Languages

- **Alphabet Σ = a non-empty finite set of letters (symbols)**
- **Language = a (potentially infinite) set of finite strings over an alphabet**
- **Σ^* = the language of all finite strings (words) over Σ**
- ∅ = {} = the empty language ≠ the language containing the empty string = {ε}
  - In fact $\varnothing^* = \{\varepsilon\}$
  - Note $|\varepsilon| = 0$ although we write it as a single character

Regular languages

DFA

- **A DFA $M=(Q, \Sigma, q_0, F, \delta)$ consumes an input from Σ^* one letter at a time and moves through its states Q in accordance with $\delta: Q \times \Sigma \mapsto Q$ starting at q_0**
  - Technically δ must be defined for all possible inputs but convention is that anything not specified maps to a dead state (a non-accepting state with no outward transitions)
  - Accepts a word iff state after consuming the entire word is in F
- If defining δ using a transition table, → denotes the start state and * denotes the accepting states
- **If defining δ using a diagram, the starting state is denoted by an unlabeled incoming transition without a source and accepting states are denoted by an extra circle around them**
- **$\hat{\delta}: Q \times \Sigma^* \mapsto Q$ describes the state moved to after consuming a string**
  - $\forall q \in Q.\ \hat{\delta}(q, \varepsilon) = q$
  - $\forall q \in Q.\ \forall w,x \in \Sigma^*.\ \forall a \in \Sigma.\ w=xa \Rightarrow \hat{\delta}(q, w) = \delta(\hat{\delta}(q, x), a)$
- **L(M) = language recognised by M = set of words accepted by M = $\{w \in \Sigma^*: \hat{\delta}(q_0, w) \in F\}$**
- A language L if regular iff there exists a DFA M such that L = L(M)
- Let M be a DFA, $w \in \Sigma^*$, and $w_1 w_2 \ldots w_n$ be the decomposition of w into symbols. Then the run of M on ε is (q_0) and the run of M on w is $(r_0=q_0, r_1, \ldots, r_n)$ such that $r_i=\delta(r_{i-1}, s_i)$ for all i
  - A run is an accepting run (the word is recognised by M) iff $r_n \in F$

Closure properties of regular languages

- **Regular languages are closed under complementation** — if $M=(Q, \Sigma, q_0, F, \delta)$ recognises L then $M'=(Q, \Sigma, q_0, Q\backslash F, \delta)$ reconginses $\neg L=\Sigma^*\backslash L$
  - Run M and take negation of answer
- **Regular languages are closed under <u>finite</u> intersection —** if $M_1=(Q_1, \Sigma, q_1, F_1, \delta_1)$ recognises L_1 and $M_2=(Q_2, \Sigma, q_2, F_2, \delta_2)$ recognises L_2 then $M'=(Q_1 \times Q_2, \Sigma, (q_1, q_2), F_1 \times F_2, \delta')$ where $\delta'((x,y), a) = (\delta_1(x, a), \delta_2(y, a))$ recognises $L_1 \cap L_2$
  - Run M_1 and M_2 in parallel and take conjunction of answers
- **Regular languages are closed under <u>finite</u> union —** if $M_1=(Q_1, \Sigma, q_1, F_1, \delta_1)$ recognises L_1 and $M_2=(Q_2, \Sigma, q_2, F_2, \delta_2)$ recognises L_2 then the same construction of intersection but accepting states $(F_1 \times Q_2) \cup (F_2 \times Q_1)$ recognises $L_1 \cup L_2$
  - Run M_1 and M_2 in parallel and take disjunction of answers
- Regular languages are closed under set difference as this can be rewritten in terms of the previous closure properties (e.g. $L_1\backslash L_2 \equiv L_1 \cap \neg L_2$)
- Regular languages are <u>not</u> closed under <u>infinite</u> union/intersection as we could union in all the elements of a non-regular L one at a time (every finite language is regular) or intersect away all the elements of $\neg L$ from Σ^* one at a time (these are the complements of a finite language and so are regular)

NFAs

- **An NFA allows transitions to be labeled with ε and can have any natural (including zero) number of transitions out of each state for each symbol** whereas a DFA only has transitions for elements of Σ and has exactly 1 for each state
- There are arbitrarily many εs before/after each symbol in a word — ε-transitions are a powerful tool for non-determinism
- **The only difference between NFA and DFA is in an NFA $\delta: Q \times (\Sigma \cup \{\varepsilon\}) \mapsto 2^Q$** — DFA can be viewed as a type of NFA where the output of δ is always a singleton set (and there are no ε-transitions)
    - If δ gives ∅ then the machine is deemed to have entered a dead state and that run dies without accepting (but there may be other runs for the word that are accepting)
- NFAs can in effect have multiple start states as we can take the one start state we are allowed and add ε-transitions to our multiple meaningful start states
- A DFA creates a path of computation accepting the word iff the run ends at an accepting state whereas an NFA creates a tree of computation **accepting the word iff a branch (a run) ends at an accepting state**
- **ECLOSE(q) = {s ∈ Q: s can be reached from q by following only ε-transitions}**
    - *The relation x~y iff x ∈ ECLOSE(y) is a preorder as it is equivalent to directed reachability in the digraph with edges where there are ε-transitions. In fact we can always combine states to produce a (acyclic in terms of ε-transitions) equivalent NFA for which this is antisymmetric and so a partial order*
- $\delta\hat{}: Q \times \Sigma^* \mapsto 2^Q$ is defined as follows:
  $\forall q \in Q.\ \delta\hat{}(q, \varepsilon) = \text{ECLOSE}(q)$
  $\forall q \in Q.\ \forall w,x \in \Sigma^*.\ \forall a \in \Sigma.\ w=xa \Rightarrow \delta\hat{}(q, w) = \bigcup_{q' \in \delta\hat{}(q, x)} \text{ECLOSE}(\delta(q',a))$
- **$L(M) = \{w \in \Sigma^*: \delta\hat{}(q_0, w) \cap F \neq \varnothing\}$**

NFA ⇔ DFA

- **Theorem: NFAs and DFAs have equal computational power**. That is for every NFA we can construct a DFA that recognizes the same language (and vice versa)
- Proof (Subset construction): It is clear that every DFA can easily be converted into an NFA (NFAs are at least as powerful as DFAs). It remains to show that NFAs are no more powerful than DFAs (for every NFA we can construct an equivalent DFA).
  Let $N=(Q, \Sigma, q_0, F, \delta)$ be an arbitrary NFA.
  Consider $\mathbf{M=(2^Q,\ \Sigma, ECLOSE(q_0), \{S \in 2^Q: S \cap F \neq \varnothing\}, \delta_1)}$ **where**
  $\mathbf{\forall q \in 2^Q.\ \forall a \in \Sigma.\ \delta_1(X, a) = \bigcup_{x \in X} ECLOSE(\delta(x,a))}$
  By induction on $|w|$, $\forall w \in \Sigma^*.\ \delta_1\hat{}(ECLOSE(q_0), w) = \delta\hat{}(q_0, w)$

Regular expressions

- $L(\varnothing) = \varnothing$
- $L(\varepsilon) = \{\varepsilon\}$
- $L(a) = \{a\}$
- $L(R_1 + R_2) = L(R_1) \cup L(R_2)$
- $L(R_1R_2) = L(R_1)L(R_2) = \{w_1w_2 : w_1 \in L(R_1) \text{ and } w_2 \in L(R_2)\}$
  - Hence, $L(\varepsilon)L(\varnothing) = \varnothing$
- $L(R_1^*) = L(R_1)^* = \{\varepsilon\} \cup L(R_1) \cup L(R_1)L(R_1) \cup \ldots$
  - Hence, $L(\varnothing^*) = \{\varepsilon\}$
- * has highest precedence and + has lowest precedence
- **L is generated by a regular expression ⇔ L is regular ⇔ there exists a DFA that recognises L ⇔ there exists an NFA that recognises L**
  - Corollary: **Regular languages are closed under concatenation and kleene star**

Regex ⇒ NFA

- Proof (Thompson's Construction):
  L(∅) is a non-accepting starting state
  L(**ε**) is an accepting starting state
  L(a) is a starting state and an accepting state transitioned to from starting state on reading an a (all other transitions are unspecified and so are to dead states)
  $L(R_1+R_2)$ is a starting state with **ε**-transitions to starting states of NFAs for $L(R_1)$ and $L(R_2)$
  $L(R_1R_2)$ is an NFA for $L(R_1)$ with accepting states made non-accepting but gaining ε-transitions to the starting state of an NFA for $L(R_2)$
  $L(R_1^*)$ is a new starting state q_0 with ε-transition to the starting state q_1 of an NFA for $L(R_1)$ with ε-transitions from accepting states to q_1

Regex $\Leftarrow$ NFA

- Proof:
  A generalised NFA (GNFA) is an NFA with transitions labelled with regexes such that the start state has transitions to every other state and no incoming transitions (including no self-loops) and there is a single accepting state and this has transitions from every other state and no outgoing transitions (including no-self loops).
  NFA $\Rightarrow$ GNFA: We can add dummy start and accepting states with ε-transitions to/from the originals. Missing transitions can be labeled with $\varnothing$ (like dead states it is acceptable to omit these in practice).
  GNFA $\Rightarrow$ Regex: If there are two states, there is one transition and we are done. Otherwise, **we can remove a state and update the remaining transitions so that no information is lost.**
  Let q_x be the state that is being removed and δ_l be the labelling function, we can define δ_l' as $\delta_l'(q_a, q_b) = \delta_l(q_a, q_b) + \delta_l(q_a, q_x)\delta_l(q_x, q_x)^*\delta_l(q_x, q_b)$ for all (q_a, q_b) for which there is a transition (that is $q_a \notin \{q_0, q_x\}$ and $q_b \notin \{q_f, q_x\}$).

(Ir)regularity

- As DFAs have finitely many states there are countably (infinitely) many of them. As the set of all languages over a given non-empty Σ is the powerset of the countably infinite set Σ^*, there are uncountably many languages (over Σ), and so as each DFA only recognises one language there are uncountably many irregular languages
- **Every finite language is regular** as we can construct a DFA with a unique accepting state for each word in the language if necessary — an infinite language being regular (being able to exactly recognise the necessary words (of arbitrary length) with finite states as our only memory of the past) is interesting
- $\{0^n1^m\}$ is regular as a DFA can keep track of which symbol we saw most recently
- $\{0^n1^m : n \equiv^3 m\}$ is regular as a DFA can keep track of the (finite domained) residues (modulo 3) of the numbers of 0s and 1s it has seen
- $\{0^n1^m : n = m\}$ is irregular as a DFA cannot keep count of the unbounded number of 0s it has seen
- As regular languages are closed under complementation and complementation is self-inverse, **irregular languages are closed under complementation**
- As regular languages are closed under reversal and reversal is self-inverse, irregular languages are closed under reversal
- Irregular languages are <u>not</u> closed under union as $L \cup \neg L \equiv \Sigma^*$ which is regular
- Irregular languages are <u>not</u> closed under intersection as $L \cap \varnothing \equiv \varnothing$ which is regular

Proving irregularity: Myhill–Nerode theorem

- If two different strings bring a DFA to the same state and have the same contents from that point onwards they will have the same run from that point onwards
- **Words x and y in Σ^* are distinguishable by a (not necessarily regular) language L iff $\exists z \in \Sigma^*$: exactly one of xz and yz is in L** — z is called a distinguishing extension
- Words are indistinguishable iff they are not distinguishable
  - **Indistinguishability (denoted $\equiv_L$)** is an equivalence relation on Σ^* — *distinguishability is not an equivalence relation (e.g. is not reflexive)*
- **Index of a language L = number of equivalence classes of $\equiv_L$**
- **Myhill-Nerode Theorem: L is regular iff $\equiv_L$ has finite index.** Moreover, index of $\equiv_L$ is the number of states in the smallest DFA that recognizes L.
  Proof: Out of scope
- **To prove that a language L is irregular it suffices to demonstrate an infinite set of strings that are (pairwise) distinguishable by L** as these must all be in different equivalence classes

Proving irregularity: Pumping lemma

- Given an infinite language over a finite alphabet and a DFA there must be words in the language that are longer than the number of states in the DFA — L(M) is infinite iff there is a cycle in M that is reachable from the start state and from which an accepting state can be reached
- Intuition for pumping lemma: If an infinite language is regular, then each sufficiently long (longer than number of states in the DFA) word contains a part corresponding to the loop in the run of the DFA, this part of the word can be repeated arbitrarily many times (and can be removed) to produce words also in the language
- **Pumping lemma: If L is a regular language, then $\exists m \in \mathbb{Z}$: $\forall w \in \{w \in L: |w| \geq m\}$. $\exists x,y,z \in \Sigma^*$: $w=xyz \wedge |y| \geq 1 \wedge |xy| \leq m \wedge \forall i \in \mathbb{N}_{\geq 0}$. $xy^iz \in L$**
  - x, y, z partition the run into before the first loop, the first loop, and the remainder of the string
  - <u>The converse does not hold, we cannot use pumping lemma to prove regularity</u> like we can with Myhill-Nerode
- Contrapositive: If for an arbitrarily large pumping length m, $\exists w \in \{w \in L: |w| \geq m\}$: $\forall x,y,z \in \Sigma^*$: ($w=xyz$ and $|y| \geq 1$ and $|xy| \leq m$) $\Rightarrow$ $\exists i \in \mathbb{N}_{\geq 0}$: $xy^iz \notin L$, then L is irregular
- Irregularity proof structure:
  Assume for the sake of contradiction that L is regular. Let m be the pumping length of L.
  Consider $w=a^{f(m)}\ldots^{g(m)}$ where $f(m) \geq m$ [our choice of word written in terms of m]
  Pick arbitrary x, y, z such that $w=xyz$ and $|xy| \leq m$ and $|y| \geq 1$
  Deduce $x=a^{\alpha}$, $y=a^{\beta}$, $z=a^{\gamma}\ldots^{g(m)}$, $\beta \geq 1$, $\alpha+\beta \leq \underline{m}$, $\alpha+\beta+\gamma = \underline{f(m)}$
  Pick i (often 0 or 2 is a good choice) **such that we can argue that $xy^iz=a^{\alpha+i\beta+\gamma}\ldots^{g(m)} \notin L$**

Grammars

Grammas

- G=(V, Σ, R, S) is a grammar where V is the finite set of variables (non-terminals), Σ is the finite set of terminals, R is the finite set of production rules, and S is the start variable
  - Convention is that non-terminals are uppercase and terminals are lowercase
  - Production rules are strings of the form $\alpha \to \beta$ where $\alpha, \beta \in (V \cup \Sigma)^*$ and $\alpha \neq \varepsilon$
    - In practice we abbreviate rules $\alpha \to \beta$, $\alpha \to \gamma$ to $\alpha \to \beta|\gamma$
- **We write $\alpha \Rightarrow \beta$ iff there exists a production rule $\alpha \to \beta$**
- **We write $\alpha \Rightarrow^* \beta$ iff there exists a finite sequence of production rules $\alpha \to \ldots \to \beta$ — this could be a sequence of length 0**
- **$L(G) = \{w \in \Sigma^* : S \Rightarrow^* w\}$**
- **A grammar is ambiguous iff the same string can be derived by distinct left-most (always the left-most non-terminal is expanded)** (or equivalently right-most) **derivations**
- **A grammar is inherently ambiguous iff it is ambiguous and there is not an equivalent unambiguous grammar**
- **Chomsky hierarchy: Let $A, B \in V$, $x \in \Sigma^*$, and $\alpha, \beta, \gamma \in (V \cup \Sigma)^*$:**
  - Type-3 **regular iff all rules are of the form $A \to xB$ or $A \to x$** (right linear form) (or equivalently all rules are of the form $A \to Bx$ or $A \to x$ (left linear form)) — cannot mix and match right and left linear or we may be context-free without being regular
  - Type-2 **context-free iff all rules are of the form $A \to \alpha$**
  - Type-1 context-sensitive iff all rules are of the form $\alpha A \beta \to \gamma$ ($\gamma \neq \varepsilon$)
  - Type-0 **recursively enumerable iff all rules are of the form $\alpha \to \beta$**

Linear grammas

- **In a strictly linear grammar the terminals in rules are single letters instead of strings**
- **Every linear grammar can be rewritten as an equivalent strictly linear grammar**
  Proof: TO DO
- **DFA ⇒ Strictly right linear grammar**
  Construction: Let $M=(Q, \Sigma, q_0, F, \delta)$ be an arbitrary DFA. Then $G=(Q, \Sigma, R, q_0)$ where $R = \{q \to aq' : \delta(q, a) = q'\} \cup \{q \to \varepsilon : q \in F\}$ is an equivalent grammar
- **DFA ⇒ Strictly left linear grammar**
  Construction: Let $M=(Q, \Sigma, q_0, F, \delta)$ be an arbitrary DFA. Then $G=(Q \cup \{q^*\}, \Sigma, R, q^*)$ where $R = \{q' \to qa : \delta(q, a) = q'\} \cup \{q^* \to q : q \in F\}$ is an equivalent grammar
  (that acts like M extended to have a single accepting state q^*)
- **These constructions are reversible to give the converses**

Context-free languages

Push-down automata (PDA)

- **$M=(Q, \Sigma, \Gamma, \delta, q_0, F)$ is a pushdown automaton (an NFA augmented with a stack with stack alphabet Γ). $\delta: Q \times (\Sigma \cup \{\varepsilon\}) \times (\Gamma \cup \{\varepsilon\}) \mapsto 2^{Q \times (\Gamma \cup \{\varepsilon\})}$**
- Unlike NFA and DFA, deterministic PDA are less powerful than non-deterministic PDA — we only consider non-deterministic in this module
- **Transitions on a state transition diagram are labeled a,b→c meaning read a, pop b (from top of stack), push c (onto top of stack)** — this corresponds to $\delta(q, a, b) \ni (q', c)$ in the fully formal model
  - Much like reading, pop/push ε always succeeds and never does anything so we can use ε to skip any action we don't want to perform for a particular transition
- For convenience we place a special symbol at **the bottom of the stack (by convention $) so if we read it we know the stack is empty**
  - We have **transitions $\varepsilon,\varepsilon\rightarrow\$$ and $\varepsilon,\$\rightarrow\varepsilon$ on the start and accepting states respectively**
- For convenience we **treat the stack as a string (read from top to bottom) and allow substrings to popped and pushed** (pushing $b_1 \ldots b_r$ pushes b_r … then … then b_1)
- **The condition for acceptance is the same as NFAs (there exists a run which ends in an accepting state)** but in practice we often construct the transitions so that accepting states can only be entered with an empty stack

PDA ⇐ CFG

- **Theorem: Every CFG is recognised by a PDA**
  Proof: We can encode every grammar rule in a single self-loop. For every rule (or equivalently every non-terminal)**, add an ε-transition popping the LHS off the stack and pushing the RHS onto the stack. For every terminal that can be pushed to the stack, add a transition popping it off the stack when read and pushing nothing (ε) onto the stack.** For example S→0S1|ε gives: ε,S→0S1; ε,S→ε; 0,0→ε; 1,1→ε.
  Create a PDA with 3 states Q_s, Q_l, and Q_f with transitions: Q_s to Q_l ε,ε→S$, Q_l to Q_l the grammar rules, and Q_l to Q_f ε,$→ε

PDA ⇒ CFG

- **Lemma: Every PDA has an equivalent normalised PDA**
  Proof: **A normalised PDA has every transition do exactly one of push and pop and has a single accepting state and can only enter the accepting state with an empty stack.** The necessary transformations are trivial
- **Theorem: Every (normalised) PDA M recognises a CFG**
  Proof:
  Define $A_{p,q}$:= non-terminal which generates all strings that move M from state p to state q with the same stack content at p and q.
  The following describe every element of the rules R of a grammar $G=(\{A_{i,j}\}, \Sigma, R, A_{Qs,Qf})$ such that L(M) = L(G):
  1. $\forall p,q,r,s \in Q.\ \forall u \in \Gamma.\ \forall a,b \in \Sigma \cup \{\varepsilon\}.\ (\delta(p, a, \varepsilon) \ni (r, u) \wedge \delta(s, b, u) \ni (q, \varepsilon)) \Rightarrow A_{p,q} \rightarrow aA_{r,s}b \in R$ [recursive case; reading a at p causes us to enter r and push u; reaching b at s causes us to enter q and pop u]
  2. $\forall p,q,r \in Q \Rightarrow A_{p,q} \rightarrow A_{p,r}A_{r,q} \in R$ [transitivity]
  3. $\forall p \in Q.\ A_{pp} \rightarrow \varepsilon \in R$ [base case]

Pumping lemma for context-free languages

- Intuition: If an infinite language is context free, then for each sufficiently long word every parse tree has a root to leaf path that repeats a non-terminal (as the RHS of production rules are only so long and so by pigeon hole we must repeat something for a tall enough parse tree), the portion of the word arising from this repetition can be repeated arbitrarily many times (and can be removed) to produce words also in the language
- **Pumping lemma: If L is a context-free language, then $\exists m \in \mathbb{Z}$: $\forall w \in \{w \in L: |w| \geq m\}$. $\exists u,v,x,y,z \in \Sigma^*$: $w = \underline{uv}xyz \wedge |\underline{v}y| \geq 1 \wedge |\underline{v}xy| \leq m \wedge \forall i \in \mathbb{N}_{\geq 0}. \underline{uv}^i x y^i z \in L$**
  - Proof sketch: Let S be the start variable and A be the variable that is repeated. $S \Rightarrow^* uAz$; $A \Rightarrow^* vAy$; $A \Rightarrow^* x$; and so $S \Rightarrow^* uv^ixy^iz$ for all i
- Contrapositive: **If for an arbitrarily large pumping length m, $\exists w \in \{w \in L: |w| \geq m\}$: $\forall u,v,x,y,z \in \Sigma^*$: $(w = \underline{uv}xyz$ and $|\underline{v}y| \geq 1$ and $|\underline{v}xy| \leq m) \Rightarrow \exists i \in \mathbb{N}_{\geq 0}$: $\underline{uv}^i x y^i z \notin L$, then L is not context-free**
  - Requires case analysis as we have absolutely no limits on how much of the string is in u and thus do not know how the string is split across v and y (but we do know that the x inbetween cannot be large) and we have to pump both v and y the same amount

Pumping lemma for context-free languages: An example

Proposition: $L = \{a^n b^n c^n\}$ is not context-free

Proof:

Assume for the sake of contradiction that L is context-free. Let m be the pumping length of L.

Consider $w = a^m b^m c^m \in L$.

Pick arbitrary u, v, x, y, z such that $w = uvxyz$ and $|vy| \geq 1$ and $|vxy| \leq m$.

Case $v, y \in \beta^* \gamma^*$ where $(\beta, \gamma) \in \{(a, a), (a, b), (b, b), (b, c), (c,c)\}$: Then, for all $i \neq 1$, the number of occurrences of β and γ are different in $uv^i xy^i z$ than in $uvxyz$ but the number of occurrences of $\alpha=\{a,b,c\}\backslash\{\beta, \gamma\}$ does not change and so $uv^i xy^i z \notin L$.

For this problem this single case is exhaustive (as * allows us to take 0 we have covered all cases except v containing an a and y containing a c which is not possible as $|vxy| \leq m$ and there are m bs inbetween the as and the cs).

Chomsky Normal Form

- **A grammar G=(V, Σ, R, S) is in Chomsky Normal Form (CNF) iff every production rule has one of the following forms:**
  - **$S \rightarrow \varepsilon$**
  - **$A \rightarrow x$; $A \in V$, $x \in \Sigma$**
  - **$A \rightarrow BC$; $A \in V$, $B, C \in V \backslash \{S\}$**
- **Proposition: Every string w is derived from a CNF grammar G in exactly $2|w|-1$ steps**
  Proof sketch: We use $|w|-1$ $A \rightarrow BC$ rules to turn S into n non-terminals, we use $|w|$ $A \rightarrow x$ rules to turn these into terminals, $|w|-1+|w| = 2|w|-1$
- **Theorem: Every context-free grammar can be written as a CNF grammar**
  Proof:
  1. Add a new variable S_0, a rule $S_0 \rightarrow S$, and make S_0 the start state
  2. Eliminate rules of the form $A \rightarrow \varepsilon$, $A \neq S_0$ by adding an alternative with A replaced with ε to the RHS of each rule that has A in its RHS. This can cause more rules of this form to appear which should then also be eliminated
  3. Eliminate rules of the form $A \rightarrow B$ by adding the all the RHS of B to the RHS of A
  4. Eliminate rules of the from $A \rightarrow \omega$, $|\omega| \geq 3$ and $\omega \in (V \cup \Sigma)^*$ or the form $A \rightarrow u$, $|u| \geq 2$ and $u \in \Sigma^*$ by introducing more non-terminals and substituting <u>in the required places</u> (substituting too generally can cause us to have to go back to step 3); for example $A \rightarrow u_1u_2u_3$ can be rewritten as $A \rightarrow A_1A_4$, $A_1 \rightarrow A_2A_3$, $A_2 \rightarrow u_1$, $A_3 \rightarrow u_2$, $A_4 \rightarrow u_3$

Chomsky Normal Form: Parsing

- **Proposition: Every word generated by a CNF grammar can be derived using a binary parse tree**
  Proof: Obvious
- The **CYK algorithm** uses bottom-up 2D dynamic programming to check whether a word w is in a given CNF grammar G:
  Let $\mathbf{w = y_1 \ldots y_n}$
  We will store a triangular matrix M such that **M[i, j] := the <u>set of all</u> variables which generate $y_j \ldots y_{i+j-1}$** (the substring of length i starting at j)
  LHS(a) = {J ∈ V: J → a ∈ R}
  LHS(P, Q) = $\bigcup_{(x,y) \in P \times Q}$ LHS(xy)

```
M[1, i] = LHS(w[i]) for i ∈ [n]
for l=2, …, n // length
    for s=1, …, n-(l-1) // start
        M[l, s] = ∅ // initialise
        // We consider all possible breakpoints b for binary partitions of w
        for b=1, …, l-1
            M[l, s] ∪= LHS(M[b, s], M[l-b, s+b])
return bool(S ∈ M[n, 1])
```

  - $O(n^3)$ — parsing in polynomial time is pretty good

Closure properties for CFLs

- **<u>Not</u> closed under intersection** as cannot simulate two stacks with one stack
  - E.g. $P = \{a^i b^j c^j\}$, $Q = \{a^i b^i c^j\}$ are CFLs but $P \cap Q = \{a^i b^i c^i\}$ which is a canonical non-context-free language
  - Intersecting a CFL with a regular language does give a CFL as we only need one stack then
- **Closed under union** as can merge the grammars (after making the non-terminals disjoint) and add a rule $S \rightarrow S_1 | S_2$
- **Closed under concatenation** as can merge the grammars (after making the non-terminals disjoint) and add a rule $S \rightarrow S_1 S_2$
- **Closed under kleene star** as can add a rule $S \rightarrow \varepsilon | S_1 S$
- **<u>Not</u> closed under complementation** as aren't closed under intersection and are closed under union and $P \cap Q \equiv \neg(\neg P \cup \neg Q)$
- Closed under reversal as can rewrite each rule $A \rightarrow \alpha$ as $A \rightarrow \alpha^{rev}$ — as every regular language is context free, this also proves this for regular languages

Turing machines

Terminology

- **$M=(Q, \Sigma, \Gamma, \delta, q_0, q_a, q_r)$ is a Turing machine with finite set of states Q, input alphabet Σ, tape alphabet $\Gamma \supseteq \Sigma \cup \{\sqcup\}$ where $\sqcup$ denotes a blank cell, $\delta: Q \times \Gamma \mapsto Q \times \Gamma \times \{L, R\}$, initial state q_0, accepting state q_a, and rejecting state q_r**
  - **A transition marked $a \rightarrow b,c$ means read a in the current cell, overwrite the current cell with b, and move head $c \in \{L, R\}$**
- Unlike DFAs and PDAs which consume their input one symbol at a time, the input string is placed on the tape at the start and may be (but is not necessarily) modified as the machine runs
  - By convention, the start of the tape (leftmost cell) is marked $\vdash$ with the actual data starting after this

Terminology: Runs

- **A configuration X = (u, q, v) is a complete snapshot of the state of a TM at a point in time where u is the string on the tape up to (but not including) the current cell, q is the current state, and v is the string on the tape from (and including) the current cell —** for this purpose we can ignore the infinite tail of ␣ to the right
  - **The initial configuration is ($\vdash$, q_0, w) where w is the input word**
- **A run is a sequence of configurations that are obtained from each other by single steps**
- Unlike DFAs and PDAs where the run ends once the input has been consumed, **a TM run ends iff it enters q_a or q_r**:
  - A run is an accepting run iff it ends in a configuration with $q=q_a$
  - A run is a rejecting run iff it ends in a configuration with $q=q_r$
  - **A TM is a decider iff it halts on every input**
- **L is Turing-recognizable (recursively-enumerable) iff there exists a TM M such that L(M) = M — words not in L can either reject or not halt but must accept iff a word is in L**
- **L is Turing-decidable (recursive) iff there exists a decider M such that L(M) = M — words not in L must reject and words in L must accept**

Generality

- A PDA with n-stacks where n ≥ 2 is actually a Turing machine with the first stack representing the u component of the configuration (with top the right-most symbol) and the second stack the v component (with top the left-most symbol)
- We can simulate an n-tape Turing machine with our typical 1-tape turing machine by storing an n-tuple in each cell and marking the corresponding element with a hat iff the corresponding head is over that cell
- We can simulate a tape that's also infinitely long to the left by cutting it at an arbitrary point to produce two tapes that are only infinite to the right and using the n-tape construction
- An enumeration machine is a TM which instead of accept/reject produces a stream of output. Has an initially blank write-only output tape as well as the standard read-write work tape (which now is also initially blank). One state is designated as the enumeration state, when entered the contents of the enumeration tape is outputted, the output tape is blanked, and the head of the output tape returns to the start
    - L is recursively-enumerable (turing-recognizable) ⇔ there exists an EM M such that L = {w: M has a run that outputs w}
    - Every TM can simulate an EM and vice versa
- A universal Turing-machine takes as input <M, w> *(this is written to the tape as some encoding of M followed by a # followed by w)* and simulates running M on w

Decidability

- Proposition: **L is Turing-decidable iff both L and ¬L are Turing-recognizable**
  Proof: ⇒: Every decider is a recognizer, a decider for L can be converted into a decider for ¬L by swapping q_a and q_r
  ⇐: Let P, Q be recognisers for L, ¬L respectively. We can construct a decider for L that runs P and Q in parallel accepting iff P accepts and rejecting iff Q accepts
  Corollary: L is undecidable iff L is not turing-recognizable or ¬L is not turing-recognizable
- Proposition: L is recursive iff L is recursively-enumerable and $L \leq_m \neg L$
  Proof: Let σ be the reduction from L to ¬L. Deduce that as L and ¬L have the same alphabet, σ is also a reduction from ¬L to L. Finally, L is recursively-enumerable and $\neg L \leq_m L$ ⇒ ¬L is also recursively-enumerable by properties of reductions ⇒ L is recursive by our previous proposition

Closure properties

- **Turing-recognizable and Turing-decidable are both:**
  - Closed under reversal — there clearly exists a decider that halts with w^{rev} on the tape, run this first then run our existing TM (as reversal is self-inverse $w^{rev} \in L \Leftrightarrow w \in L^{rev}$)
  - **Closed under union/intersection** — run the TMs in parallel and accept iff either/both accepts (and reject iff both/either rejects)
  - **Closed under kleene star** — the input is of finite length so can only be decomposed finitely many different ways, accept iff there is a decomposition for which every component is accepted
  - **Closed under concatenation** — same idea as kleene star but simpler as decomposition must be binary
- **Turing-recognizable (recursively-enumerable) are not closed under complementation** — e.g. HP is RE but not decidable and so ¬HP cannot be RE
- **Turing-decidable (recursive) are closed under complementation** — L is decidable ⇔ L and ¬L are RE ⇔ ¬L and ¬¬L are RE ⇔ ¬L is decidable

Undecidability

- **We call a problem undecidable iff it is not Turing-decidable**
- **HP** (halting problem) **:=** {<M, x>: M halts on x} = **{<M, x>: M accepts x or M rejects x}**
- **MP** (membership problem) **:=** {<M, x>: x ∈ L(M)} = **{<M, x>: M accepts x}**
- **Theorem: HP is Turing-recognizable but undecidable**
  Proof: We can obviously tell when M has halted to know when to accept (recognizability) but we will show that there is no way of knowing when to give up on waiting to know when to reject (decidability).
  We will use a diagonalisation argument. Assume for the sake of contradiction HP is decidable. Then we can construct a (countably infinite) table of the behaviour (either halt or loop for decider accepts or rejects respectively) of every TM on every x.
  We can construct a TM M' that does the opposite of what the leading diagonal says for each x. By diagonalization [for all i M' cannot be the machine in row i as it has the opposite to row i in column i], the behaviour of M' is not in our table even though it contained the behaviour of every TM?!
- *As there are uncountably many languages and only countably infinitely many Turing machines there are uncountably many languages that are not even recursively-enumerable*

(Un)decidability: Reductions

- **Let $A \subseteq \Sigma^*$ and $B \subseteq \Delta^*$. Then, $\sigma: \Sigma^* \mapsto \Delta^*$ is a (mapping) reduction from A to B iff $\forall x \in \Sigma^*. x \in a \Leftrightarrow \sigma(x) \in B$ and <u>σ is computable</u>** (there exists a decider that for an input x halts with σ(x) on the tape). **We denote there exists a reduction from A to B as $A \leq_m B$**
  - <u>Both conditions should be at least handwaved</u> whenever we are constructing a reduction
  - This is a generalisation of polynomial time reductions in CS260, having removed the specific bound on the amount of time taken by the reduction we must explicitly specify that it does always terminate (computability)
- Theorem: **$A \leq_m B$ and B is decidable ⇒ A is decidable**
  Proof: Obvious
  Corollary: **$A \leq_m B$ and A is undecidable ⇒ B is undecidable**
- Theorem: **$A \leq_m B$ and B is recursively-enumerable ⇒ A is recursively-enumerable**
  Proof: Obvious
  Corollary: **$A \leq_m B$ and A is not recursively-enumerable ⇒ B is not recursively-enumerable**
- Lemma: $HP \leq_m MP$
  Proof: Consider $\sigma(\langle M, x\rangle) = \langle M_x', x\rangle$ where **M_x' = the turing machine that simulates running M on x and accepts iff M halts (note we ignore the input we are given and instead only consider the fixed x we are encoding into the machine)**
  $\langle M, x\rangle \in HP \Leftrightarrow$ M halts on x $\Leftrightarrow M_x'(x)$ accepts $\Leftrightarrow \langle M_x', x\rangle \in MP$
  Corollary: **MP is undecidable**

Undecidability: Rice's Theorem

- *Rice's Theorem: "All non-trivial semantic properties of programs are undecidable". Roughly, L = {<M>: P(M)} is undecidable if P(M) is a statement about the behaviour of M and there exists at least one M for which it is true and at least one M for which it is false*
  Proof:
  It suffices to show that $HP \leq_m L$
  Consider $\sigma(\langle M_1, x\rangle) = \langle M_2, x\rangle$ *where* M_2 *= the TM that simulates running* M_1 *on x until* M_1 *halts then carries out a process that satisfies P(M). As the process that satisfies P(M) manages to occur iff* M_1 *halts on x,* $\langle M_1, x\rangle \in HP \Leftrightarrow M_1$ *halts on x* $\Leftrightarrow \langle M_2\rangle \in L$
- *We don't have the tools in this module to rigorously define Rice's theorem but we can use this proof structure for direct proofs*

Undecidability: Problems to reduce from

- **HP is undecidable**
- **MP is undecidable**
- **Theorem: ε-ACCEPTANCE = {<M>: ε ∈ L(M)} is undecidable**
  Proof: HP $\leq_m$ ε-ACCEPTANCE by σ(<M, x>) = <M_x'>
- **Theorem: ∃-ACCEPTANCE = {<M>: L(M) ≠ ∅} is undecidable**
  Proof: Same proof
  Corollary: As decidable languages are closed under complementation, **∅-ACCEPTANCE = {<M>: L(M) = ∅} is undecidable** as if it was decidable than ∃-ACCEPTANCE would be decidable
- **Theorem: ∀-ACCEPTANCE = {<M>: L(M) = Σ*} is undecidable**
  Proof: Same proof