CS 341 Homework 22
Review

1. Given the following language categories:

A: L is finite.
B: L is not finite but is regular.
C: L is not regular but is deterministic context free
D: L is not deterministic context free but is context free
E: L is not context free but is Turing decidable
F: L is not Turing decidable but is Turing acceptable
G: L is not Turing acceptable

Assign the appropriate category to each of the following languages. Make sure you can justify your answer.

a. _____ $\{a^n b^{kn} : k = 1 \text{ or } k = 2, n \geq 0\}$
b. _____ $\{a^n b^{kn} : k = 0 \text{ or } k = 1, n \geq 0\}$
c. _____ $\{a^n b^n c^n : n \geq 0\}$
d. _____ $\{a^n b^n c^m : n \geq 0, m \geq 0\}$
e. _____ $\{a^n b^n : n \geq 0\} \cup a^*$
f. _____ $\{a^n b^m :$ n is prime and m is even$\}$
g. _____ $\{a^n b^m c^{m+n} : n \geq 0, m \geq 0\}$
h. _____ $\{a^n b^m c^{mn} : n \geq 0, m \geq 0\}$
i. _____ $\{a^n b^m : n \geq 0, m \geq 0\}$
j. _____ $\{xy : x \in a^*, y \in b^*, |x| = |y|\}$
k. _____ $\{xy : x \in a^*, y \in a^*, |x| = |y|\}$
l. _____ $\{x :$ x $\in \{a, b, c\}^*,$ and x has 5 or more a's$\}$
m._____ {"M" : M accepts at least 1 string}
n._____ {"M" : M is a Turing machine that halts on input ε and |"M"| ≤ 1000}
o._____ {"M" : M is a Turing machine with ≤ 50 states}
p._____ {"M" : M is a Turing machine such that L(M) = a*}
q. _____ $\{x :$ x $\in \{A, B, C, D, E, F, G\},$ and x is the answer you write to this question$\}$

Solutions

a. __D__ $\{a^n b^{kn} : k = 1 \text{ or } k = 2, n \geq 0\}$
We haven't discussed many techniques for proving that a context free language isn't deterministic, so we can't prove that this one isn't. But essentially the reason this one isn't is that we don't know what to do when we see b's. Clearly, we can build a pda M to accept this language. As M reads each a, it pushes it onto the stack. When it starts seeing b's, it needs to start popping a's. But there's no way to know, until either it runs out of b's or it gets to the $n+1^{st}$ b, whether to pop an a for each b or hold back and pop an a for every other b. So M is not deterministic.
b. __C__ $\{a^n b^{kn} : k = 0 \text{ or } k = 1, n \geq 0\}$
This one is looks very similar to **a**, but it's different in one key way. Remember that the definition of deterministic context free is that it is possible to build a deterministic pda to accept L$. So now, we can build a deterministic pda M as follows: Push each a onto the stack. When we run out of a's, the next character will either be $ (in the case where k = 0) or b (in the case where k = 1). So we know right away which case we're dealing with. If M sees a b, it goes to a state where it pops one b for each a and accepts if it comes out even. If it sees $, it goes to a state where it clears the stack and accepts.
c. __E__ $\{a^n b^n c^n : n \geq 0\}$
We proved that this is recursive by showing a grammar for it in Lecture Notes 24. We used the pumping theorem to prove that it isn't context free in Lecture Notes 19.

d. __C__ $\{a^n b^n c^m : n \geq 0, m \geq 0\}$

This one is context free. We need to compare the a's to the b's, but the c's are independent. So a grammar to generate this one is:

$S \rightarrow A\ C$
$A \rightarrow a\ A\ b$
$A \rightarrow \varepsilon$
$C \rightarrow c\ C$
$C \rightarrow \varepsilon$

It's deterministic because we can build a pda that always knows what to do: push a's, pop an a for each b, then simply scan the c's.

e. __C__ $\{a^n b^n : n \geq 0\} \cup a^*$

This one is equivalent to **b**, since $a^* = a^n b^{0n}$.

f. __E__ $\{a^n b^m : n$ is prime and m is even$\}$

This one is recursive because we can write an algorithm to determine whether a number is prime and another one to determine whether a number is even. The proof that it is essentially the same as the one we did in class that a^n: n is prime is not context free.

g. __C__ $\{a^n b^m c^{m+n} : n \geq 0, m \geq 0\}$

This one is context free. A grammar for it is:

$S \rightarrow a\ S\ c$
$S \rightarrow b\ S\ c$
$S \rightarrow \varepsilon$

It's deterministic because we can build a deterministic pda M for it: M pushes each a onto its stack. It also pushes an a for each b. Then, when it starts seeing c's, it pops one a for each c. If it runs out of a's and c's at the same time, it accepts.

h. __E__ $\{a^n b^m c^{mn} : n \geq 0, m \geq 0\}$

This one is similar to **g**, but because the number of c's is equal to the product of n and m, rather than the sum, there is no way to know how many c's to generate until we know both how many a's there are and how many b's. Clearly we can write an algorithm to do it, so it's recursive. To prove this , we need to use the pumping theorem. Let $w = a^M b^M c^{MM}$. Call the a's, region 1, the b's region 2, and the c's region 3. Clearly neither v nor y can span regions since, if they did, we'd get a string with letters out of order. So we need only consider the following possibilities:

(1, 1) The number of c's will no longer be the product of n and m.
(1, 2) The number of c's will no longer be the product of n and m.
(1, 3) Ruled out by $|vxy| \leq M$.
(2, 2) The number of c's will no longer be the product of n and m.
(2, 3) The number of c's will no longer be the product of n and m.
(3, 3) The number of c's will no longer be the product of n and m.

i. __B__ $\{a^n b^m : n \geq 0, m \geq 0\}$

This one is regular. It is defined by the regular expression a*b*. It isn't finite, which we know from the presence of Kleene star in the regular expression.

j. __C__ $\{xy : x \in a^*, y \in b^*, |x| = |y|\}$

This one is equivalent to $a^n b^n$, which we've already shown is context free and not regular. W showed a deterministic pda to accept it in Lecture Notes 14.

k. __B__ $\{xy : x \in a^*, y \in a^*, |x| = |y|\}$

This one is $\{w = a^*: |w|$ is even$\}$. We've shown a simple two state FSM for this one.

l. __B__ $\{x : x \in \{a, b, c\}^*,$ and x has 5 or more a's$\}$

This one also has a simple FSM F that accepts it. F has six states. It simply counts a's, up to five. If it ever gets to 5, it accepts.

m. __F__ {"M" : M accepts at least 1 string}

This one isn't recursive. We know from Rice's Theorem that it can't be, since another way to say this is

{"M" : L(M) contains at least 1 string}

We can also show that this one isn't recursive by reduction, which is done in the Supplementary Materials.

n. __A__ {"M" : M is a Turing machine that halts on input ε and |"M"| ≤ 1000}

This one is finite because of the limit on the length of the strings that can be used to describe M. So it's finite (and thus regular) completely independently of the requirement that M must halt on ε. You may wonder whether we can actually build a finite state machine F to accept this language. What we know for sure is that F exists. It must for any finite language. Whether we can build it or not is a separate question. The undecidability of the halting problem tells us that we can't build an algorithm to determine whether an arbitrary TM M halts on ε. But that doesn't mean that we can't look at most Turing Machines and tell. So, here, it is likely that we could write out all the TMs of length less than 1000 and figure out which ones accept ε. We could then build a deciding FSM F. But even if we can't, that doesn't mean that no such FSM exists. It just means that we don't know what it is. This is no different from the problem of building an FSM to accept all strings of the form mm/dd/yy, such that mm/dd/yy is your birthday. A simple machine F to do this exists. You know how to write it. I don't because I don't know when your birthday is. But that fact that I don't know how to build F says nothing about its existence.

o. __E__ {"M" : M is a Turing machine with ≤ 50 states}

This one looks somewhat similar to **n**. But it's different in a key way. This set isn't finite because there is no limit on the number of tape symbols that M can use. So we can't do the same trick we can do in **n**, where we could simply list all the machines that met the length restriction. With even a single state, I can build a TM whose description is arbitrarily long. I simply tell it what to do in state one if it's reading character 1. Then what to do if it's reading character 2. Then character 3, and so forth. There's no limit to the number of characters, so there's no limit to the length of the string I must write to consider all of them. Given that the language is not finite, we need a TM to decide it. Why? What we need to do is to check to make sure that the string is a syntactically valid encoding of a Turing Machine. Recall the syntax of an encoding. When we see the first a??? symbol that encodes a tape symbol, we know how many digits it has. All the others must have the same number of digits. So we have to remember that number. Since there's no limit to it, we can't remember it in a finite number of states. Since we need to keep referring to it, we can't remember it on a stack. So we need a TM. But the TM is a straightforward program that will always halt. Thus the language is recursive.

p. __G__ {"M" : M is a Turing machine such that L(M) = a*}

This one isn't recursive. Again, we know that from Rice's Theorem. And we can prove it by reduction, which we did in the supplementary materials for the more general case of any alphabet Σ. But this language is even harder than many we have considered, such as H. It isn't even recursively enumerable. Why? Informally, the TM languages that are re are the ones where we can discover positive instances by simulation (like, H, where we ask whether M halts on a particular w?). But how can we try all strings in a*? Proving this formally is beyond the scope of this class.

q. __A__ {x : x ∈ {A, B, C, D, E, F, G}, and x is the answer you write to this question}

This one is finite. In fact, it is a language of cardinality 1. Thus it's regular and there exists an FSM F that accepts it. You may feel that there's some sort of circularity here. There really isn't, but even if there were, we can use the same argument here that we used in **n**. Even if we didn't know how to build F, we still know that it exists.