaluating  $Expression_{alternate}$  (and  $Expression_{consequent}$  is not evaluated).)

- a. Show a derivation for the expression: `a . f`

**Answer:**

```

Expression
→ Expression1 OptAssignmentOperator
→ Expression1
→ Expression2 OptExpression1Rest
→ Expression2
→ Expression3 OptExpression2Rest
→ Expression3
→ a SelectorList
→ a . f SelectorList
→ a . f

```

- b. Consider the following Java expression:

```
true ? false ? true == true : false : false == false
```

which evaluates to `false`. By adding only parentheses, transform it into a grammatical Java expression that evaluates to `true`.

**Answer:**

```
(true ? false ? true == true : false : false) == false
```

- c. Explain how you would change the grammar rules so all *Expressions* that can be produced by the above grammar are still value expressions, but the original expression in the previous part evaluates to `true` in the modified grammar.

**Answer:** This is a tricky one. Essentially, we want to change the grammar so the `==` gets bound after the `?`. To do this, we need to swap  $Expression_3$  and  $Expression_1$  in the grammar. To keep things simple, we do this by adding a production for *Primary* that produces the *ConditionalExpression*. We change the *OptAssignmentOperator* rule to be:

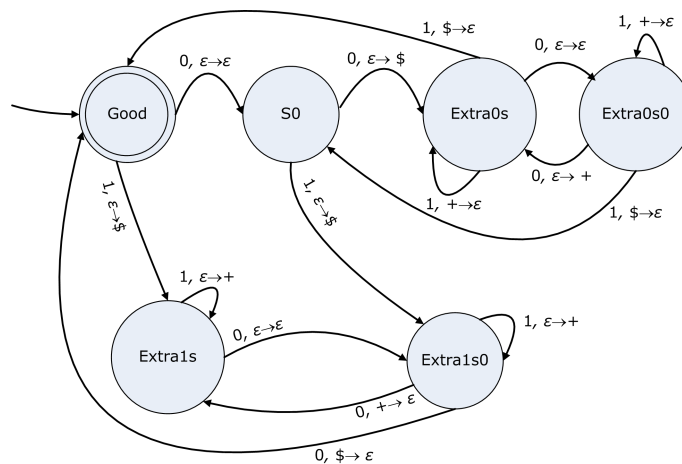
```
OptAssignmentOperator → ε | AssignmentOperator Expression2
```

eliminating *Expression1*. Then, add a new rule for *Primary*:

$$\textit{Primary} \rightarrow \textit{Expression} ? \textit{Expression} : \textit{Expression}$$

Note that this makes the grammar ambiguous, but one of the possible parses of the expression now corresponds to the parenthesize expression from the previous part (since we parenthesized *Expression* was a *Primary*, which can now be a conditional expression (without needing the parentheses. Finding a way to do this without making the grammar ambiguous is more challenging.

**Problem 7: Deterministic Pushdown Automata.** Precisely describe the language recognized by the deterministic pushdown automata shown below. (The state names are intended to be somewhat helpful, but not completely revealing. If your answer is correct, you should be able to find a simple way to describe the language.) (Hint: try comparing this machine to the machine from Class 6 from class and Example 3 of the notes.)



**Answer:**  $\{w \mid w \in \{0, 1\}^* \text{ and } w \text{ contains twice as many 0s as 1s.}\}$