CS 361 Meeting 16 — 4/6/20

Announcements

(Click for video)

1. First, welcome back to CS 361 in online form. We are all facing the challenge of trying to find the *new normal* in the midst of a world that is very far from normal. I am writing this two weeks before you are supposed to read it and/or view the video of this "class". I hope things are better by then! I fear they may be worse. In any case, I am sure that we are all still facing challenges that will make the task of completing this course and other school work more difficult than we fully appreciate. With this in mind, I want to encourage you to remember that we all need to support one another. Don't hesitate to reach out to me if you need extra help, extra time, or just want to talk about things. At the same time, be patient with me (and all the other students and faculty you may be working with this semester). We are all in the same boat.
2. I have already tried to talk about issues of the mechanics of the course in announcements I have sent to you through Glow. Let me know if anything is unclear.
3. There will be no midterm this semester. We will still have a final exam.
4. Homework assignment 6 is available. It is due 24 hours before your group meeting time.

Welcome Back (sort of)

(Click for video)

1. I want to help you refresh your foggy memories of what we were doing back in March just before the virus changed all of our lives.
2. We had completed our discussion of regular languages. This had included:
    - Examining several formalisms for describing regular languages.
        - Deterministic Finite Automata
        - Non-deterministic Finite Automata
        - Regular Expressions
    - Proving that these ways to describe languages were of equivalent power.
        - The subset construction to build a DFA equivalent to an NFA.
        - GNFAs and an algorithm for deriving a regular expression given a DFA.
    - Showing that there were languages that were not regular.
        - The Pumping Lemma
        - The Myhill-Nerode Theorem
3. We were beginning to study another, larger class of languages called the context-free languages. Our goal is to hit the same three bullet points I just listed for regular languages:
    - Examine several formalisms for describing context-free languages.
    - Prove that these formalisms are of equivalent power.
    - Show that there are languages that are not context-free.

    In addition, to appreciate that the mechanisms used to describe context-free languages are more powerful than those we studied before, we want to show that there are languages that are not regular but are context-free.
4. The first step in our exploration of context-free languages was to introduce the notion of a context-free grammar in several forms.
    - We first considered an example of a grammar for a fragment of a typical programming language:

      < stmt > → < var > = < expr >
      | **if** (<expr>) <stmt>
      | **if** (<expr>) <stmt> **else** <stmt>
      | **while** (<expr>) <stmt>
      <expr> → <var>
      <var> → x | y | z

Click here to view the slides for this class

- We learned to understand such a grammar as a set of rules for replacing symbols in angle brackets with the right-hand side of an appropriate rule to form derivations that determined the set of strings described by the grammar:

  < stmt > ⟹ if (<expr>) <stmt>
  ⟹ if (<var>) <stmt>
  ⟹ if (x) <stmt>
  ⟹ if (x) < var > = < expr >
  ⟹ if (x) y = < expr >
  ⟹ if (x) y = < var >
  ⟹ if (x) y = z

5. We quickly moved away from the world of grammars that actually describe useful things like programming languages to grammars for the sorts of silly languages of a's, b's, #'s, 0's and 1's that we had seen repeatedly when working with regular languages. In particular, we saw a grammar for binary strings with even parity and one for binary strings representing multiples of 3.

   $M \rightarrow 0Z \mid 1U$
   $Z \rightarrow 0Z \mid 1U \mid \epsilon$
   $U \rightarrow 0D \mid 1Z$
   $D \rightarrow 0U \mid 1D$

6. Seeing that these examples of languages known to be regular could also be described by context-free grammars raised a question. Can all regular languages be described by context-free grammars? To pursue a question like this, we needed to be more precise about how grammars worked, so we gave a formal definition:

   Context-free Grammar A context free grammar is composed of:

   (a) A finite alphabet V of variables (or non-terminals).
   (b) A distinct finite alphabet Σ called the terminals or terminal symbols.
   (c) $S \in V$ referred to as the start symbol.
   (d) A finite set R of pairs composed of one element from V and one element from $(V \cup \Sigma)^*$ called the rules. Rules are usually written in the form:

   $$A \rightarrow X_1 X_2 ... X_m$$

   rather than $(A, X_1 X_2 ... X_m)$.

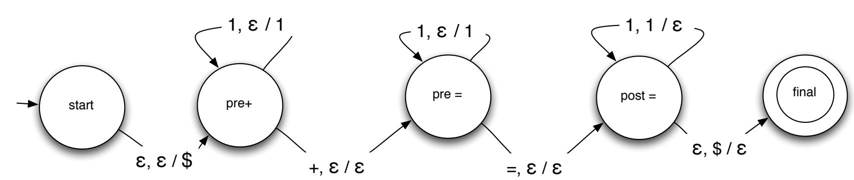
7. With this definition (and definitions of derivations, sentential forms and sentences) we reached the point where we could proved that all regular languages are context-free.

8. We next considered another mechanism for describing languages which we will ultimately show to be equivalent to context-free grammars — the pushdown automaton or PDA. A PDA is basically a finite automaton extended by giving it a stack onto which it can push symbols and whose contents can help determine the transitions the machine makes.

9. We considered a few simple examples of PDA's including one for the language

   $$L_{add} = \{1^i + 1^j = 1^{i+j} \mid i, j \geq 0\}$$

   of simple examples of valid unary addition like "111+11=11111".



10. We then formalized what we meant when we talked about a PDA and its operation with two definitions:

    (a) A *pushdown automaton* is a 6-tuple $(Q, \Sigma, \Gamma, \delta, q_0, F)$ where:

    Q is a finite set of states,
    Σ is a finite input alphabet,
    Γ is a finite stack alphabet,
    $\delta : Q \times \Sigma_\epsilon \times \Gamma_\epsilon \rightarrow \mathcal{P}(Q \times \Gamma_\epsilon)$ is the transition function, and
    $F \subset Q$ is the set of final or accepting states.

(b) We say that a PDA $M = (Q, \Sigma, \Gamma, \delta, q_0, F)$ accepts a string $w = w_1 w_1 ... w_n, w_i \in \Sigma_\epsilon$ if $\exists q_1, ...q_n \in Q$ and $s_0, s_1, ...s_n \in \Gamma^*$ such that:

- $s_0 = \epsilon$
- $\forall i, 1 \leq i \leq n, \exists h_i, p_i \in \Gamma_\epsilon$ and $t_i \in \Gamma*$ such that $s_{i-1} = h_i t_i, s_i = p_i t_i$ and $(q_i, p_i) \in \delta(q_{i-1}, w_i, h_i)$
- $s_n \in F$

Thinking Nondeterministically

(Click for video)

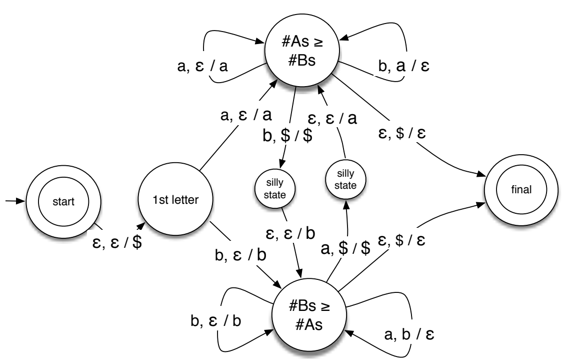
1. When we studied finite automata, we started with the deterministic model and then moved on to consider nondeterminism later. With pushdown automata, we have started right away using nondeterminism (at least in the form of epsilon transitions).

2. To reinforce the power of nondeterminism in this model, I want to explore a few solutions to the problem of building a pushdown automaton for the language

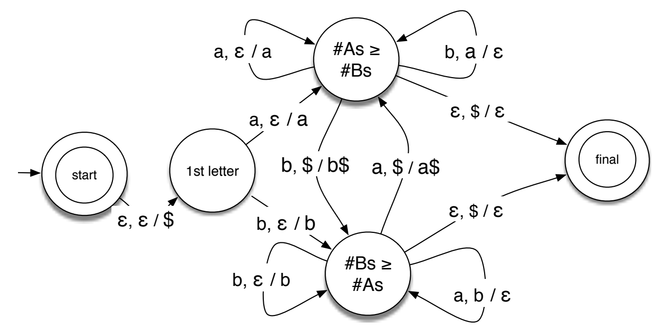
$$L_{eq\_occur} = \{w \mid w \in \{a, b\}^* \text{ and } w \text{ contains as many a's as b's}\}$$

3. Last time I showed a machine that recognized this language (almost). Unfortunately, I made a mistake and the machine shown on the slides during last class removed the $ that indicated the bottom of the stack when it should not have done so.

   - A corrected version of this machine appear below (and in the posted version of the slides from last class period).



   - This version works by using an epsilon transition to push a pair symbols onto the stack during what it logically one transition from one of the ≥ states to the other. This is easy enough to do in general, that we will let ourselves label a transition with a sequence of symbols to push on the stack even though Sipser's formalism doesn't officially allow this.



4. What I really want you to notice is the peculiar way this machine uses nondeterminism.

   - The only nondeterministic transitions in the machine are the ϵ transitions leaving "start" and leading to "final".
   - The first ϵ-transition puts a marker at the bottom of the stack so that other states can tell when it is (near) empty.
   - The transitions to "final" reflect the fact that there is no deterministic way to make a transition only when the machine reaches end of input.

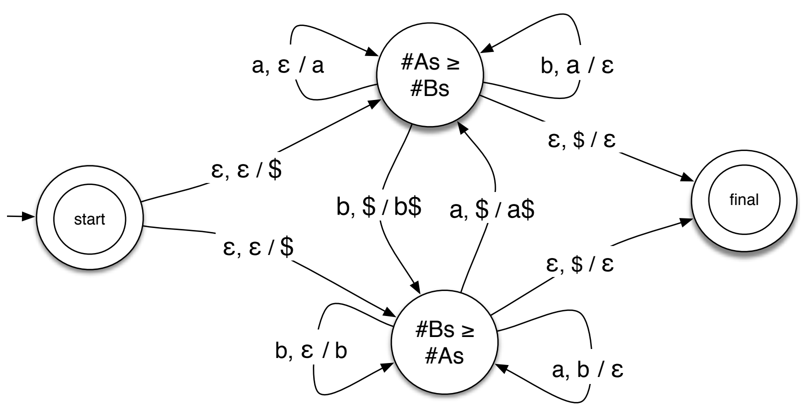
5. Once you realize that the machine is being nondeterministic in this case, you might notice some other possibilities.

6. Consider the state "1st letter".

   - If the next input is an a, this state pushes it onto the stack and goes to the "#As ≥ #Bs" state.

- Notice that if we just allowed an ϵ transition from "1st letter" to "#As $\geq$ #Bs", the "#As $\geq$ #Bs" state would push the next a onto the stack anyway,
- Notice also that the same is true for bs and the combination of the "1st letter" state and the "#Bs $\geq$ #As"

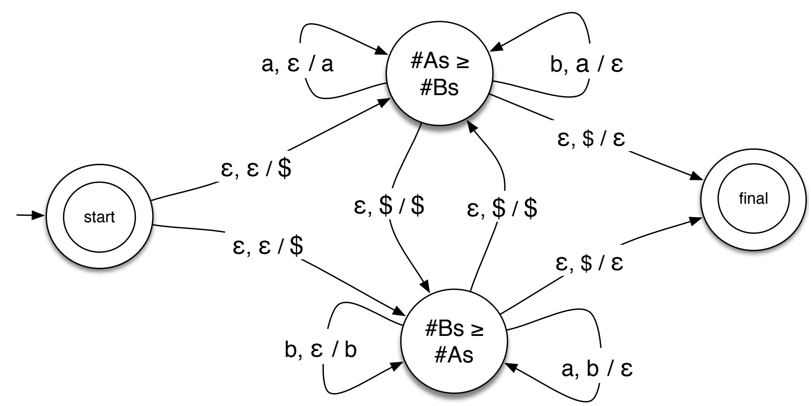
7. So, we can actually get rid of the "1st letter" state and let the start state guess whether the first letter will be an a or b by taking an epsilon transition to the appropriate $\geq$ state:



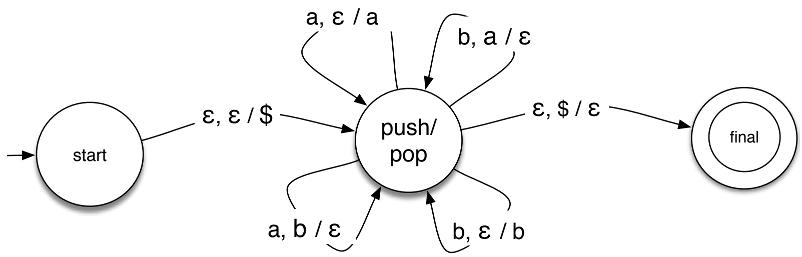
8. Now, let's think about the transitions between the two $\geq$ states.

   - Like the transitions from the "1st letter" state we just eliminated, each of these transitions consumes a letter of input and pushes the identical letter onto the stack.
   - In addition, if we had just made the transition without consuming input or pushing anything on the stack, the destination states would consume and push the same symbols.

9. So, we can let the machine do more guessing. Whenever the stack becomes empty, we can just guess whether the remaining input will start with an a or b and epsilon transition to the appropriate $\geq$ state. If we guess wrong, we can just jump back. This leads to:



10. Can you see where this is going? We have given the machine the option of jumping back and forth between the two $\geq$ states at will. It is almost as if it doesn't matter which one of the two $\geq$ states the machine is in. In fact, it doesn't. We can merge them giving:



Another Example of Nondeterminism (and a Deterministic PDA!)

(Click for video)

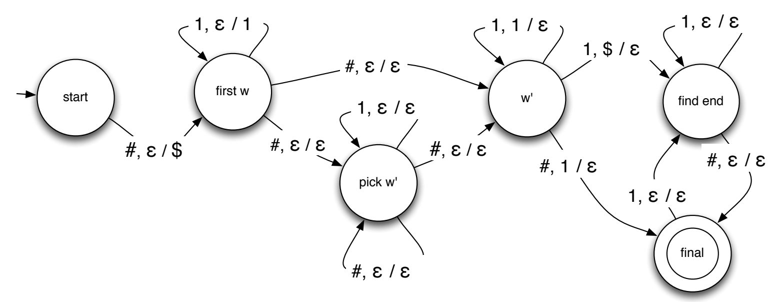
1. As another example of the use of nondeterminism, let's consider how to construct a PDA that recognizes the language

$$L_{\text{Unary-diff}} = \{\#1^{p_1}\#1^{p_2}\#\ldots\#1^{p_n}\# \mid |p_i| > 0 \;\&\; \text{ for some } i, j,\ p_i \neq p_j\}$$

   - This is the language of non-empty unary strings separated by pound signs which contain at least two unary substrings of different lengths.
   - This is an interesting example because to solve it you have to take non-determinism seriously.
   - The machine cannot check every pair of 1^{p_i} and 1^{p_j} because if it pushes p_i symbols on the stack to remember the value p_i, once it clears the state to see if p_i equals p_j the p_i symbols are gone and cannot be compared to any other string of 1s.
   - An important thing to observe, is that if there is any pair of 1^{p_i} and 1^{p_j} of different lengths, we can replace one of them with 1^{p_1} and still have a pair of distinct strings. This is because p_1 must be different from one of 1^{p_i} and 1^{p_j} and that string together with 1^{p_1} is just as good an example of two strings of different length as 1^{p_i} and 1^{p_j}.

2. Given this observation about 1^{p_1}, we can describe a machine that recognizes this language by a) pushing p_1 symbols on its state, a) non-deterministically skipping substrings of 1's until it guesses the next substrings is a different length than p_1, c) comparing the stack to the length of the guessed substring, and d) if the substring's length is different, skipping over any additional substrings and accepting as long as the string ends with a #.

   The machine below follows this strategy.



   - In state "first w" it fills the stack with as many ones as it finds in the first string.
   - In state "pick w" it uses non-determinism to guess which # precedes a string of 1s of a different length. It can also use non-determinism to skip "pick w" and move right to w' if it guesses that the second substring does not match the first.
   - In state w', it verifies its guess leaving state w' only if the number of 1s in the stack is different from the length of the selected string of ones.
   - Depending on whether there were fewer 1s in the second substring (it reached a # before the stack was empty) or more 1s in the

second substring (the stack ran out of 1s while there was still another 1 in the input) it either moves to "find end" or "final".

- We need states "find end" and "final" because to accept a PDA must read all the way to the end of the input and be in a final state when it gets there. Also for this language, the last symbol before the end must be a #. So...

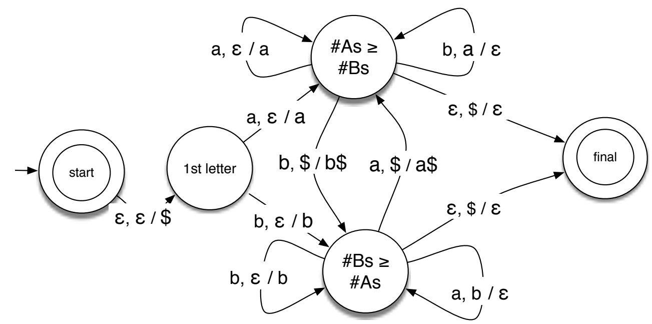
- We go to and state in "find end" as long as there are 1s still remaining. It transfers to "final" each time it sees a #. If the # is the end of input, the machine accepts. If it see another 1 it returns to "find end" where it can skip over 1s looking for another # that is potentially the end of the input.

3. The clear emphasis of the two examples we have just considered is that non-determinism often plays a big role in the design of PDAs. In the world of finite automata, we learned that nondeterminism was often helpful but never essential. We will eventually want to explore the question of whether nondeterminism is essential in PDAs (Spoiler alert: it is!). With this in mind I would like to turn back to the language

$$L_{eq\_occur} = \{w \mid w \in \{a, b\}^* \text{ and } w \text{ contains as many a's as b's}\}$$

and see how it can be recognized using a deterministic machine.

4. To accomplish this, let's look back at the least nondeterministic PDA we considered for L_{eq_occur}

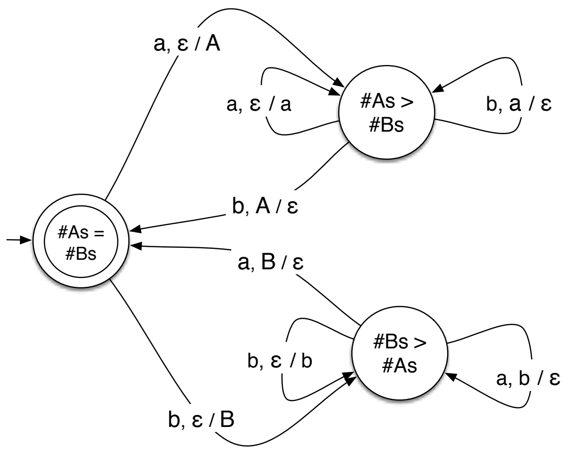


- The only nondeterministic element of this PDA is the use of ϵ-transition from the two "$\geq$" states to the final state.

- Fundamentally, the problem is that the "$\geq$" states are "$\geq$" states rather than "$>$" states. For example, if we are in the "$As \geq Bs$" state and read a "b" we will pop an "a" off the stack. This might be the last "a" on the stack! But, until we see what is underneath the "a" we don't know whether the total number of "a"s seen is greater than the number of "b"s or actually equal to the number of "b"s. If we did, we could transition to a final state when we popped the last "a" (and similarly when we pop the last "b" in the "$As \geq Bs$" state).

5. We can address this by taking advantage of the fact that be can use any set of symbols we want as our stack alphabet. In the original machine, every "a" read as input while the number of a's exceeds the number of b's was represented on the stack by the same symbol, another "a". If we instead use a special symbol when we push the bottom-most a on the stack, the presence of the distinct symbol will enable the machine to tell that the stack is about to become empty.

6. This idea is made concrete in the machine shown below:



- First, notice that unlike every other PDA I have shown you, this machine never bothers to push a "\$" onto the stack!
- Instead, this machine marks the bottom of the stack with a capital letter (an A or a B) and pushes lower-case letters whenever it knows that the stack already is non-empty.
- The start state, "#As = #Bs", is also the only final state in the machine. The machine is designed to enter this state whenever it has read a prefix of the input string in which the number of a's and b's is equal. The stack will always be empty when the machine reaches this state.
- Like the other machine, when the stack contains one or more a's (whether upper or lower case), the total number of a's in the stack will equal the difference between the total number of a's read so far and the total number of b's read so far.
- Similarly, when the stack contains one or more b's (regardless of case), the number of b's in the stack will equal the difference between the total number of b's read so far and the total number of a's read so far.

7. The punchline is that this machine is completely deterministic.

Parse Trees

(Click for video)

1. In previous classes, we considered the context-free language

$$L_{add} = \{1^i + 1^j = 1^{i+j} \mid i, j \geq 1\}$$

which is generated by the context-free grammar:

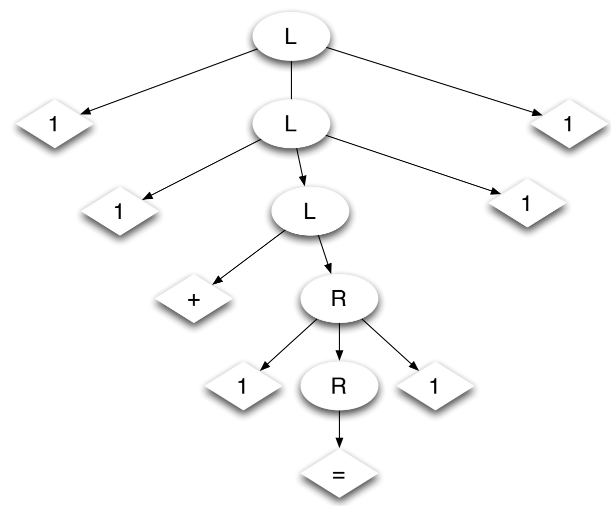
$$L \to 1L1 \mid +R$$
$$R \to 1R1 \mid =$$

2. We have seen that we can establish the fact that a string belongs to the language of a grammar like this by describing a derivation of the string from the start symbol of the grammar.

$$L \implies 1L1 \implies 11L11 \implies 11 + R11 \implies 11 + 1R111 \implies 11 + 1 = 111$$

3. As an alternative to using derivations to show that a given string belongs to the language of a grammar G is to discover a *parse tree* for the string relative to the grammar.

   - To illustrate this idea, consider the parse tree for "11+1=111" relative to our grammar for L_{add}:



- In general, we say that a labeled tree is a parse tree relative to a context-free grammar $G = (V, \Sigma, R, S)$ for a sentential form $w \in (\Sigma \cup V)^*$ if:
  - The root is labeled with the start symbol, S.
  - All interior nodes of the tree a labeled with symbols in the alphabet of variables, V.
  - All leaves are labeled with terminal symbols and the sequence of symbols found as one visits the leaves in order form left to right along the frontier forms the string w.
  - If an interior node is labeled with $L \in V$ and its children are labeled with the symbols in the string $\beta \in (\Sigma \cup V)^*$ then $V \rightarrow \beta \in R$.