ARSDIGITA VNIVERSITY

Month 8: Theory of Computation

Problem Set 4 Solutions - Rusty Chris, Dimitri Kountourogiannis, and Mike Allen

1. Context Free or Not

a. **CF**. Here is a grammar that will generate the language.

```
S -> 1A0
A -> 1A0 | B
B -> CC
C -> 0D1
D -> 0D1 | e
```

b. **Not CF**. Assume for the purpose of contradiction that it is. Then let the pumping length be p. Consider the string $s=0^p1^p\#0^p1^p$, which is in the language. If we decompose s into s=uvwxy as in the statement of the pumping lemma, there are three cases to consider.

1. If vwx is contained in the first half of the string s, then pumping up even once (that is, taking the string uv^2wx^2y) will give us a string where the first half is longer than then second half, which means it can't be a substring of the second half.
2. If vwx is contained completely in the second half, then when we pump down (that is, take the string uwy). Then the second half will be shorter than the first half (since |vwx|>=1) and so the pumped down string will not be in the language.
3. If vwx overlaps with the symbol #, then it must be the case that the # is contained in w, or else pumping up would give too many # symbols. So either the v will be a string of 1's or the x will be a string of 0's or both (We need the fact that |vwx| <= p here). If v is a nonempty string of 1's then pumping up will give the left half more 1's then the right side, so the left side will not be a subset of the right side. If x is a nonempty string of zeros then pumping down will give the right side fewer zeros than the left so the left side will not be a subset of the right side.

In any case we come to the conclusion that no matter how we choose the decomposition, the conditions of the pumping lemma will be violated. So the language cannot be regular.

c. **Not CF**. Suppose it is. Then let the pumping length be p. Consider the string $s=0^p1^p01^p$, which is in the language. If we decompose s into s=uvwxy as in the statement of the pumping lemma, then an argument like the previous one shows that if vwx is anything but the lone zero between the ones, then pumping up will give you something that is not in the language. On the other hand if vwx is the lone zero, then pumping down will give you. $0^p1^p1^p$, which is not in the language. This contradicts the pumping lemma, so therefore the language is not context free.

d. **CF**. Here is a grammar that generates the language, with some comments on the side to explain what each non-terminal represents.

```
S -> ABA |
A -> e | AC
C -> 01 | 0C1     ; 0^n1^n
B -> 0B1 | D | F  ; 0^m1^nm!=n
D -> 0 | 0D       ; one or more 0s
F -> 1 | 1F       ; one or more 1s
```

Why this works: Since R=(00*11*)(00*11*)* is regular, the complement ~R includes all strings which are not even of the correct form. We can Union 0*1* complement... nondeterministically guess which pair mismatches.

2. Decision Algorithms

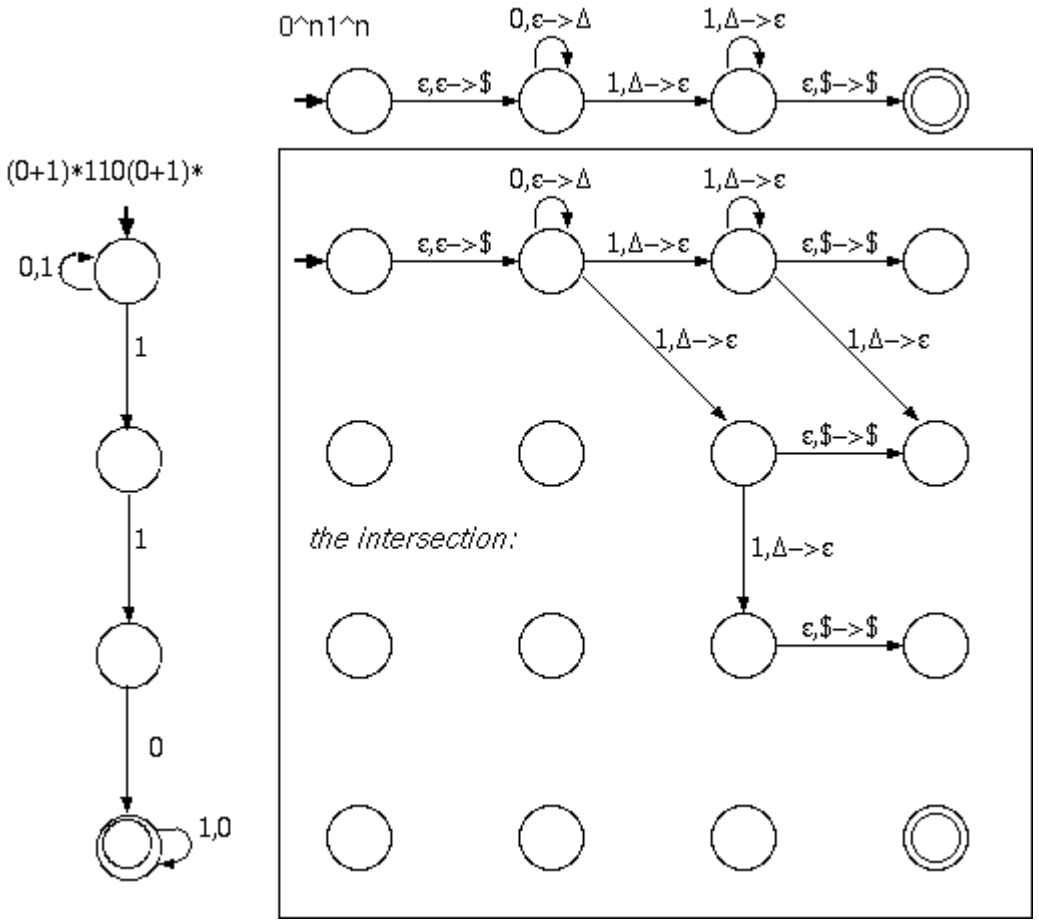
a. If it were a non-deterministic machine, then the problem would be undecidable, but since Deterministic machines are closed under complement, we can do this. Complement the machine. Then form a grammar from the machine. Then check whether the start symbol is useless. If the start symbol is useless then the complemented machine has an empty language and so the original machine generates every string. If the start symbl is not useless, then the original machine does not generate everything.

b. Convert the grammar to Chomsky Normal Form. Check all possible derivations of 2n-1 steps, where n is the length of the string z in the language.

c. We need to see if the language has any intersection with 1*. Convert the CFG into a PDA called M.Since 1* is regular and Context Free Languages are closed under intersection with regular languages, we can intersect the DFA that generates 1* with M. Convert the resulting machine back to a Grammar. Check if the resulting language is empty by testing whether the start symbol is useless. Another way to do this is to put the grammar in Chomsky Normal form, and then compute all derivations of length less than or equal to $2^{|V|+1}$, where $|V|$ is the number of nonterminal variables in the CNF grammar. If no 1* is derivable in this many steps, then none will ever be.

3. Closure Problems of CFLs

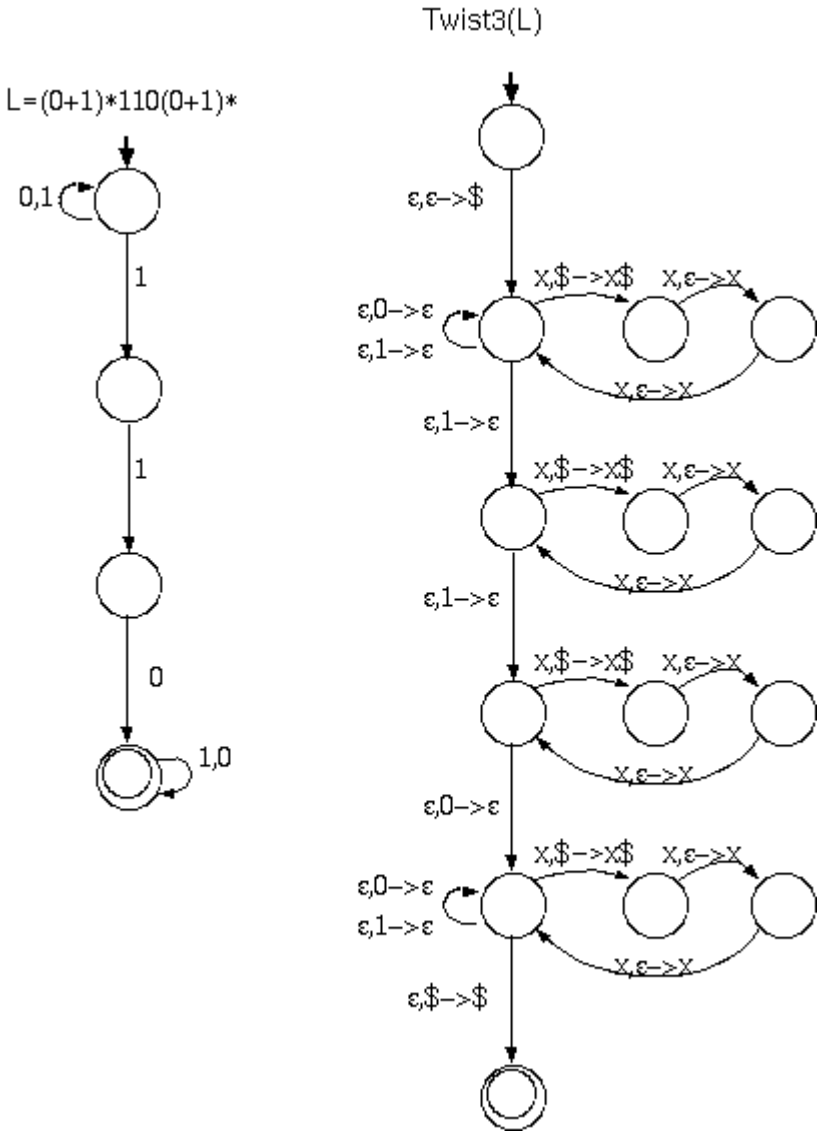
a. The intersection of a PDA M1 and an FSM M2 is constructed in a manner analogous to how two FSMs are intersected. The stack of the constructed PDA is used to simulate the stack of M1.



Note the accept state is not reachable in this case, which is good since the intersection of the two languages is empty.

b. This is trivial because we can take any CFL that is not regular and intersect in with any Regular language that contains it, for example we can let L= $\{0^n1^n \mid n >=0\}$ and R = (0+1)*. Then L intersected with R is just L again, which we know to be non-regular.

c. We modify the DFA for L so that the stack to supplies the input to the machine and to each state we add a gizmo that reads in string input and pushes it onto the stack 3 characters at a time.



4. Parsing and the CYK Decision Algorithm

a. The CYK parse table for 00000

```
   |   1     2     3     4     5
---|-----------------------------------
 1 |  A,C   A,C   A,C   A,C   A,C
 2 |   B     B     B     B
 3 | S,A,C S,A,C S,A,C
 4 |   B     B
 5 | S,A,C
```

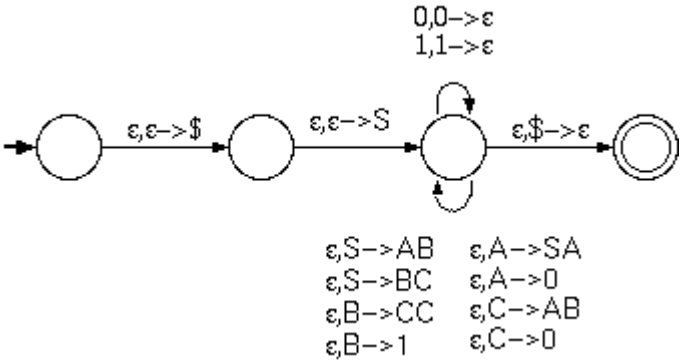
Here we conclude that 00000 is derivable from the start symbol. Since S is in the fifth row, first column of the table.

The CYK parse table for 000000

```
   |   1     2     3     4     5     6
---|-----------------------------------------
 1 |  A,C   A,C   A,C   A,C   A,C   A,C
 2 |   B     B     B     B     B
 3 | S,A,C S,A,C S,A,C S,A,C
 4 |   B     B     B
 5 | S,A,C S,A,C
 6 |   B
```

Here we conclude that 000000 is *not* derivable from the start symbol. Since S is not in the sixth row, first column of the table. It's not so hard to see that all strings of zeros of odd length >= 3 are derivable from this grammar.

b. An NPDA for the grammar above.



6. Turing Machine Basics

a. (text 3.1a) Run M_2 on "0"

$q_1$0-
-q_2-
--q_{accept}

b. (text 3.1c) Run M_2 on "000"

$q_1$000-
-$q_2$00-
-x$q_3$0-
-x0q_4-
-x0-q_{reject}

c. (text 3.2a) Run M_1 on "11"

$q_1$11-
-$q_3$1-
-1q_3-
reject

d. (text 3.2d) Run M_1 on "10#11"

q_{01}10#11-	-q_{09}0#11-
-q_{03}0#11-	-0q_{09}#11-
-0q_{03}#11-	-0#q_{11}11-
-0#q_{05}11-	-0q_{12}#x1-
-0#1q_{05}1-	-q_{12}0#x1-
-0#11q_{05}-	q_{12}-0#x1-
-0#1q_{07}1-	-q_{13}0#x1-
-0#q_{07}11-	-xq_{08}#x1-
-0q_{07}#11-	-x#q_{10}x1-
-q_{07}0#11-	-x#xq_{10}1-
q_{07}-0#11-	reject

7. Turing Machine Design

a. To accept odd-valued binary strings, we only have to look at the last bit. The TM moves right until it reads a blank, moves left one space and accepts if and only if there is a 1 on the tape.

b. Simulate adding a and b, marking the digits right to left as we go and verifying the digits of c.

1. Add a $ to the beginning of the tape and shift the rest of the input.
2. Move right past the input string, write another # and a 0. This is the carry bit. Move left until you hit the $.
3. Move right to the first #, then move left until you find an unmarked digit.
4. Remembering this digit, move right past a #.
5. Move right to the second #, move left until you find an unmarked digit.
6. Remembering this second digit, move right past the second #.
7. Move right past the last #.
8. If the two bits read, plus the carry bit, are equal 2 or 3, write a one to the carry bit.
9. Move left past the # and stop at the first unmarked bit. If the the remembered bits plus the old carry bit are equal to 1 or 3, check for a 1 under the head, and mark it. Otherwise, check for a zero, and mark it. Reject if the check fails.
10. Move left to the $, and repeat from 3.
11. When there are no unmarked digits in a and b, move to the carry

bit.
12. If the carry bit is 1, check for an unmarked 1 in c. Accept as long as there are no unmarked 1s in c.

c. This solution is a bit non-intuitive, because it does not construct the odds by adding one each time, but by constructing all possible odd-valued n-bit strings by prepending 0 and 1 to all odd-valued n-1-bit strings. The resulting machine, though, is considerably simpler.
   1. Write $*1* to the tape
   2. Move to the left pass a * and until you hit another *.
   3. Moving right, copy symbols to the end of the tape, replacing the first * with *0, and #s with #0. Stop when you encounter a second *. Don't copy it.
   4. Move left back past two stars and stop on the third.
   5. Copy the string between the stars again to the end of the tape, this time prefixing each bit with a 1. Also, this time write a # instead of a leading *. Stop when you get to the second *, but copy it to the end of the tape.
   6. From the end of the tape, repeat back to step 2.

8. Turing Decidability and Recognition

Show that the set of decidable languages is closed under...

a. (text 3.14a) UNION.

Create a TM M from the two input machines M_1 and M_2 where:

M = "On input w:

1. Run M_1. If it accepts, accept.
2. Run M_2. If it accepts, accept.
3. Otherwise, reject."

Since M_1 and M_2 are decidable, we are guaranteed that they will accept or reject in finite time (ie they will not loop forever). So, TM M will be decidable.

b. (text 3.14d) COMPLEMENTATION.

We again create a TM M from the input machine M_1 where:

M = "On input w:

1. Run M_1. If it accepts, reject.
2. Otherwise, accept."

Show that the set of recognizable languages is closed under...

c. (text 3.15a) UNION.

Create a TM M from the two input machines M_1 and M_2 where:

M = "On input w:

1. Run M_1 and M_2 in parallel. If either accepts, accept.
2. Otherwise, reject."

Since M_1 and M_2 might not ever reject, we must run them in parallel to guarantee that if either of them accepts, we can accept. Note that step 2 might never be reached, but that is okay, since if we only recognize the language, we do not have to reject.

d. (text 3.15c) STAR.

Create a TM M from the input machine M_1 where:

M = "On input w: In parallel:

1. Enumerate all the strings that M_1 accepts onto a list L1
2. Create a list L2 which contains the * of all the items in L1. If w appears on L2, accept."

We know that we can enumerate all the strings that M_1 accepts to create L1, but creating the * of that list is much trickier. We must choose an ordering that reaches any string in finite time. One possible ordering would be to include the strings made by repeating the first n strings of L1 up to n times letting n go from 0 to infinity. Since all strings in the *'d language will eventually appear on L2, we can accept w if it is in the language.