601.231 - Automata and Computation Theory

Kaushik Srinivasan

April 10, 2019

Introduction

These notes were partially live-TEXed—the rest were TEXed from course videos—then edited for correctness and clarity. I am responsible for all errata in this document, mathematical or otherwise; any merits of the material here should be credited to the lecturer, not to me.

Acknowledgements

In addition to the course staff, acknowledgment goes to Zev Chonoles, whose online lecture notes "inspired" me to create my own. I have also borrowed his format for this introduction page.

Topic 1 — Regular Languages

Remark. *Finite Automata* is a a good model for very limited memory. States represent their memory.

Topic 1.1 — Finite Automata

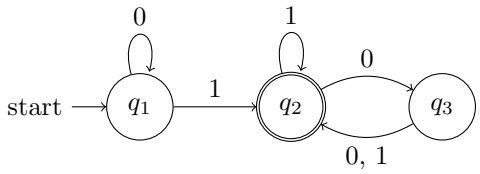
Definition 1.1.1. Our definition of important characters.

- **Alphabet** Σ - any finite set of symbols e.g. $\{0,1\}$
- **Empty string** ϵ - string with length zero
- **String over** Σ - any finite sequence of symbols e.g. $0, 01$, etc.
- **Concatenation** of strings x and y is xy
- **Language over** Σ - set of strings over Σ e.g. $\{0, 1, 10, 01\}$.

A **string** over an alphabet is a **finite sequence of symbols** - infinitely many possibilities.
A **language** over an alphabet is a **set of strings.** - e.g. $\{0, 1, 10\}, \{0, 00, 000, 0000, \ldots\}, \{\}, \{\epsilon\}$

Definition 1.1.2. A **finite automaton** is a 5-tuple $(Q, \Sigma, \delta, q_0, F)$

1. Q - is a finite **set of states**
2. Σ - finite set called **alphabet**
3. $\delta : Q \times \Sigma \to Q$ - is **transition function**.
4. $q_0 \in Q$ - is the **start state**
5. $F \subseteq Q$ - is the **set of accepted state**



Definition 1.1.3. A string is **accepted** by the FA if the string ends in a final state. Machine can accept many strings but only one language.

Accept no string $\Rightarrow$ Accepts $\emptyset$

Let $w = w_1 w_2 w_3 \ldots w_n$, be the string where $w_i \in \Sigma$. FA M accepts w if there is a sequence of states $r_0, r_1, \ldots, r_n \in Q$ and it satisfies the conditions:

1. $r_0 = q_0 \leftarrow$ Sequence starts properly
2. $\delta(r_i, w_{i+1}) = r_{i+1} \leftarrow$ sequence uses legal transitions
3. $r_n \in F \leftarrow$ sequence ends in final state..

Definition 1.1.4. A machine M **recognizes** a language A if A is the set containing every string w that the machine M accepts and no strings that it doesn't.

Definition 1.1.5. A language is called **regular** if some finite automaton recognizes it.

Regular Operations

Let A & B be Regular Languages

- **Union** - $A \cup B = \{x | x \in A \text{ or } x \in B\}$
- **Intersection** - $A \cap B = \{x | x \in A \text{ and } x \in B\}$
- **Concatenation** - $A \circ B = \{xy | x \in A \text{ and } y \in B\}$
- **Star** - $A^* - \{x_1 x_2 x_3 \ldots x_k | k \geq 0 \text{ and each } x_i \in A\}$
- **Complement** - $\bar{A} = \{w |\ w \in \Sigma^* \cap w \notin A\}$

Theorem 1.1.6. *The Set of Regular Languages is Closed Under Union. (Using DFA)*

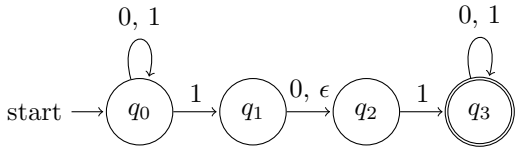
Proof. $\mathtt{M}_1 = (Q_1, \Sigma, \delta_1, q_1, F_1)$, $\mathtt{M}_2 = (Q_2, \Sigma, \delta_2, q_2, F_2)$. Now construct $\mathtt{M} = (Q, \Sigma, \delta, q, F)$

1. $Q = \{(r_1, r_2) |\ r_1 \in Q_1 \cap r_1 \in Q_2\}$
2. $\Sigma = \Sigma_1 \cup \Sigma_2$
3. $\delta((r_1, r_2), a) = (\delta_1(r_1, a), \delta_2(r_2, a))$
4. $q_0 = (q_1, q_2)$
5. $F = \{(r_1, r_2) |\ r_1 \in F_1 \cup r_2 \in F_2\}$

■

Topic 1.2 — Nondeterminism

Remark. In a ***nondeterministic*** machine, several choices may exist for the next state at any point.

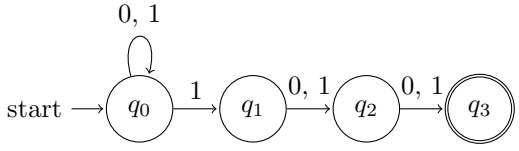


Transition arrows in NDA can be labelled with alphabet symbols OR labelled w/ϵ, travel anytime without using input.

Definition 1.2.1. A **non-deterministic** finite automaton is $(Q, \Sigma, \delta, q_0, F)$

1. Q - is a finite **set of states**
2. $\Sigma_\epsilon = \Sigma \cup \{\epsilon\}$ - finite set called **alphabet**
3. $\delta : Q \times \Sigma_\epsilon \to \mathcal{P}(Q)$ - is **transition function**.
4. $q_0 \in Q$ - is the **start state**
5. $F \subseteq Q$ - is the **set of accepted state**

Example. Let $\Sigma = \{0, 1\}$. Draw a NFA that recognizes $L = \{w |\ 1 \text{ is the 3rd last position}\}$



Definition 1.2.2. Two machines are *equivalent* if they recognize the same language.

Theorem 1.2.3. *Every nondeterministic finite automaton has an equivalent deterministic finite automaton.*

Proof. Let $\mathtt{N} = (Q, \Sigma, \delta, q_0, F)$ be an NFA A. We construct DFA $\mathtt{M} = (Q', \Sigma, \delta', q_0', F')$ recognizing A. First construct without ϵ transitions.

1. $Q' = \mathcal{P}(Q)$ $\leftarrow$ power set of states of N.
2. Σ stays the same.
3. $\delta'(R, a) = \bigcup_{r \in R} \delta(r, a)$ $\leftarrow$ For $R \in Q'$ and $a \in \Sigma$.
4. $q_0' = \{q_0\}$
5. $F' = \{R |\ R \in Q' \text{ and } R \cap F \neq \emptyset\}$

Now suppose $\mathtt{N}$ has ϵ-transitions. Now construct $E(R) = \{q | q \text{ can be reached from } R \text{ by travelling along 0 or more } \epsilon \text{ arrows } \}$. Now define

- $\delta'(R, a) = \bigcup_{r \in R} E(\delta(r, a)) = \{q |\ q \text{ in } E(\delta(r, a)), q \in Q \text{ for some } r \in R\}$
- $q_0' = E(\{q_0\})$

■

Subset Construction

Subset construction should follow from the proof above. This is mainly through practice and recognition. Some important points are to include the epsilon transition *after* travelling along an edge. Work from the powerset $\mathcal{P}(Q)$ and perform the δ operation on each element.

NFA Closure Proofs

Theorem 1.2.4. *The class of regular languages is closed under the* ***union*** *operation (for NFAs)*

Proof. Let $\mathtt{N}_1 = (Q_1, \Sigma, \delta_1, q_1, F_1)$ recognizing language A_1, $\mathtt{N}_2 = (Q_2, \Sigma, \delta_2, q_2, F_2)$ recognizing language A_2. Now construct $\mathtt{N} = (Q, \Sigma, \delta, q, F)$ recognizing $A_1 \cup A_2$

1. $Q = \{q_0\} \cup Q_1 \cup Q_2$ $\leftarrow$ with new state q_0
2. Σ stays the same
3. $q_0 = \{q_0\}$ $\leftarrow$ new start state.
4. $F = F_1 \cup F_2$
5. For any $q \in Q$ and any $a \in \Sigma_\epsilon$

$$\delta(q, a) = \begin{cases} \delta_1(q, a) & q \in Q_1 \\ \delta_2(q, a) & q \in Q_2 \\ \{q_1, q_2\} & q = q_0 \text{ and } a = \varepsilon \\ \emptyset & q = q_0 \text{ and } a \neq \varepsilon \end{cases}$$

■

Theorem 1.2.5. *The class of regular languages is closed under the* ***concatenation*** *operation (for NFAs)*

Proof. Let $\mathtt{N}_1 = (Q_1, \Sigma, \delta_1, q_1, F_1)$ recognizing language A_1, $\mathtt{N}_2 = (Q_2, \Sigma, \delta_2, q_2, F_2)$ recognizing language A_2. Now construct $\mathtt{N} = (Q, \Sigma, \delta, q_0, F)$ recognizing $A_1 \circ A_2$

1. $Q = Q_1 \cup Q_2 \leftarrow$ all states of $\mathtt{N}_1$ and $\mathtt{N}_2$
2. Σ stays the same
3. $q_0 = \{q_1\} \leftarrow$ Same start state of $\mathtt{N}_1$
4. $F = F_2 \leftarrow$ same accept states as $\mathtt{N}_2$
5. For any $q \in Q$ and any $a \in \Sigma_\epsilon$

$$\delta(q, a) = \begin{cases} \delta_1(q, a) & q \in Q_1 \text{ and } q \notin F_1 \\ \delta_1(q, a) & q \in F_1 \text{ and } a \neq \varepsilon \\ \delta_1(q, a) \cup \{q_2\} & q \in F_1 \text{ and } a = \epsilon \\ \delta_2(q, a) & q \in Q_2 \end{cases}$$

■

Theorem 1.2.6. *The class of regular languages is closed under the* ***star*** *operation (for NFAs)*

Proof. Let $\mathtt{N}_1 = (Q_1, \Sigma, \delta_1, q_1, F_1)$ recognizing language A_1. Now construct $\mathtt{N} = (Q, \Sigma, \delta, q_0, F)$ recognizing A_1^*

1. $Q = \{q_0\} \cup Q_1 \leftarrow$ states of $\mathtt{N}_1$ and new start state
2. Σ stays the same
3. $q_0 = \{q_0\} \leftarrow$ new start state
4. $F = \{q_0\} \cup F_1$
5. For any $q \in Q$ and any $a \in \Sigma_\epsilon$

$$\delta(q, a) = \begin{cases} \delta_1(q, a) & q \in Q_1 \text{ and } q \notin F_1 \\ \delta_1(q, a) & q \in F_1 \text{ and } a \neq \varepsilon \\ \delta_1(q, a) \cup \{q_1\} & q \in F_1 \text{ and } a = \varepsilon \\ \{q_1\} & q = q_0 \text{ and } a = \varepsilon \\ \emptyset & q = q_0 \text{ and } a \neq \varepsilon \end{cases}$$

■

Topic 1.3 — Regular Expressions

Definition 1.3.1. We say that R is a **regular expression** if R is -

1. a (represented by $\{a\}$) for some a in alphabet Σ
2. ε (represented by $\{\varepsilon\}$) - language containing a single string
3. $\emptyset$ - language containing no strings.
4. $(R_1 \cup R_2)$, where R_1 and R_2 are regular expressions.
5. $(R_1 \circ R_2)$, where R_1 and R_2 are regular expressions.
6. (R_1^*), where R_1 is a regular expression. (contains ε)

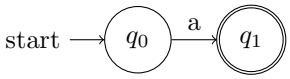
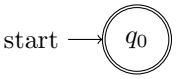
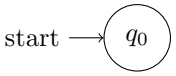
Precedence: $\star > \circ > \cup$

Example. $\{w |\ w \in \Sigma^* \text{ and } w \text{ has length less than 4}\}$
$\varepsilon \cup \Sigma \cup \Sigma\Sigma \cup \Sigma\Sigma\Sigma$

Example. $\{w |\ w \in \Sigma^*, w \text{ has length greater than 4}\}$
$\Sigma\Sigma\Sigma\Sigma\Sigma\Sigma^*$

Theorem 1.3.2. *A language is regular **iff** some regular expression describes it.*

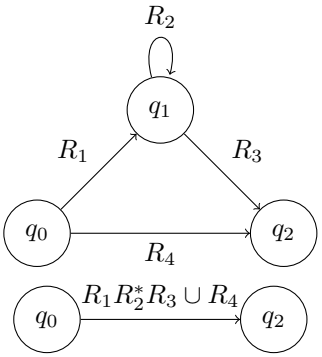
Proof. ($\Rightarrow$) If a language is described by a regular expression, then it is regular. To do this, let's construct NFA N from regular expression R.

Regular Expression	NFAs
a	
ε	
$\emptyset$	
$R_1 \cup R_2$ $R_1 \circ R_2$ R_1^*	Use Constructions from proofs and closure of regular languages.

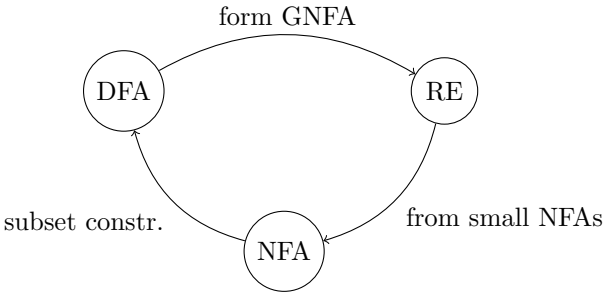
($\Leftarrow$) If a language is regular, then it is described by a regular expression. We prove this by creating a ***generalized nondeterministic finite automaton*** (GNFA). Proof is sophisticated and is not explored here. ■

Definition 1.3.3. A GNFA is $(Q, \Sigma, \delta, q_{\text{start}}, q_{\text{accept}})$

1. Q - finite set of states.
2. Σ - input alphabet
3. $\delta : (Q - \{q_{\text{accept}}\}) \times (Q - \{q_{\text{start}}\}) \to \mathcal{R}$ - transition function
4. q_{start} - start state.
5. q_{accept} - accept state.



Equivalent in Power



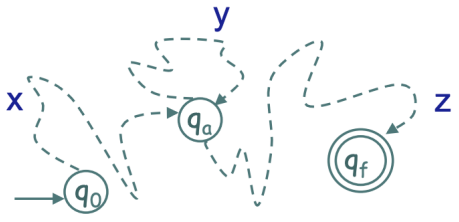
Topic 1.4 — Nonregular Languages

Remark. Let the language $A = \{0^n1^n | \ n \geq 0\}$. We cannot constuct a NFA to recognize A. Hence A is a nonregular language.

Theorem 1.4.1. ***(Pumping Lemma)*** *If A is a regular language, then there exists some number p (pumping length) where if $s \in A$ is any string of length atleast p, then s may be divided into three pieces, $s = xyz$, satisfying:*

1. for each $i \geq 0$, $xy^iz \in A$
2. $|y| > 0$
3. $|xy| \leq p$

Proof. This is not a rigorous proof, but just presenting the idea. Let M be a DFA which recoginizes language A. Then $s \in A$ of length n goes through $n+1$ states. This a string of length atleast p will go through atleast one state twice *(by pigeonhole priciple)*. Let $s = xyz$ where x, z are before and after the repeat state respectively and y is the repeat substring. if $xyz \in A$, then also $xyyz \in A$, $\ldots xy^iz \in A$.



Here, it is clear that $xy^iz \in A$. We know that $|y| > 0$ as for the loop to occur $|y| \geq 1$. This also brings the fact that $|xy| \leq p$ as the first repetition has to happen within first $p+1$ states visited since there are only p states in DFA M. We use contradiction to prove pumping lemma. ■

Example. Prove that $A = \{0^n1^n | n \geq 0\}$ is not regular.

Proof. Assume, FSOC, that A is a regular language. Let p be the pumping length for language A guarenteed by Pumping Lemma.

Consider string $s = 0^p1^p$. We know that $s \in A$ and $|s| \geq p$. By the PL, there is atleast one way to split s into x, y, z s.t. $\forall i \geq 0,\ s' = xy^iz \in A$.
Since PL says that $s = xyz$ and $|xy| \leq p$, regardless of split type, we can only have 0's in y. Let $k = |y|$, so we have $y = 0^k$. Since $|y| > 0$, we have $k > 0$.
Now choose $i = 2$, PL says $s = xyyz \in A$. But $s' = 0^{p+k}1^p$. Since $k > 0$, $p + k \neq p$. Thus s' contains strictly more 0's than 1's $\rightarrow s' \notin A$. $\Rightarrow\Leftarrow$. A is not regular. ■

Example. Let $C = \{w|\ w \in \{0,1\}^* \cap n_0(w) = n_1(w)\}$. Show C is nonregular.

Example. Let $F = \{ww|\ w \in \{0,1\}^*\}$. Show F is nonregular.

Proof. Assume FSOC that F is regular, p be the pumping length given by lemma. Let $s' = 0^p10^p1$. Clearly $s' \in F$, $|s'| > p$, so s' can be split s.t. $s' = xyz$. Satisfying the three conditions of the lemma. We show that this is impossible. Due to PL, we take $y = a^k,\ k > 0$.
Now the new string $s' = 0^{p+k}10^p1$ has length $2p + 2 + k$. Since $k > 0$, we cases for which k is odd or even.

1. *k is odd.* Then s' is odd and so $s' \notin F$ as all strings F are even length.
2. *k is even.* Then $k \geq 2$. We know the right half of s' must contain $(2p+2+k)/2 = p+1+k/2 \geq p+2$ characters. But this means right half must contain both 1's as rightmost characters of s' are 10^p1, which is $p+2$ long. This means that left half contains nothing but 0's, so half cannot be equal. Hence $s' \notin F$

Regardless of way of partition $s' = xy^iz$. We know $s' \notin F$. $\Rightarrow\Leftarrow$. F is not regular. ■

Remark. If we choose $s' = 0^p0^p$, we fail to prove the contradiction.

Example. Consider the language $F = \{a^ib^jc^k|i, j, k \geq 0 \cap i = 1 \rightarrow \text{ if } j = k\}$, where $\Sigma = \{a, b, c\}$

Proof. FSOC, assume F is regular. Let $F' = F \cap \{ab^jc^k \mid j, k \geq 0\} = \{ab^ic^i|\ i \geq 0\}$. Note $\{ab^ic^i|\ i \geq 0\} = ab^*c^*$ and is regular. Hence we need to prove that $\{ab^ic^i|\ i \geq 0\}$ is not regular. We can set $s = ab^pc^p$. We have that $s \in F'$ and $|s| \geq p$. Now we need to split into x, y, z s.t. $s' = xy^iz \in F'$. We know that $|xy| \leq p$ and $|y| > 0$. Hence we can set $y = a, ab^{k-1}, b^k$. Let's choose $i = 2$.

- if $y = b^k$, then there are more b's than c's
- Otherwise there are more than one a's

Hence $s' \notin F'$ by contradiction. Hence F' is not a regular language. Since F' is not regular, then F is also not regular. ■

Things to remember.

- You **don't** get to choose a value of p. - use p as a fixed constant given to you by the PL
- You **do** get to choose a specific string s - s must be in L and have length atleast p.
- You **don't** get to choose a specific partition $s = xyz$ - must consider **all** possible partitions.
- You **do** get to choose the value for i - can be $0, 2, 3, \ldots$ (never pick $i = 1$) - select different i's for different partitions.

Topic 2 — Context-Free Languages

Topic 2.1 — Context-Free Grammars

Definition 2.1.1. A ***Context-Free Grammar*** G is a 4-tuple (V, Σ, R, S), where

- V - is a finite set called **variables**.
- Σ - is a finite set, disjoint from V, called **terminals**.
- R - is finite set of **rules/productions**.
- $S \in V$ - is the **start symbol**.

Each rule consists of an arrow, with variable on LHS and sequence of variable, terminals on the RHS. Variables usually capital letters.
If u, v, w are strings of variables and terminals, and $A \rightarrow w$ is a rule of the grammar. We say that uAw ***yields*** uwv, written $uAv \Rightarrow uwv$. We say u ***derives*** v written $u \overset{*}{\Rightarrow} v$

Claim 2.1.2. *CFGs are closed under Union, Kleene Star, and Concatenation.*

Example. An implementation of CFG called G_1.

1. $V = \{E, O\}$
2. $\Sigma = \{+, *, a, b\}$
3. $R = \{E \rightarrow a \mid b \mid EOE,\ \ O \rightarrow +|*\}$
4. $S = E$

The "Language of grammar G_1", denoted $L(G_1)$, is the set of all songs over Σ that can be generated from these rules, starting from E.

Example. $CFG\ G_2$ where $L(G_2) = \{0^n1^n|n \geq 1\}$

1. $V = \{S\}$
2. $\Sigma = \{0, 1\}$
3. $S \rightarrow 0S1 \mid 01$

Remark. To check for construction can work. Ensure

1. Check for ***completeness***: Does G_2 generate every string in $\{0^n1^n|\ n \geq 1\}$
2. Check for ***consistency***: Does G_2 generate only string in $\{0^n1^n|\ n \geq 1\}$

Example. $L(G_3) = \{w|\ w \text{ is a palindrome over } \{a, b\}\}$

1. $V = \{S\}$
2. $\Sigma = \{a, b\}$
3. $R = \{S \longrightarrow aSa \mid bSb \mid a \mid b \mid \varepsilon\}$
4. S is start state

Types of Construction (Closure Properties)

- **Union**: $S \to A \mid B$
- **Kleene Star**:
  - $S \to Sab \mid \varepsilon$ - where ab can be added to the back.
  - $S \to aSa \mid bSb \mid \varepsilon$ - growing outwards.
- **Concatenation**: $S \to AB$ - where pattern A followed by B

NOTE: CFGs are not closed under **complement** or **intersection**.

Example. G_5 s.t. $L = \{w|w \in \{a, b\}^* \cap n_a(w) = n_b(wz)\}$

$$S \longrightarrow aSbS \mid bSaS \mid \varepsilon$$

How to prove two claims?

1. (***Correctness***) Every string w that can be generated G_5 is in L
2. (***Completeness***) Every string $w \in L$ can be generated by G_5

Claim 2.1.3. *Every string w that can be generated G_5 is in L*

Proof. We use proof by induction.

- **Base Case:** $N = 1$. The string is derivable from start symbol S in $N = 1$ steps is ϵ. Derivation is $S \Rightarrow \epsilon$. Clearly $n_a(\epsilon) = 0 = n_b(\epsilon)$, hence $\epsilon \in L$
- **Inductive Step**: (for $N > 0$). Assume that for some N, all strings w derivable from S in $\leq N$ steps are in L *(inductive hypothesis)*.
  - Consider string w' derivable from G in exactly $N + 1$ steps.
  - Clearly first sttep is either $S \Rightarrow aSbS$ or $S \Rightarrow bSaS$. Either case, the remaining N steps can be split into 2 derivations, one that generates string w_1 and first S_1 and one that generates string w_2 from second S_2.
  - By inductive hypothesis, it holds that both w_1 and w_2 there are equal numbers of a's and b's
  - Hence since w' equals either aw_1bw_2 or bw_1aw_2, there are equal numbers of a's and b's in all of w'. So w' is in L.

■

Claim 2.1.4. *Every string $w \in L$ can be generated by G.*

Proof. We use proof by induction. Note all strings $w \in L$ must have even length.

- **Base Case:** $N = 0$. Only one string is L contains 0 a's $\to \varepsilon$. This is in G as $S \Rightarrow \varepsilon$
- **Inductive Step**: (for $N \geq 0$). Assume that for some N, any string in language L containing at most N a's can be generated by G. *(inductive hypothesis)*.
  - Consider string $w \in L$ containing $N + 1$ a's.
  - Let w be denoted $w_1w_2w_3 \dots W_{2N+2}$, where each w_i in $\{a, b\}$. Suppose WLOG that w begins with a.
  - Let j denote the smallest index s.t. $w_1 \dots w_j \in L$. A value j must exist s.t. $2 \leq j \leq 2N + 2$.
  - Then $w = aXbY$, where $|aXb| = j$. Note that $X, Y \in L$ and X, Y have length no greater than $2N$.
  - By Inductive hypothesis, we know we can generate both X and Y from the start symbol S, and therefore we can generate w as follows:

$$S \Rightarrow aSbS \stackrel{\star}{\Rightarrow} aXbY = w$$

  - since we have shown that w can be generated from G, we have proven the inductive step.

■

Definition 2.1.5. A language is called **context-free** if it is generated by a context free grammar.

Claim 2.1.6. *Any Regular language is also a CFG.*

Proof. A table with the proof

RegEx	NFAs
a	$S \to a$
ε	$S \to \varepsilon$
$\emptyset$	$S \to S$
$R_1 \cup R_2$	$S \to S_1 \mid S_2$
$R_1 \circ R_2$	$S \to S_1S_2$
R_1^*	$S \to S_1S \mid \varepsilon$

■

Definition 2.1.7. A **parse tree** from a grammar $G = (V, \Sigma, R, S)$ is labeled tree rooted at S where

1. each leaf of a tree is labeled with some $a \in \Sigma$
2. each non-leaf of tree is labeled with some $a \in V$
3. if tree A contains subtree $x_1, x_2, \dots, x_m$, then $A \longrightarrow x_1x_2 \dots x_m \in R$

Ambiguity

Definition 2.1.8. When a string can be derived from a grammar in two fundamentally different ways, we say the string may be **ambiguously derived**.

Definition 2.1.9. When any string in a language is ambiguously derived in a grammar G, then G is said to be an **ambiguous grammar**.

Example. (Dangling Else Problem) `if` $x > 0$ `then if` $y < 0$ `then output yes else output no`

```
if x > 0                    if x > 0
then                        then
    if y < 0                    if y < 0
    then output yes             then output yes
    else output no          else output no
```

To show that a specific string s can be ambiguously derived, one can:

1. Give two different parse trees for s
2. Give two different *left-most* derivations for s, where leftmost is one in which east step, the left-most non-terminal remaining in the string is replaced.

Definition 2.1.10. A CFG G with start symbol S is in **Chomsky Normal Form** if every rule is in of one of the two following forms:

1. $A \to BC$
2. $A \to a$

where a is any terminal, A is any variable, and B and C are any variables except S. In addition, the rule $S \to \varepsilon$ is allowed, but ε may not appear in the grammar elsewhere.

Theorem 2.1.11. *Any context-free language can be expressed by a context-free grammar in Chomsky Normal Form.*

Proof. Provide algorithm to convert any CFG into an equivalent CFG in CNF.

1. Create a new sttart symbol S_0, and add rule $S_0 \to S$. This ensures that symbol isn't on RHS of any rule.
2. Remove rules from $A \to \varepsilon$, and "fix up": for each rule with an RHS that includes A, add a copy of that rule with A removed. Do this for all combinations.
3. Remove rules of form $A \to B$ (called "unit rules") and "fix": for each rule $B \to RHS$, add a rule $A \to RHS$.
4. Put remaining rules in proper form. May require introducing new variables. Reuse new variables where possible, to keep resulting grammar cleaner.

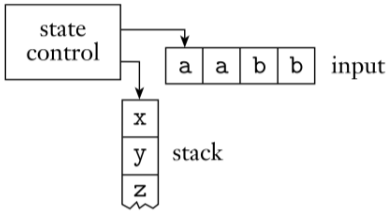
■

Topic 2.2 — Pushdown Automata

Remark. Pushdown Automata (PDAs), like NFAs **but** have extra memory: a stack. PDA can write symbols on the stack and read them back later. Stack characters come from set of symbols called Γ. Reading and writing only from top of the stack. More powerful than NFA — recognize some non-regular languages.

Definition 2.2.1. The formal definition of ***Pushdown Automaton*** P is a 6-tuple $(Q, \Sigma, \Gamma, \delta, q_0, F)$ where

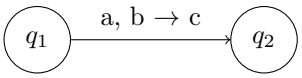
1. Q - is a finite **set of states**
2. Σ - finite set called **input alphabet**
3. Γ - finite set called **stack alphabet**
4. $\delta : Q \times \Sigma_\varepsilon \times \Gamma_\varepsilon \to P(Q \times \Gamma_\varepsilon)$ - **transition function**.
5. $q_0 \in Q$ - is the **start state**
6. $F \subseteq Q$ - is the **set of accepted state**



Transition Label

PDA transition function:

- $\delta : Q \times \Sigma_\varepsilon \times \Gamma_\varepsilon \to P(Q \times \Gamma_\varepsilon)$
- Given current state, input symbol, and TOS symbol, it outputs **set of states** that are legal pairs.



The start state q_1 can move to q_2 if (current state is $a \cap$ symbol at TOS us b). If transition is taken, then pop b of TOS and push c.

ε in transition label

1. $a, b \to c$
   - current input is a **and** TOS is b.
   - a is consumed, b is popped, c is pushed
2. $\varepsilon, b \to c$.
   - current TOS is b
   - no input consumed, b popped, c pushed.
3. $a, \varepsilon \to c$.
   - input symbol is a (don't care about TOS)
   - input a consumed, nothing popped, c pushed.
4. $a, b \to \varepsilon$.

- current input symbol is a **and** TOS is b
- input a consumed, b popped, nothing's pushed

Similarly, we can define more ε transitions.

5. $a, \varepsilon \to \varepsilon$
   - input a consumed, no change to stack
6. $\varepsilon, b \to \varepsilon$
   - no input consumed, b popped, no push
7. $\varepsilon, \varepsilon \to c$
   - no input consumed, nothing popped, c pushed
8. $\varepsilon, \varepsilon \to \varepsilon$
   - no input consumed, no change to stack

Theorem 2.2.2. *A language L is generated by a CFG **iff** a PDA recognizes language L. (2.2.3 & 2.2.4)*

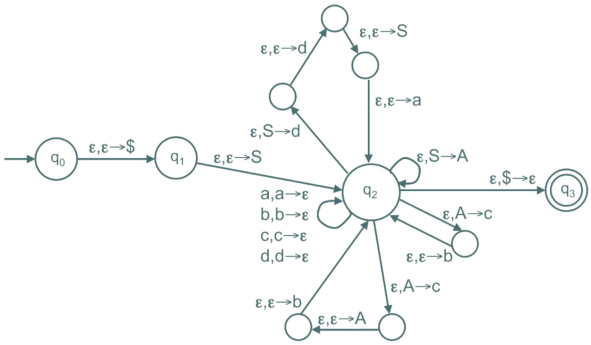
Claim 2.2.3. *($\Rightarrow$) Any language generated by a CFG can be recognized by a PDA*

Idea. Use stack to hold terminals and variables that are part of left-most derivation of a valid string, and match input with terminals on the stack.

Example. Given rules of G where $\Sigma = \{a, b, c, d\}$. Construct PDA.

$$S \to aSdd \mid A$$
$$A \to bAc \mid bc$$

PDA Construction



Claim 2.2.4. *($\Leftarrow$) Any language recognized by a PDA can be generated by a CFG*

We will not be required to know the details of the construction, just to know construction is possible. For full proof check textbook.

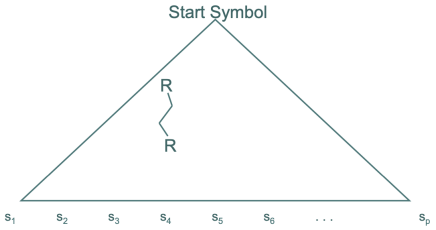
Topic 2.3 — Non-Context Free Languages

Example. Consider $A = \{a^n b^n c^n \mid n \geq 0\}$ Can we write a CFG that generates this, or a PDA that generates this?

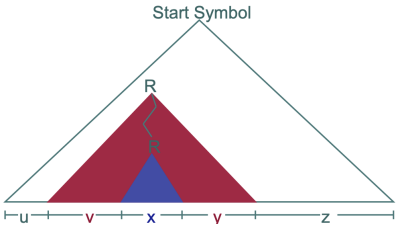
Theorem 2.3.1. ***(Pumping Lemma for CFLs)*** *If A is a context-free language, then there is a number p (pumping length) where if s is any string in A of length at least p., then s may be divided into **five** pieces $s = uvxyz$ such that:*

1. *for each $i \geq 0$, $uv^i xy^i z \in A$*
2. *$|vy| > 0$*
3. *$|vxy| \leq p$*

Proof. **Idea:**



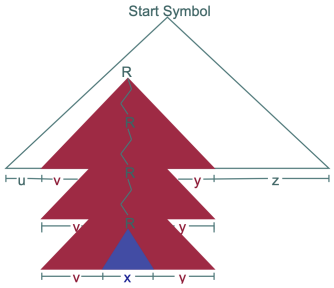
Let A be a CFL and G be a CFG that generates it. Suppose in the derivation of of a long string $s \in A$ some grammar symbol R must be repeated.



Each occurance R is the root of a subtree, and there is

- Name the string of terminals in small (blue) subtree x
- Name string of terminals in big (red) subtree vxy
- Name the string of terminals before red subtree u and after it z

Replace small blue subtree x with another occurence of big red subtree.



Here $uvvvxyyyz$ must also be in $L(G)$! ■

Example. Prove $A = \{a^n b^n c^n \mid n \geq 0\}$ is not Context-Free

Proof. Proof as shown

- Assume FSOC that A is a Context-free language. Then let p be the pumping length given by PL for CFLs.
- Consider string $s = a^p b^p c^p$, which is in A and has length $\geq p$.
- Since $|vxy| \leq p$, we can see that no matter how s is partitioned, vxy cannot contain mire than two different types of symbol.
- Thus, since $|vy| > 0$, when we pump up to $i = 2$, we are increasing the number of at least one, but at most two types of symbols.
- But there are three different types of symbols in the string so the number of occurrences of atleast one of the symbols is not changing. This means that the number of occurrences of one type of symbol is still at p, while the number of occurrences of some other symbol is now $p+k$, for some positive integer k.
- Since there are different number of occurrences of two symbols $s' = uv^2xy^2z$ is not in A. $\Rightarrow\Leftarrow$. Hence A is not context-free

■

Example. $B = \{w \mid w \in \{a,b,c\}^*, n_a(w) = \max(n_b(w), n_c(w))\}$. Prove B is not context free.

Proof. Proof is below.

- Assume FSOC, that B is a context-free language. Then let p be its pumping length by the PL
- Choose string $s = a^p b^p c^p$. We can see that $s \in B$ and $|s| \geq p$. Consider all partitions u, v, x, y, z such that $s = uvxyz$, $|uv| > 0$ and $|vxy| \leq p$.
- **CASE 1**: vy contains at leastt one a since $|vxy| \leq p$, vy can't also contain any c.
  Choose $i = 0$, So $s' = uv^0xy^0z = uxz$ conttains fewer than p a's', but exactly p c's, meaning s' can't be in B.
- **CASE 2**: vy contains at least one b or c, but no a's
  Choose $i = 2$, So $s' = uv^2xy^2z$. Contains only p a's but more than p b's or more than p c's (or both) . This means $s' \notin B$.
- Each case, we found that $s' \notin B$, so we have contradiction $\Rightarrow\Leftarrow$. Therefore B is not context-free.

■

Topic 3 — The Church-Turing Thesis

Topic 3.1 — Turing Machines

Remark. Turing Machines were proposed by Alan Turing in 1936 which is more powerful than an FA and PDA. More accurate model of general purpose computer.

- **Initial configuration**: input is on tape, starting at left end. Tape head positioned at left end.
- **Operations**: read a symbol, write a symbol, move one square (L/R)

Differences between FA and TM

- TM can both read and write on the tape and read from it
- Read-write head can move both left and right
- tape is infinite
- special states for rejecting and accepting take effect immediately

Definition 3.1.1. A ***Turing Machine*** is a 7-tuple $(Q, \Sigma, \Gamma, \delta, q_0, q_a, q_r)$

1. Q - is a finite **set of states**
2. Σ - finite set called **input alphabet** not containing $\_$ (blank)
3. Γ - **tape alphabet**, $\_ \in \Gamma$ and $\Sigma \subseteq \Gamma$
4. $\delta : Q \times \Gamma \to Q \times \Gamma \times \{L, R\}$ - **transition function**.
5. $q_0 \in Q$ - is the **start state**
6. $q_a \in Q$ - is the **accept state**
7. $q_r \in Q$ - is the **reject state** where $q_a \neq q_r$

Turing Machine Configurations

Definition 3.1.2. The ***configurations*** of a TM `M` describes its current state, current tape conttents, and current head location. A TM configuration is written uqv, where

- uv is the contents of the tape
- the head is currently pointing to first symbol in v q is the current state

More definitions

- Start configuration on input w is q_0w
- An *accepting configuration* includes state q_a
- A *rejecting configuration* includes state q_r
- Configuration C_1 *yields* configuration C_2 if the TM can legally go from C_1 to C_2
- Accepting and rejecting configurations are *halting configurations* and do not yield further configurations.

Definition 3.1.3. A TM `M` ***accepts*** input w if a sequence of configurations $C_1, C_2, \ldots, C_k$ exists where

- C_1 is the start configuration of `M` on input w
- Each C_i yields C_{i+1}
- C_k is an accepting configuration

Definition 3.1.4. The collection of strings that TM M accepts is the *language recognized by* M denoted $L(M)$

On input string w, a Turing Machine M can either:

1. enters an accept state q_a and *accepts* w
2. enters a reject state q_r and *rejects* w
3. does neither 1) or 2) and *"loops"* on w

If 1) or 2) occur → we say TM **halts**.
If 3) occurs → we say TM **does not halt**.

Example. Build TM M_1 to recognize $L_1 = \{0^n 1^n | n > 0\}$

1. If leftmost symbol is a 0, mark it with a dot and move right.
   Else REJECT
   [Need atleast one 0].
   $0 \to \dot{0}, R$
2. While the current symbol is a 0, write a 0 and move right
   $0 \to 0, R$
3. If current symbol is a 1, write 1 and move right
   Else current symbol is ␣ (blank) or anything else REJECT
   [Need atleast one 1, can't allow any other].
4. While symbol is 1, write 1 and move right
   $1 \to 1, R$
5. If current symbol is anything other than ␣. REJECT
   Else if current symbol is a ␣, write ␣ and move left
6. While current symbol is 1 or 0, do not change it and move left.
7. If current state is not $\dot{0}$, REJECT
   If current symbol is $\dot{0}$, replace with X, move right.
   [Mark off first 0.]
8. While current symbol is 0 or Y, do not change, move right
   [Looking for next unmatched 1 in string.]
9. If current symbol is ␣, REJECT
   [There were no unmatched 1's left]
   Else if current symbol is 1, write Y, move left.
   [Found a 1 to match 0, so mark it off].
10. While current symbol is Y or 0, no change, move left
    [Go back to leftmost 0 in string that's not matched.]
11. If current state is X, write an X and move right.
    Else, REJECT
    [Marked off everything left of here]
12. While current symbol is a Y, write Y and move right.
    [No unmatched 0 left].
13. If current symbol is 1, REJECT
    [There were unmatched 1's remaining]
    Else if current symbol is ␣, ACCEPT
    [All 0's and 1's were matched!]
    Else if current symbol is 0, replace with X, move right and to step 8.
    [Found new 0 to match, go look for 1].

Definition 3.1.5. Call a language ***Turing-recognizable*** a.k.a. *recognizable* if some Turing Machine recognizes it.

But TM need not always halt!. Suppose TM M recognizes L. If some string s is not in L

- M may reject s (enter reject state)
  OR
- M may "loop" s (not halt, M fails to enter accept/reject state).

Definition 3.1.6. A TM is a *decider* if it halts on all inputs. A TM M which is a decider that recognizes L is also said to *decide* L.

Definition 3.1.7. We call a language ***Turing-decidable*** or simply ***decidable*** if some Turing Machine *decides* it. That is, some language L is Turing-decidable if some TM M accepts on all strings in L AND rejects on all strings not in L - (not allowed to loop on any string)

Recognizers and Deciders

- If L is *Turing-recognizable* language, there exists some TM M such that
  - If $x \in L$, M will halt on and accept input x
  - If $x \notin L$, M won't "*halt and accept*" x - no guarentee M will do either halt and reject/run forever.
- If L is a *Turing-decidable* language, then there exists some TM M s.t.
  - If $x \in L$, M will halt on and accept input x
  - If $x \notin L$, M will halt on and reject input x.

Remark. Every TM that is a *decider* is also a *recognizer*.

Remark. Every language L that is *decidable* is also *Turing-recognizable* (known as recusively enumerable).

Remark. Regular ⊂ Context-Free ⊂ Turing Decidable ⊂ Turing-recognizable.

Topic 3.2 — Variations of TMs

Definition 3.2.1. We say the definition of TMs is **robust** - power of this model of computation is invariant to reasonable changes in definition.

Stay-put Turing Machines

We define a new transition function

$$\delta : Q \times \Gamma \to Q \times \Gamma \times \{L, R, S\}$$

where S means tape head "stays where it is". We can show this is no more powerful than a regular TM by showing one can stimulate the other.

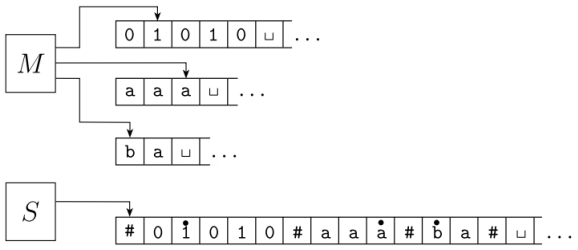
Multitape Turing Machines

We create a new type of TM called k-tape TM that could have k different tapes and k independent moving tape heads.

$$\delta : Q \times \Gamma_1 \times \ldots \times \Gamma_k \to Q \times \Gamma_1 \times \ldots \times \Gamma_k \times \underbrace{\{L, R\} \times \ldots \times \{L, R\}}_{k}$$

Theorem 3.2.2. *Every Turing Machine has an equivalent multi-tape Turing Machine.*

Idea. Store contents of all k tapes on our single tape use a long pass from left to right on single tape to determine the k current symbols - remember them in state control. All the tapes are separated in the single tape with a # and there is a dot above the values to keep track of the heads in the multitape.



Corollary 3.2.3. *A language is Turing-recognizable **iff** some multitape Turing machine recognizes it.*

Proof. If a Turing-recognizable language is recognized by a single-tape TM, it can also be recognized by a multitape TM. ■

Nondeterministic Turing Machines

Create a Nondeterministic TM that has the transition function as:

$$\delta : Q \times \Gamma \to \mathcal{P}(Q \times \Gamma \times \{L, R\})$$

No need for ε-transitions because it's not consuming input.

Theorem 3.2.4. *Every Nondeterministic TM has an equivalent TM.*

Idea. Convert a Nondeterministic TM to 3-tape Turing Machine, which can be converted to a standard TM. We design it through a breadth-first search to local the final state.

Enumerator

Definition 3.2.5. An ***enumerator*** is a Turing Machine with an attached printer, used as an output device to print strings.

Theorem 3.2.6. *A language is Turing-recognizable **iff** some enumerator recognizes it.*

Topic 3.3 — Algorithms

Definition 3.3.1. (Algorithm). An ***algorithm*** is a collection of simple instructions for carrying out some task.

Definition 3.3.2. (Church-Turing Thesis). Intuitive notion of algorithm equals Turing machine algorithm

Topic 4 — Decidability

Note. Investigate power of algorithms to solve problems.

Topic 4.1 — Decidable Languages

Question. For a given DFA D and string w, does D accept w?

Acceptence Problems for DFAs

Example. Express the problem as a language:

$$A_{DFA} = \{\langle D, w\rangle | D \text{ is a DFA that accepts string } w\}$$

Here A_{DFA} is the sett of all strings describing a DFA D and string w such that B accepts string w. Recall that languages are sets of strings. all items in language are of the form $\langle D, w\rangle$

- Suppose D_1 is a DFA such that $L(D_1) = 0^*11$

$$\langle D_1, 011\rangle \in A_{DFA}$$
$$\langle D_1, 1\rangle \notin A_{DFA}$$

- Suppose D_2 is a DFA s.t. $L(D_2) = (ab)^* \cup bbb$

$$\langle D_2, bbb\rangle \in A_{DFA}$$

- the $\langle\rangle$ indicated an ***encoding*** - we represent the DFA D and string w together as one string. The languages contains the encodings of all the DFAs together with strings that the DFAs accept.

Theorem 4.1.1. *A_{DFA} is a decidable language.*

Proof. M_{dfa} = "On input $\langle D, w\rangle$, where D is a DFA and w is a string"

1. Simulate D on input w. (Run $D(w)$)
2. If simulation ends in accept state, ACCEPT
   If simulation ends in non-accept state, REJECT

- Is M_{dfa} a decider? - Yes! Always halts
- Why or why not? Step 1 takes finitely many steps, Step 2 - trivial if statements.

■

Acceptence Problems for NFAs

Example. Express the problem as a language:

$$A_{NFA} = \{\langle N, w\rangle | N \text{ is a NFA that accepts string } w\}$$

Theorem 4.1.2. *A_{NFA} is a decidable language*

Idea. We already know a decider M_{dfa} for A_{DFA}. Use it as a helper method in our new TM M_{nfa}. We first need to convert the NFA N into a DFA

Proof. M_{nfa} = "On input $\langle N, w\rangle$, where N is an NFA and w is a string"

1. Let $\langle D\rangle \leftarrow T_{\text{nfa}\to\text{dfa}}(\langle N\rangle)$
2. Run $M_{\text{dfa}}(\langle D, w\rangle)$ (M_{dfa} is a TM that takes string as input)
   If M_{dfa} says accept, then ACCEPT, else REJECT.

- Is M_{nfa} a decider? Yes
- Why or why not? Because $T_{\text{nfa}\to\text{dfa}}$ takes a finite number of steps and M_{dfa} is a decider, so M_{dfa} must halt on any input.

■

Remark. This is similar proof for A_{REX} where regular expressions are decidable languages.

Emptiness Testing for DFAs

Question. For a given automaton D, does it accept any strings at all? In the preceding theorem we determined if a automaton accepts a particular string.

Example. Express the problem as a language:

$$E_{DFA} = \{\langle D\rangle | D \text{ is a NFA and } L(D) = \emptyset\}$$

Is E_{DFA} decidable? How might we prove this? Notice here the language of the DFA is empty, so it accepts no string and has no final states.

Idea. • Assume alphabet is say $\{0, 1\}$. Consider TM Q = "On input $\langle A\rangle$ where A is a DFA:

1. For $w = \varepsilon, 0, 1, 00, 01, 10, 11, \ldots$ (infinite list)
   Simulate A on input string w
   if A accepts, REJECT (as there are no final states).
   End FOr
2. ACCEPT"

- Is TM Q a decider? No!
- Why or why not? For loop could run infinitely!

Theorem 4.1.3. *E_{DFA} is a decidable language*

Proof. Need a TM $M_{\text{dfa}}\emptyset$ which decides E_{dfa}. TM $M_{\text{dfa}\emptyset}$ = "On input $\langle A\rangle$, where A is a DFA.

1. Mark the start state of A.
2. Repeat until no new states get marked: Mark any unmarked state that has an incoming transition from some already marked state.
3. If no final state of A is marked, then ACCEPT, else REJECT"

Note: No matter what string $\langle A\rangle$ is given to $M^{\emptyset}_{dfa}$ as input, the TM runs for only finitetly-many steps. And it accepts all strings in E_{dfa} but no others. So it ***decides*** E_{dfa} ■

Equivalence Problem for DFAs

Theorem 4.1.4. *EQ_{DFA} is a decidable language*

Idea. Need a TM M_{dfa} = to decide EQ_{DFA}. On input $\langle A, B\rangle$ where A and B are DFAs, we construct new DFA D where

$$L(D) = \big(L(A) \cap \overline{L(B)}\big) \cup \big(L(B) \cap \overline{L(A)}\big) = L(A) \triangle L(B)$$

which is the symmetric difference. If $L(D)$ is zero, then we can conclude that $L(A) = L(B)$, otherwise $L(A) \neq L(B)$.

Proof. We use the framework (Given) / Want / Construction / Correctness of Construction.
Want TM M_{dfa} = to decide EQ_{DFA}, meaning

- if $\langle A, B\rangle \in EQ_{DFA}$, then $M_{dfa}\big(\langle A, B\rangle\big)$ ACCEPTS
- if $\langle A, B\rangle \notin EQ_{DFA}$, then $M_{dfa}\big(\langle A, B\rangle\big)$ REJECTS

Construction

1. $\langle \overline{A}\rangle \leftarrow T^{comp}_{dfa}\big(\langle A\rangle\big)$
2. $\langle \overline{B}\rangle \leftarrow T^{comp}_{dfa}\big(\langle B\rangle\big)$
3. $\langle \text{left}\rangle \leftarrow T^{\cap}_{dfa}\big(\langle A, \overline{B}\rangle\big)$
4. $\langle \text{right}\rangle \leftarrow T^{\cap}_{dfa}\big(\langle B, \overline{A}\rangle\big)$
5. $\langle D\rangle \leftarrow T^{\cup}_{dfa}\big(\langle \text{left}, \text{right}\rangle\big)$
6. Run $M^{\emptyset}_{dfa}(\langle D\rangle)$. If it accepts, then ACCEPT. if it rejects, then REJECT.

Correctness
Each transform-computing TM used by $M^{=}_{dfa}$, and $M^{\emptyset}_{dfa}$ always halt in a finite number of steps, so $M^{=}_{dfa}$ is a decider. $M^{=}_{dfa}$ decides EQ_{DFA} specifically, since

- if $\langle A, B\rangle \in EQ_{DFA}$, then $L(D)$ is empty, causing $M_{dfa}^{\emptyset}$ to accept, meaning $M_{dfa}^{=}$ ACCEPT.
- if $\langle A, B\rangle \notin EQ_{DFA}$, then $L(D)$ is non-empty and there is atleast one string w which one of A or B accepts and the other rejects. causing $M_{dfa}^{\emptyset}$ to reject, meaning $M_{dfa}^{=}$ REJECTS.

■

Template of Proof

Want

- if x in L, $M(x)$ ACCEPTS
- if x not in L, then $M(x)$ REJECTS

Construction

TM M = On input x

- <fill in pseudocode here>
-

Ensure you check if it's a decider?.. always halts? How do you know?

Correctness of Construction

M decides on L, since

- if $x \in L$, then <fill in argument> ... so $M(x)$ ACCEPTS
- if $x \notin L$, then <fill in argument> ... so $M(x)$ REJECTS

Acceptance Problem for CFGs

Test whether two context-free grammars generate the same language.

$EQ_{CFG} = \{\langle G, H\rangle | G \text{ and } H \text{ are CFGs and } L(G) = L(H)\}$

Remark. We cannot go through all tthe strings generated by G as it's infinite - would give a recognizer and not a decider. We need to convertt G into a CNF. Derivattion of a string of length $n > 0$ will require exactly $2n-1$ steps if grammar is in Chomsky normal form. For some w, that is only a finite number! Check and reject string if w is not generated.

Proof. **Want** TM M_{cfg} = to decide A_{CFG}, meaning

- if $\langle G, w\rangle \in A_{CFG}$, then $M_{cfg}\big(\langle G, w\rangle\big)$ ACCEPTS
- if $\langle G, w\rangle \notin A_{CFG}$, then $M_{cfg}\big(\langle G, w\rangle\big)$ REJECTS

Construction

TM M_{cfg} = "On input $\langle G, w\rangle$, where G is a CFG and w is a string"

1. Let $\langle G\rangle \leftarrow T_{cfg\to cnf}(\langle G\rangle)$
2. Let $n \leftarrow |w|$
3. if $n = 0$, then list all derivatives with 1 step
   Else list all derivatives with $2n-1$ steps
4. If any listed derivations generate w, then ACCEPT
   Else REJECT

Correctness

Since we convert grammar G into an equivalent grammar G' in CNF, G' accepts the same strings as G. Additionally, properties of CNF guarentee that a string of length n is generated by a grammar G' in CNF:

- is generated in exactly 1 step when $n = 0$
- is generated in exactly $2n-1$ steps when $n > 0$

So, S is a decider for A_{CFG} since:

- if $\langle G, w\rangle$ is in A_{CFG}, then w will be generated in Step 3, so $M_{cfg}\big(\langle G, w\rangle\big)$ ACCEPTS
- if $\langle G, w\rangle$ is not in A_{CFG}, then $M_{cfg}\big(\langle G, w\rangle\big)$ REJECTS

■

Emptiness Testing for CFGs

$$E_{CFG} = \{\langle G\rangle | G \text{ is a CFG and } L(G) = \emptyset\}$$

Idea. Cannot decide this language by enumerating all possible strings, there are infinite. Instead we see if grammar's start variable can generatte any string of terminals. Begin by marking all terminal symbols in grammar. Then mark all symbols A where RHS symbols in rule such as $A \to U_1, \ldots, U_k$ have been marked. Go through list of rules this way repeatedly until no new symbols have been marked. Is startt symbol marked? If so, $L(G)$ is not empty

Proof. We construct TM R = "On input $\langle G\rangle$, where G is a CFG:

1. Let $\langle G\rangle \leftarrow T_{cfg\to cnf}\big(\langle G\rangle\big)$
2. Mark all terminal symbols in G'
3. Repeat until no new variables get marked:
   Mark each new variable A where G' contains rule $A \to U_1U_2\ldots U_k$ and each U_i has already been marked.
   End repeat.
4. If start symbol is not marked, then ACCEPT
   Else REJECT."

Remark. In a sense, we work backwards until we find that the start symbol can be made into a string or not.

■

Equivalence Problem for CFGs

$EQ_{CFG} = \{\langle G, H\rangle \mid G \text{ and } H \text{ are CFGs and } L(G) = L(H)\}$

Remark. EQ_{CFG} is ***not*** decidable. For DFA we used the fact that DFAs are closed under the complement and intersection. However, this is not true for CFGs so will not work here. The strategy of trying all sttrings w_i to see if one grammar can generate it and other can't does work as there are infinite number of strings.

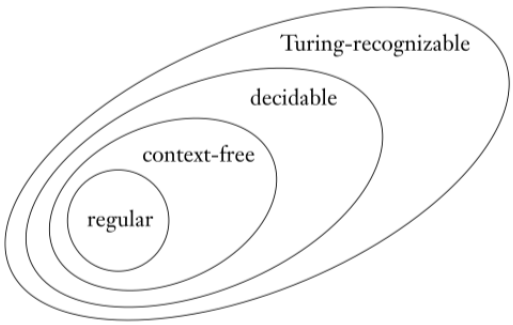
Construction

TM R = "On input $\langle G, H\rangle$ where G, H are CGFs

1. Let $L = \{w_1, w_2, \ldots\}$ be the dictionary ordering of strings over union of alphabets of G and H
2. For each i in $\{1, 2, 3, \ldots\}$
   (a) Let $g = M_{cfg}\big(\langle G, w_i\rangle\big)$
   (b) Let $h = M_{cfg}\big(\langle H, w_i\rangle\big)$
   (c) if $g \neq h$, then REJECT

   End For
3. ACCEPT - this is unreachable since infinite number of strings."

Relationship among classes of languages.



Topic 4.2 — Undecidable Languages

Definition 4.2.1. If L is a **Turing-recognizable** language, there exists some TM M such that

- If $x \in L$, M will halt on and accept input x
- If $x \notin L$, M won't "halt and accept" input x - can either halt and reject or simply run forever.

Definition 4.2.2. If L is a **co-Turing-recognizable** language, there exists some TM M such that

- M recognizes the **complement** of L, meaning
- If $x \in \text{comp}(L)$, M will halt on and accept input x
- If $x \notin \text{comp}(L)$, M won't "halt and accept" input x - can either halt and reject or simply run forever.

Demonstrating co-Turing recognizability

- Determine and state what comp(L) is. Common mistake is taking complement wrong
- Prove that comp(L) is Turing-recognizable - use Want/ Construction / Correctness of Construction format for this sttep

Example. Prove that EQ_{CFG} is co-Turing-recognizer. where

$$EQ_{CFG} = \{\langle G, H\rangle | G, H \text{ are CFGs and } L(G) = L(H)\}$$
$$\text{comp}(EQ_{CFG}) = \{\langle G, H\rangle | G, H \text{ are CFGs and } L(G) \neq L(H)\}$$

Proof.
Want: TM R to recognize comp(EQ_{CFG}) meaning

- if $\langle G, H\rangle$ in comp(EQ_{CFG}), $R\big(\langle G, H\rangle\big)$ ACCEPTS.
- if $\langle G, H\rangle$ not in comp(EQ_{CFG}), $R\big(\langle G, H\rangle\big)$ REJECTS or LOOPS

Construction: TM R = "On input $\langle G, H\rangle$, where G, H are CFGs:

- Let $L = \{w_1, w_2, w_3, \ldots\}$ be a shortlex ordering of strings over union of alphabets of G, H
- For each i $\{1, 2, 3, \ldots\}$ (infinite loop!)
  1. Let $g = M_{cfg}\big(\langle G, w_i\rangle\big)$
  2. Let $h = M_{cfg}\big(\langle H, w_i\rangle\big)$
  3. If $g \neq h$, then ACCEPT

  End For".

Correctness of Construction: We first note that M_{cfg} is a decider, so always halts. TM R recognizes comp(EQ_{CFG}) because

- if $\langle G, H\rangle$ in comp(EQ_{CFG}), then there exists some string w which one of G, H can generate, but the other can't. Eventually, our infinite loop will check string w, and therefore R will ACCEPTS.
- if $\langle G, H\rangle$ not in comp(EQ_{CFG}), then $L(G) = L(H)$, so there is no string w which can be generated by one of G or H and not the other. Thus our loop runs forever, meaning R LOOPS.

■

Countable and Uncountable Sets

Definition 4.2.3. A set S is **countable** if and only if it is finite or there is a bijection (one-to-one and onto) between the $\mathbb{N}$ and S

Theorem 4.2.4. $\mathbb{R}$ *is uncountable.*

Proof. Use diagonalization proof. ■

Question. Is the set of all languages over some alphabet Σ countable?

Answer: No! we use diagonalization argument.

Theorem 4.2.5. *The set of all Turing Machines is countable*

Idea. We observe that the set of all strings Σ^* is countable for any alphabet Σ - with only finitely many strings of each length, we may form a list of Σ^* by writing down all strings of length 0, 1, 2, so on.
The set of all Turing Machines is countable because each Turing Machine M has an encoding into string $\langle M \rangle$. If we simply omit those strings that are not legal encodings of Turing Machines, we can obtain a list of all Turing Machines.

Remark. We know that the number of languages is uncountable. We know the set of all TMs is countable. Hence, there are languages that cannot be recognized by a TM.

Undecidable Language

Idea: use a *universal TM* U to simulate M on input w
TM U = "On input $\langle M, w \rangle$, where M is a TM and w is a string:

1. Simulate M on input w
2. If M accepts, then ACCEPT IF M rejects, then REJECT."

We know that U recognizes A_{TM} since - 1. if $M(w)$ accepts then $U(\langle M, w \rangle)$ accepts. 2. If $M(w)$ does not accepts then $U(\langle M, w \rangle)$ doesn't either - may reject or loop forever. Hence U does not decide A_{TM}

Theorem 4.2.6. *A_{TM} is undecidable*

Proof. Proof by contradiction.. Assume A_{TM} is decidable, and suppose H is a decider for it. This means that on input $\langle M, w \rangle$:

- If M accepts w, then H accepts $\langle M, w \rangle$
- If M rejects or loops on w, then H rejects $\langle M, w \rangle$

Now we construct new TM D which uses the TM H as a subroutine:
TM D = "On input $\langle M \rangle$, where M is a TM:

1. Run H on input $\langle M, \langle M \rangle \rangle$
2. If H accepts, REJECT. If H rejects, ACCEPTS."

So what does D do on input $\langle M \rangle$.

1. D ACCEPTS $\langle M \rangle$ if M does not accept $\langle M \rangle$
2. D REJECTS $\langle$ M $\rangle$ if M accepts $\langle M \rangle$

■