CS 181 HW7 2021 CS181

YIQIAO JIN

TOTAL POINTS

16 / 18

QUESTION 1

1 Ambiguity in CFG **6 / 8**

- **- 0 pts** Correct
- **- 2 pts** Wrong answer for 1a, e.g., did not provide evidence that shows the given grammar is ambiguous.
- **- 3 pts** Wrong answer for 1b, did not provide an explanation to show why the given CFG is ambiguous.
- **- 1 pts** Minor mistake for 1b, logic is incomplete
- ✓ **- 2 pts Mistake for 1c, provided CFG is not unambiguous**
- **- 1 pts** 1c, no explanation given
- **- 3 pts** Wrong answer for 1c.

QUESTION 2

2 TM Pseudo-code: Alg vs. Proc **4 / 4**

- ✓ **+ 4 pts Both parts Correct**
- **+ 1 pts** a. Answer ("C0 is incorrect") is correct
- **+ 1 pts** a. Brief explanation is adequate
- **+ 2 pts** b. is corrrect ("procedure, not algorithm")
- **+ 1 pts** b. is partially correct
- **+ 0 pts** b. is very incorrect or no answer
- **+ 0 pts** Both parts very incorrect
- **+ 0 pts** Both parts: No answer.

QUESTION 3

3 CFG for Language **6 / 6**

- ✓ **- 0 pts Correct or nearly correct**
- **- 2 pts** Partially correct
- **- 1 pts** Minor errors
- **- 4 pts** Attempted
- **- 1 pts** No explanations or not sufficent explanation

Homework 7

Name: Yiqiao Jin
UID: 305107551

1

a

The grammar is ambiguous. The string ac can be derived in two different ways using this grammar:

$S \to P \to ac$

$S \to P \to a\underline{S}c \to a\varepsilon c \to ac$

Similarly, the string bd can also be derived in two different ways:

$S \to B \to bd$

$S \to P \to b\underline{S}d \to b\varepsilon d \to bd$

So this grammar is ambiguous.

b

The ambiguity is partly due to the rule $S \to \varepsilon$, which means we can either generate a variable or not. Rule 1 provides a way to go from S to P, B, and Rule 2-3 provide ways to to go from P, B back to S. So the left-hand-side variable S is not always used as the start variable. This couples with the rule $P \to aSc|ac$ and becomes a repetition.

We can find two different reduction for the same reduction string $acbd$ due to this rule.

$\underline{ac}bd \mapsto \underline{a\varepsilon c}bd \mapsto P\underline{bd} \mapsto \underline{P}B \mapsto S\underline{B} \mapsto \underline{SS} \mapsto S$

$\underline{ac}bd \mapsto P\underline{bd} \mapsto \underline{P}B \mapsto S\underline{B} \mapsto \underline{SS} \mapsto S$

c

We can convert the grammar to Chomsky normal form by using the following conversion procedure

1. Make a new start variable

$$
\begin{aligned}
S_0 &\to S \\
S &\to SS|P|B|\varepsilon \\
P &\to aSc|ac \\
B &\to bSd|bd
\end{aligned}
$$

2. Remove ε rules, where LHS is NOT the start variable

$$
\begin{aligned}
S_0 &\to S|\varepsilon \\
S &\to SS|P|B \\
P &\to aSc|ac \\
B &\to bSd|bd
\end{aligned}
$$

3a. Remove unit rules $S \to P$, $S \to B$

$$
\begin{aligned}
S_0 &\to S|\varepsilon \\
S &\to SS|aSc|ac|bSd|bd \\
P &\to aSc|ac \\
B &\to bSd|bd
\end{aligned}
$$

Since P, B are no longer used, we can remove them.

$$
\begin{aligned}
S_0 &\to S|\varepsilon \\
S &\to SS|aSc|ac|bSd|bd
\end{aligned}
$$

3b. Remove $S_0 \to S$

$$
\begin{aligned}
S_0 &\to SS|aSc|ac|bSd|bd|\varepsilon \\
S &\to SS|aSc|ac|bSd|bd
\end{aligned}
$$

4. Replace rules with the form $U \to a$

$$
\begin{aligned}
S_0 &\to SS|U_aSU_c|U_aU_c|U_bSU_d|U_bU_d|\varepsilon \\
S &\to SS|U_aSU_c|U_aU_c|U_bSU_d|U_bU_d \\
U_a &\to a \\
U_b &\to b \\
U_c &\to c \\
U_d &\to d
\end{aligned}
$$

5. Convert 3-variable rules to 2-variable rules

$$
\begin{aligned}
S_0 &\to SS|V_aU_c|U_aU_c|V_bU_d|U_bU_d|\varepsilon \\
S &\to SS|V_aU_c|U_aU_c|V_bU_d|U_bU_d \\
V_a &\to U_aS \\
V_b &\to U_bS \\
U_a &\to a \\
U_b &\to b \\
U_c &\to c \\
U_d &\to d
\end{aligned}
$$

Since the grammar is in Chomsky normal form, the grammar is now unambiguous.

Another simpler but unambiguous grammar would be:

$$
\begin{aligned}
S &\to PS|BS|\varepsilon \\
P &\to aSc \\
B &\to bSd
\end{aligned}
$$

This grammar is unambiguous because it only allows generation of P, B with tail recursion. It also removes the cyclic generation given by Rule 2 ($P \to aSc|ac$) and Rule 3, as in **a**.

2

a

$C0$ does not correctly recognize the language. In this grammar, we are looking for directed cycles that contains $Node0$. However, $C0$ only looks at the first outgoing edge for each node, and the choice of the first edge can be arbitrary. This is equivalent to DFS along just a single path and the string is rejected if the path fails. Essentially we need some ways to keep track of the edges we have visited and see which edges we have not visited.

In the following example, as $C0$ starts at $Node0$, it may fail to choose edge 1 as its next edge, and directly go on edge 2 and reject the graph.

1 Ambiguity in CFG 6 / 8

- **0 pts** Correct
- **2 pts** Wrong answer for 1a, e.g., did not provide evidence that shows the given grammar is ambiguous.
- **3 pts** Wrong answer for 1b, did not provide an explanation to show why the given CFG is ambiguous.
- **1 pts** Minor mistake for 1b, logic is incomplete

✓ **- 2 pts** **Mistake for 1c, provided CFG is not unambiguous**

- **1 pts** 1c, no explanation given
- **3 pts** Wrong answer for 1c.

Since the grammar is in Chomsky normal form, the grammar is now unambiguous.

Another simpler but unambiguous grammar would be:

$$\begin{aligned} S &\to PS|BS|\varepsilon \\ P &\to aSc \\ B &\to bSd \end{aligned}$$

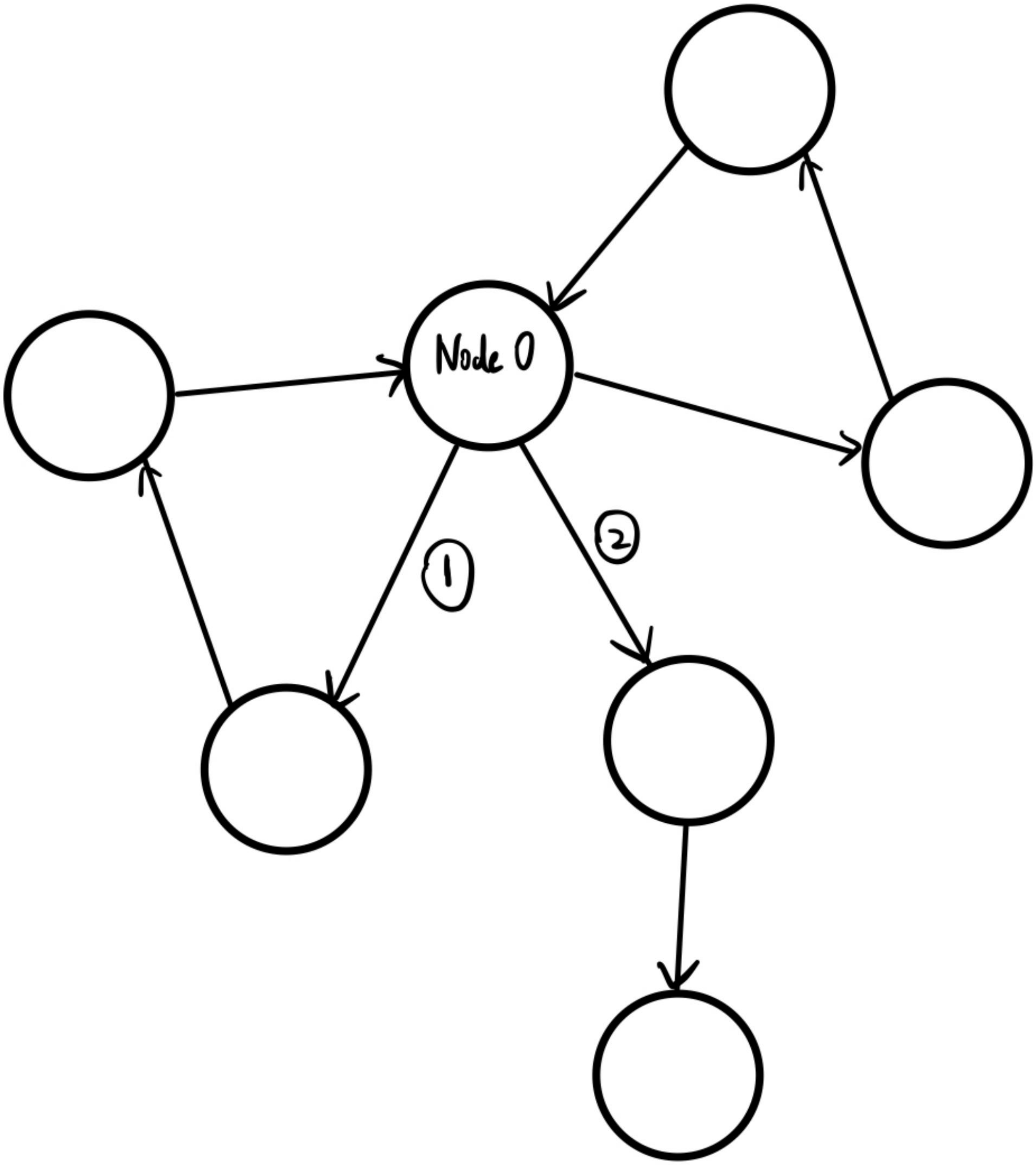
This grammar is unambiguous because it only allows generation of P, B with tail recursion. It also removes the cyclic generation given by Rule 2 ($P \to aSc|ac$) and Rule 3, as in **a**.

2

a

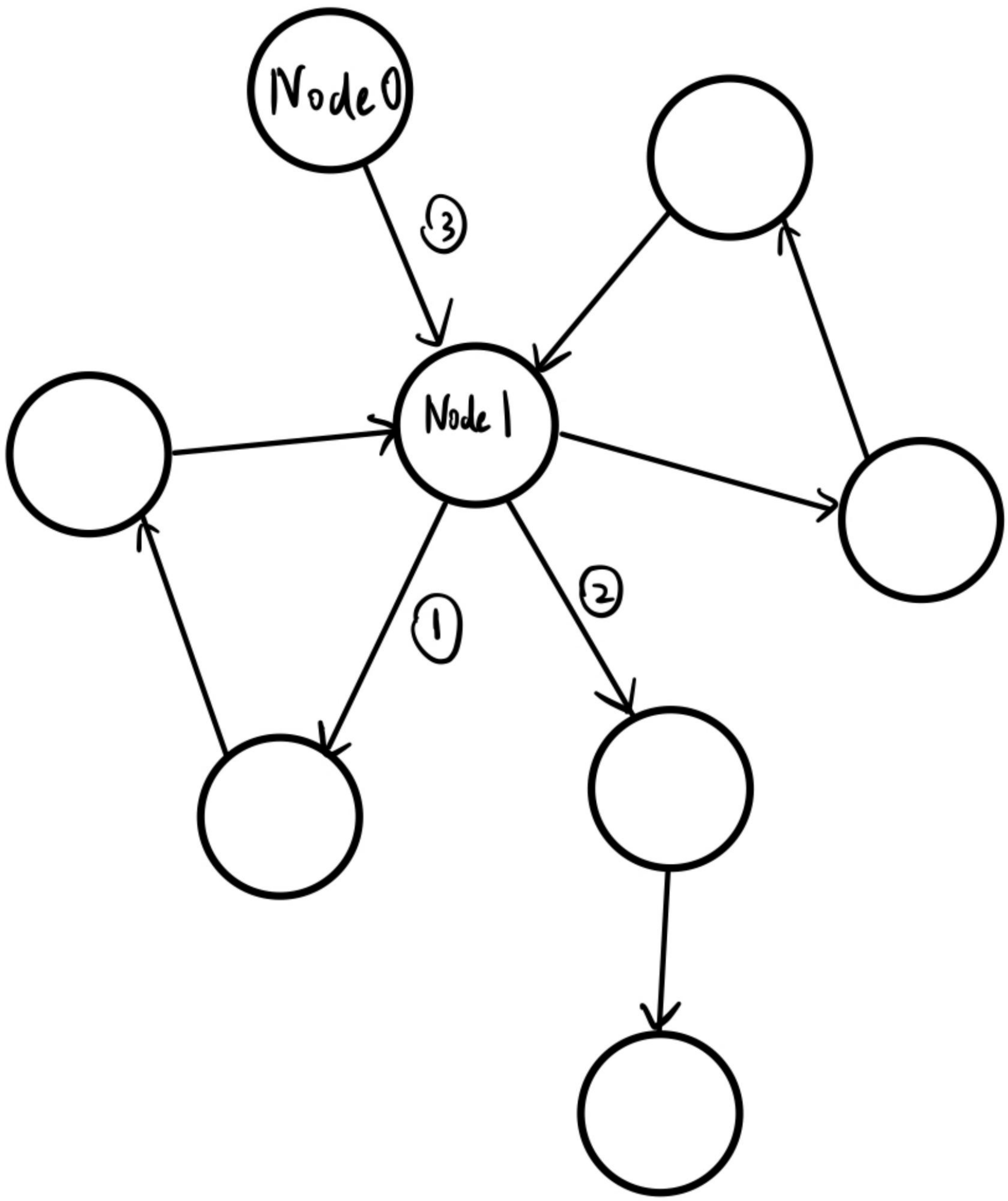
$C0$ does not correctly recognize the language. In this grammar, we are looking for directed cycles that contains $Node0$. However, $C0$ only looks at the first outgoing edge for each node, and the choice of the first edge can be arbitrary. This is equivalent to DFS along just a single path and the string is rejected if the path fails. Essentially we need some ways to keep track of the edges we have visited and see which edges we have not visited.

In the following example, as $C0$ starts at $Node0$, it may fail to choose edge 1 as its next edge, and directly go on edge 2 and reject the graph.



b

This is a **procedure** because it can fall into infinite loops. **Procedures** are Turing Machines that can fall into infinite loops, whereas **Algorithms** are **TM**'s which halt (and accept / reject) for all inputs. We have cycle in the graph G. Since $Node0$ is not necessarily the node that is involved in the loop, and $C0$ does NOT keep track of all the edges it has visited, it is possible for $C0$ to go on the same loop by visiting the same set of nodes several times.



In this example, $C0$ can start on $Node0$, go on edge 3 and land in some $Node1$ which is involved in a loop (for example, the loop that start with edge 1 in the figure), and go on this loop, eventually go back to $Node1$, and go on the same loop for the 2nd time and so on. $C0$ never halts because every node on this loop is NOT $Node0$ and has at least 1 outgoing edge, and $C0$ may choose the node involved in this loop as $NextNode$ every time.

3

2 TM Pseudo-code: Alg vs. Proc **4 / 4**

- ✓ **+ 4 pts** **Both parts Correct**
- **+ 1 pts** a. Answer ("C0 is incorrect") is correct
- **+ 1 pts** a. Brief explanation is adequate
- **+ 2 pts** b. is corrrect ("procedure, not algorithm")
- **+ 1 pts** b. is partially correct
- **+ 0 pts** b. is very incorrect or no answer
- **+ 0 pts** Both parts very incorrect
- **+ 0 pts** Both parts: No answer.

Below is a grammar for the language. Let $G = (V, \Sigma, R, S)$ be a CFG, where $\Sigma = \{a, b\}, V = \{S, A, B, C, D\}$

$$\begin{aligned} S &\to aAa|aBb|bCa|bDb \\ A &\to aAa|aBb|bAa|bBb \\ B &\to aBa|bBa|\varepsilon \\ C &\to aAa|aBb|bCa|bDb \\ D &\to aBa|bDa \end{aligned}$$

All of these variables generate 2 symbols at a time, so the length is guaranteed to be even. Also, the shortest string that can be generated by this grammar is ab.

S is the start variable. It generates strings that contain variables $\{A, B, C, D\}$. Each of these variables describes the specific requirements the string has satisfied.

A means that

- the first part of the string s already contains an a, BUT
- the second part does NOT have b's yet.

Thus we can continue generate either a or b in both first half and second half of the string.

B means that

- the first part of the string s already contains an a, AND
- the second part already contains a b

This means that conditions on both first part and second part of the string s have been temporarily satisfied. Now only more a's but no more b's are allowed in the second part. The first part can contain more a and b.

Note that B can yield ε because it satisfies restrictions on both 1st and 2nd part of the string, thus can terminate its generation.

C means that

- the first part of the string s does NOT contain an a, AND
- the second part does NOT contain b

This means that conditions on neither first part or second part of the string s has been temporarily satisfied. Thus we can generate either a or b in both first half and second half of the string.

D means that

- the first part of the string s does NOT contain an a, BUT

- the second part already contains a b

This means we need to generate a in the first part of the string. Also, only a's but not b's are allowed in the second part.

Example strings that can be accepted by L: $aaba, ab, baab$

Example strings that CANNOT be accepted by L: $b, \varepsilon, aabb, bb$

3 CFG for Language 6 / 6

- ✓ **- 0 pts** **Correct or nearly correct**
- **- 2 pts** Partially correct
- **- 1 pts** Minor errors
- **- 4 pts** Attempted
- **- 1 pts** No explanations or not sufficent explanation