Assignment 3

COMPSCI 3331

Due: November 22, 2022 at 11:59 PM

(4 marks) 1. Construct a context-free grammar for the language $L = \{x\#1^n : x \in \{a,b\}^* \text{ and } n - 1 \leq |x|_a \leq n+1\}$ over the alphabet $\Sigma = \{a,b,1,\#\}$.

Solution:

$$
\begin{aligned}
S &\rightarrow aS1 \\
S &\rightarrow bS \\
S &\rightarrow T1 \mid \# \mid aT \\
T &\rightarrow aT1 \\
T &\rightarrow bT \\
T &\rightarrow \#
\end{aligned}
$$

The grammar is broken into three parts, generated by each of three productions that converts S to T. Notice that the number of b's is never constrained in any production. At any point, any number of b's can be generated. This is because the conditions only relate to the number of a's in x.

- Using $S \rightarrow T1$ generates the language $\{x\#1^n : |x|_a = n-1\}$. To do this, first S generates words of the form $xS1^i$ where $i = |x|_a$. Then after applying the rule to get T, we get a word of the form $xT1^i$ where $i = |x|_a + 1$, since this rule adds a single 1 without any matching a. Then T produces a's and 1's in equal number, before generating #. Thus, when using this production, S generates those words $x\#1^n$ where $n = |x|_a + 1$.

- Using $S \rightarrow \#$ generates the langauge $\{x\#1^n : |x|_a = n\}$. This is similar to the previous construction, but without the ability to generate any extra 1, since we end with $S \rightarrow \#$..

- Using $S \rightarrow aT$ generates the language $\{x\#1^n : |x|_a = n+1\}$. Like before, we first note that S generates words of the form xS_31^i where $i = |x|_a$. Then we use the rule $S \rightarrow aT$ that gives words of the form $xT1^i$ where $i = |x|_a - 1$. Finally, the rules for T produce a's and 1's in equal number before generating #. Thus, S generates those words $x\#1^n$ where $n = |x|_a + 1$ when we use the rule $S \rightarrow aT$.

(4 marks) 2. Convert the following grammar to CNF

$$\begin{array}{rcl} S & \to & aSa \\ S & \to & B \\ B & \to & bbCaa \\ B & \to & bb \\ C & \to & cC \\ C & \to & \varepsilon \end{array}$$

The solution is:

$$\begin{array}{rcl} S & \to & Y_{aS}X_a \mid X_aA \mid Y_{bba}X_a \mid X_bX_b \mid a \\ A & \to & X_aA \mid a \\ X_a & \to & a \\ X_b & \to & b \\ Y_{aS} & \to & X_aS \\ Y_{bba} & \to & Y_{bb}X_a \\ Y_{bb} & \to & X_bX_b \end{array}$$

Note that other solutions are possible. For instance, we could also have

$$S \to Y_{aS}X_a \mid X_aE \mid Y_{bb}Y_{aa} \mid X_bX_b \mid a$$

with Y_{aa} and Y_{bb} defined.

(4 marks) 3. For a word $z \in \Sigma^*$, define the operation $\text{out}(z)$ as

$$\text{out}(z) = \{uw \ : \ \exists u, v, w \in \Sigma^* \text{ such that } z = uvw\}.$$

For example, $\text{out}(abacca)$ contains words such as $abcca$, aca, $abaa$ and $abacca$.

Let G be a fixed context free grammar in CNF. For an input word z of length n, give an $O(n^3)$ time algorithm to determine if there are any words in $out(z) \cap L(G)$.

Before starting the solution, we note that the brute force algorithm does not satisfy the conditions. Since there are $O(n^2)$ words of the form xz where $w = xyz$, performing the CYK algorithm on each of them cannot be done in $O(n^3)$ time. Through analysis, you can see that this algorithm will take $O(n^5)$ time.

To solve the problem, we need to reuse the CYK table. Let $G = (V, \Sigma, P, S)$ be our grammar. Since G is assumed to be in CNF, we can first construct the CYK table $T[i, j]$ for w in $O(n^3)$ time. Let $w = w_1 w_2 \cdots w_n$ where $w_i \in \Sigma$ are the letters of w. Recall that $T[i, j] = \{A \in V \ : \ A \Rightarrow^*$

$w_i w_{i+1} \cdots w_j\}$. In particular, $T[1,i]$ are the nonterminals that generate a prefix $w_1 \cdots w_i$ of w and $T[j,n]$ are those nonterminals that generate a suffix $w_j \cdots w_n$ of w.

Now, we need to determine whether or not $xz \in L(G)$ for every factorization xyz of the word w. To do this, we look at pairs $T[1,i]$ and $T[j,n]$ and do one step of CYK with those pairs. That is, we determine if there is a production $S \to AB$ in P with $A \in T[1,i]$ and $B \in T[j,n]$. If so, then S generates $w_1 \cdots w_i w_j \cdots w_n$.

Additionally, we need to consider whether $S \in T[j,n]$ or $S \in T[1,i]$ as this respresents $u = \varepsilon$ or $v = \varepsilon$ in the definition of $\text{out}(z)$.

Thus, the algorithm is

```
Build table T for w using CYK algorithm.
for i from 1 to n do
    for j from i+1 to n do
        if A ∈ T[1,i], B ∈ T[j,n] and S → AB is a production then
            return True
        end if
    end for
end for
for i from 1 to n do
    if S ∈ T[1,i] then
        return True
    end if
end for
for j from 1 to n do
    if S ∈ T[j,n] then
        return True
    end if
end for
return False
```

It remains to show that this runs in $O(n^3)$. The first line takes $O(n^3)$ time. The nested loops take $O(n^2)$ time. The if statement runs in time $O(|V|^2|P|)$, which does not depend on n. Finally, the last two loops take$O(n)$ time. Thus, the CYK algorithm takes time $O(n^3)$, the nested loop takes time $O(n^2)$, the loops after take $O(n)$ time and the entire algorithm runs in $O(n^3)$ time in the worst case.

(4 marks) 4. For a word $w \in \{0,1\}^*$ let $\text{bin}(w)$ be the value of w when interpreted as a binary number. For example, $\text{bin}(101) = 5$.

Constrct a PDA for the language

$$L = \{u\#v \ : \ u,v \in \{0,1\}^* \text{ and } \text{bin}(u) = \text{bin}(v^R) \text{ or } \text{bin}(v^R) = \text{bin}(u)+1\}$$

over the alphabet $\{0,1,\#\}$. For example, $10\#01, 11\#001 \in L$. In this question you may assume that u and v are well-formed binary numbers: that is, the most-significant digit is always 1, and u is written from most-significant digit to least significant digit, while v is written from least-significant digit to most-significant digit.

The language is the union of the two simpler languages

$$L_1 = \{u\#v : u,v \in \{0,1\}^* \text{ and } \mathrm{bin}(u) = \mathrm{bin}(v^R)\}$$
$$L_2 = \{u\#v : u,v \in \{0,1\}^* \text{ and } \mathrm{bin}(v^R) = \mathrm{bin}(u)+1\}$$

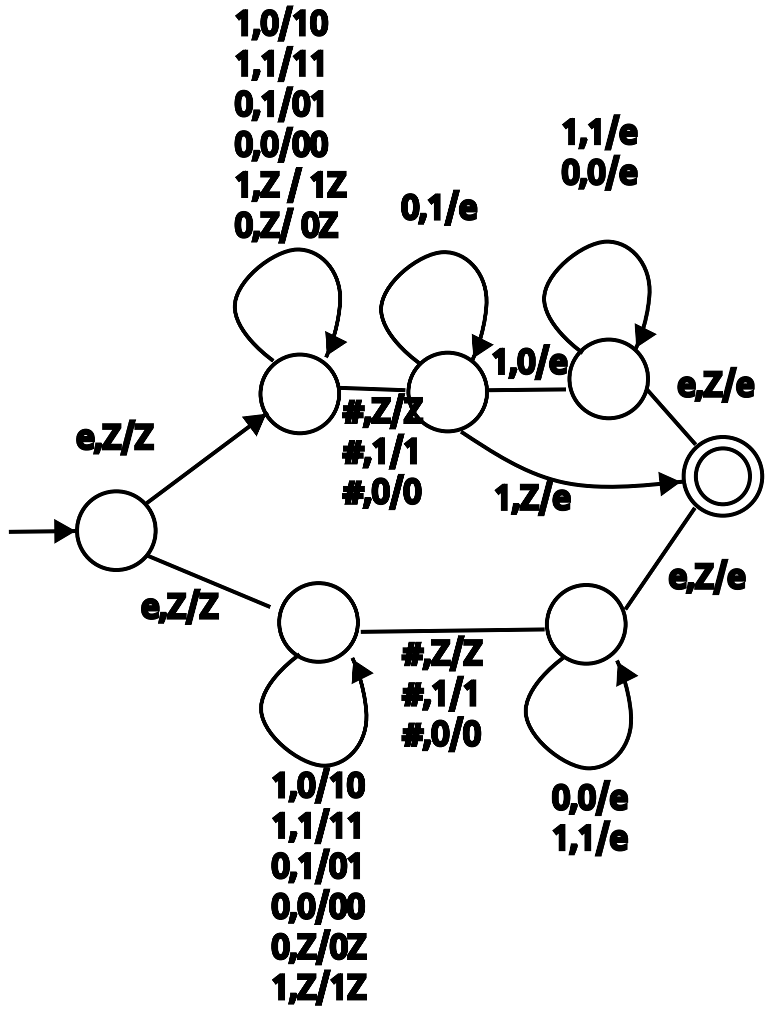
We show that each of these can be accepted by a PDA with empty stack. Then we can join them together using a new start state that makes an ε-transition to the start states of the individual PDAs.

For L_1, we simply match the sections before and after #. This can be done with the stack by pushing $0,1$ on the stack before we see # and then matching what appears after this symbol with what is on the stack.

For L_2, we do something similar, except we also execute addition at the same time. First we push all of what appears before the # onto the stack. Then we look at what appears in the input compared to what is on the stack.

- if $\mathrm{bin}(u)$ is even and $\mathrm{bin}(v^R)$ is odd, then u and v^R are exactly the same except that u ends with a 0 and v ends with a 1. (e.g., 10#11.) In this case, the PDA just ensures that the first character after # is a 1 and what is on the stack is a 0, and after that, everything else should match between the stack and the input.
- if $\mathrm{bin}(u)$ is odd and $\mathrm{bin}(v^R)$ is even, then u ends with some number of 1's and v ends with the same number of 0's. For instance, consider 1011#0011.
  - If u is just 1's, then v has the form $0^{|u|}1$.
  - Otherwise, there is a zero in u after the 1's. In v these are all 0's and then there is a 1 (e.g., $u = 1011$ and $v = 1100$). After this 'flipped' region, all the rest of u and v are the same.

The PDA is given below.


1,0/10
1,1/11
0,1/01
0,0/00
1,Z / 1Z
0,Z/ 0Z
0,1/e
1,1/e
0,0/e
1,0/e
e,Z/e
e,Z/Z
#,Z/Z
#,1/1
#,0/0
1,Z/e
e,Z/Z
e,Z/e
#,Z/Z
#,1/1
#,0/0
1,0/10
1,1/11
0,1/01
0,0/00
0,Z/0Z
1,Z/1Z
0,0/e
1,1/e