4510 F23
Ladha

Turing-Completeness

Recall last time we gave a ~~free~~ definition of the Church-Turing Thesis, which is an association (correspondence) between the informal, intuitive notion of an algorithm, and the formal definition of a Turing machine. The simple, humble Turing-machine is the supreme.

CTT: $\exists C \quad \mathcal{L}(TM) \subseteq \mathcal{L}(C)$.
$\forall C \quad \mathcal{L}(C) \subseteq \mathcal{L}(TM)$

where C is a realizable model of computation. A model of computation is said to be Turing-complete (or Turing equivalent) if $\mathcal{L}(TM) \subseteq \mathcal{L}(C)$. If you can simulate a Turing machine on that computational model. By the Church-Turing-Thesis, we get $\mathcal{L}(C) \subseteq \mathcal{L}(TM)$ for free, so to prove $\mathcal{L}(C) = \mathcal{L}(TM)$, you need only to prove that $\mathcal{L}(TM) \subseteq \mathcal{L}(C)$. It could only be the case then that $\mathcal{L}(TM) = \mathcal{L}(C)$.

there is no level 4

Lets consider the four models from last time. stayTM, 2wayTM, kTM, NTM. These were all defined as generalizations of the Turing machine, so for each one, we got ~~for~~ for free that $\mathcal{L}(TM) \subseteq \mathcal{L}(stayTM)$, and then we proved that $\mathcal{L}(stayTM) \subseteq \mathcal{L}(TM)$.

Lets prove Python, an ideal programming language is Turing-complete. First, as a computational model, what is python? It is only a notation, to abstract us away from the hardware. It is Turing-complete as you could write a Turing machine simulator in python relatively easily, so obviously $\mathcal{L}(TM) \subseteq \mathcal{L}(PY)$. Any language you could write a Turing machine simulator in is Turing complete.

We exercised the CTT in two ways here. First, we didn't have to prove $\mathcal{L}(PY) \subseteq \mathcal{L}(TM)$. A python simulator as a Turing machine, yielded. Second, to prove $\mathcal{L}(TM) \subseteq \mathcal{L}(PY)$, we never actually gave the python program! Just supposed that we could. The CTT allows you to conceptualize an algorithm, and then just the fact you could implement it if you had to is enough to argue about the existence of a program for some task. To what level, you will formalize this on the next homework.

Recall the kinds of grammars we have seen so far

regular grammar $V \to \Sigma^* V \mid \Sigma \mid V \mid \varepsilon$ $\quad A \to aA$

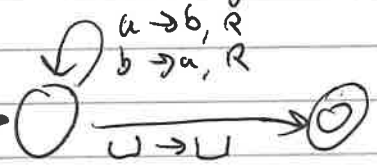
context-free $V \to (V \cup \Sigma)^*$ $\quad A \to bCdE\ldots$

context-sensitive $aVb \to a(V \cup \Sigma)^* b$, $\quad q, b \in (V \cup \Sigma)^*$ $\quad aAa \to aBa$

each one defined a class of languages greater than the previous one. We relaxed and generalized the kinds of productions allowed, we gained power to produce more and more languages. What about an unrestricted grammar?

unrestricted grammar $(V \cup \Sigma)^* \to (V \cup \Sigma)^*$

we prove $\mathcal{L}(TM) \subseteq \mathcal{L}(UG)$ unrestricted grammars are Turing-complete.
we first investigate a Turing-machine computation.



q_0 abaaaaaaaaaaa
␣ q_0 baaaaaaaaaaa
b a q_0 aaaaaaaaaa
b a b q_0 aaaaaaaaa
b a b b q_0 aaaaaaaa
...

observe that computation is local. for each configuration only a constant amount of it is changed from step to step. we can easily simulate each configuration as a working string in a grammar,
for states $\{q_0 \ldots q_k\}$, add states $\{S, Q_0, \ldots, Q_k, A, B, \sqcup\}$

for transition $(q_i) \xrightarrow{a \to b, R} (q_j)$ add production $Q_i a \to b Q_j$
for transition $(q_i) \xrightarrow{a \to b, L} (q_j)$ add productions $aQ_i a \to Q_j ab$
$bQ_i a \to Q_j bb$
$\sqcup Q_i a \to Q_j \sqcup b$

great! But we need to begin on the input.
A grammar nondeterministically produces all correct strings But a Turing machine begins on input. we just nondeterministically simulate the TM on all inputs.

$S \Rightarrow Q_0 AB \quad A \Rightarrow aA \mid bA \mid \varepsilon, \quad B \Rightarrow \sqcup B \mid \varepsilon.$

so $S \Rightarrow q_0 \Sigma^* \sqcup^*$ basically. If we need more blanks, B can produce more for us.

Now how do we terminate? We make our single halting state clean up the output for us. A real Turing machine would loop left and right to destroy stuff, so we simulate it doing the same

$$aQ_h \Rightarrow Q_h a \qquad Q_h a \Rightarrow aQ_h$$
$$bQ_h \Rightarrow Q_h b \qquad Q_h b \Rightarrow bQ_h$$
$$Q_h \sqcup \Rightarrow Q_h \qquad \sqcup Q_h \Rightarrow Q_h$$

Unnecessary, but its fine. Then we remove the state, $Q_h \Rightarrow \varepsilon$ with no more nonterminals we end with a produced string. Could also pretend $\sqcup$ is a nonterminal and add $\sqcup \Rightarrow \varepsilon$. From this we conclude $\mathcal{L}(TM) \subseteq \mathcal{L}(UG)$. It also is an interesting characterization of the power of Grammars

$$\mathcal{L}(RG) \subset \mathcal{L}(CFG) \subset \mathcal{L}(CSG) \subset \mathcal{L}(UG)$$

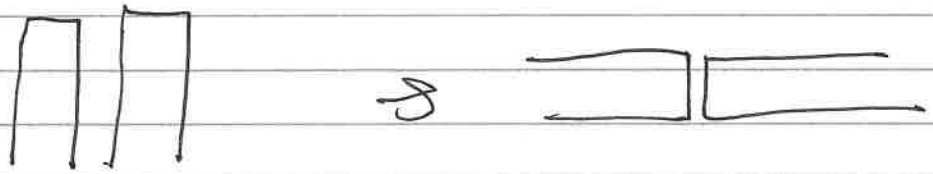
by the Church-Turing Thesis, there is no grammar greater than a unrestricted one.

~

Lets go back and consider the PDA. Certainly one stack is greater than no stack. What about two stacks? Define a 2PDA with transition function like $\delta : Q \times \Gamma \times \Gamma \times \Sigma \to (Q \times \Gamma \times \Gamma)$. It can read from the input, pop from both stacks and push to both stacks simultaneously.

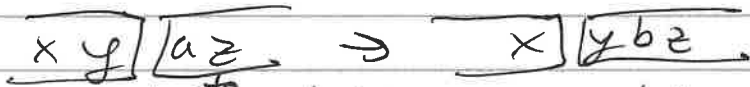
We prove $\mathcal{L}(TM) \subseteq \mathcal{L}(2PDA)$. Two stacks is Turing complete.

Heres the visual proof:



point the stacks at each other to get a 2 way tape!

from $a \to b, R$ 
pop a from stack 2, push b to stack 1.

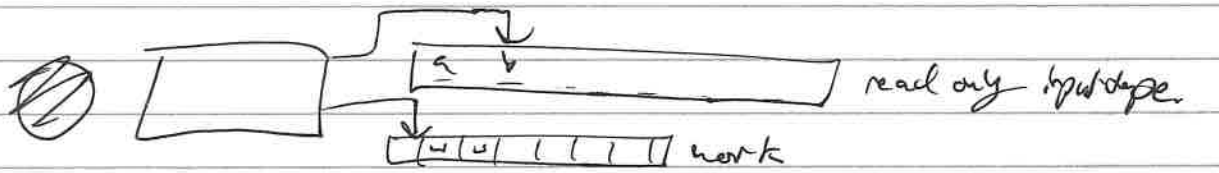
for $a \to b, L$ 
pop a from stack 2, push b ~~from~~ to stack 2, pop symbol from stack 1 and push it to stack 2. I will leave it to you to fill in the details. This also gives us an interesting hierarchy in terms of the number of stacks.

$$\mathcal{L}(\text{0PDA}) \supseteq \mathcal{L}(\text{1PDA}) \supseteq \mathcal{L}(\text{2PDA})$$

By the Church-Turing Thesis, two stacks is enough! $\mathcal{L}(\text{3PDA}) \subseteq \mathcal{L}(\text{2PDA})$
Any more than two stacks does not give you more power.
Just like more than one tape doesn't give more power.

~

for any of these systems, unbounded memory is important. We prove a Turing machine with fine (work) tape is not Turing-complete.



Such a device is in fact regular! If there is finite tape, there is also only a finite amount of configurations. To each configuration assign a state and define δ appropriately. This is simply a DFA.

$q_0aba \rightarrow bq_0ba$ do 

Essentially, if you have finite space, move it all into the registers. I see this argument a lot in favor of why no real computer is Turing complete. This is fallacious. A Turing machine doesn't have infinite memory in any usable sense. ~~If has~~ Any computation which halts can only use as much tape as the number of steps. So any computation we care about has some space bound.

Do not think of the tape as part of the machine, no more than a road is part of a car. The tape is nature, the environment, the earth, the universe. We do not exist and "do" on earth in any way limited by the size of the planet. In fact, this is a good marker to test if any kind of system is Turing-complete.

Defn: A computation is a process which modifies an environment via repeated applications of a set of rules.

Other examples of Turing complete systems include boolean circuits, conways game of life, most programming languages, minecraft redstone, and much more.

$\mathcal{L}(DFA)$, $\mathcal{L}(NFA)$, $\mathcal{L}(REX)$, $\mathcal{L}(GNFA)$, $\mathcal{L}(RG)$ $\subsetneq$ $\mathcal{L}(CFG)$, $\mathcal{L}(PDA)$ $\subsetneq$ $\mathcal{L}(CSG)$ $\subsetneq$ $\mathcal{L}(TM)$, $\mathcal{L}(2PDA)$, $\mathcal{L}(UG)$, $\mathcal{L}(NTM)$, $\mathcal{L}(2wayTM)$, and many more variants.