Student Information

Full Name : Beyazıt Yalçınkaya
Id Number : 2172138

Answer 1

a.

A pushdown automaton for bottom-up parser of $G = (V, \Sigma, R, S)$ is defined as follows. $M = (K, \Sigma, \Gamma, \Delta, s, F)$, where

- $K = \{p, q\}$,
- $\Sigma = \{a, b, c\}$,
- $\Gamma = \{a, b, c, S, X\}$,
- $\Delta = \{((p, a, e), (p, a)), ((p, b, e), (p, b)), ((p, c, e), (p, c)), ((p, e, XaXSa), (p, S)),$
  $((p, e, XbXSb), (p, S)), ((p, e, c), (p, S)), ((p, e, Xa), (p, X)), ((p, e, Xb), (p, X)),$
  $((p, e, e), (p, X)), ((p, e, S), (q, e))\}$,
- $s = p$, and
- $F = \{q\}$

b.

$$
\begin{aligned}
(p, abbcbabbaa, e) &\vdash_M (p, bbcbabbaa, a) \\
&\vdash_M (p, bcbabbaa, ba) \\
&\vdash_M (p, cbabbaa, bba) \\
&\vdash_M (p, babbaa, cbba) \\
&\vdash_M (p, babbaa, Sbba) \\
&\vdash_M (p, babbaa, XSbba) \\
&\vdash_M (p, abbaa, bXSbba) \\
&\vdash_M (p, abbaa, XbXSbba) \\
&\vdash_M (p, abbaa, Sba) \\
&\vdash_M (p, bbaa, aSba) \\
&\vdash_M (p, bbaa, XaSba) \\
&\vdash_M (p, bbaa, XSba) \\
&\vdash_M (p, baa, bXSba) \\
&\vdash_M (p, baa, XbXSba) \\
&\vdash_M (p, baa, Sa) \\
&\vdash_M (p, aa, bSa) \\
&\vdash_M (p, a, abSa) \\
&\vdash_M (p, a, XabSa) \\
&\vdash_M (p, a, XbSa) \\
&\vdash_M (p, a, XSa) \\
&\vdash_M (p, e, aXSa) \\
&\vdash_M (p, e, XaXSa) \\
&\vdash_M (p, e, S) \\
&\vdash_M (q, e, e)
\end{aligned}
$$

Answer 2

a.

Formal description for the Turing machine that computes the function f is given as follows.
$M = (K, \Sigma, \delta, s, H)$ where,

- $K = \{q_0, q_{Even}, q_{Odd}, q_{MarkGoLeft}, q_{MarkGoRight}, q_{Begin}, q_{End}, q_{PutOnes}, q_{DelteMarks}, h\}$,
- $\Sigma = \{\sqcup, \triangleright, 1, \#\}$,
- δ is given in Table 1 (Note that, due to the constraint on the input strings and implementation of the machine some of the transitions can never be enabled, for the sake of simplicity those transitions are omitted in the table.),

- $s = q_0$, and
- $H = \{h\}$.

Table 1: δ function

q	σ	$\delta(q, \sigma)$
q_0	$\sqcup$	$(q_{Even}, \sqcup, \rightarrow)$
q_{Even}	$\sqcup$	$(q_{MarkGoLeft}, \sqcup, \leftarrow)$
q_{Even}	1	$(q_{Odd}, 1, \rightarrow)$
q_{Odd}	$\sqcup$	$(h, 1, \rightarrow)$
q_{Odd}	1	$(q_{Even}, 1, \rightarrow)$
$q_{MarkGoLeft}$	1	$(q_{Begin}, \#, \leftarrow)$
$q_{MarkGoLeft}$	#	$(q_{PutOnes}, 1, \leftarrow)$
$q_{MarkGoRight}$	1	$(q_{End}, \#, \rightarrow)$
q_{Begin}	$\sqcup$	$(q_{MarkGoRight}, \sqcup, \rightarrow)$
q_{Begin}	1	$(q_{Begin}, 1, \leftarrow)$
q_{Begin}	#	$(q_{MarkGoRight}, \#, \rightarrow)$
q_{End}	1	$(q_{End}, 1, \rightarrow)$
q_{End}	#	$(q_{MarkGoLeft}, \#, \leftarrow)$
$q_{PutOnes}$	$\sqcup$	$(q_{DeleteMarks}, \sqcup, \rightarrow)$
$q_{PutOnes}$	#	$(q_{PutOnes}, 1, \leftarrow)$
$q_{DeleteMarks}$	$\sqcup$	$(h, \sqcup, \rightarrow)$
$q_{DeleteMarks}$	1	$(q_{DeleteMarks}, 1, \rightarrow)$
$q_{DeleteMarks}$	#	$(q_{DeleteMarks}, \sqcup, \rightarrow)$

Answer 3

The set of languages that is decided by Move-restricted Turing Machines is the set of regular languages. Even though Move-restricted Turing Machines are able to write on the input tape since they cannot move their head left, they do not have any memory, so they can only read the input once and make computations accordingly. This makes them equivalent to finite state automata and it is known that the set of languages accepted by finite state automata is the set of regular languages. Hence, the set of languages decided by Move-restricted Turing Machines is the set of regular languages.

Answer 4

a.

A Queue-based deterministic Turing Machine is a tuple $(K, \Sigma, \delta, s, F)$, where

- K is a finite set of states,
- Σ is an alphabet (the input symbols),
- $s \in K$ is the initial state,
- $F \subseteq K$ is the set of final states, and
- $\delta : K \times \Sigma \to K \times \Sigma^*$ is the transition function.

b.

A configuration of a Queue-based deterministic Turing Machine $M = (K, \Sigma, \delta, s, H)$ is a member of $K \times \Sigma^*$: The first component is the state of the machine, the second component is the contents of the queue, read from front to rear. Notice that for a Queue-based deterministic Turing Machine, it is assumed that the queue initially contains the input string so we use same alphabet for input and queue.

c.

If $(p, \alpha\beta)$ and $(q, \beta\gamma)$ are configurations of M, we say that $(p, \alpha\beta)$ yields in one step $(q, \beta\gamma)$ $\big($notation: $(p, \alpha\beta) \vdash_M (q, \beta\gamma)\big)$ if there is a transition $\delta(p, \alpha) = (q, \gamma)$. We denote the reflexive, transitive closure of $\vdash_M$ as $\vdash_M^*$. We assume that the front and rear heads initially point to the beginning and end of the input string, that is queue initially contains the input string. The language accepted by $\vdash_M^*$ (which is the language accepted by M) denoted in set notation as follows.

$$L(M) = \{w \in \Sigma^* \mid (s, w) \vdash_M^* (f, e) \text{ for the initial state } s \text{ and for some } f \in F\} \tag{1}$$

d.

To prove that the queue-based deterministic TM is equivalent to the standard TM, we provide a one-to-one mapping between the operations of two machines.
Below, we list the mapping from the standard TM to the queue-based TM.

- To go left in the standard TM, it is necessary to enqueue read symbols while properly marking the needed positions of the queue with special symbols and then dequeue until the marking symbols to reach desired position in the queue-based TM.
- To go right in the standard TM, it is necessary to dequeue (while enqueuing needed symbols to not lose them for the other parts of the computation, if necessary) in the queue-based TM.
- To write a symbol on tape in the standard TM, it is necessary to enqueue in the queue-based TM.
- To read a symbol on tape in the standard TM, it is necessary to just check the symbol at the front in the queue-based TM.

Below, we list the mapping from the queue-based TM to the standard TM.

- To access the front element in the queue-based TM, it is necessary to go first symbol in the tape and read it in the standard TM.
- To access the rear element in the queue-based TM, it is necessary to go last symbol in the tape and read it in the standard TM.
- To enqueue in the queue-based TM, it is necessary to add desired symbol after the last symbol of the input tape in the standard TM.
- To dequeue in the queue-based TM, it is necessary to write blank symbol onto the first symbol of the input tape in the standard TM.

This completes the proof. Hence, the queue-based deterministic TM is equivalent to the standard TM.

e.

We formally define a queue-based deterministic TM M such that $L(M) = L$ where $L = \{wcw \colon w \in \{a, b\}^*\}$ as follows. $M = (K, \Sigma, \delta, s, F)$, where

- $K = \{q_0, q_1, q_2, q_3, q_4, y, n\}$,
- $\Sigma = \{a, b, c\}$,
- δ is given in Tab. 2,
- $s = q_0$, and
- $F = \{y, n\}$.

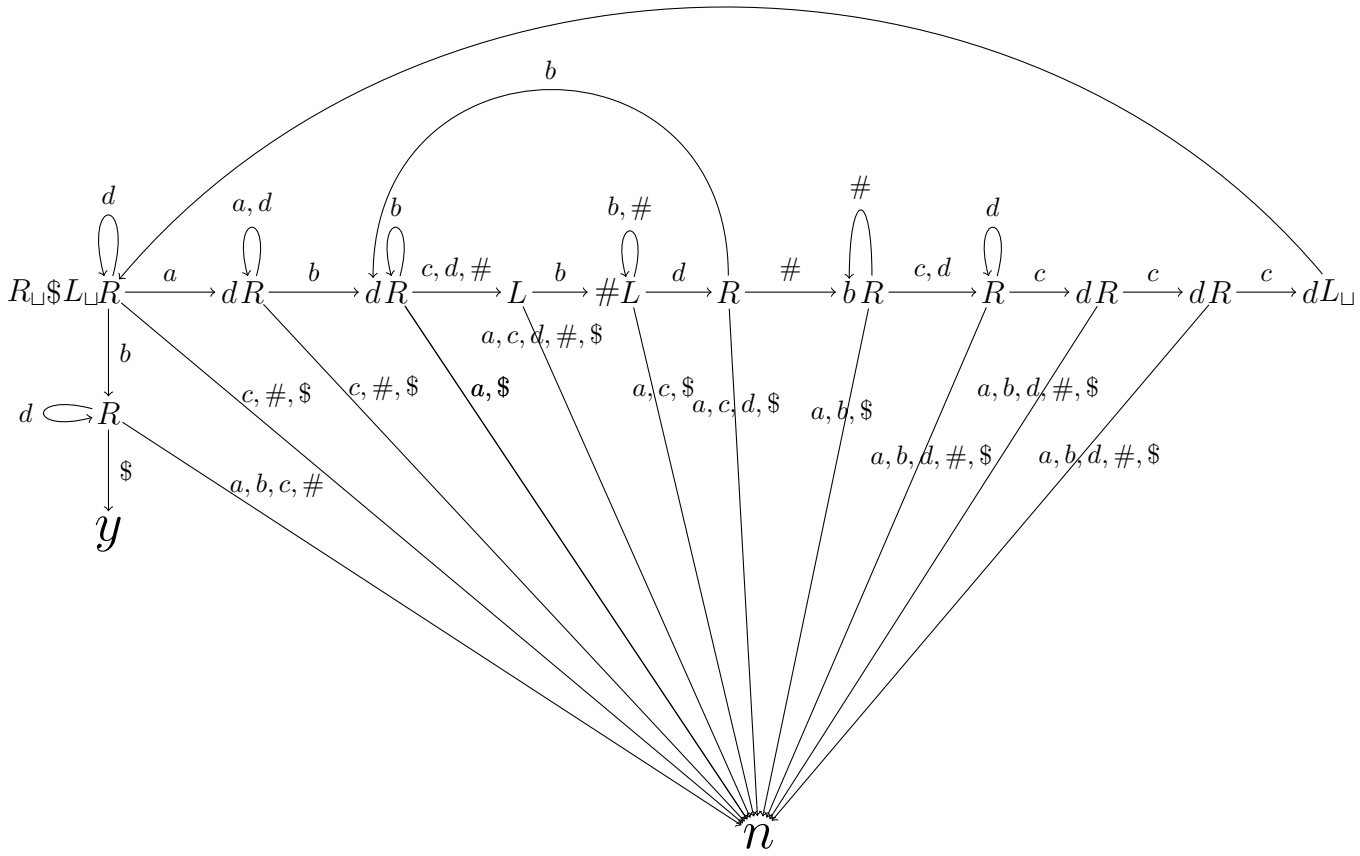
Table 2: δ function

q	σ	$\delta(q, \sigma)$
q_0	a	(q_1, c)
q_0	b	(q_2, c)
q_0	c	(y, e)
q_1	a	(q_1, a)
q_1	b	(q_1, b)
q_1	c	(q_3, e)
q_2	a	(q_2, a)
q_2	b	(q_2, b)
q_2	c	(q_4, e)
q_3	a	(q_0, e)
q_3	b	(n, e)
q_3	c	(n, e)
q_4	a	(n, e)
q_4	b	(q_0, e)
q_4	c	(n, e)

Answer 5

a.

Notice that for the machine given below, it is assumed that initially, the input is always in the following form: $\triangleright \sqcup w \sqcup$ for some $w \in \{a, b, c\}^*$. The notation in the book has been adopted and basic machines defined in the book are used such as $R, L, R_{\sqcup}, L_{\sqcup}$.



b.

We formally define a grammar G such that $L(G) = L$ where $L = \{a^n b^{2^n} c^{3n} : n \in \mathbf{N}\}$ as follows, $G = (\{a, b, c, S, S', T, G, B, C, \$\}, \{a, b, c\}, R, S)$, where R is given below.

$$
\begin{aligned}
S &\to aTBBCCC\$ \mid b \\
S' &\to aTGCCC \\
GB &\to BBG \\
GC &\to CG \\
G\$ &\to \$ \\
aTB &\to aS'B \mid aB \\
CB &\to BC \\
aB &\to ab \\
bB &\to bb \\
bC &\to bc \\
cC &\to cc \\
c\$ &\to c
\end{aligned}
\tag{2}
$$

Answer 6

a.

- Since L_1 can be generated by a regular grammar, it is a regular language. Since each regular language is accepted by some finite automaton, L_1 is accepted by some finite automaton. Any finite automaton can be mimicked by a Turning Machine, then, trivially, there exists a Turing Machine M_1 that accepts L_1.
- The information "A deterministic top-down parser can be built for context-free grammar whose language is L_2" implies that there exists a deterministic pushdown automaton that accepts L_2. Since any deterministic pushdown automaton can be mimicked by a Turning Machine, there exists a Turing Machine M_2 that accepts L_2.
- Even though it is ambiguous, a context-free grammar can provide L_3, then there exists a pushdown automaton that accepts L_3. Since any pushdown automaton can be mimicked by a Turning Machine, there exists a Turing Machine M_3 that accepts L_3.
- Since there is a Turing Machine that decides $\overline{L_4} \cap \overline{L(a^*b^*)}$, $\overline{L_4} \cap \overline{L(a^*b^*)}$ is a recursive language. If L is a recursive language, then its complement $\overline{L}$ is also recursive (Elements of Theory of Computation, page 199, Theorem 4.2.1), then $\overline{\overline{L_4} \cap \overline{L(a^*b^*)}} = L_4 \cup L(a^*b^*)$ is also recursive. $L(a^*b^*)$ is a regular language, then it is trivially recursive. Since a nondeterministic Turing Machine can be easily constructed for union of two recursive languages and it is know that nondeterministic Turing Machines are equivalent to deterministic Turing Machines, recursive languages are closed under union. Then since $L_4 \cup L(a^*b^*)$ and $L(a^*b^*)$ are recursive, L_4 is also a recursive language. Then there exists a Turning Machine M_4 that accepts L_4.
- A language is generated by a grammar if and only if it is recursively enumerable (Elements of Theory of Computation, page 229, Theorem 4.6.1), then L_5 is a recursively enumerable language. A language is recursively enumerable if and only if there is a Turing Machine that semidecides it (Elements of Theory of Computation, page 198, Definition 4.2.4), then there exists a Turing Machine M_5 that accepts L_5.

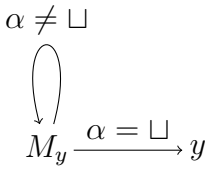
b.

- As it is argued in part a, L_1 is a regular language. Regular languages are subset of recursive languages in Chomsky hierarchy, then L_1 can be considered as a recursive language. For any recursive language there exists a Turning Machine that decides it, meaning a decider can be constructed for L_1 for membership problems. Hence there is always an algorithm for membership problems associated with L_1.
- As it is argued in part a, L_2 is a deterministic context-free language, then it is trivially a context-free language. Same argument as above applies for L_2, too. Thus, there is always an algorithm for membership problems associated with L_2.
- As it is argued in part a, L_3 is a context-free language. Same argument as above applies for L_3, too. Thus, there is always an algorithm for membership problems associated with L_3.

- As it is argued in part a, L_4 is a recursive language. For any recursive language there exists a Turning Machine that decides it. Thus, there is always an algorithm for membership problems associated with L_3.
- As it is argued in part a, L_5 is a recursively enumerable language. There are recursively enumerable languages for which there is no TM that decides them. Thus, we cannot say that there is always an algorithm for membership problems associated with L_5, in some cases there is and in some cases there is no such algorithm.

c.

To show this, we define the following machines. Notice that we assume any given input $w \in \Sigma^*$, where Σ is the alphabets of the machines, the tape is in the following form: $\triangleright \sqcup w \sqcup$. As it is argued in part b, there exists deciders for L_1, L_2, L_3, and L_4. Say machines M_1, M_2, M_3, and M_4 are deciders, i.e. they halt on any given input. Define two TMs $\overline{M_2}$ that accepts $\overline{L_2}$ and $\overline{M_4}$ that accepts $\overline{L_4}$. Those machines can be easily defined, just switch halting states of the original machines, i.e. say y is n and n is y. Define another TM M_x as follows: Operate as $\overline{M_2}$ if it halts on n for the given string then halt on n, else if it halts on y then for rest of the string operate as M_1. M_x accepts the language $\overline{L_2}L_1$. Define a TM M_y that halts on y for a given string if and only if both M_x and M_5 halt on y for the same string. Notice that M_y semidecides the language $\overline{L_2}L_1 \cap L_5$. Define M_y^* in basic machines notation as follows.



Define a TM M_3^* in a very similar manner as M_y^* (it is omitted since it is almost the same as M_y^*). Define a new TM M_z in basic machines notation as follows.

$$M_3^* \rightarrow \overline{M_4}$$

Finally, we define a TM M that accepts the given language L. M halts on y for a given string if and only if M_y^* halts on y for the given string or M_z halts on y for the given string. Notice that M semidecides L. M accepts any string in the given language L.

d.

The TM M defined for L in part c semidecides L. We cannot always come up with a TM that semidecides $\overline{L}$. When the machine does not halt for a string, we do not know whether the given string is not in the language or the machine is making a very long computation which will eventually accept the string, Since we cannot distinguish these two cases we cannot define a TM for $\overline{L}$. Notice that some TMs that semidecides their languages can be converted to a decider; however, there are TMs that cannot be converted into a decider. Due to this argument, it is known that recursively enumerable languages are not closed under complementation. Thus, we concluded that, we cannot always come up with a machine that semidecides $\overline{L}$.