1 Regular Languages, DFAs and NFAs

1. Σ^*: Set of all words constructed by Σ.

2. Ex: $\Sigma = \{a, b\}$, $\Sigma^* = \{a, b, ab, aa, ba, bb, aaa, \dots\}$

3. Regular expressions are used to represent RLs. For the above language, we can define the language as $L = (a^*b^*)^*$ as it's all possible combinations of a and b.

   (i) $\emptyset$ is a reg. ex.
   (ii) ϵ is a reg. ex.
   (iii) If $a \in \Sigma$, a is a reg. ex.
   (iv) If R_1, R_2 are reg. ex., so is $R_1 + R_2$
   (v) If R is reg. ex., so is R^*
   (vi) Regular expressions define exactly the same languages as DFAs.

4. An NFA with ϵ moves is equivalent to an NFA which is equivalent to a DFA in strength. A determinization algorithm can be run on the NFA to convert it into a DFA, similar to how an algorithm can be run to convert an NFA + ϵ into an NFA.

5. Equality of regular expressions is decidable, but only by going through DFAs and minimization. You cannot assume that regular expressions can be tested for equality directly.

6. **Kleene's Theorem**: A language is regular if and only if it is defined by a regular expression. This is proven by the fact that NFAs can recognize regular expressions by induction and the definition of a regular language is a language recognized by a finite automaton.

7. **The Myhill-Nerode Theorem**: The minimal form is unique for DFAs. The minimization algorithm finds this form. There is no analogous statement for NFAs (thus determinization must be run first).

8. Closure Properties of Regular Languages (L_1, L_2 regular)

   (i) $L_1 \cup L_2$, $L_1 \cap L_2$ regular
   (ii) $L_1 \cdot L_2$ regular (Concatenation)
   (iii) $\overline{L}, L^*$ regular
   (iv) $L_1 \subseteq L_2 \iff L_1 \cap \overline{L_2} = \emptyset$

9. Pigeonhole Principal: If you have N boxes and M objects, M > N, when you put the objects into the boxes at least one box will have >1 object.

10. **Pumping Lemma for Regular Languages**

    If L is regular, $(\exists p \in N)$, $p > 0$ s.t. $\forall w \in L$, $|w| \geq p$ $\exists x, y, z \in \Sigma^*$ s.t.:
    (i) $w = xyz$
    (ii) $|xy| \leq p$
    (iii) $|y| > 0$
    (iv) $\forall n \geq 0$ $xy^n z \in L$

11. Algorithms
    1. Regex to NFA + ϵ
    2. NFA + ϵ to NFA
    3. Determinization: NFA to DFA
    4. Minimization: DFA to minimized DFA
    5. Reachability: Graph algorithms can be used on DFAs to find reachable states

2 Context-free Languages and PDAs

1. A Language can have both an ambiguous and unambiguous CFG

   (i) A context-free grammar **(CFG)** consists of a set of symbols called terminal, a disjoint set of symbols V called variables or non-terminals, a set of rules called productions.

   (ii) Example of a CFG for $L = \{a^n b^n | n \geq 0\}$:
   $S \to aSb|\epsilon$

(iii) Example $L = \{a^m b^n c^k d^j | m, n, k, j \geq 0 \text{ and } m = k \text{ and } k = j\}$
$S \to XY|\epsilon$
$X \to aXb|\epsilon$
$Y \to cYd|\epsilon$

(iv) Context-free languages are recognized by pushdown automata (PDA). Pushdown automata cannot always be made deterministic.

2. **Every Regular Language is a CFL**
Proof: Let A be a regular language. Then there exists a DFA N = (Q, Σ, δ, q_0, F) such that L(N) = A. Build a context-free grammar G = (V, Σ, R, S) as follows. Set V = $\{R_i | q_i \in Q\}$ (that is, G has a variable for every state of N). Now, for every transition $\delta(qi, a) = q_j$ add a rule $R_i \to$ aRj. For every accepting state $q_i \in F$ add a rule $R_i \to \epsilon$. Finally, make the start variable $S = R_0$.

3. Closure Properties and Facts for CFL's (L_1, L_2 regular)

(i) $L_1 \cup L_2$ CFL
(ii) $L_1 \cdot L_2$ CFL
(iii) L^* CFL
(iv) The complement of a CFL may not be a CFL.
(v) The intersection of two CFLs may not be a CFL.

4. Facts about CFGs, CFLs, and PDAs

- The intersection of a CFL and a Regular Language is a CFL
- For a one letter language, L is regular $\iff$ L is a CFL
- There is an algorithm to put a grammar G into Chomsky normal form G' so that $L(G) = L(G') \cup \{\epsilon\}$.
- The Coecke-Kasami-Younger (CKY) algorithm determines whether or not $w \in L(G)$ $\forall w \in \Sigma^*$
- There is an algorithm to decide if the language of a CFG is empty
- It is undecidable whether $L(G) = \Sigma^*$ for a CFG G
- Intersection of two non-context-free languages could be context-free. Think $\emptyset$.
- It is undecidable whether $L(G1) \cap L(G2) = \emptyset$, where G1, G2 are context-free languages.
- For infinite context-free languages, if every word in it cannot be pumped using the pumping lemma for regular languages, then that language does not contain an infinite regular subset.

5. **Proof for CFLs**

This is a two-part proof. First, show that every string generated by the grammar is the given language. Second, show that only strings in the language can be generated by our grammar, through proof by induction.

6. **Pumping Lemma for Context-free Languages**

For all CFLs L, $\exists p > 0$, $\forall s \in L$ $|s| \geq p$, $\exists u, v, w, x, y \in \Sigma^*$ s.t.:

(i) $s = uvwxy$
(ii) $|vx| > 0$
(iii) $|vwx| \leq p$
(iv) $\forall i \geq 0$, $uv^i wx^i y \in L$

3 Computability Theory

1. **Turing Machines**

Turing machines are equivalent to while programs, to RAM machines, to machines with two stacks, to machines with two counters, to multi-tape Turing machines, to nondeterministic Turing machines, to multidimensional Turing machines, to Post production systems, to λ-calculus and to any of the common programming languages.

A TM consists of a finite set of states Q, a finite set of input symbols Σ, a finite set of tape symbols Γ, the transition function δ:
Q x $\Gamma \to Q$ x Γ x $\{R, L\}$

Start state s, accept state a, reject state $r \in Q$

(i) $\delta(q, a) = (q', b, L)$ means if the machine reads a and is in state q, it changes state to q', erases a and writes b, and moves to the left.

(ii) Once you go left and reach the left-end symbol, you can no longer continue left. The tape is infinite to the right, but stops there on the left.

(iii) Once the machine enters a or r, it never leaves.

(iv) A configuration is a description of the machine at an instant of time. For example, uaq_ibv yields uq_iacv if $\delta(q_i, b) = (q_i, c, L)$ The head is written to the left of the character being read.

(v) Given M and input w, the start configuration is q_0w or sw.

(vi) An accept configuration is any configuration where the state is a or q_a. Similarly, any state r or q_r is a reject configuration. An accept or reject configuration is called a halting configuration.

(vii) M accepts w if there is a finite sequence of configurations $c_1, c_2, \ldots, c_k$ such that:

(a) c_1 is the start configuration sw

(b) Each c_i yields c_{i+1}

(c) c_k is an accepting configuration

(viii) $L(M) = \{w \in \Sigma^* | M \text{ accepts } w \}$. 3 outcomes are possible: accept, reject, or loop forever.

(a) L is Turing recognizable if $\exists$ TM M s.t. $L = L(M)$

(b) L is Turing decidable if $\exists$ TM M s.t. $L = L(M)$ and $\forall w \in \Sigma^*$ M halts on w

(ix) We say L is computably enumerable or **CE** if $L = L(M)$ for some TM

(x) We say L is computable or decidable if $L = L(M)$ and M always halts

2. Computability Facts and Definitions

(i) A set $x \subseteq N$ is computable or decidable if its characteristic function is computable

(ii) An infinite set X of naturals is computable $\iff$ X is in the range of some total non-decreasing computable function

(iii) A computable set is one for which we have a decision procedure, i.e. one which always terminates with a definite answer. For example, the set of prime numbers is computable: we definitely have an algorithm that can take a number as input and decide whether it is prime or not. This algorithm will always terminate.

(iv) If we write $< M, x >$ to be the encoding of a pair of numbers where $< M >$ is the code number of a Turing machine and x is the code of an input string, we know these two sets are not computable:

(a) $A_{TM} \stackrel{\text{def}}{=} \{< M, x > | M \text{ accepts x}\}$

(b) $H_{TM} \stackrel{\text{def}}{=} \{< M, x > | M \text{ halts on x}\}$

(v) Every finite language is regular and thus decidable. To expand on it, every regular language is a CFL and through the CKY algorithm we can determine whether or not $w \in L(G)$ $\forall w \in \Sigma^*$. Thus, we can determine membership with our TM.

3. CE, co-CE

(i) A set $X \subseteq N$ is computably enumerable (CE) if there is an algorithm A that lists all the members of X, and only members of X in some arbitrary order. A would only halt when enumerating a finite set.

(ii) For a set to be CE, one of the following must hold:

(a) X is the domain of a computable function

(b) The semi-characteristic function of X is computable:

$$\begin{cases} 1 & n \in X \\ undefined & n \notin X \end{cases}$$

(c) X is the range of a computable function

(iii) A set X is co-CE if it's compliment is CE.

(iv) The union and intersection of two CE sets is CE

(v) In every infinite CE set, there exists an infinite computable set.

4. Post's Theorem

   (i) If X is computable $\implies$ X is CE

   (ii) If X is CE and $\overline{X}$ CE (i.e X is CE and co-CE), X is computable

   Proof: For the second part, we assume that A enumerates X and that B enumerates $\overline{X}$. In order to show that X is computable we need a decision procedure for X. We run both A and B in parallel and as soon as one of them enumerates n we know whether n is in the set X or not. This has to terminate because n is in one of the sets and the enumerator for that set has to produce the answer in some finite time.

3.1 Reductions

1. "P reduces to Q" means that if you can solve problem Q then you can solve problem P. A reduction is an explicit proof of this implication. The notation is $P \leq Q$. If $P \leq Q$ then Q is more difficult than P, this is what the notation suggests. If Q is more difficult than P then if we show $P \leq Q$ and P is something that we know is undecidable then we know that Q must be undecidable. This is usually how we use reductions

2. Example: We start with $HP \leq Q$ where HP stands for the halting problem and thus show Q is undecidable. When we know that Q is undecidable we can do another reduction Q $\leq$ R and now we know that the problem R is undecidable. Thus the relation $\leq$ is transitive: we can chain reductions together.

3. For reductions, we assume that we are working with some fixed alphabet Σ, considering strings in Σ^* and that all the problems we consider are **language membership** problems. Now we can talk about a language A being reduced to B: this means that if you can solve the problem "is the word w in B?" you can solve the problem "is the word x in A?"

4. A language A is mapping reducible to B, written $A \leq_m B$ if there is a total computable function $f : \Sigma^* \to \Sigma^*$ s.t $(x \in A \iff f(x) \in B)$. This function f is called a reduction.

5. If $A \leq_m B$ and B is decidable, then A is decidable. If A is undecidable then B must be undecidable.

6. If $A \leq_m B$ and B is CE, then so is A. If A is not CE then neither is B. If B is co-CE then so is A.

7. If $A \leq_m B$, the same holds for $\overline{A}$ and $\overline{B}$

8. If $A \leq_m B$ and A is CE but not decidable, then B is not co-CE.

   Proof: If B were co-CE then $\overline{B}$ would be CE and the mapping reduction could be rewritten as $\overline{A} \leq_m \overline{B}$ so we would have that $\overline{A}$ is CE. But if A is CE and $\overline{A}$ is also CE then A is decidable. This contradicts the assumption that A is not decidable

9. Mapping vs. Turing Reductions

   For Turing reductions we ask the oracle a question and do some post-processing on the oracle's answer. In mapping reduction we package a question to the oracle (this is what f does) but we do not get to do anything with the answer other than return it; we cannot even negate it. More generally, with Turing reduction we can ask the oracle many questions but with mapping reduction we get just one question and no post-processing of the answer. Any mapping reduction is a Turing reduction but the converse is not true. Turing reductions are good for proving undecidability results but they are not good for showing the difference between CE and co-CE.

10. Two completely different programs P_1 and P_2 may have the same input-output relation where $P_1 \neq P_2$ but $[[P_1]] = [[P_2]]$. We say $P_1 \sim P_2$ iff $[[P_1]] = [[P_2]]$. This is read as "P_1 and P_2 are extensionally equivalent". For Turing machines, $M_1 \sim M_2$ iff $L(M_1) = L(M_2)$. We say Q:Prog $\implies \{True, False\}$ is a property of programs. We say Q is an extensional property if $M_1 \sim M_2 \implies Q(M_1) = Q(M_2)$. Example: a program having a running time of $O(n^2)$ is not extensional but intensional because you can have various programs with the same input and output running at different complexities (like sorting algorithms). However, a program sorting its input is indeed extensional, as you can write different programs but the output depends wholly on the input. The two trivial properties are False for every input and True for every input.

11. **Rice's Theorem**: Every non-trivial extensional property of a program is undecidable.

12. **VALCOMPS**: $VALCOMPS(M, w)$ is a set of valid computations of the TM M on the word w ending in q_a or q_r. This may or may not be $\emptyset$. If M is non-deterministic, there may be many such strings. We write consecutive configurations separated by a #, for example:
$\#\ldots\#abqbaab\#abaq'aab\#\ldots$ for $\delta(q, b) = \delta(q', a, R)$. The change is always confined to a 3 character window, the head, the cell being read, and the movement that results from this.

    A valid computation for (M, w) is a sequence of configurations $\#\alpha_0\#\alpha_1\#\ldots\alpha_n\#$ such that:

    (a) α_0 is the start configuration

    (b) α_n is a halting configuration

    (c) $\forall n$, α_{n+1} follows from α_n by the rules of M

    - If M does not halt on w, then $VALCOMPS(M, w) = \emptyset$
    - Let $\Delta = \Gamma \cup Q \cup \#$. If $VALCOMPS(M, w) = \emptyset$ then $\overline{VALCOMPS(M, w)} = \Delta^*$
    - Given a TM and a word $< M, w >$ we can effectively construct a PDA (hence also a grammar) which recognizes (generates) the complement of VALCOMPS

    VALCOMPS2(M, w): $\#\alpha_0\#{\alpha_1}^{REV}\#\alpha_2\#{\alpha_3}^{REV}\ldots\#{\alpha_N}^*\#$
    VALCOMPS2(M, w) = $L(G_1) \cap L(G_2)$ and $HP \leq_m \{< G_1, G_2 > |\ L(G_1) \cap L(G_2) \neq \emptyset\}$

13. **Validity**

    - Validity of formulas in first-order logic is **undecidable**, Validity is **CE**
    - A propositional logic formula is valid $\iff$ it is provable
    - $PCP \leq_m VALIDITY$

14. **Post Correspondence Problem**: $A_{TM} \leq_m PCP$. Since A_{TM} is CE, so is PCP.