Week 6 - Sets and Functions

Set Theory: A set is a collection of definite, whole and distinct objects. The objects are called elements or members of a set.

$$A \in B$$

Notation: a is member of set A

Principle of Extentionality: For all sets A and B

$$A = B \Leftrightarrow \ \forall x \ (x \in A \Leftrightarrow x \in B)$$

Meaning: If A and B are the same set, then x occurs in A iff x occurs in B.

Set Abstraction: Used for large sets, including infinite sets.
If P is a property of objects x, then the following abstraction denotes the set of things x that have the property P.

$$\{x \mid P(x)\}$$

Hence a $\in$ { x | P(x) } is equivalent to P(a)

Well-founded: A set is well-founded if there is no infinite sequence within it. Eg. W $\in$ W is an infinite set.

Subset: A is a subset of B, if A is contained in B.

$$A \subseteq B$$

Notation: A is subset of set B

Proper Subset: If A is a subset of B, and A is not equal to B.

$$A \subset B$$

Notation: A is a proper subset of B

Reflexivity: A is a subset of itself.

$$A \subseteq A$$

Antisymmetry: A and B are only subsets of each other if they are the same. Otherwise either A can be a subset of B, or B can be a subset of A (but not both)

$$A \subseteq B \ \wedge \ B \subseteq A \ \Rightarrow \ A = B$$

Transitivity: If A is a subset of B, and B is a subset of C, then A is a subset of C.

$$A \subseteq B \ \wedge \ B \subseteq C \Rightarrow \ A \subseteq C$$

Partial Ordering: From the three laws above, we can conclude that $\subseteq$ is a partial ordering.

Empty Set: A set with no elements, always contained in other sets. Denoted $\emptyset$

Singleton: A set with one element. Eg. {{ 1,2 }}

Pair: A set with two elements.

Ordered Pair: A set with two elements that are ordered. Eg. {1,2}

Intersection:

$$A \cap B = \{x \mid x \in A \ \wedge x \in B\}$$

Union:

$$A \cup B = \{x \mid x \in A \ \vee x \in B\}$$

Difference:

$$A \setminus B = \{x \mid x \in A \ \wedge x \notin B\}$$

Symmetric Difference:

$$A \oplus B = (A \setminus B) \ \cup (B \setminus A)$$

Complement: Where X is the universal set

$$A^c = X \setminus A$$

Absorption: Anything with itself is itself

$$A \cap A = A$$
$$A \cup A = A$$

Commutativity: Order doesn't matter

$$A \cap B = B \cap A$$
$$A \cup B = B \cup A$$

Associative: Brackets don't matter

$$A \cap (B \cap C) = (A \cap B) \cap C$$
$$A \cup (B \cup C) = (A \cup B) \cup C$$

Distributivity: Expanding brackets

$$A \cap (B \cup C) = (A \cap B) \cup (A \cap C)$$
$$A \cup (B \cap C) = (A \cup B) \cap (A \cup C)$$

Double Complement:

$$A = (A^c)^c$$

DeMorgan: Complements flip sign and variable

$$(A \cap B)^c = B^c \cup A^c$$
$$(A \cup B)^c = B^c \cap A^c$$

Duality: Universal and empty sets are complements of each other

$$X^c = \emptyset$$
$$\emptyset^c = X$$

Identity:

$$A \cup \emptyset = A$$
$$A \cap X = A$$

Dominance:

$$A \cap \emptyset = \emptyset$$
$$A \cup X = X$$

Complementation:

$$A \cap A^c = \emptyset A \cup A^c = X$$

Subset Characterisation: A is a subset of B is the same as saying A is the intersection of A and B, and B is the union of A and B

$$A \subseteq B \equiv A = A \cap B \equiv B = A \cup B$$

Contraposition:

$$A^c \subseteq B^c \equiv B \subseteq A$$
$$A \subseteq B^c \equiv B \subseteq A^c$$
$$A^c \subseteq B \equiv B^c \subseteq A$$

Powerset: P(x) is the set $\{A \mid A \subseteq X\}$ of all subsets of X. This includes $\emptyset$ and X. If X is finite and of cardinality n, then P(x) is of cardinality 2^n

Generalised Union:

$$\bigcup_{i \in I} A_i = \{x \mid \exists i (i \in I \wedge x \in A_i)\}$$

Generalised Intersection:

$$\bigcap_{i \in I} A_i = \{x \mid \forall i (i \in I \Rightarrow x \in A_i)\}$$

Set-theoretic notion: Used to capture the notion of ordered pairs with set notation

$$\text{statement to hold: } (a, b) = (c, d) \Leftrightarrow (a = c) \wedge (b = d)$$
$$\text{define: } (a, b) = \{\{a\}, \{a, b\}\}$$

Cartesian Product:

$$A \times B = \{(a, b) | (a \in A) \wedge (b \in B)\}$$

We define the set of A^n of n-tuples over A as follows:

$$A^0 = \{\emptyset\}$$
$$A^{n+1} = A \times A^n$$

Cartesian Product Rules:

$$(A \times B) \cap (C \times D) = (A \times D) \cap (C \times B)$$
$$(A \cap B) \times C = (A \times C) \cap (B \times C)$$
$$(A \cup B) \times C = (A \times C) \cup (B \times C)$$
$$(A \cap B) \times (C \cap D) = (A \times C) \cap (B \times D)$$
$$(A \cup B) \times (C \cup D) = (A \times C) \cup (A \times D) \cup (B \times C) \cup (B \times D)$$

Relation: A n-ary relation is a set of n-tuples. I.e a subset of some cartesian product.

Binary Relation: A set of pairs, or 2-tuples. Being unifiable, less than, is a subset and divides are all binary relations.

Domain: For a relation R

$$dom(R) = \{x \mid \exists y R(x, y)\}$$

Range: For a relation R

$$ran(R) = \{y \mid \exists x R(x, y)\}$$

Relations Definitions:

- A relation is *from* A *to* B if $dom(R) \subseteq A$ and $ran(R) \subseteq B$, we can also say that R is *between* A and B.
- A relation from A to A is a relation *on* A
- Being unifiable is a relation on *Term*
- "$<$" is a relation on integers
- "$\subseteq$" is a relation on P(A)
- "'Acted in" is a relation between actors and films
- $A \times B$ is a relation, the *full* relation from A to B
- $\emptyset$ is a relation
- $\triangle_A = \{(x, x) \mid x \in A\}$ is the identity relation on A
- R^{-1} is the inverse relation of R.
- $R^{-1} = \{(b, a) \mid (a, b) \in R\}$
- All relations are sets, so set operations are applicable to relations.

Relations Properties: Let A be a non-empty set, and R be a relation on A

- R is reflexive iff R(x,x) for all x in A
  i.e for every x value, there is a (x,x) value
- R is irreflexive iff R(x,x) holds for no x in A
  i.e for no x values, does (x,x) exist
- R is symmetric iff R(x,y) $\Rightarrow$ R(y,x) for all x,y in A
  i.e if (x,y) is in the set, then (y,x) is also in the set
- R is asymmetric iff R(x,y) $\Rightarrow \neg$ R(y,x) for all x,y in A
  i.e if (x,y) is in the set, then (y,x) is not in the set
- R is antisymmetric iff R(x,y) $\wedge$ R(y,x) $\Rightarrow$ x = y for all x,y in A
  i.e if (x,y) and (y,x) then the only case this can occur is if x and y are the same
- R is transitive iff R(x,y) $\wedge$ R(y,z) $\Rightarrow$ R(x,z) for all x,y,z in A
  i.e if x maps to y, and y maps to x, then x maps to z

Reflexive, Symmetric and Transitive Closures:

- The full relation is transitive
- Transitive relations are closed under intersection, that is, if R_1 and R_2 are transitive, then so is $R_1 \cap R_2$
- This means that for any binary relation R, there is a unique, smallest, transitive relation R^+ which includes R
- We call R^+ the transitive closure of R

Transitive Closure: Can be defined in terms of union and composition

The transitive closure of R is given as:

$$R^+ = \bigcup_{n \geq 1} R^n$$

The reflexive, transitive closure is

$$R^* = \bigcup_{n \geq 0} R^n = R^+ \cup \triangle_A$$

Equivalence relation: A binary relation that is reflexive, symmetric and transitive. The smallest equivalence relation for a set A is the identity relation $\triangle_A$. The largest equivalence relation on A is the full relation A^2

Partial Order:

- R is a pre-order iff R is transitive and reflexive
- R is a strict partial order iff R is transitive and irreflexive
- R is a partial order iff R is an antisymmetric preorder
- R is a linear iff R(x,y) $\vee$ R(y,x) $\vee$ x = y for all x,y in A
- A linear partial order is also called a total order
- In a total order, every two elements from A are comparable

Co-Domain: Also the range, when the range is a subset of Y

If we have a function f from X to Y, then $f : X \to Y$
If $dom(f) = X$ and $ran(f) \subseteq Y$., then Y is the co-domain of f.

Image and Co-Image:

Let $A \subseteq X$, $B \subseteq Y$, and consider $f : X \to Y$

$f[A] = \{f(x) \mid x \in A\}$ is the image of A under f
i.e Take x, where x is an element of A, and record where it maps to in the domain. (get the range value)

$f^{-1}[B] = \{x \in X \mid f(x) \in B\}$ is the co-image of B under f
i.e Take x, where x is an element of B, and record what maps to x in the domain. (get the domain value)

Surjective: A function $f : X \to Y$ is surjective iff $F[X] = Y$
i.e if every y value has at least one corresponding x value

Injective: iff $f(x) = f(y) \Rightarrow x = y$
i.e every y value has only one corresponding x value.

Bijective: iff it is both surjective and injective
i.e every x value has only one y value, and every y value has only one x value

Composition: The function of $f : X \to Y$ and $g : Y \to Z$ is the function $g \circ f : X \to Z$ defined by:

$$(g \circ f)(x) = g(f(x))$$

This is read as g after f.
We assume that g's domain coincides with f's co-domain, although the composition will make sense as long as $ran(f) \subseteq dom(g)$
$\circ$ is also associative.

Function Composition:

- $g \circ f$ injective $\to f$ injective
- $g \circ f$ surjective $\to g$ surjective
- g, f injective $\to g \circ f$ injective
- g, f surjective $\to g \circ f$ surjective

Partial Functions: Where a domain is unknown, or can be undefined for certain values. Can have a function for part of a domain, and then a different function for another part.

Week 7 - Automata and Natural Languages

Finite Automaton: A 5 -tuple $(Q, \Sigma, \delta, q_0, F)$ where

- Q is a finite set of possible states (eg. {1,2,3})
- Σ is a finite alphabet (eg.{a,b,c})
- $\delta : Q \times \Sigma \to Q$ is the transition function
  (eg. $1 \times a \to 2$ means that from state 1, if we have the alphabet a, we can transition to state 2)
- $q_0 \in Q$ is the start state
- $F \subseteq Q$ are the accept states

Alphabet: denoted Σ and can be any non-empty finite set. Generally we use alphabetical symbols, and sometimes numbers.

String: a string over Σ is a finite sequence of symbols from Σ

Concatenation: We can combine a string y to string x by writing it as xy

Empty String: Denoted as ϵ, and doesn't contain anything

Language: A language over alphabet Σ is a (finite or infinite) set of finite strings over Σ.
$\Sigma*$ denotes the set of all finite strings over Σ

Acceptance: A machine M, will accept a string $v_1...vn$, iff there is a sequence of states $r_0, r_1, ...r_n$ with each $r_i \in Q$ such that

1. $r_0 = q_0$
   i.e start state from string is a valid start state
2. $\delta(r_i, v_{i+1} = r_{i+1})$ for $i = 0...n-1$
   i.e every transition from a state r, and using a letter v leads to another valid state
3. $r_n \in F$
   i.e the final state at the end of the string is the same as the end state

Recognition: M will recognise a language A iff A = { w | M accepts w}

Regular: A language is regular iff there is a finite automaton that recognises it. We can perform regular operations on it.

- Union: $A \cup B$
- Concatenation: $A \circ B = \{xy \mid x \in A, y \in B\}$
- Kleene Star: $A* = \{x_1x_2..x_k \mid k \geq 0, x_i \in A\}$
  note: kleene star contains the empty string ϵ

Deterministic Finite Automaton (DFA): Always has one path it can follow. Once it gets to the end of the string it terminates and we determine if the string is accepted or not.

Non-Deterministic Finite Automaton (NFA): May not always have an avaliable path to follow, and could have multiple paths that a state can follow. May terminate before reaching the end of the string.

Epsilon Transition: Allows a NFA to move from one state to another without consuming input. Helpful for combining the union of two languages.

NFA Definition: Let Σ_ϵ denoted $\Sigma \cup \{\epsilon\}$

Given a NFA that is a 5-tuple $(Q, \Sigma, \delta, q_0, F)$

- Q is a finite set of possible states
- Σ is a finite alphabet
- $\delta : Q \times \Sigma_\epsilon \to P(Q)$ is the transition function
- $q_0 \in Q$ is the start state
- $F \subseteq Q$ are the accept states

Acceptance: For an NFA to accept a string, it has to be possible to make the necessary transitions.

- $r_0 = q_0$
- $r_{i+1} \in \delta(r_i, v_{i+1})$ for $i = 0...n-1$
- $r_n \in F$

Subset Construction: Every NFA has an equivalent DFA, which can be proven by subset construction.

Basically, from a start state 'A' we get a set of states that an alphabet can map to. For example if 'a' goes to states {1,2,3}, then that set will be a state in the DFA, which we can name 'B' in our DFA. If a set contains an end state, then that set will be an accept state for the DFA. If an alphabet character doesn't map to anything, then we create a dead state (empty set), where all characters after go back to the dead state. We continue until all characters are accounted for in each state.

Formally:

Let $N = (Q, \Sigma, \delta, q_0, F)$.
Let $\to_\epsilon^*$ be the reflexive transitive closure of $\to_\epsilon$, which in turn is defined by $s \to_\epsilon$ s' iff s' $\in \delta(s, \epsilon)$

Let E(S) be the "ϵ closure" of $S \subseteq Q$, that is S together with all states reachable from states in S, using only ϵ steps:

$$E(S) = \bigcup_{s \in S} \{s' \in Q | s \to_\epsilon^* s'\}$$

We construct $M = (P(Q), \Sigma, \delta', q_0', F')$ as follows:

- $q_0' = E(\{q_0\})$
- $\delta'(S, v) = \cup_{s \in S} E(\delta(s, v))$
- $F' = \{S \in Q | S \cap F \neq \emptyset\}$

Union of Regular Languages: The class of regular languages is closed under union
We can construct a union of two regular languages by adding a start state, then connecting the start state to the start state of both languages using two ϵ transitions.

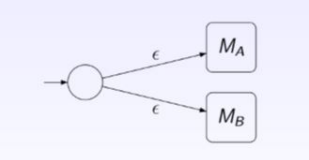

Figure 1: Union of two natural languages M_A and M_B

Composition of Regular Languages: The class of regular languages is closed under $\circ$ (A $\circ$ B)
We can construct a NFA by connecting the end states of A and connecting it to the start state of B.

Kleene Star of Regular Languages: The class of regular languages is closed under Kleene star.
We can construct a NFA by adding a final state as the start state (for ϵ), and connecting the final state to the original accept state.

Regular languages: Closed under intersection, complement, difference, and reversal.

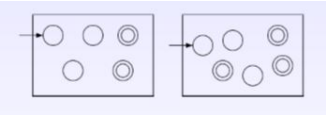

Figure 2: Given two languages A and B

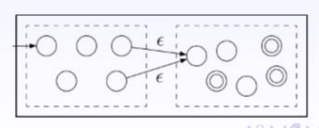

Figure 3: Creating a NFA of A $\circ$ B

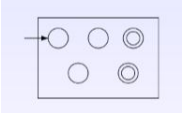

Figure 4: Given a language A

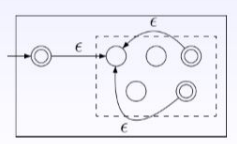

Figure 5: Creating a NFA to recognise A^*

Minimal DFA: Has the smallest possible number of states

1. Reverse the NFA
2. Determinize the result
3. Reverse again
4. Determinize

Reversing an NFA with an initial state q_0 and accept states $F \neq \emptyset$:

1. If F = {q}, then let q be the accept state
2. Otherwise add a new state q which becomes the only accept state; then, for each state in F, add an ϵ transition to q.
3. Reverse every transition in the resulting NFA, making q the initial state and q_0 the only accept state

Week 8 - Non-Regular Languages & Context-Free Languages

Regular Expressions: The regular expressions over an alphabet $\Sigma = \{a_1, ...a_n\}$ are given by the grammar:

$$regexp \rightarrow a_1 \mid ... \mid a_n \mid \epsilon \mid \emptyset \mid regexp \ \cup \ regexp \mid regexp \ regexp \mid regexp*$$

$$\begin{aligned} L(a) &= \{a\} \\ L(\epsilon) &= \{\epsilon\} \\ L(\emptyset) &= \emptyset \\ L(R_1 \cup R_2) &= L(R_1) \cup L(R_2) \\ L(R_1 R_2) &= L(R_1) \circ L(R_2) \\ L(R^*) &= L(R)^* \end{aligned}$$

Regular Expressions vs Automata: L is regular iff L can be described by a regular expression.

- Start by constructing a NFA from the innermost expressions.
- Build an NFA with at most one accept state, by using ϵ transitions from the previous final states to the new final state
- Label arcs with regular expressions
- Repeatedly eliminate states that are neither start nor accept states.

Laws for Regular Expressions:

$$\begin{aligned} A \cup A &= A \\ A \cup B &= B \cup A \\ (A \cup B) \cup C &= A \cup (B \cup C) = A \cup B \cup C \\ (A \ B) \ C &= A \ (B \ C) = A \ B \ C \\ \emptyset \cup A &= A \cup \emptyset = A \\ \epsilon \ A &= A \ \epsilon = A \\ \emptyset \ A &= A \ \emptyset = \emptyset \\ (A \cup B) \ C &= A \ C \cup \ B \ C \\ A(B \cup C) &= A \ B \cup A \ C \\ A \ (B \cup C) &= A \ B \cup A \ C \\ (A^*)^* &= A^* \\ \emptyset_* &= \epsilon^* = \emptyset \\ (\epsilon \cup A)^* &= A^* \\ (A \cup B)^* &= (A^* B^*)^* \end{aligned}$$

Pumping Lemma: A standard tool for proving languages are non-regular.
If A is regular then there is a number p such that for any string $s \in A$ with $|s| \geq p$ s can be written as $s = xyz$ satisfying:

- $xy^i z \in A$ for all $i \geq 0$
- $y \neq \epsilon$
- $|xy| \leq p$

Where p is the pumping length.

Grammar: A grammar is a set of substitution rules or productions.

Derivation: Using two grammar rules to rewrite a string.

Variable: 'A' is a variable, it can represent other letters.

Terminal: 'a', 'b', '0', '1' are terminals. They have the same value (cannot be substituted).

Sentence: A string of terminals (cannot substitute anymore)

Sentential form: Intermediate strings that mix variables and terminals during the process of derivation.

Context-free language (CFL): A language which can be generated by some context-free grammar is a context free language.
More formally, it can be represnted as a 4-tuple (V, Σ, R, S)

1. V is a finite set of variables
2. Σ is a finite set of terminals
3. R is a finite set of rules, each consisting of a variable (the left-hand side) and a string in $(V \cup \Sigma)^*$ the right hand side
4. S is the start variable

The binary relation $\Rightarrow$ on $(V \cup \sigma)^*$ is defined as follows:
Let u,v,w $\in (V \cup \Sigma)^*$. Then $uAw \Rightarrow uvw$ iff $A \rightarrow v$ is a rule in R. That is $\Rightarrow$ captures a single derivation step.

Let $\overset{*}{\Rightarrow}$ be the *reflexive transitive closure* of $\Rightarrow$.

$$L(G) = \{s \in \Sigma^* | S \overset{*}{\Rightarrow} s\}$$

Parse Trees: Represents the order from which a sentence is derived. They differ in the order we choose to replace variables. *Leftmost* derviation means we start replacing variables from the left.

Ambiguous: If a grammar has different parse trees for a sentence. Sometimes the same language can be represented by different grammar, one which is ambiguous and one which may not be ambiguous.

Week 9 - Pushdown Automata

Pushdown Automaton: A finite-state automaton, equipped with a stack. Instead of consuming an input, it can also move the input into a stack and later 'pop' it. Based on the input symbol, top stack symbol and current state, the PDA will decide which state to go to next and the operation it will apply to the stack (ie. pop, push or both)

More formally, a pushdown automaton is a 6-tuple $(Q, \Sigma, \lceil, \delta, q_0, F)$

- Q is a finite set of states
- Σ is a finite input alphabet
- $\lceil$ is the finite stack alphabet
- $\delta : Q \times \Sigma_\epsilon \times \lceil_\epsilon \rightarrow P(Q \times \lceil_\epsilon)$ is the transition function
- $q_0 \in Q$ is the start state
- $F \subseteq$ Q are the accept states

Example Transitions:

$\delta(q_5, a, b) = \{(q_7, \epsilon)\}$
If in state q_5, if the input is a, and the top of the stack is b, then consume a, pop b and move to state q_7

$\delta(q_5, \epsilon, b) = \{(q_6, a), (q_7, b)\}$
If the state is q_5 and the top of the stack is b, then either replace b by a and go to state q_6 or leave the stack as is and go to state q_7. Neither of these options consumes an input symbol.

Recognisers: Every context free language has a PDA which can recognise said language. Every PDA also recognises a context-free language.

CFL Properties: Closed under union, concatenation, Kleene star, reversal. Not closed under intersection or complement.

Deterministic PDA: Less powerful than PDAs as it doesn't allow epsilon transitions and doesn't know when to start popping a stack or where the middle is.

Week 10 - Well-foundness and Turing Machines

Partial Functions: More commonly used in computer science, as a function may be undefined for some x.

Notation: $f : F \hookrightarrow Y$ means that f has a domain that is a subset of X, but f(x) may be undefined for some $x \in X$

Total Functions: A function that is defined for all inputs of the correct type. (I.e defined for the domain it's given, not necessarily all real numbers)

Collatz's problem: Deciding whether or not a function is total and will terminate for all n.

Termination:
We can prove termination by looping through a program.

1. The measure is a natural number
2. The measure gets smaller with each loop iteration

The program must terminate for all input, because a natural number cannot be made smaller indefinitely (stops at 0 or 1)

Well-founded: A binary relation $\prec$ over some set X is well-founded iff there is no infinite sequence of X-elements x_1, x_2, x_3 such that

$$x_1 \succ x_2 \succ x_3....$$

Hence (X, $\prec$) is a well-founded structure (will end at some point)

Component-Wise Ordering: The total of each components contributes to its ordering. Eg. (1,1) and (2,0) would be the same (level/order)

$$(x_1, x_2) \preceq (y_1, y_2) \text{ iff } x_1 \leq y_1 \wedge x_2 \leq y_2$$
$$p \preceq q \text{ iff } p \preceq q \wedge p \neq q$$

Hasse Diagram: Can be used for a component wise ordering of $\mathbb{N} \times \mathbb{N}$ (i.e tuples)
Values increase as you travel up edges

Lexographical ordering: Similar to the alphabet, the earlier positions take precence when ordering. Eg. (1,100) $\preceq$ (100,1)

Well founded orderings on tuples:

- **Theorem:** If $\prec$ is well-founded, then so is its component wise extension to tuples.
- **Theorem:** If $\prec$ is well-founded then so is its lexicographic extension to tuples.
- If all of the elements and the ordering function is well-founded, then any combination of tuples with said element and ordering can be grouped component wise or lexicographically.

Well-founded Induction: An induction principle that goes with well-founded relations. Given that a structure is well-founded (X, $\prec$), we can prove a statement "for all x $\in$ X, S(x)" by assuming that S(x') holds for all $x' \prec x$, and use that to establish S(x).

Undeciable Problems: Problems that have no algorithm for solving it.

Turing Machine: Has an unbounded tape through which it takes its input and performs computations. It is able to both write and read to the tape, and move left and right over the tape. The machine's accept and reject states are irrespective of the tape head.
Formally:
A Turing Machine is a 7-tuple $M = (Q, \Sigma, \lceil, \delta, q_0, q_a, q_r)$ where:

- Q is a finite set of states
- $\lceil$ is a finite tape alphabet, including the blank character $\sqcup$
- $\Sigma \subseteq \lceil \setminus \{\sqcup\}$ is the input alphabet (Sigma is a subset of the tape alphabet without the blank character)
- $\delta : Q \times \lceil \to Q \times \lceil \times \{L, R\}$ is the transition function
- q_0 is the initial state
- q_a is the accept state
- $q_r (\neq q_a)$ is the reject state

Transition function $\delta(q_i, x) = (q_j, y, d)$ depends on two things

1. The current state q_i
2. The current symbol x under the tape head

It consists of three actions

1. change state to q_j
2. overwrite tape symbol x by y
3. move the tape head in the direction of d

Computations:

For all $q_i, q_j \in Q$, $a, b, c \in \lceil$ and $u, v \in \lceil^*$ we have

$$\begin{aligned} uq_ibc &\Rightarrow ucq_iv \text{ if } \delta(q_i, b) = (q_j, c, R) \\ q_ibv &\Rightarrow q_jcv \text{ if } \delta(q_i, b) = (q_j, c, L) \\ uaq_ibv &\Rightarrow uq_jacv \text{ if } \delta(q_i, b) = (q_i, c, L) \end{aligned}$$

The start configuration of M on input w is q_0w
M accepts w iff there is a sequence of configurations $C_1, C_2 ... C_k$ such that

1. C_1 is the start configuration q_0w
   It is a valid start state
2. $C_i \Rightarrow C_{i+1}$ for all $i \in \{1...k-1\}$
   Every configuration has a valid transition
3. The state of C_k is q_a
   The final configuration ends in an accept state

Turing recognisable: A language A is Turing reconisable (or recursively enumerable) iff A = L(M) for some TUring machine M. There exists a Turing Machine that recognises it.

Turing behaviors: A Turing machine M can accept an input w, can reject w or fail to halt on w.

Turing decidable: If a Turing machine M halts on all input of language A, then M decides A.

Week 11 - Decidable and Undecidable Languages

Multitape Machines: A multitape Turing machine has k tapes. It takes input on tape 1 and the other tapes are blank. It can write and move to other tapes, and specifies through the transitions how k tape heads behave when the machine is in state q_i and reading some alphabet.

$$\text{Transition Function: } \delta : Q \times \lceil^k \rightarrow Q \times \lceil^k \times \{L, R\}^k$$

eg.

$$\delta(q_i, a_1, ...a_k) = (q_j, (b_1, ...b_k)(d_1...d_k))$$

Theorem: A language is Turing recognisable iff some multitape Turing Machine recognises it. We can simulate a multitape machine M by using a standing Turing machine S. Instead of using seperators #, we can just put new string segments into a new tape.

Nondeterministic Turing Machines: If some computation leads to an accept state, then the machine accepts the input. A nondeterministic Turing machine has a transition function of type:

$$\delta : Q \times \lceil \rightarrow P(Q \times \lceil \times \{L, R\})$$

Theorem: A language is Turing recognisable iff some nondeterministic Turing machine recognises it. I.e every nondeterministic Turing machine N can be simulated by a deterministic Turing machine D.

Enumerator: A Turing machine built to generate all strings in a language L. For an enumerator to enumerate a language L, for each string in L, it must eventually print said string.

Theorem: L is Turing recognisable iff some enumerator enumerates L.
Proof:

1. Let E be an enumerator for L
2. Let w be the input
3. Simulate E. For each string s output by E, if s = w then accept.

Alternatively:

1. Let i = 1, and M be a Turing machine that recognises L. E produces strings 's_i'.
2. Simulate M for i steps on each of $s_1, ...s_i$
3. For each accepting computation, print that s
4. Increment i and go to step 2

Acceptance Problem: For DFAs, the acceptance problem is if a DFA D accepts input w.

Theorem: A_{DFA} is a decidable language
Proof:
We can use a Turing Machine M to simulate a DFA D. For example, a turing machine M can have a tape with the input:
Q(states) ## Σ(alphabet) ## δ(transitions) ## q_0(initial state) ## F(final states) ## w(string) $

eg.

#1...n ##ab..z##1a2#...# nbn##1##3 7 ## baa...$

M first checks if the first five components represent a valid DFA, and if it doesn't it will reject it. Then M simulates the moves of D, keeping track of D's state and the current position in w by writing the details on the tape after $. When the last symbol w has been processed, M accepts if D is in a state in F, otherwise rejects.

Theorem: A_{NFA} is decidable. We can do this by using a halting Turing Machine to translate an NFA to a DFA, and then use the previous theorem to run it.

Theorem: E_{DFA} is decidable (Determining if a DFA is empty)

$$E_{DFA} = \{< D > | D \text{is a DFA and L(D)} = \emptyset\}$$

Proof:

1. Design a Turing machine that takes <D> = $(Q, \Sigma, \delta, q_0, F)$ as input and performs a reachability analysis:
2. Set reachable = $\{q_0\}$, D's start state
3. Set new = $\{q | \delta(m, x) = q$, m $\in$ reachable$\}$ $\setminus$ reachable
4. If new $\neq \emptyset$, set reachable = reachable $\cup$ new and go to step 3
5. If reachable $\cap$ F = $\emptyset$ accept, otherwise reject (If we can't reach the accept state, then accept the DFA)

Theorem: EQ_{DFA} DFA equivalence is decidable

$$EQ_{DFA} = \{< A, B > | \text{A and B are DFAs and L(A)} = \text{L(B)}\}$$

Proof:
We can construct DFAs using intersection, union and complement which are mechanistic and finite. Therefore a Turing Machine M can perform those actions. From a DFA A and B, M can produce a DFA C to recognise

$$\text{L(C)} = (\text{L(A)} \cap \text{L(B)}^c) \cup (\text{L(A)}^c \cap \text{L(B)})$$
$$\text{L(C)} = \emptyset \text{ iff L(A)} = \text{L(B)}$$

We can use the emptiness checker on C.

Theorem: A_{CFG} Generation by CFGs is Decidable.

$$A_{CFG} = \{< G, w > | \text{G is a CFG that generates w}\}$$

Proof:
We can reqrite any CFG to Chomsky Normal Form. Each production takes one of two forms:

$$\text{A} \rightarrow \text{B C or A} \rightarrow \text{a and S} \rightarrow \epsilon \text{ (start)}$$

For every grammar in Chomsky Normal Form form, if string w can be derived, then the derivation has exactly 2(w) - 1 steps. It is similar to a binary tree. To decide A_{CFG} we can simply try out all possible derivations of that length, in finite time and see if one generates w.

Theorem: E_{CFG} CFG Emptiness is Decidable

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

Proof:
We can design a Turing Machine which takes $< G >= (V, \Sigma, R, S)$ as input, and performs a 'producer' analysis

1. Set producers = Σ, all of G's terminals
2. Set new = $(\{A | A \to U_1 ... U_n \in R\}$ and $\{A | \{U_1 ... U_n\} \subseteq$ producers$\}) \setminus$ producers
3. If new $\neq \emptyset$ producers = producers $\cup$ new and go to step 2
4. If S $\in$ producers, reject otherwise accept

Theorem: Every CFL is decidable.

Proof: We can specialise a decider S. Let G_0 be a CFG for L_0. The decider for L_0 simply takes input w and runs S on $< G_0, w >$

Undecidable problems: If a turing machine accepts a given string, then the turing machine may fail to halt.

Theorem: A_{TM} Turing machine acceptance is undecidable.

$$A_{TM} = \{< M, w > \mid \text{M is a TM and M accepts w}\}$$

Proof:
Assume (for contradiction) that A_{TM} is decided by a TM H, where H accepts if M accepts w, and rejects if M rejects w.
Using H we can construct a Turing machine D which decides whether a given machine M fails to accept its own encoding $< M >$:

1. Input is $< M >$, where M is some Turing machine.
2. Run H on $< M, < M >>$
3. If H accepts, reject. If H rejects, accepts.

Step 3 produces a contradiction, as we're saying if M accepts, then reject M. Therefore no such Turing Machine can evaluate it's own encoding.

Bijections and Enumeration: A bijection in $\mathbb{N} \to X$ gives us an enumeration of the set X.

Sizes of Infinite Sets:

- card(X) $\leq$ card(Y) iff there is a total, injective f: X $\to$ Y
- card(X) = card(Y) iff card(X) $\leq$ card(Y) and card(Y) $\leq$ card(X)

Countable: X is countable iff card(X) $\leq$ card($\mathbb{N}$)

Countably Infinite: X is countably infinite iff card(X) = card($\mathbb{N}$)

Uncountable:$\mathbb{N} \to \mathbb{N}$ is uncountable, and can be shown through diagonalization

Theorem: There is no bijection h: $\mathbb{N} \to (\mathbb{Z} \to \mathbb{Z})$

Proof:
Assume h exists. Then h(0), h(1) ... h(n) ... contains every function in $\mathbb{Z} \to \mathbb{Z}$ without duplicates.
Now constructing f: $\mathbb{Z} \to \mathbb{Z}$ as f(n) = h(n)(n) + 1.
Then f $\neq$ h(n) for all n, so we have a contradiction.

Week 12 - Reducibility

Theorem: A language L is decidable iff both L and the complement of L is Turing recognisable. If L is decidable, then L and l^c are recognisable.

Theorem: E_{TM}, Emptiness of a Turing Machine is Undecidable.

$$E_{TM} = \{< M > \mid \text{M is a TM and L(M)} = \emptyset\}$$

Proof:
A_{TM} is reducable to E_{TM}. Given $< M, w >$, a Turing machine can modify the encoding of M so as to turn M into M' which recognises L(M) $\cap$ {w}. Running the machine then putting the output into the modified machine, we receive a contradiction.

Rice's Theorem: Every interesting semantic Turing machine property is undecidable. A property is interesting iff it holds for some TMs but not others. It is semantic iff it holds for all Turing machines that accept the same language.