University of Virginia - cs3102: Theory of Computation Spring 2010

PS4C

Problem 1: Designing Turing Machines. Design a deterministic Turing Machine that decides the language $\{0^n1^n2^n \mid n \geq 0\}$. You should provide descriptions of your Turing Machine at the three levels of detail described in Section 3.3:

a. Provide a *high-level description* of your machine.

Answer: The machine will go back and forth over the tape. Each iteration will start at the leftmost 0, replace it with a mark, then move right to find the leftmost 1 (if there is no such 2, reject), replace it with a mark, then move right to find the leftmost 2 (if there is no such 2, reject), replace it with a mark, and then return to the left edge of the tape. At the end, if there is no leftmost 0, then the tape is scanned and if there is any other symbol (not a mark or a blank) reject.

b. Provide an *implementation description* of your machine.

Answer: Our machine has 6 main states: q_0 (the start state); q_{find1}, a state for finding the leftmost 1; q_{find2}, a state for finding the leftmost 2, q_{find0}, a state for finding the rightmost remaining 0; and q_{check}, a state for checking the final tape configuration.

We start in state q_0 by reading a 0 from the tape, replacing it with a \$ that marks the left edge, moving right and transitioning to state q_{find1}. If the input symbol is blank in q_0, accept (the empty input string). For any other tape symbol in state q_0, reject.

In state q_{find1}, move right (passing over 0s and Xs), until a 1 is found. Then, cross of that 1 by writing an X, transition to state q_{find2} and move right. If any other symbol is found, reject.

In state q_{find2}, move right (passing over 1s and Xs), until a 2 is found. Then, cross of that 2 by writing an X, transition to state q_{find0} and move left. If any other symbol is found, reject.

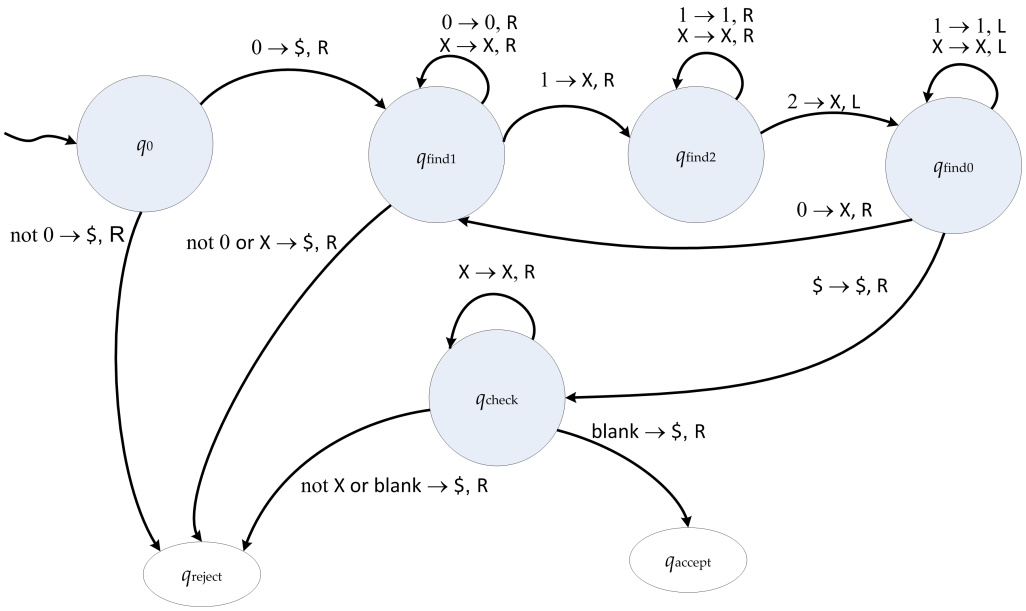
In state q_{find0}, move left (passing over 1s and Xs), until a 0 is found. Then, cross of that 0 by writing an X, transition to state q_{find1} and move right. If the \$ marker at the left edge of the tape is found, transition to state q_{check} and move right. If any other symbol is found, reject.

In state q_{check}, we are checking that there are no unmatched symbols left on the tape. Move right across the tape, checking that each square contains an X until the first blank is reached, then accept. If any other symbol is read, reject.

c. Provide a full *formal description* of your machine (this is tedious, but everyone should do it once!). You may use any TM simulator you want to develop and test your solutiemon, but it is not necessary to do so. One good TM simulator is JFLAP: http:

//www.jflap.org/jflaptmp/. But be careful, it simulates a doubly-infinite tape TM (see Problem 4), not the standard TM we defined.

Answer: The full formal description needs to show the states and all the transition rules. It can be built from the implementation description.



Problem 2: Deciding *SQUAREFREE*. Provide an *implementation description* of a Turing Machine that decides the language of squarefree sequences in $\{a,b,c\}^*$ (from Exam 1). A *squarefree* sequence is a sequence that contains no adjacent repeating subsequences of any non-zero length. For example, ab and $abcba$ are squarefree, but aa is not since it contains a^2 and $abcbacbab$ is not since it contains $(cba)^2$. Your description can be at a high-level, but should be detailed enough to convince a skeptical reader that there is a TM that decides *SQUAREFREE*. (For clarity of presentation, you may want to start by providing a high-level description.)

Answer. We can use a brute-force solution to exhaustively look for all possible squares. If a square is found, we reject. If no square is found after trying all possibilities, we accept.

To exhaustively check for all possible squares, we can exhaustively check for squares starting with each symbol in the input sequence. Here is a Python program:

```
def is_square_free(w):
   for i in range(len(w)):
      for j in range(1, len(w)):
         if w[i:(i+j)] == w[(i+j):(i+j+j)]:
            return False
   return True
```

If we believe the Church-Turing thesis, a Python program that is guaranteed to terminate (this one is, because both the loops are over finite ranges of integers) means there must be a Turing machine that decides the language. We could create it by first creating a TM that simulates a Python interpreter, then run that TM on the given Python program.

We could describe an implementation-level TM more directly by following the same algorithm. The challenge is how to keep track of the loops with a TM. Note that we cannot use the states to do this, since the length of the input is unbounded so a finite number of states is not enough. Instead, we can use the tape. One way would be to add marked versions of all the input symbols to the tape alphabet:

$$\Gamma = \left\{a, b, c, \hat{a}, \hat{b}, \hat{c}, \bar{a}, \bar{b}, \bar{c}, \ddot{a}, \ddot{b}, \ddot{c}, \times, \sqcup\right\}$$

To model the outer loop, we move left to right across the input, replacing each symbol with an $\times$ after it has been completely checked. To model the inner loop, we ust the marked input symbols to keep track of the string we are matching. We could start marking the leftmost j symbols by adding the $\hat{x}$ to the symbol (we don't need to keep track of j, just mark one more symbol each time). Then, to check if there is a square, we go left-to-right across this string, replacing $\hat{x}$ with $\bar{x}$, and checking if the next symbol that is not hatted or barred matches x. If it does, we have a possible square, so mart it as $\ddot{x}$, search back across the tape for $\bar{x}$ and move on to the next symbol. If we reach a symbol marked as $\ddot{x}$, we have found a square, *reject*. If not, keep going until there is either a mismatch of a square is found. If a mismatch is found, we need to wipe all the markings of the test match (replacing $\ddot{x}$ with x as we move across the tape), and then look for the next longest square. Once a $\sqcup$ is reached, we know there are no squares that start with this symbol. Cross off that symbols with a $\times$, and move on to the next one. If there is no next symbol (we have reached a $\sqcup$), *accept*.

Problem 3: Equivalence of 2-DPDA+ϵ and Turing Machine. In Class 14, we informally argued that a 2-stack deterministic pushdown automaton (with forced ϵ-transitions) is equivalent to a Turing Machine. For this problem, prove one direction of the equivalence: that a 2-DPDA+ϵ can simulate any Turing Machine. (Hint: your proof should show how to construct a 2-DPDA+ϵ from a given Turing Machine. For a clear proof, you will need to define a formal notation for describing a 2-DPDA+ϵ.)

Answer. First, we need a notation for describing a 2-DPDA+ϵ, which is similar to a regular DPDA but with two stacks. We assume both stacks use the same alphabet. So, we can describe it similarly to a regular DPDA, except the transition function has extra inputs and outputs. A 2-DPDA+ϵ is a 6-tuple $(Q, \Sigma, \Gamma, \delta, q_0, F)$ where Q, Σ, Γ, q_0, and F are defined as they are for a DPDA, and

$$\delta\colon Q \times \Sigma_\epsilon \times \Gamma_\epsilon \times \Gamma_\epsilon \to Q \times \Gamma_\epsilon \times \Gamma_\epsilon$$

The two Γ_ϵ inputs correspond to the two stacks, and we have added ϵ to Σ to allow the forced ϵ transitions.

Note that we include Σ_ϵ in the δ inputs to be consistent with the PDA model where the input is separate from the tape. We will model a TM with a 2-DPDA by first pushing the input onto the stack, and then simulating the TM. So, all the rules except for the special input pushing state will use ϵ as the input symbol (that is, we don't start simulating the TM execution, until the PDA input has been fully consumed and placed on the stacks).

Now, we need to show that we can simulate any Turing Machine with a 2-DPDA+ϵ. We can model the TM tape using the 2 stacks. The left stack will contain everything to the left of the TM head, so the top of the stack corresponds to the square one position to the left of the head and the bottom of the stack is the left edge of the tape. The right stack contains everything from the TM head to the right, so the top of the right stack corresponds to the current square and the bottom of the right stack contains the rightmost non-blank symbol on the tape. (Note that is would also be perfectly valid to make the top of the left stack represent the TM head, but the rules for the construction would be different, and setting up the initial input would be slightly more complicated.)

Before simulating the TM, the 2-DPDA processes the input by pushing it on the left stack first, then moving it onto the right stack. We need to do this to get the input in the correct order on the right stack - if we just pushed it directly on the right stack, we would end up with the input in reverse order. To do this, we introduce two states, $q_{pushinput}$ (for pushing the input on the left stack) and $q_{popinput}$ (for moving the input onto the right stack in the correct order). We also put \$ markers on the bottoms of both stacks.

$\delta(q_0, \epsilon, \epsilon, \epsilon) = (q_{pushinput}, \$, \$)$
$\delta(q_{pushinput}, a, \epsilon, \epsilon) = (q_{pushinput}, a, \$)$ for all $a \in \Sigma$
$\delta(q_{pushinput}, \epsilon, \epsilon, \epsilon) = (q_{popinput}, \epsilon, \epsilon)$
$\delta(q_{popinput}, \epsilon, a, \epsilon) = (q_{popinput}, \epsilon, a)$ for all $a \in \Sigma$
$\delta(q_{popinput}, \epsilon, \$, \epsilon) = (q_0, \$, \epsilon)$

The state, q_0 corresponds to the state q_0 in the simulated TM. When we reach q_start, the right stack contains the original input in the correct order (with a \$ on the bottom) and the left stack only contains \$.

To simulate the TM, we need to define the corresponding δ rule for each possible TM rule. The states of the 2-DPDA will include the input producing states above, a state corresponding to each state in the TM, and two special version of each TM state (we will see how these are used next):

$$Q_{2DPA} = \{q_0, q_{pushinput}, q_{popinput}\} \cup Q_{TM} \sup \{\hat{q} | q \in Q_{TM}\} \sup \{\dot{q} | q \in Q_{TM}\}$$

If the TM δ_{TM} function includes a rule, $\delta_{TM}(q, b) \rightarrow (q_r, c, R)$ then, for the 2-DPDA+ϵ,

$\delta(q, \epsilon, b) = (q_r, c, \epsilon)$
That is, we pop a b from the right stack and push a c on the left stack.

If the TM δ_{TM} function includes a rule, $\delta_{TM}(q, b) \rightarrow (q_r, c, L)$ then, for the 2-DPDA+ϵ,

$\delta(q, \epsilon, b) = (\hat{q}_r, \epsilon, c)$
$\delta(\hat{q}_r, a, \epsilon) = (q_r, \epsilon, a)$ for all $a \in \Gamma - \{\$\}$

Note that it is necessary to add an extra state to simulate this rule, since we need to push *two* symbols on the right stack when we move left. We cannot do this with one transition rule, so we need to add the extra state $\hat{q}$ with a transition to q that moves the top of the left stack onto the right stack.

The only remaining issue to deal with is simulating the left and right edges of the tape correctly.

For the left edge, there is nothing to the left of the tape head. This corresponds to the situation where the top of the left stack is the \$ marker. If the rule moves left, we should not move the \$ to the right stack:

$\delta(\hat{q}_r, \$, \epsilon) = (q_r, \$, \epsilon)$ for all states $q \in Q_{TM}$

For the right edge, the Turing Machine adds new blanks to move right. This corresponds to the situation where the top of the right stack is the \$ marker. If this arises, we whould push a blank on the right stack:

$\delta(q, \epsilon, \$) = (\dot{q}, \epsilon, \$)$ for all states $q \in Q_{TM}$
$\delta(\dot{q}, \epsilon, \epsilon) = (q, \epsilon, \sqcup)$ for all states $q \in Q_{TM}$

Note that we need two rules, and a new set of states, to do this since it requires keeping the \$ on the bottom of the stack as well as pushing the $\sqcup$.

Problem 4: Robustness of Turing's Model. (Sipser problem 3.11) A *Turing machine with doubly infinite tape* is similar to an ordinary Turing machine, but its tape is infinite to the left as well as to the right. The tape is initially filled with blanks except for the portion that contains the input. Computation is defined as usual except that the head never encounters an end to the tape as it moves leftward. Show that the class of languages recognized by a Turing machine with doubly infinite tape is equivalent to the class of languages recognized by a regular Turing machine.

Answer:

To show equivalence, we need to show proofs in both directions.

A doubly-infinite TM can simulate a regular TM. This seems obvious (since the doubly-infinite tape adds functionality to the TM), but is a bit subtle because of the way a regular TM behaves when it would go off the left edge of the tape. We need to show we can simulate that same behavior with a doubly-infinite tape. One way to do this is to add a new symbol to the tape alphabet, #, to mark the edge of the tape. Add a step at the beginning of the TM that moves left one square, writes a #, and moves right back to the original starting square. Then, from each state, add a transition rule that says if you see a #, move right:

$\delta(q, \#) = (q, \#, R)$ for all $q \in Q$

This simulates the same behavior as the regular TM since if the machine ever tries to go to the # square, it moves back to the square representing the left-most square on the regualar TM.

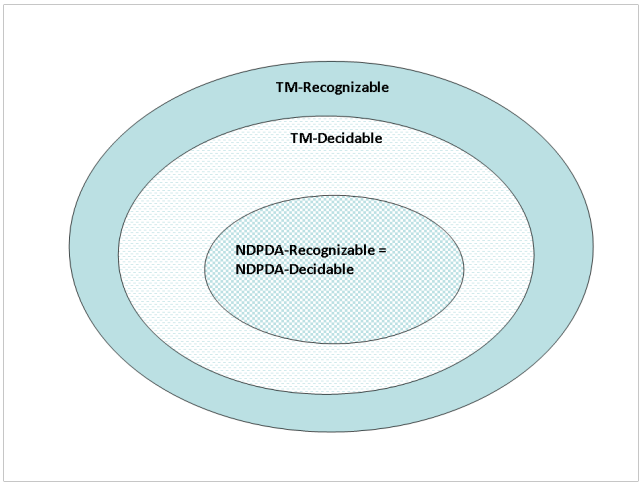
A regular TM can simulate a doubly-infinite TM. One way to show this is to take advantage of the property established in the previous question: a 2-DPDA+ϵ is equivalent to a TM. Thus, if we can simulate a TM with a doubly-infinite tape using a 2-DPDA+ϵ, then we can simulate it with a regular TM, which we know is equivalent to a 2-DPDA+ϵ. To prove this, we just need to modify our proof of equivalence to the 2-DPDA+ϵ to change how we deal with the bottom of the stack when moving left. This is actually simpler than the original proof: we just add an extra set of states ($\bar{q}$ for each state $q \in Q_{TM}$) and replace the special rules we added for the left edge with rules similar to those we added for the right edge that push a blank on the left stack:

$\delta(q, \$, \epsilon) = (\bar{q}, \$, \epsilon)$ for all states $q \in Q_{TM}$
$\delta(\bar{q}, \epsilon, \epsilon) = (q, \sqcup, \epsilon)$ for all states $q \in Q_{TM}$

Problem 5: Deciders and Recognizers. In Class 15, we argue that DFAs, DPDAs, and NFAs always terminate, hence the sets of languages that can be recognized by these machines are equivalent to the sets of languages that can be decided by them. By contrast, a Turing Machine may run forever on some inputs, so there may be languages which can be recognized but not decided by a Turing Machine (we will see examples of such languages next week). What about nondeterministic pushdown automata?

a. Draw a Venn diagram including these sets:
   - *DPDA-Decidable* (DPDA-D): languages that can be *decided* by a deterministic PDA.
   - *NPDA-Decidable* (NPDA-D): languages that can be *decided* by a nondeterministic PDA.
   - *TM-Decidable* (TM-D): languages that can be *decided* by a Turing Machine.
   - *NPDA-Recognizable* (NPDA-R): languages that can be *recognized* by a nondeterministic PDA.
   - *TM-Recognizable* (TM-R): languages that can be *recognized* by a Turing Machine.

   Answer: The set of languages that can be decided by an NDPDA is the same as the set of languages that can be recognized by a NDPDA, namely the context-free languages.



b. Prove that your diagram correctly depicts the relationship between *NDPDA-Recognizable* and *TM-Decidable*.

Answer:
This is really just asking if a TM can decide all context-free languages and at least one language that is not a context-free language. Your answer to Problem 1 demonstrates that there are some languages in TM-Decidable that are not in NDPDA-Recognizable. We know a TM can decide all context-free languages since we can simulate a NDPDA with a TM.

c. Prove that your diagram correctly depicts the relationship between *NDPDA-Recognizable* and *NDPDA-Decidable*. (For example, if your diagram showed them as the same circle, prove that the sets are equivalent. If your diagram shows one set inside the other, prove that all languages in one set are in the other set but that there is some language in the outer set that is outside the inner set.)

Answer: To show that *NDPDA-Recognizable* and *NDPDA-Decidable* are the same we need to argue for every language described by a CFG, there is some way of describing the same language for which the language acceptance is guaranteed to terminate. One way to do this is to use the fact that every CFL can be generated by a context-free grammar in Chomsky normal form (Sipsers Theorem 2.9). In this form, it is clear to see that for a given-length string, there are a limited number of steps that could be used to generate the string. Hence, the process of generating the string will always terminate. This shows that any language that can be recognized by an NDPDA can also be decided by an NDPDA, since there is an NDPDA equivalent to any CFG.