Models of Computation Notes

William Guest

January 2023

1 Regular Languages

1.1 Deterministic Finite Automata

Def 1. *A **deterministic finite automata (DFA)** is a 5-tuple $(Q, \Sigma, \delta, q_0, F)$ where*

- *Q is a finite set called the states*
- *Σ is a finite set called the alphabet*
- *$\delta : Q \times \Sigma \to Q$ is the transition function*
- *q_0 is the start state*
- *$F \subseteq Q$ is the set of accepting states*

We write $q \xrightarrow{a} q'$ to mean $\delta(q, a) = q'$.

Let $M = (Q, \Sigma, \delta, q_0, F)$ be a DFA. For $w = a_1 a_2 \dots a_n$ a string over σ, we write $q \xrightarrow{w} *q'$ if there exists states $q_1, q_2, \dots, q_n = q'$ such that

$$q \xrightarrow{a_1} q_1 \xrightarrow{a_2} \dots \xrightarrow{a_n} q_n = q'$$

Note that

$$q \xrightarrow{\epsilon}{}^* q' \text{ iff } q = q'$$

and

$$q \xrightarrow{a}{}^* q' \text{ iff } q \xrightarrow{a} q'$$

$L(M)$, the language **recognised** by the DFA M, consists of all strings w over Σ satisfying $q_o \xrightarrow{w}{}^* q$ where q is an accepting state.

Def 2. *A language is called **regular** if some DFA recognises it.*

1.2 Non-Deterministic Finite Automata

Def 3. *A nondeterministic finite automata (NFA) is a 5-tuple* $(Q, \Sigma, \delta, q_0, F)$ *where*

- Q *is a finite set called the states*
- Σ *is a finite set called the alphabet*
- $\delta : Q \times (\Sigma \cap \{\epsilon\}) \to \mathcal{P}(Q)$ *is the transition function*
- q_0 *is the start state*
- $F \subseteq Q$ *is the set of accepting states*

We write $q \xrightarrow{a} q'$ to mean $\delta(q, a) = q'$.

Let $N = (Q, \Sigma, \delta, q_0, F)$ be an NFA. Let $q \overset{w}{\Longrightarrow} q'$ be defined by:

- $q \overset{\epsilon}{\Rightarrow} q'$ iff $q = q'$ or there is a sequence $q \xrightarrow{\epsilon} \ldots \xrightarrow{\epsilon} q'$ of one or more ϵ-transitions in N from q to q'.
- For $w = a_1 \ldots a_{n+1}$ where each $a_i \in \sigma$, $q \overset{w}{\Longrightarrow} q'$ iff there are $q_1, q_1', \ldots, q_{n+1}, q_{n+1}'$ such that

$$q \overset{\epsilon}{\Rightarrow} q_1 \xrightarrow{a_1} q_1' \overset{\epsilon}{\Rightarrow} q_2 \xrightarrow{a_2} \ldots q_n' \overset{\epsilon}{\Rightarrow} q_{n+1} \xrightarrow{a_{n+1}} q_{n+1}' \overset{\epsilon}{\Rightarrow} q'$$

$L(N)$, the language recognised by N, consists of all strings w over Σ satisfying $q_0 \overset{w}{\Longrightarrow} q$, where q is an accepting state.

Theorem 1. *Every NFA has an equivalent DFA.*

This is proved by subset construction, each state in the DFA is a member of the powerset of states in the NFA.

If we fix an NFA $N = (Q_N, \Sigma_N, \delta_N, q_N, F_N)$, then we construct a DFA $\mathcal{P}(N) = (Q_{\mathcal{P}N}, \Sigma_{\mathcal{P}N}, \delta_{\mathcal{P}N}, q_{\mathcal{P}N}, F_{\mathcal{P}N})$ by

- $Q_{\mathcal{P}N} \overset{\text{def}}{=} \{S \mid S \subseteq Q_N\}$
- $\Sigma_{\mathcal{P}N} \overset{\text{def}}{=} \Sigma_N$
- $S \xrightarrow{a} S'$ in $\mathcal{P}N$ iff $S' = \{q' \mid \exists q \in S.(q \overset{a}{\Rightarrow} q' \text{ in } N)\}$
- $q_{\mathcal{P}N} \overset{\text{def}}{=} \{q \mid q_N \overset{\epsilon}{\Rightarrow} q\}$
- $F_{\mathcal{P}N)} \overset{\text{def}}{=} \{S \in Q_{\mathcal{P}N} \mid F_N \cap S \neq \emptyset\}$

1.3 Regular Operations

Let A and B be languages. Define

- Union - $A \cup B = \{x | x \in A \text{ or } x \in B\}$
- Concatenation - $A \cdot B = \{xy \mid x \in A \text{ and } y \in B\}$
- Star - $A^* = \{x_1 x_2 \dots x^k \mid k \geq 0 \text{ and each } x_i in A\}$

Theorem 2. *Regular languages are closed under union, concatenation and star. (If A_1 and A_2 are regular languages, then so are $A_1 \cup A_2$, $A_1 \cdot A_2$, and A_1^*)*

Take NFAs N_1 and N_2 that recognise $L(N_1)$ and $L(N_2)$ respectively.

We can see that regular languages are closed under union by constructing a new NFA from N_1 and N_2 which adds a new start state outside of each NFA, and joins them to each start state of N_1 and N_2 via ϵ-transitions.

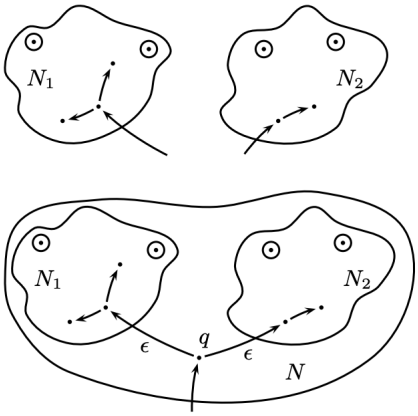

Figure 1: Union Construction

To show that regular languages are closed under concatenation, we connect each accepting state of N_1 to the start state of N_2 via ϵ-transitions. The start state of the new NFA is the start state of N_1, the accepting states are the accepting states of N_2.

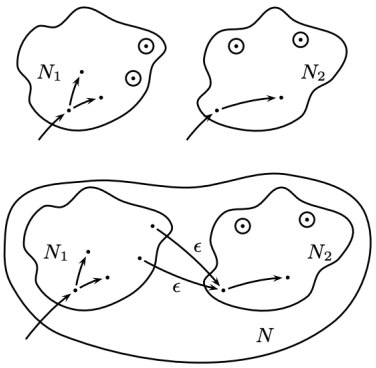

Figure 2: Concatenation Construction

To show that regular languages are closed under star, we add a dummy accept state at the start which connects via an ϵ-transition to the start state of N_1. We also connect each accept state to the start state of N_1 via ϵ-transitions. Note that we can't just make the start state of N_1 an accept state as this would not account for cases where the start state can be reached via transitions that aren't ϵ-transitions from the accept states in N_1

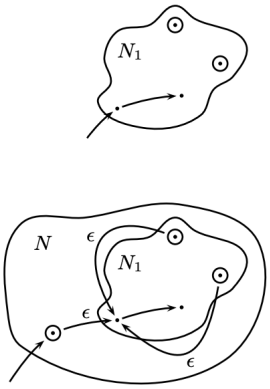

Figure 3: Star Construction

1.4 Regular Expressions

Def 4. *We fix an alphabet Σ. We define regular expression E and the language denoted by E, $L(E)$ recursively:*

- *The constants ϵ and $\emptyset$ are regular expressions, with $L(\epsilon) \stackrel{def}{=} \{\epsilon\}$ and $L(\emptyset) \stackrel{def}{=} \{\emptyset\}$.*

- *For $a \in \Sigma$, a is a regular expression, with $L(a) \stackrel{def}{=} \{a\}$.*
- *If E and F are regular expressions, then so are $(E + F)$, $(E \cdot F)$, (E^*) with*

$$L((E + F)) \stackrel{def}{=} L(E) \cup L(F) \qquad \text{Union}$$
$$L((E \cdot F)) \stackrel{def}{=} L(E) \cdot L(F) \qquad \text{Concatenation}$$
$$L((E^*)) \stackrel{def}{=} (L(E))^* \qquad \text{Star}$$

Note $+$ is sometimes written as $\cup$ or $|$, and $(E \cdot F)$ is sometimes written as (EF). Note also that the order of precedence is star, concatenation, union, so $(0 + 1)01^*)$ is formally $((((0 + 1) \cdot 0) \cdot (1^*)) \cdot 0)$.

We have some equivalences for regular expressions:

$$(E + F) + G \equiv E + (F + G) \qquad \text{Associativity}$$
$$(EF)G \equiv E(FG) \qquad \text{Associativity}$$
$$E + F \equiv F + E \qquad \text{Commutativity}$$
$$E\emptyset \equiv \emptyset$$
$$\emptyset^* \equiv \epsilon$$
$$E + \emptyset \equiv E \equiv \emptyset + E$$
$$E\epsilon \equiv E \equiv \epsilon E$$

Theorem 3 (Kleene's Theorem)**.** *Let $L \subseteq \Sigma$. The following are equivalent:*

- *L is regular*
- *L is denoted by some regular expression E, i.e. $L = L(E)$*

Proving that any regular expression has an NFA is simple, there are NFAs that recognise ϵ, $\emptyset$ and a, for all $a \in \Sigma$. Since languages are closed under union, concatenation and star, for regular expressions E and F, with NFAs N_E and N_F respectively, there are also NFAs that recognise $L(N_E) \cup L(N_F), L(N_E) \cdot L(N_F)$, and $(L(N_E))^*$, which are equivalent to $E + F$, $E \cdot F$, E^* respectively.

To prove that a given NFA has a regular expression, we construct a regular expression recursively. Given NFA $M = (Q, \Sigma, \delta, q, F)$, for $X \subseteq Q$ and $q, q' \in Q$, we construct on induction on the size of X, the regular expression

$$E^X_{q,q'}$$

which denotes the set of strings such that there is a path from q to q' labelled w, i.e. $q \stackrel{w}{\Longrightarrow} q'$ such that all **intermediate** states along that path lie in X. Hence

to find the regular expression for M is

$$L(M) = \sum_{f \in F} E^{Q}_{q_0, f}$$

To construct such a regular expression, we construct recursively on the size of X The base case is when $X = \emptyset$. For $q = q'$, take

$$E^{\emptyset}_{q,q'} \stackrel{\text{def}}{=} \begin{cases} a_1 + \ldots + a_k & \text{if } k \geq 0 \\ \emptyset & \text{otherwise} \end{cases}$$

and for $q = q'$, take

$$E^{\emptyset}_{q,q'} \stackrel{\text{def}}{=} \begin{cases} a_1 + \ldots + a_k + \epsilon & \text{if } k \geq 0 \\ \epsilon, & \text{otherwise} \end{cases}$$

In this way, we only can go from one state to another if there is a direct transition, or if the state we are on currently is the state that we are trying to get to, as there are no intermediate states.

For the inductive step, we choose some $r \in X$ called the *separating state*. Any path from q to q' must either:

- Never visit r
- Visit r, loop from r to r some non-negative number of times, and then visit p'

Hence for non-empty X, we take

$$E^{X}_{q,q'} \stackrel{\text{def}}{=} E^{X-\{r\}}_{q,q'} + E^{X-\{r\}}_{q,r} \cdot (E^{X-\{r\}}_{r,r})^* \cdot E^{X-\{r\}}_{r,q'}$$

To simplify this process, we choose a separating state that splits the automaton as much as possible.

1.5 Kozen's Axioms

Kozen's axioms define a set of rules and equivalences that define equivalences between regular expressions. The axioms are sound (meaning that all rules are valid) and complete (meaning that any equivalence between regular expression can be derived from these axioms).

1. $E + (F + G) \equiv (E + F) + G$
2. $E + F \equiv F + E$
3. $E + \emptyset \equiv E$
4. $E + E \equiv E$

5. $(EF)G \equiv E(FG)$
6. $\epsilon E \equiv E\epsilon \equiv E$
7. $E(F+G) \equiv EF + EG$
8. $(E+F)G \equiv EG + FG$
9. $\emptyset E \equiv E\emptyset \equiv \emptyset$
10. $\epsilon + EE^* \equiv E^*$
11. $\epsilon + E^*E \equiv E^*$
12. $F + EG \leq G \implies E^*F \leq G$
13. $F + GE \leq G \implies FE^* \leq G$

where $E \leq F$ means $E + F \equiv F$, which is true iff $L(E) \subseteq L(F)$.

1.6 Characterisations of Regular Languages

1.6.1 The Pumping Lemma

The pumping lemma is a way of proving that languages are not regular. It is *not* a way of proving that languages are regular.

Theorem 4. *If A is a regular language, then there is some number p, the pumping length, such that if $s \in A$ is of length at least p, then we can divide s into three pieces $s = xyz$ such that*

- *for each $i \geq 0$, $xy^iz \in A$*
- $|y| > 0$
- $|xy| \leq p$

Intuitively, this is the case because a regular language corresponds to an NFA with a finite number of states, so taking the pumping length to be the number of states, we must see that any word accepted by a regular language longer than p, must loop.

In practice we assume that a language is regular, choose some word in the language for an arbitrary pumping length and use this to derive a word that is not in the language, which is a contradiction.

Theorem 5. *If for all $p \geq 1$ there exists some $s \in L$ with $|s| \geq p$ such that for all $x, y, z \in \Sigma^*$ with $s = xyz$ and $|xy| \leq p$ there is an $i \geq 0$ such that $xy^iz \notin L$, then L is not regular.*

1.7 Myhill-Nerode Theorem

The Myhill-Nerode theorem provides an exact characterisation of regular languages, we can use it to determine when a language is regular or not.

We first define an equivalence relation. Let $x, y \in \Sigma^*$ be strings and let $L \in \Sigma^*$. We say that x and y are L-indistinguishable ($x \equiv_L y$) if for every $z \in \Sigma^*$,

$$xz \in L \text{ iff } yz \in L$$

We define the index of a regular language to be the number of equivalence classes of $\equiv_L$.

Theorem 6 (Myhill-Nerode Theorem)**.** *A language is regular iff $\equiv_L$ has finite index. Moreover, the index is the number of states of the smallest DFA that recognises L.*

This comes from the fact that if L is recognised by a DFA with k states then there are at most k equivalence classes, and that if L has a finite index k, then there it is recognised by a DFA with k states.

2 Context-Free Languages

2.1 Context-Free Grammars

Def 5. *A **Context-Free Grammar (CFG)** is a 4-tuple $G = (V, \Sigma, \mathcal{R}, S)$ where*

- *V is a finite set of **variables***
- *Σ is finite set of **terminals** (disjoint from V)*
- *$\mathcal{R}$ is a finite set of **rules** or productions.*
  - *A rule is an element of $V \times (V \cup \Sigma)^*$, written $A \to w$*
  - *We write $A \to \alpha \mid \beta \mid \gamma$ to mean $A \to \alpha$, $A \to \beta$, $A \to \gamma$*
- *$S \in V$ is the start symbol.*

Given a CFG G, we can define a binary relation, $\Rightarrow$ over $(V \cup \Sigma)^*$ as

$$uAv \Rightarrow uwv$$

for each $u, v \in (V \cup \Sigma)^*$, for each $A \to w$ in $\mathcal{R}$.

We define $\Rightarrow^*$ as the reflexive and transitive closure of $\Rightarrow$. Hence a language generated by some CFG G can be written as

$$L(G) = \{w \in \Sigma^* \mid S \Rightarrow^* w\}$$

A CFG is right-linear if every rule is of the form $A \to wT$ or $A \to w$, where R, T are variables and w is a terminal.

Theorem 7. *A language is regular iff it is generated by a right-linear CFG.*

We prove this by showing that all right-linear CFGs are equivalent to a *strongly right-linear* CFG, meaning all rules are of the form $A \to T$, $A \to wT$ or $A \to \epsilon$, and that each strongly right-linear CFG is equivalent to a DFA/NFA.

2.2 Parse Trees

A CFG derivation can determine a parse tree. Given a CFG $G = (V, \Sigma, \mathcal{R}, S)$, a parse tree is a tree satisfying

- The root of the the tree is S
- Each vertex of the parse tree is a variable, a terminal or ϵ
- If $A \to B_1, \ldots, B_k$ is a rule, then $B_1, \ldots, B_k$ are the children of A
- Leaf nodes are variables and interior nodes are terminals

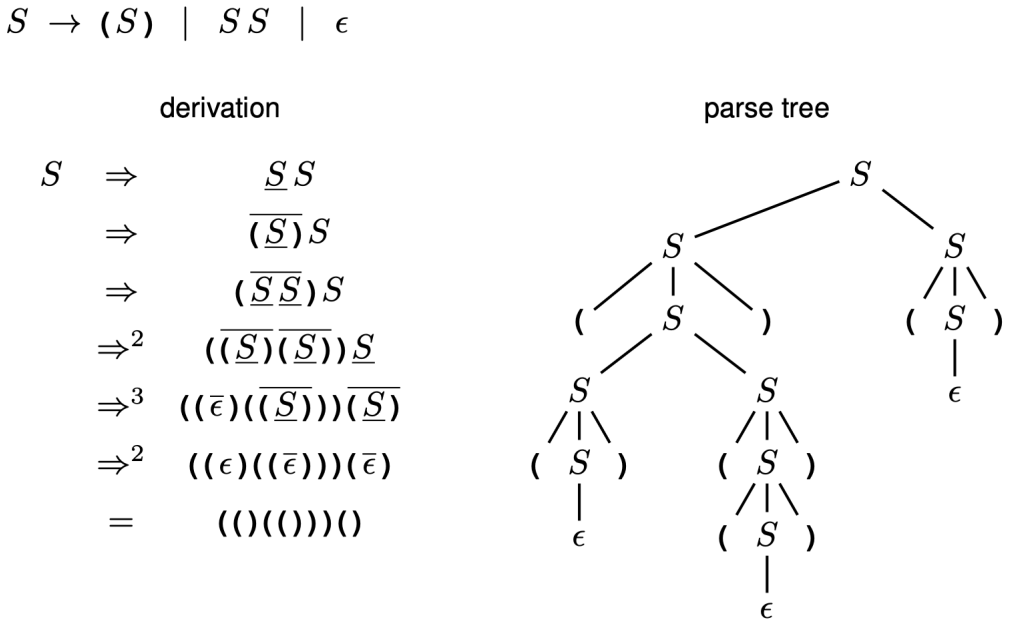

Figure 4: A CFG derivation and a parse tree

2.3 Ambiguity

Two derivations are *essentially different* if they determine different parse trees. In some CFGs, a string may have different derivations, in these cases the CFG

is **ambiguous**. A **leftmost derivation** is a derivation in which the leftmost variable is chosen for replacement. A CFG G is ambiguous if there is some word in the language of G that has two or more different leftmost derivations. Languages are **inherently ambiguous** if they can only be generated by ambiguous grammars.

2.4 Pumping Lemma

Theorem 8. *If L is a context-free language, then there is a number p, the **pumping length** such that if $w \in L$ is of length at least p, then w can be divided into five pieces $w = uxyzv$ such that*

- *For each $i \geq 0$, $ux^iyz^iv \in L$*
- *$|xz| > 0$*
- *$|xyz| \leq p$*

2.5 Chomsky Normal Form

A CFG is in Chomsky Normal Form if every rule is of the form $A \to BC \mid a$, where a is any terminal and A, B, C are variables with B and C not the start variable. We also permit $S \to \epsilon$

Theorem 9. *Any CFL can be generated by a CFG in Chomsky Normal Form*

2.6 Non-Deterministic Pushdown Automata

Just as how regular languages are recognised by DFAs and NFAs, context-free langugages are recognised by NDPAs.

Def 6. *A **Non-Deterministic Pushdown Automata (NDPA)** is a 7-tuple $(Q, \Sigma, \Gamma, \delta, q_0, \bot, F)$ such that $Q, \Sigma, \Gamma, \delta, F$ are all finite sets and*

- *Q is the set of states*
- *Σ is the input alphabet*
- *Γ is the stack alphabet*
- *$\delta : Q \times (\Sigma \cup \{\epsilon\}) \times \Gamma \to \mathcal{P}(Q \times \Gamma^*)$ is a transition function*
- *$q_0 \in Q$ is the start state*
- *$\bot \in \Gamma$ is the inital start symbol*
- *$F \subseteq Q$ is the set of accept states*

A NPDA M has *configurations* describing the current state of the NPDA, the portion of the input yet unread and the current stack contents. The start configuration is

$$(q_0, w, \bot)$$

meaning that M always starts in the start state with its input head scanning the leftmost input symbol and the stack containing $\bot$.

We can then define a next-configuration relation that describes how M moves from one configuration to another in one step.

- If $(q, \gamma) \in \delta(p, a, A)$ then for any $v \in \Sigma^*$ and $\beta \in \Gamma^*$,

  $$(p, av, A\beta) \to (q, v, \gamma\beta)$$

  (We consume the input symbol a, we pop A and replace it with β)

- If $(q, \gamma) \in \delta(p, \epsilon, A)$ then for any $v \in \Sigma^*$ and $\beta \in \Gamma^*$,

  $$(p, v, A\beta) \to (q, v, \gamma\beta)$$

  (We don't consume any input)

Similarly to DFAs and NFAs, we can define the reflexive, transitive closure of $\to$,

$$C \xrightarrow{0} D \iff C = D$$
$$C \xrightarrow{n+1} D \iff \exists E s.t. C \xrightarrow{n} E \wedge E \to D$$

and define $C \xrightarrow{*} D$ just when there is some $n \geq 0$ such that $C \xrightarrow{n} D$.

We say that M **accepts an input x by final state** if there is some $q \in F$ and some $\gamma \in \Gamma^*$ such that $(q_0, x, \bot) \xrightarrow{*} (q, \epsilon, \gamma)$. The **language** of M, $L(M)$ is the set of strings accepted by M. We also have another accepting convention, M **accepts an input x by empty stack** if for some $q \in Q$, $(q) \xrightarrow{\epsilon} \epsilon$, hence meaning the set F is irrelevant. The two conventions are equivalent.

Theorem 10. *A language is context free iff there is some NPDA that recognises it.*

We can show that any given CFG has an NDPA that simulates it. Given G, we can construct P_G by

1. Place the start variable on the stack.
2. Repeat forever: Pop top-of-stack x, there are three cases:
   (a) x is a variable A. Nondeterministically choose a rule for A and replace A by the rule's rhs.
   (b) x is a terminal a. Read the next input symbol and compare it with a. If they do not match, then reject this branch.

(c) $x = \perp$ We enter the accept state.

To show that any given NPDA has an equivalent CFG, we first show that any NPDA has an equivalent NPDA with one state. Given an NPDA $M = (Q, \Sigma, \Gamma, \delta, s, \perp, \{t\}$ we constuct an NPDA $M' = (*, \Sigma, \Gamma', \delta', *, \langle s \ \perp \ t\rangle, \perp, \emptyset$, where

$$\Gamma' = Q \times \Gamma \times Q$$

and for each transition

$$(q_0, B_1, \ldots, B_k) \in \delta(p, c, A)$$

where $c \in \Sigma \cup \{\epsilon\}$, we include in δ',

$$*, \langle q_0 \ B_1 \ q_1\rangle\langle q_1 \ B_2 \ q_2\rangle \ldots \langle q_{k-1} \ B_k \ q_k\rangle) \in \delta'(*, c, \langle p \ A \ q_k)$$

for all choices of $q_1, \ldots, q_k$. Intitively, M' simulates M guessing non-deterministically what state M will be in at certain points in the future, saving those guesses on the stack, and verifying later that those guesses were correct.

Hence given an NPDA M with only one state, $(\{q\}, \Sigma, \Gamma, \delta, q, \perp, \emptyset)$ that accepts by empty stack, we define a CFG $G_M = (\Gamma, \Sigma, P, \perp)$ where P contains a rule

$$A \to cB_1 \ldots B_k$$

for every transition,$(q, B_1 \ldots B_k) \in \delta(q, c, A)$ where $c \in \Sigma \cup \{\epsilon\}$. Hence $L(M) = L(G_M)$.

2.7 Regular Operations

Theorem 11. *Context free languages are closed under union, concatenation and star.*

We can show this, given $G_1 = (\Gamma_1, \Sigma, \mathcal{R}_1, S_1)$ and $G_2 = (\Gamma_2, \Sigma, \mathcal{R}_2, S_2)$ with $\Gamma_1 \cap \Gamma_2 = \emptyset$, we can define the CFGs using the new start variable:

$$\begin{aligned} G_{\text{union}} &= (\Gamma_1 \cup \Gamma_2 \cup \{S\}, \Sigma, \mathcal{R}_1 \cup \mathcal{R}_2 \cup \{S \to S_1, S \to S_2\}, S) \\ G_{\text{concat}} &= (\Gamma_1 \cup \Gamma_2 \cup \{S\}, \Sigma, \mathcal{R}_1 \cup \mathcal{R}_2 \cup \{S \to S_1 S_2\}, S) \\ G_{\text{star}} &= (\Gamma_1 \cup \Gamma_2 \cup \{S\}, \Sigma, \mathcal{R}_1 \cup \mathcal{R}_2 \cup \{S \to S_1 S, S \to \epsilon\}, S) \end{aligned}$$

Then $L(G_{\text{union}}) = L(G_1) \cup L(G_2)$, $L(G_{\text{concat}}) = L(G_1) \cdot L(G_2)$, and $L(G_{\text{star}}) = L(G)^*$

Context-free languages are **not** closed under intersection.

3 Recursively Enumerable Languages

3.1 Turing Machines

Def 7. *A Turing Machine is a 9-tuple*

$$(Q, \Sigma, \Gamma, \vdash, \llcorner, \delta, q_0, q_{acc}, q_{rej})$$

where

- Q *is a finite set (the states)*
- Σ *is a finite set (the input alphabet)*
- Γ *is a finite set (the tape alphabet)*
- $\vdash \in \Gamma - \Sigma$ *is the left endmarker*
- $\llcorner \in \Gamma - \Sigma$ *is the blank symbol*
- $\delta : Q \times \Gamma \to Q \times \Gamma \times \{L, R\}$ *is the transition function*
- q_0, q_{acc}, q_{rej} *are the start, accept and reject states respectively.*

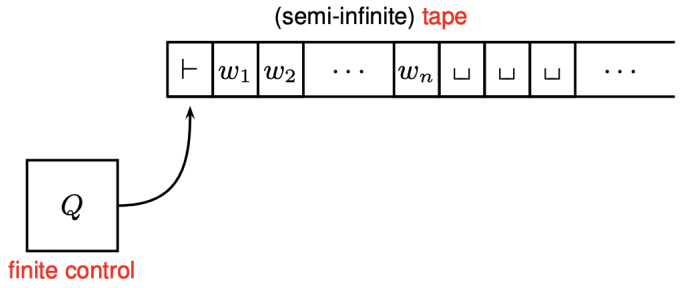

Figure 5: Turing Machine

At the start of the computation, the contents of the Turing Machine are

$$\vdash w_1, \ldots, w_n, \llcorner, \llcorner, \ldots$$

where $w = w_1 \ldots w_n$ is the input string and the tape head is over $\vdash$, the left endmarker. To the right of the input word is an infinite number of $\llcorner$.

Def 8. *A* ***configuration*** *of a Turing Machine is a triple (u, q, v) where the current state is q, the current tape content is u, v and the current head location is over the first symbol of v.*

Hence we can define the next configuration relation $\to$ as

- $(ua, q, bv) \to (u, q', acv)$ if $\delta(q, b) = (q', c, L)$

- $(ua, q, bv) \to (uac, q', v)$ if $\delta(q, b) = (q', c, R)$

We also define the transitive, reflexive closure of this relation $\xrightarrow{*}$ as

- $C \xrightarrow{0} C$ for all configurations C
- $C \xrightarrow{n+1} C''$ iff $C \xrightarrow{n} C'$ and $C' \to C''$

where $C \xrightarrow{*} C'$ means there is some $n \geq 0$ such that $C \xrightarrow{n} C'$..

Intuitively, $\delta(q, a) = (q', b, L)$ means that when in state q, scanning symbol a, then enter state q', write b over the tape cell and move the head left by one cell. Note that the left endmarker is never overwritten and once the machine enters q_{acc} or q_{rej}, it will never leave these states.

- The **start configuration** of a Turing Machine on an input w is $(\epsilon, q_0, \vdash w)$.
- An **accepting configuration**is any configuration with state q_{acc}
- A **rejecting configuration** is any configuration with state q_{rej}.
- M **accepts** input w just when $(\epsilon, q_0, \vdash w) \stackrel{*}{\Rightarrow} C$

3.1.1 Halting

When a TM M starts on an input it may either **accept**, **reject**, or **loop forever**. We say that M halts on an input if it either accepts it or rejects it. M is **total** or a decider if it halts on all possible inputs. We call a language decidable if it is recognised by some total TM, in which case this TM decides the language.

A **recursively enumerable** language is a language such that there is a TM that halts and accepts when presented with any string in that language, *although it may fail to halt when presented with a string not in that language.* A language L is **co-RE** if its complement, $\overline{L}$ is recursively enumerable.

A language is decidable iff it is both RE and co-RE. If we have languages L and $\overline{L}$ that are both RE, then we can simulate their corresponding TMs, accepting or rejecting when either one accepts or rejects. Since both L and $\overline{L}$ are RE, this machine can never halt, so it must be a decider.

We can encode Turing Machines as strings over an alphabet. We write the encoding of the TM M as $\langle M \rangle$.

3.2 Turing Machine Variations

3.2.1 Multi-Tape Turing Machines

$k-tape$ TMs have k semi-infinite tapes each with its own independent read/write head. Initially the input is occupied by the first tape, with the other tapes blank. In each step, the machine reads each of the k symbols under its tape head and moves each tape head accordingly (different tape heads may move in different directions). The transition function is hence of the type

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

Each multi-tape TM is equivalent to a single tape one. This is done by simulating each cell of the different tracks in one cell, using a marker in each subcell to track where the corresponding tape head is for the simulated track. Hence one move of the multi-tape TM is simulated by several moves of the single-tape TM, as the TM needs to traverse the tape, modifying the cells accordingly until it has modified all of the markers, at which point it returns to the left of the tape.

3.2.2 Non-Deterministic Turing Machines

A NDTM is non-deterministic, so that at any point in the computation, it may proceed in several possible directions. If any of these directions accepts the input, then the entire NDTM accepts.

Theorem 12. *A language is recognised by some TM iff it is recognised by some NDTM*

3.2.3 Universal Turing Machines

A **universal TM** is a TM U that can simulate the actions of any TM. For any TM M and input x, U accepts, rejects or loops $\langle M, x\rangle$ iff M accepts, rejects or loops on x respectively.

The universal TM has three tracks:

- The top track holds the transition function δ_M of the input TM M.
- The middle track holds the simulated contents of M's tape
- The bottom track holds the current state of M, alongside the position of M's tape head.

Note that the input alphabet of U and M may be different.

4 Decidability

Recall a language is decidable iff it is recognised by some total TM.

There are several decidable problems:

- DFA Acceptance
  - Given a DFA B and an input w, does B accept w?
  - $A_{DFA} = \{\langle B, w\rangle \mid B \text{ is a DFA that accepts input sting } w\}$
  - Construct a TM that on input $\langle B, w\rangle$ simulates B on input w, accepting and rejecting if B accepts or rejects w
- DFA Emptiness
  - Given a DFA A, is $L(A) = \emptyset$?
  - $E_{DFA} = \{\langle A\rangle \mid B \text{ is a DFA such that } L(A) = \emptyset\}$
  - Construct a TM that on input $\langle A\rangle$ marks the start state of A, and repeatedly marks any state that has a transition coming into it from any marked state
  - If no accept state is marked, *accept*, otherwise *reject*
- DFA Equivalence
  - Given two DFA's are they equivalent?
  - $EQ_{DFA} = \{\langle A, B\rangle \mid A \text{ and } B \text{ are DFAs such that } L(A) = L(B)\}$
  - We use the idea that $P \subseteq Q$ iff $P \cap \overline{Q} = \emptyset$, using T the algorithm for E_{DFA}
  - On input $\langle A, B\rangle$, where A, B are DFAs, run T on $\langle (A \cap \overline{B}) \cup (\overline{A} \cap B)\rangle$, accepting just when T accepts.
- CFG Acceptance
  - Given a CFG G and an string w, can G generate w?
  - $A_{CFG} = \{\langle G, w\rangle \mid G \text{ is a CFG that generates } w\}$
  - We convert G into Chomsky normal form, hence any derivation of w has $2n - 1$ steps where $n = |w|$.
  - We then construct a TM that on input $\langle G, w\rangle$, converts G into CNF, lists all derivations of length $2n - 1$ and accepts just when any of these derivations are w
- CFG Emptiness
  - Given a CFG G is $L(G) = \emptyset$
  - $E_{CFG} = \{\langle G\rangle \mid G \text{ is a FG such that } L(G) = \emptyset\}$

- We construct a TM T, that on input $\langle G \rangle$, where $G = (V, \Sigma, R, S)$, marks all terminal symbols in G and repeatedly marks any variables A, where G contains a rule $A \to U_1 \dots U_k$, where each $U_i \in \Sigma \cup V$ has been marked already.
- If the start symbol is not marked, *accept*, otherwise *reject*.

Note the problem of CFG equivalence is undecidable, we cannot use the techniques used in DFA equivalence as CFGs are not closed under intersection nor complementation.

The acceptance problem for TMs is recursively enumerable.

$$A_{TM} = \{\langle M, w \rangle, M \text{ is a TM that accepts input } w\}$$

We can show that this is true by defining a TM U that accepts A_{TM}: One input $\langle M, w \rangle$ where M is a TM and w is a string,

1. Simulate M on input w
2. If M ever enters its accept state, *accept*: if M ever enters its reject state, *reject*.

Note that U loops whenever M loops, so U is not a decider.

We can further see that A_{TM} is undecidable, meaning that there is no decider for A_{TM}. We can prove this by contradiction, suppose there was a decider for A_{TM}, H. This means that

$$H(\langle M, w \rangle) = \begin{cases} \textit{accept} & \text{if } M \text{ accepts } w \\ \textit{reject} & \text{if } M \text{ does not accept } w \end{cases}$$

We can then construct a new TM D. On input $\langle M \rangle$, where M is a TM,

1. Run H on input $\langle M, \langle M \rangle \rangle$
2. If H accepts, *reject*, if H rejects, *accept*

If we then run D on input $\langle D \rangle$, we find that

$$D(\langle D \rangle) = \begin{cases} \textit{reject} & \text{if } D \text{ accepts } \langle D \rangle \\ \textit{accept} & \text{if } D \text{ does not accept } \langle D \rangle \end{cases}$$

hence meaning that D accepts $\langle D \rangle$ iff D rejects $\langle D \rangle$, which is a contradiction. Hence no machine H exists.

This means that A_{TM} is RE but undecidable.

4.1 Halting Problem

We can use the fact that A_{TM} is undecidable to show that the halting problem is undecidable, i.e., given M and x, does M halt on x?

$$HALT_{TM} = \{\langle M, w\rangle \mid M \text{ is a TM that halts on } w\}$$

We can show that this is true by contradiction. Suppose H is a decider for $HALT_{TM}$. Then we can construct a TM S that on input $\langle M, w\rangle$:

1. Run TM H on input $\langle M, w\rangle$
2. If H rejects, *reject*
3. If H accepts, run M on w until it halts
4. If M has accepted, *accept* otherwise if M has rejected, *reject*

This new TM S is a decider for A_{TM}. However since A_{TM} is undecidable, this is a contradiction, so H cannot be a decider for $HALT_{TM}$.

4.2 Emptiness Problem

The emptiness problem for TMs is also undecidable.

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

We can prove this is true by contradiction. Suppose E_{TM} is decidable by some TM E. Then we construct a TM S that on input $\langle M, x\rangle$:

1. Construct a TM M_x defined as: On input y
   (a) If $y \neq x$ then *reject*
   (b) Run M on x and *accept* if M does
2. Run E on input $\langle M_x\rangle$. If E accepts, *reject*, if E rejects, *accept*

Hence if M accepts x, then $L(M_x) = \{x\}$, meaning S accepts $\langle M, x\rangle$. On the other hand if M does not accept x, then $L(M_x) = \emptyset$, meaning that S rejects $\langle M, x\rangle$. Hence S is a decider for A_{TM}, which is a contraction. Hence no such E exists.

4.3 Equivalence Problem

The equivalence problem for EQ_{TM} is also undecidable.

$$EQ_{TM} = \{\langle M_1, M_2\rangle \mid M_1 \text{ and } M_2 \text{ are TMs with } L(M_1) = L(M_2)\}$$

We can show this by contradiction, suppose that EQ_{TM} is decidable by Q. Then define a TM S such that on input $\langle M\rangle$:

1. Run Q on input $\langle M, M_1 \rangle$, where M_1 is a TM that accepts nothing
2. If Q accepts, *accept*, if Q rejects, *reject*

S is hence a decider for E_{TM}, which is a contradiction as E_{TM} is undecidable, hence no such Q exists.

4.4 Total Problem

The TOTAL problem for TMs is neither RE nor co-RE.

$$TOTAL = \{\langle M \rangle \mid M \text{ halts on all inputs}\}$$

Since problems are decidable iff they are RE and co-RE, and $HALT_{TM}$ is RE, $\overline{HALT_{TM}}$ is not RE. Hence we reduce $\overline{HALT_{TM}}$ to $TOTAL$ and to $\overline{TOTAL}$

4.5 Undecidability

There are three mains methods of proving undecidability. These are

- Diagonalization
  - We show that problem cannot be undecidable by creating a TM which cannot exist without any contradictions
  - This only really works for a small portion of problems such as A_{TM} or $HALT_{TM}$
- Turing Reductions
  - Create an algorithm for a known undecidable problem using the problem to be reduced as a subroutine
  - If the new TM is a decider for the undecidable problem, this is a contradiction, meaning that the problem to be reduced is also undecidable
- Many-One Reductions
  - Let A be a decision problem over Σ_A^* and let B be a decision problem over Σ_B^*.
  - A many-one reduction is a computable function $f : \Sigma_A^* \to \Sigma_B^*$ such that for all $w \in \Sigma_A^*$,

    $$w \in A \iff f(w) \in B$$

  - A computable function f if some TM M on every input w halts with just $f(w)$ on its tape
  - If such a function exists then A is **mapping reducible** to B, written $A \leq_m B$

– This is a stronger type of reduction than Turing reductions, so it may be harder to find these reductions.

Note that if $A \leq_m B$ and B is decidable, then A is decidable, (and hence by contraposition, if A is undecidable, then B is also undecidable). Also if $A \leq_m B$ and B is RE, then A is also RE.

4.6 Rice's Theorem

Theorem 13. *Every non-trivial property of the RE sets is undecidable.*

This means that if P is a decision problem about TMs (expressed as a language) that satisfies:

- For any TMs M_1 and M_2 where $L(M_1) = L(M_2)$, we have that $\langle (\rangle M_1) \in P$ iff $\langle M_2 \rangle \in P$
- There are TMs M_1 and M_2 where $\langle M_1 \rangle \in P$ and $\langle M_2 \rangle \notin P$

Then P is undecidable.

4.7 Computational Complexity

For a deterministic, total TM M, the **running time** of M is the function $f : \mathcal{N} \to \mathcal{N}$, where $f(n)$ is the maximum number of steps that M takes on any input of length n.

Since the number of tapes and alphabet size on a TM is unlimited, we an always speed up the computation by any constant factor by simulating multiple steps at once. This is known as **linear speed up**.

Theorem 14 (Speed Up Theorem)**.** *If L is recognised by some k-tape TM M_1 with runnign time bounded by $f(n)$, then L is also recognised by a k-TAPE TM M_2 with running time bounded by $c \cdot f(n)$, for any $c > 0$, provided $\frac{f(n)}{n} \to \infty$ as $n \to \infty$.*

Using the running time, we can define **time complexity classes**. We define the time complexity class $TIME(t(n))$, to be the set of all languages decidable by a Turing machine with running time $O(t(n))$.

We can also define the running time for nondeterministic TMs as the function $f : \mathcal{N} \to \mathcal{N}$, where $f(n)$ is the maximum number of steps that a TM uses on any branch of its computation on any input of length n.

Theorem 15. *Let $t(n)$ be a function where $t(n) \geq n$. Then every nondeterministic, single-tape TM whose running time is bounded by $t(n)$ has an equivalent $2^{O(t(n))}$ time deterministic single-tape TM.*

This can be proven by constructing a deterministic TM D from nondeterministic N that simulates N by exploring all the branches of N's computation tree.

4.7.1 P

P is the class of all languages that are decidable in polynomial time, i.e.

$$P = \bigcup_{k \geq 0} TIME(n^k)$$

Problems in P are known as **tractable**.

4.7.2 NP

Let $t : \mathcal{N} \to \mathcal{R}^+$ be a function. The time complexity class $NTIME(t(n))$ is the set of all languages decidable by an $O(t(n))$ *nondeterministic* TM.

NP is the class of all languages that are decidable in polynomial time on a nondeterministic single-tape TM:

$$NP = \bigcup_{k \geq 0} NTIME(n^k)$$