Problem 1

Consider the following context-free grammar $G$:

\[
E \rightarrow E + T \mid T \\
T \rightarrow x \mid (E)
\]

It generates expressions like $x$, $x + (x + x) + (x + x)$, and so on.

(a) Write all items in this grammar and construct an NFA for the valid item updates.

(b) Convert the NFA to a DFA. Which of the states are shift states and which are reduce states? Are there any conflicts?

(c) Using the DFA, show an execution of the LR(0) parsing algorithm on the input

\[x + (x + x) + (x + x)\].

Show the state of the stack, input, and DFA throughout the execution.

(d) Now consider the following extended context-free grammar $G'$:

\[
E \rightarrow E + T \mid T \\
T \rightarrow T * F \mid F \\
F \rightarrow x \mid (E)
\]

Show that $G'$ is not an LR(0) grammar.

(e) (Extra credit) Give an LR(1) DFA for $G'$. Show an execution of the LR(1) parsing algorithm on input $(x + x) * (x + x * x)$.

Solution

(a) These are the possible items:

\[
E \rightarrow \bullet E + T \quad E \rightarrow \bullet T \quad T \rightarrow \bullet x \quad T \rightarrow \bullet (E) \\
E \rightarrow E \bullet + T \quad E \rightarrow T \bullet \quad T \rightarrow x \bullet \quad T \rightarrow (\bullet E) \\
E \rightarrow E + \bullet T \quad T \rightarrow (E \bullet) \\
E \rightarrow E + T \bullet \quad T \rightarrow (E) \bullet
\]
The following is a diagram of the NFA. Edges with no label represent $\epsilon$-transitions.

(b) The following is a diagram of the DFA. State 1 is the start state. States 1, 2, 3, 5, and 7 are shift states; states 4, 6, 8, and 9 are reduce states. Notice there are no shift/reduce conflicts.

(c) Parse Table:

<table>
<thead>
<tr>
<th>Action</th>
<th>Stack</th>
<th>DFA State</th>
<th>Input</th>
</tr>
</thead>
<tbody>
<tr>
<td>Shift</td>
<td>$x$</td>
<td>$\epsilon$</td>
<td>1</td>
</tr>
<tr>
<td>Reduce</td>
<td>$T \rightarrow x$</td>
<td>$T$</td>
<td>16</td>
</tr>
<tr>
<td>Reduce</td>
<td>$E \rightarrow T$</td>
<td>$E$</td>
<td>18</td>
</tr>
<tr>
<td>Shift</td>
<td>$+$</td>
<td>$E+$</td>
<td>157</td>
</tr>
<tr>
<td>Shift</td>
<td>$(E)$</td>
<td>$E + (E)$</td>
<td>1572</td>
</tr>
<tr>
<td>Shift</td>
<td>$x$</td>
<td>$E + (x)$</td>
<td>15726</td>
</tr>
<tr>
<td>Reduce</td>
<td>$T \rightarrow x$</td>
<td>$E + (T)$</td>
<td>15728</td>
</tr>
<tr>
<td>Reduce</td>
<td>$E \rightarrow T$</td>
<td>$E + (E)$</td>
<td>15723</td>
</tr>
<tr>
<td>Shift</td>
<td>$+$</td>
<td>$E + (E + x)$</td>
<td>157237</td>
</tr>
<tr>
<td>Shift</td>
<td>$x$</td>
<td>$E + (E + x)$</td>
<td>1572376</td>
</tr>
</tbody>
</table>
(d) In the DFA for the parsing of the language, the start state contains transitions $T \rightarrow \epsilon T \epsilon F$ and $E \rightarrow \epsilon T$. By consuming $T$, we reach a state that contains transitions $E \rightarrow E + T$ and $T \rightarrow T \epsilon \epsilon F$. This is a shift/reduce conflict, so the grammar is not LR(0).

(e) This is what the DFA looks like, with the purely reduce states omitted:

```
Accept!
```
The items corresponding to each state are as follows:

1. \( E \rightarrow \bullet E + T, \varepsilon + \)
   - \( F \rightarrow \bullet x, \varepsilon + \)
   - \( E \rightarrow \bullet E + T, \varepsilon + \)
2. \( E \rightarrow \bullet + T, \varepsilon + \)
3. \( E \rightarrow \bullet E + T, \varepsilon + \)
   - \( F \rightarrow \bullet x, \varepsilon + \)
   - \( F \rightarrow \bullet (E), \varepsilon + \)
4. \( E \rightarrow \bullet T, \varepsilon + \)
   - \( T \rightarrow \bullet T, \varepsilon + \)
5. \( T \rightarrow \bullet F, \varepsilon + \)
   - \( F \rightarrow \bullet + + \)
   - \( T \rightarrow \bullet F, \varepsilon + \)
6. \( F \rightarrow \bullet (E), \varepsilon + \)
7. \( E \rightarrow \bullet E + T, \varepsilon + \)
8. \( E \rightarrow \bullet E + T, \varepsilon + \)
9. \( T \rightarrow \bullet F, \varepsilon + \)
10. \( F \rightarrow \bullet x, \varepsilon + \)
11. \( F \rightarrow \bullet (E), \varepsilon + \)
12. \( E \rightarrow \bullet T, \varepsilon + \)

**Problem 2**

In this problem, you will design Turing machines for the following three languages:

(a) \( L_1 = \{a^n\#a^n\#a^n : n \geq 0\} \), \( \Sigma = \{a, \#\} \). Give both a high-level description and a state diagram of your Turing Machine.

(b) \( L_2 = \{a^n\#b^2n : n \geq 0\} \), \( \Sigma = \{a, b\} \). Give only a high-level description of the Turing Machine. A state diagram is not necessary.

**Solution**

(a) The Turing Machine for \( L_1 \) works in stages, where in every stage the first remaining \( a \) in every block is crossed off (i.e., replaced by an \( x \)). If at any stage in the process, there are no \( a \)s left to cross off in some block, the TM rejects. If after all the \( a \)s in the first block have been crossed off there are still some uncrossed \( a \)s in other blocks, the TM rejects also. Here is a high-level description of the Turing Machine:
$M_1$: “On input $w$:

1. Scan the input from left to right to cross off the leftmost $a$ before the first $#$ then cross off the first $a$ we see after every $#$.
2. When there are no $a$s remaining in the first block: If there are no $a$s left on the tape, accept, otherwise reject.
3. Return the head to the left-hand end of the tape and repeat stage 1.”

Here is a state diagram for $M_1$. After all the $a$s in the first block have been crossed off, the TM goes into state $q_8$, which verifies that there are no $a$s remaining anywhere on the tape.

The Turing Machine for $L_1$ works in stages, where in every stage the first remaining $a$ in every block is crossed off (i.e., replaced by an $x$). If at any stage in the process, there are no $a$s left to cross off in some block, the TM rejects. If after all the $a$s in the first block have been crossed off there are still some uncrossed $a$s in other blocks, the TM rejects also. Here is a high-level description of the Turing Machine:

(b) The Turing Machine for $L_2$ performs the following in rounds: Cross off an $a$ and every other (uncrossed) $b$, checking that an even number of $b$s is crossed off in each round. This ensures we eliminate one $a$ and half the remaining $b$s in every round. If the input is of the form $a^n # b^{2^n}$, then after $n$ rounds we will have crossed every $a$ and all but exactly one $b$ (recall that $2^0 = 1$).
$M_2$: “On input $w$:

1. Scan the input from left to right to determine whether it is of the form $a^n \# b^m$ where $n \geq 0, m \geq 0$ and reject if it isn’t.

2. Repeat the following until all $a$s have been crossed off: Cross off the first uncrossed $a$ and half the remaining $b$s. If there are no $b$s left to cross or there is an odd number of them, reject.

3. If all $a$s and all all but exactly one $b$s are crossed off, accept.”

For completeness, here is a state diagram of $M_2$. In this implementation, steps 1 and 2 are combined.

---

**Problem 3**

A *queue automaton* is like a push-down automaton except that the stack is replaced by a queue. A queue is a tape allowing symbols to be read only at the left end and written only at the right end. Each read operator (called a *pop*) reads and removes a symbol from the left end of the tape and each write operation (called a *push*) writes a symbol at the right end. For example, if the state of the tape is $abcaab$, the operation *pop a* yields $bcbaab$. Now *push c* yields $bcbaabc$. Initially, the queue contains the input followed by the blank symbol □. The automaton accepts (rejects) by going into a special state $q_{accept}$ ($q_{reject}$). The transitions in a queue automaton are deterministic, but the automaton has the option of not popping or not pushing any symbol (that is, *pop ε* and *push ε* are allowed).

You will argue that a queue automaton is equivalent to a Turing Machine: Every queue automaton can be simulated on a Turing Machine, and vice versa.

(a) Write a formal definition of a queue automaton.

(b) Show how to simulate a queue automaton on a Turing Machine. For this, you need to specify

- how the tape of the Turing Machine will be used to represent the queue automaton;
• how the Turing Machine tape should be set up initially;
• what the Turing Machine should do when the automaton performs a push or a pop;
• what the Turing Machine should do when the queue automaton accepts / rejects.

(c) Show how to simulate a Turing Machine on a queue automaton.

Solution

(a) A Queue automaton is a 6-tuple \((Q, \Sigma, \Gamma, \delta, q_0, F)\), where \(Q, \Sigma, \Gamma\) and \(F\) are all finite sets, and

- \(Q\) is the set of states,
- \(\Sigma\) is the input alphabet,
- \(\Gamma\) is the queue alphabet,
- \(\delta: Q \times (\Sigma \cup \{\epsilon\}) \times (\Gamma \cup \{\epsilon\}) \rightarrow Q \times (\Gamma \cup \{\epsilon\})\) is the transition function,
- \(q_0 \in Q\) is the initial state,
- \(F \subseteq Q\) is the accept state.

(b) We show how to simulate a Queue Automaton with a Turing Machine. Write \(*\) and \(\$\) on the tape to denote the head and tail of the queue.

(i) Pop: Find \(*\) and replace it by \(B\), move right, read it and replace it by \(*\).

(ii) Push: Find \(\$\) and move left, read it, move right, replace the \(\$\) by the last memorized symbol, move right, write \(\$\).

(iii) Accept and Reject: The same as on the Queue Automaton.

(c) We show how to simulate a Turing Machine with a Queue Automaton. First push \(w\) into the queue.

(i) Move right: Pop the head of the queue and then push it to the tail of queue.

\[
abc \rightarrow bca
\]

(ii) Move left: Push \(\$\) to the tail of the queue as mark.

\[
abc \rightarrow abc\$
\]

We pop the head of the queue and the next one, if the second one is not \(\$\), then push the prior one to the queue and go on checking the next one. Otherwise, push \(\$\) and then the prior one, end.

\[
abc\$ \rightarrow c\$a\ (pop \ a \ and \ b, \ because \ b \ is \ not \ \$), \ push \ a)
\rightarrow $ab\ (pop \ c, \ since \ c \neq \$, \ push \ b)
\rightarrow ab\$c\ (pop \ $, \ then \ push \ $ \ and \ c)
\]

Next, we pop the head and if it is not \($\,\), push it, repeatedly. If it is $, end.

\[
ab\$c \rightarrow b\$ca\ (pop \ a \ and \ push \ a)
\rightarrow $ab\ (pop \ b \ and \ push \ b)
\rightarrow cab\ (pop \ $, \ end)
\]
(iii) **Replacement:** Pop the head, push the symbol that is used to replace, then move left.

\[ abc \rightarrow bc* \text{ (pop } a \text{ and push } *) \rightarrow *bc \text{ (move left)} \]

(iv) **Accept and Reject:** The same as in Turing Machine.

**Problem 4**

The Church-Turing Thesis claims that Turing Machines are a universal model of computation: Any computation that can be performed on any computer we will ever build can also be done on a Turing Machine. Here are some possible objections to the Church-Turing Thesis. For each of these objections, say if you think it is reasonable or not, and explain why. (You won’t be graded based on whether your answer is “right” or “wrong”, but based on how well you explain your answer.)

(a) Suppose I want to know what is the smallest country in the world. In real life, I would use google, type in "smallest country", and I find out the answer after a few clicks. But I cannot do this on a Turing Machine. How do I even connect a Turing Machine to the internet? Since there are computations we can do in real life but not on a Turing Machine, the Church-Turing thesis is false.

(b) Look at this computer program:

```java
int F(int n) {
    if (n == 1) return 1;
    else return F(n - 1) + F(n - 2);
}
```

This program uses **recursion:** A procedure makes a subroutine call to itself. But Turing Machines do not support recursion. Therefore Turing Machines are not as powerful as ordinary programming languages, and the Church-Turing thesis is false.

(c) Humans can also be modeled as computers: We take inputs from the environment (by seeing, hearing, touching) and produce outputs (via speaking and gestures). If the Church-Turing thesis is true, then any task that humans can do can also be done on a Turing Machine, and so on any machine. But there are tasks that humans are better at than machines: Learning foreign languages, identifying objects in images, winning basketball games, and so on. Therefore the Church-Turing Thesis cannot be true.

**Solution**

(a) We can think of the internet as an external memory resource that is attached to our computer. Its contents can be represented on the tape of the Turing Machine. When we look for information on the internet, we can think of the process as running a Turing Machine that gets as its input, in addition to the search pattern, the whole contents of the internet. So this scenario does not really invalidate the Church-Turing thesis.
(b) Recursion is a common feature in programming languages. However, not all of them have it; for example, assembly languages, like the command set of the Random Access Machine, typically do not have recursion. When a program in a language like Java is compiled, the recursion calls are eliminated. Since programming languages that support recursion can be simulated (i.e., compiled) on a Random Access Machine, which is in turn equivalent to a Turing Machine, the Church-Turing thesis is not invalidated by this example.

(c) Whether humans are machines or not is a matter of debate among philosophers and experts in artificial intelligence. A verdict has not been reached yet. If we believe that humans are machines, then we expect to model their behavior by simple steps like those taken by a Turing Machine. If we believe humans are not machines, then the Church-Turing thesis does not say anything about them. Until we gain a better understanding, this question is out of scope for the Church-Turing thesis.