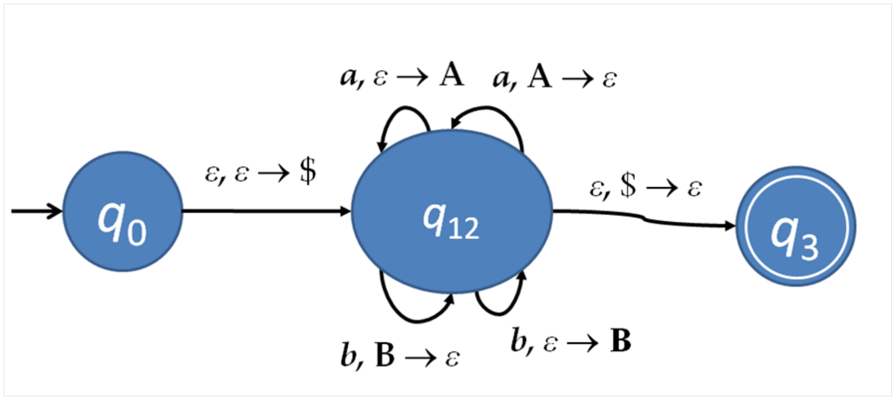
University of Virginia - cs3102: Theory of Computation Spring 2010

PS3 - Comments

Average: 46.6 (full credit for each question is 55 points)

Problem 1: Mystery Language. (Average 8.5 / 10) In Class 7, we considered how the language would change if states q_1 and q_2 from the NPDA on slide 7 were merged, producing the NPDA not shown here but included in the PS3 PDF handout.


$a, \varepsilon \rightarrow$ A
a, A $\rightarrow \varepsilon$
$\varepsilon, \varepsilon \rightarrow \$$
q_0
q_{12}
$\varepsilon, \$ \rightarrow \varepsilon$
q_3
b, B $\rightarrow \varepsilon$
$b, \varepsilon \rightarrow$ B

Describe precisely the language accepted by this nondeterministic PDA.

Answer: There are several ways to precisely describe a language. A good description makes it clear exactly what strings are in the language. One way to describe this language is by providing a context-free grammar that produces the language:

$$S \quad \rightarrow \quad SS \mid aSa \mid bSb \mid \epsilon$$

Another way would be do describe it in English, but this is hard to do precisely:

> The language of all strings consisting of as and bs where the as and bs are "balanced". We can consider each a to be either a (or a), and each b to be either a [or a]. If there is some way to assing the parentheses to the as and brackets to the bs that produces a well-balanced string (that is, follows the rules typical in programming languages for matching parentheses and brackets) the string is in the language.

Challenge Bonus. Is there a *deterministic* PDA that recognizes the same language? Include a convincing proof that supports your answer.

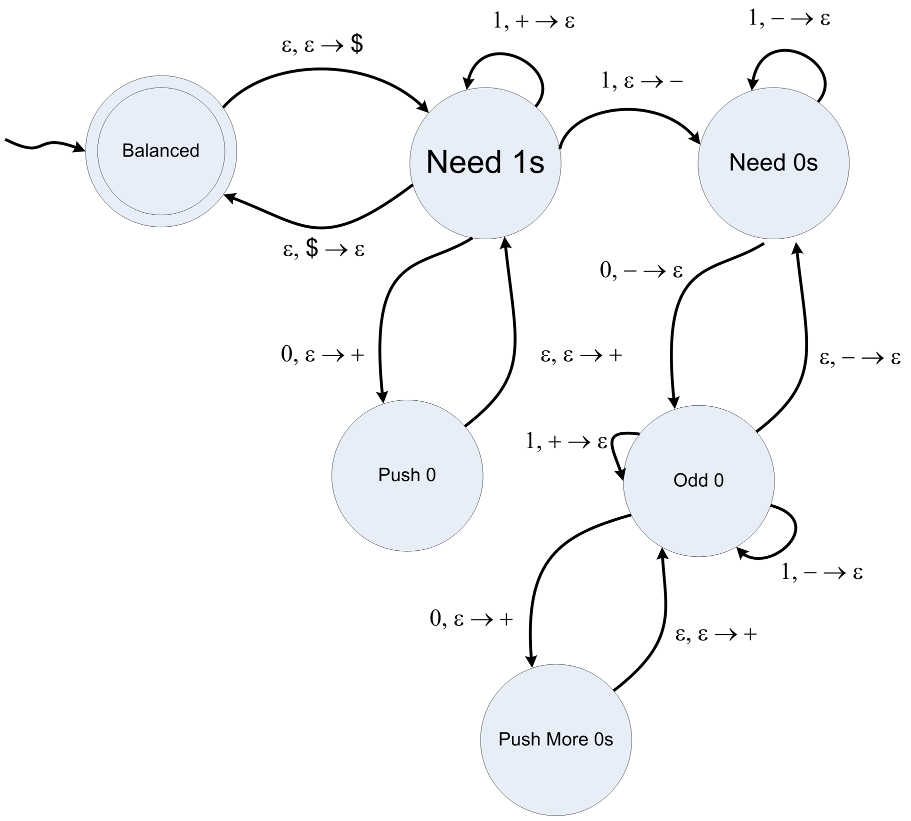
Answer: The language is inherently ambiguous, so there is no deterministic PDA that can recognize it. Note that the language is the Kleene closure of the language of palindromes, but the language ww^R cannot be recognized by a deterministic PDA.

Problem 2: NPDAs and CFGs. (Average 8.5 / 10) Describe a NPDA that recognizes the language produced by this context-free grammar:

$$\begin{array}{lll} S & \rightarrow & 1A \mid A1 \mid 0B \mid B0 \mid \epsilon \\ A & \rightarrow & 0S1 \mid 1S0 \mid CS \mid SC \\ B & \rightarrow & S11 \mid 1S1 \mid 11S \\ C & \rightarrow & 01 \mid 10 \end{array}$$

Answer: Although it is possible to follow the mechanical construction used in the proof that NPDAs and CFGs are equivalent, this would be very painful and tedious, and exceedingly unlikely to do correctly by hand. A better approach is to try to understand the language produced by the CFG. It is the language of all strings with twice as many 1s as 0s. We can see this by examining the grammar rules. The variable S produces either the empty string, or $1A$ or $A1$, or $0B$ or $B0$. The A variable produces all arrangements of S with one 0 and one 1. So, $1A$ and $A1$ produce all arrangments where there is a 1 on one of the outsides and there are two 1s and one 0 with an S in any arrangment. For the $0B$ and $B0$ productions, B produces all arrangements of S and two 1s. For $S \rightarrow B0 \mid 0B$, produces all arrangements with two 1s and one 0 and S where there is a 0 on one of the outsides. Combined with the A productions, this produces all strings with twice as many 1s as 0s.

Now we understand the language, producing an NPDA that recognizes is is similar to the NPDA we produces in class for recognizing the language with as many 1s as 0s. The difference is we need to push/pop two stack symbols for each 0, and only pop/push one for each 1. The tricky part is dealing with the case where we see a 0 but there is only one stack symbol to pop. We deal with this by adding an extra state that keeps track of the deficeit of one 1.



Problem 3: Context-Free Grammars. Provide a context-free grammar that recognizes the language:

$$\{w | w \in \{a, b\}^* \wedge w \text{ contains more } a\text{s than } b\text{s.}\}$$

Use as few nonterminals as possible.

Answer: See the answer to question 2.6a in the book (p. 132).

Problem 4: Priming the Pump Redux. Prove the language, $PRIMES$, is not context-free:

$$PRIMES = \{1^n | n \text{ is a prime number}\}$$

Answer: Our answer is very similar to the question on Problem Set 2.

Assume $PRIMES$ is context-free with some pumping length p. We use the pumping lemma for context-free languages to obtain a contradiction.

Choose $s = 1^q$ where q is some prime number greater than p. (Note that we cannot choose $s = 1^p$ since p might be composite. But, since there is no maximum prime, we know that for any p such a q must exist.)

The pumping lemma for context-free languages requires that s can be broken into $uvxyz$ with $|vy| \geq 1$ and $|vxy| \leq p$. Note that since the string is all 1s, however the pieces are divided, we can rearrange them without changing the string:

$uvxyz = uxvyz = rtz$ where $r = ux$, and $t = vy$. Note that the two parts that can be pumped, vy are now in t, so pumping $v^i y^i$ is the same as pumping t^i. Thus, we have taken whatever 5-part division is selected, and converted it to the same 3-part division as needed in the regular languages pumping lemma. In the last problem set, we saw how to get a contridiction by setting $i = q + 1$.

We have a contradiction, so $PRIMES$ is not context-free.

Problem 5: Closure Properties.

a. Prove that the context-free languages are closed under concatenation.

Answer: Given two context-free language, A and B, we can construct a context-free grammar G_{AB} that is the concatenation of the languages. Since A and B are context-free, there exist context-free grammars $G_A = (V_A, \Sigma, R_A, S_A)$ and $G_B = (V_B, \Sigma, R_B, S_B)$ that produce each language. First, rename the variables in G_B so they are disjoint from the variables in G_A. Then, construct

$$G_{AB} = (V_A \cup V_B \cup \{S_{AB}\}, \Sigma, R_A \cup R_B \cup S_{AB} \rightarrow S_A S_B, S_{AB})$$

We have added the rules $S_{AB} \rightarrow S_A S_B$ to the language. S_A is the start variable for G_A, so produces all strings in A; S_B is the start variable for G_B, so produces all strings in B. Thus, S_{AB} produces all strings in the concatenation language AB.

b. Prove that the intersection of a context-free language with a regular language is always a context-free language.

Answer: We use proof by construction. Since the context-free language, A, is context-free there is some nondeterministic pushdown automaton, $M_A = (Q_A, \Sigma, \Gamma, \delta_A, q_{0A}, F_A)$ that recognizes A. Since B is a regular language, there exists some DFA, $M_B = (Q_B, \Sigma, \delta_B, q_{0B}, F_B)$ that recognizes B. We show how to construct a nondeterministic pushdown automata, $M_{A \cap B}$, that recognizes the language $A \cap B$. The key point to observe is that $M_{A \cap B}$ should accept a string if and only if both M_A and M_B would accept the string. So, the states of $M_{A \cap B}$ need to keep track of both the state of M_A and the state of M_B. Thus, we label the states using pairs from Q_A and Q_B:

$$M_{A \cap B} = (Q_A \times Q_B, \Sigma, \Gamma, \delta_{AB}, q_{0A} \times q_{0B}, F_A \times F_B)$$

The new $\delta_{AB} : Q_A \times Q_B \times \Sigma_\epsilon \times \Gamma_\epsilon \rightarrow Q_A \times Q_B \times \Gamma_\epsilon$ is defined by:

$$\delta_{AB}(q_a, q_b, \sigma, \gamma) = (q_a n, \delta_B(q_b, \sigma), \gamma_{an})$$

where $(q_{an}, \gamma_{an}) = \delta_A(q_a, \sigma, \gamma)$.

c. Prove that the context-free languages are **not** closed under intersection.

Answer: We prove by finding a counter-example. We show that there is some non-context-free language that is the intersection of two context-free languages. The non-context-free language is $\{a^n b^n c^n | n \geq 0\}$. We proved in class that this language is not context-free. This language is the intersection of two context-free languages: $\{a^n b^n c^* | n \geq 0\}$ and $\{a^* b^m c^m | m \geq 0\}$. (We use different variables for n and m to emphasize that there is no connection between the values in the two languages.)

Problem 6: Parsing. Below is a slightly simplified excerpt from the actual Java grammar specification (from http://java.sun.com/docs/books/jls/third_edition/html/syntax.html#18.1, Chapter 18). I have changed the syntax to match the context-free grammar notation used in Sipser and the class.

Expression	$\rightarrow$	*Expression1 OptAssignmentOperator*
OptAssignmentOperator	$\rightarrow$	ϵ | *AssignmentOperator Expression1*
Expression1	$\rightarrow$	*Expression2 OptExpression1Rest*
OptExpression1Rest	$\rightarrow$	ϵ | *Expression1Rest*
Expression1Rest	$\rightarrow$	**?** *Expression* **:** *Expression1*
AssignmentOperator	$\rightarrow$	**=**
Expression2	$\rightarrow$	*Expression3 OptExpression2Rest*
OptExpression2Rest	$\rightarrow$	ϵ | *Expression2Rest*
Expression2Rest	$\rightarrow$	*InfixExpressionList*
InfixExpressionList	$\rightarrow$	ϵ | *InfixExpression InfixExpressionList*
InfixExpression	$\rightarrow$	*InfixOp Expression3*
InfixOp	$\rightarrow$	**||** | **&&** | **==** | **+**
Expression3	$\rightarrow$	*Primary SelectorList*
Primary	$\rightarrow$	**(** *Expression* **)** | **Identifier** | **Literal**
SelectorList	$\rightarrow$	ϵ | *Selector SelectorList*
Selector	$\rightarrow$	**[** *Expression* **]** | **. Identifier**

The terminal **Identifier** is any valid Java identifier (see Section 3.8 of the Java Language Specification for the grammar for Identifiers) and **Literal** is any literal.

a. Consider the following Java expression:

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

which evaluates to `false`. The Boolean values `true`, and `false` are **Literal**s. 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).

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

Answer:

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

The expression in parentheses evaluates to `false+, so the whole expression evaluates to true+.`

b. Explain how to change the grammar rules so the original expression in the previous part evaluates to `true`. Your new grammar should produce exactly the same language as the original grammar. It is acceptable if your answer leads to an ambiguous grammar, as long as one possible parse of the expression in your grammar evaluates to `true`. (**Challenge Bonus.** Produce an unambiguous grammar for this question. It should produce exactly the same language as the original grammar, but the only possible parse of the expression evaluates to `true`.)

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 *Expression3* and *Expression1* 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:

$$\textit{OptAssignmentOperator} \rightarrow \epsilon \mid \textit{AssignmentOperator Expression2}$$

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

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

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.