1. (4 points) The grammar shown here on the left is for the language $L = \{a^n b^n c^n \mid n \in \mathbb{N}\}$. The grammar on the right is for the language $M = \{ww \mid w \in \{a, b\}^*\}$.

   $S \rightarrow XTZ$
   $T \rightarrow AbCT$
   $T \rightarrow \varepsilon$
   $bA \rightarrow Ab$
   $CA \rightarrow AC$
   $Cb \rightarrow bC$
   $X A \rightarrow a X$
   $C Z \rightarrow Z c$
   $X \rightarrow \varepsilon$
   $Z \rightarrow \varepsilon$

   $S \rightarrow HTE$
   $T \rightarrow a A T$
   $T \rightarrow b B T$
   $A a \rightarrow a A$
   $A b \rightarrow b A$
   $B a \rightarrow a B$
   $B b \rightarrow b B$
   $A E \rightarrow E a$
   $B E \rightarrow E b$
   $H a \rightarrow a H$
   $H b \rightarrow b H$
   $H E \rightarrow \varepsilon$
   $T \rightarrow \varepsilon$

   (a) Using the grammar on the left, give a derivation for the string $aabbcc$, which is in $L$.

   (b) Using the grammar on the right, give a derivation for the string $abaaba$, which is in $M$.

   Answer:

   (a) $S \Rightarrow XTZ$
       $\Rightarrow XAbCTZ$
       $\Rightarrow XAbC\!AbCTZ$
       $\Rightarrow XAbC\!Ac\!bCZ$
       $\Rightarrow XAbCA\!bCZ$
       $\Rightarrow XAA\!bC\!bCZ$
       $\Rightarrow XAAbbCCZ$
       $\Rightarrow aXabbCCZ$
       $\Rightarrow aabbCCZ$
       $\Rightarrow aabbcc$

   (b) $S \Rightarrow HTE$
       $\Rightarrow HaA\!T\!E$
       $\Rightarrow HaAbB\!T\!E$
       $\Rightarrow HaAbBa\!A\!T\!E$
       $\Rightarrow HaAbBaAE$
       $\Rightarrow HabA\!b\!a\!A\!E$
       $\Rightarrow HabA\!b\!AA\!E$
       $\Rightarrow HabAA\!E$
       $\Rightarrow HabA\!Eba$
       $\Rightarrow hab\!HEaba$
       $\Rightarrow abaaba$

2. (3 points) Create a general grammar for the language $\{a^n b^n a^n \mid n \in \mathbb{N}\}$, and indicate how the grammar works. You can show how the grammar works by giving comments on the rules. (Note that this language is similar to $\{a^n b^n c^n \mid n \in \mathbb{N}\}$. It is OK to have a grammar in which derivations can get “stuck.” My grammar has 10 rules.)

   Answer:
\[
S \rightarrow XTZ \\
T \rightarrow AaCT \\
T \rightarrow \varepsilon \\
aA \rightarrow Aa \\
CA \rightarrow AC \\
Ca \rightarrow aC \\
XA \rightarrow aX \\
CZ \rightarrow Za \\
X \rightarrow b \\
Z \rightarrow b
\]

Make strings of the form \( X(AaC)^nTZ \).

Get rid of the \( T \).

Allow \( A \)'s to move left and \( C \)'s to move right, making strings of the form \( XA^n a^n Ca^n Z \).

Convert \( A \)'s to \( a \)'s

Convert \( C \)'s to \( a \)'s, giving \( a^n Xa^n Za^n \).

Change \( X \) and \( Z \) to \( b \), leaving \( a^n ba^n ba^n \).

\[
S \rightarrow GHTE \\
T \rightarrow aCAT \\
T \rightarrow bDBT \\
T \rightarrow \varepsilon \\
Aa \rightarrow aA \\
Ab \rightarrow bA \\
AC \rightarrow CA \\
AD \rightarrow DA \\
Ba \rightarrow aB \\
Bb \rightarrow bB \\
BC \rightarrow CB \\
BD \rightarrow DB \\
Ca \rightarrow aC \\
Cb \rightarrow bC \\
Da \rightarrow aD \\
Db \rightarrow bD \\
AE \rightarrow Ea \\
BE \rightarrow Eb \\
Ha \rightarrow aH \\
Hb \rightarrow bH \\
HC \rightarrow CH \\
HD \rightarrow DH \\
HE \rightarrow F \\
CF \rightarrow Fa \\
DF \rightarrow Fb \\
Ga \rightarrow aG \\
Gb \rightarrow bG \\
GF \rightarrow \varepsilon
\]

The first four rules make strings matching the regular expression \( GH(aCA)bDB)^*E \).

These 8 rules let \( A \) and \( B \) move past the other symbols to get to the right end of the string, but keeping the \( A \)'s and \( B \)'s in the same relative order.

These 4 rules let \( C \) and \( D \) move past \( a \)'s and \( b \)'s, but not past \( A \)'s and \( B \)'s, to get to get the middle of the string, in the same relative order.

\( E \) converts \( A \) and \( B \) to \( a \) and \( b \).

\( H \) moves through \( a, b, C, D \), reaches the \( E \), and \( HE \) converts to \( F \). This can only happen after \( E \) has finished converting the \( A \)'s and \( B \)'s.

Now \( F \) converts \( C \) and \( D \) to \( a \) and \( b \) giving strings such as \( GaababFaababaabab \).

Finally, \( G \) moves past \( a \) and \( b \) and meets the \( F \), and \( GF \) disappears. This can only happen after all \( C \)'s and \( D \)'s are gone.

3. (4 points) Create a general grammar for the language \( \{www \mid w \in \{a, b\}^*\} \), and indicate how the grammar works. You can show how the grammar works by giving comments on the rules. (Note that this language is similar to \( \{ww \mid w \in \{a, b\}^*\} \). Idea: Using rules similar to the above grammar for \( \{ww \mid w \in \{a, b\}^*\} \), make strings like \( abaabCDCCDHEabaab \), then instead of disappearing, the \( HE \) changes to a symbol that can convert the \( C \)'s and \( D \)'s to \( a \)'s and \( b \)'s. It is OK to have a grammar in which derivations can get “stuck,” but it’s not too hard to extend this idea to one that can’t get stuck. My grammar has 28 rules.)

Answer:

\[
S \rightarrow GHTE \\
T \rightarrow aCAT \\
T \rightarrow bDBT \\
T \rightarrow \varepsilon \\
Aa \rightarrow aA \\
Ab \rightarrow bA \\
AC \rightarrow CA \\
AD \rightarrow DA \\
Ba \rightarrow aB \\
Bb \rightarrow bB \\
BC \rightarrow CB \\
BD \rightarrow DB \\
Ca \rightarrow aC \\
Cb \rightarrow bC \\
Da \rightarrow aD \\
Db \rightarrow bD \\
AE \rightarrow Ea \\
BE \rightarrow Eb \\
Ha \rightarrow aH \\
Hb \rightarrow bH \\
HC \rightarrow CH \\
HD \rightarrow DH \\
HE \rightarrow F \\
CF \rightarrow Fa \\
DF \rightarrow Fb \\
Ga \rightarrow aG \\
Gb \rightarrow bG \\
GF \rightarrow \varepsilon
\]
4. (6 points) Consider the language \( L = \{a^{2^n} \mid n \in \mathbb{N}\} \).

(a) Create a general grammar for the language \( \{a^{2^n} \mid n \in \mathbb{N}\} \). The grammar contains all strings of a’s whose length is a power of 2. (As a hint, note that if you start with one a and double it \( n \) times, then there will be \( 2^n \) a’s. For full credit, write a grammar for which derivations cannot get “stuck.” This can be done with a grammar that has seven production rules.)

(b) Explain in words why your grammar works. How can it generate every string in \( L \)? Why can’t it generate any other strings?

(c) Using your grammar, write derivations for the strings \( a \) and \( aaaaaaaa \). Note that \( a = a^{2^0} \) and \( aaaaaaaa = q^{2^3} \), so both of these strings are in \( L \).

Answer:

\[
\begin{align*}
\text{a)} & & S \rightarrow HTE \\
& & T \rightarrow DT \\
& & T \rightarrow a \\
& & Da \rightarrow aaD \\
& & DE \rightarrow E \\
& & Ha \rightarrow aH \\
& & HE \rightarrow \varepsilon \\
\text{c)} & & S \Rightarrow HTE \\
& & S \Rightarrow HDTE \\
& & S \Rightarrow HDaDaE \\
& & S \Rightarrow HDaaaE \\
& & S \Rightarrow HaaaaaDaE \\
& & S \Rightarrow HaaaaaaaaDaE \\
& & S \Rightarrow HaaaaaaaaaaDaE \\
& & S \Rightarrow HaaaaaaaaaaaDaE \\
\end{align*}
\]

(b) To make the string \( a^{2^n} \), the first rule is applied to give \( HTE \). Then the second rule is applied \( n \) times, giving \( HD^nTE \). The third rule replaces the \( T \) with an \( a \), giving \( HD^nDaE \). Any string produced by the first four rules can only be of this form, for some \( n \in \mathbb{N} \).

The fourth rule lets each \( D \) move through the string of \( a \)'s. As it does this, the number of \( a \)'s is doubled. After passing through all the \( a \)'s, the \( D \) hits the \( E \), and the fifth rule then deletes the \( D \). This rule can only be applied after the \( D \) has passed every \( a \), so that the number of \( a \)'s must be doubled by each \( D \).

The sixth rule lets the \( H \) move past all the \( a \)'s and hit the \( E \). Since \( H \) can’t pass a \( D \), this can only happen after all the \( D \)'s have been deleted, and at that point, there are \( 2^n \) \( a \)'s. When \( H \) reaches \( E \), the last rule deletes them both, leaving only the \( 2^n \) \( a \)'s.
5. (3 points) This is a small exercise to help you get used to working with the Turing machine simulator. In class, we looked at a simple example of a Turing machine that moves to the right searching for two $$'s in a row. When (and if) it encounters them, it halts, and the machine is left on the second $$.

Create such a Turing machine in the simulator. You should assume that the tape contains only a’s, b’s, blanks, and $$’s.

This can be done using two states (in addition to the halt state), if you use the “stay” option (S) as a direction of motion at the end. Without that option, it requires 3 states to put the machine in the proper position at the end. If you use the “other” and “same” options for “Old Symbol” and “New Symbol” in some of your rules, you can do this with just four or five lines in the rule table.

Answer:

```
{
  "name": "Search Right for $$", 
  "max_state": 25, 
  "symbols": "xyzabc01$$", 
  "tape": "ab $aa $ b a $$a b$a", 
  "position": 0, 
  "rules": [
    [ 0, "$", "$", 1, "R" ], 
    [ 0, "*", "+", 0, "R" ], 
    [ 1, "$", "$", "h", "S" ], 
    [ 1, "*", "*", 0, "R" ]
  ]
}
```
6. (4 points) Create a Turing machine that checks whether the number of a’s in a string of a’s and b’s is a multiple of 3. The input is a string of a’s and b’s with the machine positioned on the right end of the string. The output of the computation should be 1 if the number of a’s is a multiple of 3, and should be 0 if the number is not a multiple of 3. Note that the b’s don’t contribute anything to the answer, but they need to be erased just like the a’s need to be erased. (That is, the only thing left on the tape should be a 0 or 1, and the machine should be positioned on that 0 or 1.)

Answer:

```json
{
    "name": "Does 3 divide the number of a’s?",
    "max_state": 25,
    "symbols": "xyzabc01$@",
    "tape": "aababbbababbaba",
    "position": 14,
    "rules": [
        [ 0, "#", "1", 4, "L" ],
        [ 0, "a", "#", 1, "L" ],
        [ 0, "b", "#", 0, "L" ],
        [ 1, "#", "0", 4, "L" ],
        [ 1, "a", "#", 2, "L" ],
        [ 1, "b", "#", 1, "L" ],
        [ 2, "#", "0", 4, "L" ],
        [ 2, "a", "#", 0, "L" ],
        [ 2, "b", "#", 2, "L" ],
        [ 4, "#", "#", "h", "R" ]
    ]
}
```
7. (6 points) In class, we looked at a “binary-to-unary” converter. The input is a string of 0’s and 1’s, considered to be a binary number. When started on the rightmost digit of a binary number, the output of the machine is a string of a’s, where the number of a’s is equal to the original binary number.

Write a “unary-to-binary” converter: When the machine is started on the right end of a string of a’s as input, the output should be a binary number equal to the original number of a’s. That is, the binary number should be the only thing on the tape, and the machine should be positioned on the rightmost digit of the binary number. Your machine does not have to work for empty input, but if you want to output the correct answer, 0, for empty input you can do it.

As an idea for the program, create the binary number to the left of the string of a’s. Erase an a from the right end of the string, move to the left end and increment the binary number, then move back to the right end of the string of a’s.

Answer:

This version works for any string of 1 or more a’s:

```json
{
    "name": "Unary-to-Binary Mark 1",
    "max_state": 25,
    "symbols": "xyzabc01$@",
    "tape": "aaaaaaa",
    "position": 6,
    "rules": [
        [ 0, "#", "#", "h", "L" ],
        [ 0, "a", "#", 1, "L" ],
        [ 1, "#", "#", 2, "L" ],
        [ 1, "a", "a", 1, "L" ],
        [ 2, "#", "1", 3, "R" ],
        [ 2, "0", "1", 3, "R" ],
        [ 2, "1", "0", 2, "L" ],
        [ 3, "#", "#", 4, "R" ],
        [ 3, "0", "0", 3, "R" ],
        [ 3, "1", "1", 3, "R" ],
        [ 4, "#", "#", 0, "L" ],
        [ 4, "a", "a", 4, "R" ]
    ]
}
```

The version on the next page also works when the input is the empty string. It uses states 1 through 5 to do the same computation that is done by the previous machine in states 0 through 4. State 0 in the new machine is used to test whether the input is empty. If so, it immediately halts, without moving, with the correct output, 0. If the machine sees an a in state 0, it just transitions to state 1, without moving, and proceeds from there exactly like the first machine.
"name": "Unary-to-binary Mark 2",
"max_state": 25,
"symbols": "xyzabc01$@",
"tape": "aaaaaaaaaa",
"position": 8,
"rules": [
  [ 0, "#", "0", "h", "S" ],
  [ 0, "a", "a", 1, "S" ],
  [ 1, "#", "#", "h", "L" ],
  [ 1, "a", "#", 2, "L" ],
  [ 2, "#", "#", 3, "L" ],
  [ 2, "a", "a", 2, "L" ],
  [ 3, "#", "1", 4, "R" ],
  [ 3, "0", "1", 4, "R" ],
  [ 3, "1", "0", 3, "L" ],
  [ 4, "#", "#", 5, "R" ],
  [ 4, "0", "0", 4, "R" ],
  [ 4, "1", "1", 4, "R" ],
  [ 5, "#", "#", 1, "L" ],
  [ 5, "a", "a", 5, "R" ]
]