6.840
Theory of Computation
2004/10/07

Dan Ports
drkp@mit.edu

10/10

Problem 2-1

Let $A = \{wxw^{\mathcal{R}} : w, x \in \{0,1\}^*, |w| = |x|\}$. We show that A is not a CFL. For contradiction, suppose that it is. Then it has a pumping length p.

Consider the string[1] $s = 0^{2p}1^p0^p0^{2p}$. This string is in A. By the pumping lemma, this string can therefore be expressed as $uvxyz$, with $|vy| > 0$ and $|vxy| \leq p$. We now consider the cases based on which part of s contains vxy.

1. $\underbrace{0^{2p}}_{vxy}1^p0^p0^{2p}$. p is contained in the initial $2p$ zeros. Then v and y contain only zeros, and either v or y must contain at least one zero. So if we pump it up by $6p$, creating $uv^{6p}xy^{6p}z$, we have the string $0^{8p}1^p0^p0^{2p}$, possibly with (possibly many) additional preceding zeros. Note that the string has length at least $12p$, so the 1^p section is in the final third. But the first third consists only of zeros. So the final third cannot be the reverse of the first third.

2. $0^{2p}1^p\underbrace{0^p0^{2p}}_{vxy}$. The argument is symmetric: v and y contain only, and at least one, zeros. So we pump it up by $6p$, and the first third contains the ones, while the final third contains only zeros.

3. $\underbrace{0^{2p}1^p}_{vxy}0^p0^{2p}$. In this case, v contains only zeros and y contains only ones (we consider the case where v or y straddles the 0-1 boundary separately). In this case, we can pump v and y up by several factors of p, so that part of the 1^p section is "pushed over" into either the first or the final third (depending on the relative lengths $|v|$ and $|y|$, while the opposite third contains only zeros. (In the vacuous cases, e.g. $|y| = 0$, this reduces to a previous case.)

4. $0^{2p}\underbrace{1^p0^p}_{vxy}0^{2p}$. In this case, v contains only ones and y contains only zeros. This case is symmetric to the previous case: we can pump the string up by several factors of p, pushing ones into one of the end thirds, depending on the relative lengths (and emptiness) of v and y.

5. Finally, we consider the case where either v or y spans the 0-1 boundary, containing both zeros and ones. Note that it cannot be the case that *both* v and y contain both zeros and ones, since $|vxy| \leq p$ and the boundaries are separated by p ones. So one contains both zeros and ones, and the other contains either zeros or ones (or possibly the empty string; this does not change the analysis. Pumping up the strings v and y increases the length of the string and pushes the ones away from the boundary being pumped, into the opposite third. Thus it cannot be the same as the other non-middle third, which contains a different quantity of zeros and ones.

We have shown that by pumping the string s we can create strings that are not in A. By violating the pumping lemma, this is a contradiction $\leftrightarrow$. So A is not a CFL.

[1] Thanks are due to Austin Clements, who devised this simpler string, saving myself (and the world) from analyzing my original string, $0^{2p}1^{2p}0^p(01)^p1^p1^{2p}0^{2p}$.

And saved me from having to check your analysis. Thanks.

10/10

6.840 **Dan Ports**

Theory of Computation drkp@mit.edu

2004/10/07

Problem 2-2

We first show that a Turing machine can simulate a queue automaton. For simplicity, we use a multi-tape Turing machine, with one tape holding the input, and one tape representing the queue. As in the queue automaton, the input tape head will only move to the right. As each input symbol is presented, we simulate the queue automaton using the same state machine that controls the queue automaton. We simply need to simulate the two operations of pushing an element onto the end of a queue, and popping an element from the front of the queue. We simulate the queue on one of the Turing machine's tapes by representing the queue as a start delimiter character $\wr_L$, followed by the elements in the queue from least-recently-inserted to most-recently-inserted, followed by an end-delimiter character $\wr_R$. To insert an element, we move the tape head right until we reach the $\wr_R$ delimiter, then write the new element over it, move right again, and write a new $\wr_R$ delimiter. To remove an element, we move the tape head left until we reach the $\wr_L$ delimiter, then move right once to find the oldest element in the queue. We read this element (and use it for processing in the state machine), then replace it with the new $\wr_L$ delimiter. Thus the Turing machine can simulate a queue automaton.

We now show that a queue automaton can simulate a Turing machine. We will represent the tape as a sequence of elements on the queue, starting with the one that the Turing machine's head is currently pointing at, and proceeding to the right until we reach the last used symbol on the tape. We then place a delimiter $\wr$ on the queue, followed by the tape symbols from the left end of the tape to the symbol immediately preceding the current head position. We begin by pushing the input onto the queue in order, followed by a $\wr$ delimiter. To read a symbol from the tape, we pop the current element from the queue and read its value; to write a symbol to the tape, we push it onto the queue. This process also moves the simulated head one symbol to the right as a side-effect, so to move the head to the right we do nothing. To move the head to the left is more complex. Instead of pushing the newly-written symbol onto the queue, we pushed a "marked" version of the symbol (this requires doubling the size of the tape alphabet, but this is not a problem). We then scan through the queue, popping elements and then pushing them back onto the queue, with a 1-element "delay" or "memory" that lets us keep track of the previous element popped. We perform this procedure until we reach the symbol we have just marked. We then mark the previous (unmarked) symbol and push it onto the queue, and push the unmarked version of the marked symbol read from the queue. We then repeat the same scanning procedure until we reach the newly-marked symbol. This is the symbol that is on the tape to the left of the head's previous position, so in performing this procedure we have moved the head one symbol to the left and read the symbol on it. By using these simulated operations, we can use the Turing machine's state machine controller on the queue automaton to simulate the Turing machine.

Thus we have shown that a Turing machine can be simulated by a queue automaton, and previously that a queue automaton can be simulated by a Turing machine, so the two are equivalent.

6.840
Theory of Computation
2004/10/07

Dan Ports
drkp@mit.edu

10

Problem 2-3

We show that a language is decidable iff an enumerator enumerates the language in lexicographic order.

First, let D be a decidable language, and M be the Turing machine that decides D. We create an enumerator E that enumerates D:

$E =$ "

1. Repeat:
   (a) Generate the next element x in lexicographic order in the set of all strings that can be generated from the symbols in the alphabet.
   (b) Apply D on input x. D is a decider, so it will terminate. If it accepts, output x. Otherwise do nothing."

Next, let E be an enumerator that enumerates a language D in lexicographic order. We show that D is decidable. There are two cases: either the enumerator E at some point enters an infinite loop without producing any more output, or it does not. If it does, then the language is finite, and hence it is trivially decidable. Otherwise D is infinite.

So we define a machine M: $M =$ "On input ¡x¿

1. Run the enumerator E
2. For every output y from E,
   (a) If $y = x$, accept.
   (b) If $y > x$, reject.

Since the output of E is infinite, it will eventually produce an output y greater than x in lexicographic order. So the machine is a decider.

6.840
Theory of Computation
2004/10/07

Dan Ports
drkp@mit.edu

10

Problem 2-4

Let C be a language. We show that C is Turing-recognizable iff and only if there is a decidable language D such that $C = \{x \ : \ \exists y | \ (\langle x, y \rangle \in D)\}$

Suppose there is a decidable language D such that $C = \{x \ : \ \exists y | \ (\langle x, y \rangle \in D)\}$. We can create a Turing machine M that recognizes C: $M =$ "On input x,

1. For every string y that can be generated from the symbols in the alphabet,
   (a) Run a decider for D on $\langle x, y \rangle$.
   (b) If D accepts, accept."

This machine accepts on any x in C, and loops infinitely otherwise.

We now show the converse. Suppose that C is Turing-recognizable. We show that there exists a decidable language D such that for all x in C, there exists a y such that $\langle x, y \rangle$ is in D. Let M be a recognizer for C.

Define D to be the language of pairs $\langle x, y \rangle$, where x is an input to M and $y \in \mathbb{N}$, for which M accepts before performing more than y state transitions when given x as input. One can construct a decider for D: it simulates M on x for at most y state transitions, accepting if M accepts, and rejecting if M rejects or does not decide before y transitions. This is clearly a decider, since it always accepts or rejects in bounded time.

Let $x \in C$ be given. Then M will accept when given x as input. By definition, it does so in finite time. Define y to be the number of state transitions performed by M before accepting with input x. Then $\langle x, y \rangle$ is in D by definition. So we have proven the claim.

10

6.840
Theory of Computation
2004/10/07

Dan Ports
drkp@mit.edu

Problem 2-5

Define the language U_{PDA} to be all PDAs that have useless states. We show that U_{PDA} is decidable. Suppose we are given a PDA P that contains a state q. We can define P' to be identical to P except that q is the only accept state. Observe that since q is useless if it is never reached, q is useless if and only if P' accepts no strings. Since the emptiness testing E_{CFG} for a CFG is decidable, U_{PDA} is decidable:

Let M be the following TM:
M = "On input P, where P is a PDA,

1. For each state q in P,
   (a) Generate the PDAP' where q is the only accept state.
   (b) Decide $E_{\mathsf{PDA}}(P')$. If P' is accepted, q is useless, so accept.
2. If every state q has been tested without accepting, then no states are useless, so reject."

6.840
Theory of Computation
2004/10/07

Dan Ports
drkp@mit.edu

Problem 2-6

10

Consider the language PAL_{DFA} containing all DFAs that accept some palindrome. We show that it is decidable.

Recall that the language PAL of all palindromes is context-free. (It can be generated using the CFG PAL $\rightarrow a\text{PAL}a|\epsilon$ for all a in the alphabet.) Therefore, for any DFA D, the language $D_{PAL} = PAL \cap D$ containing all palindromes accepted by D is a CFL. This is empty iff D accepts no palindromes, and determining the emptiness of a CFL is decidable, so we can decide PAL_{DFA} with the following machine:

$M =$ "On input D, where D is a DFA,

1. Generate the CFL D_{PAL}.
2. Decide E_{CFL} on input D_{PAL}
3. If $D_{PAL} \in E_{\mathsf{CFL}}$, reject, otherwise accept.