Announcements

1. Homework 7 is (or will soon me) available.

Getting Formal

1. We can formalize our understanding Turing Machines, with a few exciting definitions:

**Definition:** A Turing machine is a 7-tuple $(Q, \Sigma, \Gamma, \delta, q_0, q_{accept}, q_{reject})$, where
- $Q$ is a finite set of states,
- $\Sigma$ is a finite input alphabet (not containing the blank symbol),
- $\Gamma$ is a finite tape alphabet which is a superset of $\Sigma$ including the blank symbol,
- $\delta : Q \times \Gamma \rightarrow Q \times \Gamma \times \{Left, Right\}$ is the transition function,
- $q_0$ is the start state,
- $q_{accept}$ is the accept state, and
- $q_{reject} \neq q_{accept}$ is the reject state.

2. Next, to capture the way we expect a Turing Machine to compute, we need a precise way of describing a snapshot of a point in some computation.

**Definition:** A configuration of a Turing machine is a triple $(u, q, v)$ where $q \in Q$ is the current state, $uv$ is the contents of the non-blank portion of the tape with $u$ being the portion to the left of the current head position and $v$ being the portion from the symbol currently under the head to the end of the non-blank tape.

3. With these definitions, we can now state exactly what it means for a Turing Machine to accept a string.

**Definition:** We say the configuration $(u, q, uv)$ yields configuration $(u', q', v')$ for $q, q' \in Q, a \in \Gamma$, and $u, v, u', v' \in \Gamma^*$ if for some $b$ and $c \in \Gamma$:
- $\delta(q, a) = (q', c, Left), u = u'b$, and $v' = cv$, or
- $\delta(q, a) = (q', c, Right), u' = uc$ and $v' = v$, or
- $\delta(q, a) = (q', c, Left), u = epsilon$, and $v' = cv$, or
- $\delta(q, a) = (q', c, Right), u' = uc, v = \epsilon$, and $v' = \epsilon$.

**Definition:** A TM accepts a string $w \in \Sigma^*$ if there is a sequence of configurations that begins with $(\epsilon, q_0, w)$ and ends in $(w', q_{accept}, w'')$ for some $w', w'' \in \Gamma^*$ where each configuration yields the following configuration in the series.

4. If Turing machines were like DFAs or PDAs, we would be done now. Turing machines, however, have a new option. A Turing machine can accept a string, reject a string, or just run forever without making a decision. So we need to distinguish two ways in which a language $L$ can be “the language of a Turing machine”.

**Definition:** A language $L$ is Turing-recognizable if some Turing machine accepts $w$ if and only if $w \in L$. We call these languages recursively-enumerable.

**Definition:** A language $L$ is Turing-decidable if some Turing machine that halts on all inputs accepts $w$ if and only if $w \in L$. We call these languages recursive.

Variations on Automata

1. An important part of our agenda at this point is to explore the case that the Turing Machine is a model that captures everything a computer can do. We cannot prove this, but we can reassure ourselves that it seems reasonable to make this assumption in two ways.

- First, we will explore whether adding features to the Turing Machine model adds computational power.
If adding features does not increase the computational power of the model, maybe the model is as powerful as it could be!

For example, we have already done a bit of this with DFAs. We saw (surprisingly?) that adding nondeterminism to our model for DFAs did not extend its computational power. On the other hand, if we changed the DFA model so that a DFA could move back and forth and write on its tape as long as it did not go beyond the space originally filled with its input, this would increase the power of the model.

In addition, we can argue that the Turing Machine model is computationally equivalent to several other models that are radically different in design.

I have already hinted broadly that a TM tape could be used as a big array suggesting it might be possible to simulate computations in traditional programming systems (Java?) that use arrays.

As a starting point, rather than adding a feature to the TM model, let’s consider the impact of adding a feature to one of our less powerful models, the PDA. What happens to the computational power of a PDA if we give it two stacks?

2-PDAs

1. Recall the formal definition of a pushdown automaton:

   **Definition:** A pushdown automaton is a 6-tuple \((Q, \Sigma, \Gamma, \delta, q_0, F)\) where:
   - \(Q\) is a finite set of states,
   - \(\Sigma\) is a finite input alphabet,
   - \(\Gamma\) 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.

2. Now, we can define a 2-tape PDA:

   - If a transition arrow is labeled “a, b, c / d, e” it means that if the machine is in the state where the arrow originates, it can transition to the target of the arrow if a is the next input character, b and c are the characters at the tops of its two states. If the transition is used, then the symbols b and c should be popped from the stacks and d and e should be pushed. Any of a, b, c, d, and e can be empty.
   - While in state pre #, the machine shown scans until it finds a # while pushing all of the as and bs it encounters onto its first stack so that at the end the first symbol from the input is at the bottom of the first stack and the last is at the top of the first stack.
   - When it hits the #, it pops each symbol from its first stack and pushes it on the second stack until the first stack is empty. When this is complete, the second stack contains a copy of the first half.
of the input with the first symbol at the top of the stack and the last symbol at the bottom.

- Next, it scans to the right popping symbols off the second stack as long as each symbol on the stack matches the next input symbol.
- If the second half of the input matches the first, it will eventually empty the second stack and then transition to its only accepting state. Thus, the machine is designed to accept \( w \# w \).

4. Remember, \( w \# w \) is not a context-free language. The machine we just described therefore shows that 2-tape PDAs are clearly more powerful than single stack PDAs.

5. To appreciate just how much more powerful a 2-tape PDA is, recall the notion of a Turing machine configuration:

**Definition:** A configuration of a Turing machine is a triple \((u, q, v)\) where \( q \in Q \) is the current state, \( uv \) is the contents of the non-blank portion of the tape with \( u \) being the portion to the left of the current head position and \( v \) being the portion from the symbol currently under the head to the end of the non-blank tape.

- Given a TM configuration, we could store all the symbols that are before the tape head \((u)\) in one stack and all of the symbols after the tape head (including the symbol under the tape head) in the second stack.
- We could arrange for this initially using a few states equivalent to the pre\# and at\# states of the sample machine for \( w \# w \) shown above.
- We could then define the remaining states and transitions of the 2-tape PDA in a way that mimicked any TM. In particular, if \( \delta_{TM} \) is the TM’s transition function and \( \delta_{PDA} \) is the transition function for our PDA then
  - If \( \delta_{TM}(q, a) = (q', b, \text{Right}) \) then \( \delta_{PDA}(q', \epsilon, a) = \{(q', b, \epsilon)\} \).
  - If \( \delta_{TM}(q, a) = (q', b, \text{Left}) \) then for all \( c \),
    
    \[ * \delta_{PDA}(q', \epsilon, a) = \{(q'-\text{push}, \epsilon, b)\}, \text{ and} \]
    
    \[ * \delta_{PDA}(q'-\text{push}, \epsilon, c, \epsilon) = \{(q', \epsilon, c)\} \]
    
    (the intermediate state \( q'-\text{push}\_c \) is only required because our definition of PDA’s limits us to pushing one symbol on the stack at a time).

Thus, a 2-tape PDA is at least as powerful as a Turing Machine!