CS 181 PRACTICE FINAL B

You may state without proof any fact taught in lecture.

1 Construct one-tape, deterministic Turing machines that decide the languages below. You must provide both a state-transition diagram and a detailed verbal description:

a. binary strings of the form $(0^n1)^k$, where $n \geqslant 0$ and $k \geqslant 0$;

b. binary strings (including ε) that represent properly nested parentheses, where 0 denotes an open parenthesis and 1 denotes a close parenthesis.

2 Give an algorithm that takes as input a DFA D and determines whether D accepts at least two palindromes.

3 A language L is called *downward-closed* if for every string w that L contains, L also contains every prefix of w. Give an algorithm that takes as input a regular expression R and decides whether the language that R generates is downward-closed.

4 A variable T in a context-free grammar G is called *essential* if every derivation of every string by G uses the variable T. Give an algorithm that takes as input a context-free grammar G and a variable T, and decides whether T is essential in G.

5 A string w is said to *cover* a given PDA P if there is a computation by P on w such that every state of P is visited. Give an algorithm that takes as input a PDA P and decides whether there is a string that covers P.

6 Prove that every infinite decidable language can be partitioned into two *disjoint* infinite decidable languages.

7 For each problem below, determine whether it is decidable and prove your answer:

a. on input a Turing machine M and a symbol σ, decide whether M ever writes σ on the tape in any computation;

b. on input a Turing machine M, decide if M recognizes a decidable language.

SOLUTIONS

CS 181 PRACTICE FINAL B

You may state without proof any fact taught in lecture.

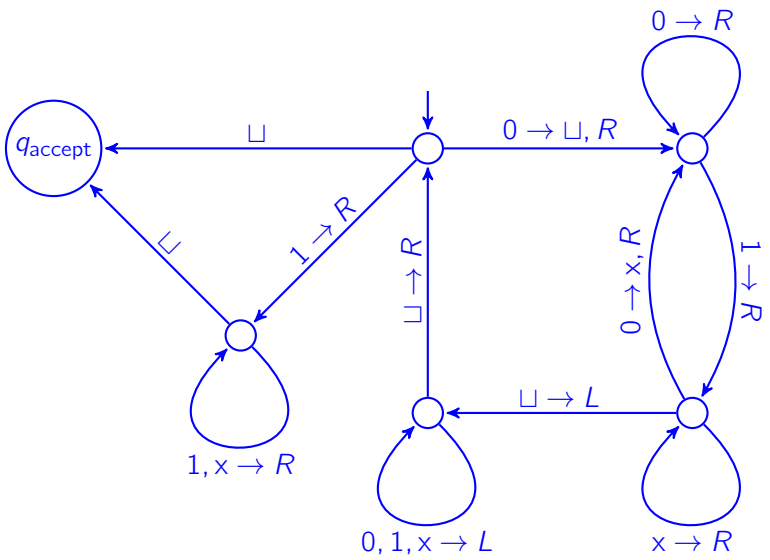
1 Construct one-tape, deterministic Turing machines that decide the languages below. You must provide both a state-transition diagram and a detailed verbal description:

a. binary strings of the form $(0^n1)^k$, where $n \geqslant 0$ and $k \geqslant 0$;

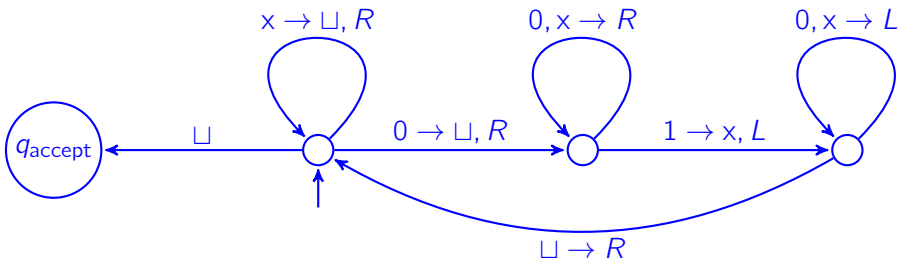
b. binary strings (including ε) that represent properly nested parentheses, where 0 denotes an open parenthesis and 1 denotes a close parenthesis.

Solution.

a. We view the input as the concatenation of blocks of the form 0^*1. Our TM iteratively removes a 0 from each block until there are no more 0s left. Each iteration starts by checking if the first block is just a 1. If so, we accept if and only if the remainder of the string contains no 0s. If the first block does begin with a 0, we erase that 0 and go to the next block (which amounts to fast-forwarding to the end of the current block, marked by a 1). We then make sure that that next block also has a 0, which we immediately replace with an x to mark it as processed. We process any remaining blocks analogously. Then, we scan left until we find the first blank, which puts us at the beginning of the string, and start a new iteration.



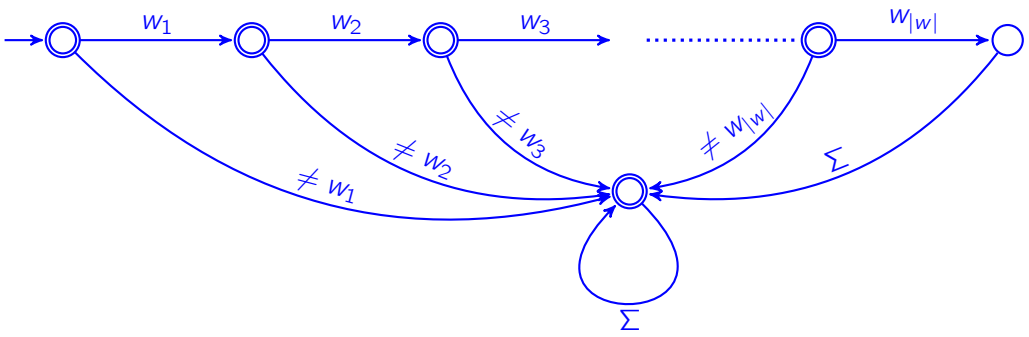
b. Recall from class that a string w is properly parenthesized iff: every prefix of w has at least as many 0s as 1s, and there are equally many 0s and 1s in w. Our TM verifies these conditions iteratively, each such iteration eliminating the nearest 0 and the nearest 1. An iteration starts by erasing any leading x's, which correspond to previously processed symbols. If that leaves us with an empty string, we accept. Otherwise, we make sure that the first unprocessed symbol is a 0, which we immediately erase to mark it as processed. Then, we scan forward to find the nearest 1 and replace it with an x to mark it as processed. Finally, we scan left until we find the first blank, which puts us at the beginning of the string, and start the next iteration. If we encounter an unexpected scenario (such as the string starting with a 1, or not having a 1 to go with the 0 found at the beginning of the iteration), we reject.



2 Give an algorithm that takes as input a DFA D and determines whether D accepts at least two palindromes.

Solution. Our solution uses a subroutine that inputs a DFA A and checks if A accepts a palindrome. This can be implemented as follows. First, construct a PDA P for the palindromes over the alphabet Σ of A, either from scratch or automatically from the following context-free grammar: $S \to \sigma S \sigma \mid \sigma \mid \varepsilon$ for every $\sigma \in \Sigma$. Now combine P and A using Cartesian product to obtain a PDA P' that recognizes the set of palindromes accepted by A. Finally, convert P' to a CFG G' and check if $L(G') \neq \varnothing$.

We now tackle the original problem. Given D, check if it accepts a palindrome. If it does not, output "no." Otherwise, run D on all palindromes ordered by length until you find one that D accepts: w. Next, form a DFA D' for the language $L(D) \setminus \{w\}$, and check if D' accepts a palindrome. If so, we output "yes"; otherwise, we know that w is the only palindrome accepted by D, and thus we output "no." The DFA D' for $L(D) \setminus \{w\} = L(D) \cap \overline{\{w\}}$ can be obtained by combining D via Cartesian product with the following DFA for $\overline{\{w\}}$:



3 A language L is called *downward-closed* if for every string w that L contains, L also contains every prefix of w. Give an algorithm that takes as input a regular expression R and decides whether the language that R generates is downward-closed.

Solution. Rephrasing, a language L is downward-closed if and only if $L = \text{prefix}(L)$. This suggests the following algorithm. Given R, first convert it into an equivalent NFA N. Then, construct an NFA N' that recognizes the prefix of the language of N. This can be done by identifying the states in N from which an accept state is reachable, and marking each such state as accepting. We output "yes" if and only if $L(N) = L(N')$. This check amounts to converting N and N' into DFAs and testing them for equivalence, as done in class.

4 A variable T in a context-free grammar G is called *essential* if every derivation of every string by G uses the variable T. Give an algorithm that takes as input a context-free grammar G and a variable T, and decides whether T is essential in G.

Solution. Let $G = (V, \Sigma, R, S)$ and $T \in V$ be given. By definition, T is not essential if and only if G generates some string without using T. So, construct the grammar $G' = (V, \Sigma, R', S)$, where R' is the set of all rules in R that do not contain T, and run the algorithm from class to check if $L(G') = \varnothing$. If so, T is essential; otherwise, T is not.

5 A string w is said to *cover* a given PDA P if there is a computation by P on w such that every state of P is visited. Give an algorithm that takes as input a PDA P and decides whether there is a string that covers P.

Solution. Let $P = (Q, \Sigma, \Gamma, \delta, q_0, F)$ be the given PDA. We first construct a new PDA P' that operates just like P but additionally keeps track of the set of states visited so far, accepting if and only if all states have been visited. Formally, $P' = (Q \times \mathscr{P}(Q), \Sigma, \Gamma, \Delta, (q_0, \{q_0\}), Q \times \{Q\})$, where the transition function Δ is easiest to describe as follows: whenever P has a transition from q to q' labeled $\sigma, \gamma \to \gamma'$, the new PDA P' will have, for every S, a transition from (q, S) to $(q', S \cup \{q'\})$ labeled $\sigma, \gamma \to \gamma'$. Now that P' has been constructed, we convert it into an equivalent context-free grammar G' and use the algorithm from class to check if $L(G') \neq \varnothing$. If so, we output "yes"; otherwise, we output "no."

6 Prove that every infinite decidable language can be partitioned into two *disjoint* infinite decidable languages.

Solution. As in class, we consider strings to be ordered according to "graded" lexicographic order: first the empty string, then all strings of length 1 in lexicographic order, then all strings of length 2 in lexicographic order, and so on. Let L be a given decidable language. We define A as the set obtained by ordering the strings in L and keeping the *odd-numbered* strings. Analogously, let B be obtained by ordering the strings in L and keeping the *even-numbered* strings. Then by construction, A and B are disjoint and infinite, with $L = A \cup B$.

To decide whether $w \in A$ for a given string w, run a decider for L consecutively on every string up to w. This computation terminates because there are finitely many strings preceding w, and the decider for L terminates on all inputs. We accept w if and only if $w \in L$ and there are an *even* number of strings in L preceding w. The decider for B works analogously, except we accept w if and only if $w \in L$ and there are an *odd* number of strings in L preceding w.

7 For each problem below, determine whether it is decidable and prove your answer:

a. on input a Turing machine M and a symbol σ, decide whether M ever writes σ on the tape in any computation;

b. on input a Turing machine M, decide if M recognizes a decidable language.

Solution.

a. Undecidable. For the sake of contradiction, suppose that this problem can be solved by some Turing machine T. Then the following algorithm is a decider for the halting problem.

Require: Turing machine M, string w

1: $\perp \leftarrow$ some symbol not in M's tape alphabet

2: $newTM \leftarrow$

```
Input: x
Run M on w
Write ⊥ in current cell
Accept x
```

3: **if** $T(newTM, \perp) =$ "yes" **then**
4: **return** "M halts on w"
5: **else**
6: **return** "M does not halt on w"
7: **end if**

On input M, w, we first identify a symbol $\perp$ not present in M's alphabet. Then, we construct a Turing machine called $newTM$ that has M and w hardwired into it. The new machine operates the same on all inputs, namely, it runs M on w, prints $\perp$ on the tape, and accepts. That way, M halts on w iff $newTM$ writes $\perp$ in some computation. Altogether this gives a decider for the halting problem. Since the halting problem is undecidable, we conclude that T cannot exist in the first place.

b. Undecidable. Let $\mathscr{C}$ be the set of decidable languages. Then $\mathscr{C}$ is a *nonempty* and *proper* subset of Turing-recognizable languages. Indeed, we showed in class that $\mathscr{C}$ contains $\{0^{2^n} : n \geqslant 0\}$ but does not contain the Turing-recognizable language HALT. Rice's theorem now implies that it is undecidable, on input a Turing machine M, whether $L(M) \in \mathscr{C}$.