CS 361 Meeting 9 — 2/28/20

Announcements

1. Homework 3 due today.

2. Homework 4 available soon.

3. Click on the footnote at the bottom left of this page to see the slides used in class.

Not Regular $\neq$ Irregular

1. In our last meeting, we obtained two different regular expressions by performing two different sequences of state eliminations to convert a DFA for the language of binary number divisible by 3 to a 2-state GNFA. Eliminating 2 then 1 then 0 and m gave us:

   $$(0 \cup (1(01^*0)^*1))(0 \cup (1(01^*0)^*1))^*$$

   Eliminating 1 then 2 then 0 and m gave us:

   $$(0 \cup 11 \cup 10(1 \cup 00)^*01)(0 \cup 11 \cup 10(1 \cup 00)^*01)^*$$

   Hopefully, these two regular expressions describe the same sets!

2. I suggested that it would be nice to have some sort of "regular expression checker" that would tell us for sure that two regular expressions like this actually do describe the same languages.

3. If you think about it, you will realize that what I really wanted is a **decider** for the language:

   $$L_{EQ-RE} = \{e = e' \mid e \,\&\, e' \text{ are regular expressions over } \Sigma \text{ and } L(e) = L(e')\}$$

4. This language is a bit more interesting than most of the examples we have been talking about so far this semester. Certainly, it would be harder for you to write a program that decided whether an input belonged to this language than it would be to decide if a binary string represented a number divisible by 3.

5. In fact, one of our goals for today will be to show that

   $$L_{EQ-RE} = \{e = e' \mid e \,\&\, e' \text{ are regular expressions over } \Sigma \text{ and } L(e) = L(e')\}$$

   is not regular.[1]

6. To learn how to accomplish this, let's start with something easier. In your last homework assignment I mentioned that $\{a + b = c \mid a, b, c \in \{0,1\}^*$ and the sum of the numbers represented by a and b in binary notation is the number represented by c $\}$ was not regular. Let's consider an even simpler representation of addition:

   > $\{a + b = c \mid a, b, c \in \{1\}^*$ and the sum of the numbers represented by a and b in *unary* notation is the number represented by c $\}$

7. That is, we would like to determine whether the language

   $$L_{UnaryAdd} = \{1^a + 1^b = 1^c | 1^k \text{ refers to a string of } k \text{ 1s and } a + b = c\}$$

   is regular.

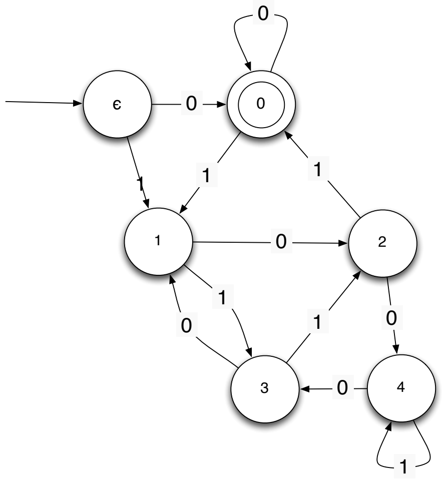
Getting Loopy

1. One approach to showing that a particular language is not regular involves recognizing that strings of sufficient length will encounter loops of states as they are processed by a DFA. I want to make this notion very concrete for you before using it in a more abstract way to show languages are not regular.

   - Last time we explored a number of loops that exist in the state diagram of a DFA that recognizes binary numbers divisible by 5.

Click here to view the slides for this class

[1] Note: It is standard to say a language is "not regular" rather than "irregular." If you use the term "irregular" I may snicker.



- We found examples of loops of strings that led the machine through loops of length 1 to 4, but nothing longer:

input	path
0**0**	$\epsilon \to \mathbf{0} \to \mathbf{0}$
1**10**01	$\epsilon \to \mathbf{1} \to \mathbf{3} \to \mathbf{1} \to 0$
1**011**01	$\epsilon \to \mathbf{1} \to \mathbf{2} \to \mathbf{0} \to \mathbf{1} \to 2 \to 0$
1**1111**01	$\epsilon \to \mathbf{1} \to \mathbf{3} \to \mathbf{2} \to \mathbf{0} \to \mathbf{1} \to 2 \to 0$

- The interesting thing about the sorts of loops in the state diagram we have been looking at is that they correspond to substrings in the input that can be repeated (or omitted) without changing the final state of the corresponding path. This means from each of our examples, we can generate an infinite set of inputs that lead to the same final state by just "starring" the input subsequence that leads the machine through the loop. The following table show how this would be done for the examples discussed above.

input	related strings
0**0**	0**0***
1**10**01	1(**10**)*01
1**011**01	1(**011**)*01
1**1111**01	1(**1111**)*01

- An alternate question we can ask about this machine is if there are inputs of various lengths that would cause the machine to visit states without encountering any loop (i.e., any repeated states).
- Here are some examples:

input	path
0	$\epsilon \to 1$
11	$\epsilon \to 1 \to 3$
111	$\epsilon \to 1 \to 3 \to 2$
1111	$\epsilon \to 1 \to 3 \to 2 \to 0$
01110	$\epsilon \to 0 \to 1 \to 3 \to 2 \to 0$

- Note that the last example visits every state in the machine. That means that any longer input sequence must visit more states than there are in the machine. Thus, for 6, 7 and any larger length inputs, a loop in the states must occur.

Addition is too Hard to be Regular

1. I suggested that to make progress on thinking about how to show a language was not regular it would be best to start with a language that was very simple:

   $$L_{UnaryAdd} = \{1^a + 1^b = 1^c | 1^k \text{ refers to a string of } k \text{ 1s and } a + b = c\}$$

2. We just saw that if a DFA has n states then it must encounter a loop in its state graph when processing any input of length greater than N. We can use this property to see that $L_{UnaryAdd}$ is not regular.

3. If this language were regular, then there would be some DFA M such that $L_{UnaryAdd} = L(M)$. if $M = (Q, \Sigma, \delta, s, F)$, and $n = |Q|$ then the input $1^{2n} + 1 = 1^{2n+1}$:

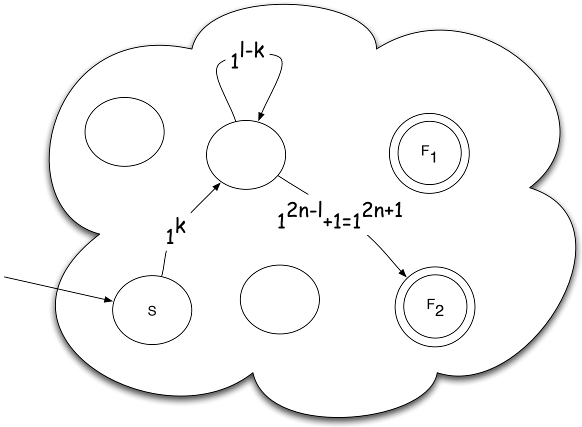
   a) must lead to a final state in M, and

   b) must lead M through at least one cycle in its set of states (since its length requires more states be visited than there are distinct states) before the machine even reaches the plus sign.

4. In case the notation $1^{2n} + 1 = 1^{2n+1}$ isn't sufficiently illuminating, we can show the structure of the input we have in mind as:

input	$111\ldots1111$	$+1 =$	$111\ldots1111$
length	$2n$	3	$2n+1$

5. The "cloudy" diagram below gives some names to the "dimensions" of the loop we know the machine must encounter. In particular we are assuming that the first loop begins after k 1's have been read and that it is completed after l steps implying that the number of 1s in the input that drive the machine around the loop is $l - k > 0$.



6. This suggests another way we can partition the input:

input	$11\ldots11$	$11\ldots11$	$11\ldots11$	$+1 =$	$111\ldots111$
length	k	$l-k$	$2n-l$	3	$2n+1$

7. Now, the key observation is that since the string of $l-k$ ones leads the machine M through a loop, we can describe an infinite set of strings that must belong to $L(M)$ by "starring" this sequence of 1's.

input	$11\ldots11$	$(11\ldots11)^*$	$11\ldots11$	$+1 =$	$111\ldots111$
length	k	$l-k$	$2n-l$	3	$2n+1$

8. Since the closure (*) means we can either repeat or delete (i.e., repeat 0 times) a substring, this would allow us to conclude that the string

input	$11\ldots11$	$11\ldots11$	$+1 =$	$111\ldots111$
length	k	$2n-l$	3	$2n+1$

which is just

input	$11\ldots11$	$+1 =$	$111\ldots111$
length	$2n-(l-k)$	3	$2n+1$

which is $1^{2n-(l-k)} + 1 = 1^{2n+1} \in L(M)$. This, however, should only be true if $2n - (l-k) + 1 = 2n + 1$ which would imply $l - k = 0$ contrary to our conclusion that there must be a cycle of length 1 or greater in the path followed processing a string longer than the number of states in the machine M.

9. This contradiction allows us to conclude that our assumption that $L(M) = L_{UnaryAdd}$ was false, so $L_{UnaryAdd}$ must not be regular.

The Pumping Lemma (for regular languages)

1. We can generalize the partitioning we performed on $1^{2n} + 1 = 1^{2n+1}$ in a way that leads to an understanding of a more general result that can be used to show that certain languages are not regular.

   - The key to our discussion of $1^{2n} + 1 = 1^{2n+1}$ was the subsequence of 1s in 1^{2n} that could be repeated. That is, there were really three key parts to our partition as shown below:

input	$11\ldots11$	$(11\ldots11)^*$	$11\ldots11+1=111\ldots111$
length	k	$l-k$	$4n-l+4$

   - Better yet, rather than counting all the digits so carefully, we can just name subparts as in:

input	$11\ldots11$	$(11\ldots11)^*$	$11\ldots11+1=111\ldots111$
name	x	y	z

- Now, we can summarize the logic behind our argument by saying that for any string w that is long enough, we must be able to write $w = xyz$ in such a way that y corresponds to a string that leads M through a loop and therefore, it must be the case that $xy^*z \subset L$ (or equivalently $xy^iz \in L, i \geq 0$).

2. This generalization of our approach to $L_{UnaryAdd}$ is encapsulated in the FAMOUS Pumping Lemma:

   > **Lemma:** Suppose L is a regular language. Then there exists a positive integer p such that any string $s \in L$ with length at least p may be partitioned into $s = xyz$ where
   >
   > (a) $|y| > 0$
   >
   > (b) $|xy| \leq p$
   >
   > (c) $xy^iz \in L$, for all $i \geq 0$.

3. It may at times be useful to use a slightly different statement of this result:

   > **Lemma:** Suppose L is a regular language. Then there exists a positive integer p such that any string $s \in L$ with length at least p may be partitioned into $s = xyz$ where
   >
   > (a) $|y| > 0$
   >
   > (b) $|xy| \leq p$
   >
   > (c) $xy^*z \subset L$.

4. The proof of this lemma is in the text. For not, we will just assume the lemma is true and work to make sure we know how to use it to show that languages are not regular.

Pumping Iron

1. I started with unary addition because I had previously mentioned that at least one language encoding addition (binary addition) was not regular in a homework assignment.

2. As our first exercise with the Pumping Lemma, let's consider an even simpler language.

3. Let's show that $L_{EQ} = \{1^n = 1^n \mid n \geq 0\}$ is not regular.

   - Using the Pumping Lemma, all we need to do is given a possible p for this language, show how to find a string that cannot be pumped (i.e., a string that turns into strings that don't belong in L_{EQ} when pumped).
   - Consider $1^p = 1^p$.
   - The pumping lemma requires that we be able to remove or duplicate some substring y which is non-empty ($|y| > 0$) that appears within the first p symbols ($|xy| \leq p$) of $1^p = 1^p$.
   - Any prefix xy of $1^p = 1^p$ of length at most p must contain only 1s. So, $x = 1^k$ and $y = 1^l$ for some l and k such that $l > 0$ and $l + k \leq p$.
   - The Pumping lemma allows us to conclude that if L_{EQ} is regular all strings of the form $1^k(1^l)^*1^{p-(l+k)} = 1^p \in L_{EQ}$. However, $1^k1^{p-(l+k)} = 1^p \notin L_{EQ}$.
   - Therefore, L_{EQ} must not be regulsr!

4. This brings us back to the example I started with to remind you that decision problems can be interesting.

   $$\begin{aligned} &L_{EQ-RE} = \\ &\quad \{e = e' \mid e \;\&\; e' \text{ are regular expressions over } \Sigma \text{ and } L(e) = L(e')\} \end{aligned}$$

   - At first this may seem like a very complicated example.
   - Remember, however, that $\{1^n = 1^n \mid n \geq 0\} \subset L_{EQ-RE}$.
   - Using closure properties, we can often focus on such a sublanguage to prove a language that contains it is not regular.
   - In this case, consider the language $1^* = 1^*$ (which is clearly regular since we used a regular expression to describe it).
   - The intersection of L_{EQ-RE} with $1^* = 1^*$ is just L_{EQ}.
   - If L_{EQ-RE} was regular, then since regular languages are closed under intersection, L_{EQ} would have to be regular.
   - We just showed, however, that L_{EQ} is not regular.

- So, L_{EQ-RE} must not be regular.
- The lesson is that you should not work too hard. You can often avoid the complication of an argument involving the Pumping Lemma by taking advantage of closure properties.

Thinking Negative Thoughts

1. One odd thing about the Pumping Lemma is that we almost never encounter an example where we use it in a positive sense. That is, instead of saying, "Yay! We know this language is regular so it must satisfy the Pumping Lemma," we almost always use it in proof's by contradiction where we instead say "Yikes! This language does not satisfy the Pumping Lemma so it must not be regular."[2]

2. As a result, while it is common to state the Lemma's conditions positively, we will normally be trying to establish that the negation of its conditions are true. So, it is worth carefully considering/understanding what happens when we negate its conditions.

   - The Pumping Lemma can be restated (and reformatted) just slightly to read:

     Lemma: Suppose L is a regular language. Then
     there exists a positive integer p such that
     for all string $s \in L$ with length at least p
     there exists a partition $s = xyz$ where
     (a) $|y| > 0$
     (b) $|xy| \leq p$, such that
     for all $i \geq 0$, $xy^iz \in L$.

   - The point of this reformatting is to emphasize that the statement of the lemma involves four "quantifiers". Two of the quantifiers are "for alls" and two of the quantifiers are "there exists".

   - In mathematical logic, the symbol $\forall$ is used for "for all" and $\exists$ is use for "there exists". Therefore, a very abbreviated (actually slightly incomplete) form of the Pumping Lemma would be

     $$\exists p \forall s \exists xyz \forall i xy^iz \in L$$

   - To negate a quantified statement you reverse the quantifier ("for all" become "there exists" and vice versa" and negate the statement that the quantifier was applied to.

     Applied to the Pumping Lemma this becomes would be

     $$\forall p \exists s \forall xyz \exists i xy^iz \notin L$$

   - Restated in English we can say that to show that L is not regular you must show that
     - **for every** sufficiently large p
     - **there exists** a string $s \in L$ of length $\geq p$ such that
     - **for every** possible partition of $s = xyz$ where
       (a) $|y| > 0$
       (b) $|xy| \leq p$
     - **there exists** $i \geq 0$, such that $xy^iz \notin L$.

   - This restatement reveals three things you need to recognize to use the Pumping Lemma effectively.
     - You cannot assume anything about the size of p.

       As a result, rather than looking at specific strings, we look for patterns (like $1^p = 1^p \in L_{EQ-RE}$ rather than $1^{967} = 1^{967}$).
     - You only need to find one string you cannot pump!

       This is a blessing and a curse. It would clearly be more work if you had to show that every string cannot be pumped. In general, however, a language that is not regular may contain many strings that can be pumped so finding a pattern that cannot be pumped may feel like looking for the proverbial needle in a haystack.

[2] It is worth noting here, although I hope to emphasize this later, that the Pumping Lemma is not an "if and only if" result. As a result, we **never** use the Pumping Lemma to arrive at the conclusion "Yay! This language satisfies the Pumping Lemma so it must be regular." In fact, there are languages that satisfy the Pumping Lemma but are not regular.

- You need to explore every possible way of partitioning strings matching your pattern.

  This is definitely bad news. It can be difficult to be sure you have covered every possibility.
- You only have to find a single value of i that doesn't work.

  In the easy examples we have considered, almost all values of i don't work. Things actually can be tricky if there is only one value of i that doesn't work and you have to find it.
- The pumped string xy^iz must not be in the language.

  It isn't enough to show that the pumped string is no longer of the pattern you were thinking of.

3. As a simple illustration of these ideas, consider the language:

$$L_{Binary-addition} = \{a + b = c | a, b, c \in \{0, 1\}^*, v(a) + v(b) = v(c)\}$$

where $v(x)$ = the value of x interpreted as a binary number.

- At first this probably seems like it will be harder than the unary addition example because binary place notation just seems more complicated that unary which depends only on the length of strings not their content.
- Recall, however, that we don't have to show that it is impossible to pump all strings (in fact, this is unlikely to be true). We only have to show that one pattern (including strings of arbitrary length since we don't know how big p might be) cannot be pumped.
- In this case, consider the pattern $1^p + 0 = 1^p$ where we assume p is the pumping length.
  - In any partition $1^p+0 = 1^p$ into xyz where $|xy| \leq p$, all of the symbols in x and y must correspond to substrings from the first set of p 1s. Therefore, any string xy^iz corresponding to a value of i other than 1, will have the form $1^k + 0 = 1^p$ with $k \neq p$ and such strings are not elements of $L_{Binary-addition}$.
  - This violates the Pumping Lemma, so we can conclude that $L_{Binary-addition}$ must not be regular.