CS4232 Lecture Notes

November 25, 2020

Contents

1 SUMMARY

1.1 All The Automata/Grammars

DFA/NFA: $A = (Q, \Sigma, \delta, q_0, F)$

- Transition: $\delta(q, a) = q'$ or $\{q'\}$

PDA: $P = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$

- Transition: $\delta(q, a, X) = (q', X')$ or $\{(q', X')\}$

TM: $M = (Q, \Sigma, \Gamma, \delta, q_0, B, F)$

- Transition: $\delta(q, X) = (q', X', L/R)$

CFG: $G = (V, T, P, S)$

UG: $G = (N, \Sigma, S, P)$

1.2 DFA/NFA: $A = (Q, \Sigma, \delta, q_0, F)$

DFA transition function: $\delta(q, a) = p$

- $\hat{\delta}(q, xa) = \delta(\hat{\delta}(q, x), a)$

NFA transition function: $\delta(q, a) = \{p_1, p_2, \ldots\}$

- $\hat{\delta}(q, xa) = \bigcup_{p \in \hat{\delta}(q,x)} \delta(p, a)$

DFA language accepted: $L(A) = \{w \mid \hat{\delta}(q_0, w) \in F\}$

NFA language accepted: $L(A) = \{w \mid \hat{\delta}(q_0, w) \cap F \neq \emptyset\}$ — i.e. *any one* state is accepting

Equivalence of NFA and DFA: Convert NFA to DFA through *subset construction*

- Idea: in constructed DFA, we simulate NFA by having each state in DFA represent *set of states* in NFA

ϵ-NFA: $Eclose(q)$ refers to all states reachable from q with as many or no ϵ transitions

Equivalence of ϵ-NFA and DFA: Same as above, can convert.

Minimising DFA: *Table building algorithm*, using idea of *distinguishable pairs*

- BEFORE YOU START, DRAW OUT THE TRANSITION TABLE. IT HELPS.
- Base case: (p, q) is distinguishable, where $p \in F$ and $q \notin F$
- Inductive step: (p, q) is distinguishable if there exists some alphabet a such that $\delta(p, a)$ is distinguishable from $\delta(q, a)$
- i.e. For each cell in the table, check if some alphabet causes a transition to a previously X-ed cell

1.3 Regular Expressions

DFAs and regular expressions are equivalent in power.

Convert DFA to regex

Let $R^k_{i,j}$ be regex for strings that can be formed going from state i to state j, using intermediate states $\leq k$.

Language accepted by DFA $= \sum_{j \in F} R^n_{1,j}$ (where 1 is the start state)

Base case: definition of $R^0_{i,j}$ (i.e. no intermediate state)

- If $i \neq j$: $R^0_{i,j} = a_1 + a_2 + \ldots + a_m$, where a_r are symbols such that $\delta(i, a_r) = j$

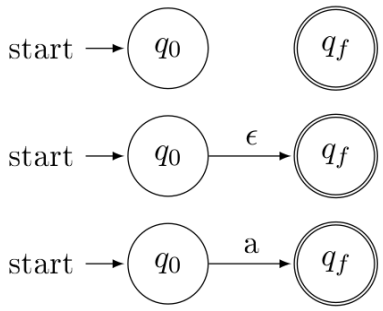
- If $i = j$: $R^0_{i,j} = \epsilon + a_1 + a_2 + \ldots + a_m$, where a_r are symbols such that $\delta(i, a_r) = i$

Inductive step: $R^{k+1}_{i,j} = R^k_{i.j} + R^k_{i,k+1}(R^k_{k+1,k+1})^* R^k_{k+1,j}$

Convert regex to ϵ-NFA: Thompson's Construction

Base cases

- ϕ: $A = (\{q_0, q_f\}, \Sigma, \delta, q_0, \{q_f\})$, where δ is an empty function (i.e. no transitions)
- ϵ: $A = (\{q_0, q_f\}, \Sigma, \delta, q_0, \{q_f\})$, where $\delta(q_0, \epsilon) = q_f$
- a: $A = (\{q_0, q_f\}, \Sigma, \delta, q_0, \{q_f\})$, where $\delta(q_0, a) = q_f$



Inductive Cases

For all these examples, let:

- Let $A_1 = (Q_1, \Sigma, \delta_1, q^1_0, F_1)$ be the automata for r_1
- Let $A_2 = (Q_2, \Sigma, \delta_2, q^2_0, F_2)$ be the automata for r_2
- Let δ contain all existing transitions from δ_1 and δ_2; we'll add additional transitions to δ later
- No overlapping states, i.e. $Q_1 \cap Q_2 = \phi$

$r_1 + r_2$ (or) — see (c)

- $\delta(q_0, \epsilon) = \{q^1_0, q^2_0\}$
- $\delta(q^1_f, \epsilon) = \{q_f\}$ for $q^1_f \in F_1$
- $\delta(q^2_f, \epsilon) = \{q_f\}$ for $q^2_f \in F_2$

$r_1 \cdot r_2$ (and) — see (b)

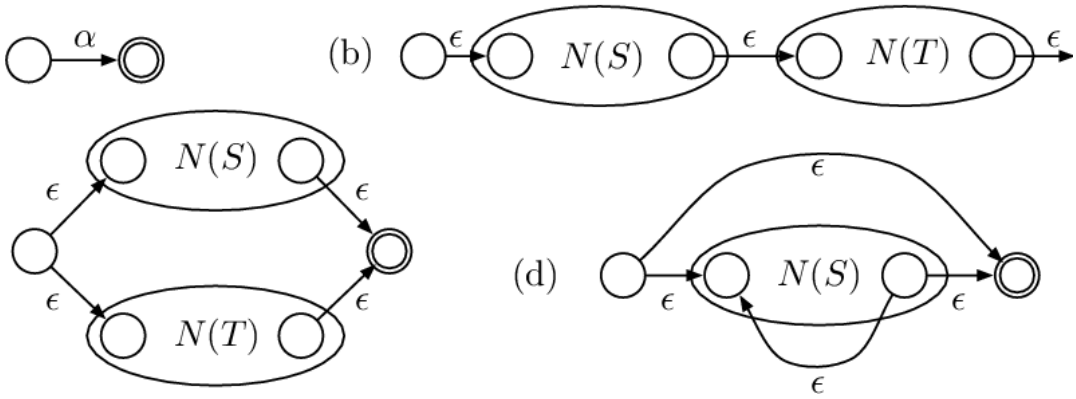
- $\delta(q_0, \epsilon) = \{q^1_0\}$
- $\delta(q^1_f, \epsilon) = \{q^2_0\}$
- $\delta(q^2_f, \epsilon) = \{q_f\}$

r^*_1 — see (d)

- $\delta(q_0, \epsilon) = \{q^1_0, q_f\}$
- $\delta(q^1_f, \epsilon) = \{q^1_0, q_f\}$

r^+_1 — see (d) but without start→final arrow

- $\delta(q_0, \epsilon) = \{q^1_0\}$
- $\delta(q^1_f, \epsilon) = \{q^1_0, q_f\}$



Properties of regex

- $L(M+N) = LM + LN$
- $L((M+N)^*) = L((M^*N^*)^*)$

Example: show that $L((R+S)^*) = L((R^*S^*)^*)$

- $\subseteq$: $L(R+S) \subseteq L(R^*S^*)$ since $L(R) \subseteq L(R^*S^*)$ and $L(S) \subseteq L(R^*S^*)$; hence $L((R+S)^*) \subseteq L((R^*S^*)^*)$
- $\supseteq$: $L(R^*S^*) \subseteq L((R+S)^*(R+S)^*) = L((R+S)^*)$; hence $L((R^*S^*)^*) \subseteq L(((R+S)^*)^*) = L((R+S)^*)$

Closure of regex

- Closed under union, intersection
- Closed under complementation, difference
- Closed under concatenation
- Closed under reversal
- Closed under *homomorphism*: where you replace a character with some string

Pumping Lemma for Regular Languages

Let L be a regular language. Then there exists n such that every sufficiently long string $w \in L$ of length $\geq n$, we can break $w = xyz$, such that:

- $y \neq \epsilon$
- $|xy| \leq n$
- $xy^kz \in L$ for all $k \geq 0$

To prove that some language L is NOT regular:

- Suppose otherwise, that L is regular.
- Let n be as in the pumping lemma.
- Let string $w = a^nb^n$ (choose this) $= xyz$ as in the pumping lemma
- Show that you can choose some k such that $xy^kz \notin L$, contradiction

1.4 CFGs: $G = (V, T, P, S)$

Right-Linear Grammar: all productions are of form $A \to wB$ or $A \to B$. Every regular language can be generated by a right-linear grammar, and vice versa.

Removing Useless Symbols

1. Find all *generating* symbols, and remove non-generating productions
   - Base case: all productions to only terminals (including ϵ) are generating
   - Inductive step: all productions to only generating symbols are generating
2. Find all *reachable* symbols, and remove non-reachable productions
   - Base case: S is reachable
   - Inductive step: If A is reachable and $A \to \alpha$, then all symbols in α are reachable

Converting to CNF

CNF: all productions of form $A \to BC$ or $A \to a$

1. Eliminate ϵ-productions
   - Find all nullable symbols (note that if S is nullable, this method will generate $L - \{\epsilon\}$)
   - Replace nullable symbols with 2^n combinations
   - Remove ϵ-productions
2. Eliminate unit productions
   - Find all non-trivial unit pairs
   - Add productions from unit pairs
   - Remove unit productions
3. Convert productions to max length 2 CNF
   - Change $A \to X_1X_2X_3$ to $A \to Z_1B_2$ and $B_2 \to Z_2Z_3$, where $Z_i \to X_i$ (terminal) or $Z_i = X_i$ (non-terminal)

Pumping Lemma for CFL

Let L be CFL. Then there exists n such that every sufficiently long string $w \in L$ of length $\geq n$, we can break $z = uvwxy$, such that:

- $vx \neq \epsilon$
- $|vwx| \leq n$
- $uv^iwx^iy \in L$ for all $i \geq 0$

To prove that some language L is NOT CFL:

- Suppose otherwise, that L is CFL
- Let $n > 1$ be as in the pumping lemma
- Let string $z = a^nb^nc^n$ (choose this) $= uvwxy$ as in the pumping lemma
- Show that you can choose some i such that $uv^iwx^iy \notin L$, contradiction

Closure

Closure under Substitution

- Consider mapping each terminal a to a CFL L_a, where $s(a) = L_a$.
- If L is CFL and s is a substitution such that $s(a) = L_a$ is a CFL, then $\bigcup_{w \in L} s(w)$ is a CFL

Closure under Reversal: if L is CFL, then L^R is CFL

($\star$) Closure under Intersection: if L is CFL and R is regular, then $L \cap R$ is CFL

- CFLs are NOT closed under intersection! e.g. $L = \{a^n b^n c^m \mid n, m \geq 1\} \cap \{a^m b^n c^n \mid n, m \geq 1\} = \{a^n b^n c^n \mid n \geq 1\}$ is NOT CFL
- Example: $L = \{w \mid w \in \{a, b, c\}^* \text{ and } \#_a(w) = \#_b(w) = \#_c(w)\}$ is NOT CFL
  - Suppose otherwise that it's a CFL, then $L \cap a^*b^*c^* = \{a^n b^n c^n \mid n \geq 0\}$ is CFL, contradiction

Closure under Union: if L_1 and L_2 are CFLs, then $L_1 \cup L_2$ is CFL

Dynamic Programming $O(n^3)$ Algorithm: CYK parsing

Tests if a string $w = a_1 \dots a_n$ can be generated by a CFL.

- Let $X_{i,j}$ be the set of nonterminals that generate the string $a_i a_{i+1} \dots a_j$. Then see if $S \in X_{1,n}$.
- Base case: $X_{i,i}$ is set of non-terminals that generate a_i
- Inductive step: $X_{i,j}$ contains all A such that $A \to BC$, where $B \in X_{i,k}$ and $C \in X_{k+1,j}$ for some $i \leq k < j$ — i.e. B generates $a_i a_{i+1} \dots a_k$ and C generates $a_{k+1} \dots a_j$

1.5 PDA: $P = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$

Two modes of acceptance: *final state* or *empty stack*. Both are equivalent in non-deterministic PDA.

CFGs and NPDAs are equivalent

Deterministic PDAs are weaker than non-deterministic PDAs; further, DPDA acceptance by final state and empty stack are different

1.6 TM: $M = (Q, \Sigma, \Gamma, \delta, q_0, B, F)$

L is RE: some TM accepts L

L is recursive: some TM accepts L and always halts on all inputs

Undecidable Languages

$L_d = \{w_i \mid w_i \notin L(M_i)\}$

- L_d is not RE
- $\bar{L}_d$ is RE-but-not-recursive

$L_u = \{(M, w) \mid M \text{ accepts } w\}$

- L_u is RE-but-not-recursive
- $\bar{L}_u$ is not RE

$L_e = \{M \mid L(M) = \emptyset\}$

- L_e is not RE
- L_{ne} is RE-but-not-recursive

1.7 Undecidability

Languages accepted by TMs are RE. Subset of RE accepted by TMs that *always halt* are recursive.

Complementation

- L is recursive $\leftrightarrow$ $\bar{L}$ is recursive
- L and $\bar{L}$ are RE $\leftrightarrow$ L is recursive
- L is RE-but-not-recursive $\leftrightarrow$ L is not RE

Union/intersection

- Union or intersection of two RE languages is also RE. (Tut 9 Q3)

Decidability

- Decidable = recursive
- Undecidable = not recursive

(Are all context-free languages recursive/decidable???) (Algorithm for CFL we did in class???)

Rice's Theorem: any nontrivial property of RE languages is undecidable.

PCP: undecidable problem

1.8 Miscellaneous things

- $L_2 - L_1 = L_2 \cap \bar{L}_1$

Divisibility Rules

- Divisible by 2: last digit is divisible by 2
- Divisible by 3: take sum and see if it's divisible by 3
- Divisible by 4: last two digits are divisible by 4
- Divisible by 5: last digit is 0 or 5
- Divisible by 6: check if both divisible by 2 and 3
- Divisible by 7: $10x + y$ is divisible by 7 if $x - 2y$ is divisible by 7
- Divisible by 8: last three digits are divisible by 8

2 Assessment

Tutorials	10%
Midterms (x2)	25% each
Final exam	40%

3 Central Concepts of Automata Theory

3.1 Alphabets

Alphabet Σ: a finite, non-empty set of symbols.

- E.g. $\{0, 1\}$, $\{A, B, \ldots, Z\}$

Powers of an alphabet

- $\Sigma^0 = \{\epsilon\}$
- $\Sigma^1 = \{0, 1\}$
- $\Sigma^2 = \{00, 01, 10, 11\}$
- $\Sigma^{\leq 2} = \Sigma^0 \cup \Sigma^1 \cup \Sigma^2$
- Σ^* (all finite length strings)

3.2 Strings

String: finite sequence of symbols, chosen from a given alphabet

- E.g. 01001, *acbbe*
- Number of strings, over a finite alphabet Σ, is countable
- Empty string: ϵ

Substring: *continuous* sub-part of a string

- E.g. *ba* is a substring of *ababb*

Concatenation: concatenation of a and b is ab

3.3 Language

Language L: set of strings over an alphabet Σ

- E.g. $L = \{00, 01, 1101\}$
- E.g. $L = \{x : x \text{ is a binary representation of a prime number}\}$

Operations

- $L_1 \cdot L_2$ (i.e. $L_1 L_2$) $= \{xy : x \in L_1, y \in L_2\}$
- $L^* = \{\epsilon\} \cup L \cup LL \cup LLL \ldots$ ($\geq$0 number of times)
- $L^+ = \{\epsilon\} \cup L \cup LL \cup LLL \ldots$ ($\geq$1 number of times)

Number of languages over any non-empty alphabet is *uncountable*

4 Finite Automata

Regular languages: these are languages accepted by finite automata

What are finite state automata?

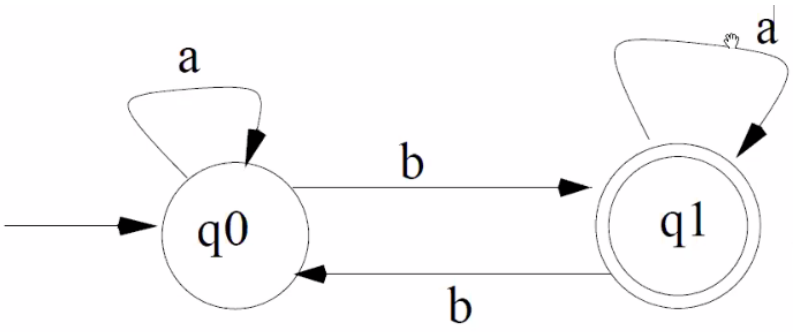
- Examples: a switch with 2 states, ON and OFF, pushing it toggles the state; lexical analysers
- Two flavours: DFA and NFA

4.1 Deterministic Finite Automata (DFA)

DFA definition: $A = (Q, \Sigma, \delta, q_0, F)$

- Q, a finite set of states
- Σ, a finite set of input symbols
- δ, the transition function: it takes a state from Q and a letter from Σ, then deterministically returns a next state
- q_0, a starting state
- F, a set of final/accepting states

Example: DFA accepts strings containing an odd number of b symbols, where $\Sigma = \{a, b\}$



Methods of Representing DFAs

Transition diagrams

- Circles for states
- Arrows for transitions
- Starting state: denoted with arrow labelled start
- Final/accepting states: denoted with double circles

Transition tables

Example: DFA accepts strings containing a substring 00

	0	1
q_0	q_1	q_0
q_1	q_2	q_0
q_2	q_2	q_2

Transition Function for Strings (DFAs)

Let's define the transition function $\hat{\delta}$ for not only characters, but strings:

- Basis: $\hat{\delta}(q, \epsilon) = q$
- Induction: $\hat{\delta}(q, xa) = \delta(\hat{\delta}(q, x), a)$ — where x is a string and a is an additional character

Language Accepted by DFAs

Language accepted by a DFA, $L(A)$, is the set of all strings, starting with q_0, that lead to accepting states in F

- $L(A) = \{w \mid \hat{\delta}(q_0, w) \in F\}$

Dead and Unreachable States in DFAs

Dead state: a state from which you can *never* reach an accepting state

- q is a dead state $\leftrightarrow \forall w \in \Sigma^*, \hat{\delta}(q, w) \notin F$

Unreachable state: a state you can *never* reach from the starting state

- q is an unreachable state $\leftrightarrow \forall w \in \Sigma^*, \hat{\delta}(q_0, w) \neq q$

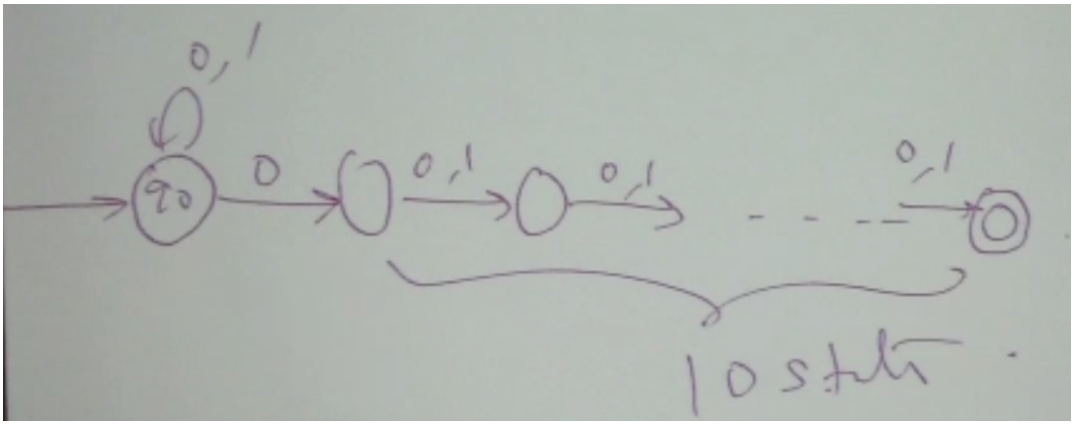
4.2 Non-Deterministic Finite Automata (NFA)

NFA definition: $A = (Q, \Sigma, \delta, q_0, F)$ (same as before)

What's different compared to DFAs?

- Now, transition function δ maps each input (state + symbol) to a *set* (!) of states, not exactly one
- So after accepting some input string, you end up with a *set* of possible states, instead of a *single* state!

Example: Language where the 10th-last symbol is a 0



Transition Function for Strings (NFAs)

Basis: $\hat{\delta}(q, \epsilon) = \{q\}$

Induction: $\hat{\delta}(q, xa) = \bigcup_{p \in \hat{\delta}(q,x)} \delta(p, a)$ — where x is a string and a is an additional character

Language Accepted by NFAs

A string is accepted in the language for the NFA, if ANY if the states in the NFA is an accepting state

$L(A) = \{w \mid \hat{\delta}(q_0, w) \cap F \neq \phi\}$ — ANY/AT LEAST ONE state is an accepting state!

4.3 Equivalence of DFA and NFA

(Note: a DFA is also an NFA.)

How to show that a language accepted by NFA is also accepted by some DFA?

- We convert a NFA to a DFA, then see if the DFAs are equivalent.

Converting NFA to DFA

Idea: subset construction

- To simulate a NFA, we need to keep track of a *set* of states
- Hence, in the constructed DFA, we use a *single* state to represent a *set* of states in the original NFA

Suppose we have NFA $A = (Q, \Sigma, \delta, q_0, F)$ Define the DFA $A_D = (Q_D, \Sigma_D, \delta_D, \{q_0\}, F_D)$ to be constructed as follows:

- $Q_D = \{S \mid S \subseteq Q\}$
- $F_D = \{S \mid S \subseteq Q \text{ and } S \cap F \neq \phi\}$
- $\delta_D(S, a) = \bigcup_{q \in S} \delta(q, a)$

Proof of Equivalence of NFA and DFA

Claim: for any string w, $\hat{\delta}_D(\{q_0\}, w) = \hat{\delta}(q_0, w)$

Proof: by induction on length of string w

- Base case: $w = \epsilon$ — then $\hat{\delta}_D(\{q_0\}, \epsilon) = \{q_0\} = \hat{\delta}(q_0, w)$
- Induction step:

$$\begin{aligned}
\hat{\delta}_D(\{q_0\}, wa) &= \delta_D(\hat{\delta}_D(\{q_0\}, w), a) \\
&= \bigcup_{q \in \hat{\delta}_D(\{q_0\}, w)} \delta(q, a) \\
&= \bigcup_{q \in \hat{\delta}(q_0, w)} \delta(q, a) \\
&= \hat{\delta}(q_0, wa)
\end{aligned}$$

$\therefore \hat{\delta}_D(\{q_0\}, w) \in F_D \;\leftrightarrow\; \hat{\delta}(q_0, w) \cap F \neq \phi$

4.4 NFA ϵ-Closures

Definition of $Eclose(q)$

- $q \in Eclose(q)$
- If $p \in Eclose(q)$, then each state in $\delta(p, \epsilon) \in Eclose(q)$
- Iterate above step, until no more changes to $Eclose(q)$

Extended Transition Function for ϵ-NFAs

Basis: $\hat{\delta}(q, \epsilon) = Eclose(q)$

Induction: $\hat{\delta}(q, wa) = \bigcup_{p \in R} Eclose(p)$, where $R = \bigcup_{p \in \hat{\delta}(q,w)} \delta(p, a)$

- i.e. $\hat{\delta}(q, wa) = \bigcup_{p \in \hat{\delta}(q,w)} \bigcup_{r \in \delta(p,a)} Eclose(r)$

Similarly, $L(A) = \{w \mid \hat{\delta}(q_0, w) \cap F \neq \phi\}$

Proof of Equivalence of ϵ-NFA and DFA

Similar to proof of equivalence of NFA and DFA (see above).

5 Regular Expressions

Basis: ϵ and ϕ are regular expressions, and $L(\epsilon) = \{\epsilon\}$ and $L(\phi) = \phi$

- If $a \in \Sigma$, then a is a regular expression, and $L(a) = \{a\}$

Induction: if r_1 and r_2 are regular expressions, then so are:

- $r_1 + r_2$ — $L(r_1 + r_2) = L(r_1) \cup L(r_2)$
- $r_1 \cdot r_2$ — $L(r_1 \cdot r_2) = \{xy \mid x \in L(r_1) \text{ and } y \in L(r_2)\}$
- r_1^* — $L(r_1^*) = \{x_1 x_2 \ldots x_k \mid x_i \in L(r_1)\}$ for any natural number k
- (r_1) — $L((r_1)) = L(r_1)$ (we use parentheses for disambiguation)

5.1 Precendence of Operators

$* > . > +$

5.2 DFA to Regular Expressions

All languages accepted by DFAs are accepted by regular expressions, and vice versa (hence these languages are called regular languages)

- How do we show this?

First, let's show that we can convert DFAs to regular expressions.

Let DFA $A = (Q, \Sigma, \delta, q_{start}, F)$

- Let $Q = \{1, 2, \ldots\}$ and $q_{start} = 1$
- Define $R_{i,j}^k$ be the regular expression for set of strings that can be formed going from state i to j, using intermediate states numbered $\leq k$

Base case: definition of $R_{i,j}^0$ (i.e. no intermediate state)

- If $i \neq j$: $R_{i,j}^0 = a_1 + a_2 + \ldots + a_m$, where a_r are symbols such that $\delta(i, a_r) = j$
- If $i = j$: $R_{i,j}^0 = \epsilon + a_1 + a_2 + \ldots + a_m$, where a_r are symbols such that $\delta(i, a_r) = i$

Induction case: $R_{i,j}^{k+1} = R_{i.j}^k + R_{i,k+1}^k (R_{k+1,k+1}^k)^* R_{k+1,j}^k$

- Idea: can use state $(k+1)$ 0 times $\rightarrow$ have $R_{i,j}^k$
- Idea: can use state $(k+1)$ $\geq$1 times $\rightarrow$ go from i to $k+1$, then go from $k+1$ to $k+1$ any number of times, then go from $k+1$ to j

Language accepted by DFA, $L(A) = \sum_{j \in F} R_{1,j}^n$

5.3 Regular Expressions to ϵ-NFA (Thompson's Construction)

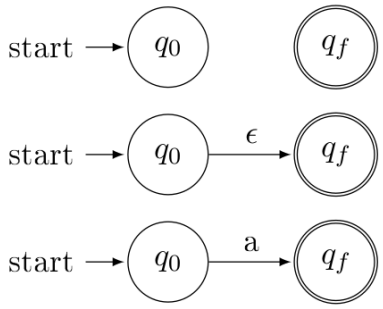
($\star$) Note: the proof is simple, but tricky: be careful, it's easy to mess up the proof because things can interfere with one another.

Some properties of our resulting ϵ-NFA:

- Only 1 starting state q_0, and only 1 final state qf; these states are different
- No transition into the starting state; no transition out of the final state

Base Cases

- ϕ: $A = (\{q_0, q_f\}, \Sigma, \delta, q_0, \{q_f\})$, where δ is an empty function (i.e. no transitions)
- ϵ: $A = (\{q_0, q_f\}, \Sigma, \delta, q_0, \{q_f\})$, where $\delta(q_0, \epsilon) = q_f$
- a: $A = (\{q_0, q_f\}, \Sigma, \delta, q_0, \{q_f\})$, where $\delta(q_0, a) = q_f$



Inductive Cases

For all these examples, let:

- Let $A_1 = (Q_1, \Sigma, \delta_1, q_0^1, F_1)$ be the automata for r_1
- Let $A_2 = (Q_2, \Sigma, \delta_2, q_0^2, F_2)$ be the automata for r_2
- Let δ contain all existing transitions from δ_1 and δ_2; we'll add additional transitions to δ later
- No overlapping states, i.e. $Q_1 \cap Q_2 = \phi$

$r_1 + r_2$ (or) — see (c)

- Let $A = (\{q_0, q_f\} \cup Q_1 \cup Q_2, \Sigma, \delta, q_0, \{q_f\})$
  - $\delta(q_0, \epsilon) = \{q_0^1, q_0^2\}$
  - $\delta(q_f^1, \epsilon) = \{q_f\}$ for $q_f^1 \in F_1$
  - $\delta(q_f^2, \epsilon) = \{q_f\}$ for $q_f^2 \in F_2$

$r_1 \cdot r_2$ (and) — see (b)

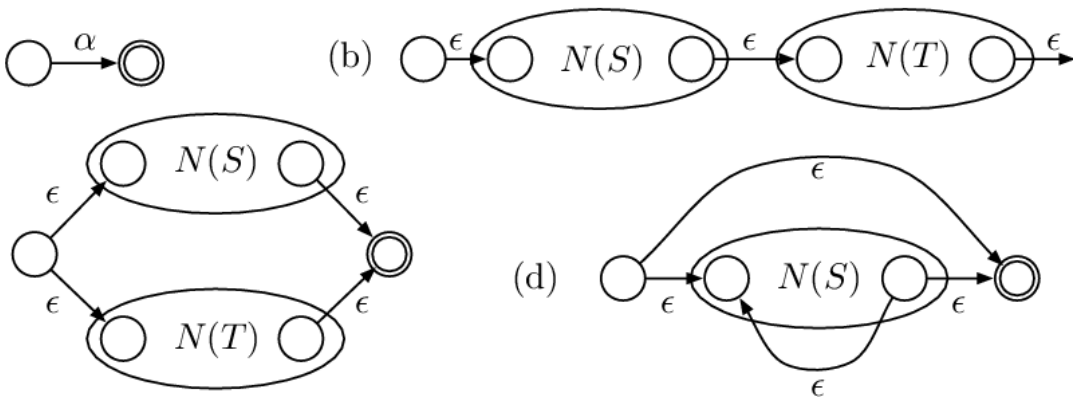
- Let $A = (\{q_0, q_f\} \cup Q_1 \cup Q_2, \Sigma, \delta, q_0, \{q_f\})$
  - $\delta(q_0, \epsilon) = \{q_0^1\}$
  - $\delta(q_f^1, \epsilon) = \{q_0^2\}$
  - $\delta(q_f^2, \epsilon) = \{q_f\}$

r_1^* — see (d)

- Let $A = (\{q_0, q_f\} \cup Q_1, \Sigma, \delta, q_0, \{q_f\})$
  - $\delta(q_0, \epsilon) = \{q_0^1, q_f\}$
  - $\delta(q_f^1, \epsilon) = \{q_0^1, q_f\}$

r_1^+ — see (d) but without start→final arrow

- Let $A = (\{q_0, q_f\} \cup Q_1, \Sigma, \delta, q_0, \{q_f\})$
  - $\delta(q_f^1, \epsilon) = \{q_0^1, q_f\}$



5.4 Properties of Regular Expressions

- $M + N = N + M$
- $L(M + N) = LM + LN$
- $L + L = L$
- $(L^*)^* = L^*$
- $\phi^* = \epsilon$ — NOTE! (from taking it 0 times)
- $\epsilon^* = \epsilon$
- $L^+ = LL^* = L^*L$
- $L^* = \epsilon + L^+$
- $(L + M)^* = (L^*M^*)^*$ — tricky

6 Equivalence Classes

Reduce DFAs to the *smallest possible DFA* that accepts the same language.

Consider any regular language L.

- $u \equiv_L w$ if $ux \in L \leftrightarrow wx \in L$ for all x
- $\equiv_L$ is an equivalence relation, as it is reflexive, symmetric, and transitive
- Let $equiv(w)$ denote the equivalence class of w

Form a DFA $(Q, \Sigma, \delta, q_0, F)$ as follows:

- $Q = \{equiv(w) \mid w \in \Sigma^*\}$
- $q_0 = equiv(\epsilon)$
- $F = \{equiv(w) \mid w \in L\}$
- $\delta(equiv(w), a) = equiv(wa)$

Proof that there cannot be a smaller DFA (by contradiction)

- Suppose $A' = (Q', \Sigma', \delta', q_0', F')$ is an automata that accepts L
- Suppose otherwise, that $u \not\equiv_L w$ but $\hat{\delta}'(q_0', u) = \hat{\delta}'(q_0', w)$
- Then there exists a string x such that $ux \in L$, but $wx \notin L$ (by definition of $\equiv_L$)
- But $\hat{\delta}(q_0', ux) = \hat{\delta}(q_0', wx)$, so A' accepts both ux and wx or accepts neither. Contradiction!
- Let $u \equiv_{A'} w$ iff $\hat{\delta}(q_0', u) = \hat{\delta}(q_0', w)$
- Then $\equiv_{A'}$ divides the equivalence classes $\equiv_L$ into finer equivalence classes
- Therefore, the DFA given using $\equiv_L$ is minimal, and same as any other minimal automata (unique)

6.1 Minimization of Automata, Equivalence

Distinguishable: (p, q) are distinguishable if there exists a string w such that either:

- $\hat{\delta}(p, w) \in F$ and $\hat{\delta}(q, w) \notin F$; or
- $\hat{\delta}(p, w) \notin F$ and $\hat{\delta}(q, w) \in F$

Indistinguishable: (p, q) are indistinguishable if there exists a string w such that:

- $\hat{\delta}(p, w) \in F \leftrightarrow \hat{\delta}(q, w) \in F$

Table Building Algorithm

Idea: iteratively find all *distinguishable pairs* $\rightarrow$ the remaining pairs are *indistinguishable*, and can be merged

- First, remove all non-reachable states
- Base case: (p, q) such that $p \in F$ and $q \notin F$ (or vice versa) is distinguishable
- Inductive step: For all $a \in \Sigma$, if $\delta(p, a)$ and $\delta(q, a)$ are distinguishable, then (p, q) are distinguishable
- Continue the inductive step until no more distinguishable pairs can be added

Example

Round 1

- Start with base case: accepting state $q_4 \in F$, but $q_0, q_1, q_2, q_3, q_5 \notin F$, so $(q_0/q_1/q_2/q_3/q_5, q_4)$ are distinguishable

Round 2

- Iterate through all remaining 'empty' cells in the table, and consider all transitions.
- $\delta(q_5, b) = q_4$, but $\delta(q_0/q_1/q_2/q_3, b) = q_0/q_1/q_2/q_3/q_5$ which is distinguishable from q_4, so $(q_0/q_1/q_2/q_3, q_5)$ are distinguishable

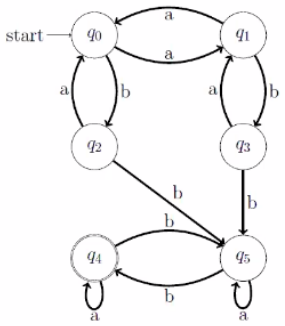
Round 3

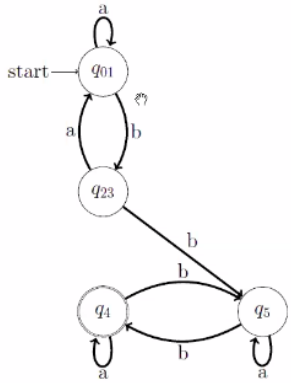
- Iterate through all remaining 'empty' cells
- $\delta(q_0/q_1, b) = q_2/q_3$, but $\delta(q_2/q_3, b) = q_5$ which is distinguishable from q_2/q_3, so $(q_0/q_1, q_2/q_3)$ are distinguishable

Round 4

- No more cells are distinguishable. Terminate.

q_1					
q_2	X3	X3			
q_3	X3	X3			
q_4	X1	X1	X1	X1	
q_5	X2	X2	X2	X2	X1
	q_0	q_1	q_2	q_3	q_4





Proof of Table Building Algorithm

The algorithm will terminate in finitely many steps; this is obvious, because there are only finitely many pairs to consider.

Why should the algorithm find *all* and *only* pairs of distinguishable states?

Finds *only* pairs of distinguishable states

- By induction on number of steps, if algorithm says that p, q are distinguishable, then they are distinguishable
- Base case: ϵ distinguishes accepting and non-accepting states
- Induction step:
  - Suppose the algorithm finds the pair (p, q), since $(\delta(p, a) = p', \delta(q, a) = q')$ are distinguishable
  - Since (p', q') are distinguishable, then for some x, $\hat{\delta}(p', x) \in F$ and $\hat{\delta}(q', x) \notin F$ (or vice versa)
  - Then $\hat{\delta}(p, ax) \in F$ and $\hat{\delta}(q, ax) \notin F$ (or vice versa)
  - Hence (p, q) are distinguishable

Finds *all* pairs of distinguishable states

- By induction on length of strings that distinguish the states
- Base case: ϵ finds all pairs of states that can be distinguished using strings of length 0
- Induction step:
  - Suppose the algorithm has found all pairs of states that can be distinguished using strings of length at most k
  - Consider any pair of states (p, q) that can be distinguished using string $w = ax$ of length $k + 1$
  - Then the algorithm will find the pair $(\delta(p, a), \delta(q, a))$ as distinguishable (by induction)
  - Hence the algorithm will find the pair (p, q) to be distinguishable

7 Properties of Regular Languages

How to prove that a language is not regular?

- Number of equivalence classes are infinite (but this is hard to show!)
- *Use pumping lemma*: show that a contradiction arises, if it's instead regular and satisfies the pumping lemma
- (Note that if a language is finite, it is already regular. Only infinite languages can be irregular.)

7.1 Pumping Lemma

Pumping Lemma: Let L be a regular language. Then there exists a constant n (depends on L) such that for every long enough string $w \in L$ where $|w| \geq n$, we can break $w = xyz$, such that:

- $y \neq \epsilon$
- $|xy| \leq n$
- For all $k \geq 0$, $xy^k z \in L$

Note:

- If L is regular, it satisfies the pumping lemma
- The converse is not true; i.e. NOT TRUE that if L satisfies the pumping lemma, then it is regular

To prove that a language is *not* regular:

- Proof by contradiction. Suppose that L is regular.
- Then we can find some string $w = xyz \in L$ such that $xy^k z \notin L$ for some k. Contradiction.

Proof of Pumping Lemma

Suppose DFA $A = (Q, \Sigma, \delta, q_0, F)$ accepts L.

- Let n be the number of states in Q
- Suppose $w = a_1 a_2 \dots a_m$, where $m \geq n$
- For $i \geq 1$, let $q_i = \hat{\delta}(q_0, a_1 \dots a_m)$
- Then by pigeonhole principle, there exists $i, j \leq n$ where $i < j$ such that $q_i = q_j$
- Let $x = a_1 \dots a_i$, $y = a_{i+1} \dots a_j$, $z = a_{j+1} \dots a_m$
- Since $\hat{\delta}(q_i, y) = q_i$, then for all k, $\hat{\delta}(q_i, y^k) = q_i$
- Therefore, $\hat{\delta}(q_0, xyz) = \hat{\delta}(q_0, xy^k z)$ for all k

(Idea: since there are finite number of states n for an infinite language, it must 'loop back' to a previous state at some point for some substring y)

Examples of Pumping Lemma

Let $L = \{a^m b^m \mid m \geq 1\}$

- Proof that L is not regular (by contradiction):
- Suppose not, that L is regular; then it satisfies the pumping lemma
- Let n be as in the pumping lemma
- Let $w = a^n b^n = xyz$ as in the pumping lemma

- Here, y consists only of all a's (because $|xy| \leq n$)
- Then $xy^2z \in L$, but now xy^2z contains more a's than b's. Contradiction!

Let $L = \{a^i b^j \mid i < j\}$

- Proof that L is not regular (by contradiction):
- Suppose not, that L is regular; then it satisfies the pumping lemma
- Let n be as in the pumping lemma
- Let $w = a^n b^{n+1} = xyz$ as in the pumping lemma
- Here, y consists only of all a's (because $|xy| \leq n$)
- Then $xy^3z \in L$, but now xy^3z contains more a's than b's. Contradiction!

Let $L = \{a^p \mid p \text{ is prime}\}$

- Proof that L is not regular (by contradiction):
- Suppose not, that L is regular; then it satisfies the pumping lemma
- Let n be as in the pumping lemma
- Let $w = a^p = xyz$, where p is a large enough prime such that $p > n$
- Then $xy^kz \in L$ for all k
- Choose $k = p + 1$
- Then $|xy^kz| = |xy^{p+1}z| = |xyz| + |y^p| = p + |y| \cdot p = p(1 + |y|)$, which is not prime. Contradiction!

7.2 Closure Properties

Suppose that L, L_1, and L_2 are regular. The following are also regular:

- $L_1 \cup L_2$
- $L_1 \cdot L_2$
- $\bar{L} = \Sigma^* - L$
- $L_1 \cap L_2$
- $L_1 - L_2$
- L^R
- $h(L)$, where h is a homomorphism

7.3 Homomorphisms

Homomorphism: $a \in \Sigma \to h(a) \in \Sigma^*$, i.e. replace a character with some string

- $h(\epsilon) = \epsilon$
- $h(a_1 a_2 \ldots) = h(a_1)h(a_2)\ldots$
- $h(L) = \{h(x) \mid x \in L\}$

If L is regular, then $h(L)$ is also regular.

Let $R(M)$ be the regular expression for $h(L(M))$, where M is a regular expression for $L(M)$. Properties of homomorphisms:

- $R(\phi) = \phi$

- $R(\epsilon) = \epsilon$
- $R(a) = h(a)$ for $a \in \Sigma$
- $R(M + N) = R(M) + R(N)$
- $R(M \cdot N) = R(M) \cdot R(N)$
- $R(M^*) = (R(M))^*$

Proof that $L(R(M + N)) = h(L(M + N))$ (others are similar):

- LHS: $L(R(M + N)) = L(R(M) + R(N)) = L(R(M)) \cup L(R(N)) = h(R(M)) \cup h(R(N))$ (by induction)
- RHS: $h(L(M + N)) = h(L(M)) \cup h(L(N))$
- Therefore, $L(R(M + N)) = h(L(M + N))$

[More details: see p157 of textbook]

7.4 Decision Problems on Regular Languages

Given a regular language, these decision problems can be solved:

- $L = \phi$?
  - If no final state is reachable by the language's DFA, then $L = \phi$
- $L = \Sigma^*$?
  - Take the complement, then it reduces to the decision problem for $\bar{L} = \Sigma^*$
- $L(A) = L(A')$?
  - Minimize both DFAs for A and A', see if they are the same (by renaming states)
  - Alternatively, build the DFA for $L(A) - L(A')$, then see if they are the same
- $w \in L$?
  - Just run the string w on the DFA for L

8 Context-Free Languages and Grammars

$G = (V, T, P, S)$

- V: finite set of *variables* i.e. non-terminals
- T: finite set of *terminals* ($V \cap T = \phi$)
- P: finite set of *productions*
  - Form is $A \to \gamma$, where $\gamma \in (V \cup T)^*$
- S: start symbol, $S \in V$

Note: every regular language is a context-free language.

8.1 Examples

Palindromes

- $S \to \epsilon \mid a \mid b$
- $S \to aSa \mid bSb$

Infix Arithmetic

- $E \to id$
- $E \to E + E$
- $E \to E * E$

If/Else Statements

- $S \to id = E$
- $S \to$ If E Then $id = E$ Else $id = E$ EndIf
- $S \to S; S$

8.2 Derivations

Let G be a grammar. Let $\Rightarrow_G$ refer to a (single-step) derivation: $\alpha A \beta \Rightarrow_G \alpha\gamma\beta$ if there is a production of the form $A \to \gamma$.

Define $\alpha \Rightarrow_G^* \beta$ (a multi-step derivation):

- Base case: $\alpha \Rightarrow_G^* \alpha$
- Induction: If $\alpha \Rightarrow_G^* \beta$ and $\beta \Rightarrow_G \gamma$, then $\alpha \Rightarrow_G^* \gamma$

Language: $L(G) = \{w \in T^* \mid S \Rightarrow_G^* w\}$

- Sentential form: α is a sentential form if $S \Rightarrow_G^* \alpha$, where α can contain both terminals and non-terminals (i.e. reachable from start symbol)
- Left-most derivation: in each derivation step, ALWAYS replace the *leftmost non-terminal* in the sentential form
- Right-most derivation: in each derivation step, ALWAYS replace the *rightmost non-terminal* in the sentential form

8.3 Right-Linear Grammars

Right-linear grammars: all productions are of the form

- $A \to wB$
- $A \to w$

Theorem: Regular Language → Right-Linear Grammar

($\star$) Theorem: Every regular language can be generated by a right-linear grammar.

Suppose L is accepted by DFA $A = (Q, \Sigma, \delta, q_0, F)$.

Then we can create a right-linear grammar $G = (Q, \Sigma, P, q_0)$, where:

- For each $\delta(q, a) = p$, we create a production $q \to ap$ in P
- For each $q \in F$, we create a production $q \to \epsilon$
- By induction, we can show that $\hat{\delta}(q_0, w) = p$ iff $q_0 \Rightarrow_G^* wp$
- Therefore, $\hat{\delta}(q_0, w) \in F$ iff $q_0 \Rightarrow_G^* w$

Proof by induction that $\hat{\delta}(q_0, w) = p \ \leftrightarrow \ q_0 \Rightarrow_G^* wp$:

- Base case: $\hat{\delta}(q_0, \epsilon) = p \leftrightarrow p = q_0$, and $q_0 \Rightarrow_G^* p \leftrightarrow q_0 = p$
- Induction step:

$$
\begin{aligned}
& \hat{\delta}(q_0, wa) = p' \\
\leftrightarrow\ & \exists p\ \hat{\delta}(q_0, w) = p \ \wedge \ \delta(p, a) = p' \\
\leftrightarrow\ & \exists p\ q_0 \Rightarrow_G^* wp \ \wedge \ p \to ap' \\
\leftrightarrow\ & q_0 \Rightarrow_G^* wap'
\end{aligned}
$$

Theorem: Right-Linear Grammar → Regular Language

($\star$) Theorem: Every language generated by a right-linear grammar is regular.

Suppose we have a right-linear grammar $G = (V, \Sigma, P, S)$.

- Assume that each production is of the form $A \to bB$ or $A \to \epsilon$ (why can we assume this?)

Define a NFA $A = (V, \Sigma, \delta, S, F)$ as follows:

- For each $A \to aB$, then $B \in \delta(A, a)$
- $F = \{A \mid A \to \epsilon \text{ is a production in } P\}$
- By induction, we can show that $A \Rightarrow_G^* wB$ iff $B \in \hat{\delta}(A, w)$
- Therefore, $S \Rightarrow_G^* w$ iff $\hat{\delta}(S, w) \cap F \neq \phi$

Why can we assume that each production is of the form $A \to bB$ or $A \to \epsilon$?

- For a production $A \to b_1 b_2 \ldots b_n B$, convert it to $A \to b_1 B_1$, $B_1 \to b_2 B_2$, $\ldots$ $B_{n-1} \to b_n B$
- For a production $A \to b_1 b_2 \ldots b_n$, do the same as above (but ending in $B_{n-1} \to b_n B_n$, $B_n \to \epsilon$)

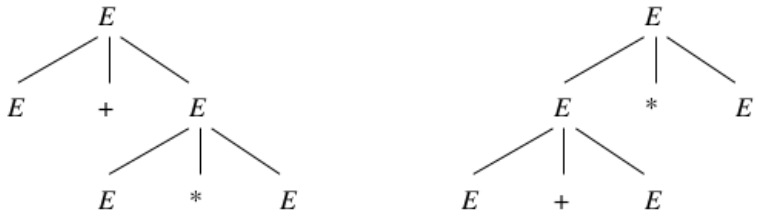
8.4 Ambiguous Grammars

Consider this:

- $E \to E + E$
- $E \to E * E$
- $E \to id$

Derivation of $id + id * id$ can be done in 2 ways:

- Apply production $E \to E + E$ first, then $E \to E * E$;
- Apply production $E \to E * E$ first, then $E \to E + E$.



Resolving Ambiguous Grammars

This one enforces + before $*$, and enforces associativity on the left.

- $S \to S + T$
- $S \to T$
- $T \to T * id$
- $S \to id$

Inherently Ambiguous Grammars

$L = \{a^n b^n c^m d^m \mid n, m \geq 1\} \cup \{a^n b^m c^m d^n \mid n, m \geq 1\}$

8.5 Removing Useless Symbols

Useless symbols: symbols that don't appear in *any* derivation from the start symbol, i.e. does not appear in $S \Rightarrow_G^* w$ for any $w \in T^*$

- Symbol is useful only if it is *generating* and *reachable*
- Generating symbol: A is *generating* if $A \Rightarrow_G^* w$ for some $w \in T^*$ (i.e. generates an ACTUAL terminal-only string, including ϵ)
- Reachable symbol: A is *reachable* if $S \Rightarrow_G^* \alpha A \beta$ for some $\alpha, \beta \in T^*$

Algorithm for Removing Useless Symbols

1. Find all generating symbols; get rid of all productions involving non-generating symbols.
2. THEN Find all reachable symbols; get rid of all productions involving non-reachable symbols.

Finding Generating Symbols

All terminals are generating (including ϵ).

If there is a production $A \to \alpha$ where α consists of *only* generating symbols, then A is generating. Iterate process until no more symbols can be added.

Finding Reachable Symbols

S is reachable.

If A is reachable, and $A \to \alpha$ is a production, then every symbol in α is reachable.

Example of Removing Useless Symbols

$$S \to Aa \text{ (generating)}$$
$$S \to AC$$
$$A \to a \text{ (generating)}$$
$$C \to EC$$
$$E \to b \text{ (generating)}$$

C is non-generating, so remove it. Then E is non-reachable, so remove it.

8.6 Converting to Chomsky Normal Form (CNF)

Chomsky Normal Form: all productions of form $A \to BC$ or $A \to a$

Converting to CNF

1. Eliminate ϵ productions
2. Eliminate unit productions
3. Convert productions to productions of length ≤ 2: length 2 (with non-terminals on RHS), or length 1 (with terminal on RHS)

1) Eliminating ϵ productions

1. Find all *nullable* non-terminals A such that $A \Rightarrow_G^* \epsilon$;
2. Then get rid of ϵ productions;
3. Then for the remaining productions $B \to \alpha$, replace it all 2^n possible productions $B \to \alpha'$, where α' can be formed from α by possibly deleting some of the non-terminals that are nullable.

($\star$) Note that if S is nullable, then this method only generates the language $L - \{\epsilon\}$.

Step 1. Finding nullable symbols

- Base case: If $A \to \epsilon$, then A is nullable
- Inductive step: If $A \to \alpha$ and every symbol in α is nullable, then A is nullable
- Repeat inductive step until no more nullable symbols can be found

Step 3. Example of replacing with 2^n combinations: $A \to ABaCdC$, where A and C are nullable

- We have 8 possible combinations

- $A \to ABaCdC$
- $A \to BaCdC$
- $A \to ABadC$
- $A \to BadC$
- ...

Theorem: will generate $L(G') = L(G) - \{\epsilon\}$

- We will prove a more general statement: for all $A \in V$, for all $w \in T^* - \{\epsilon\}$, $A \Rightarrow_G^* w$ iff $A \Rightarrow_{G'}^* w$
- Claim: if $A \Rightarrow_G^* w$, then $A \Rightarrow_{G'}^* w$
  - Proof: in the derivation $A \Rightarrow_G^* w$, we can try "dropping" each symbol which eventually produces ϵ in the derivation
- Claim: if $A \Rightarrow_{G'}^* w$, then $A \Rightarrow_G^* w$
  - Proof: consider first step in derivation, $A \Rightarrow_{G'} \alpha \Rightarrow_{G'}^* w$
  - Suppose the corresponding production in G was $A \to \alpha'$
  - Then we have $\alpha' \Rightarrow_G^* \alpha$, by having the "nulled" symbols generate ϵ

How to get back ϵ in the language?

- Simple: create new start symbol $S' \to S \mid \epsilon$

2) Eliminating unit productions

Unit production: $A \to B$ is a unit production.

Unit pair: (A, B) is a unit pair if $A \Rightarrow_G^* B$

- Base case: (A, A) is a unit pair
- Inductive step: If (A, B) is a unit pair and $B \to C$, then (A, C) is a unit pair (this requires removing ϵ productions first)

(Only concerned with non-trivial unit pairs)

To eliminate unit productions, we add $A \to \gamma$ for all non-unit productions of form $B \to \gamma$.

Example

- $S \to A \mid AB$
- $A \to B \mid ab \mid b$
- $B \to b \mid c$
- Then the unit pairs are: (S, A), (A, B), and also (S, B) (alongside the trivial unit pairs (S, S), (A, A), (B, B))
- For unit pair (S, A), add $S \to ab \mid b$
- For unit pair (A, B), add $A \to b$ (already inside) $\mid c$
- For unit pair (S, B), add $S \to b \mid c$

3) Convert productions to max length 2

Say we have a production $A \to X_1 X_2 \dots X_k$. Change it to the following:

- $A \to Z_1 B_2,\ B_2 \to Z_2 B_3,\ \dots,\ B_{k-1} \to Z_{k-1} Z_k$

- $Z_i \to X_i$ if X_i is terminal, $Z_i = X_i$ if X_i is non-terminal

8.7 Size of Parse Tree: 2^{s-1}

Suppose we have a parse tree derived from a CNF grammar. If the length of longest path from root to node is s, then size of string w generated is at most 2^{s-1}.

- (Because for a full binary tree with only single $A \to a$ productions at the lowest level, the number of terminal nodes is 2^{s-1})

8.8 Pumping Lemma for CFL

($\star$) Let L be a context-free language. Then there exists a constant n such that, for any $z \in L$ where $|z| \geq n$, we can write $z = uvwxy$ such that:

- $|vxw| \leq n$
- $vx \neq \epsilon$ (either can be empty, but not both)
- $uv^iwx^iy \in L$ for all $i \geq 0$ (both are pumped equal number of times)
- (Idea: now we pump at 2 places, not just 1!)

Example: equal numbers of a, b, c

Let $L = \{a^m b^m c^m \mid m \geq 1\}$. L is not a CFL. *Proof by contradiction*: suppose otherwise that L is a CFL.

- Let $n > 1$ be as in the pumping lemma.
- Consider $z = a^n b^n c^n = uvwxy$ as in the pumping lemma.
- We know that $|vwx| \leq n$, so vwx cannot contain both a and c.
- Case 1: vwx does not contain a
  - uv^2wx^2y is not in L, because it contains only n a's, even though its length $> 3n$.
- Case 2: vwx does not contain c
  - uv^2wx^2y is not in L, because it contains only n c's, even though its length $> 3n$.

Example: repeated string

Let $L = \{\alpha\alpha \mid \alpha \in \{a, b\}^*\}$. L is not a CFL. *Proof by contradiction*: suppose otherwise that L is a CFL.

- Let $n > 1$ be as in the pumping lemma.
- Consider $z = a^{n+1}b^{n+1}a^{n+1}b^{n+1} = uvwxy$ as in the pumping lemma.
- Case 1: vwx is in first $a^{n+1}b^{n+1}$
  - Then $uwy = a^{n+1-k}b^{n+1-s}a^{n+1}b^{n+1}$ where $wx = a^k b^s$ and $0 < k + s \leq n$
  - (Idea: uwy cannot be written as $\alpha\alpha$!)
  - Suppose that uwy can be written as $\alpha\alpha$
  - Then second α must end in b^{n+1}, and so first α must end somewhere in the first b^{n+1-s}
  - Then second α must end in $a^{n+1}b^{n+1}$
  - But that means $|\alpha| \geq 2n + 2$, so $k + s \leq 0$, contradiction
- Case 2: vwx is in middle $b^{n+1}a^{n+1}$
  - Then $uwy = a^{n+1}b^{n+1-k}a^{n+1-s}b^{n+1}$ where $wx = b^k a^s$ and $0 < k + s \leq n$ ...

- Case 3: vwx is in last $a^{n+1}b^{n+1}$ ...

Proof of Pumping Lemma for CFL

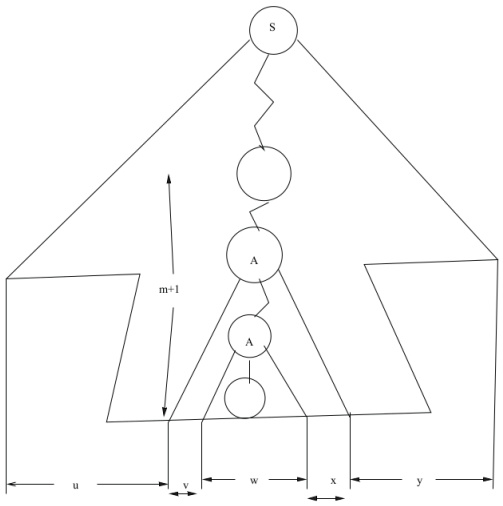
Let L be a CFL. Assume without loss of generality that $L \neq \emptyset$ and $L \neq \{\epsilon\}$. Choose a CNF grammar $G = (V, T, P, S)$ for $L - \{\epsilon\}$.

Let $m = |V|$, let $n = 2^m$. Suppose a string $z \in L$ of length at least $n = 2^m$.

Consider the parse tree for z.

Consider the longest path, which must have a length of at least $m+1$. Then among the last $m+1$ nonterminals, there must be 2 repeated nonterminals (by pigeonhole principle).

- vwx has length at most n
- $vx \neq \epsilon$, because of CNF properties
- v and w can be 'pumped'
  - $z = uvwxy$, where $S \Rightarrow^* uAy \Rightarrow^* uvAxy \Rightarrow^* uvwxy$
  - Then $A \Rightarrow^* vAx$ and $A \Rightarrow^* w$
  - Then $A \Rightarrow^* v^iAx^i \Rightarrow^* v^iwx^i$
  - Hence $S \Rightarrow^* uv^iwx^iy$



Idea: there will be a repetition in a long-enough path (A then A in the above figure)

8.9 Closure

Closure under Substitution

Consider mapping each terminal a to a CFL L_a, where $s(a) = L_a$.

- $s(a) = L_a$
- $s(w)$: $s(\epsilon) = \{\epsilon\}$ and $s(wa) = s(w) \cdot s(a)$, i.e. $s(a_1a_2 \ldots a_n) = s(a_1) \cdot s(a_2) \cdot \ldots \cdot s(a_n)$
- Theorem: If L is CFL and s is substitution such that $s(a) = L_a$ is a CFL, then $\bigcup_{w \in L} s(w)$ is a CFL

Let $G = (V, T, P, S)$ be grammar for L. Let $G_a = (V_a, T_a, P_a, S_a)$ be grammar for L_a for each a.

Let $G' = (V', T', P', S')$ be grammar for $\bigcup_{w \in L} s(w)$:

- $V' = V \bigcup \bigcup_{a \in T} V_a$

- $T' = \bigcup_{a \in T} T_a$
- $P' = P_{new} \bigcup_{a \in T} P_a$, where P_{new} is formed using productions in P, where in each of the productions, terminal a is replaced by S_a

$w = a_1 a_2 \dots a_n$, then $\alpha = S_{a_1} S_{a_2} \dots S_{a_n}$

Closure under Reversal: if L is CFL, then L^R is CFL

- Create $G^R = (V, T, P^R, S)$ where P^R consists of productions obtained by reversing productions in P

Closure under Intersection: if L is CFL and R is regular, then $L \cap R$ is CFL

- Idea: run PDA for L and DFA for R in parallel; we can form new PDA for $L \cap R$
- Let PDA $P = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$ for L, and DFA $A = (Q', \Sigma, \delta', q_0', F')$ for R
- Form PDA $P'' = (Q'', \Sigma, \Gamma, \delta'', q_0'', Z_0, F'')$ as follows:
  - $Q'' = Q \times Q'$
  - $q_0'' = (q_0, q_0')$
  - $F'' = F \times F'$ (both reach final state)
  - $\delta''((p, q), \epsilon, Z) = \delta(p, \epsilon, Z) \times \{q\}$ for all p, q, Z
  - $\delta''((p, q), a, Z) = \delta(p, a, Z) \times \{\delta'(q, a)\}$

  We can't exactly run two PDAs in parallel; the two stacks don't work out

  NOTE: CFLs are NOT closed under intersection in general! Example:

- $L_1 = \{a^n b^n c^m \mid n, m \geq 1\}$, CFL
- $L_2 = \{a^m b^n c^n \mid n, m \geq 1\}$, CFL
- $L_3 = L_1 \cap L_2 = \{a^n b^n c^n \mid n \geq 1\}$ is NOT CFL!

  Example: $L = \{w \mid w \in \{a, b, c\}^* \text{ and } \#_a(w) = \#_b(w) = \#_c(w)\}$ is NOT CFL
- If it's a CFL, then $L \cap a^* b^* c^* = \{a^n b^n c^n \mid n \geq 0\}$ is CFL, contradiction

Closure under Union: if L_1 and L_2 are CFLs, then $L_1 \cup L_2$ is CFL

Just merge the grammars; more details in tutorial 7 question 3b).

8.10 Testing

Testing if CFL is $\emptyset$

Check if S is useless symbol. If it's useless, then the language is $\emptyset$; otherwise it is not-empty.

Testing if a string $w = a_1 \dots a_n$ is part of a CFL

Run CYK parsing algorithm (dynamic programming algorithm) on the language's CNF, in $O(n^3)$:

- Let $X_{i,j}$ be the set of nonterminals that generate the string $a_i a_{i+1} \dots a_j$. Then see if $S \in X_{1,n}$.
- Base case: $X_{i,i}$ is set of non-terminals that generate a_i
- Inductive step: $X_{i,j}$ contains all A such that $A \to BC$, where $B \in X_{i,k}$ and $C \in X_{k+1,j}$ for some $i \leq k < j$ — i.e. B generates $a_i a_{i+1} \dots a_k$ and C generates $a_{k+1} \dots a_j$

9 Pushdown Automata

Pushdown automata: we introduce a *stack*. Non-deterministic by default, which matches context-free languages; deterministic is weaker. Can accept by final state or empty stack.

$P = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$

- Γ: stack alphabet
- Z_0: initial symbol on the stack
- δ takes an input state q, input letter a or ϵ, stack symbol X (on top of stack)
  - Pop the top symbol from X
  - Push as many new symbols onto X' as you like, Γ^*
- $\delta(q, a, X) = (q', X')$ if deterministic
- $\delta(q, a, X) = \{(q', X')\}$ if non-deterministic

9.1 Examples

Balanced Parentheses

$L = \{a^n b^n \mid n \geq 0\}$

Idea: keep the excess a's in the stack.

PDA: $(\{q_0, q_1, q_2\}, \{a, b\}, \{a, Z_0\}, \delta, q_0, Z_0, \{q_2\})$

- $\delta(q_0, a, Z_0) = (q_0, aZ_0)$ (last character Z_0 goes first)
- $\delta(q_0, a, a) = (q_0, aa)$
- $\delta(q_0, \epsilon, a) = (q_1, a)$
- $\delta(q_0, \epsilon, Z_0) = (q_2, \epsilon)$
- $\delta(q_1, b, a) = (q_1, \epsilon)$
- $\delta(q_1, \epsilon, Z_0) = (q_2, \epsilon)$

Equal Count of Each

$L = \{w \mid \#_a(w) = \#_b(w)\}$

Idea: keep the excess a's or b's in the stack.

PDA: $(\{q_0, q_1\}, \{a, b\}, \{a, b, Z_0\}, \delta, q_0, Z_0, \{q_1\})$

- $\delta(q_0, a, Z_0) = (q_0, aZ_0)$
- $\delta(q_0, b, Z_0) = (q_0, bZ_0)$
- $\delta(q_0, a, b) = (q_0, \epsilon)$
- $\delta(q_0, a, a) = (q_0, aa)$
- $\delta(q_0, b, a) = (q_0, \epsilon)$
- $\delta(q_0, b, b) = (q_0, bb)$
- $\delta(q_0, \epsilon, Z_0) = (q_1, \epsilon)$

9.2 Instantaneous Descriptions

Instantaneous descriptions are like a 'snapshot' of the computational process of a PDA. (q, w, α):

- q: current state
- w: input left to read
- α: on the stack (first symbol of α is top of stack)

Defining $\vdash$: $(q, aw, \alpha) \vdash_P (p, w, \beta\alpha)$ if $(p, \beta) \in \delta(q, a/\epsilon, X)$ (where X is first symbol of α i.e. top of stack??)

Defining $\vdash^*$:

- $I \vdash_P^* I$
- $I \vdash_P^* J \ \wedge \ J \vdash K \ \rightarrow \ I \vdash^* K$

9.3 Language accepted by PDA

Acceptance by final state: $\{w \mid (q_0, w, Z_0) \vdash_P^* (q_f, \epsilon, \alpha)$ for some $q_f \in F\}$

Acceptance by empty stack: $\{w \mid (q_0, w, Z_0) \vdash_P^* (q, \epsilon, \epsilon)$ for some $q \in Q\}$

Either one is fine.

9.4 Equivalence of Acceptance by Final State and Empty Stack

Acceptance by Empty Stack → Acceptance by Final State

Idea: Initially, put a special symbol X_0 onto the stack. If ever the top of the stack is that symbol, go to the new final state p_f.

(By empty stack) $P = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$

(By final state) $P_F = (Q \cup \{p_0, p_f\}, \Sigma, \Gamma \cup \{X_0\}, \delta_F, p_0, X_0, \{p_f\})$

- $\delta_F(p_0, \epsilon, X_0) = \{(q_0, Z_0X_0)\}$ — first step add Z_0 on top of X_0, then continue as normal
- δ_F is otherwise equivalent: for all $Z \in \Gamma$ and $a \in \Sigma \cup \{\epsilon\}$, $\delta_F(p, a, Z)$ contains all (q, γ) in $\delta(p, a, Z)$
- $\delta_F(p, \epsilon, X_0)$ contains (p_f, ϵ) for all $p \in Q$

Acceptance by Final State → Acceptance by Empty Stack

Idea: From all final states, place a transition to a new final state p_f that pops from the stack repeatedly till empty.

But the original PDA with final state could the empty stack accidentally; so add a special symbol X_0 that it can't 'throw away' otherwise unless it reaches p_f

(By final state) $P = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$

(By empty stack) $P_E = (Q \cup \{p_0, p_f\}, \Sigma, \Gamma \cup \{X_0\}, \delta_E, p_0, X_0, \{p_f\})$

- $\delta_E(p_0, \epsilon, X_0) = \{(q_0, Z_0X_0)\}$ — first step add Z_0 on top of X_0, then continue as normal
- δ_E is otherwise equivalent: for all $Z \in \Gamma$ and $a \in \Sigma \cup \{\epsilon\}$, $\delta_F(p, a, Z)$ contains all (q, γ) in $\delta(p, a, Z)$
- $\delta_E(p, \epsilon, Z)$ contains (p_f, ϵ) for all $p \in F$ and $Z \in \Gamma \cup \{X_0\}$
- $\delta_E(p_f, \epsilon, Z)$ contains (p_f, ϵ) for all $Z \in \Gamma \cup \{X_0\}$

9.5 Equivalence of CFGs and PDAs

CFG $\rightarrow$ PDA that accepts CFG language

Idea: mimic the productions in a CFG using pushdown automata (empty stack model).

- Use the stack to keep track of 'what is left to derive', beginning the stack with start symbol S
- For each non-terminal a on the stack, consume/match it as it is
- For each terminal A on the stack, try all productions $A \rightarrow \gamma$ (non-deterministically) and push γ onto the stack

Let $G = (V, T, P, S)$.

Construct PDA $P = (\{q_0\}, \Sigma, \Gamma, \delta, q_0, S, F)$:

- $\Sigma = T$
- $\Gamma = V \cup T$
- (Match terminals) $\delta(q_0, a, a) = \{(q_0, \epsilon)\}$ for all $a \in T$
- (Match non-terminals) $\delta(q_0, \epsilon, A) = \{(q_0, \gamma) \mid A \rightarrow \gamma \text{ in } P\}$ for all $A \in V$

PDA $\rightarrow$ CFG that accepts PDA language

Idea: mimic the transitions $[qZp]$ in a PDA using productions.

- $[qZp]$ now represents a single non-terminal in our new CFG: from state q in the PDA, go to state p, depending on the current top symbol of the stack Z
- If $Y_1 \dots Y_k$ are pushed onto the stack, then we need to 'use' each of these stack symbols in a way that respects transitions from $r_1, r_2, \dots, r_k$ in order

Let PDA $P = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$.

Define grammar $G = (V, \Sigma, R, S)$:

- $V = \{S\} \cup \{[qZp] \mid q, p \in Q,\ Z \in \Gamma\}$
- $S \rightarrow [q_0 Z_0 p]$ for each $p \in Q$
- $[qXr_k] \rightarrow a[rY_1r_1][r_1Y_2r_2] \dots [r_{k-1}Y_kr_k]$ if $\delta(q, a, X)$ contains $(r, Y_1 \dots Y_k)$, for all $r_1, r_2, \dots, r_k \in Q$

9.6 Deterministic PDA

1. For all $a \in \Sigma \cup \{\epsilon\}$, $Z \in \Gamma$ and $q \in Q$, there is at most one element in $\delta(q, a, Z)$
2. If $\delta(q, \epsilon, X)$ is non-empty, then $\delta(q, a, X)$ is empty for all $a \in \Sigma$

Deterministic PDA is weaker than Non-Deterministic PDA: some language is accepted by NPDA but not DPDA.

- DPDA acceptance by final state: every regular language can be accepted by a DPDA
- DPDA acceptance by empty stack: $\{a, aa\}$ is not accepted by a DPDA

10 Turing Machines

- Infinite tape divided into *cells*, each cell holds any of finite number of symbols
- *Tape head* positioned at one of the cells; initially at leftmost cell of input
- *Finite control* can be in any of finite number of states
- In each step, head can (1) change state, (2) read/write, and (3) move one step left/right

Turing machine $M = (Q, \Sigma, \Gamma, \delta, q_0, B, F)$

- Q: set of states (of finite control)
- Σ: alphabet, $\Sigma \subseteq \Gamma$
- Γ: tape alphabet. E.g. Blank is a tape symbol $B \in \Gamma$, but not an input symbol $B \notin \Sigma$
- δ: transition function, $(Q \times \Gamma) \to (Q \times \Gamma \times \{L, R\})$
- q_0: starting state
- B: blank symbol, assume $B \in \Gamma - \Sigma$. Blanks on both ends
- F: set of final states, $F \subseteq Q$

10.1 Example: 0^n1^n

$L = \{0^n1^n \mid n \geq 1\}$

E.g. 0011, so tape has $\ldots B0011B \ldots$

State	0	1	X	Y	B
q0	q1, X, R	-	-	q3, Y, R	
q1	q1, 0, R	q2, Y, L	-	q1, Y, R	
q2	q2, 0, L	-	q0, X, R	q2, Y, L	
q3	-	-	-	q3, Y, R	q4, B, R
q4	-	-	-	-	-

- q0: Move right to find a 0, replace it with X, then transition to q1
  - but if no 0 (it's already Y), then transition to q3
- q1: Move right to find a 1, replace it with Y, then transition to q2
- q2: Move left to find an X, then transition to q0
- q3: Move right along the Ys until B, then transition to q4
- q4: Accepting state

10.2 Example: Matching a's and b's

E.g. *aababa*, so tape has $\ldots BaababaB \ldots$

State	a	b	B	X
q0	q1, X, R	q2, X, R	qA, B, R	q0, X, R
q1	q1, a, R	q3, X, L		q1, X, R
q2	q3, X, L	q2, b, R		q2, X, R
q3	q3, a, L	q3, b, L	q0, B, R	q3, X, L
qA				

- q1: Already matched a, search for matching b then mark X => q3
- q2: Already matched b, search for matching a then mark X => q3

- q3: Move back left to the beginning
- qA: Accepting state

10.3 Instantaneous Description

The tape is infinite, so leave out blanks on both ends (unless the head is among the blanks).

- q is the current state/head position, it is right before the state you're reading
- Example: $x_0x_1 \ldots x_{n-1}qx_nx_{n+1} \ldots x_m$
- $ID_1 \vdash ID_2 \vdash \ldots \vdash ID_n$, so $ID_1 \vdash^* ID_n$

10.4 Language Accepted by Turing Machine

TM accepts x if $q_0x \vdash^* \alpha q_f \beta$, where $q_f \in F$

10.5 Function Computed by Turing Machine

1. Require that machine halts on input x iff $f(x)$ is defined
2. Interpret contents of machine after it halts as the output of f

10.6 Languages/Functions

A language L is:

- Recursively enumerable (RE): Turing Machine accepts L
- Recursive/decidable: Turing Machine accepts L, and halts on all possible inputs

A function f is:

- Partial recursive: Turing Machine computes f, and it halts on all inputs on which f is defined, but not necessarily for inputs on which f is not defined
- Recursive/computable: Turing Machine computes f, and f is defined on all elements of Σ^*

10.7 Turing Machine and Halting Problem

Machine may never halt; in general, we cannot determine if a machine will halt on a particular input

10.8 Turing Machine Modifications

Tricks to make Turing Machine construction easier:

- Stay where you are, 'S' move
- Storage in Finite Control
- Multiple Tracks
- Subroutines

Semi-infinite tapes: left-end is fixed, right-end is infinite. This one-way infinite tape is equivalent to a two-way infinite tape

- Consider $1, 2, 3, 4, 5, 6, \ldots$ as $1, -1, 2, -2, 3, -3, \ldots$
- Or consider two different tracks: $END, 1, 2, 3, \ldots$ and $0, -1, -2, -3, \ldots$

Initialisation: $\delta(q_S, (X, B)) = ((q_0, U), (X, *), S)$ — input on top

Simulation:

- $m \in \{L, R\}$. $\bar{m}$ is the opposite of m, i.e. $\bar{m} = L$ if $m = R$; otherwise $\bar{m} = R$

If $\delta(q, X) = (q', Y, m)$:

- $\delta'((q, U), (X, Z)) = ((q', U), (Y, Z), m)$ — up
- $\delta'((q, D), (Z, X)) = ((q', D), (Z, Y), \bar{m})$ — down

If $\delta(q, X) = (q', Y, R)$:

- $\delta'((q, U), (X, *)) = ((q', U), (Y, *), L)$
- $\delta'((q, D), (X, *)) = ((q', U), (Y, *), L)$

If $\delta(q, X) = (q', Y, L)$:

- $\delta'((q, U), (X, *)) = ((q', D), (Y, *), R)$
- $\delta'((q, D), (X, *)) = ((q', D), (Y, *), R)$

10.9 Non-Deterministic Turing Machines

Now $\delta(q, a)$ gives a finite set of possibilities.

Acceptance: accept a string if there exists an accepting state q_f such that $q_0 x \vdash^* \alpha q_f \beta$

10.10 Church-Turing Thesis

Whatever can be computed by an algorithmic device can be computed by a Turing Machine.

11 Undecidability

11.1 Encodings of Strings and Turing Machines

Strings: each string can be encoded in binary as 0/1.

- Every number has a string corresponding to it, and vice versa
- There is no real difference between strings and numbers, just a matter of interpretation

Turing Machines: each Turing Machine can be encoded in binary as 0/1.

- (It's possible that TMs that accept the same language have different codes, that's fine.)
- One-to-one correspondence between TM and natural numbers

Let M_i denote TM with code number i, $W_i = L(M_i)$ denote the language accepted by that TM.

- States: $q_1, q_2, \ldots$ where q_1 is the start state and q_2 is the only accepting state
- Tape symbols: $X_1 \ldots X_s$ — X_1 is 0, X_2 is 1, X_3 is blank
- Directions: L is D_1 and R is D_2
- Transitions: $\delta(q_i, X_j) = (q_k, X_l, D_m)$ can be coded as $0^i\ 1\ 0^j\ 1\ 0^k\ 1\ 0^l\ 1\ 0^m$ (where each of $i, j, k, l, m \geq 1$)
- TM: coded as $C_1\ 11\ C_2\ 11\ C_3 \ldots C_n$, where C_i are its transitions

For example: $M = (\{q_1, q_2, q_3\}, \{0, 1\}, \{0, 1, B\}, \delta, q_1, B, \{q_2\})$

- $\delta(q_1, 1) = (q_3, 0, R)$: 0100100010100
- $\delta(q_3, 0) = (q_1, 1, R)$: 0001010100100
- $\delta(q_3, 1) = (q_2, 0, R)$: 00010010010100
- $\delta(q_3, B) = (q_3, 1, L)$: 00010001000010010

11.2 Non-RE Languages

Let $L_d = \{w_i \mid w_i \notin L(M_i)\}$. L_d is the *diagonalisation language*, and L_d is not RE.

- i.e. L_d consists of all strings w, such that the TM whose code is w does not accept w as input.

Suppose any M_j is given. We will show that $L(M_j) \neq L_d$.

- Case 1: $w_j \in L(M_j)$. Then $w_j \notin L_d$ by definition of L_d, so $L(M_j) \neq L_d$
- Case 2: $w_j \notin L(M_j)$. Then $w_j \in L_d$ by definition of L_d, so $L(M_j) \neq L_d$
- In both cases, $L(M_j) \neq L_d$. Since this applies for any M_j, we have that L_d is not accepted by any TM.
- Hence L_d is not RE.

11.3 Recursive Languages

Theorem: If L is recursive, then $\bar{L}$ is recursive.

Suppose $M = (Q, \Sigma, \Gamma, \delta, q_0, B, F)$ accepts L and halts on all inputs. Modify M to form a new machine M':

- Assume WLOG that there's only one accepting state q_{acc} in M, and no transition from q_{acc} to another state
- Let a new state q_{new} be the only accepting state of M' (and q_{acc} is a non-accepting state in M').
- For any accepting state q and letter a of the alphabet, if $\delta(q, a)$ is not defined in M, then let $\delta(q, a) = q_{new}$ in M'.

Other transitions in M' are as in M

Theorem: L is recursive $\leftrightarrow$ L is RE and $\bar{L}$ is RE.

- ($\rightarrow$) If L is recursive then $\bar{L}$ is also recursive, so both L and $\bar{L}$ are RE
- ($\leftarrow$) Suppose M accepts L, and M' accepts $\bar{L}$. Create a new TM M'' for L as follows:
  - $M''(x)$ copies x into two different tapes. Then it runs $M(x)$ on the first tape and $M'(x)$ on the second tape in parallel
  - Then if at any point in time $M(x)$ halts and accepts, then $M''(x)$ halts and accepts; if $M'(x)$ halts and accepts, then $M''(x)$ halts and rejects
  - M'' is guaranteed to halt because either M or M'' will halt and accept on x

11.4 Universal Turing Machine

$L_u = \{(M, w) \mid M \text{ accepts } w\}$

We can build the universal TM that accepts L_u. It has the following tapes:

- M (coded)
- w (coded in $\{0,1\}^*$)

Let's have three tapes:

- Tape 1: contains M w. M is left as it is, w is copied over to tape 2 for initialisation
- Tape 2: contains simulated TM's current state tape, e.g. $BB|0|00|000 * 0^j|0|0|0000|00000|0BB$ — each segment between | stands for a cell, each | stands for 11, each 0 stands for 0^* separated by a 1
- Tape 3: contains simulated TM's state, e.g. 00000 — for state 5
- Tape 4: scratch tape for local computations

E.g. $110^i10^j10^k10^l10^m11$

- Making a transition and changing the tape: How to say replace 0^j with 0^l without ruining everything else?
- Copy into scratch tape, but replacing 0^j with 0^l: e.g. $\ldots|000 * 0^l|0|\ldots$
- If the new state is accepting, halt and accept; otherwise, carry on

L_u is RE but not recursive.

$\bar{L}_u$ is not RE.

- Suppose by way of contradiction that M accepts $\bar{L}_u$. Then construct machine M' to accept L_d as follows:
- For $M'(w_i)$, extract code i from w_i
- Run M on (M_i, w_i), where M_i is coded appropriately
- Accept iff M accepts the above execution
- Since L_d is not RE, so M' cannot exist, so $\bar{L}_u$ is not RE.
- $w_i \in L_d \leftrightarrow f(w_i) \in \bar{L}_u$

$\bar{L}_d$ is RE.

- $\bar{L}_d = \{w_i \mid w_i \in L(M_i)\}$. So $w_i \in \bar{L}_d$ iff $M((M_i, w_i))$ is accepted

12 Undecidable Problems: Reductions

($\star$) P_1 reduces to P_2, $P_1 \leq_m P_2$ if some recursive function f behaves as follows: $x \in P_1$ iff $f(x) \in P_2$

- i.e. transform instance in problem 1 to instance in problem 2; then solve transformed instance in problem 2 to see if instance is solvable in problem 1

Statements about reductions:

- If P_2 is recursive, then P_1 is recursive
- If P_2 is RE, then P_1 is RE
- If P_1 is undecidable, then P_2 is undecidable (contrapositive of above)
- If P_1 is non-RE, then P_2 is non-RE (contrapositive of above)
- (Idea: If P_2 is easy, then P_1 is easy. If P_1 is hard, then P_2 is hard)

So to show that P_2 is undecidable ("hard"), reduce a known undecidable problem P_1 to it.

12.1 Example: TMs Accepting Empty Set/Language

Let $L_e = \{M \mid L(M) = \emptyset\}$, let $L_{ne} = \{M \mid L(M) \neq \emptyset\}$

- Theorem: L_{ne} is RE.
- Theorem: L_e is not recursive.
- Corollary: L_e is not RE.

Theorem: L_{ne} is RE

We can construct TM M' to accept L_{ne}. M' takes in input M (in coded form) as follows:

- For $t = 0$ to ∞:
- For $i = 0$ to t:
  - If $M(w_i)$ accepts within t steps, then accept

Theorem: L_e is not RE

Idea: reduce $\bar{L}_u$ to L_e.

- $(M, w) \in \bar{L}_u \leftrightarrow M' \in L_e$, i.e.
- $M(w)$ does not accept $\leftrightarrow L(M') = \emptyset$

Given $M\#w$, construct M' as follows:

- $M'(x)$
- For $t = 0$ to ∞:
  - If $M(w)$ accepts within t steps, then accept

Note that the reduction function $f : M\#w \rightarrow M'$ is recursive

Here, $M(w)$ does not accept, i.e. $M\#w \in \bar{L}_u$, iff $L(M') = \emptyset$; and $M(w)$ accepts, i.e. $M\#w \notin \bar{L}_u$, iff $L(M') = \Sigma^* \neq \emptyset$

Variations

Same proof technique can be used to show not RE for several variations:

- $L = \{M \mid M \text{ does not accept } a\}$
- $L = \{M \mid M \text{ does not accept } a \text{ or does not accept } b\}$
- $L = \{M \mid L(M) \text{ is finite}\}$

Example of not RE: $L_5 = \{M \mid L(M)$ has more than 5 elements$\}$

Idea: Reduce L_e to L_5.

- f maps M to M'
- $M \in L_e \leftrightarrow M' \in L_5$
- $L(M) = \emptyset \leftrightarrow |L(M')| \leq 5$

$f : M \rightarrow M'$

- $M'(x)$
- For $t = 0$ to ∞
- For $i = 0$ to t:
  - If $M(w_i)$ accepts within t steps, then accept

If $L(M) = \emptyset$, then $L(M') = \emptyset$, and thus $M' \in L_5$. If $L(M) \neq \emptyset$, then $L(M') = \Sigma^*$, and thus $M' \notin L_5$. Thus $L_e \leq_m L_5$.

12.2 Rice's Theorem

Suppose P is a property on RE languages.

- Example of a property: whether the language is context-free/regular/finite/empty
- A *property* is simply a *set* of RE languages; e.g. the property of being empty is the set $\{\emptyset\}$, the empty property is $\emptyset$

Is $L_P = \{M \mid L(M)$ satisfies property $P\}$ decidable or RE?

A property P about RE languages is *non-trivial* if there exists at least one RE language which *satisfies* the property, and there exists at least one RE language which *does NOT satisfy* the property.

Rice's Theorem: suppose P is a *non-trivial* property about RE languages. Then L_P is undecidable.

Proof of Rice's Theorem

Suppose L is a RE language that does not satisfy P. Let M'' be the machine that accepts L. Define f as follows, $f(M) = M'$ such that M' is defined as follows:

- $M'(x)$:
- For $t = 0$ to ∞ do:
- For $i = 0$ to t do:
  - If $M(w_i)$ accepts within t steps and $M''(x)$ accepts within t steps, then accept x

If $L(M) = \emptyset$, then $L(M') = \emptyset$. If $L(M) \neq \emptyset$, then $L(M') = L$. Thus f reduces L_e to L_p.

Since L_e is not recursive, L_P is not recursive.

(??? See textbook page 398-399)

12.3 Post's Correspondence Problem (PCP)

Input: two lists of strings $A = w_1, \ldots, w_k$ and $B = x_1, \ldots, w_k$

Question: does there exist $i_1, \ldots, i_m$ such that $w_{i_1} w_{i_2} \ldots w_{i_m} = x_{i_1} x_{i_2} \ldots x_{i_m}$?

- e.g. $w_2 w_1 w_1 w_3 = x_2 x_1 x_1 x_3$

Modified PCP (MPCP)

Fix the first pair.

Question: does there exist $i_1, \ldots, i_m$ such that $w_1 w_{i_1} \ldots w_{i_m} = x_1 x_{i_1} \ldots x_{i_m}$?

Reducing MPCP to PCP

- From an MPCP instance, construct a PCP instance.
- Suppose that we have a solution to MPCP instance, $i_1, \ldots, i_m$. Then $*y_1 y_{i_1} \ldots y_{i_m} = z_1 z_{i_1} \ldots z_{i_m} *$, and $0, i_1, \ldots, i_m, k+1$ must be solution to PCP
- Suppose that we have a solution to constructed PCP instance, which must be $0, i_1, \ldots, i_m, k+1$. Then $i_1, \ldots, i_m$ must be solution to MPCP

Reducing $\mathrm{L_u}$ to MPCP

- Given a pair (M, w), construct instance (A, B) of MPCP such that M accepts w iff (A, B) has a solution

12.4 Other Undecidable Problems

Idea: reduce PCP to these problems.

Undecidability of Ambiguity of CFGs

Reduce PCP to the problem of whether a given CFG is ambiguous.

Take a PCP instance with lists $A = w_1, w_2, \ldots, w_k$ and $B = x_1, x_2, \ldots, x_k$.

- CFG for list A: $A \to w_1 A a_1 \mid \ldots \mid w_k A a_k \mid w_1 a_1 \mid \ldots \mid w_k a_k$ — grammar G_A and language L_A
- CFG for list B: $B \to x_1 B a_1 \mid \ldots \mid x_k B a_k \mid x_1 a_1 \mid \ldots \mid x_k a_k$ — grammar G_B and language L_B
- where a_i is an *index symbol*: represents choice of w_i in list A, and choice of x_i in list B

There is a solution to the PCP instance $\leftrightarrow$ There is ambiguity in the grammar of $G_{AB} = G_A \cup G_B$.

- Terminal strings from A are of form $w_{i_1} \ldots w_{i_m} a_{i_m} \ldots a_{i_1}$
- Terminal strings from B are of form $x_{i_1} \ldots x_{i_m} a_{i_m} \ldots a_{i_1}$
- $\mathrm{\$G_{AB}}$ has ambiguity if and only if we see an overlap in strings generated by G_A and G_B, i.e. solves the PCP instance

Complement of List Language

We can show that $\bar{L}_A$ and $\bar{L}_B$ are CFGs.

Further Undecidable Problems of CFGs

Let G_1 and G_2 be CFGs.

1. $L(G_1) \cap L(G_2) = \emptyset$
    - Reduce PCP to this problem

- Let $G_1 = G_A$ and $G_2 = G_B$
- Then $L(G_1) \cap L(G_2) = \emptyset$ iff PCP has no solution, i.e. complement of PCP has a solution

2. $L(G_1) = L(G_2)$
3. $L(G) = T^*$ for some alphabet T
4. $L(G_1) \subseteq L(G_2)$
   - Reduce PCP to this problem
   - Let G_1 be CFG for $(\Sigma \cup I)^*$ and G_2 be CFG for $\bar{L}_A \cup \bar{L}_B$
   - Then $L(G_1) \subseteq L(G_2)$ iff $\bar{L}_A \cup \bar{L}_B = (\Sigma \cup I)^*$ or $L_A \cap L_B = \emptyset$, i.e. PCP instance has no solution
5. $L(R) \subseteq L(G)$ for some regular expression R

12.5 Unrestricted Grammars

$G = (N, \Sigma, S, P)$

- N: non-terminals
- Σ: terminals
- S: start symbol
- P: productions of form $\alpha \to \beta$, where $\alpha \in (N \cup \Sigma)^+$ and $\beta \in (N \cup \Sigma)^*$

Theorem

- If G is UG, then $L(G)$ is RE.
- If L is RE, then there exists UG G for L.

Example: $\{a^n b^n c^n \mid n \geq 1\}$

- $S \to aSBC \mid aBC$
- $CB \to BC$
- $aB \to cc$
- $bB \to cc$
- $bC \to cc$
- $cC \to cc$

13 Complexity

Model: multitape Turing Machines

13.1 Time Complexity

$Time_M(x)$: Time used by M on input x before halting

- $Time_M(x) = \infty$ if it doesn't halt
- Non-deterministic TM: $Time_M(x)$ is the *maximum* time on any path, even non-accepting ones

Time bounded: M is $T(n)$ time bounded, if for any input x of length n, $Time_M(x) \leq T(n)$

13.2 Space Complexity

$Space_M(x)$: Maximum number of cells used by M on input x (excluding input)

- $Space_M(x) = \infty$ if it doesn't halt

Space bounded: M is $S(n)$ space bounded, if for any input x of length n, $Space_M(x) \leq S(n)$

13.3 Complexity Classes

Set of languages that are accepted by a $S(n)$ bounded TM

- $DSPACE(S(n)) = \{L \mid \text{some } S(n) \text{ space bounded deterministic machine accepts } L\}$
- $DTIME(T(n)) = \{L \mid \text{some } T(n) \text{ time bounded deterministic machine accepts } L\}$
- $NSPACE(S(n)) = \{L \mid \text{some } S(n) \text{ space bounded non-deterministic machine accepts } L\}$
- $NTIME(T(n)) = \{L \mid \text{some } T(n) \text{ time bounded non-deterministic machine accepts } L\}$

13.4 Dealing with Constants

Tape Compression

Idea: constants don't matter in how much *space* you're going to use.

Theorem: Fix $c > 0$. If L is accepted by M that is $S(n)$ space bounded, then it is also accepted by M' that is $\lceil cS(n) \rceil$ space bounded.

- We can construct M' that uses less space by *combining m cells*, by using *more alphabets*;

finite control keeps track of which of m cells is being represented

Linear Speedup

Idea: constants don't matter in how much *time* you're going to use.

Theorem: Fix $c > 0$. If L is accepted by M that is $T(n)$ time bounded, then it is also accepted by M' that is $\lceil cT(n) \rceil$ time bounded.

- One *basic step* of M' simulates several steps of M

13.5 Blum Complexity Measure?

Blum complexity measure Φ: $\Phi(x, y)$ or $\Phi_x(y)$, Turing Machine of code x on input y

- If it halts, it should be defined how much resource is used
- If the machine does not halt, then complexity should be ∞

13.6 Space/Time Constructible Functions

$S(n)$ is *fully space constructible* if there's a TM that on all inputs of length n, uses space $S(n)$

$T(n)$ is *fully time constructible* if there's a TM that on all inputs of length n, halts and uses time $T(n)$

13.7 Relationship between Complexity Classes

$DTIME(S(n)) \subseteq DSPACE(S(n))$

- i.e. $\forall L \in DTIME(S(n)), L \in DSPACE(S(n))$, i.e. all languages accepted by TMs time-bounded by $S(n)$ are also space-bounded by $S(n)$
- Reason: any computation halting in $S(n)$ time cannot use more than $S(n)$ space

If $L \in DSPACE(S(n))$ where $S(n) \geq \log n$, then $L \in DTIME(c^{S(n)})$ (for some c depending on L)

- ???

If $L \in DTIME(T(n))$, then $L \in DTIME(c^{T(n)})$ (for some c depending on L)

- ???

13.8 Hierarchy Theorem

Space Hierarchy

Theorem: Suppose L is accepted by $S(n) \geq \log n$ space-bounded machine. Then L can be accepted by a $S(n)$ space-bounded machine which *halts on all inputs* (either accepts/rejects).

- Proof omitted.

Theorem: Suppose $S_2(n)$ and $S_1(n)$ are both $\geq \log n$; $S_2(n)$ is fully space constructible; $lim_{n\to\infty} \frac{S_1(n)}{S_2(n)} = 0$. Then there exists some language in $DSPACE(S_2(n))$ but not $DSPACE(S_1(n))$.

Time Hierarchy

Theorem: Suppose $T_2(n)$ and $T_1(n)$ are both $\geq (1+\epsilon)n$; $T_2(n)$ is fully space constructible; $lim_{n\to\infty} \frac{T_1(n)\cdot\log(T_1(n))}{T_2(n)} = 0$. Then there exists some language in $DTIME(T_2(n))$ but not $DTIME(T_1(n))$.

13.9 Efficient Computations

$\boldsymbol{P} = \{L \mid$ some polynomial-time bounded deterministic machine accepts $L\}$

$\boldsymbol{NP} = \{L \mid$ some polynomial-time bounded non-deterministic machine accepts $L\}$

$\boldsymbol{coNP} = \{L \mid \bar{L} \in \boldsymbol{NP}\}$

13.10 NP

Suppose $L \in \boldsymbol{NP}$. Then there exists a deterministic polynomial time computable predicate $P(x, y)$, such that $x \in L$ iff $\exists y |y| \leq q(|x|)\ P(x, y)$

i.e. proofs are verifiable in polynomial time by a deterministic TM, whereby *proof* that $x \in L$ is a y such that $P(x, y)$ is true

Reducibility

Polynomial-time reducibility

$L_1 \leq_m^p L_2$: L_1 is poly time, many-to-one, reducible to L_2

- There exists a polynomial time computable function f such that $x \in L_1 \Leftrightarrow f(x) \in L_2$

$L_1 \leq_T^p L_2$: L_1 is poly time, Turing, reducible to L_2

- There exists a polynomial time TM M such that M^{L_2} accepts L_1

13.11 NP-Completeness

L is NP-complete iff:

- L is in NP
- L is NP-hard, i.e. $\forall L' \in NP$, $L' \leq_p^m L$

How to show NP-complete?

- Show L is in NP
- Show another known NP-complete problem $L' \leq_p^m L$ (polynomial-time reduction)

Examples of NP-Complete Problems

- Satisfiability, 3-SAT: $(A \vee B \vee \neg C) \wedge (E \vee F \vee \neg A)$ — is there a satisfying truth assignment?
- (Min-)Vertex-Cover: is there a subset of vertices of size $\leq k$, such that for each edge (u, v), either u or v belongs to this subset?
- (Max-)Clique: is there a clique of size k or more?
- (Max-)Independent-Set: is there an independent set of size k or more? (i.e. subset such that for all distinct vertices u and v, $(u, v) \notin E$)
- Hamiltonian circuit
- Partition
- Set cover
- TSP

Example: Proof that Vertex Cover is NP-Complete

Vertex Cover is in NP

- Given a graph $G = (V, E)$
- Guess a subset V' as a candidate vertex cover:
  - Verify that $|V'| \leq k$
  - Verify that for all $(u, v) \in E$, at least one of u or v is in V'
- This can be done in polynomial time

Vertex Cover is NP-hard

- Reduction from 3-SAT