Ungraded Homework Solutions

CSC 135 – Programming Languages and Computer Theory

If you see any mistakes, please let me know.

1) *Design both a CFG and a PDA for each of the following languages over alphabet* $\{0, 1\}$.

Here are example grammars (there's usually more than one way to solve these problems). The PDAs are below. Note that the first of these languages is regular, so the PDA just ignores the stack.

$\{w \mid w$ has at least three 1s$\}$

$$\begin{aligned} S &\rightarrow T1T1T1T \\ T &\rightarrow 0T \mid 1T \mid \lambda \end{aligned}$$

$\{w \mid$ the length of w is odd and the middle symbol is $0\}$

$$S \rightarrow 0S0 \mid 0S1 \mid 1S0 \mid 1S1 \mid 0$$

$\{0^m1^n \mid m \neq n\}$. I'll do two helper grammars: A has the same or more 0s than 1s and B has the same or more 1s than 0s. S forces at least one or the other to ensure $m \neq n$.

$$\begin{aligned} S &\rightarrow 0A \mid B1 \\ A &\rightarrow 0A1 \mid 0A \mid \lambda \\ B &\rightarrow 0B1 \mid B1 \mid \lambda \end{aligned}$$

2) *It is rather easy to tell if a sequence of parentheses is legal (ie, properly balanced). When looking at the parentheses from left to right, an opening parenthesis is always legal, but a closing one is only legal if more opening parentheses than closing parentheses have occurred before it. So, after "(()(" either an opening or closing parentheses is legal, but after "(()())" only an opening parentheses is legal. Also, a legal sequence of parentheses has the same number of opening and closing parentheses. Design a PDA that recognizes legal sequences of parentheses.*

Focussing on strings you want to accept, a reasonable algorithm would be (i) push each '(' from input onto the stack, (ii) pop each '(' from the stack when ')' is seen in the input, and (iii) allow a transition to an accept state when the stack is empty (the string will be accepted in this case if the input is all consumed). Bad strings take care of themselves: If ')' is on the input and the stack is empty, it indicates that we've seen more ')' than '(' at that point; and if we see more of '(' than ')', the stack will not be empty when the input is. See drawing below.

3) *Design a context-free grammar that generates all legal sequences of parentheses.*

A string of balanced parens begins with an open paren and has a matching close paren, and between them is a balanced string of parens. Another string of balanced parens comes afterward.

$$S \rightarrow (S)S \mid \lambda$$

As an example leftmost derivation, let's verify that "(()())" is in the language of the grammar: $S \rightarrow (S)S \rightarrow ((S)S)S \rightarrow (()S)S \rightarrow (()(S)S)S \rightarrow (()()S)S \rightarrow (()())S \rightarrow (()())$.

4) *Here is a context-free grammar that shows that left-to-right associativity, operator precedence and parentheses can be captured with an unambiguous context free grammar.* $S \rightarrow E$, $E \rightarrow E + T \mid T$, $T \rightarrow T \times F \mid F$, $F \rightarrow (E) \mid a$. *Draw both a parse tree and a leftmost derivation for each of the following: (i)* $a + a \times a$, *(ii)* $(a + a) \times a$.

Here is one leftmost derivation. Ask in class or office hour if you wish to see any of the other solutions: (i) $S \rightarrow E \rightarrow E + T \rightarrow T + T \rightarrow F + T \rightarrow a + T \rightarrow a + T \times F \rightarrow a + F \times F \rightarrow a + a \times F \rightarrow a + a \times a$.

5) *The following context-free grammar generates the same language. Demonstrate that it is ambiguous by finding a string in its language that has two different parse trees.* $S \rightarrow E, E \rightarrow E + E \mid E \times E \mid (E) \mid a$.

The simplest is $a + a \times a$. You will get two different leftmost derivations depending on your choice of first E production: $E \to E + E$ or $E \to E \times E$. Same goes for a parse tree. Depending on which production you apply first in building your tree, you'll get two different structures for the same string.

6) *Following the context-free grammar to PDA conversion process seen in class convert your grammar from Problem 3 into a PDA. Simulate the acceptance of "(()())" by giving a sequence of instantaneous descriptions from the start state to the final state, consuming all the input.*

See PDA drawing below. Note however that the self-loop is missing one triple. It should have λ, S, λ too. Here's a sequence of "instantaneous descriptions" showing its operation. (Each triple below is (current state, remaining input, current stack).

$(1, (()()), \oslash) \vdash (2, (()()), S\oslash) \vdash (2, (()()), (S)S\oslash) \vdash (2, ()()), S)S\oslash) \vdash (2, ()()), (S)S)S\oslash) \vdash (2,)()), S)S)S\oslash) \vdash (2,)()),)S)S\oslash) \vdash (2, ()), S)S\oslash) \vdash (2, ()), (S)S)S\oslash) \vdash (2,)), S)S)S\oslash) \vdash (2,)),)S)S\oslash) \vdash (2,), S)S\oslash) \vdash (2,),)S\oslash) \vdash (2, \lambda, S\oslash) \vdash (2, \lambda, \oslash) \vdash (3, \lambda, \oslash)$.

7) *Use the pumping lemma to argue that the language $L = \{a^i b^j c^k \mid i \leq j \leq k\}$ is not context-free.*

Assume for contradiction that L is context free.
This means there's a pumping length p.
Consider string $w = a^p b^p c^p$ which is in L.
Pumping lemma says $w = uvxyz$ exists where $|vxy| \leq p$, v or y is not empty, and $uv^ixy^iz \in L$ for $i \geq 0$.
Because v or y is not empty, uv^ixy^iz will change the number of as, bs, and/or cs as i changes.
Because $|vxy| \leq p$, vxy cannot contain both as and cs.
If vxy omits as, then uxz will have fewer bs or c's than as meaning it is not in L.
If vxy omits cs, then $uvvxyyz$ will have more as or b's than cs meaning it is not in L.
In either case, the pumped string is not in L even though the pumping lemma says it must.