

Automate deterministe si nedeterministe

Problem 1

(part a) Design a DFA or NFA (with or without ϵ -transitions) that accepts all strings over the alphabet $\{\$,c,0,1,2,3,4,5,6,7,8,9,.\}$ that correspond to valid currency amounts. A valid string is either a dollar sign followed by a number which has no leading 0's, and may have a decimal point in which case it must be followed by exactly two decimal digits, OR a one or two-digit amount followed by the cent sign c . The single exception to this rule is that strings which begin with "\$0." and are followed by exactly two digits are also acceptable. Thus, \$432.63, \$1, \$0.29, 47c, 2c are all accepted, but \$021, \$4.3, \$8.63c, \$0.0 are not accepted.

Solution

This DFA (reject state, transitions not shown) has two sections, corresponding to two types of currency strings: dollars and cents. The states q_1 - q_3 represent the cents, and q_4 - q_9 represent the dollars.

The cents representation of currency is a one or two-digit number followed by "c". The number is either a positive one-digit number (which takes the machine to q_2), 0 (which takes the machine to q_1), or a positive two-digit number where the first digit is not 0 (which takes the machine to q_1 through q_2). Note that the last two cases can lead to the same state since both are only valid if followed immediately by "c". This machine accepts a string at q_3 after a single "c" following this number.

The dollar representation of currency is either (1) a whole amount of dollars (e.g. \$5, \$221), or (2) dollars plus change (e.g. \$3.21, \$19.95). Both of these must be preceded by a dollar sign, hence the transition to q_4 . Here q_5 represents a positive number (no leading 0s) and q_6 represents 0. These states are accept states, to satisfy case (1). Case (2) requires representation for cents. States q_8 , and q_9 process the cents, and q_7 processes the decimal point. Notice q_8 is not an accept state, but q_9 is, since there must be 2 digits in the cents value. There are transitions from q_5 and q_6 to q_7 to connect the dollars with the cents, and thus satisfy case (2).

(part b) Write a regular expression for the set of valid currency amounts.

Solution

The regular expression is $([1-9]|\epsilon)[0-9]c \mid \$ (0|[1-9][0-9]^*)(\epsilon|. [0-9][0-9])$ where $[0-9] = (0|1|2|3|4|5|6|7|8|9)$ and $[1-9] = (1|2|3|4|5|6|7|8|9)$

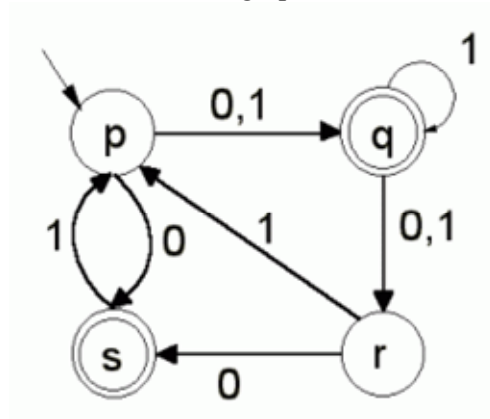
Explanation: Break the expression down into the accept states.

- q_4 is the cents representation, and represents $(0|[1-9][0-9])c = ([0-9][1-9][0-9])c = ([1-9]|\epsilon)[0-9]c$
- q_5 represents $\$[1-9][0-9]^*$, and q_6 represents $\$0$, so $q_5|q_6 = \$ (0|[1-9][0-9]^*)$
- q_9 is q_5 or q_6 , followed by $. [0-9][0-9]$, so $q_9 = \$ (0|[1-9][0-9]^*) . [0-9][0-9]$
- $q_5|q_6|q_9 = (q_5|q_6)(q_5|q_6) . [0-9][0-9]$, and so can be written as $(q_5|q_6)(\epsilon|. [0-9][0-9])$

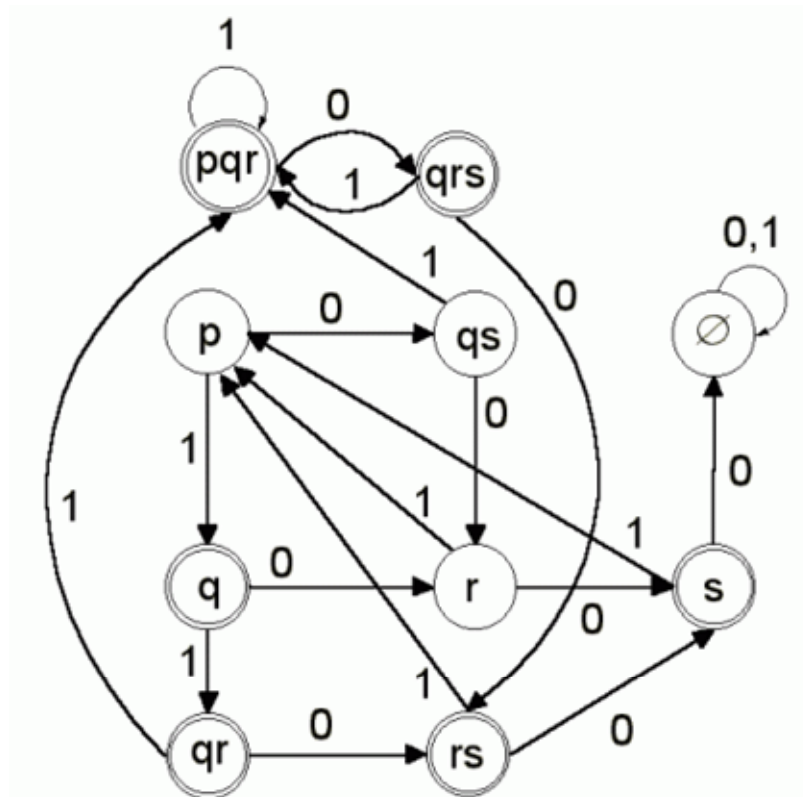
Therefore, the final expression is $q_4|(q_5|q_6|q_9) = ([1-9]|\epsilon)[0-9]c \mid \$ (0|[1-9][0-9]^*)(\epsilon|. [0-9][0-9])$

Problem 2

Here is the NFA in graphical form:



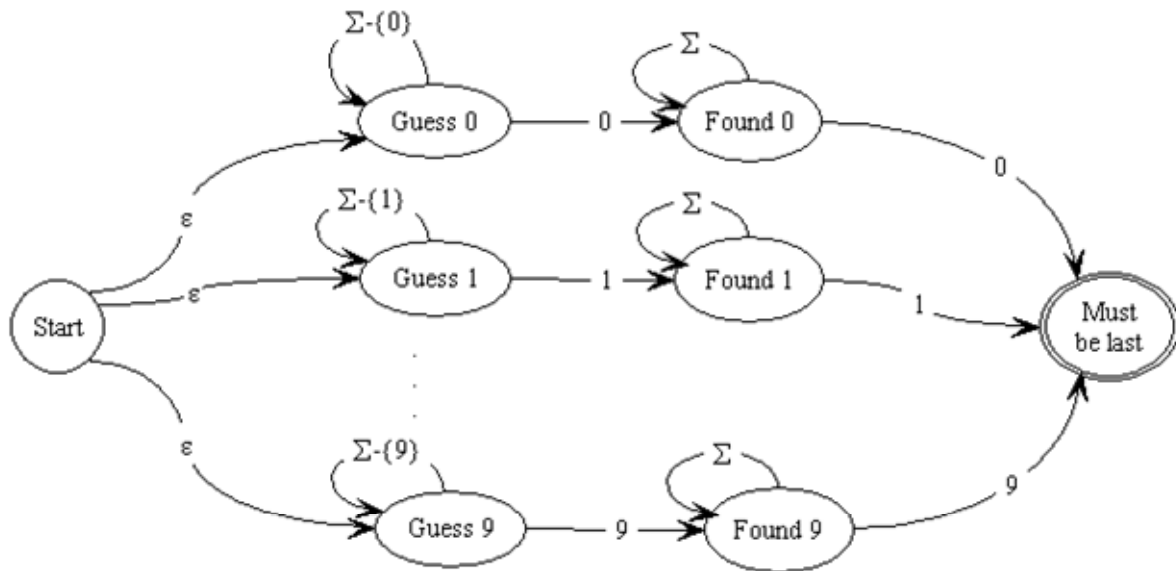
Solution



Notice that $\{p,q\}$, $\{p,r\}$, $\{p,s\}$, $\{p,r,s\}$, $\{p,q,s\}$, and $\{p,q,r,s\}$ are not reachable from the start state, and are thus not included in the DFA.

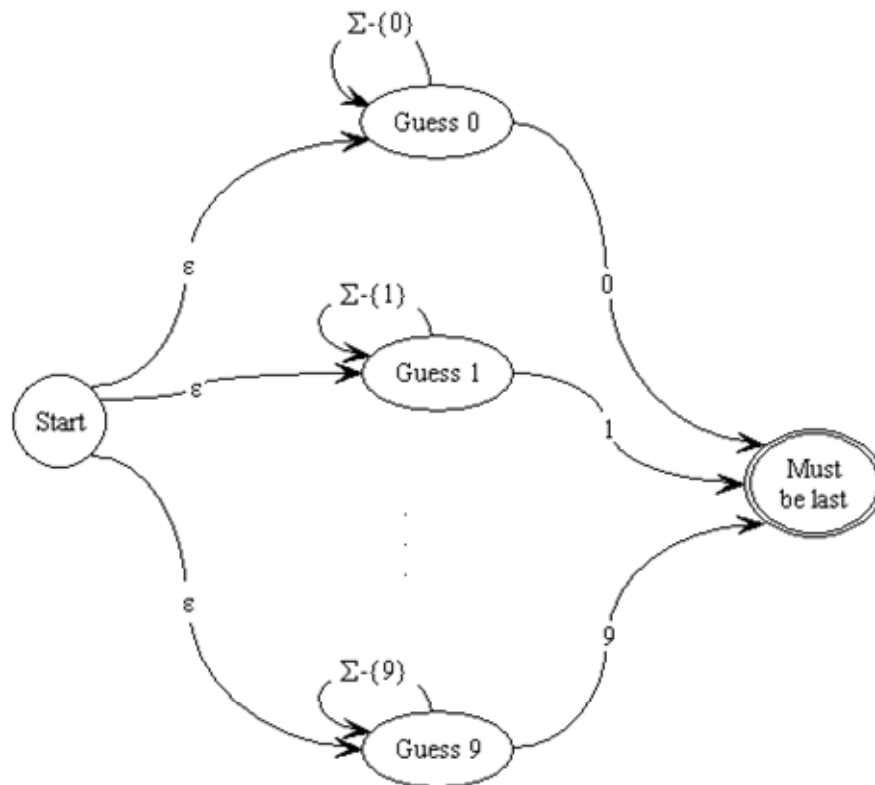
Problem 3

Part a) Final digit appeared before



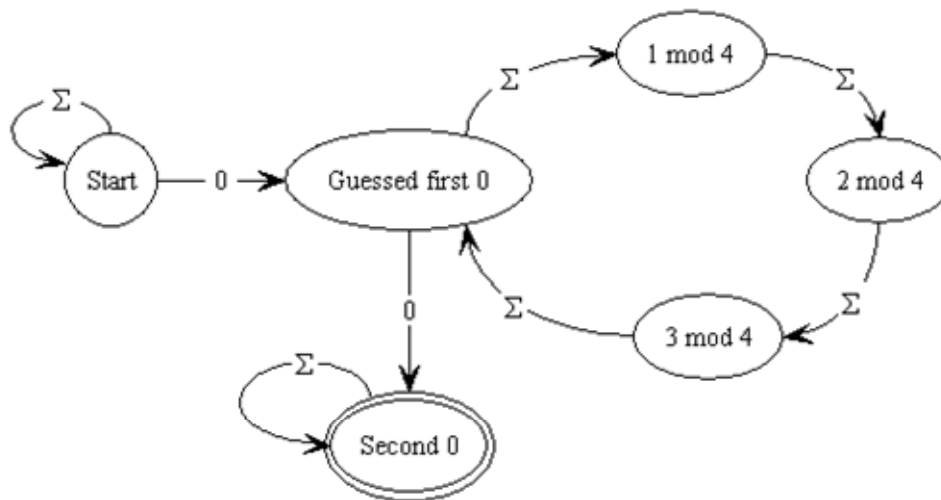
We first guess which digit is going to be the last one, and we do that by having epsilon transitions to the ten "guess" states above. Then we look for an occurrence of the digit we guessed in the middle of the string. When we find one, we go to the respective "found" state. After we have found the proposed last digit in the middle of the string, we must confirm that our guess is correct. Each "found" state has a loop around itself for every character so that we can read the rest of the string. When we run into the digit we guessed, it just might be the last digit, so we have an outgoing arc labeled with that digit going to the "Must be last" state. Note that when we run into the character we guessed, we can either follow the Σ arc or the outgoing arc. This accounts for the fact that any occurrence of the guessed digit could come either in the middle of the string or at the end. At the "Must be last" state, we will exit if we read any more characters. If we don't find any more characters, we have confirmed our guess and hence "Must be last" is the accepting state.

Part b) Final digit is unique



Like the answer for part b), we guess the last digit, and go to ten "guess" states. Here, if we run into the digit we guessed, it must be the last one, and hence we go to the "Must be last" state. If it was the last, then we accept.

Part c) There are a pair of 0's that are spaced by a multiple of 4 characters



We cycle at the start state for every character, until we guess a certain "0" to be the first of the pair that is separated by $0 \bmod 4$ characters. We have a loop of interconnected states that consecutively count the character that come in between, and keep track of the number modulo 4. When we are at $0 \bmod 4$, which is the "Guessed first 0" state, and we encounter a "0", we can consider it the second "0" of the pair and go to the accepting state. Now that we have found a pair that is separated by $0 \bmod 4$ characters, no matter what we see thereafter, we stay in the accept state.

Problem 4

Write regular expressions for each of the following languages over the alphabet $\{a,b\}$.

(part a) The set of all strings with at most one pair of consecutive a's and at most one pair of consecutive b's. Argue that your expression is correct.

SOLUTION

First, think of the expression for strings that have no consecutive a's or b's. It should basically be an alternation of the two characters, where the beginning and end might either be an "a" or a "b":

$$(a+\epsilon)(ba)^*(b+\epsilon)$$

Now we must introduce the possibility of having at most one "aa" and at most one "bb" into the expression. We can divide the possibilities into two cases:

1. "bb" comes before "aa" : The "bb" can be thought of as a renegade "b" coming inside a sequence of "baba"s. The "aa" can be thought of likewise. The only case this conceptualization breaks down is when the string itself is either "bb" or "aa", which we will check later. Then, the string can be seen as a sequence of alternating a's and b's followed by a possible renegade "b", followed by a sequence of alternating a's and b's followed by a possible renegade "a", and then a trailing sequence of alternating a's and b's:

$$(a+\epsilon)(ba)^* (b+\epsilon)(ba)^*(a+\epsilon) (ba)^*(b+\epsilon)$$

Now, we can check that the expression above also includes the string "bb", since we can let both "(b+ ϵ)" parts to be "b" and all the rest to be empty. Likewise, "aa" is included in the expression.

2. "aa" comes before "bb" : We use the same exact reasoning, and just switch around the renegade a's and b's to get:

$$(a+\epsilon)(ba)^* (a+\epsilon)(ba)^*(b+\epsilon) (ba)^*(b+\epsilon)$$

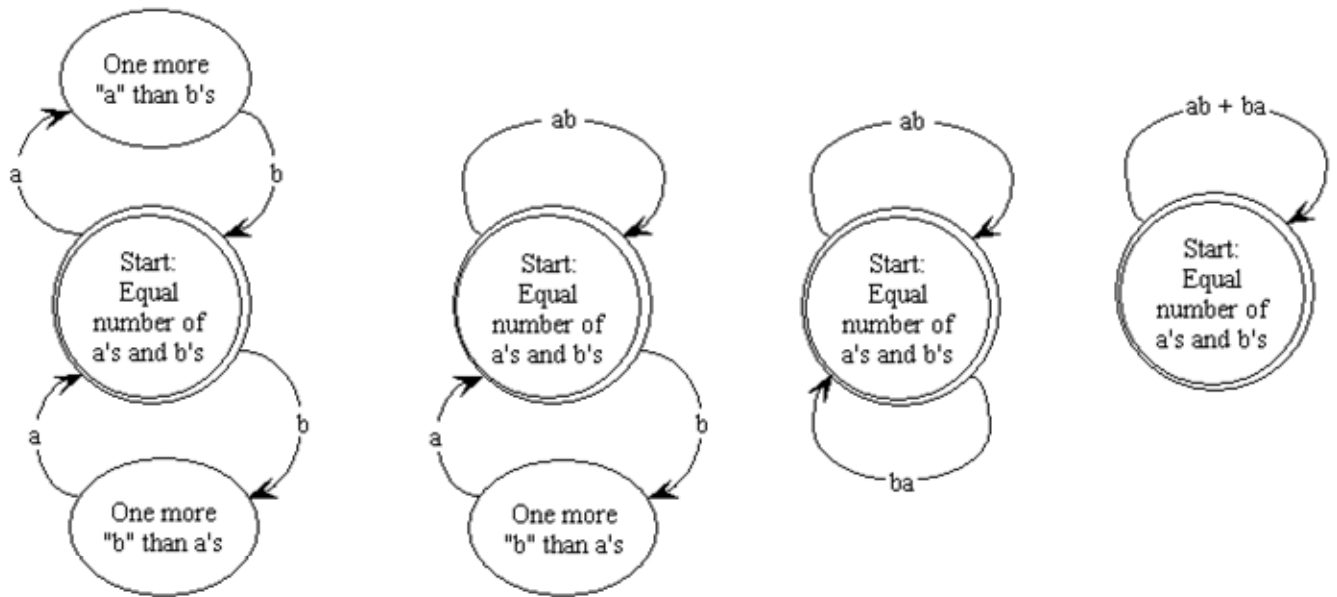
Again, we can check that "aa" and "bb" are included in the expression by the same reasoning as above.

We need to OR the two expressions above to get the answer. Notice that the outer parts of the expressions are the same, and hence we can factor them out:

$$(a+\epsilon)(ba)^* (((b+\epsilon)(ba)^*(a+\epsilon)) + (a+\epsilon)(ba)^*(b+\epsilon)) (ba)^*(b+\epsilon)$$

(part b) The set of all strings with an equal number of a's and b's such that no prefix has two more a's than b's nor two more b's than a's. For this part, first construct a DFA and then use the state-elimination method. You need not argue that the expression is correct, since you are (presumably) following an algorithm known to be correct. (But do briefly justify that your DFA accepts all and only the desired strings.)

SOLUTION



There is a hidden trap state for each of the DFA's above, which is the destination for all of the undrawn arcs. In the above, the left-most machine correctly recognizes the desired language, since it keeps track of the relative number of a's and b's. The three states are enough, since the imbalance never goes over one in a valid string, because the the problem statement says that it is so in any prefix. The state elimination proceeds from left to right, and the final result is $(ab+ba)^*$.

Lema de pompare & stuff

Problem 1:

a) Prove $L = \{0^n 1^m \mid n < m\}$ is not regular

Solution

Let $w = 0^p 1^{p+1} \in L$, where p is the pumping length. Since $|xy| \leq p$ and $|y| > 0$, $y = 0^k$, where $1 \leq k \leq p$. Therefore, $xy^2z = 0^{p+k} 1^{p+1}$. But since $k \geq 1$, $p+k \geq p+1$, so $xy^2z \notin L$. Therefore, L is not regular.

b) Prove $L = \{0^n 1^m \mid n > m\}$ is not regular

Solution

Let $w = 0^p 1^{p-1} \in L$, where p is the pumping length. Since $|xy| \leq p$ and $|y| > 0$, $y = 0^k$, where $1 \leq k \leq p$. Therefore, $xy^0z = xz = 0^{p-k} 1^{p-1}$. But since $k \geq 1$, $p-k \leq p-1$, so $xy^0z \notin L$. Therefore, L is not regular.

Problem 2

Prove that the following languages are also not regular.

1. $\{0^n : n \text{ is a perfect cube (i.e. } n = k^3 \text{ for some positive integer } k)\}$

Solution

We show that the pumping lemma does not hold for this language. To use proof by contradiction, assume that it holds. Then, there is an n such that for any string of length longer than n , its first n characters has a portion that can be "pumped". In particular, if repeat that portion just once, we get a string that is at most n characters longer. Let's examine $x = 0^{n^3}$, the string that has n^3 0's. The next longer string in the language has to have length equal to the next perfect square, which is $(n+1)^3$. Hence the difference in length between x and the next longer string is $(n+1)^3 - n^3 = 3n^2 + 3n + 1$. However, the string we get when we only pump the string once is at most n characters longer, and hence cannot be in the language.

2. $\{0^i 1^j : i \text{ and } j \text{ have a common factor greater than } 1\}$

Solution

Assume that the pumping lemma holds for strings longer than n . Pick a prime number p that is larger than n . Consider the string $0^p 1^p$. The pumpable portion must reside in the first n characters, and hence should consist entirely of 0's. Since the length of the repeatable portion is from 1 to n , if we pump this once, the length of the initial string of 0's will have length $p+1$ to $p+n$. Since p is a prime larger than n , none of these lengths will be able to have a common factor with p . Hence the string with the pumping portion pumped once will not be in the language, and this is a contradiction.

Problem 3

Prove that for any irrational number r such that $0 < r < 1$, the language consisting of all prefixes of the binary representation of r is not regular.

Solution

We observe that for any DFA that recognizes all of the prefixes of a fixed string, all of the states that can reach an accepting state will also be accepting. Suppose that in the middle of reading an input string that will eventually be accepted, we are at a certain state s . Since the string is in the language, the substring we've read thus far should also be accepted. Therefore, s must also be an accepting state.

So as to induce a contradiction, assume that for some irrational number r , the language is regular. Then, there is a DFA for the language with, say, n states. Since the binary representation of r has no end to it, we could easily find a prefix of it that is longer than n . Following the proof technique of the Pumping lemma for regular languages (there are other pumping lemmas for other types of languages), we see that the DFA has to have a loop in it, since the path that the prefix follows is longer than the number of states available. From the argument in the previous paragraph, all of the states in that loop are accepting. Therefore, running through the loop any number of times produces a string that is a prefix of r . Hence r must have an infinitely repeating segment at its end. This contradicts the fact that r is irrational.

Problem 4:

Show if L is regular, so is L/a

Solution

Generate a DFA $D=(Q,\Sigma,\delta,q_0,F)$ for L . Construct a new DFA $D'=(Q,\Sigma,\delta,q'_0,F)$ from it where $q'_0 = \delta(q_0,a)$. In other words, move the start state to where D would be after input " a ", and leave everything else the same. This works because after one transition on input aw , D is in q'_0 with input w . This is the same as D' before any transitions. The other DFA details are all the same, so the future transitions will be the same for both machines. Therefore D' is on the same state after n characters of w as D is after $n+1$ characters of aw . Therefore $aw \in L \iff w \in L/a$. Therefore D' recognizes L/a , so L/a is regular.

Problem 5:

If $w=a_1...a_n$ and $x=b_1...b_m$ are strings of the same length, define $alt(w,x)$ to be the string in which the symbols of w and x alternate, starting with w (that is, $a_1b_1a_2b_2...a_nb_n$). If L and M are languages, define $alt(L,M)$ to be the set of strings of the form $alt(w,x)$, where w is any string in L , x is any string in M , and w and x have the same length. Prove that if L and M are regular, then so is $alt(L,M)$.

Solution

Construct DFAs $D_1=(Q_1,\Sigma_1,\delta_1,q_{10},F_1)$ that recognizes L and $D_2=(Q_2,\Sigma_2,\delta_2,q_{20},F_2)$ that recognizes M . Then construct DFA $D_3=(Q_1 \times Q_2 \times \{0,1\}, \Sigma_1 \sqcup \Sigma_2, \delta', (q_{10}, q_{20}, 0), F_1 \times F_2 \times \{0\})$ to recognize $alt(L,M)$. Each state represents the state in D_1 after reading just the odd characters, the state in D_2 after reading just the even characters, and a parity value indicating whether the character just read is odd or even. The start state must be even, as the first character read is odd. Similarly the final states -- which correspond to states that are final in both D_1 and D_2 -- must be even, as all strings in $alt(L,M)$ are even. The transition function is defined as:

- $\delta'((q_a, q_b, 0), c) = (\delta_1(q_a, c), q_b, 1)$
- $\delta'((q_a, q_b, 1), c) = (q_a, \delta_2(q_b, c), 0)$

This transition function uses δ_1 on the characters and δ_2 on the even characters. Thus D_3 alternates between D_1 and D_2 , updating the state to one machine while keeping track of the other for next time. By looking at every other transition, it becomes clear that D_3 accepts w iff D_1 accepts the odd characters, D_2 accepts the even characters, and w has even length. Therefore D_3 recognizes $alt(L,M)$, so $alt(L,M)$ is regular.

Problem G6

As in problem 3, if L is a language, and a is a symbol, then define $a \backslash L$ to be the set of strings w such that aw is in L . Let $\mathbf{C}$ be a collection of languages over S^* for some finite alphabet S . Suppose that $\mathbf{C}$ has the following properties:

1. There are a finite number of languages in $\mathbf{C}$
2. For any language L in $\mathbf{C}$ and for any symbol a in S , the language $a \backslash L$ is also in $\mathbf{C}$.

Prove that every language in $\mathbf{C}$ is regular.

Solution

If we think about what the $a \backslash L$ does to L intuitively, it effectively reduces the length of each string in L that starts with an a by 1. Regardless of how many times we repeat this process, we still have to be within the class $\mathbf{C}$. Let's assume that we have some language L in $\mathbf{C}$ that has an infinite number of strings. Then there has to be no bound to the lengths of the strings in L , since if there were, L would have to be finite (if the bound was, say, l , and the size of the alphabet was c , then there could only be at most c^l strings). Since the lengths are unbounded, no matter how many times we apply " $\backslash$ " to L , there will still be a string of non-zero length left in the modified language. Hence, an infinite number of languages will have to be in $\mathbf{C}$, which is a contradiction. Therefore, the initial assumption that there was a language of infinite size in $\mathbf{C}$ was incorrect. We have thus shown that every language in $\mathbf{C}$ is finite, and hence is regular.

Problem G7

part a. Prove an extended, more powerful version of the pumping lemma:

Let L be a regular language. Then there exists a constant n for L satisfying the following: Given any string $z_1 z_2 z_3$ in L such that $|z_2| = n$, z_2 can be written as $z_2 = uvw$ so that v is not empty and for all non-negative integers i (including 0), $z_1 u v^i w z_3$ is in L .

Solution

Since we are to prove that a constant n exists, we may set it to whatever we find convenient. Let's set n to the number of states in the DFA that recognizes L . Assume that we are given a string $z_1 z_2 z_3$ in L such that $|z_2| = n$, since this was the "if" part of what we have to prove. Consider the path the DFA will have to go through when reading z_2 . The path will include $n+1$ states, since there are n edges that are traversed by z_2 . Since we only have n states, there must be a state that comes up twice in the path. The rest of the proof is the same as the original pumping lemma.

part b. Any regular language must satisfy the pumping lemma, but the converse does not necessarily hold. There may be languages that satisfy the pumping lemma but are not regular. Clearly, the extended pumping lemma above implies the original pumping lemma, and more. Construct a language that satisfies the extended pumping lemma and yet is not regular.

Solution

The idea is to start with a basic non-regular language $L_{\text{base}} = \{a^n b^n : n \geq 0\}$, and to modify it to satisfy the extended pumping lemma (let's call it the EPL from here on) and still be non-regular. Note that L cannot satisfy the EPL because pumping any portion of the string of a 's will cause the number of a 's and b 's to become unequal. Hence, we put in portions in regular intervals into the strings that actually can be repeated. Intuitively, we want to get something that looks like:

$$a \ c^+ \ a \ c^+ \ a \ c^+ \ \dots \ a \ c^+ \ b \ c^+ \ b \ c^+ \ b \ c^+ \ \dots \ b \ c^+ \\ \text{(number of } a\text{'s and } b\text{'s are equal)}$$

This way, if we set the n in the EPL to be 3, any substring of 3 characters would have to include a "c+" portion in it, and this will be able to be repeated. Let's call the language that has these kinds of strings, L_1 . The only trouble L_1 has with the EPL is when the "c+" portion is a single "c", and deleting it would get us two consecutive a's or b's, which is not allowed in the regular expression above. We will take care of this later by unioning L_1 with another language, say L_2 , that also satisfies the EPL and includes these exceptions.

To formally write out L_1 , first note that we could write L_{base} as:

$$L_{\text{base}} = \bigcup_{k=0}^{\infty} (a)^k (b)^k$$

Now we can add the "c+"s as follows:

$$L_1 = \bigcup_{k=0}^{\infty} (ac+)^k (bc+)^k$$

Now we have to make up L_2 , which will include the cases when two a's or b's are adjacent to each other, and will also satisfy the EPL. To satisfy the EPL, the easiest thing to do would be to make L_2 regular. But recall that unioning a non-regular language with a regular language could result in a regular language (e.g. $0^n 1^n$ unioned with $\{0,1\}^*$). Intuitively, this is because the regular language we are unioning could "cover up" the non-regular language in such a way that it eliminates its complexity. To avoid this "cover up", we make L_2 disjoint with L_1 .

To check that this would work, say L_2 is disjoint from L_1 . Assume to the contrary that $L_1 \cup L_2$ is regular. Since the two sets are disjoint, $L_1 = (L_1 \cup L_2) - L_2$. Since regular languages are closed under set difference, L_1 has to be regular and this is a contradiction.

So, we set $L_2 = (a+b+c)^* (a+b)^2 (a+b+c)^*$. Since no string in L_1 has $(a+b)^2$ as a substring, L_1 and L_2 are disjoint. Setting $L = L_1 \cup L_2$ gives us the desired language.

Problem G8

For any language L that is a subset of 0^ , show that L^* is regular.*

Solution

The basic idea is that if there are two strings of relatively prime lengths in L , then any long enough string can be produced by multiplying and combining these strings. Therefore any string beyond a certain length is in L^* , and so L^* must be regular.

Consider the two shortest strings in the language are 0^i and 0^j such that i and j are relatively prime (if such i and j cannot be found, see below).

- **Lemma:** $\forall n \geq (i-1)(j-1) \exists a, b \in \mathbb{N}$ such that $ai + bj = n$, and therefore $0^n = (0^i)^a (0^j)^b \in L^*$.
- **Proof:** It is already known that $ai + bj = n$ has solutions for $a, b \in \mathbb{Z}$ if i and j are relatively prime. They can be found by using the Euclidean Algorithm to solve $ai + bj = 1$, multiplying by n , and converting to a general solution:
 - o $a = a_0 + jt$
 - o $b = b_0 - it$

where $a_0 i + b_0 j = n$. This general solution works because $(a_0 + jt)i + (b_0 - it)j = a_0 i + b_0 j + ijt - ijt = a_0 i + b_0 j = n$. The question then becomes whether there exists a solution for nonnegative

a,b. To find such solutions, let t be any integer such that $0 \leq b_0 - it \leq i-1$. Then $ai = n - bj = n - (b_0 - it)j \geq n - ij + j$. If $n \geq (i-1)(j-1) = ij - i - j + 1$, $n - ij + j \geq 1 - i$. Therefore $ai \geq 1 - i$, so $a \geq -1 + 1/i \geq 0$ (since $a \in \mathbb{Z}$). Therefore if $n \geq (i-1)(j-1)$, there exist positive solutions to $ai + bj = n$.

Set $m = (i-1)(j-1)$. Clearly $\{0^t \mid t \geq m\}$ is regular; it can be represented by a regular expression containing m 0s followed by 0^* . Similarly, $L^* \cap \{0^t \mid t < m\}$ is regular, as it is finite. Therefore one can construct a regular expression for L^* by listing all strings in L^* shorter than m separated by *or* operators, followed by the expression given above for $\{0^t \mid t \geq m\}$. Therefore L^* is regular, regardless of L .

A few special cases:

- *L has one element.* Then L is regular, and so is L^* .
- *Every element in L has a common factor k .* Define homomorphism h such that $h(0) = 0^k$. The language $L' = h^{-1}(L)$ must have two relatively prime strings. Using the methods above, L'^* is regular. Therefore $L^* = h(L'^*)$ is also regular.
- *There is no common factor, but no two elements have relatively prime lengths.* This is only possible if L is finite. Therefore L is regular, and so is L^* .

Problem 1

Prove the following languages are not regular, using only standard closure properties of regular languages (Concatenation, Union, Kleene Closure, Intersection, Reverse, Homomorphism, Inverse Homomorphism, Substitution, Quotient) and the fact that $\{0^n 1^n \mid n \geq 0\}$ is not regular.

1. $\{0^n 1^{2n} \mid n \geq 0\}$

Solution

Use the inverse homomorphism of h , where $h(0)=0$ and $h(1)=11$. Each pair of 1's "shrink down" to a single 1.

2. $\{ww \mid w \in \{0,1\}^*\}$

Solution

The initial difficulty lies in the fact that w can be any string, so let's intersect this language with 10^*10^* to "force" w to start with a 1 and then 0's. We now have $\{10^n 10^n \mid n \geq 0\}$. Let's strip off the initial 1 from all strings using a quotient, to get $\{0^n 10^n \mid n \geq 0\}$. Now, we want to change all of the 0's that come after the 1 to all 1's. No kind of homomorphism can do this, since homomorphisms cannot discern between the 0's that come before the 1 and those that come after the 1. The only thing that can send the same character to "different" characters is an inverse homomorphism, but this can't discern between the 0's in the front and back either. But let's try it anyways. Since we need the 0's to become both 0's and 1's, we set $h(0)=0$ and $h(1)=0$. There's nothing that maps to 1, so let say $h(2)=1$, since we've already set $h(1)=0$. The language we get from the inverse homomorphism is $\{(0+1)^n 2(0+1)^n \mid n \geq 0\}$. Luckily, a lone 2 is in there in between the two sets of $(0+1)$'s we want to make different. So, if we intersect with 0^*21^* , we're done!

3. $\{0^a 1^b 2^c \mid a \neq b, b \neq c, a \neq c\}$

Solution

The 2^c is not welcome, so we do a homomorphism $h(0)=0$, $h(1)=1$, $h(2)=\epsilon$ to get rid of it, and get $\{0^a 1^b \mid a \neq b, b \neq c, a \neq c\} = \{0^a 1^b \mid a \neq b\}$. Take the complement of this to get $\{0^a 1^b \mid a=b\}$ and we're done.

Problem 2

For each of the following language operators, prove whether or not the resulting language is necessarily regular whenever SL is.

1. $\text{inside-out}(L) = \{v^R u^R \mid uv \in L\}$.
(Recall that for a string w , w^R denotes the reversal of w .)

Solution

In the definition of $\text{inside-out}(L)$ we observe that the reverse of uv , $(uv)^R = v^R u^R$. Hence $\text{inside-out}(L)$ is exactly L^R , and is trivially regular.

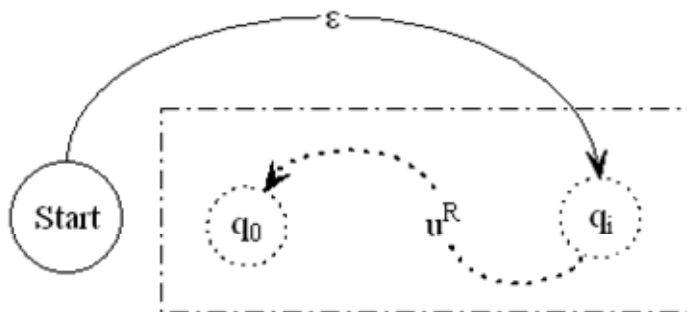
There was an announcement that $v^R u^R$ was a typo, and people who solved the problem for $u^R v^R$ would be able to skip a question on the next homework. This should be solved as follows.

We show that there is an NFA that accepts $\text{inside-out}(L)$ for any regular L . Given any DFA, a string defines a path in the machine (for example, "01" means "follow the arc labeled 0, and then follow the arc labeled 1"). Call the DFA for L , D , and consider the process of

reading uv . Assume that after reading u , D is at state q_i . The entire process of reading the string can be depicted as follows.



Let's call the NFA that we will construct to recognize $\text{inside-out}(L)$, N . N will first need to read u^R . Hence, starting from q_i it will have to run D backwards until it reaches the start state of D . In the image below, note that the singly dotted box represents D with all of its arcs reversed (the start and accepting states of D are just left as is).

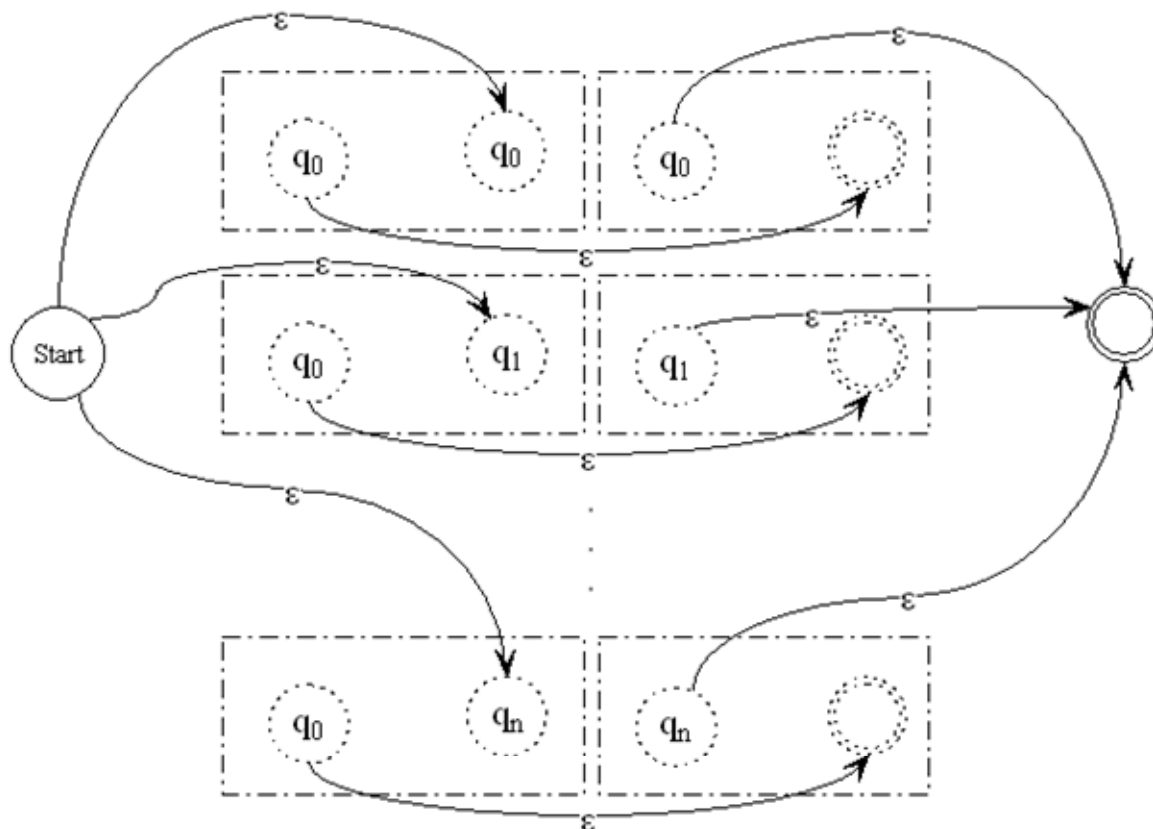


After that, N needs to read v^R . So we start from the accepting state and arrive at q_i .



Up until now, we have only been making sure that if uv is in L , and we arrive at state q_i after reading u , $u^R v^R$ would be accepted by our new machine. We have to make sure that whatever our machine accepts is in $\text{inside-out}(L)$. It is clear from the diagram that whenever the NFA accepts a string, it passes through the q_i of the left box, then the q_0 of the left box, then an accepting state of the right box, and then the q_i of the right box. Say that the string that the machine reads while going through the left box is x , and the string it reads while going through the right box is y . If we run D on $x^R y^R$, it would arrive at q_i after reading x^R and then arrive at some accepting state after reading y^R . Since $\text{inside-out}(L)$ can be written as $\{xy \mid x^R y^R \text{ in } L\}$, we have proved that any string read by our machine is in $\text{inside-out}(L)$.

Note that the machine we have made now only works when D ends up at the specific state q_i after reading u . We just need to have portions like this for every possible intermediate state q_i , and we're done.



To be eligible for the skip, your solution must clearly be an honest attempt at solving the problem, and not something you did to just get entitled to skipping a problem on the next homework.

2. $\text{scramble}(L) = \{x \mid \text{for some reordering } w \text{ of the characters of } x, w \text{ is in } L.\}$

Solution

Take $L = \{(01)^*\}$. Since we must consider all reorderings, $\text{scramble}(L) = \{x \mid x \text{ has an equal number of 0's and 1's}\}$. This can be proven to be irregular using the pumping lemma. Given pumping length n , consider the string $0^n 1^n$, which is clearly in $\text{scramble}(L)$ and longer than n . The substring to be pumped must be the initial 0^n portion, so pumping that string will cause the number of 0's and 1's to become different. This concludes the proof.

Problem 3

For any regular language L and homomorphism h , define $h^*(L)$ as the infinite union

$$h^*(L) = L \sqcup h(L) \sqcup h(h(L)) \sqcup h(h(h(L))) \sqcup \dots$$

Is $h^*(L)$ necessarily regular? Prove your assertion.

Solution

Consider $L = \{01\}$, and h defined as $h(0) = 00$, $h(1) = 11$. Then $h^*(L) = \{01, 0011, 00001111, \dots\} = \{0^n 1^n \mid n = 2^k \text{ for } k \geq 0\}$. This is already known to be not regular (don't do this for exams).

Problem 4

Recall that the equivalence relation on strings that is induced by a language L is the relation R_L defined by

$xR_L y$ iff (for every string z , xz in L iff yz in L)

For each of the following languages, describe the equivalence classes, and prove that your answer is correct, i.e., that they are form a partition of all strings, that any two strings in the same class are indeed equivalent with respect to R_L , and that for any strings x, y chosen from different classes, NOT $x R_L y$. Conclude whatever you can regarding the regularity of the set.

1. $\{x \in \{0,1\}^* \mid x \text{ contains at least 3 0s}\}$

Solution

The classes are $E_0 = \{x \mid x \text{ contains no 0's}\}$, $E_1 = \{x \mid x \text{ contains one 0}\}$, $E_2 = \{x \mid x \text{ contains two 0's}\}$, $E_3 = \{x \mid x \text{ contains three 0's or more}\}$. These classes cover the entire range of the possible number of 0's in a string, and clearly don't overlap, so they are a partition of all strings.

For any two strings x, y picked from the same class, they either have the same amount of 0's or have more than three 0's. Therefore, if we append the same string z to both, xz and yz will either have the same amount of 0's or have more than three 0's. Hence xz and yz will be in the same class again.

For any two strings x, y picked from different classes, we can consider two cases. The first is when one of them is in class E_3 . When we append the empty string to both of them one is in E_3 and hence is accepted and the other is not, so $x \mathbf{R}_L y$ is false. The second case is when both of them are not in E_3 . Then, the two strings have a different number of 0's in them, and the number of 0's is less than three for each. Therefore, we can append just enough 0's to both so that just one of the strings has three 0's while the other doesn't (details omitted).

2. $\{0^i \mid i \text{ is divisible by 2 or by 3}\}$

Solution

We can get a hint from how we designed a machine that checked if a string had a certain multiple m of 0's in it. If the number of 0's was i , we basically kept track of $k \bmod m$. Here i has to be a multiple of either 2 or 3, so we have to keep track of both $i \bmod 2$ and $i \bmod 3$. Since the GCD of 2 and 3 is 6, we can simply keep track of $i \bmod 6$ (you don't need to understand why we're taking the GCD here).

So the equivalence classes are, for $k=0$ to 5 , $E_k=\{0^i \mid i \bmod 6 = k\}$, and $E_{\text{one}}=\{x \mid x \text{ has a 1 in it}\}$. These clearly partition the set of all strings.

Problem 5

[HUM] exercise 4.4.2 a) Draw the table of distinguishabilities for this automaton.

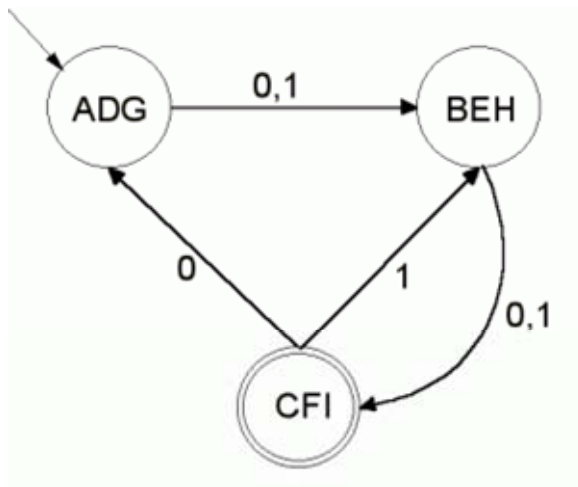
Solution:

B		X
C		XX
D		XX
E		X XX
F		XX XX
G		XX XX
H		X XX XX
I		XX XX XX
- + - - - - - - - -		
		ABCDEFGH

b) Construct a minimum-state equivalent DFA

Solution:

From the table in part (a) it is clear that $\{A, D, G\}$, $\{B, E, H\}$, and $\{C, F, I\}$ are the equivalence classes. Therefore the minimum DFA has 3 states:

**GRADUATE PROBLEMS****Problem G9 (required only for 1-unit graduate students)**

Let N_1 be an NFA. Our goal is to (not necessarily efficiently) find the smallest DFA that accepts the language $L(N_1)$. Of course, we could just convert it to a deterministic D_1 , and then minimize, but that would be no fun.

Instead, prove that the following bizarre procedure works:

1. Convert N_1 to D_1 using the subset construction.
2. Construct an NFA, N_2 , that recognizes the reverse of the language that is recognized by D_1 , by applying the reversal procedure in [HUM p.137] on D_1 :
 1. Reverse all the arcs
 2. The new accepting state will be the old start state
 3. Likewise, the new start state will be the old accepting states. Since the start state cannot be a multiple of states, simply create a completely new state and name it the start state, and have epsilon transitions to all of the old accepting states.
3. Repeat the steps as follows: step 1 again to get D_2 , step 2 again to get N_3 (the reverse of D_2), and step 1 again to get D_3 . D_3 is the minimal machine for N_1 .

Solution

We present the intuition here and postpone the formal proof for later (since the proof doesn't pertain to midterm 1). We learned in class that in order to minimize a DFA, we need to merge all of the states that behave the same, i.e. accept and reject the same set of strings. Therefore, vaguely, two states are identical if we can reach both of them beginning from an accepting state and reading the same strings backwards. This is exactly what we are checking when we construct N_2 and then do a subset construction. Each subset would be something like the set of equivalent states. We just reverse N_2 again to get a proper DFA

Decizii si minimizari:

Problem 1

For the following, describe a step-by-step procedure to obtain the result.

1. *Given a DFA, determine if it accepts all possible strings over the designated alphabet.*

Solution

Since it seems to be difficult to determine this directly, construct the DFA that accepts the complement by using the construction we learned (namely, "flip" the accepting states to be the ones that didn't used to be). Now we just need to determine if this DFA accepts no strings. For this, we just do a depth-first search over the arcs in the DFA starting from the start state. If we find any accept state, then there is a path from the start state to that, so there the string that we obtain by reading off the labels of that path is a string that is accepted by the complemented DFA. If we don't find any accept state, we know that there is no way for a string to be accepted by the DFA. So, if the complement DFA doesn't accept any strings, we know that the original accepted all possible strings.

2. *Given a DFA, determine if the complement of the language it recognizes is finite.*

Solution

We first complement the DFA so that it recognizes the complement of the language it originally did. Let us show that the following two are equivalent.

- (1) The DFA recognizes a finite language
- (2) The DFA has no loops in it

If condition (1) holds, there cannot be any loops in the DFA, since we could loop an unbounded number of times, generating an arbitrary number of strings, so (2) holds. Going the other way, if (2) holds, the structure of the DFA is a tree. Since the number of states are finite, the depth and degree is finite, and hence there are a finite number of paths. Therefore, the language is finite and (1) holds.

Therefore, we just need to check that the DFA has no loops in it. To do this, we perform a depth-first search and if we don't run into duplicate nodes in any single path, there are no loops, and we conclude that the language the DFA recognizes is finite.

3. *Given two DFA's that work over the same alphabet, determine if there is a string that is in neither of the two languages recognized by the DFA's.*

Solution

Name the two DFAs D_1 and D_2 , and the two languages recognized by them, L_1 and L_2 , respectively. The question is asking if the complement of $L_1 \cap L_2$ is non-empty. Using DeMorgan's law, if call the complement of L_1 and L_2 , L_1' and L_2' , respectively, this is the same as asking if $L_1' \cap L_2'$ is non-empty.

Hence, we first obtain the complements of D_1 and D_2 , which we call D_1' and D_2' . We do a cross-product construction on the two machines, to get a machine that recognizes $L_1' \cap L_2'$. To check that this machine recognizes at least one string, we do a depth-first search starting

from the start state, and if we reach any accepting state, conclude that a string is in $L_1' \cap L_2'$ and hence in the complement of $L_1 \sqcup L_2$.

Problem 3

*Carefully define what it means for an equivalence relation on strings to be **left**-invariant. Then prove or disprove the "democratic" version of the Myhill-Nerode theorem: A language is regular if and only if it is the union of some of the equivalence classes of a **left**-invariant equivalence relation of finite index.*

Solution An equivalence relation $\equiv$ is left-invariant if and only if the following condition holds:

$$\square z, \text{ if } x \equiv y \text{ then } zx \equiv zy$$

Note that the only difference from right-invariant relations is that z is put on the left instead of on the right. Intuitively, we could reverse all the strings to get a sort-of right-invariance condition. We could define a new equivalence relation $\equiv^R$ such that $x \equiv^R y$ means $x^R \equiv y^R$ (note that this is the same as saying $x^R \equiv^R y^R$ means $x \equiv y$). Then the right-invariance stated above implies:

$$\square z, \text{ if } x^R \equiv^R y^R \text{ then } (zx)^R \equiv^R (zy)^R$$

which is the same as:

$$\square z, \text{ if } x^R \equiv^R y^R \text{ then } x^R z^R \equiv^R y^R z^R$$

To reduce the clutter, we set $a=x^R$ and $b=y^R$. Also note that since the quantifier on z is "for all", we can simply replace z^R with z (since all if the statement is true for all r , it would also be true for all z^R , and vice-versa):

$$\square z, \text{ if } a \equiv^R b \text{ then } az \equiv^R bz$$

So $\equiv^R$ is **right**-invariant! We provide the gist of the proof and postpone the formal one for later. Imagine two parallel universes of strings, say **U** and **R**, the only difference between the two being that all strings are reversed, and equivalence relations are also "reversed" as shown above. Since both universes still simply consist of strings, the Myhill-Nerode theorem (we will call it the MNT from now on) holds in both of them. For the "democratic" MNT, we need to show that the following two are equivalent in, say **U**:

- (1) L is regular.
- (2) L is the union of some of the equivalence classes of a left-invariant equivalence relation.

To show (1) implies (2), assume (1). By the original MNT, L^R in **R** consists of some union of the equivalence classes of a right-invariant equivalence relation. The reverses of these equivalence classes will constitute L . Also, they will be the equivalence classes of the reverse of the equivalence relation, which, as shown above, will be left-invariant. Hence, (2) holds.

To show (2) implies (1), assume (2). L is the union of some of the equivalence classes of a left-invariant equivalence relation in **U**. Taking all of this to **R** makes L^R the union of some of the equivalence classes of a right-invariant equivalence relation. By the MNT, L^R is regular. Therefore, L is regular and (1) holds.

Problem 4

What is a Buchi automaton (you may use any sources including the web)? Why are they important?

What (infinite) strings are accepted by the Buchi automaton below, where the alphabet is $\{0,1\}$?

Solution

A [Buchi automaton](#) is a DFA for infinite strings. The definitions of states, start states, transitions, and accept states are the same, but with a Buchi automaton a string is accepted iff processing the string causes the DFA to pass through an accept state an infinite number of times. More formally:

$\exists i > 0, \exists j > i$ such that $s_j \in F$

Where s_j is the state reached after processing j characters. In other words, no matter how far you are into the string, you will always reach another accept state. This is written as $\{\text{always}\} \{\text{eventually}\} F$ in temporal logic.

Buchi Automata are useful in model checking, for verifying the correctness of nonterminating computer programs. A DFA can only simulate terminating programs.

In this particular Buchi automaton, a 0 always leads to q_0 and a 1 always leads to q_1 . Since q_1 is the only accept state, this means that the Buchi automata accepts an infinite string iff it contains an infinite number of 1s.

Problem 5

Consider the CFG, G , defined by productions:

$$S \rightarrow aS \mid Sb \mid a \mid b$$

Part a) Prove by induction on the string length that no string in $L(G)$ has "ba" as substring.

Solution:

Let n be the length of the string w .

Base Case: $n=1$. Since $S \rightarrow a$ and $S \rightarrow b$, "a" and "b" are in the language. Neither of these contain "ba".

Inductive Hypothesis: All strings in $L(G)$ with length $n-1$ do not contain "ba".

Inductive Step: Let w be a string of length n . There are two recursive productions, and thus two cases:

- $S \rightarrow aS$. Therefore $w = aw'$, where $|w'| = n-1$, so w' does not contain "ba". The only way for w to contain "ba" is if the new character -- the "a" at the beginning -- is a part of it. The new character can not be the "b", as it is an "a", and it cannot be the "a", as it is the first character. Therefore, if $w = aw'$, w does not contain "ba".
- $S \rightarrow Sb$. Therefore $w = w'b$, where $|w'| = n-1$, so w' does not contain "ba". The only way for w to contain "ba" is if the new character -- the "b" at the end -- is a part of it. The new character can not be the "a", as it is a "b", and it cannot be the "b", as it is the last character. Therefore, if $w = w'b$, w does not contain "ba".

Therefore, the inductive hypothesis also holds for n , and therefore for all strings. Therefore, no strings in G contain "ba".

Part b) Describe $L(G)$ informally. Justify your answer using part a.

Solution

Since we know that strings in $L=L(G)$ do not contain "ba", we can construct a language L' of all strings that do not contain "ba". Simply build a DFA that recognizes strings containing "ba" and invert it (details left as an exercise to the reader). The result is $L'=a^*b^*$. Since it is clear that L does not contain ϵ , modify L' to exclude it, so $L'=a^*b^*-\epsilon=a^*ab^*|a^*b^*b$. Clearly $L \subseteq L'$; it remains to show $L' \subseteq L$.

To do this, note that if $w \in L'$ contains at least one a , then $w=aw'=aS$, where $w' \in L'$. This matches the first production. Similarly, if $w \in L'$ contains at least one b , then $w=w'b=Sb$, matching the second production. Therefore, any $w \in L'$ can be reduced, one character at a time, to a string of length 1, thus matching the third or fourth production in G . Therefore all strings in L' are produced by G .

Therefore $L' \subseteq L$.

Since $L \subseteq L'$ and $L' \subseteq L$, $L=L'$. Therefore, $L(G)=a^*a|b^*b = \{a^ib^j \mid i+j \geq 1\}$.

Gramatici independente de context si automate PD:

Problem 1

Build a grammar for the following languages:

$$1) \{a^i b^j c^k \mid j=i+k\}$$

Solution

$$S \rightarrow AC$$

$$A \rightarrow aAb \mid \varepsilon$$

$$C \rightarrow bCc \mid \varepsilon$$

Break the language into 2 pieces: $a^i b^{i+k} c^k = a^i b^i b^k c^k = (a^i b^i) (b^k c^k)$. **A** generates the familiar $a^i b^i$ language (the first half), and **C** generates the similarly-structured $b^k c^k$ (the second half). The $S \rightarrow AC$ production pieces together these two sublanguages to produce the desired language.

$$2) \{w \in \{0,1\}^* \mid w \text{ has more 0s than 1s}\}$$

Solution

$$S \rightarrow A0A \mid A0S$$

$$A \rightarrow 0A1A \mid 1A0A \mid \varepsilon$$

A represents all strings with an equal number of 0s and 1s. This is the grammar from problem 2 below. **S** represents a string of A's separated by 0s. It is clear that this grammar produces only strings with more 0s than 1s, as each production either leaves the difference between 0s and 1s unchanged (the **A** productions), or increases the difference by one, in favor of the 0s (the **S** productions).

To show that this grammar produces all strings with more 0s than 1s, for each string w , define functions $f(i,w)$ and $g(j,w)$. Function $f(i,w)$ is the number of 0s minus the number of 1s in the first i characters of w , and $g(j,w)$ is the minimum i such that $f(i,w)=j$. Since $f(i,w)$ changes by 1 each step starting at 0 and ending at $f(|w|,w)>0$, $g(j,w)$ is defined for all j from 1 to $f(|w|,w)$ and each j corresponds to a 0 in w . Since $f(g(j+1,w)) = f(g(j,w))+1$, the string between $g(j+1,w)$ and $g(j,w)$ will have the same number of 0s as 1s. Therefore, every string with more 0s than 1s can be written as a series of strings with an equal number of 0s as 1s, separated by 0s. This is exactly the type of strings generated by this grammar.

$$3) \{x \in \{0,1\}^* \mid x \neq x^R\}$$

Solution

$$S \rightarrow 1S1 \mid 0S0 \mid T$$

$$T \rightarrow 1R0 \mid 0R1$$

$$R \rightarrow 0R \mid 1R \mid \varepsilon$$

In order for a string not to be a palindrome, there must be some character such that the corresponding character in the reverse is different. **T** represents all strings where this character is the first character, i.e. all strings where the first and last character are different. Once this different character has been found, the remaining characters are irrelevant, so **R** represents all binary strings. But since this "wrong" character may not be the first one, the **S** productions are needed. The $S \rightarrow 1S1$ and $S \rightarrow 0S0$ productions frame the **T** string in a palindrome-like fashion so that the first "wrong" character can appear anywhere in the first half of the string. But since **S** cannot reach a terminal without going through **T**, the derivation must eventually find a "wrong" character.

$$4) \{x \in \{0,1\}^* \mid x \text{ cannot be written as } ww \text{ for any } w \in \{0,1\}^*\}$$

Solution

$S \rightarrow AB \mid BA \mid A \mid B$

$A \rightarrow CAC \mid 0$

$B \rightarrow CBC \mid 1$

$C \rightarrow 0 \mid 1$

A represents odd strings with a 0 at the center, and **B** represents odd strings with 1 at the center. Together they represent all odd strings. Since an odd string can never be written as the sum of two identical strings, all odd strings are in the language. Hence the $S \rightarrow A \mid B$ productions cover all odd strings.

As for even strings, if $|w|$ is even, then $w=uv$, where $|u|=|v|$. If $u \neq v$, they must differ by at least one character. Without loss of generality, assume this character is a 0 in u and a 1 in v . Therefore $u=u_10u_2$ and $v=v_11v_2$, so $w=u_10u_2v_11v_2$, where $|u_1|=|v_1|$ and $|u_2|=|v_2|$. u_2v_1 can be regrouped and rewritten as $u'_2v'_1$, where $|u'_2|=|u_1|$ and $|v'_1|=|v_2|$. This can be done because both strings have length $|u_1|+|u_2|$. Therefore $w=u_10u'_2v'_11v_2 = (u_10u'_2)(v'_11v_2)$. $u_10u'_2$ is an odd string with a 0 at the center, and v'_11v_2 is an odd string with 1 at the center. Therefore $S \rightarrow AB$ produces all strings of this type. Similarly, $S \rightarrow BA$ produces all strings of the form $u_11u_2v_10v_2$ (with the same restrictions as above). Therefore the above produces only and all strings that cannot be written as ww .

Problem 2

Consider the CFG G defined by the productions

$S \rightarrow aSbS \mid bSaS \mid \varepsilon$

Prove that $L(G)$ is the set of all strings with an equal number of a 's and b 's.

Solution

There are two components to this proof. Let L' be the language of strings over $\{a,b\}^*$ with an equal number of a 's and b 's. To prove $L(G)=L$, one needs to prove (1) $L(G) \subseteq L'$, and (2) $L' \subseteq L(G)$.

1) $L(G) \subseteq L'$

For every string w in $L(G)$, show that w has an equal number of a 's and b 's. Proof by induction over the number of steps in the derivation $S \xRightarrow{*} w$:

Base Case: $w=\varepsilon$, $S \xRightarrow{*} \varepsilon$, w has an equal number (0) of a 's and b 's.

Inductive hypothesis: If $S \xRightarrow{*} w$ in less than n steps, then w has an equal number of a 's and b 's.

Inductive step: Let w be a string such that $S \xRightarrow{*} w$ in n steps. If w begins with a , $S \xRightarrow{*} aSbS \xRightarrow{*} aw_1bw_2$. Clearly $S \xRightarrow{*} w_1$ and $S \xRightarrow{*} w_2$, each with less than n steps. By the Induction Hypothesis, w_1 and w_2 each have an equal number of a 's and b 's. Therefore w has an equal number $((|w_1|+|w_2|)/2+1)$ of a 's and b 's. If w begins with b , $S \xRightarrow{*} bSaS \xRightarrow{*} bw_1aw_2$, and proceed as above.

2) $L' \subseteq L(G)$

For every string w with an equal number of a 's and b 's, $S \xRightarrow{*} w$. Proof by induction over $|w|$:

Base Case: $w=\varepsilon$, like in (1).

Inductive Hypothesis: If $|w|<n$ and w has an equal number of a 's and b 's, then $S \xRightarrow{*} w$.

Inductive Step: Let w be a string with an equal number of a 's and b 's. Without loss of generality, assume w begins with a . Find the first character after the first one such that the running total of a 's minus b 's up to that character is 0 (this is like $g(0,w)$ in 1(b) above, the first $i>1$ such that $f(i,w)=0$). Since the running total starts at one, and must eventually decrease to 0, this character must be a b . Therefore $w=aw_1bw_2$, where w_1 is the string between this a and b , and w_2 is the string after the b . Since aw_1b is defined to have an equal number of a 's and b 's, w_1 also has an equal number. Similarly w_2 must also have an equal number of a 's and b 's, as it is formed by taking a string with an equal number (w) and subtracting an equal number of a 's and b 's (aw_1b). Clearly $|w_1|<n$ and $|w_2|<n$. By the inductive hypothesis, $S \xRightarrow{*} w_1$ and $S \xRightarrow{*} w_2$. Therefore the derivation $S \xRightarrow{*} aSbS \xRightarrow{*} aw_1bS \xRightarrow{*}$

$aw_1bw_2=w$. If w begins with b , exchange the a 's and b 's in the above derivation to prove $S \Rightarrow bSaS \Rightarrow^* bw_1aS \Rightarrow^* bw_1aw_2=w$.

Problem 3

[HUM] 7.1.3

Solutions

a) Remove useless symbols

Considering the reachability of the non-terminals: S reaches A, B and A reaches C . So every non-terminal is reachable.

Let's check if all the non-terminals are generating: S generates since we can take $S \rightarrow 0A0 \rightarrow 0C0 \rightarrow 00$. $A \rightarrow C \rightarrow S$ and $B \rightarrow S$ so all non-terminals are generating.

So there are no useless symbols.

b) Remove epsilon productions

C is nullable. $A \rightarrow C$ so A is nullable. $B \rightarrow A$ so B is nullable. $S \rightarrow BB$, so S is nullable. We must look at all productions that involve any of the nullable non-terminals. For $S \rightarrow 0A0$, we must add $S \rightarrow 00$ (for when A is made epsilon). For $S \rightarrow 1B1$, we include $S \rightarrow 11$. For $S \rightarrow BB$, we include $S \rightarrow B$ (when the first or second B is nullified). One thing to note is that the original language produced epsilon via $S \rightarrow BB \rightarrow AA \rightarrow CC \rightarrow \text{epsilon}$, but when we remove epsilon productions, this is removed from the language.

We now have:

$S \rightarrow 0A0 \mid 1B1 \mid BB \mid 00 \mid 11 \mid B \mid A \rightarrow C \mid B \rightarrow S \mid A \mid C \rightarrow S$

c) Remove unit productions

Even without going through the textbook's algorithm, $A \rightarrow C \rightarrow S \rightarrow B \rightarrow A$, making a complete cycle, so every non-terminal is a unit pair with every other non-terminal. The only productions that are not unit productions are those for S , so we have:

$S \rightarrow 0A0 \mid 1B1 \mid BB \mid 00 \mid 11 \mid A \rightarrow 0A0 \mid 1B1 \mid BB \mid 00 \mid 11 \mid B \rightarrow 0A0 \mid 1B1 \mid BB \mid 00 \mid 11 \mid C \rightarrow 0A0 \mid 1B1 \mid BB \mid 00 \mid 11$

d) Construct CNF grammar

The set of productions for S, A, B , and C are the same, so we just consider the one for S . First, we get rid of all terminals that show up on the right hand side of the productions. Set two non-terminals that just generate terminals:

$T_0 \rightarrow 0$

$T_1 \rightarrow 1$

Now we get:

$S \rightarrow T_0AT_0 \mid T_1BT_1 \mid BB \mid T_0T_0 \mid T_1T_1$

Second, we should get rid of productions that don't have exactly two non-terminals on the right hand side. Here, these are the two productions, $S \rightarrow T_0AT_0 \mid T_1BT_1$. We take care of this by defining new non-terminals that "cascade" across the right hand side.

$S \rightarrow C_{0A}T_0$
 $C_{0A} \rightarrow T_0A$

$S \rightarrow C_{1B}T_1$
 $C_{1B} \rightarrow T_1B$

So the final result we get is:

$S \rightarrow C_{0A}T_0 \mid C_{1B}T_1 \mid BB \mid T_0T_0 \mid T_1T_1$
 $A \rightarrow C_{0A}T_0 \mid C_{1B}T_1 \mid BB \mid T_0T_0 \mid T_1T_1$
 $B \rightarrow C_{0A}T_0 \mid C_{1B}T_1 \mid BB \mid T_0T_0 \mid T_1T_1$
 $C \rightarrow C_{0A}T_0 \mid C_{1B}T_1 \mid BB \mid T_0T_0 \mid T_1T_1$
 $C_{0A} \rightarrow T_0A$
 $C_{1B} \rightarrow T_1B$

Problem 4

Solution

Part (a)

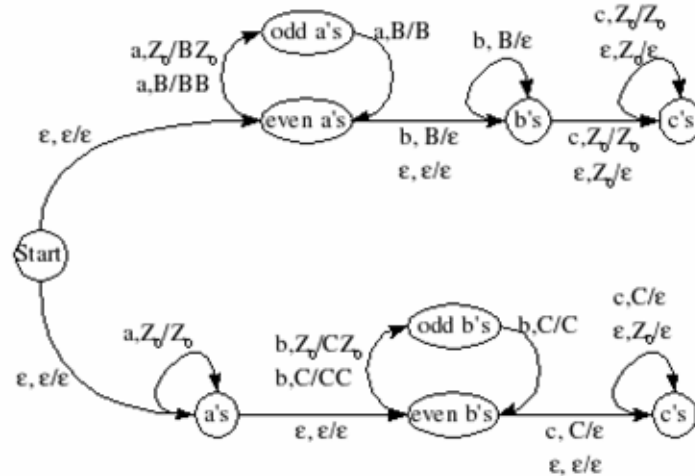
The number of c's has to be twice the sum of the number of a's and b's. Hence, for each a we see, there must be two c's, and likewise for each b. Let's use a stack symbol, say C, to denote a 'c' that has to exist. Then each time we see an a, we push two C's, and likewise for b. To make sure that the c's come after the b's, and the b's come after the a's, we make three consecutive states that come after each other -- the first reading only a's, the next b's, and the last c's. Since we are accepting by empty stack, there is no accept state. We just have to make sure that after we are done, we pop the Z_0 at the very bottom of the stack.



Part (b)

The given language can be seen as the union of $\{a^ib^jc^k : i=2j\}$ and $\{a^ib^jc^k : j=2k\}$. We can construct a PDA for each, and the final PDA will be their union, with a new start state that non-deterministically chooses one of them to run.

Let's consider the first machine we should make, $\{a^ib^jc^k : i=2j\}$. There need to be twice as many a's as there are b's. We could say that there needs to be a b for every other a. So we can push an A symbol onto the stack for every other a and later on pop a b for each A. Then we just need to read the trailing c's. The construction is similar for $\{a^ib^jc^k : j=2k\}$.



Problem 5

Create a PDA for the following languages:

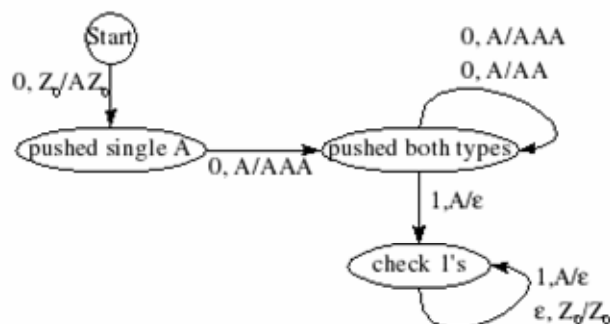
1. $\{0^n 1^m : n < m < 2n\}$

Solution

Looking at problem 4, it would be easy to make PDAs for $m=n$ and $m=2n$. To achieve $m=n$, we push a symbol, say A, onto the stack for every 0 we see, and then later pop one A for each 1 we see. To do $m=2n$, we push two A's onto the stack for every 0.

Now, we need to do something "in between" to make $n < m < 2n$. Instead of always pushing one A or two A's, we non-deterministically push one A or two A's for each 0 we see. We can see that, this way, the total number of A's on the stack will be able to acquire any value between n and $2n$: $n = 1 + 1 + \dots + 1$, $n+1 = 1 + 1 + \dots + 1 + 2$, $n+2 = 1 + 1 + \dots + 1 + 2 + 2$, ..., $2n = 2 + 2 + \dots + 2$.

But, in the language given, m should not equal n or $2n$. In our method, this occurs only when we keep pushing one A for each 0, or keep pushing two A's for each 0. So, we just make sure that we do each type of push at least once. In achieving any total of A's on the stack, the order in which we do the two different types of pushes doesn't matter: only the total number of times we pushed two A's and the total number of times we pushed a single A, matter. So, we could just start off by pushing one A once, and then two A's. Here is a PDA that accepts by empty stack.

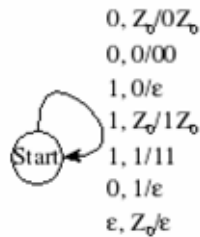


2. $\{w: \#0(w) = \#1(w)\}$ ($\#x(w)$ denotes the number of occurrences of the character x in w)

Solution

As we read the string we need to keep track of the difference between the number of 0's and 1's, and accept when we are done reading and there is no difference. Let's say that when there is an excess of 0's, we put that many 0's onto the stack, and when there is an excess of 1's, we put that many 1's.

Let's say that we have an excess of 0's. Then, the stack will consist entirely of 0's. If we read a 0, we should push another 0 onto the stack. If we read a 1, there is one less excess, so we pop a 0. If we keep reading 1 and eventually make the stack empty, and then we encounter another 1, we are starting to have excess 1's, so we push a 1 on the stack. We have the similar transitions when there is an excess of 1's. Here is a PDA that accepts by empty stack.



Lema de pompare pentru LIC si proprietati de inchidere:

Problem 1

[HUM] 7.2.1 b, c, e, f Use the pumping Lemma to show each of these languages not to be context-free.

$$b) L = \{a^n b^n c^i \mid i \leq n\}$$

Solution

Let $s = a^n b^n c^n \in L = vwxyz$, where n is the pumping length. If w or y each contain characters of different types (e.g. $w = a^i b^j$), then vw^2xy^2z will not be of the form $a^n b^n c^i$, and so will not be in L . Similarly, if generating vw^jxy^jz pumps the a 's and not the b 's, or vice versa, vw^jxy^jz will not be of the form $a^n b^n c^i$, as the a 's and b 's will not be equal. That leaves two cases: (1) $w = a^j$ and $y = b^j$, or (2) $w = c^j$ and $y = c^k$. In case (1), $vw^0xy^0z = vxz = a^{n-j}b^{n-j}c^n$. Since $n > n-j$, $vxz \notin L$. In case (2), $vw^2xy^2z = a^n b^n c^{n+j+k}$. Since $n+j+k > n$, $vw^2xy^2z \notin L$. Therefore s cannot be pumped, so L is not context free.

$$c) M = \{0^p \mid p \text{ is prime}\}$$

Solution

Let $s = 0^n \in M = vwxyz$, where n is an arbitrary prime larger than the pumping length. Set $j = |w| + |y| > 2$, so $vw^jxy^jz = 0^{n+(j-1)j}$. Set $i = n+1$, $vw^{n+1}xy^{n+1}z = 0^{n+nj} = 0^{n(j+1)} \notin M$. Therefore s cannot be pumped, so M is not context free.

$$e) N = \{a^n b^n c^i \mid n \leq i \leq 2n\}$$

Solution

Let $s = a^n b^n c^n \in N = vwxyz$, where n is the pumping length. The same reasoning as in 1(b) handles all cases except (1) $w = a^j$ and $y = b^j$, and (2) $w = c^j$ and $y = c^k$. In case (1), $vw^2xy^2z = a^{n+j}b^{n+j}c^n$. Since $n < n+j$, $vw^2xy^2z \notin N$. In case (2), $vw^0xy^0z = vxz = a^n b^n c^{n-j-k}$. Since $n-j-k < n$, $vxz \notin N$. Therefore s cannot be pumped, so L is not context free.

$$f) P = \{ww^Rw \mid w \in \{0,1\}^*\}$$

Solution

Let $0^n 1^n$ be the repeated string, and thus set $s = ww^Rw = 0^n 1^{2n} 0^n 1^n \in P = vwxyz$, where n is the pumping length. Define a *segment* of s to be a maximal string of all 1s or all 0s (s has 4). There are a variety of strategies of setting w and y in an attempt to pump s , all of which fail:

- w and y can be in the same segment. For example, if $v = \epsilon$ w and y are both in the first segment. Set $k = |w| + |y|$. Depending on the choice of segments, $vw^2xy^2z = 0^{n+k} 1^n \cdot 1^n 0^{n+k} \cdot 0^{n-k} 1^n$, $0^n 1^{n+k/2} \cdot 1^{n+k/2} 0^n \cdot 0^n 1^n$, $0^n 1^n \cdot 1^n 0^n \cdot 0^{n+k} 1^n$, or $0^n 1^n \cdot 1^n 0^n \cdot 0^n 1^{n+k}$. The strings have been broken up to show the best decomposition of s as ww^Rw' . But in none of these cases does $w = w'$.

Therefore $vw^2xy^2z \notin P$ in all cases.

- w and y can each be in a different segment. Since $|wxy| \leq n$, they must be in adjacent segments. Set $j = |w|$ and $k = |y|$. Depending on the choice of segments, $vw^2xy^2z = 0^{n+j} 1^{n+k/2} \cdot 1^{n+k/2} 0^{n+j} \cdot 0^{n-j} 1^n$, $0^n 1^{n+j/2} \cdot 1^{n+j/2} 0^n \cdot 0^{n+k} 1^n$, or $0^n 1^n \cdot 1^n 0^n \cdot 0^{n+j} 1^{n+k}$. The spacing and reasoning are the same as in the previous case. Therefore $vw^2xy^2z \notin P$ in all cases.
- w and y can be across the boundaries between segments. Since $|wxy| \leq n$, and the boundaries are all $2n$ characters across, w and y cannot span multiple boundaries, and only one of the two can be across a boundary. The other must be completely contained in an adjacent segment. There are 3 boundaries and the segment can be before or after the boundary, so

there are six possibilities. If w and y are of the form 0^m , 1^m , 0^j1^k , or 1^j0^k (depending on the choice of boundary and segment), the possibilities for vw^2xy^2z are $0^{n+m}1^k0^j1^{2n}0^{2n}1^n$, $0^n1^k0^j1^{2n+m}0^{2n}1^n$, $0^n1^{2n+m}0^k1^j0^{2n}1^n$, $0^n1^{2n}0^k1^j0^{2n+m}1^n$, $0^n1^{2n}0^{2n+m}1^k0^j1^n$, and $0^n1^{2n}0^{2n}1^k0^j1^{n+m}$. The decompositions have not been shown, for it is fairly clear from simple inspection that it is not possible to match the 0^j1^k and 1^j0^k substrings to fit into the ww^Rw format. Therefore $vw^2xy^2z \notin P$ in all cases.

Therefore s cannot be pumped, so P is not context free.

Problem 2

Reiterate the proof of the pumping lemma for CFLs, assuming that the given grammar only has productions of the following form:

$A \rightarrow a$ where A a non-terminal and a a terminal in Σ

$A \rightarrow aB$ where A and B are non-terminals and a in Σ

$A \rightarrow \epsilon$ where A is a non-terminal.

Note that the statement of the lemma will change because of the new conditions. You must give enough details in the new proof to show how certain steps are changed and why.

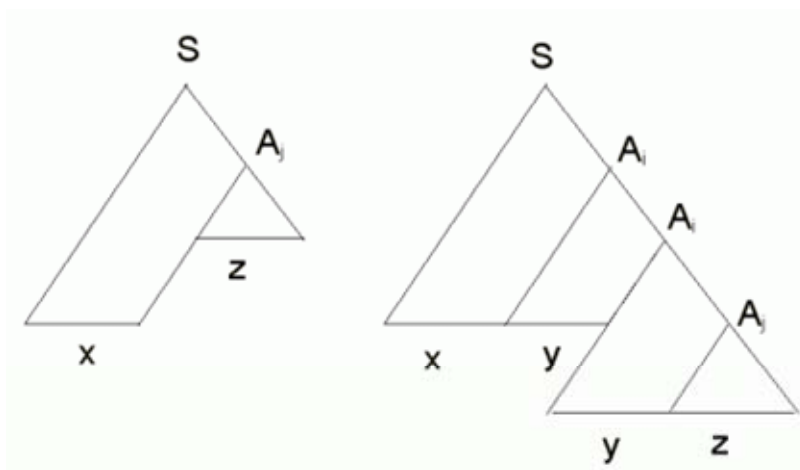
Solution

First note that if the longest path in the derivation of a string w is n , then $|w| \leq n$. This can be proven inductively, similarly to the proof in 7.17: The base case is $S \rightarrow a$, which produces a string of length 1 with 1 production. The inductive hypothesis is that if w has a longest path of $n-1$ steps, $|w| \leq n-1$. For the inductive step, consider the string w , which has a longest path of n steps. The root production must be of the form $A \rightarrow aB$, and so the longest path from B to w' (where $w = aw'$) must have $n-1$ steps. Therefore the longest path from B to w' has length $n-1$, so by the inductive hypothesis, $|w'| \leq n-1$. Therefore $|w| = |aw'| = |w'| + 1 \leq n$. Notice this result is different from 7.17 because each production only has at most 1 symbol on each RHS. Using the same induction with a different base case ($S \rightarrow \epsilon$: 1 production, $|w| = 0$) shows that $|w| \geq n-1$.

Set m to be the number of nonterminals in the grammar. Set $n = m+1$ and consider all strings length n or longer. Since if the longest path is $\leq m$, then $|w| < n$, the longest path must have length $k \geq m+1$. Consider the nonterminals along the longest path. Since this path is longer than the number of nonterminals, by the pigeon-hole principle at least one nonterminal must repeat. Suppose $A_i = A_j$, where $i < j < k \leq n+1$ (j cannot equal k because the last term in the sequence is a terminal). Then the parse tree looks something like the figures below. Notice that the longest path must be along the right edge, since all productions not at the edge are of the form $A \rightarrow aB$. The figure on the left is comparable to 7.5 in HUM and shows the longest path, while the figure on the right is comparable to 7.6 and shows the dividing of w into components based on A_i and A_j .



According to the figure, A_j yields z , A_i yields yz , and S yields xyz . If A_i is replaced by A_j , S yields xz . Similarly, if A_j is replaced by another copy of A_i , S yields xy^2z . Repeating this substitution $i-1$ times shows that S yields $xy^{i-1}z$. The resulting parse trees are below. This figure is comparable to 7.7, but the restriction that nonterminals be only along the right side reduces the number of variables and makes the diagrams simpler.



Since $i < j$, $|y| \geq 1$. Similarly, since $j < k \leq n+1$, $j \leq n$, so $|xy| \leq n$. Therefore, the final lemma becomes:
 If L is a language with a CFG of the form defined above, then there exists a constant n such that for all strings $w \in L$ such that $|w| \geq n$, w can be written as $w = xyz$ such that:

1. $|y| \geq 1$.
2. $|xy| \leq n$.
3. $xy^iz \in L$ for all $i \geq 0$.

Note that this is the pumping lemma for regular languages.

Problem 3

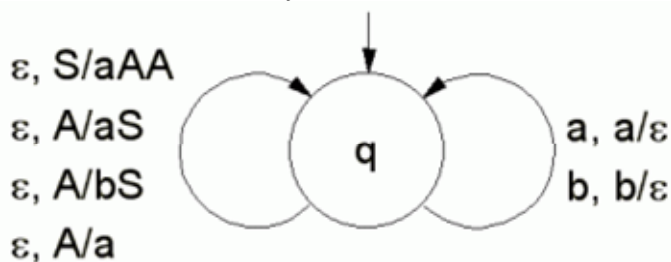
[HUM] 6.3.2 Convert the following grammar to a PDA that accepts the same language by empty stack:

$S \rightarrow aAA$

$A \rightarrow aS \mid bS \mid a$

Solution

Use the construction on HUM 238. Productions are processed on the left, input symbols on the right. The PDA starts with symbol S on the stack, and terminates with an empty stack.



Problem 4

[HUM] 7.3.1a Show CFLs are closed under the init operator, where $init(L) = \{a \mid \exists x \text{ such that } ax \in L\}$

Solution

Given a language L and a PDA P that accepts L with an empty stack, construct a PDA that accepts $init(L)$ with an empty stack. With NFAs, this was done by making all states that could reach a final state into a final state. But with PDAs this is more complicated, as the stack needs to be dealt with. Here are the steps for building a PDA P'' to accept $init(L)$:

1. **Construct twin PDA P' .** P' looks just like P , but with different state labels (for every state q_i in P , there is a state q'_i in P'), and transitions (If $(q_j, \alpha) \rightarrow \delta(q_i, a, A)$, then $(q'_j, \alpha) \rightarrow \delta'(q'_i, \epsilon, A)$). P' does the same things as P on the stack and states, only without looking at the input. On its own, P' is not very useful (it recognizes ϵ iff $N(P)$ is not empty), but when paired with P , it empties the stack after finishing the input (if it is a prefix of some string in L).
2. **Combine P with P' to form P'' .** P'' links the two PDAs together. It has the same alphabets, start symbol, etc. and has all states from both P and P' . The transition function is the union of the two δ 's from P and P' , plus transitions of the form $\delta(q_i, \epsilon, \epsilon) \rightarrow (q'_i, \epsilon)$, which nondeterministically jumps from the P component to the same point in the P' component. P'' recognizes all prefixes of strings in $N(P)$ by processing the string in the P component, jumping to the P' component after reading the entire prefix, and then emptying the stack by acting as if the rest of the string were still in the input.

To prove P'' recognizes all strings in $\text{init}(L)$, consider a string $w \in N(P)$. There is a series of states and transitions leading from the start state to a state with an empty stack. Write out all of these states and transitions, including the contents of the stack at each state; call this the *transcript* of w in P . This transcript can be converted to a transcript of any prefix a of w in P'' :

1. Select the first character after the last character in a . Find the transition in the transcript that reads that particular character. It will be of the form $\delta(q_i, a, A) \rightarrow (q_j, \alpha)$.
2. Insert a $\delta(q_i, \epsilon, \epsilon) \rightarrow (q'_i, \epsilon)$ transition before this transition.
3. replace every transition after and including the transition found in step 1 with the corresponding transition and state in P' . (q_i, α) becomes (q'_i, α) and $\delta(q_i, a, A) \rightarrow (q_j, \alpha)$ becomes $\delta(q'_i, \epsilon, A) \rightarrow (q'_j, \alpha)$.

Therefore P'' accepts every prefix of every string in $N(P)$.

To prove P'' recognizes only strings in $\text{init}(L)$, simply do the above process in reverse. Start with a string $a \in P''$, and determine its transcript. From there, build a transcript in P of a w such that a is a prefix of w :

1. Find the $\delta(q_i, \epsilon, \epsilon) \rightarrow (q'_i, \epsilon)$ transition.
2. Replace every transition after this one with a corresponding transition and state in P . (q'_i, α) becomes (q_i, α) and $\delta(q'_i, \epsilon, A) \rightarrow (q'_j, \alpha)$ becomes $\delta(q_i, a, A) \rightarrow (q_j, \alpha)$. It doesn't matter which value of a is chosen, as long as the resulting transition is found in P .
3. Remove the transition found in step 1.
4. Collect all the transitions that read input; concatenating the characters in order produces w .

Therefore every string recognized by P'' is in $\text{init}(L)$. Therefore $N(P'') = \text{init}(N(P))$.

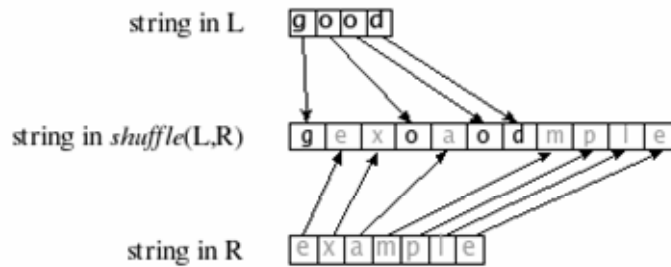
Problem 5

[HUM] 7.3.4 : do only parts **d** and **e**.

Solution

Part d)

We are given two languages L and R , where L is context free, and R is regular. We can generate any string in $\text{shuffle}(L, R)$ by taking some string from L and another from R , and non-deterministically removing one character at the front of one of the strings and placing it at the end of our new string.



Let's say L is recognized by PDA P , and R is recognized by DFA D . We do the same process described above by non-deterministically choosing to run a single step of P or a single step of D . Let's call the new PDA we construct for $shuffle(L,R)$, P_S . P_S 's state will keep track of both of the states of P and D , by the same cross-product construction we used when we constructed the machine for recognizing the intersection of two regular languages. The stack of P_S will keep track of P 's stack. P_S will accept by final state when both P and D accept.

Formally, say $P = (A, \Sigma, \Gamma, \delta_P, a_0, Z_0, F_P)$ and $D = (B, \Sigma, \delta_D, b_0, F_D)$. Note that we assume that the two machines work over the same alphabet, since if they don't, we can simply make a new alphabet that includes both of their alphabets and set that as Σ . Also, we can assume that P accepts by final state (if it doesn't, modify it so that it does). Let $P_S = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$ where:

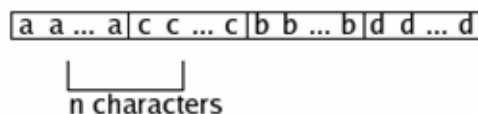
- $Q = \{ \langle a, b \rangle : a \text{ is in } A, b \text{ is in } B \}$
- $\delta(\langle a, b \rangle, i, X) = \{ \langle r, b \rangle, \alpha \} : (r, \alpha) \text{ is in } \delta_P(a, i, X) \} \cup \{ \langle a, s \rangle : s = \delta_D(b, i) \}$ [we union the two sets here to choose to either let P read a character or D read a character]
- $q_0 = \langle a_0, b_0 \rangle$
- $F = \{ \langle a, b \rangle : a \text{ is in } F_P \text{ and } b \text{ is in } F_D \}$

Part e)

Let $L_1 = \{a^n b^n : n \text{ is a non-negative integer}\}$ and $L_2 = \{c^m d^m : m \text{ is a non-negative integer}\}$. Then $shuffle(L_1, L_2)$ contains all strings where the number of a 's is equal to the number of b 's, and the number of c 's is equal to the number of d 's, and all the b 's come after all the a 's, and all the d 's come after the c 's.

We can verify that this is not a CFL by intersecting it with a regular language, since CFL's are supposed to be closed under intersection with regular languages. $shuffle(L_1, L_2) \cap a^*c^*b^*d^* = \{a^n c^n b^n d^n : n \text{ is non-negative}\}$ which we know not to be a CFL, and hence $shuffle(L_1, L_2)$ is not a CFL.

We can also show that $shuffle(L_1, L_2)$ is not a CFL by using the pumping lemma. Assume that the CFL pumping lemma holds, and the pumping constant is n . Consider the string $z = a^n c^n b^n d^n$, which is clearly in $shuffle(L_1, L_2)$ and is longer than n characters. For any decomposition of z into $z = uvwxy$ where the length of vw is no more than n and vx is not empty, vx can only contain at most two types of characters, since vw has length at most n and will span at most two consecutive "segments" of the string.



Because of this, v and x will never contain both a and b , or both c and d . Hence, uv^0wx^0y , would delete some number of a 's but not b 's or vice versa, or delete some number of c 's but not d 's or vice versa. Therefore uv^0wx^0y would have an unequal number of a 's and b 's, or c 's and d 's, and this shows that the CFL pumping lemma does not hold for $shuffle(L_1, L_2)$.

Problem 6

[HUM] 7.3.6

Solution

As mentioned in the skeleton proof provided in the textbook, all the sentential forms of G^R are reverses of the sentential forms of G , and vice-versa. We show this by induction on the number of derivation steps.

- Base Case : We are considering derivations after just a single step. For every production $S \rightarrow x$ in G , there is a production $S \rightarrow x^R$ in G^R , and vice versa. Therefore, for single-step derivations, the sentential forms of G^R are exactly the reverse of all the sentential forms of G .
- Induction step : We assume that for all sentential forms that are generated after n steps those in G^R are exactly the reverse of those in G . Let's consider a sentential form, say s , of G^R that was obtained after $n+1$ steps. Say that the last production used in the derivation was $A \rightarrow y$, and $s = xyz$. The derivation looks like $S \Rightarrow^n xAz \Rightarrow xyz$. Using the induction hypothesis, there is a sentential form of G that is the reverse of xAz , which would be $z^R A x^R$. Also, by construction, there should be a production in G that looks like $A \rightarrow y^R$. Hence there is a derivation in G that is $S \Rightarrow^n z^R A x^R \Rightarrow z^R y^R x^R$. $z^R y^R x^R = (xyz)^R = s^R$, so we have shown that for every sentential form in G^R that is obtained in $n+1$ steps, its reverse can be obtained in $n+1$ steps in G . An almost identical argument can be given to show that every sentential form of G gotten after $n+1$ steps is a reverse of a sentential form of G^R gotten after $n+1$ steps.

Problem 7

(a)

A *middle window* PDA is almost the same as a standard PDA, except that it can also view the "middle" symbol on the stack (in case there are an even number of symbols on the stack, and there are two middle symbols, it just sees the closer one to the top). Formally, the difference is that the transition function now also depends on the symbol in the middle of the stack. Note that nothing else is changed. Most importantly, the PDA *cannot* insert or delete symbols in the middle. Show that a middle window PDA is more powerful than a standard PDA (an example will be sufficient).

Solution

This can be made to recognize $0^n 1^n 2^n$. We can have the states check that we get a bunch of 0's followed by 1's followed by 2's. When we read the 0's and the 1's, we push them onto the stack. We can check that we have an equal number of 0's and 1's by looking at the middle window. If the middle window was two symbols wide, the problem would be simple: after pushing all the 0's and 1's, we just check that we see a "01" in the window, meaning that the boundary between the 0's and 1's is at the center. However, we have to work with just one symbol, so we constantly keep checking the window after reading pushing a character in, and make sure that the time at which we start seeing a 1 at the middle window is the same time we are done pushing 1's on the stack. Now that we are done checking that we have equal number of 0's and 1's, we just check that we have an equal number of 2's and 1's by popping a 1 for each 2. Formal definition will be provided later.

(b)

A *constant stack PDA* is a PDA in which the maximum number of symbols in the stack is bounded by a constant. Formally, a constant stack PDA has one more element in the tuple, say c , that represents the maximum number of symbols allowed in the stack. During computation, if the stack becomes taller than c at any time, the machine rejects immediately. Consider the class of languages, CSPDA, consisting of the all of the languages that are recognized by constant stack PDAs.

1. Would this class, CSPDA, be strictly smaller than the class of CFLs? That is, would there be languages that constant stack PDAs cannot recognize that regular PDAs can? Prove your claim.
2. Would CSPDA be strictly larger than the class of regular languages? Prove your claim.

Solution

CSPDA equals the class of regular languages. Every regular language is in CSPDA, since we can change any DFA into a constant stack PDA that doesn't use the stack at all. Let's show that every language in CSPDA is regular, by showing that any constant stack PDA can be converted into an NFA. The intuition is that when the size of the stack is bounded by a constant c , there are only a constant number of states that the stack can be in (namely $|\Gamma|^c$, where $|\Gamma|$ is the size of the stack alphabet). Since this is only a constant amount of information, we encode this into the states of the NFA.

Let's say we are given a constant stack PDA $P = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F, c)$, where c is the upper bound on the stack size, and assume that it accepts by final state. The NFA we make for this will be $N = (Q', \Sigma, \delta', q'_0, F')$:

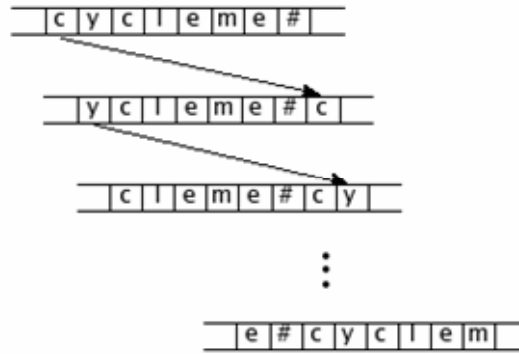
- $Q' = \{ \langle q, s_1, s_2, \dots, s_c \rangle : q \text{ is in } Q \text{ and } s_i \text{ is in } \Gamma \cup \{ \# \} \}$ where $\#$ is a symbol not appearing in Γ . The s_i 's encode the entire stack. The bottom of the stack will be s_1 , and the s_i 's that are "above" the top of the stack will be filled with the $\#$'s.
- $\delta'(\langle q, s_1, s_2, \dots, s_c \rangle, a) = \{ \langle q', s'_1, s'_2, \dots, s'_c \rangle : (q, \gamma) \text{ is in } \delta(q, a, s_t) \text{ where } t \text{ is the largest } i \text{ such that } s_i \text{ is not } \#, \text{ and } s'_i \text{ is equal to } s_i \text{ for most } i, \text{ except that } s'_t s'_{t+1} \dots s'_{t+|\Gamma|-1} = \gamma \}$
- $q'_0 = \langle q_0, Z_0 \# \dots \# \rangle$
- $F' = \{ \langle q, s_1, s_2, \dots, s_c \rangle : q \text{ is in } F \}$

Problem 8

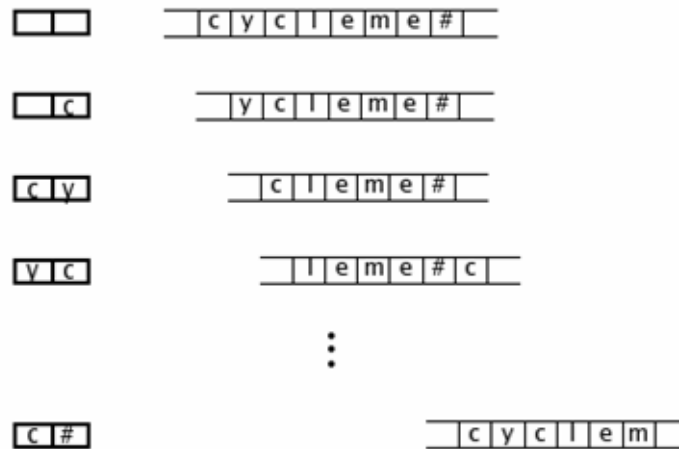
A *queue machine* is a PDA that has a queue instead of a stack. A queue machine can recognize the language $\{ ww : w \text{ is in } \Sigma^* \}$, by enqueueing the beginning part of the input string and then nondeterministically dequeuing it while comparing against the rest of the input. It can also recognize $\{ ww^R : w \text{ is in } \Sigma^* \}$. Explain how. (A hint will be released on the newsgroup on Wednesday)

Solution

The trouble with a queue machine is that after we enqueue a string, we have access to its first character and then its second and so on. What we want is to access the last character in the queue first. Now, the hint given later was "cycle through". We can dequeue a character and then enqueue it. By repeating this, we can cycle through the entire string until we get to the last character.



However, we need to know when to stop. We mark the end of the string with a special marker, say `#`. We know that we just saw the last character when the head of the queue is `#`. Since we can store a finite amount of information in our state, we can store a buffer of two characters. Instead of enqueueing a character right after it was dequeued, we let it linger in a two-character buffer for a while.



If the character that comes right after some character is a `#`, we know that that character was the last character. At this point, we compare the input with this last character. Since we are now done with this character, we decide not to enqueue it back at all, so that we end up with the last character removed. By repeating this "cycle-through" process, we can read the string in the queue in a backwards fashion.

Masini turing and others :

Problem 1

Give a *complete* description of a deterministic TM that computes the function $f: \mathbb{N} \times \mathbb{N} \setminus \{0\} \rightarrow \mathbb{N}$, given by $f(x,y) = \text{floor}(x^{1/y})$. That is, $f(x,y)$ is the y^{th} root of x , rounded down to the closest integer. You may assume that x , y , and your output, are in unary. So, on input $1111111111\#111$, the output is 111, since the cube root of 10 is between 3 and 4.

You should list every state, every transition, and document your construction so it is clear what each state or set of states does. Think top-down.

Solution

One way to find $\text{floor}(x^{1/y})$ is to start with $z=1$, check z^y against x , and continue increasing z by 1 until $z^y > x$. At this point $f(x,y)=z-1$. Calculating z^y on a TM is a matter of calculating z^k where k goes from 1 to y by successive multiplications. Here is one way of doing all this.

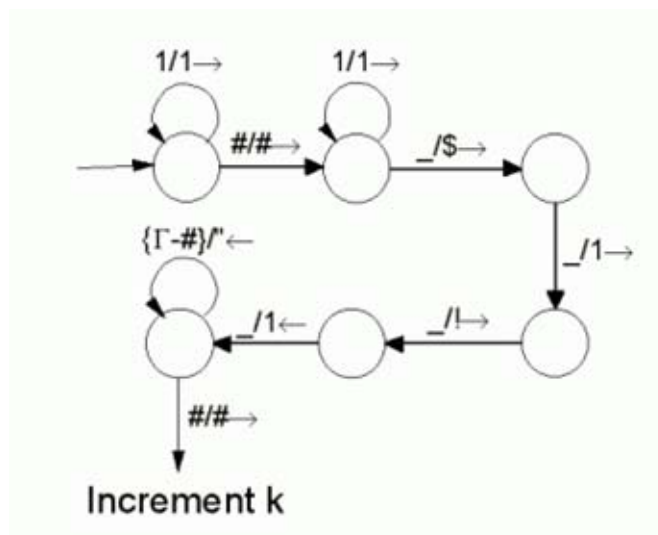
First the general structure:



The Initialize segment sets up the tape in the format $1^x \# 1^y \$ 1^z 1^{z^k}$ that will be used throughout the TM (At the beginning $z=1$ and $k=0$). The Increment k /Multiply/Copy loop calculates $z^k = z * z^{k-1}$. If it exceeds x , then the solution has been found, so the TM jumps to cleanup. If not, the Increment k /Increment z loop starts over with $z+1$. The cleanup portion removes marked and excess symbols, and decrements z , leaving the answer on the tape.

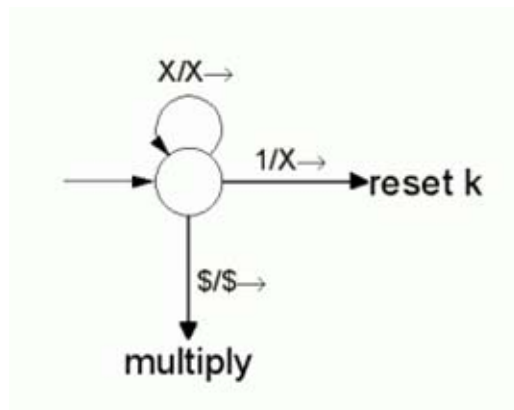
Now to show how each component works. In all of the descriptions of tape contents, the q shows where the head is on the tape. In all the transition diagrams, $S/\text{"} \rightarrow$ means read any symbol in set S , don't write, and move accordingly.

Initialize



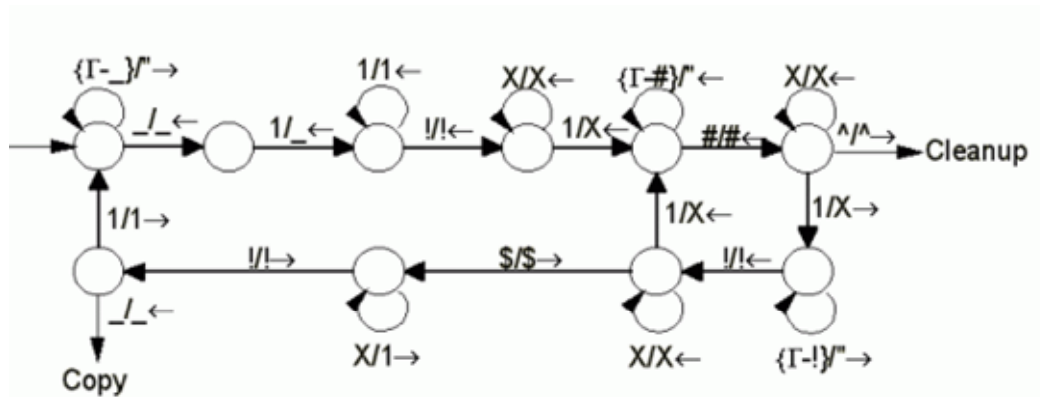
Initialize starts with $q1^x \# 1^y$ and produces $1^x \# q1^y \$ 1^1 1$, which is the desired tape for $z=1$.

Increment k



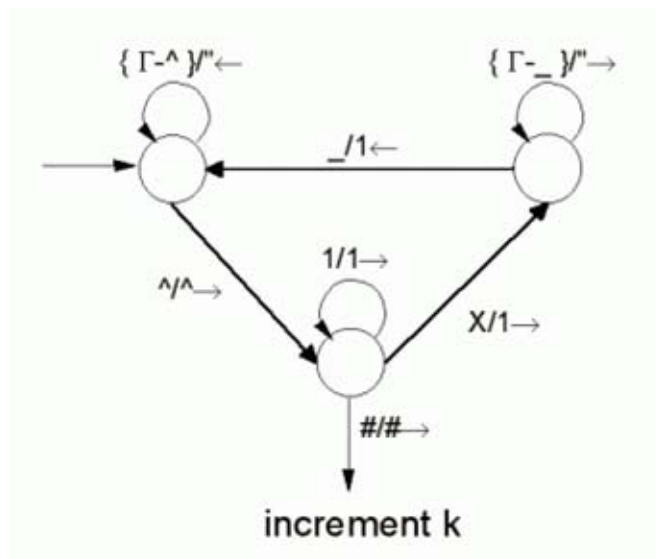
Increment k starts with $1^x \# q 1^{y-k} \$ 1^z 1^{z^k}$ and produces $1^x \# X^{k+1} q 1^{y-k-1} \$ 1^z 1^{z^k}$. If $k=y$, then z^y has been computed and is less than x , so the TM moves to Reset k / Increment z . Otherwise, the TM proceeds to the multiply component.

Multiply



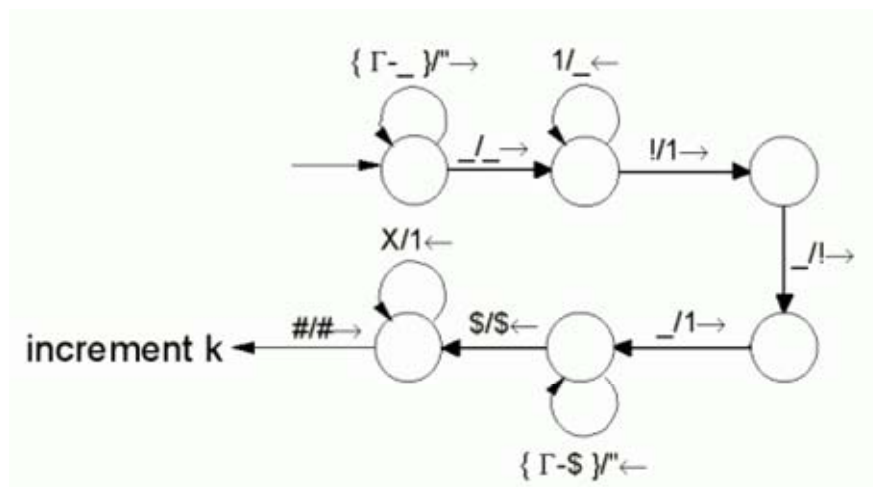
Multiply starts with $1^x \# X^k q 1^{y-k} \$ 1^z 1^{z^{k-1}}$ and produces $X^{z^k} 1^{x-z^k} \# X^k 1^{y-k} \$ 1^z q$. The product z^k is written onto the x component to (1) keep z^k separate from z^{k-1} , and to (2) compare z^k to x . If x is greater, proceed. If z^k is greater, then $z^y > x$, so $f(x,y)=z-1$. In this case Multiply aborts to cleanup. Notice that Multiply erases the right-most operand while multiplying. After the multiplication is completed, the TM moves to the Copy component.

Copy



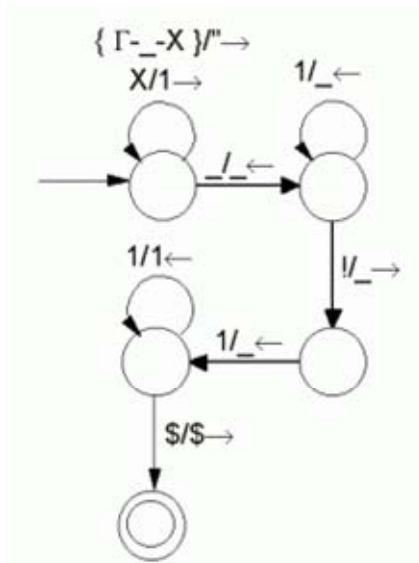
Copy starts with $\wedge X^{zk} 1^{x-zk} \# X^k 1^{y-k} \$ 1^z ! q$ and produces $\wedge 1^x \# q X^k 1^{y-k} \$ 1^z ! 1^{zk}$, which simply copies z^k from the left side to the right side. After doing so, the TM returns to increment k.

Reset k, Increment z



Reset k, Increment z starts with $\wedge 1^x \# X^y \$ q 1^z ! 1^{zy}$ and produces $\wedge 1^x \# q 1^y \$ 1^{z+1} ! 1$. In other words, unmark k, reset z^y , and increment z. Afterwards the TM can return to Increment k and the exponentiation loop.

Cleanup

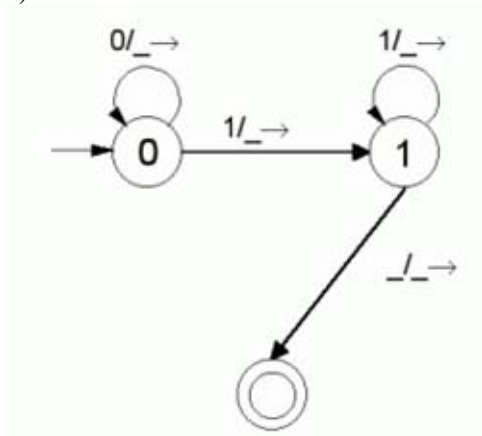


Cleanup starts with $^{\wedge}qX^x\#X^k1^{y-k}\$q1^{z-m}X^m!1^n$ and produces $^{\wedge}q1^x\#1^y\$q1^{z-1}$. Thus reading the tape from here shows the solution z .

Problem 2

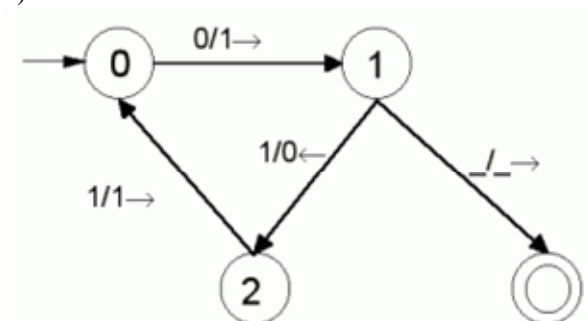
Determine $L(M)$ for the following 2 TMs.

b)



The q_0 loop recognizes an arbitrary string of 0s, the q_1 loop recognizes an arbitrary string of 1s, and the q_0 - q_1 path requires at least one 1. Therefore $L(M)=0^*1^+1$.

c)



From the q_0 - q_1 path, clearly every string must begin with 0, and the string 0 is in $L(M)$. The q_0 - q_1 - q_2 loop turns q_001 into $1q_00$. Therefore, (1) strings in $L(M)$ cannot have a second 0, since

$q_000 \rightarrow 1q_10 \rightarrow \text{fail}$, and (2) all strings of the form 01^* are accepted, since $q_001^n \xrightarrow{*} 1q_01^{n-1} \rightarrow 1^i q_01^{n-i} \xrightarrow{*} 1^n q_00 \rightarrow q_f$. Therefore $L(M) = 01^*$.

Problem 3

Show these modified TMs are equally as powerful as regular TMs: (I.e., show how to simulate any given standard TM M by some M' that meets the restriction given.)

1. A TM that can only write to each space once (writing the input to the tape counts, but you can choose how to write the input to the tape)

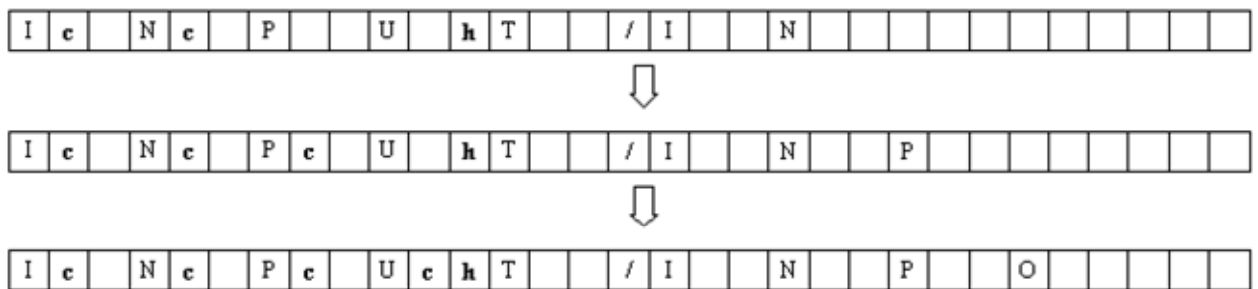
Solution

We can only write to each space once, but we can still use an unbounded amount of space. So, we can simulate each rewrite to a cell using extra space. To make it straightforward, we simply copy the entire tape contents to unused space, using special delimiters to separate the "current" contents. During the copy, we can change whichever cell we wished to overwrite.

The difficulty lies in how to actually copy stuff. Since we are moving the head back and forth between the source and the destination, we have to keep track of our position in the source and destination areas. When we could write as many times as we wanted, we simply kept a "marker" on the last cell that we had copied. We can do something similar by preparing spaces in-between the actual cells and using them to keep the marks.

We need to keep track of up to where we've copied, and also where the head was before we initiated the copy (since we must continue to execute the machine afterwards). Hence, we need to spaces in between each space. The problem says we can decide in which form the input is placed on the tape, so we assume that two spaces are in between each cell. For each two spaces, the first will be the place to put a marker when the cell before it is already copied, and the second will have a marker if that cell had the head on it before the copy started.

Here is an example to illustrate this. The "already copied" markers are **c**'s, and the "head position" marker is **h**. The original tape content was "INPUT", and the head was over the 'U'. We were just about to write 'O' there, and hence the copy was started. The diagram when 'T' and 'N' have just been copied and what happens afterwards is shown below.



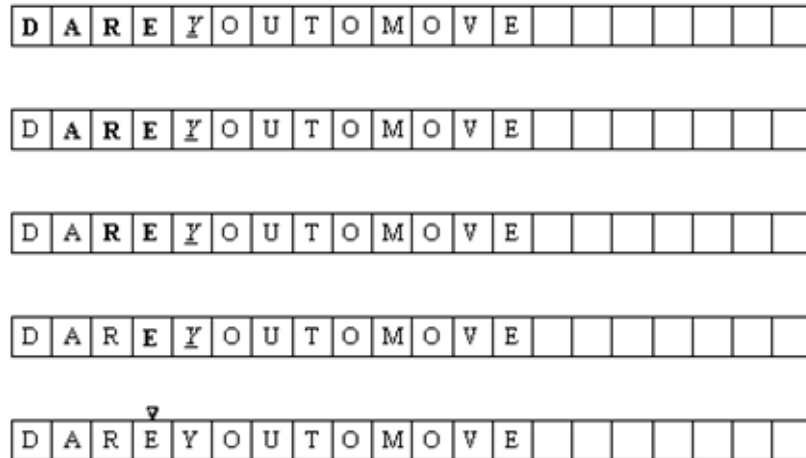
2. A TM that can only move right or reset to the starting position (instead of moving left)

Solution

It's the old-typewriter-machine! Clearly, we can move left by resetting and then moving right by one less space than where we were before. The trouble is figuring out where that one-less-space is.

Before resetting we leave an "old position" mark on our current cell. After we reset, we start to move right and mark all of the cells we are on with a "candidate" mark, until we meet the "old position" mark (which we don't overwrite). Everything to the left of "old position" is a "candidate".

Our objective is to slowly shrink the "candidate" marks towards the right until we are only left with one, which will be the position that's left of the old position. We perform a loop, cycling between two activities: checking that there is still more than one "candidate", and getting rid of the left-most candidate. An illustration is given below, where the characters are bolded to represent the "candidate" marks, and the underlined and italicized character is the one with the "old-position" mark.



We can check that we have more than one candidate by resetting and moving right until we hit the old position. In the finite state, we only need to remember if we had seen one candidate before, and the instant we see another one, we know that we should still continue the loop. Getting rid of the left-most candidate is simply moving right until we hit a candidate, and then we remove the mark.

After the loop is done, we move further right to erase the "old position" marker, and then reset and move right until we see the now-unique "candidate" marker.

Problem 4

Call a TM "left-averse" if it refuses to move left: all transitions either leave the tape head stationary, or move it to the right. Let C be the class of languages accepted by left-averse TMs. (Thus, a language L is in C iff there is a left-averse TM that accepts all and only the strings in L .)

What well-known class of languages is C ? *Prove* that your answer is correct.

Solution

Intuitively, Turing machines are powerful because they have access to an unbounded amount of memory. But, if we can only move in one direction, we cannot access what we wrote in the past. We only have our finite state and the finite amount of information we can store in the current tape cell, so we are as good as a DFA.

More formally, consider a left-averse TM, say M , and the process it takes as it slowly moves rightward on its tape. Sometimes it will move right, sometimes it will stand still, and it may write something on the current tape position. Let A be the instant in time when M first got to cell i , and B be the instant when it first got to cell $i+1$. Let p be the state of M at time A . At time A , M will be

looking at the i -th input symbol. It can linger on the current tape cell until it becomes time B , at which time its finite state will change to, say q . The important thing to notice, is that since M is deterministic, q is a function of p and the i -th input symbol, and nothing else. Between times A and B , M could stay stationary and write to the current tape cell and on then go to some other state based on this new tape symbol it's seeing and then write something new, and so forth. But given what was originally on the tape in the i -th position at time A and what state p is, the state M is going to be in right when it issues a right-move is fixed. Hence, the behavior of M is exactly like a DFA.

Problem 5

If L is a language, define $\text{permute}(L) = \{w : w \text{ is the permutation of some word in } L\}$. Prove or disprove:

1. *The recursive languages are closed under permutation. That is, if L is recursive, then so is $\text{permute}(L)$.*

Solution

For a given input, there are only a finite number of permutations of that string. We can make a Turing machine that generates all of the permutations (somewhat like a n -ary counter), and always halts. We can use that Turing machine to generate all of the permutations on some tape, and run the Turing machine for L on all of them until we find a permutation that is accepted. Since the Turing machine for L always halts, we are guaranteed to never get into an infinite loop.

2. *The recursively enumerable languages are closed under permutation.*

Solution

Call the Turing machine for L , M . We use the same construction as above, except that we cannot sequentially run M on all of the permutations, since M may not halt on one of the permutations. To prevent this, we run multiple copies of M , one for each permutation, in parallel. Notice that we don't even have to do dovetailing here since the number of inputs we have to test is finite. If there is a permutation that is in L , we will eventually find one execution of M that halts, and we will halt.

Problem 6

Define a queue machines as in problem 8 of the last problem set, except that we now have a deterministic version. Prove that (deterministic) queue machines are as powerful as TMs. Hint: see the hint from problem 8 of the last problem set.

Solution

A deterministic queue machine can be shown to be capable of simulating a Turing machine. Let the queue hold the contents of the tape in a circular fashion. The head of the queue will be where the TM tape head is one, and the tail of the queue will be the tape cell to the immediate left. We can write something on the tape and move right by simply dequeuing and enqueueing a new character. We can write something and stay still by dequeuing the current character and then enqueueing the new character we wish to write with a mark on it to specify that it has. We then "cycle-through" until we meet the marker. We can do a "write and move-left" in essentially the same fashion.

Problem 1

Show that $L = \{i : L(M_i) \text{ contains some string that is a palindrome}\}$ is not recursive by reducing either the halting problem or the universal language L_U to L . Do not use Rice's theorem.

Solution

Let's reduce L_U to L . The term "reducing" is always confusing: here, it means that you slightly change the problem of determining strings of L_U to look just like the problem for L . If you do this, you can run the Turing machine for L on this slightly changed problem to determine the answer for the original problem of L_U . The other confusing thing is that less-than-or-equal-to symbol with funny subscripts that we see in the material. In our case, we would write, " L_U is less than L ". This sort-of means that L_U is easier than L . That's why we could change the input for L_U so that the machine for L would solve it.

Any input for L_U looks like " $\langle M \rangle \# w$ ", where $\langle M \rangle$ is a Turing machine encoding, and w is some input we want to feed to M . We have to answer "yes" if and only if M accepts w . On the other hand, let's say we have a Turing machine for L called M_L . From the definition of L , M_L would take some Turing machine encoding and determine if it is possible for it to accept any palindromes at all. To complete the reduction, we have to make it so that M_L says "yes" if and only if M accepts w . In other words, we should modify the input " $\langle M \rangle \# w$ " so that M_L would take the modified input and say "yes" if and only if M accepts w .

When M does not accept w , the modified input should be a machine that does not accept any palindromes. When M accepts w , the modified input should be a machine that accepts at least one palindrome. So, we'll modify the input, by inserting some fixed code and cutting and pasting " $\langle M \rangle$ " and " w ", like this (writing Turing machine code is tedious, so we usually use diagrams or pseudocode instead, which is shown inside double-quotes below):

Change

" $\langle M \rangle \# w$ "

to

"Run M on w . If M accepts w then if the input was 0110, accept. If M doesn't accept w , regardless of what the input was, reject."

So, if M accepts w , the set of strings accepted by the TM code above would be $\{0110\}$. Otherwise, it would be the empty set. Hence if we run the above input through M_L , it would only say "yes" if M accepted w . We're done!

Getting back to the big picture, we've reduced L_U to L , so we have shown that L is even harder than L_U . Since L_U is not recursive, so is L . But you can't write something like this on exams. You have to say: "We have shown that if we had a Turing machine for L , we could solve L_U by this reduction procedure. Hence, L could not have been recursive."

Problem 2

Let L be recursively enumerable but not recursive. Consider the language the union of the two languages, $\{0w : w \text{ is in } L\} \sqcup \{1w : w \text{ is not in } L\}$, and call it L' . Determine if L' or its complement is r.e. or non-r.e. Prove your answer.

Solution

Let's consider how we'd tackle this problem. The first thing you notice is that the latter half of the union, $\{1w : w \text{ is not in } L\}$, is not recursively enumerable. This is because L is not recursive (though recursively enumerable) and $\{1w : w \text{ is not in } L\}$ is simply the complement of L with a '1' stuck in front. Clearly, sticking a '1' in front wouldn't make a non-r.e. problem become easy.

On the other hand, $\{0w : w \text{ is in } L\}$ is simply L with 0's stuck in front of every string, so it's recursively enumerable. L' is the union of a non-r.e. language and an r.e. language. What can we know from just this fact? Nothing!

As noted on the newsgroup, unioning two different classes of languages doesn't tell us much about the result of the union. For example, we could take A to be the set of all strings, which is clearly r.e., and the diagonal language, which is non-r.e., and union them together. The result would be the set of all strings, which is r.e. Here, we're unioning a complex language with a simple one, and we get a simple one as a result. But, please note that this kind of intuition is for your enrichment only and should not be used as an answer in exams.

However, notice that this case is different. The two sets, $\{0w : w \text{ is in } L\}$ and $\{1w : w \text{ is not in } L\}$ are disjoint. It is a good rule of thumb to assume that when the union is disjoint, the complexity of the union is the maximum of the parts.

Now, for a contradiction, let's try assuming that L' is recursively enumerable. Hence, there is a machine, say M' , that accepts L' . Then, given some input x , we can determine if x is in the complement of L by feeding " $1x$ " into M' and waiting for it to say "yes". However, L was not recursive, so creating a procedure for accepting the complement of L must not be possible, and we have a contradiction.

Now let's consider the complement of L' :

This is essentially L' with just the 0 prefix and 1 prefix interchanged, so the complement of L' is also not recursively enumerable.

Problem 3

Recall that in a normal substitution, for every symbol a in the alphabet, we have an associated language L_a from which we can choose any string to replace a . In an epsilon-free substitution, none of the L_a 's have the empty string in them. So, we never replace a symbol with the empty string.

Prove that the recursive languages are closed under epsilon-free substitution and Kleene star. Consider only substitutions where all the L_a 's are recursive. Does the same hold for recursively enumerable languages? Again, you may consider only substitutions where all the L_a 's are recursive (the original problem, however, would be when all the L_a 's are only recursively enumerable). Prove your answers.

Solution

Let's consider the case for epsilon-free substitution on recursive languages. Say that we have a recursive language L , and we perform a certain substitution on it. Say that for each symbol a in the alphabet, we have L_a as the language to replace it with. Call the machine that decides each respective L_a , M_a . Call the machine that decides L , M_L . We can construct a machine, say M , to decide the set of strings formed after substitution performed over strings of L , as follows:

1. M nondeterministically breaks up the input into any number of pieces, say $x_1, x_2, \dots, x_k$.
2. For each piece, x_i , M nondeterministically chooses an L_a it might be included in. To confirm its guess, M runs M_a on x_i , and sees if it is accepted. If each piece has been assigned a plausible L_a , we go to the next step. Before we do so, we make sure we have all of the guessed a 's on a separate tape, which becomes the pre-image string (the string that we are guessing to have been the original string before substitution). This process is guaranteed to halt since each M_a always halts.
3. We run M_L on the pre-image string, which is guaranteed to halt since L is recursive. We accept if and only if M_L accepts.

The construction for recursively enumerable languages is exactly the same if you assumed that each L_a was recursive. If you assumed that the L_a 's are only r.e., then in step 2, you have to do dovetailing.

The Kleene closure construction is also very similar. We non-deterministically chop the input up into parts. We run each part on the machine for L . If any of the parts are empty strings, we accept that part without running the machine for L . Note that we don't have to dovetail stage 2 even when L is r.e., since we are only concerned that the procedure halts when all of the pieces are confirmed to be in L .

Problem 4

Part a)

Prove that the r.e. languages are not closed under infinite union.

Solution

Take the diagonal language, L_D , which isn't r.e. L_D is still a language, so it consists of strings, say $\{x_1, x_2, \dots\}$. We can make each individual string into trivial languages that each have one string in them: $\{x_1\}$, $\{x_2\}$, ... and so forth. Each of these languages are r.e. (since they're regular). The union of all these languages form L_D .

Part b)

Suppose that $a_1, a_2, \dots$ are indices of Turing machines. Suppose that the set $\{a_1, a_2, \dots\}$ is recursively enumerable. Show that the infinite union of all the languages $L(M_{a_1}), L(M_{a_2}), \dots$ is recursively enumerable. Thus, the r.e. Languages are closed under "r.e. Union".

Solution

Create a TM, say M , that accepts this union. Given input x , M needs to see if any of the M_{a_i} 's accept x . Since $\{a_1, a_2, \dots\}$ is r.e., there is a generator, say G , that generates all of them. M runs G , and runs each generated machine, M_{a_i} , on x . Since some M_{a_i} might not halt, we dovetail the executions.

Problem 5

We can define languages as subsets of natural numbers. Let the language denoted by the regular expression $0 + 1(0+1)^$ be the set of all (binary encodings of) the natural numbers N . Then if L is a subset of N , we say L is recursive (respectively, r.e.), if the set of binary strings that encode elements of L is recursive (respectively, r.e.).*

Recall that a total recursive function (from N to N) is one that is computable by some Turing machine that is defined on all elements of N , and that a partial recursive function is one that is computable by some Turing machine M , but M may not be defined on all natural numbers. Prove that the following are equivalent:

- 1. L is recursively enumerable*
- 2. L is the range of some partial recursive function*
- 3. L is the domain of some partial recursive function*

Solution

For convenience, we have numbered the properties we need to show equivalent.

For "1 implies 3", we assume 1 is true. Then there is a TM that recognizes L -- so let's call it M . We just change M so that when it is about to accept, it outputs "yes, ja, oui" on the output tape and halts. Also, when M is just about to reject, we put it in an infinite loop so it doesn't halt. Then 3 is satisfied, since this new machine implements a partial recursive function that is only defined when the input is in L .

For 3 implies 1, assume 3 is true. Let's call the TM that computes the partial recursive function, F . To satisfy 1, our goal is to construct a TM that recognizes exactly those strings that are in the domain of F , i.e. those strings that will let F halt. We can simply change F so that when it halts, it also moves to the accept state.

For 1 implies 2, assume 1 is true. We have a TM, say M again, that recognizes L . The goal is to make L the range of some partial recursive function. We name the TM that will compute the function we are about to make, F . We make F take an input - call it x -, interpret it as a natural number, and use M to generate the x 'th string in L (we did this in class). This way, all of the outputs of F would consist of all of the strings in L .

For 2 implies 1, assume 2 to be true. There is a partial recursive function - call it F - whose range is L . Our goal is to show that L is recursively enumerable, and hence we should build a TM that recognizes it. Let's call our goal TM, M . To meet the goal, given any string x as input, M must correctly determine if it is in the range of F . We can do this by feeding all possible input strings in lexicographic order to F , and examining its output for each input. Since some executions of F might not halt, we dovetail its execution.

Altele:

Problem 1

Use the pumping lemma to prove that $L = \{a^i b^j c^k \mid i \leq j \leq k\}$ is not a context free language.

Solution

For pumping length n pick $z = a^n b^{n+1} c^{n+2} = uvwx \in L$. There are 7 different cases for dividing z into u, v, w, x , and y such that $|vx| > 0$ and $|vwx| \leq n$:

1. Either v or x contains multiple character types, e.g. $w = ab$. In this case $uv^2wx^2y \notin \{a^i b^j c^k \mid i, j, k \geq 0\}$, so $uv^2wx^2y \notin L$.
2. $vwx = a^i b^j c^k$. Since there are $n+1$ b's in the center, $|vwx| > n$, so this is invalid.
3. $vwx = a^j$, where $1 \leq j \leq n$. Set $k = |v| + |x| \geq 1$. $uv^2wx^2y = a^{n+k} b^{n+1} c^{n+2} \notin L$, since $n+k \geq n+1$.
4. $vwx = b^j$, where $1 \leq j \leq n+1$. Set $k = |v| + |x| \geq 1$. $uv^2wx^2y = a^n b^{n+k+1} c^{n+2} \notin L$, since $n+k+1 \geq n+2$.
5. $vwx = c^j$, where $1 \leq j \leq n+2$. Set $k = |v| + |x| \geq 1$. $uv^0wx^0y = uwy = a^n b^{n+1} c^{n+2-k} \notin L$, since $n+1 \geq n+2-k$.
6. $vwx = a^m b^p$, where $2 \leq m+p \leq 2n+1$. Therefore $v = a^j$, $w = b^k$, where $1 \leq j \leq n, 1 \leq k \leq n+1$.
 $uv^2wx^2y = a^{n+j} b^{n+k+1} c^{n+2} \notin L$, since $n+k+1 \geq n+1$.
7. $vwx = b^m c^p$, where $2 \leq m+p \leq 2n+3$. Therefore $v = b^j$, $w = c^k$, where $1 \leq j \leq n+1, 1 \leq k \leq n+2$.
 $uv^0wx^0y = uwy = a^n b^{n-j+1} c^{n-k+2} \notin L$, since $n \geq n-j+1$.

Therefore z cannot be pumped, so L is not context free.

Problem 2

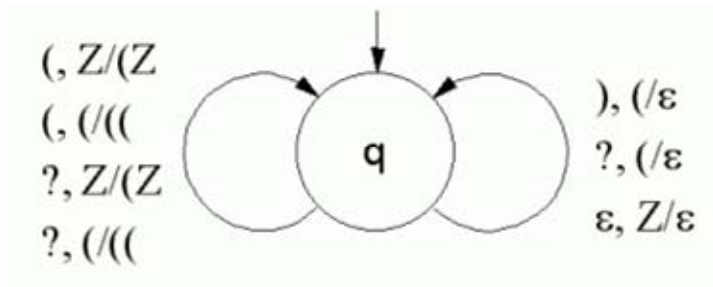
$smudgy-parens = \{w \in \{(),?\}^* \mid \text{There is a way to replace each "?" in } w \text{ by "(" or ")" to produce a string of balanced parentheses}\}$

Part a) Create a PDA that accepts $smudgy-parens$

Solution

Start with a PDA that accepts balanced parentheses. One such PDA pushes every "(" in the string, and pops a "(" every time it sees a ")". If it sees a ")" without there being a "(" on the stack, it would have no transitions, and would thus fail, as this would be an unbalanced string. Such a PDA would accept with empty stack.

To convert this PDA to a PDA for $smudgy-parens$, add transitions for "?" that treat each "?" nondeterministically as both "(" and ")". Thus for every "?" there should be transitions that push "(", and transitions that pop "(". Therefore if there exists a way to substitute parentheses for "?"s in w , then there is a corresponding route in the PDA that processes the entire string and yields an empty stack:



Here Z is the botom-of-stack marker.

Part b) Give a context-free grammar for *smudgy-parens*

Solution

The grammar for balanced parentheses is $S \rightarrow (S)S | \epsilon$. This comes from the grammar for strings with an equal number of "("s as ")"s, with the added restriction that the "(" comes first in each matching pair. To build a grammar for *smudgy-parens*, replace some of the parentheses with "?"s in every way possible:

$$S \rightarrow (S)S \mid ?S)S \mid (S?S \mid ?S?S \mid \epsilon$$