CS 208 : Automata Theory and Logic

Spring 2024

Instructor : Prof. Supratik Chakraborty

Disclaimer

This is a compiled version of class notes scribed by students registered for CS 208 (Automata Theory and Logic) in Spring 2024. Please note this document has not received the usual scrutiny that formal publications enjoy. This may be distributed outside this class only with the permission of the instructor.

Contents

1 Propositional Logic 7

1.1 Syntax . . . 7

1.2 Semantics . . . 8

1.2.1 Important Terminology . . . 10

1.3 Proof Rules . . . 11

1.4 Natural Deduction . . . 13

1.5 Soundness and Completeness of our proof system . . . 14

1.6 What about Satisfiability? . . . 16

1.7 Algebraic Laws and Some Redundancy . . . 17

1.7.1 Distributive Laws . . . 17

1.7.2 Reduction of bi-implication and implication . . . 17

1.7.3 DeMorgan's Laws . . . 17

1.8 Negation Normal Forms . . . 17

1.9 From DAG to NNF-DAG . . . 18

1.10 An Efficient Algorithm to convert DAG to NNF-DAG . . . 22

1.11 Conjunctive Normal Forms . . . 26

1.12 Satisfiability and Validity Checking . . . 28

1.13 DAG to Equisatisfiable CNF . . . 29

1.14 Tseitin Encoding . . . 29

1.15 Towards Checking Satisfiability of CNF and Horn Clauses . . . 30

1.16 Counter example for Horn Formula . . . 31

1.16.1 Example . . . 31

1.17 Davis Putnam Logemann Loveland (DPLL) Algorithm . . . 32

1.18 DPLL in action . . . 34

1.18.1 Example . . . 34

1.19 Applying DPLL Algorithm to Horn Formulas . . . 35

1.20 DPLL on Horn Clauses . . . 36

1.21 Rule of Resolution . . . 36

1.21.1 Completeness of Resolution for Unsatisfiability of CNFs . . . 37

2 DFAs and Regular Languages 39

2.1 Definitions . . . 39

2.2 Deterministic Finite Automata . . . 40

2.3 DFA Design - Example 1 . . . 41

2.4 DFA Design - Example 2 . . . 42

3 Non-Deterministic Finite Automata (NFA) 44
3.1 NFA Representation 44
3.1.1 Formalization of Non-Deterministic Finite Automata 45
3.1.2 NFA into action 45
3.2 Equal expressiveness of DFA or NFA 46
3.2.1 Construction of DFA from NFA 46
3.2.2 Step Wise Conversion from NFA to DFA 46
3.2.3 Proof of Equivalence 48
3.3 Reflective Insights 48
3.4 Proof of Equivalence – Correctness 48
3.4.1 Claim 48
3.4.2 Proof 49
3.5 Compiler Special Case – Lexical Analyzer 50
3.6 NFA with $\epsilon-$edges 52
3.6.1 Equivalence with DFA 53
3.7 Recap 55
3.7.1 Converting ε-NFA to non-ε-NFA 55
3.7.2 Correctness of Algorithm 58
3.7.3 Intuition of the algorithm 59
3.7.4 Extras 59
3.7.5 Significance of ε-edges 60
3.7.6 Examples 61
3.8 Equivalence in Finite Automata 62
3.9 DFA definition 62
3.9.1 Combinations of DFAs 62
3.9.2 Combinations of NFAs 63
3.9.3 Closure Properties 63
3.10 Substitution 64
3.10.1 Infinite Languages 64

4 Regular Expressions 65
4.1 Introduction 65
4.2 Formal Definition of a Regular Expression 65
4.3 Semantics of Regular Language 66
4.3.1 Atomic Expressions 66
4.3.2 Union Operation 66
4.3.3 Concatenation Operation 66
4.3.4 Example 66
4.3.5 Order of Precedence 66
4.3.6 Kleene Star 66
4.3.7 Example 66
4.3.8 Further Examples 67
4.4 Kleene's Theorem 67
4.4.1 Part 1: L(Reg. Ex.) $\subseteq$ L(NFAs with $\boldsymbol{\epsilon}$ edges) 68
4.4.2 Part 2: L(Reg. Ex.) $\supseteq$ L(NFAs with $\boldsymbol{\epsilon}$ edges) 70
4.4.3 Checking Subsethood of Languages 75

5 DFA Minimisation 77
5.1 Minimum States in a DFA 77
5.2 Indistinguishability 78
5.3 Equivalence classes of Indistinguishability relation 79
5.4 Further Analysis 81
5.4.1 Optimality of Acquired DFA 81
5.4.2 Uniqueness of Acquired DFA 82
5.5 From states to words 84
5.5.1 Language of word 84
5.5.2 Relation between states of minimal DFA and equivalence classes for $\tilde{}_L$ 84
5.6 Setting up the Parallel 84
5.6.1 Can $| \sim_L | > | \equiv |$? 85
5.6.2 Can$| \sim_L | < | \equiv |$? 86
5.7 Myhill-Nerode Theorem 87

6 Pumping Lemma for Regular Languages 88
6.1 Example 88
6.2 Formal Statement of Pumping Lemma 88
6.3 Pumping Lemma as Adversarial Game 89
6.4 If a language L is regular and the number of states in DFA for L is n, is L infinite? . 89
6.5 Example 90

7 Pushdown Automata 91
7.1 Pushdown Automata for non-regular Languages 91
7.2 Description 92
7.2.1 Conventions 92
7.2.2 Example 92
7.2.3 Example 93
7.3 Acceptance through empty stack 93
7.3.1 For any automaton A, $L(A)$ is not necessarily equivalent to $N(A)$ 94
7.3.2 Construction for A_2 such that $L(A_1) = N(A_2)$ 94
7.4 From Empty Stack PDA to Final State PDA 96
7.4.1 Construction 96
7.4.2 Required to show $N(P) = L(P_F)$ 97
7.5 Context Free Languages 97
7.5.1 DPDA and NPDAs 97
7.5.2 Build-up and emptying of the stack of a PDA 98
7.6 Acceptance by PDA 98
7.6.1 Run on the PDA 99
7.6.2 Recurrence Relations on Languages 100
7.7 More on recurrence relations 100
7.7.1 Smallest and Largest Languages 101
7.7.2 Context Free Grammar 101

8 Context Free Grammer **102**
8.1 Context-free Grammar 102
8.1.1 PDA to CFG 103
8.2 Parse Trees and Ambiguous Grammars 103
8.2.1 CFG to PDA conversion 105
8.3 Cleaning Context-free Grammar 107
8.3.1 Eliminating Useless Symbols 107
8.3.2 Eliminating ϵ-Productions 108
8.3.3 Eliminating Unit Productions 108
8.3.4 Reducing further to get at least two non-terminals on the right hand side or a single terminal 109
8.3.5 Reducing further to get exactly two non-terminals on the RHS of any production rule which has >2 non-terminals on it's RHS 109
8.3.6 Chomsky Normal form (CNF) 109
8.4 Pumping Lemma for Context-free Languages 110
8.4.1 Lemma: 110
8.5 Closure Properties of CFL 111
8.5.1 Union & Concatenation 111
8.5.2 Intersection 112
8.5.3 Complement 112
8.5.4 Substitution 112

9 Turing Machine **113**
9.1 Configuration of the Turing Machine 114
9.2 Language described by a TM 117
9.2.1 Inter conversion between acceptance by final state and by halting: 117
9.3 Languages accepted by a TM 118
9.4 Multiple Tapes in a TM 119
9.4.1 Converting a Multiple Tape TM to a Single Tape TM 119
9.4.2 Working of the Single Tape TM 122
9.5 Non Deterministic Turing Machines-NTM 123
9.6 Turing Machine Subroutines 125
9.7 Restricted Turing Machines 126
9.7.1 Turing Machines with Semi-infinite Tapes 126
9.7.2 Two Stack Machine 127
9.7.3 Counter Machines 128
9.8 Simulators for Turing Machine 129
9.8.1 Turing Machine as Two Stack Machine 129
9.8.2 Simulation of Two Stack Machine using Three Counters 130
9.9 Stimulating a computer using a TM 131
9.10 Representing a Turing Machine as a Natural Number 133
9.10.1 Mapping Natural Numbers to Turing Machines 133
9.10.2 Encoding Input Strings as Natural Numbers 134
9.11 Recursively enumerable languages 135
9.11.1 Enumerating the strings of a recursively enumerable language 135
9.11.2 Is $\overline{\mathbf{L_u}}$ a R.E. Language? 137

9.11.3 Construction of a language which is not recursively enumerable 138
9.11.4 Undecidable Languages 139
9.11.5 Universal language 139
9.11.6 Complement of Universal Language 139
9.12 Recursive Languages 140
9.12.1 Recursive or not? 140
9.12.2 Complementation of Recursive languages 141
9.12.3 Recursive → Recursively Enumerable 141
9.13 Decidable languages 141
9.14 Rice Theorem 142
9.14.1 Classification of Properties 142
9.14.2 Decidability of Non-Trivial Properties 143
9.14.3 Proof of Rice's Theorem 143

10 First Order Logic 146
10.1 Introduction 146
10.1.1 Notation 146
10.1.2 FOL formulas as strings 148
10.1.3 Free variables in a formula 149
10.1.4 Substitution in First Order Logic (FOL) 151

Chapter 1

Propositional Logic

In this course we look at two ways of computation: a state transition view and a logic centric view. In this chapter we begin with logic centered view with the discussion of propositional logic.

Example. Suppose there are five courses $C_1, \ldots, C_5$, four slots $S_1, \ldots, S_4$, and five days $D_1, \ldots, D_5$. We plan to schedule these courses in three slots each, but we have also have the following requirements:

- For every course C_i, the three slots should be on three different days.
- Every course C_i should be scheduled in at most one of $S_1, \ldots, S_4$.
- For every day D_i of the week, have at least one slot free.

	D_1	D_2	D_3	D_4	D_5
S_1					
S_2					
S_3					
S_4					

Propositional logic is used in many real-world problems like timetables scheduling, train scheduling, airline scheduling, and so on. One can capture a problem in a propositional logic formula. This is called as encoding. After encoding the problem, one can use various software tools to systematically reason about the formula and draw some conclusions about the problem.

1.1 Syntax

We can think of logic as a language which allows us to very precisely describe problems and then reason about them. In this language, we will write sentences in a specific way. The symbols used in propositional logic are given in Table 1.1. Apart from the symbols in the table we also use variables usually denoted by small letters $p, q, r, x, y, z, \ldots$ etc. Here is a short description of propositional logic symbols:

- **Variables**: They are usually denoted by smalls ($p, q, r, x, y, z, \ldots$ etc). The variables can take up only true or false values. We use them to denote propositions.
- **Constants**: The constants are represented by $\top$ and $\bot$. These represent truth values true and false.

- **Operators**: $\wedge$ is the conjunction operator (also called AND), $\vee$ is the disjunction operator (also called OR), $\neg$ is the negation operator (also called NOT), $\rightarrow$ is implication, and $\leftrightarrow$ is bi-implication (equivalence).

Name	Symbol	Read as
true	$\top$	top
false	$\bot$	bot
negation	$\neg$	not
conjunction	$\wedge$	and
disjunction	$\vee$	or
implication	$\rightarrow$	implies
equivalence	$\leftrightarrow$	if and only if
open parenthesis	(	
close parenthesis	)	

Table 1.1: Logical connectives.

For the timetable example, we can have propositional variables of the form p_{ijk} with $i \in [5]$, $j \in [5]$ and $k \in [4]$ (Note that $[n] = \{1, \ldots, n\}$) with p_{ijk} representing the proposition 'course C_i is scheduled in slot S_k of day D_j'.

Rules for formulating a formula:

- Every variable constitutes a formula.
- The constants $\top$ and $\bot$ are formulae.
- If φ is a formula, so are $\neg\varphi$ and (φ).
- If φ_1 and φ_2 are formulas, so are $\varphi_1 \wedge \varphi_2, \varphi_1 \vee \varphi_2, \varphi_1 \rightarrow \varphi_2$, and $\varphi_1 \leftrightarrow \varphi_2$.

Propositional formulae as strings and trees:

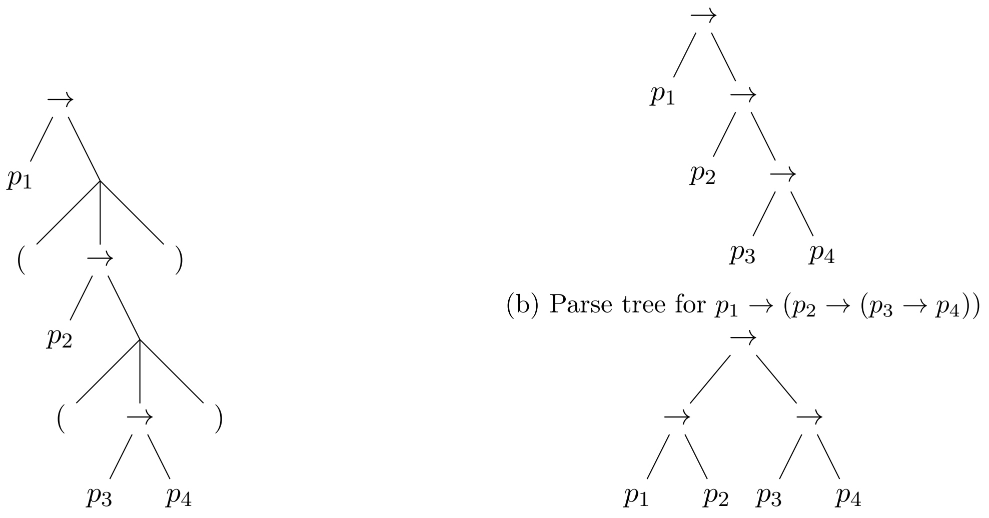
Formulae can be expressed as a strings over the alphabet $\mathbf{Vars} \cup \{\top, \bot, \neg, \wedge, \vee, \rightarrow, \leftrightarrow, (,)\}$. **Vars** is the set of symbols for variables. Not all words formed using the alphabet qualify as propositional formulae. A string constitutes a well-formed formula (wff) if it was constructed while following the rules. Examples: $(p_1 \vee \neg q_2) \wedge (\neg p_2 \rightarrow (q_1 \leftrightarrow \neg p_1))$ and $p_1 \rightarrow (p_2 \rightarrow (p_3 \rightarrow p_4))$.

Well-formed formulas can be represented using trees. Consider the formula $p_1 \rightarrow (p_2 \rightarrow (p_3 \rightarrow p_4))$. This can be represented using the parse tree in figure Figure 1.1a. Notice that while strings require parentheses for disambiguation, trees don't, as can be seen in Figure 1.1b and Figure 1.1c.

1.2 Semantics

Semantics give a meaning to a formula in propositional logic. The semantics is a function that takes in the truth values of all the variables that appear in a formula and gives the truth value of the formula. Let 0 represent "false" and 1 represent "true". The semantics of a formula φ of n variables is a function

$$[\![\varphi]\!] : \{0,1\}^n \rightarrow \{0,1\}$$



(a)

(b) Parse tree for $p_1 \to (p_2 \to (p_3 \to p_4))$

(c) Parse tree for $(p_1 \to p_2) \to (p_3 \to p_4)$

Figure 1.1: Parse trees obviate the need for parentheses.

It is often presented in the form of a truth table. Truth tables of operators can be found in table Table 1.2.

φ	$\neg\varphi$
0	1
1	0

(a) Truth table for $\neg\varphi$.

φ_1	φ_2	$\varphi_1 \wedge \varphi_2$
0	0	0
0	1	0
1	0	0
1	1	1

(b) Truth table for $\varphi_1 \wedge \varphi_2$.

φ_1	φ_2	$\varphi_1 \vee \varphi_2$
0	0	0
0	1	1
1	0	1
1	1	1

(c) Truth table for $\varphi_1 \vee \varphi_2$.

φ_1	φ_2	$\varphi_1 \to \varphi_2$
0	0	1
0	1	1
1	0	0
1	1	1

(d) Truth table for $\varphi_1 \to \varphi_2$.

φ_1	φ_2	$\varphi_1 \leftrightarrow \varphi_2$
0	0	1
0	1	0
1	0	0
1	1	1

(e) Truth table for $\varphi_1 \leftrightarrow \varphi_2$.

Table 1.2: Truth tables of operators.

Remark. Do not confuse 0 and 1 with $\top$ and $\bot$: 0 (false) and 1 (true) are meanings, while $\top$ and $\bot$ are symbols.

Rules of semantics:

- $[\![\neg\varphi]\!] = 1$ iff $[\![\varphi]\!] = 0$.
- $[\![\varphi_1 \wedge \varphi_2]\!] = 1$ iff $[\![\varphi_1]\!] = [\![\varphi_2]\!] = 1$.
- $[\![\varphi_1 \vee \varphi_2]\!] = 1$ iff at least one of $[\![\varphi_1]\!]$ or $[\![\varphi_2]\!]$ evaluates to 1.

- $\llbracket \varphi_1 \rightarrow \varphi_2 \rrbracket = 1$ iff at least one of $\llbracket \varphi_1 \rrbracket = 0$ or $\llbracket \varphi_2 \rrbracket = 1$.
- $\llbracket \varphi_1 \leftrightarrow \varphi_2 \rrbracket = 1$ iff at both $\llbracket \varphi_1 \rightarrow \varphi_2 \rrbracket = 1$ and $\llbracket \varphi_2 \rightarrow \varphi_1 \rrbracket = 1$.

Truth Table: A truth table in propositional logic enumerates all possible truth values of logical expressions. It lists combinations of truths for individual propositions and the compound statement's truth.

Example. Let us construct a truth table for $\llbracket (p \vee s) \rightarrow (\neg q \leftrightarrow r) \rrbracket$ (see Table 1.3).

p	q	r	s	$p \vee s$	$\neg q$	$\neg q \leftrightarrow r$	$(p \vee s) \rightarrow (\neg q \leftrightarrow r)$
0	0	0	0	0	1	0	1
0	0	0	1	1	1	0	0
0	0	1	0	0	1	1	1
0	0	1	1	1	1	1	1
0	1	0	0	0	0	1	1
0	1	0	1	1	0	1	1
0	1	1	0	0	0	0	1
0	1	1	1	1	0	0	0
1	0	0	0	1	1	0	0
1	0	0	1	1	1	0	0
1	0	1	0	1	1	1	1
1	0	1	1	1	1	1	1
1	1	0	0	1	0	1	1
1	1	0	1	1	0	1	1
1	1	1	0	1	0	0	0
1	1	1	1	1	0	0	0

Table 1.3: Truth table of $(p \vee s) \rightarrow (\neg q \leftrightarrow r)$.

1.2.1 Important Terminology

A formula φ is said to (be)

- **satisfiable** or **consistent** or SAT iff $\llbracket \varphi \rrbracket = 1$ for some assignment of variables. That is, there is at least one way to assign truth values to the variables that makes the entire formula true. Both a formula and its negation may be SAT at the same time (φ and $\neg\varphi$ may both be SAT).
- **unsatisfiable** or **contradiction** or UNSAT iff $\llbracket \varphi \rrbracket = 0$ for all assignments of variables. That is, there is no way to assign truth values to the variables that makes the formula true. If a formula φ is UNSAT then $\neg\varphi$ must be SAT (it is in fact valid).
- **valid** or **tautology**: $\llbracket \varphi \rrbracket = 1$ for all assignments of variables. That is, the formula is always true, no matter how the variables are assigned. If a formula φ is valid then $\neg\varphi$ is UNSAT.
- **semantically entail** φ_1 iff $\llbracket \varphi \rrbracket \preceq \llbracket \varphi_1 \rrbracket$ for all assignments of variables, where 0 (false) $\preceq$ 1 (true). This is denoted by $\varphi \models \varphi_1$. If $\varphi \models \varphi_1$, then for every assignment, if φ evaluates to 1 then φ_1 will evaluate to 1. Equivalently $\varphi \rightarrow \varphi_1$ is valid.

- **semantically equivalent** to φ_1 iff $\varphi \models \varphi_1$ and $\varphi_1 \models \varphi$. Basically φ and φ_1 have identical truth tables. Equivalently, $\varphi \leftrightarrow \varphi_1$ is valid.
- **equisatisfiable** to φ_1 iff either both are SAT or both are UNSAT. Also note that, semantic equivalence implies equisatisfiability but **not** vice-versa.

Term	Example
SAT	$p \vee q$
UNSAT	$p \wedge \neg p$
valid	$p \vee \neg p$
semantically entails	$\neg p \models p \to q$
semantically equivalent	$p \to q,\ \neg p \vee q$
equisatisfiable	$p \wedge q,\ r \vee s$

Table 1.4: Some examples for the definitions.

Example. Consider the formulas $\varphi_1 : p \to (q \to r)$, $\varphi_2 : (p \wedge q) \to r$ and $\varphi_3 : (q \wedge \neg r) \to \neg p$. The three formulas φ_1, φ_2 and φ_3 are semantically equivalent. One way to check this is to construct the truth table.

On drawing the truth table for the above example, one would realise that it is laborious. Indeed, for a formula with n variables, the truth table has 2^n entries! So truth tables don't work for large formulas. We need a more systematic way to reason about the formulae. That leads us to proof rules...
But before that let us get a closure on the example at the beginning of the chapter. Let p_{ijk} represent the proposition 'course C_i is scheduled in slot S_k of day D_j'. We can encode the constraints using the encoding strategy used in tutorial 1 - problem 3. That is, by introducing extra variables that bound the sum for first few variables (sum of i is atmost j). Using this we can encode the constraints as : $\sum_{k=1}^{4} p_{ijk} \leq 1$, $\sum_{j=1}^{5} p_{ijk} \leq 1$, $\sum_{i=1}^{5} p_{ijk} \leq 1$, $\sum_{k=1}^{4}\sum_{i=1}^{5} p_{ijk} \leq 3$, $\sum_{k=1}^{4}\sum_{j=1}^{5} p_{ijk} \leq 3$ and $\neg\big(\sum_{k=1}^{4}\sum_{j=1}^{5} p_{ijk} \leq 2\big)$.

1.3 Proof Rules

After encoding a problem into propositional formula we would like to reason about the formula. Some of the properties of a formula that we are usually interested in are whether it is SAT, UNSAT or valid. We have already seen that truth tables do not scale well for large formulae. It is also not humanly possible to reason about large formulae modelling real-world systems. We need to delegate the task to computers. Hence, we need to make systematic rules that a computer can use to reason about the formulae. These are called as proof rules.
The overall idea is to convert a formula to a normal form (basically a standard form that will make reasoning easier - more about this later in the chapter) and use proof rules to check SAT etc.

Rules are represented as

$$\frac{\text{Premises}}{\text{Inferences}}\ \text{Connector}_{i/e}$$

- **Premise**: A premise is a formula that is assumed or is known to be true.

- **Inference**: The conclusion that is drawn from the premise(s).
- **Connector**: It is the logical operator over which the rule works. We use the subscript i (for introduction) if the connector and the premises are combined to get the inference. The subscript e (for elimination) is used when we eliminate the connector present in the premises to draw inference.

Example. Look at the following rule

$$\frac{\varphi_1 \wedge \varphi_2}{\varphi_1} \wedge_{e_1}$$

In the rule above $\varphi_1 \wedge \varphi_2$ is assumed (is premise). Informally, looking at $\wedge$'s truth table, we can infer that both φ_1 and φ_2 are true if $\varphi_1 \wedge \varphi_2$ is true, so φ_1 is an inference. Also, in this process we eliminate (remove) $\wedge$ so we call this AND-ELIMINATION or $\wedge_e$. For better clarity we call this rule $\wedge_{e_1}$ as φ_1 is kept in the inference even when both φ_1 and φ_2 could be kept in inference. If we use φ_2 in inference then the rule becomes $\wedge_{e_2}$.

Table 1.5 summarises the basic proof rules that we would like to include in our proof system.

Connector	**Introduction**	**Elimination**
$\wedge$	$\dfrac{\varphi_1 \quad \varphi_2}{\varphi_1 \wedge \varphi_2} \wedge_i$	$\dfrac{\varphi_1 \wedge \varphi_2}{\varphi_1} \wedge_{e_1} \qquad \dfrac{\varphi_1 \wedge \varphi_2}{\varphi_2} \wedge_{e_2}$
$\vee$	$\dfrac{\varphi_1}{\varphi_1 \vee \varphi_2} \vee_{i_1} \qquad \dfrac{\varphi_2}{\varphi_1 \vee \varphi_2} \vee_{i_2}$	$\dfrac{\varphi_1 \vee \varphi_2 \quad \varphi_1 \rightarrow \varphi_3 \quad \varphi_2 \rightarrow \varphi_3}{\varphi_3} \vee_e$
$\rightarrow$	$\dfrac{\boxed{\begin{matrix}\varphi_1 \\ \vdots \\ \varphi_2\end{matrix}}}{\varphi_1 \rightarrow \varphi_2} \rightarrow_i$	$\dfrac{\varphi_1 \quad \varphi_1 \rightarrow \varphi_2}{\varphi_2} \rightarrow_e$
$\neg$	$\dfrac{\boxed{\begin{matrix}\varphi \\ \vdots \\ \bot\end{matrix}}}{\neg\varphi} \neg_i$	$\dfrac{\varphi \quad \neg\varphi}{\bot} \neg_e$
$\bot$		$\dfrac{\bot}{\varphi} \bot_e$
$\neg\neg$		$\dfrac{\neg\neg\varphi}{\varphi} \neg\neg_e$

Table 1.5: Proof rules.

In the $\rightarrow_i$ rule, the box indicates that we can *temporarily* assume φ_1 and conclude φ_2 using no extra non-trivial information. The $\rightarrow_e$ is referred to by its Latin name, *modus ponens*.

Example 1. We can now use these proof rules along with $\varphi_1 \wedge (\varphi_2 \wedge \varphi_3)$ as the premise to conclude $(\varphi_1 \wedge \varphi_2) \wedge \varphi_3$.

$$\frac{\varphi_1 \wedge (\varphi_2 \wedge \varphi_3)}{\varphi_2 \wedge \varphi_3}\wedge_{e_2} \qquad \frac{\varphi_1 \wedge (\varphi_2 \wedge \varphi_3)}{\varphi_1}\wedge_{e_1}$$

$$\frac{\varphi_2 \wedge \varphi_3}{\varphi_2}\wedge_{e_1} \qquad \frac{\varphi_2 \wedge \varphi_3}{\varphi_3}\wedge_{e_2}$$

$$\frac{\varphi_1 \quad \varphi_2}{\varphi_1 \wedge \varphi_2}\wedge_i$$

$$\frac{\varphi_1 \wedge \varphi_2 \quad \varphi_3}{(\varphi_1 \wedge \varphi_2) \wedge \varphi_3}\wedge_i$$

1.4 Natural Deduction

If we can begin with some formulas $\phi_1, \phi_2, \ldots, \phi_n$ as our premises and then conclude φ by applying the proof rules established we say that $\phi_1, \phi_2, \ldots, \phi_n$ syntactically entail φ which is denoted by the following expression, also called a *sequent*:

$$\phi_1, \phi_2, \ldots, \phi_n \vdash \varphi.$$

We can also infer some formula using no premises, in which case the sequent is $\vdash \varphi$.
Applying these proof rules involves the following general rule:

> We can only use a formula φ at a point if it occurs prior to it in the proof and if **no box enclosing that occurrence of φ has been closed already**.

Example. Consider the following proof of the sequent $\vdash p \vee \neg p$:

1.	$\neg(p \vee \neg p)$	assumption
2.	p	assumption
3.	$p \vee \neg p$	$\vee_{i_1}$ 2
4.	$\bot$	$\neg_e$ 3,1
5.	$\neg p$	$\neg_i$ 2–4
6.	$p \vee \neg p$	$\vee_{i_2}$ 5
7.	$\bot$	$\neg_e$ 6,1
8.	$\neg\neg(p \vee \neg p)$	$\neg_i$ 1–7
9.	$p \vee \neg p$	$\neg\neg_e$ 8

Example. Proof for $p \vdash \neg\neg p$ which is $\neg\neg_i$, a derived rule

1.	p	premise
2.	$\neg p$	assumption
3.	$\bot$	$\neg_e$ 1, 2
4.	$\neg\neg p$	$\neg_i$ 2–3

Example. A useful derived rule is *modus tollens* which is $p \to q, \neg q \vdash \neg p$:

1.	$p \to q$	premise
2.	$\neg q$	premise
3.	p	assumption
4.	q	$\to_e$ 3,1
5.	$\bot$	$\neg_e$ 4,2
6.	$\neg p$	$\neg_i$ 3–5

Example. $\neg p \wedge \neg q \vdash \neg(p \vee q)$:

1.	$\neg p \wedge \neg q$	premise
2.	$p \vee q$	assumption
3.	p	assumption
4.	$\neg p$	$\wedge_{e_1}$ 1
5.	$\bot$	$\neg_e$ 3,4
6.	$p \to \bot$	$\to_i$ 3–5
7.	q	assumption
8.	$\neg q$	$\wedge_{e_2}$ 1
9.	$\bot$	$\neg_e$ 7,8
10.	$q \to \bot$	$\to_i$ 7–9
11.	$\bot$	$\vee_e$ 2,6,10
12.	$\neg(p \vee q)$	$\neg_i$ 2–11

1.5 Soundness and Completeness of our proof system

A proof system is said to be sound if everything that can be derived using it matches the semantics.

Soundness: $\Sigma \vdash \varphi$ implies $\Sigma \models \varphi$

The rules that we have chosen are indeed individually sound since they ensure that if for some assignment the premises evaluate to 1, so does the inference. Otherwise they rely on the notion of contradiction and assumption. Hence, soundness for any proof can be shown by inducting on the length of the proof.
A complete proof system is one which allows the inference of *every* valid semantic entailment:

Completeness: $\Sigma \models \varphi$ implies $\Sigma \vdash \varphi$

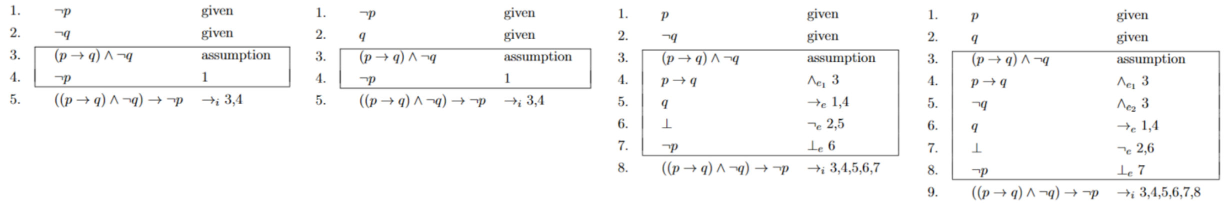
Let's take some example of semantic entailment. $\Sigma = \{p \to q, \neg q\} \models \neg p$.

p	q	$p \to q$	$\neg q$	$\neg p$
0	0	1	1	1
0	1	1	0	1
1	0	0	1	0
1	1	1	0	0

As we can see, whenever both $p \to q$ and $\neg q$ are true, $\neg p$ is true. The question now is how do we derive this using proof rules? The idea is to 'mimic' each row of the truth table. This means that we assume the values for p, q and try to prove that the formulae in Σ imply φ[1]. And to prove an implication, we can use the $\to_i$ rule. Here's an example of how we can prove our claim for the first row:

1.	$\neg p$	given
2.	$\neg q$	given
3.	$(p \to q) \land \neg q$	assumption
4.	$\neg p$	1
5.	$((p \to q) \land \neg q) \to \neg p$	$\to_i$ 3,4

Similarly to mimic the second row, we would like to show $\neg p, q \vdash ((p \to q) \land \neg q) \to \neg p$. Actually for every row, we'd like to start with the assumptions about the values of each variable, and then try to prove the property that we want.



1.	$\neg p$	given
2.	$\neg q$	given
3.	$(p \to q) \land \neg q$	assumption
4.	$\neg p$	1
5.	$((p \to q) \land \neg q) \to \neg p$	$\to_i$ 3,4

1.	$\neg p$	given
2.	q	given
3.	$(p \to q) \land \neg q$	assumption
4.	$\neg p$	1
5.	$((p \to q) \land \neg q) \to \neg p$	$\to_i$ 3,4

1.	p	given
2.	$\neg q$	given
3.	$(p \to q) \land \neg q$	assumption
4.	$p \to q$	$\land_{e_1}$ 3
5.	q	$\to_e$ 1,4
6.	$\bot$	$\neg_e$ 2,5
7.	$\neg p$	$\bot_e$ 6
8.	$((p \to q) \land \neg q) \to \neg p$	$\to_i$ 3,4,5,6,7

1.	p	given
2.	q	given
3.	$(p \to q) \land \neg q$	assumption
4.	$p \to q$	$\land_{e_1}$ 3
5.	$\neg q$	$\land_{e_2}$ 3
6.	q	$\to_e$ 1,4
7.	$\bot$	$\neg_e$ 2,6
8.	$\neg p$	$\bot_e$ 7
9.	$((p \to q) \land \neg q) \to \neg p$	$\to_i$ 3,4,5,6,7,8

Figure 1.2: Mimicking all 4 rows of the truth table

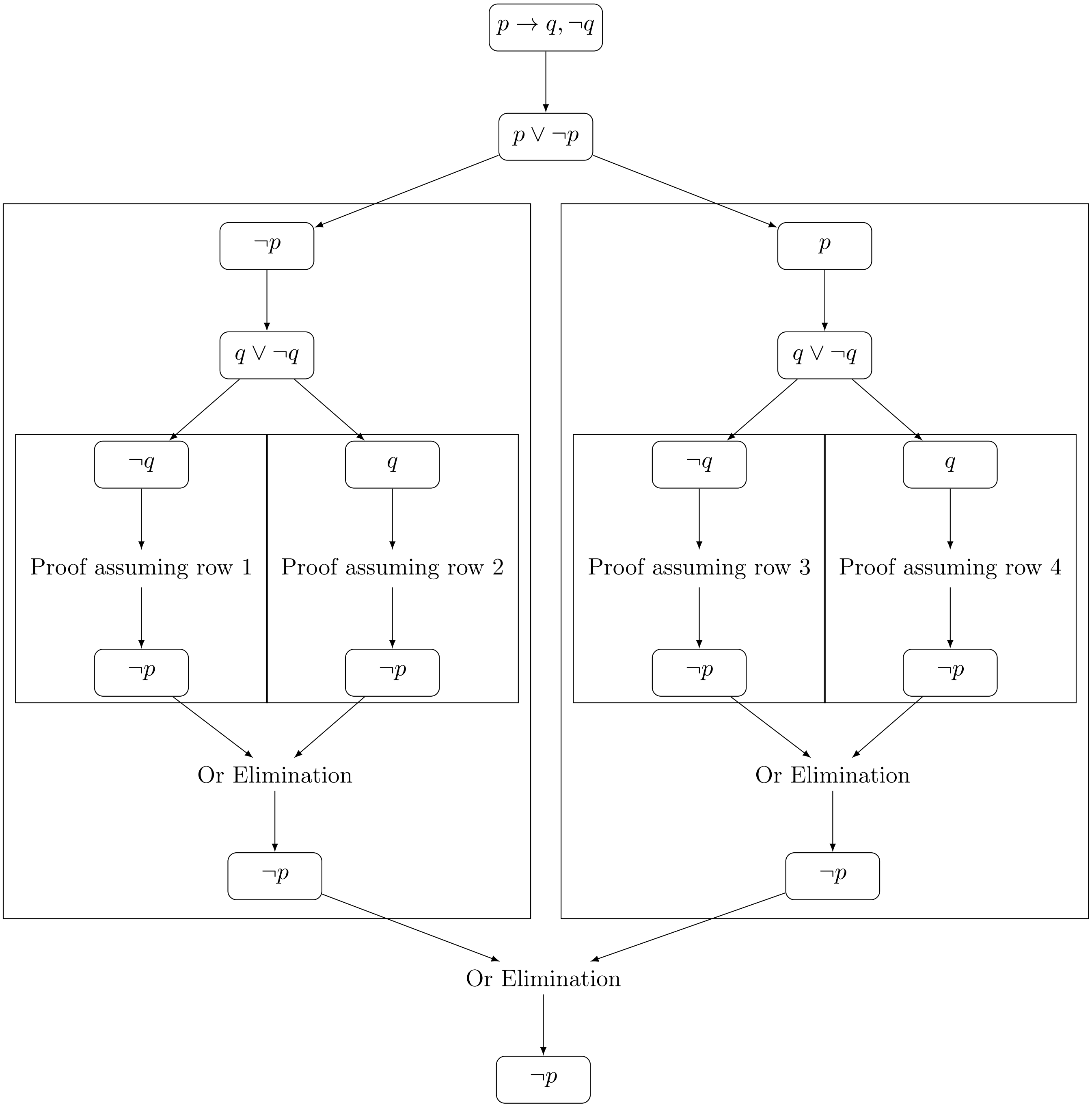
This looks promising, but we aren't done, we have only proven our formula under all possible assumptions, but we haven't exactly proven our formula from nothing given. But note that the reasoning we are doing looks a lot like case work, and we can think of the $\vee_e$ rule. In words, this rule states that if a formula is true under 2 different assumptions, and one of the assumptions is always true, then our formula is true. So if we just somehow rigorously show at least one of our row assumptions is always true, we will be able to clean up our proof using the $\vee_e$ rule.

But as seen above, we were able to show a proof for the sequent $\vdash \varphi \vee \neg\varphi$. If we just recursively apply this property for all the variables we have, we should be able to capture every row of truth table. So combining this result, our proofs for each row of the truth table, and the $\vee_e$ rule, the whole proof is constructed as below. The only thing we need now is the ability to construct proofs for each row given the general valid formula $\bigwedge_{\phi \in \Sigma} \phi \to \varphi$.

This can be done using **structural induction** to prove the following:
Let φ be a formula using the propositional variables $p_1, p_2, \ldots, p_n$. For any assignment to these variables define $\hat{p}_i = p_i$ if p_i is set to 1 and $\hat{p}_i = \neg p_i$ otherwise, then:

$$\hat{p}_1, \hat{p}_2, \ldots, \hat{p}_n \vdash \varphi \text{ is provable if } \varphi \text{ evaluates to 1 for the assignment}$$
$$\hat{p}_1, \hat{p}_2, \ldots, \hat{p}_n \vdash \neg\varphi \text{ is provable if } \varphi \text{ evaluates to 0 for the assignment.}$$

[1] Σ semantically entails φ is equivalent to saying intersection of formulae in Σ implies φ is valid



1.6 What about Satisfiability?

Using Natural Deduction, we can only talk about the formulas that are a contradiction or valid. But there are formulas that are neither i.e. they are satisfiable for some assignment of variables but not all. Example, for some p and q,

$$\nvdash p \wedge q$$

But clearly, $p \wedge q$ is satisfiable when both p and q are true. Natural deduction can only claim statements like,

$$\vdash \neg p \rightarrow \neg(p \wedge q)$$

$$\vdash (p \rightarrow (q \rightarrow (p \wedge q))$$

An important link between the two situations is that,

A formula ϕ is valid **iff** $\neg\phi$ is not satisfiable

1.7 Algebraic Laws and Some Redundancy

1.7.1 Distributive Laws

Here are some identities that help complex formulas to some required forms discussed later. These formulas are easily derived using the natural deduction proof rules as discussed above.

$$\phi_1 \wedge (\phi_2 \vee \phi_3) \dashv\models (\phi_1 \wedge \phi_2) \vee (\phi_1 \wedge \phi_3)$$

$$\phi_1 \vee (\phi_2 \wedge \phi_3) \dashv\models (\phi_1 \vee \phi_2) \wedge (\phi_1 \vee \phi_3)$$

1.7.2 Reduction of bi-implication and implication

We also see that bi-implication and implication can be reduced to $\vee$,$\wedge$ and $\neg$ and are therefore redundant in the alphabet of our string.

$$\phi_1 \longleftrightarrow \phi_2 \dashv\models (\phi_1 \rightarrow \phi_2) \wedge (\phi_2 \rightarrow \phi_1)$$

$$(\phi_1 \rightarrow \phi_2) \dashv\models ((\neg\phi_1) \vee \phi_2)$$

1.7.3 DeMorgan's Laws

Similar to distributive laws, the following laws (again easily provable via natural deduction) help reduce any normal string to a suitable form(discuissed in the next section).

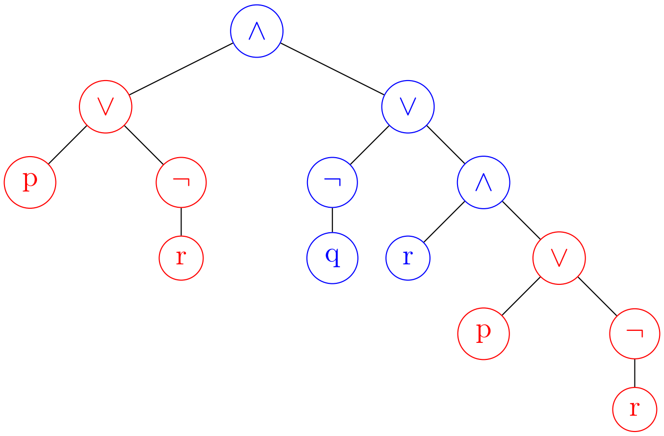
$$\neg(\phi_1 \wedge \phi_2) \dashv\models (\neg\phi_1 \vee \neg\phi_2)$$

$$\neg(\phi_1 \vee \phi_2) \dashv\models (\neg\phi_1 \wedge \neg\phi_2)$$

1.8 Negation Normal Forms

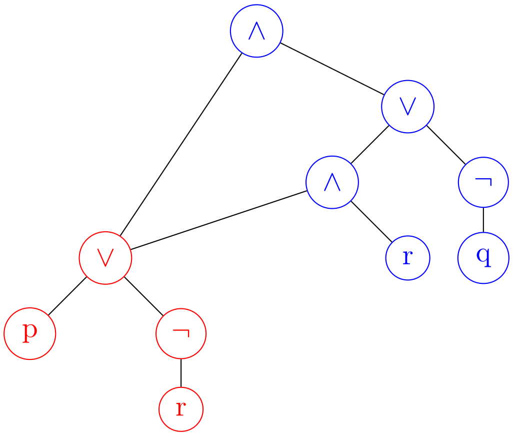
In mathematical logic, a formula is in negation normal form (NNF) if the negation operator ($\neg$) is only applied to variables and the only other allowed Boolean operators are conjunction ($\wedge$) and disjunction($\vee$) One can convert any formula to a NNF by repeatedly applying DeMorgan's Laws to any clause that may have a $\neg$, until only the variables have the $\neg$ operator. For example
A NNF formula may be represented using it's parse tree, which doesn't have any negation nodes except at the leaves, consider (p $\vee$ $\neg$ r) $\wedge$ ($\neg$ q $\vee$ (r $\wedge$ (p $\vee$ $\neg$ r)))

Parse Tree Representation of the Above Example:



Since , the red part of the tree is repeating twice , we can make a **DAG(Directed Acyclic Graph)** instead of the parse tree.

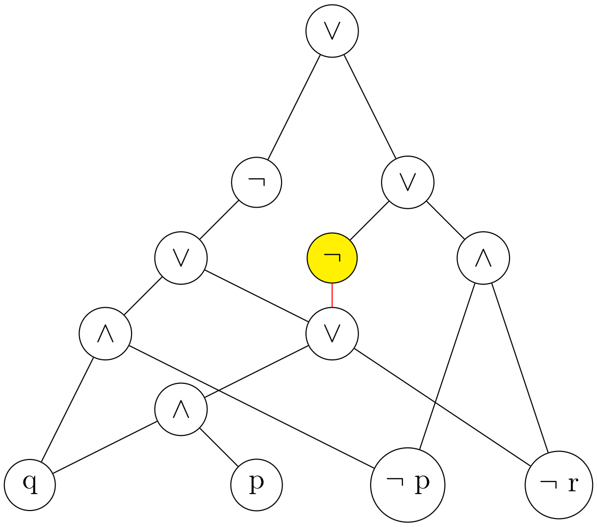
DAG Representation



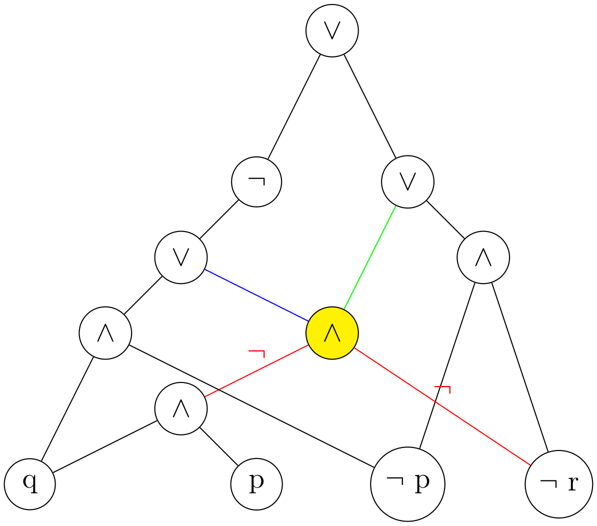
1.9 From DAG to NNF-DAG

Given a DAG of propositional logic formula with only ∨ , ∧ and ¬ nodes , can we efficiently get a DAG representing a semantically equivalent NNF formula ?

Idea 1: Let's push the "¬" downwards by applying the De Morgans Law and see what happens. Lets Consider the following example and the highlighted ¬.



Pushing down the highlighted ¬ across the red edge.



Now , we have an issue in the blue edge. The blue edge wanted the non - negated tree node but due to the above mentioned change , it is getting the negated node. So, this idea won't work. We want to preserve the non-negated nodes as well.

Modification : Make two copies of the DAG and negate(i.e , ¬ pushing) only 1 of the copies and if a nodes wants non - negated node then take that node from the copied tree.

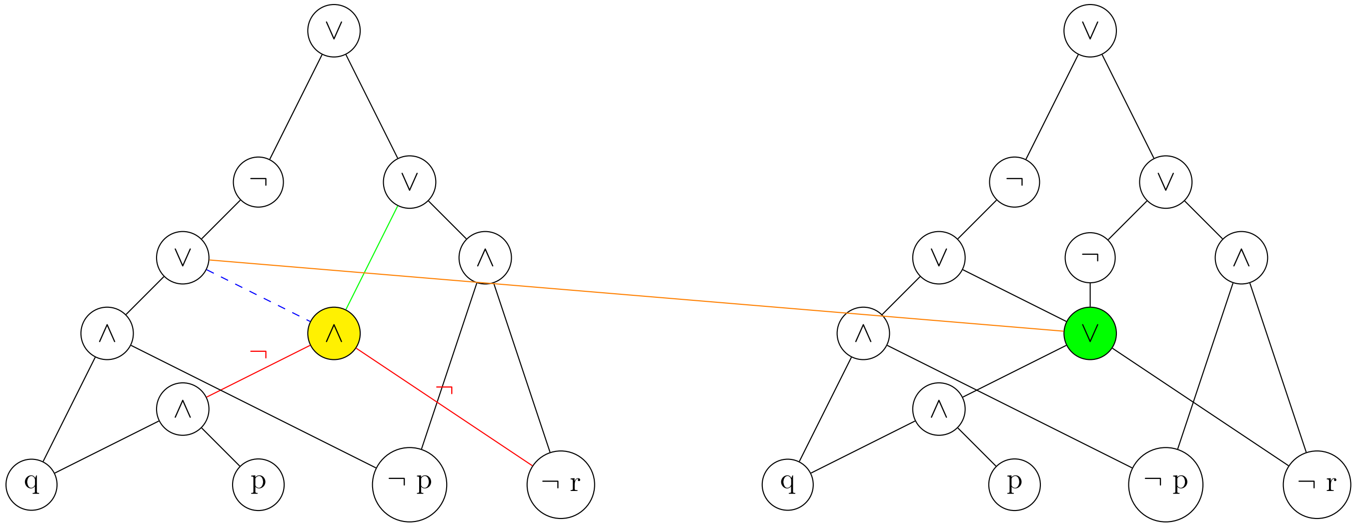


Figure 1.3: Step 1

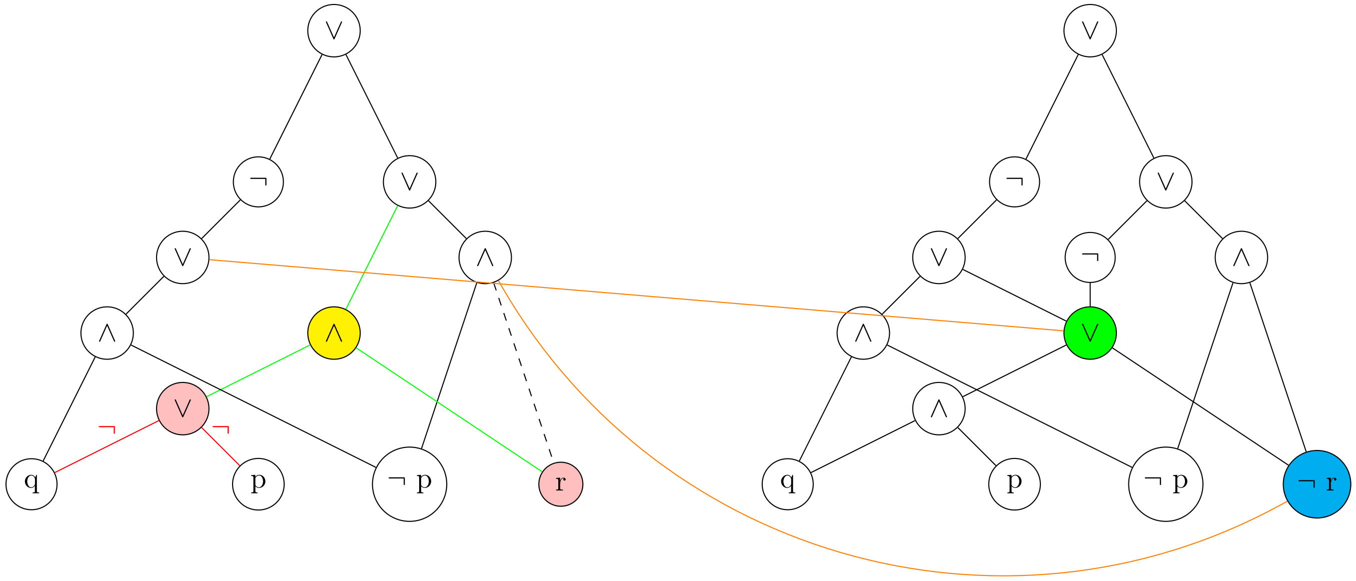


Figure 1.4: Step 2

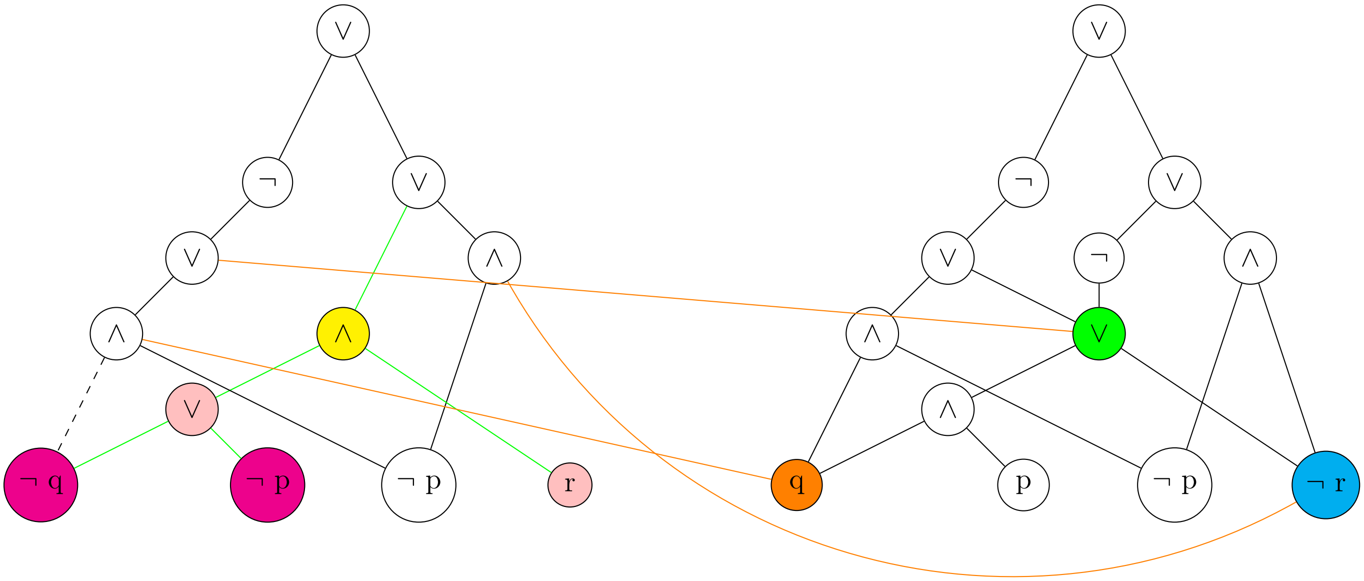


Figure 1.5: Step 3

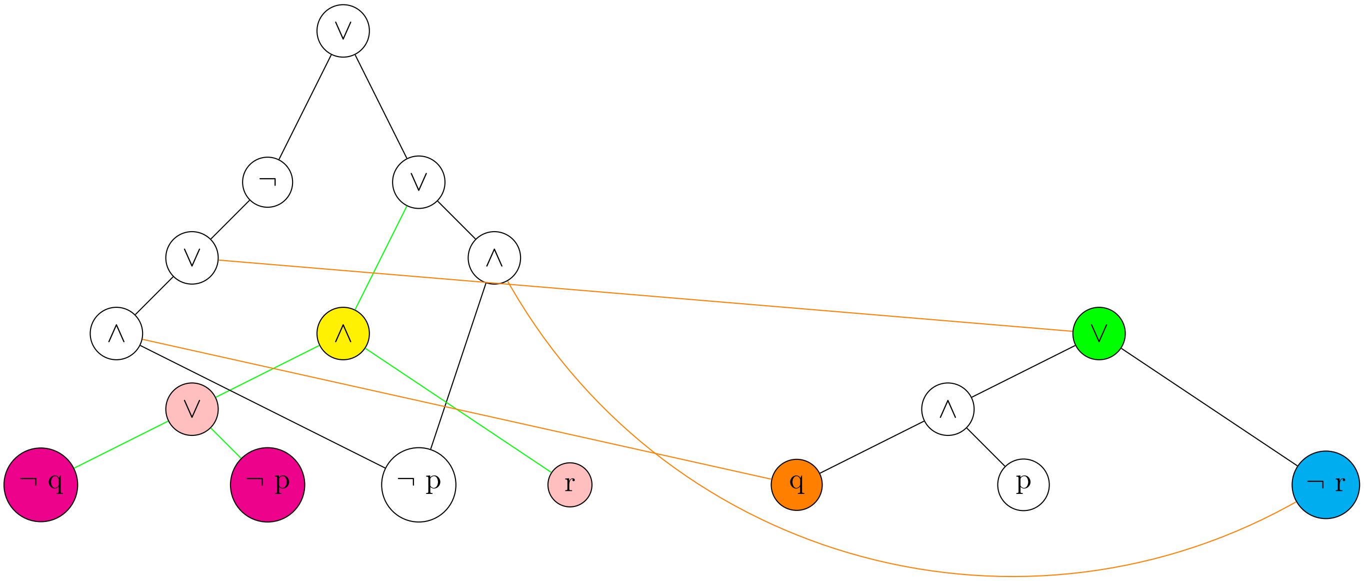


Figure 1.6: Step 4 : Remove all redundant nodes

1.10 An Efficient Algorithm to convert DAG to NNF-DAG

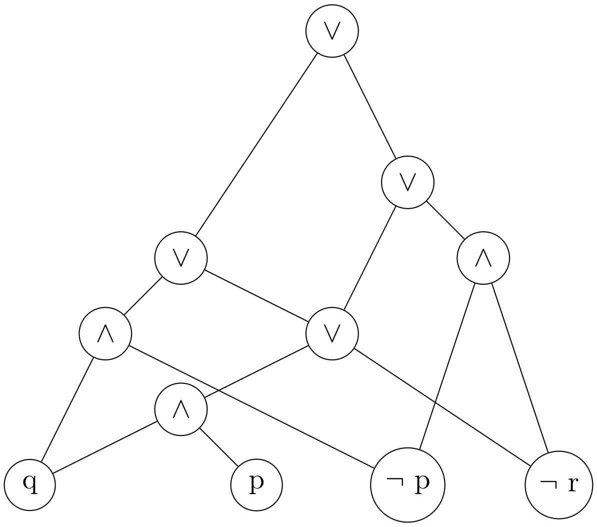


Figure 1.7: Step 1:Make a copy of the DAG and Remove all "¬" nodes except the ones which are applied to the basic variables

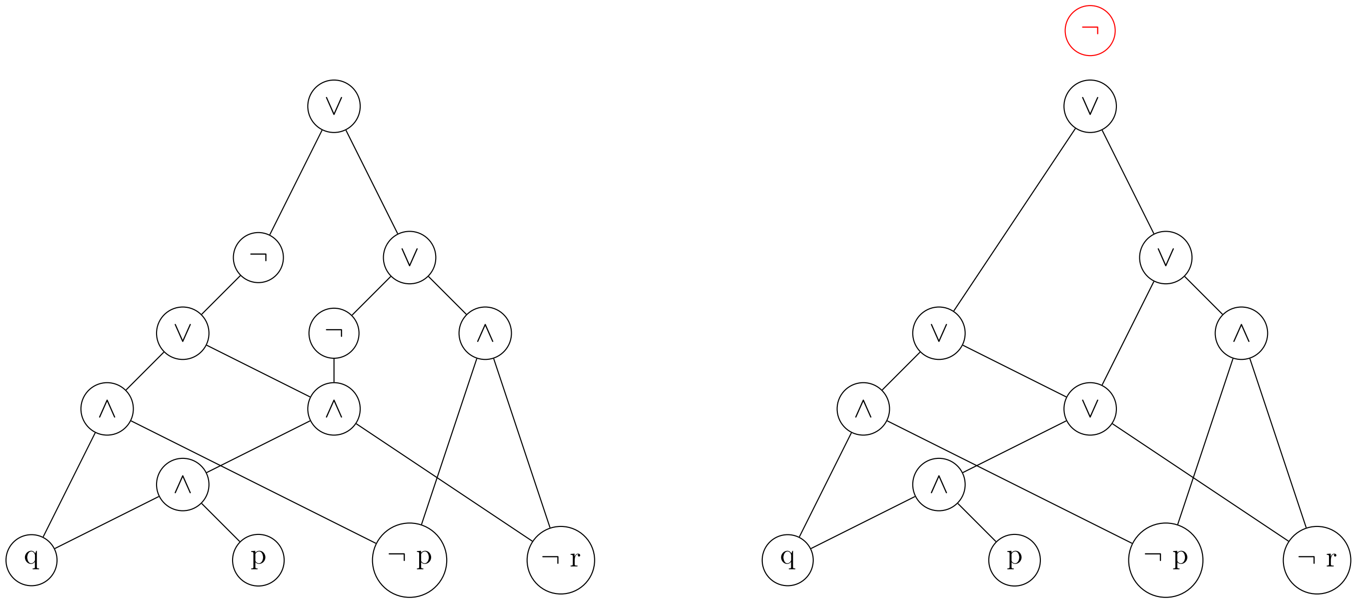


Figure 1.8: Step 2: Negate the entire DAG obtained in step 1

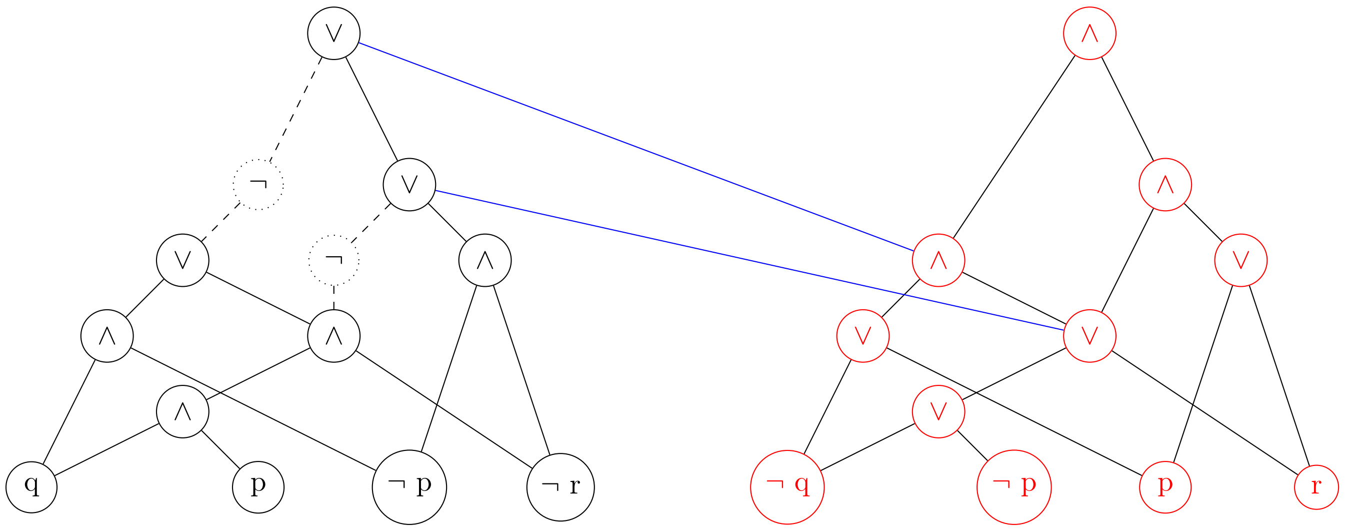


Figure 1.9: Step 3: Remove the "¬" nodes from the first DAG by connecting them to the corresponding node in the negated DAG.

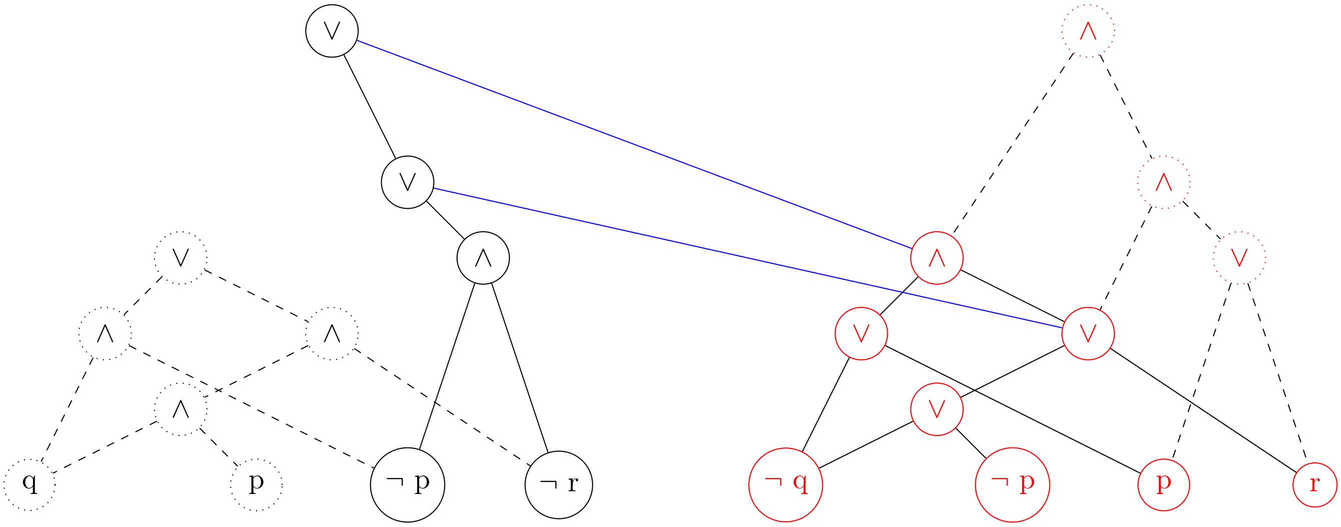


Figure 1.10: Step 4: Remove all the redundant nodes.

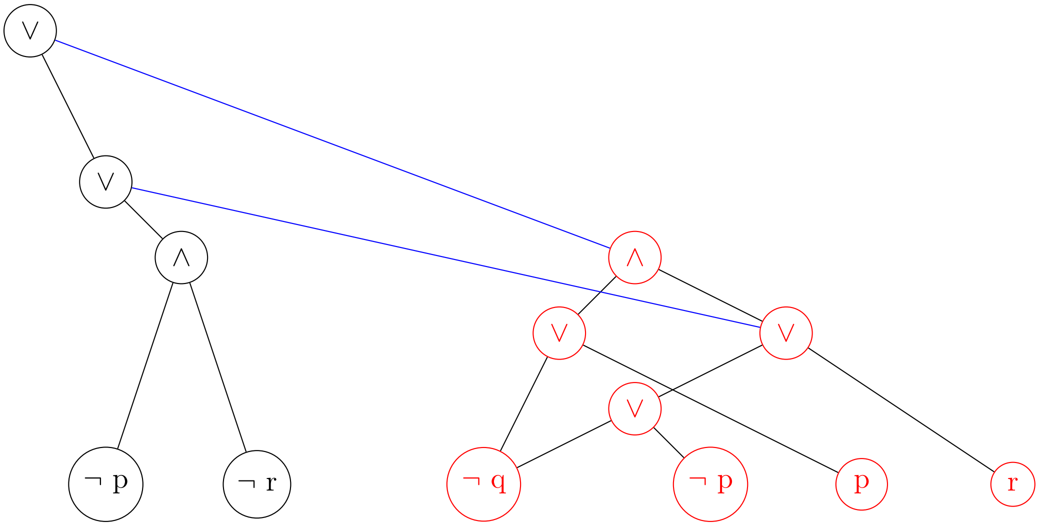


Figure 1.11: NNF - DAG

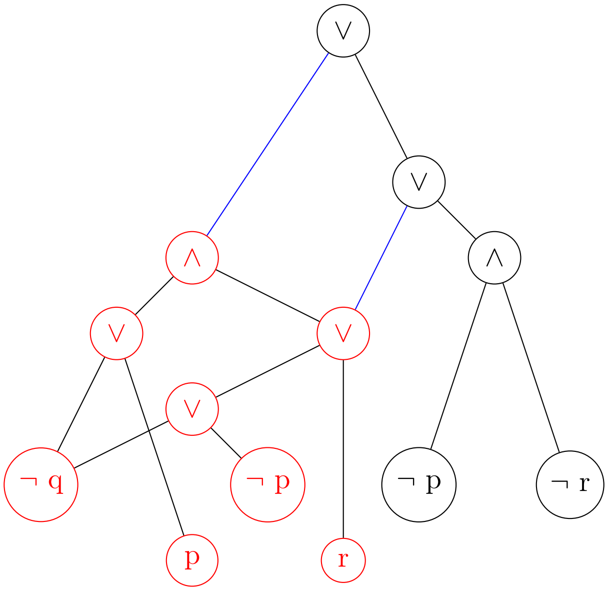


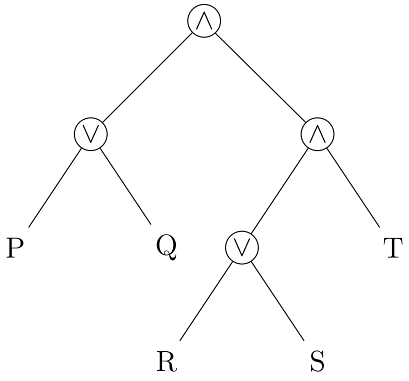
Figure 1.12: The NNF DAG (rearranged)

NOTE : The size of the NNF - DAG obtained using the above algorithm is atmost two times the size of the given DAG. Hence we have an O(N) formula for converting any arbitrary DAG to a semantically equivalent NNF - DAG.

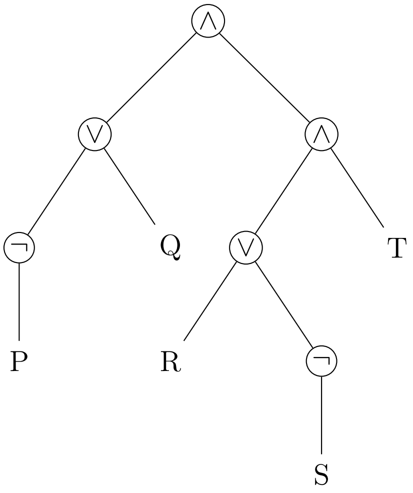
1.11 Conjunctive Normal Forms

A formula is in conjunctive normal form or clausal normal form if it is a conjunction of one or more clauses, where a clause is a disjunction of variables.
Examples:

- Parse Tree for a formula in CNF



- Parse Tree for the formula $(\neg(p \wedge \neg q) \wedge (\neg(\neg r \wedge s)) \wedge t)$ in NNF



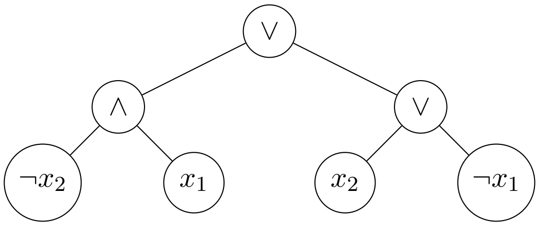
Some Important Terms

- **LITERAL :**
  - Variable or it's complement.
  - Example : p , ¬ p , r , ¬ q

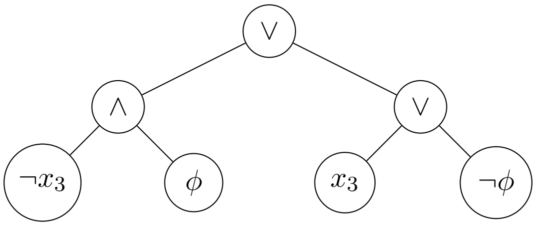
- **CLAUSE :**
  - A clause is a disjunction of literals such that a literal and its negation are not both in the same clause, for any literal
  - Example : p ∨ q ∨ (¬ r) , p ∨ (¬ p) ∨ (¬ r) -> Not Allowed
- **CUBE :**
  - a Cube is a conjunction of literals such that a literal and its negation are not both in the same cube, for any literal
  - Example p ∧ q ∧ (¬ r) , p ∧ (¬ p) ∧ (¬ r) -> Not Allowed
- **CONJUNCTIVE NORMAL FORM(CNF)**
  - A propositional formula is said to be in Conjunctive Normal Form (CNF) if it is a conjunction of clauses
  - Product of Sums
  - Example (p ∨ q ∨ (¬ r)) ∧ (q ∨ r ∨ (¬ s) ∨ t)
- **DISJUNCTIVE NORMAL FORM(DNF)**
  - A formula is said to be in Disjunctive Normal Form (DNF) if it is a disjunction of cubes.
  - Sum of Products
  - Example (p ∧ q ∧ (¬ r)) ∨ (q ∧ r ∧ (¬ s) ∧ t)

Given a DAG of propositional logic formula with only ∨ , ∧ and ¬ nodes , can we efficiently get a DAG representing a semantically equivalent CNF/DNF formula ?

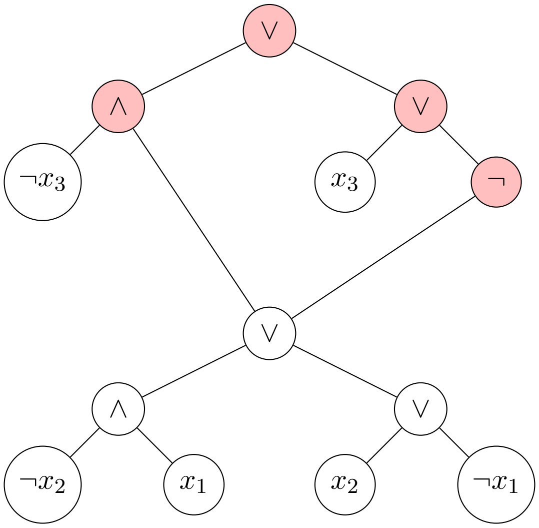
Tutorial 2 Question 2:
The Parity Function can be expressed as (((....($x_1 \oplus x_2$) $\oplus\ x_3$) .... $\oplus\ x_n$)
The Parse Tree for $x_1 \oplus x_2$ is



Now consider $\phi = x_1 \oplus x_2$ then Parse tree for ($x_1 \oplus x_2$) $\oplus\ x_3$ will be



Now putting $\phi = x_1 \oplus x_2$ in the tree, we get the following DAG representation



We notice that on adding x_i we are adding 4 nodes. Hence, the **size of DAG of the parity function is atmost 4n**. And we have already shown that size of NNF-DAG is atmost 2 times the size of DAG. So, the **size of the semantically equivalent NNF-DAG is atmost 8n**.

Also , in the Tutorial Question we have proved that the DAG **size of the semantically equivalent CNF/DNF formula is atleast** 2^{n-1}.

NNF -> CNF/DNF (Exponential Growth in size of the DAG)

1.12 Satisfiability and Validity Checking

It is **easy to check validity of CNF**. Check for every clause that it has some p , ¬ p. If **there is a clause which does not have both(p , ¬ p), then the Formula is not valid** because we can always assign variables in such a way that makes this clause false.
Example - If we have a clause p ∨ q ∨ (¬ r) , we can make it 0 with assignments p = 0, q = 0 and r = 1.
And if both a variable and its conjugate are present in a clause then since p ∨ ¬ p is valid and
$1 \vee \phi = 1$ for any ϕ.So, the clause will be always valid.
Hence , we have an **O(N) algorithm to check the validity of CNF** but the price we pay is conversion to it(which is exponential).
It is hard to check validity of DNF because we will have to find an assignment which falsifies all the cubes.
What is meant by satisfiability?
Given a Formula , is there an assignment which makes the formula valid.
It is **easy to check satisfiability of DNF**. Check for every cube that it has some p , ¬ p. If there is a cube which does not have both(p , ¬ p), then the Formula is satisfiable because we can always assign variables in such a way that makes this cube true and hence the entire formula true. Overall, **We just need to find a satisfying assignment for any one cube.**
Hard to check satisfiability using CNF. We will have to find an assignment which simultaneosly

satisfy all the cubes.

NOTE : A formula is valid if it's negation is not satisfiable. Therefore , we can convert every validity problem to a satisfiability problem. Thus, it suffices to worry only about satisfiability problem.

1.13 DAG to Equisatisfiable CNF

Claim: Given any formula, we can get an equisatisfiable formula in CNF of linear size efficiently.
Proof: Let's consider the DAG $\phi(p, q, r)$ given below:

1. Introduce new variables $t_1, t_2, t_3, \ldots., t_n$ for each of the nodes. We will get an equisatisfiable formula $\phi'(p, q, r, t_1, t_2, t_3, \ldots., t_n)$ which is in CNF.
2. Write the formula for ϕ' as a conjunction of subformulas for each node of form given below:

$$\begin{aligned}\phi' =&(t_1 \iff (p \land q)) \land \\ &(t_2 \iff (t_1 \lor \neg r)) \land \\ &(t_3 \iff (\neg t_2)) \land \\ &(t_4 \iff (\neg p \land \neg r)) \land \\ &(t_5 \iff (t_3 \lor t_4)) \land \\ &t_5\end{aligned}$$

3. Convert each of the subformula to CNF. For the first node it is shown below:

$$\begin{aligned}(t_1 \iff (p \land q)) =&(\neg t_1 \land (p \land q)) \land (\neg(p \land q) \lor t_1) \\ =&(\neg t_1 \lor p) \land (\neg t_1 \lor q) \land (\neg p \lor \neg q \lor t_1)\end{aligned}$$

4. For checking the equisatisfiabiliy of ϕ and ϕ': Think about an assignment which makes ϕ true then apply that assignment to ϕ'.

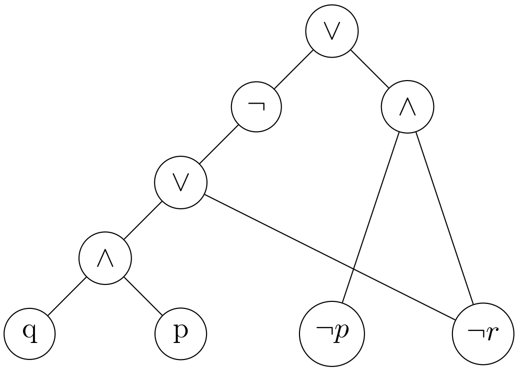
1.14 Tseitin Encoding

The Tseitin encoding technique is commonly employed in the context of Boolean satisfiability (SAT) problems. SAT solvers are tools designed to determine the satisfiability of a given logical formula, i.e., whether there exists an assignment of truth values to the variables that makes the entire formula true.

The basic idea behind Tseitin encoding is to introduce additional auxiliary variables to represent complex subformulas or logical connectives within the original formula. By doing this, the formula can be transformed into an equivalent CNF representation.

Say you have a formula $Q(p, q, r, \dots)$. Now we can use and introduce auxiliary variables $t_1, t_2, \dots$ to make a new formula $Q'(p, q, r, \dots t_1, t_2, \dots)$ using Tseitin encoding which is equisatisfiable as Q. Q' is equisatisfiable as Q but not sematically equivalent. Size of $Q'$is linear in size of Q.

Lets take an example to understand better. Consider the formula $(\neg((q \wedge p) \vee \neg r)) \vee (\neg p \wedge \neg r)$



We define a new equisatisfiable formula with auxiliary variables t_1, t_2, t_3, t_4 and t_5 as follows:

$$(t_1 \iff p \wedge q) \wedge (t_2 \iff t_1 \vee \neg r) \wedge (t_3 \iff \neg t_2) \wedge (t_4 \iff \neg p \wedge \neg r) \wedge (t_5 \iff t_3 \vee t_4) \wedge (t_5)$$

1.15 Towards Checking Satisfiability of CNF and Horn Clauses

A Horn clause is a disjunctive clause (a disjunction of literals) with at most one positive literal.
A Horn Formula is a conjuction of horn clauses, for example:

$$(\neg x_1 \vee \neg x_2 \vee x_3) \wedge (\neg x_4 \vee x_5 \vee \neg x_3) \wedge (\neg x_1 \vee \neg x_5) \wedge (x_5) \wedge (\neg x_5 \vee x_3) \wedge (\neg x_5 \vee \neg x_1)$$

Now we can convert any horn clause to an implication by using disjunction of the literals that were in negation form in the horn clause on left side of the implication and the unnegated variable on the other side of the implication. So all the variables in all the implications will be unnegated.
So the above equation can be translated as follows.

$$x_1 \wedge x_2 \implies x_3$$

$$x_4 \wedge x_3 \implies x_5$$

$$x_1 \wedge x_5 \implies \bot$$

$$x_5 \implies x_3$$

$$\top \implies x_5$$

Now we try to find a satisfying assignment for the above formula.
From the last clause we get $x_5 = 1$, now the fourth clause is $\top \implies x_3$.
Then from the fourth clause we get that $x_3 = 1$.
Now in the remaining clauses none of the left hand sides are reduced to $\top$.
Hence, we set all remaining variables to 0 to get a satisfying assignment.

Algorithm 1: HORN Algorithm

```
Function HORN($\phi$):
    foreach occurrence of $\top$ in $\phi$ do
        mark the occurrence
    while there is a conjunct $P_1 \wedge P_2 \wedge \ldots \wedge P_{k_i} \to P'$ in $\phi$ do
        // such that all $P_j$ are marked but $P'$ isn't
        if all $P_j$ are marked and $P'$ isn't then
            mark $P'$
    if $\perp$ is marked then
        return'unsatisfiable'
    else
        return'satisfiable'
```

Complexity:
If we have n variables and k clauses then the solving complexity will be $O(nk)$ as in worst case in each clause you search for each variable.

1.16 Counter example for Horn Formula

In our previous lectures, we delved into the Horn Formula, a valuable tool for assessing the satisfiability of logical formulas.

1.16.1 Example

We are presented with a example involving conditions that determine when an alarm (a) should ring. Let's outline the given conditions:

1. If there is a burglary (b) in the night (n), then the alarm should ring (a).
2. If there is a fire (f) in the day (d), then the alarm should ring (a).
3. If there is an earthquake (e), it may occur in the day (d) or night (n), and in either case, the alarm should ring (a).
4. If there is a prank (p) in the night (n), then the alarm should ring (a).
5. Also it is known that prank (p) does not happen during day (d) and burglary (b) does not takes place when there is fire (f).

Let us write down these implications

$$b \wedge n \Rightarrow a \quad f \wedge d \Rightarrow a \quad e \wedge d \Rightarrow a$$
$$e \wedge n \Rightarrow a \quad p \wedge n \Rightarrow a \quad d \wedge n \Rightarrow \perp$$
$$b \wedge f \Rightarrow a \quad p \wedge d \Rightarrow \perp$$

Now we want to examine the possible behaviour of this systen under the assumption that alarm rings during day. For this we add two more clauses:

$$\top \Rightarrow a \quad \top \Rightarrow d$$

This directly gives us that a, d have to be true, what about the rest? We can see that setting all the remaining variables to false is a satisfying assignment for this set of formulae.
Hence we have none of prank, earthquake, burglary or fire and hence alarm should not ring.
This means that our formulae system is incomplete.
To achieve this, we try to introduce new variables Na (no alarm), Nf (no fire), Nb (no burglary), Ne (no earthquake), and Np (no prank).
We extend the above set of implications in a natural way using these formulae:

$$a \wedge Na \Rightarrow \bot \quad b \wedge Nb \Rightarrow \bot \quad f \wedge Nf \Rightarrow \bot \quad e \wedge Ne \Rightarrow \bot \quad p \wedge Np \Rightarrow \bot$$

$$Nb \wedge Nf \wedge Ne \wedge Np \Rightarrow Na$$

All the implications will hold true for the values $b = p = e = f = Nb = Ne = Nf = Np = 0$.
Here, we are getting $b = Nb$, which is not possible, hence, we need the aditional constraint that $b \iff Nb$. But on careful examination we see that this cannot be represented as a horn clause.
Therefore, it becomes necessary to devise an alternative algorithm suited for evaluating satisfiability.

1.17 Davis Putnam Logemann Loveland (DPLL) Algorithm

This works for more general cases of CNF formulas where it need not be a Horn formula. Let us first discuss techniques and terms required for our Algorithm.

- **Partial Assignment (PA) :** It is any assignment of some of the propositional variables.
  Ex. $PA = \{x_1 = 1, x_2 = 0\}$; $PA = \{\}$, etc.

- **Unit Clause :** It is any clause which only has one literal in it. Ex. $.. \wedge (\neg x_5) \wedge ..$
  Note: If any Formula has a unit clause then the literal in it has to be set to true.

- **Pure Literal :** A literal which doesn't appear negated in any clause. Say a propositional variable x appear only as $\neg x$ in every clause it appears in., or say y appears only as y in every clause.
  Note: If there is a pure literal in the formula, it does not hurt any clause to set it to true. All the clauses in which this literal is present will become true immediately.

We will now utilize every techniques we learnt to simplify our formula. First we check if our formula has unit clause or not. If yes then we assign the literal in that clause to be 1. Note, $\varphi[l = 1]$ is the formula obtained after setting $l = 1$ everywhere in the formula. We also search for pure literals. If we find a pure literal then we can simply assign it 1 (or 0 if it always appears in negated form) and proceed, this cannot harm us (cause future conflicts) due to the definition of Pure Literal. If we do not have any of these then we have only one option left at the moment which is try and error.

We assign any one of the variable in the formula a value which we choose by some way (not described here). Then we go on with the usual algorithm until we either get the whole formula to be true or false. At this step we might have to backtrack if the formula turns out to be false. If it is true we can terminate the algorithm.

Note: Our algorithm **can be as worse as a Truth Table** as we are trying every assignment. But as we are applying additional steps, after making a decision there are high chances that we get a unit clause or a pure literal.

Now as we have done all the prerequisites let us state the algorithm.

Algorithm 2: SAT(φ, PA)

```
{// These are the base cases for our recursion}
if $\varphi = \top$ then
  return SAT(sat, PA)
else if $\varphi = \bot$ then
  return SAT(unsat, PA)
else if $C_i$ is a unit clause (literal $l$) and $C_i \in \varphi$ then
  {//This step is called Unit Propagation}
  return SAT($\varphi[l = 1], \mathrm{PA} \cup \{l = 1\}$) {// Here recursively call the algorithm on the simplified}
                                                          //formula $\varphi[l = 1]$
else if $l$ is a pure literal and $l \in \varphi$ then
  {//This step is called Pure Literal Elimination}
  return SAT($\varphi[l = 1], \mathrm{PA} \cup \{l = 1\}$)
else
  {//This step is called Decision Step}
  $x \leftarrow$ choose_a_var($\varphi$)
  $v \leftarrow$ choose_a_value($\{0, 1\}$)
  if SAT($\varphi[x = v], \mathrm{PA} \cup \{x = v\}$).status = sat then
    return SAT(sat, $\mathrm{PA} \cup \{x = v\}$)
  else if SAT($\varphi[x = 1 - v], \mathrm{PA} \cup \{x = 1 - v\}$).status = sat then
    return SAT(sat, $\mathrm{PA} \cup \{x = 1 - v\}$)
  else
    return SAT(unsat, PA)
  end if
end if
```

Question Can the formula be a horn formula after steps 1 and 2 can't be applied anymore? i.e. If our formula does not have any unit clause or pure literal can it be a horn formula?

ANS. Yes. Here is an example

$$(a \lor \neg b) \land (\neg a \lor b)$$

1.18 DPLL in action

1.18.1 Example

Consider the following clauses, for which we have to find whether all can be satisfied for a variable mapping or not using DPLL algorithm-

$$\begin{array}{cccc} C_1 : (\neg P_1 \vee P_2) & C_2 : (\neg P_1 \vee P_3 \vee P_5) & C_3 : (\neg P_2 \vee P_4) & C_4 : (\neg P_3 \vee P_4) \\ C_5 : (P_1 \vee P_5 \vee \neg P_2) & C_6 : (P_2 \vee P_3) & C_7 : (P_1 \vee P_3 \vee P_7) & C_8 : (P_6 \vee \neg P_5) \end{array}$$

Lets make two possible decision trees for these clauses.
PLE - Pure literal elimination UP - Unit Propagation D - Decision

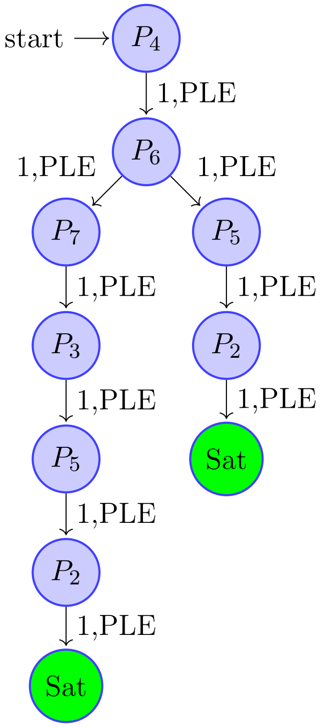


Figure 1.13: DPLL

The following is the decision tree if we remove the point 2 of DPLL. Note the increase in number of operations.

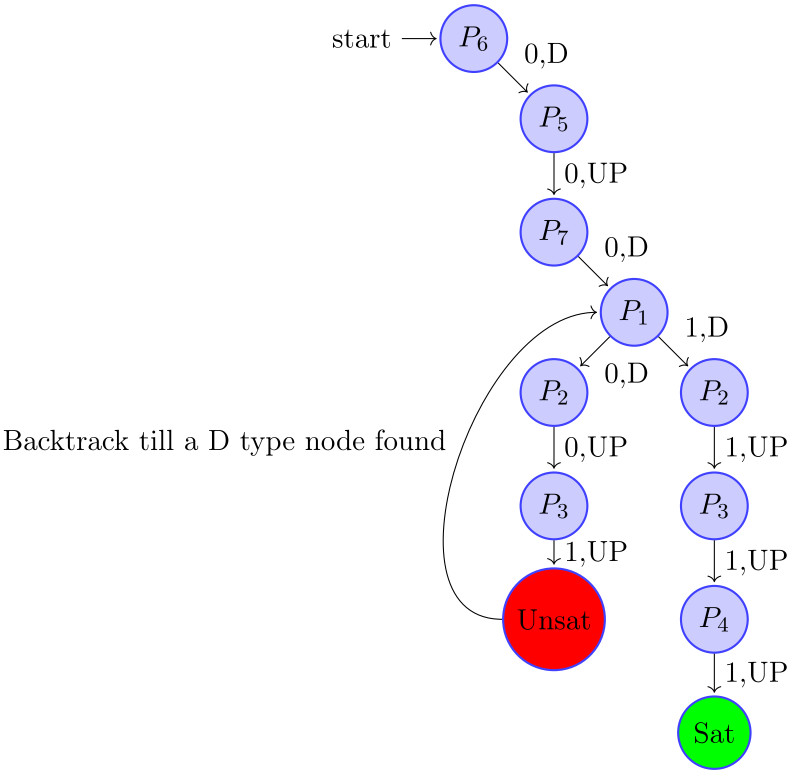


Figure 1.14: DPLL without point 2

1.19 Applying DPLL Algorithm to Horn Formulas

Let us apply DPLL algorithm to Horn Formula
If there are no variables on LHS, it becomes a Unit Clause *i.e.* $\top \rightarrow x_i$ and is equivalent to (x_i).
Horn Method can be viewed in terms of DPLL algorithm as following:

- Apply Unit Propagation until you can't apply.
- After that, set all remaining variables to 0.

Advantage of Horn's method is after all possible Unit Propagations are done, it sets all remaining variables to 0, but in DPLL we need to go step by step for each remaining variable.

But Horn's method can only be applied in a special case, moreover, in Horn's method we only figure out which variables to set true as opposed to DPLL which can figure out whether variable needs to be set to true or false via the Pure Literal Elimination.

1.20 DPLL on Horn Clauses

We shall quickly investigate what happens when we feed in **Horn clauses** to the **DPLL** (Davis-Putman-Logemann-Loveland) algorithm.
Consider the following,

- Consider the first step in solving for the satisfiability of a given set of Horn clauses in implication form where,
  - If the LHS of an implication is true we set the literal on the RHS of the implication to be true in all its implications.
  - Repeating the above step sets all essential variable which are to be set to 1, true.

  This step is equivalent to the first two steps of the DPLL algorithm,

  - Satisfy unit literal clauses by assignment.
  - Recomputing the formula for the above bullet.
  - Repeating this procedure until no unit clause is left.

  The above steps in the two different schemes do the same are essentially doing the same thing. Now if the given clauses were Horn, we know that putting all the remaining variables false is a satisfying assignment. This means if our DPLL algorithm preferentially assigns 0 to each decision, the procedure thus converges to the method for checking the satisfiability for Horn formulae.

1.21 Rule of Resolution

This is yet another powerful rule for inference. Let us first jot down the rule here:

$$\frac{(a_1 \lor a_2 \lor a_3 \cdots \lor a_n \lor x) \land (b_1 \lor b_2 \lor b_3 \cdots \lor b_m \lor \neg x)}{(a_1 \lor a_2 \lor a_3 \cdots \lor a_n \lor b_1 \lor b_2 \lor b_3 \cdots \lor b_m)} \text{ resolution}$$

However intuitive it may look this rule poses as a powerful tool to check the satisfiability of logical formulae, we can reason out an algorithm to check the satisfiability of a formula (CNF) as follows:
Let us first define a formula to be **unresolved** if there exists a literal and its negation in the formula (they cannot be in the same clause by the definition of a clause). If a formula is **resolved** (i.e., not unresolved) then it is satisfiable ('**SAT**'), as the variables which appear in their negated form we assign false, and the other variables to true.

Let $\mathfrak{C}$ be the set of clauses for a give CNF.

1. If $\mathfrak{C}$ contains tautologies we can drop them, if $\mathfrak{C}$ becomes empty upon dropping the tautologies, we mark the given CNF **SAT**. (However by definition, clauses by themselves cannot be tautologies.)
2. As the formula is unresolved, we can apply the resolution rule, this gives us a new clause.
3. If the formula so formed is the empty clause, we deem the formula to be **UNSAT** otherwise check if the formula is resolved, if not from repeat step 1.

Before rationalizing the soundness of the above sequence of steps let us first see an example.
An Example: Consider $\mathfrak{C} = \{C_1, C_2, C_3\}$ as given below:

- $C_1 :=\ \neg p_1 \vee p_2\ (\ p_1 \implies p_2)$
- $C_2 :=\ p_1 \vee \neg p_2\ (\ p_2 \implies p_1)$
- $C_3 :=\ \neg p_1 \vee \neg p_2\ (\ p_1 \wedge p_2 \implies \bot)$

Then, a dry run of the above method would look like:

1. Since both p_1 and p_2 appear in negated and un-negated form, we apply resolution on C_1 and C_2, which generates C_4, as follows:
$$\frac{(\neg p_1 \vee p_2)\quad (\ p_1 \vee \neg p_2)}{(\neg p_1 \vee p_1)}\ \text{resolution}$$
2. Once again we apply resolution on C_3 and C_4 (this is not really a clause by definition, one can choose to drop the tautologies as soon as encountered),
$$\frac{(\neg p_1 \vee \neg p_2)\quad (\neg p_1 \vee p_1)}{(\neg p_2 \vee \neg p_1)}\ \text{resolution}$$
3. The clause that we have got is now resolved and thus, our formula is satisfiable.

1.21.1 Completeness of Resolution for Unsatisfiability of CNFs

Here as we claimed above, given any CNF, if it is unsatisfiable the remainder of continuous resolutions is the empty clause which we deem **UNSAT**. We here prove the consistency of the claim.
For this we employ the method of mathematical induction, we induct on the number of propositional variables in our CNF. Let $p_1, p_2 \ldots p_m$ be our propositional variables.
Base: $n = 1$ An unsatisfiable CNF in a single literal **must** contain the clauses (p_1) and $(\neg p_1)$ which upon resolution gives us **( )** the empty clause hence we raise **UNSAT**.
Inductive Hypothesis: Assume that our claim holds $\forall m \leq n-1$ we now show that it holds from $m = n$ as follows,

- Remove tautologies from $\mathfrak{C}$, the set of all clauses.
- Choose a literal p_i such that both p_i and $\neg p_i$ both appear in the CNF. (If no such literal exists the formula is resolved as defined earlier and has a satisfying assignment). Apply resolution repeatedly as long as the same p_i satisfies this condition.

- If the CNF contains ( ), in which case we raise **UNSAT**, otherwise
  - If p_i **vanishes** from the CNF, then calling our hypothesis, we can raise **UNSAT** as the equivalent form that we have got must be unsatisfiable independent of the value of the vanished literal as the initial formula was unsatisfiable.
  - If p_i **exists** in one of negated or un-negated forms. In which case we repeat the procedure. This time the number of available **pairs** has reduced by 1 as p_i cannot be selected again.
- As the selection step can take place at most n times, (as a new **pair**(as in bullet 2) cannot be generated in the CNF by resolution operations), Consider the case with a **pair** available for every literal then the procedure must conclude **UNSAT** in n steps otherwise at the end of n steps we have no **pair**s, which ensures a satisfying assignment for the formula.

Broadly speaking, what we are showing is that upon repeated resolution of an unsatisfiable CNF, if ( ) has not been encountered, the number of propositional variables must decrease.

Chapter 2

DFAs and Regular Languages

Consider a set of formulas made up of propositional variables $\{x_1, x_2 \ldots, x_n\}$, $\phi(x_1, x_2 \ldots, x_n)$. We defined the set $L \subseteq \{0,1\}^n$, the **language** defined by the formula as the set of strings which form a **satisfying assignment** for the formula ϕ.
Basically using **Propositional Logic**, we were able to represent a large set of **finite length** strings having some properties in a **compact form**. This leads us to a question what about string of arbitrary length having some properties. How do we formulate them?
The answer to this is **Automata**: A way to formulate arbitrary length strings in a compact form.

2.1 Definitions

- **Alphabet**: A **finite, non-empty** set of symbols called **characters**. We usually represent an alphabet with Σ. For example $\Sigma = \{a, b, c, d\}$.
- **String**: A **finite** sequence of letters form an alphabet. An important thing to note here is that even though the alphabet may contain just 1 character, it can form **countably infinite** number of strings, each of which are **finite**. In this course we only deal with **finite strings** over a **finite alphabet**.
- **Concatenation Operation (·)** We can start from a string , take another string an as the name suggests concatenate them to form another string:

$$\begin{aligned} a \cdot b &\neq b \cdot a \qquad \text{Not Commutative} \\ (a \cdot b) \cdot c &= a \cdot (b \cdot c) \qquad \text{Associative} \end{aligned}$$

- **Identity Element** The algebra of the strings defined over the concatenation operator has the identity element : ε **: empty string**:

$$\sigma \cdot \varepsilon = \varepsilon \cdot \sigma = \sigma$$

  Note the the empty strings remains same for strings of all alphabets.
- **Language** A subset of all finite strings on Σ. This set doesn't have to be finite even though the strings are of finite length.
  Note that a set of all finite strings of Σ is countably infinite (cardinality: $\mathbb{N}$), so the number of languages of Σ is uncountably infinite (cardinality: $2^{\mathbb{N}}$).

- Σ^* is defined to be the set of all finite strings on Σ, including ε. Note that $\Sigma^* = \bigcup_{k\geq 0}\Sigma^k$, where Σ^k is the set of all strings on Σ with exactly k letters. Note that we can prove that the number of strings for Σ^* are countably finite by representing each string as a unique number in base $(n+1)$ system , where $|\Sigma| = n$, we can get an injection to natural numbers.

2.2 Deterministic Finite Automata

Generalization of Parity function: Lets go back to the question we asked first. Suppose we are given a string of arbitrary length and don't know the length of the string. This can be done by propositional logic. So we need a new formalism to represent sets of strings with any length. we want to develop a mechanism where we are given the bits of the string one by one, and I don't know when it will stop. So I must be ready with the answer each time a new bit arrives.

The solution to this lies in our discussion during the first lecture. I will record just one bit of information: whether I have received an even or odd number of 1s till now. Every time I receive a new bit, I will update this information: if it's a 0, I won't do anything, and if it's a 1, I will change my answer from even to odd, or vice-versa.

States: These are nodes which contain relevant summary of what we have seen so far

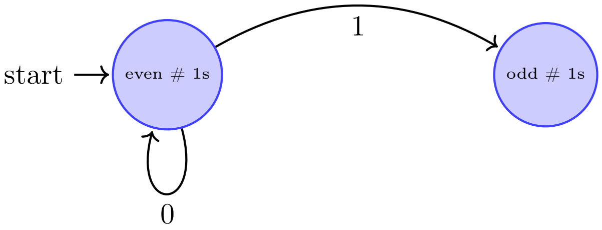
In our case we want to know whether there were even or odd number of 1s.

even # 1s
odd # 1s

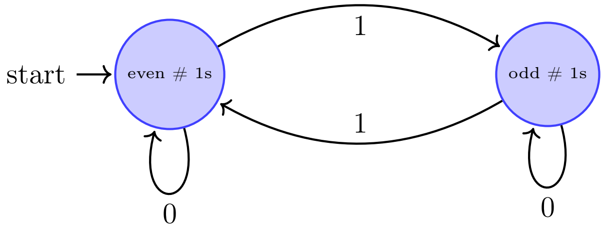
Where do we start from ? When I have seen nothing there are even number of 1s.

start
even # 1s
odd # 1s

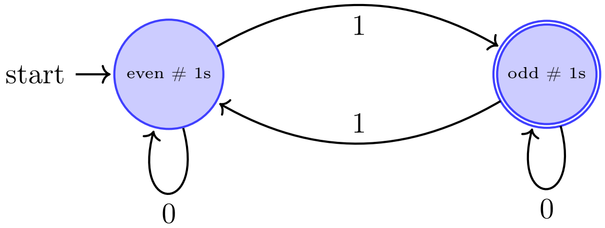
Now, suppose I receive a 0, I would remain in the same state, but if I get a 1 , the parity changes.



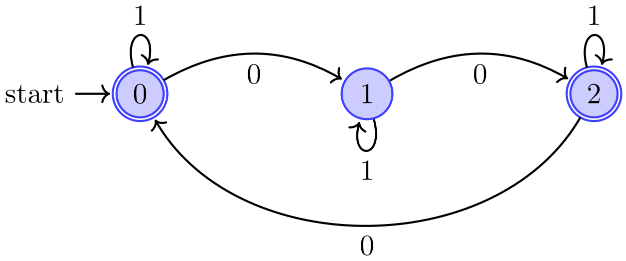
Now, if I am in the second state, if I get a 1 I will change states, and if I get a 0, parity is unchanged to I remain in the same state:



So when do we know that our string we have seen till now belongs to some language or not, we know that by marking some states as **accepting states**: usually represented by double circles, if we end up on this state, the string recieved till now belongs to out language: *is accepted*.



Such a formalism with **finite** states is known as **Finite Automata**.
Further, if for every string in Σ^*, there exists a unique path we will follow in the automata, such automata are also known **Deterministic Finite Automata (DFA)**.
Example
$\Sigma = \{0, 1\}$, let $L = \{w | w \in \Sigma^*, \text{ no. of 0's is a 0 mod 3 or 2 mod 3}\}$



2.3 DFA Design - Example 1

Draw a Deterministic Finite Automaton (DFA) for $L := \{w \in \{a, b\}^*\} : 2 \text{ divides } n_a(w) \text{ and } 3 \text{ divides } n_b(w)\}$. Here $n_a(w)$ stands for the number of a's in w, and $n_b(w)$ stands for the number of b's in w. For example, $n_a(abbaab) = n_b(abbaab) = 3$. Therefore, $ababbaa \in L$ but $aabababa \notin L$.

In certain scenarios, expressing a language solely through propositional logic becomes impractical, particularly when the length of the strings is unknown or variable. For instance, consider above example. In this case, the length n of the string is not explicitly provided, making it challenging to construct a propositional logic expression directly. Propositional logic typically operates on fixed, predetermined conditions or patterns within strings, which cannot accommodate variable lengths.
However, deterministic finite automata (DFAs) offer a suitable alternative for such situations. DFAs

are well-suited for languages where the structure and properties depend on the characters within the string rather than on fixed string lengths. By employing states and transitions based on input characters, DFAs can effectively recognize languages with variable-length strings and complex patterns, making them a more appropriate choice when string length is not predetermined.
We will denote Σ as the set of alphabets.
$\sum = \{a, b\}$
$L = \{\omega \in \sum^* : n_a(\omega)\, is\, divisible\, by\, 2\, and\, n_b(\omega)\, is\, divisible\, by\, 3\}$

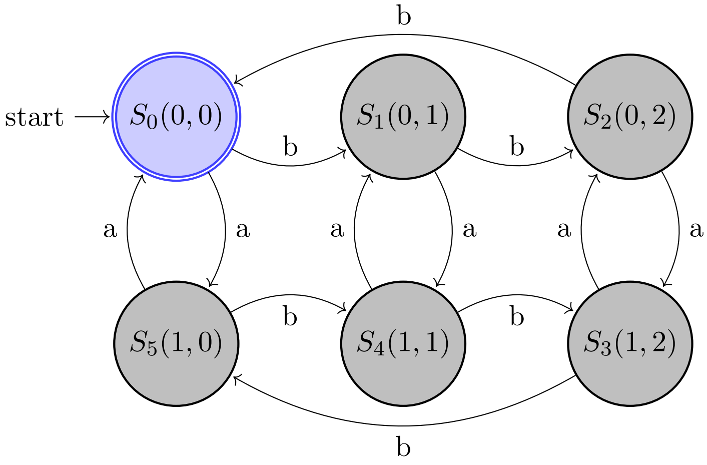


Figure 2.1: DFA for above task

S_0 : $\mathbf{n}_a(\mathbf{w})\%2 = 0$ and $\mathbf{n}_b(\mathbf{w})\%3 = 0$
S_1 : $\mathbf{n}_a(\mathbf{w})\%2 = 0$ and $\mathbf{n}_b(\mathbf{w})\%3 = 1$
S_2 : $\mathbf{n}_a(\mathbf{w})\%2 = 0$ and $\mathbf{n}_b(\mathbf{w})\%3 = 2$
S_3 : $\mathbf{n}_a(\mathbf{w})\%2 = 1$ and $\mathbf{n}_b(\mathbf{w})\%3 = 2$
S_4 : $\mathbf{n}_a(\mathbf{w})\%2 = 1$ and $\mathbf{n}_b(\mathbf{w})\%3 = 1$
S_5 : $\mathbf{n}_a(\mathbf{w})\%2 = 1$ and $\mathbf{n}_b(\mathbf{w})\%3 = 0$

2.4 DFA Design - Example 2

We will look at another example now.

$$\Sigma = \{a, b\}$$

$$L = \{\mathbf{w} \in \Sigma^* \mid \mathbf{n}_{ab}(\mathbf{w}) \quad = \quad \mathbf{n}_{ba}(\mathbf{w})\}$$

$$w \quad = \underline{a\overline{b}}\overline{a}\underline{a\overline{ba}b} \quad (\mathbf{n}_{ab}(\mathbf{w}) = 3 \ \& \ \mathbf{n}_{ba}(\mathbf{w}) = 2)$$

If we try to cleverly convert the problem into a simpler one, we will observe that $\mathbf{n}_{ab}(\mathbf{w}) = \mathbf{n}_{ba}(\mathbf{w})$ will always be true if the start and end alphabets are same (be it a or b).

So the above problem simplifies to:

$$L = \{\mathbf{w} \in \Sigma^* \mid \text{First and Last alphabet are same}\}$$

Forming an automation for this task can be done as:

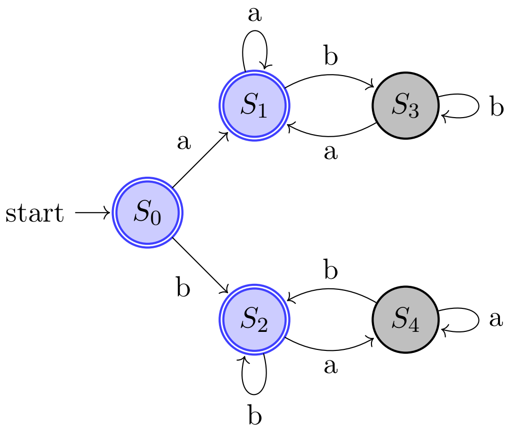


Figure 2.2: DFA: Last and First alphabets are same

Chapter 3

Non-Deterministic Finite Automata (NFA)

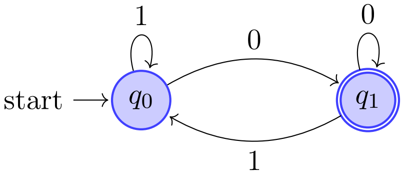
In the last few lectures, we covered the formalization of deterministic finite automata (DFA) where the transition function outputs a single state for a given input and current state. In this lecture, we will discuss non-deterministic finite automata (NDFA) where the transition function can output multiple states (or a set of states) instead.

3.1 NFA Representation

As discussed earlier, a DFA is a 5-tuple $(Q, \Sigma, q_0, \delta, F)$ where:

- Q is a finite set of all states
- Σ is the alphabet, a finite set of input symbols
- $q_0 \in Q$ is the initial state
- $\delta : Q \times \Sigma \to Q$ is the transition function
- $F \subseteq Q$ is the set of final/accepting states

For example, consider the following automaton:



It can be represented as:

$$\text{DFA} \left(\ \{q_0, q_1\}, \ \{0, 1\}, \ q_0, \ \delta, \ \{q_1\} \ \right)$$

where δ is the transition function:

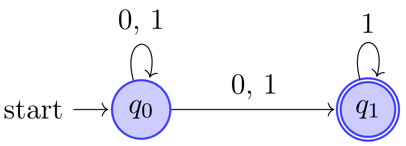
Q	Σ	Q’
q_0	0	q_1
q_0	1	q_0
q_1	0	q_1
q_1	1	q_0

3.1.1 Formalization of Non-Deterministic Finite Automata

However, in NDFA,

- $q_0 \subseteq Q$ is the set of initial states
- $\delta : Q \times \Sigma \rightarrow 2^Q$ is the transition function

Hence, consider the following non-deterministic finite automaton (A):



It can be represented as:

$$\text{NDFA}\left(\ \{q_0, q_1\},\ \{0, 1\},\ \{q_0\},\ \delta',\ \{q_1\}\ \right)$$

where δ' is the transition function: As we can see, δ' is a partial function whose output is a set of states instead of a single state.

Q	Σ	2^Q
q_0	0	$\{q_0, q_1\}$
q_0	1	$\{q_0, q_1\}$
q_1	1	$\{q_1\}$

3.1.2 NFA into action

Due to its transition function, an NDFA gives **choices for path taken** at some states for a given input string. Any string for which there exists a path from the initial state to a final state is considered to be accepted by the NDFA.

Hence, the NDFA shown above has the language $L(A) = \Sigma^* \setminus \{\epsilon\}$, i.e., the set of all strings over the alphabet Σ except the empty string.
For example, the string 011 is accepted by A as it has the following path $q_0 \xrightarrow{0} q_0 \xrightarrow{1} q_0 \xrightarrow{1} q_1$.

To determine if a string is accepted by an NDFA, we can check if the set of states reachable from the initial state by reading the string contains any final state.

Here we have: Here we have:

Initial State	Σ^*	Reachable States
q_0	0	$\{q_0, q_1\}$
q_0	01	$\{q_0, q_1\}$
q_0	011	$\{q_0, q_1\}$

As $\{q_0, q_1\}$ contains $q_1 \in F$, the string 011 is accepted by A. By construction, there will always be a set of choices which reach q_1 from q_0 for the input 011.

3.2 Equal expressiveness of DFA or NFA

We claim that regular languages, ie, the set of languages which can be defined by DFAs, is exactly the set of languages which can be defined by NFAs. In other words, for every NFA, there exists a DFA that accepts the same language, and vice versa.Even though NDFA gives choices at some states, it can still be represented as some equivalent DFA. Even though NDFA gives choices at some states, it can still be represented as some equivalent DFA. This implies that **NDFAs have no more expressive power than DFAs** in terms of string acceptance. However, representation in form of NDFA is much more succinct than DFA.

We can convert a NDFA to a DFA by considering the set of reachable states as states of the equivalent DFA.

3.2.1 Construction of DFA from NFA

For an NFA $(Q, \sum, Q_0, \delta, F)$, construct a DFA $(\mathcal{P}(Q), \sum, Q_0, \delta', F')$ with:

$$\delta'(G, \sigma) = \bigcup_{q \in G} \delta(q, \sigma)$$

$$F' = \{q : q \in \mathcal{P}(Q), q \cap F \neq \phi\}$$

Notice that in our new DFA, the states are labelled as subsets of the states of the NFA. This means that we have 2^n states in our DFA if the NFA had n. It is left to the reader to verify that the DFA we have defined satisfies all the requirements of a DFA.

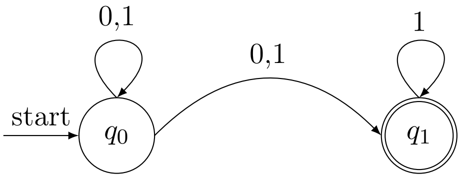
We claim that after the same characters are inputted into both the NFA and DFA, the state of the DFA is labelled the same as the set of current states of the NFA. This claim is easy to check using the definition of the δ' function.

Now, in an NFA, a string is accepted if any one of the active states at the end of the string is in the set of accepting states. Clearly, with our interpretation of the DFA, this is equivalent to being in a state that belongs to the F' we have defined.

3.2.2 Step Wise Conversion from NFA to DFA

Now we'll see how to convert a NFA to a DFA through an example.

We call the following NFA 'A'



The language depicted by this NFA is all the strings formed using {0,1} excluding the empty string(ϵ). We represent this as:

$$L(A) = \{w \in \{0,1\}^* | w \text{ is accepted by A}\}$$

that is,

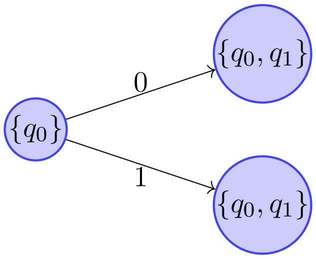
$$L(A) = \Sigma^* \backslash \{\epsilon\}$$

The transition function(δ) table looks as follows:

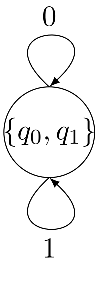
Q	Σ	2^Q
q_0	0	$\{q_0, q_1\}$
q_0	00	$\{q_0, q_1\}$
q_0	01	$\{q_0, q_1\}$

To convert this NFA to a DFA, we need to track the states that can be reached after n choices which will be a subset of 2^Q.

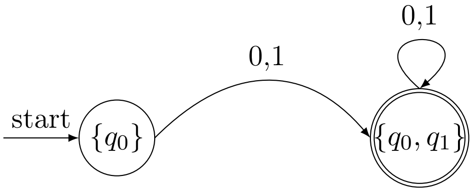
Step 1



Step 2



Final DFA



Call this DFA 'A''.
One property we've extensively used in the conversion is:

$$\textbf{If } S \subseteq Q\textbf{, then } \delta(S,0) = \bigcup_{q \in S} \delta(q,0)$$

Now, to show that the languages represented by the NFA(A) and the DFA(A') are the same, i.e, $L(A) = L(A')$, we need to show the following:

1. $L(A) \subseteq L(A')$
2. $L(A') \subseteq L(A)$

3.2.3 Proof of Equivalence

To show that the languages represented by the NFA(A) and the DFA(A') are the same, i.e, $L(A) = L(A')$, we need to show the following:

- $L(D) \subseteq L(N)$
- $L(N) \subseteq L(D)$

We can prove that the equivalent DFA D accepts exactly the same language as the original NDFA N, i.e, $L(D) = L(N)$.

3.3 Reflective Insights

Even with all the extra "powers" and behaviours NFAs can have on top of those of DFAs, they are equally expressive. This shows us that the limitation in expressiveness is not in the behaviour of state transitions but in the finiteness of states. However, due to the exponential blowup in the number of states, it is often more human-readable to express certain automata/languages through NFAs, making it a convenient representation tool.

3.4 Proof of Equivalence – Correctness

In the last lecture, we discussed the conversion of Nondeterministic Finite Automata (NFA) to its equivalent Deterministic Finite Automata (DFA)

3.4.1 Claim

We aim to demonstrate the equivalence of the languages accepted by NFA A and DFA A', denoted as $L(A)$ and $L(A')$ respectively. In other words, we want to show that $L(A) = L(A')$, where A represents the original NFA and A' represents the DFA obtained through subset construction from NFA A. Alternatively, we can establish that $L(A) \subseteq L(A')$ and $L(A') \subseteq L(A)$, which implies $L(A) = L(A')$.

3.4.2 Proof

We'll prove this claim by showing that for all $n \geq 0$ and for every word $w \in \Sigma^*$ with $|w| = n$, NFA A can reach state $q \in Q$ [1] upon reading w if and only if DFA A' reaches state $S \subseteq Q$ such that $q \in S$.
Given the condition that for all $n \geq 0$, and for every word w, where $w \in \Sigma^*$ such that $|w| = n$, we aim to demonstrate the equivalence between the NFA A's ability to reach state $q \in Q$ upon reading word w and the DFA A''s capability to reach state $S \subseteq Q$, where $q \in S$. In other words, we want to show that words in the language recognized by NFA A reach a certain state (say, a final state q), **if and only if** words in the language recognized by DFA A' reach state S (where $S \subseteq Q$, $q \in S$, and S is a final state in the equivalent DFA A').
Thus, we can establish that the set of words accepted by NFA A is equivalent to the set of words accepted by DFA A', implying $L(A) = L(A')$, thereby validating our claim.

We will demonstrate this by induction on n.

1. Base case: When $n = 0$, i.e., $|w| = 0$, it essentially means that we are at the initial state of the automaton. The initial state of the DFA A' is represented by the singleton set containing the initial states of the NFA A. So, definition of initial state of DFA A' satisfies the claim.

2. Induction hypothesis: Assume the claim holds for all $0 \leq n < k$ for some $k > 0$.

3. Inductive step: We'll show that the claim holds for $n = k$.
   Since $|w| = k$, let $w = w'.a$ (w' concatenate a) , where $w \in \Sigma^*$, $w' \in \Sigma^*$, $a \in \Sigma$, and $|w'| = k-1$.

   By hypothesis , there exists the following path in both automata :

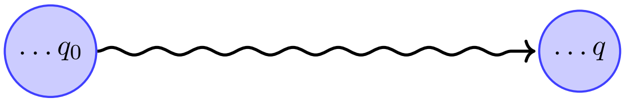
   $$\xrightarrow{\qquad\qquad\qquad w' \qquad\qquad\qquad}$$

   This basically means that there exists some path in NFA and equivalent DFA which is as follows :

   In NFA A :

   $$q_0 \rightsquigarrow q$$

   In DFA A' [2]:



   Now, if a symbol a, $a \in \Sigma$ comes, word w is formed :

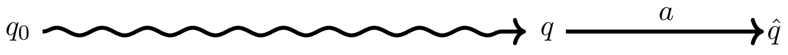
   $$\xrightarrow{\qquad\qquad\qquad w' \qquad\qquad\qquad}\xrightarrow{\quad a \quad}$$

   $$\xrightarrow{\qquad\qquad\qquad\qquad w \qquad\qquad\qquad\qquad}$$

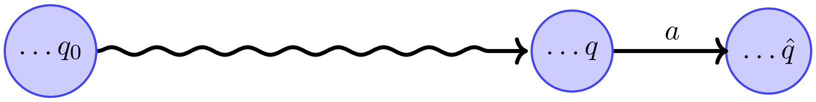
[1] Q is the set of states in our original NFA A.

[2] $\ldots q_0 \Rightarrow$ set containing initial states of the original NFA A
$\ldots q \Rightarrow \subseteq Q$ containing the state q, where Q is the set of states of the original NFA A
$\ldots \hat{q} \Rightarrow \subseteq Q$ containing the state $\hat{q}$, where Q is the set of states of the original NFA A

NFA A will reach some state $\hat{q}$



DFA A' will reach some state $\dots\hat{q}$ ($\dots\hat{q} \subseteq Q$ and contains $\hat{q}$)



This is because state $\dots q$ of DFA A' contains q, which transitions DFA A' to state $\dots\hat{q}$ (a set containing $\hat{q}$) when symbol a is encountered.
Thus, we have shown that for all $n \geq 0$, and for every word w where $w \in \Sigma^*$ such that $|w| = n$, NFA A can reach state $q \in Q$ on reading word w **AND** DFA A' reaches state $S \subseteq Q$ such that $q \in S$.

Hence, proved the condition and the claim.

Remarks:

(a) As the length of the word increases, the number of choices for state transitions in the NFA grows exponentially.

(b) If an NFA has N states, the equivalent DFA can have an exponential number of states in the worst case.

3.5 Compiler Special Case – Lexical Analyzer

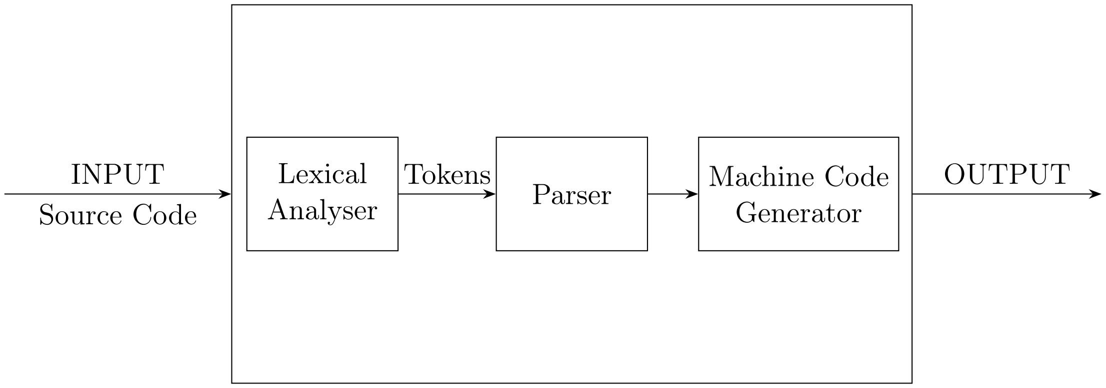


Figure 3.1: A Simple Compiler

In simpler terms, a compiler is composed of three main components: a lexical analyzer, a parser, and a code generator. Its primary function is to process a sequence of characters as input. The lexical analyzer breaks down a sequence of characters into tokens, which are then analyzed by a parser. To accomplish this, the lexical analyzer employs a NFA to determine whether a particular state can be reached in the corresponding DFA and traces a path accordingly. When faced with a long string, finding the final state can be challenging. At each step, there are multiple choices to

explore. However, as the length of the input increases, exhaustively exploring each choice becomes increasingly difficult. Converting an NFA to a DFA may result in an exponential increase in states. However, itâs important to note that not all states are necessary to reach the final state. This excessive expansion of states may lead to unnecessary complexity, creating the entire DFA when itâs not actually required.
So, the lexical analyzer determines whether there exists a path in the automaton that leads to an accepting state for a given word without constructing the equivalent DFA.
Suppose the Lexical Analyser of Compiler have to check whether a GIVEN WORD is accepted by the following NFA.

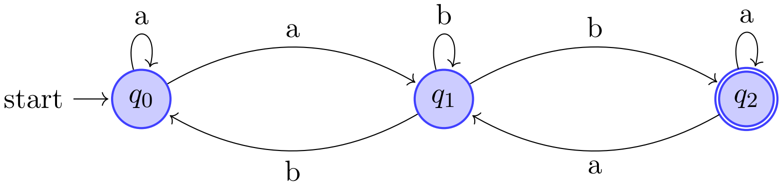


Figure 3.2: Lexical analyser Automaton

Question: How can a lexical analyzer determine whether a given word leads to an accepting state without constructing the equivalent DFA?
Answer:

$$\begin{tikzcd}
\{q_0\} \arrow[r, "a"] & \{q_0, q_1\} \arrow[r, "b"] & \{q_0, q_1, q_2\} \arrow[r, "a"] & \{q_0, q_1, q_2\} \arrow[d, "b"] \\
 & \{q_0, q_1, q_2\} & \{q_0, q_1, q_2\} \arrow[l, "b"'] & \{q_0, q_1, q_2\} \arrow[l, "a"']
\end{tikzcd}$$

Lexical Analyser can track the set of states the NFA could be in after reading each symbol of the GIVEN WORD, updating the state based on the transitions specified by the GIVEN NFA.

- If the final set of states contains at least one accepting state of the original NFA, then there exists a successful path for the given word to reach an accepting state, indicating acceptance by the given automaton.
- The time complexity for this process is $O(n \cdot k)$, where n represents the size of the automaton and k denotes the length of the word. This complexity is significantly lower than the usual exponential time complexity observed in similar processes.[3]
- Here the GIVEN WORD is accepted by the GIVEN NFA[4]

[3] Size of automaton = No. of states in Automaton + No. of Transition Arrows in Automaton
[4] q_2 is contained in the last set

3.6 NFA with ϵ−edges

A variation of NFA is NFA with ϵ edges, it may expand the language by making words accepted that were otherwise unaccepted.

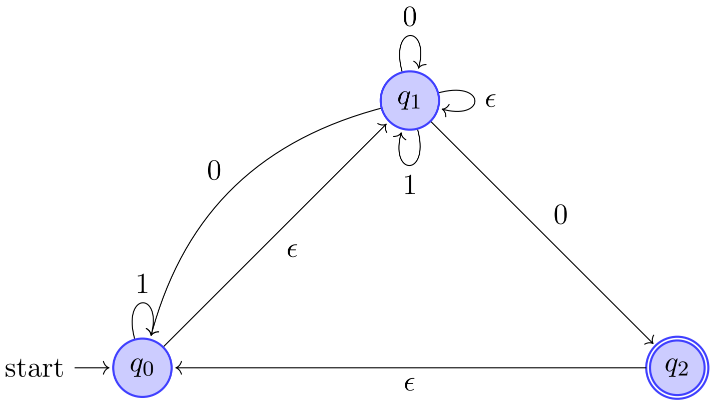


Figure 3.3: Automaton with Epsilon Edges

ϵ−edges bring non-determinism to the NFA, as you can sit on a node and take one of the ϵ−edges possible from that node to jump to another node without consuming any letter of the input.
Figure 3.3 shows how ϵ edges are used to connect states of an automaton for free(without consuming any letter from input), we can see that $10 \notin L$ without the ϵ edge between q_0 and q_1, but with the presence of this ϵ edge $10 \in L$.
ϵ−edge also allows us to connect two automatons. In Figure 3.4, the accepting states of L_1 are connected to the start states of L_2 automaton, which generates an automaton for accepting $L_1 \cdot L_2$. Here $\cdot$ is the concatenation operator. So if $\omega_1 \in L_1$ and $\omega_2 \in L_2$, then $\omega_1 \cdot \omega_2 \in L_1 \cdot L_2$ will be accepted by this new automaton.

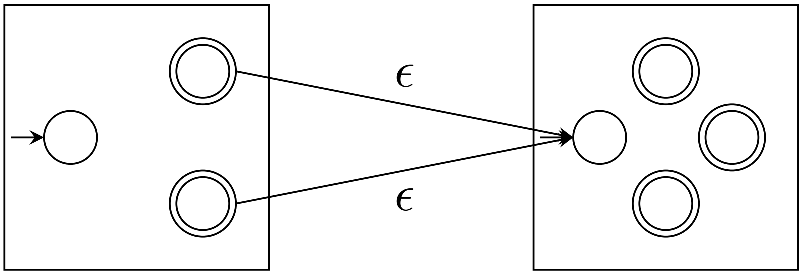


Figure 3.4: L_1 accepting automaton and L_2 accepting automaton connected

Now, we will try to find an equivalent DFA of this NFA having ϵ edges. For that, we will first find an NFA without ϵ edges which will preserve the original NFA, and this obtained NFA can then be constructed into a DFA.
Initially, just look at the ϵ−edges only and find for each state its ϵ−closure, which is the set of the states that we can reach from it by taking only ϵ−edges.
From each node, we can go to every node that is present in the ϵ−closure of that node for free. So wherever non-ϵ edges take us from these nodes in ϵ−closure, we can reach from the node itself whose

closure it was. So all these states will be connected to this node, in the new NFA.
The starting states and the final states of this new NFA will be the $\epsilon-$closures of the start and final states respectively of the original NFA.
So NFA without $\epsilon-$edges for the NFA in Figure 3.3 will look like as shown in Figure 3.5

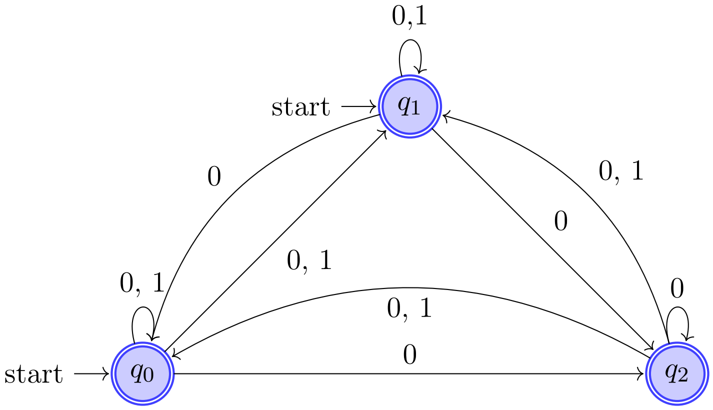


Figure 3.5: Automaton without Epsilon Edges

3.6.1 Equivalence with DFA

- If we can show that ϵ-NFA have an equivalent NFA, then the equivalence of ϵ-NFA and DFA is proved because every NFA has an equivalent DFA[5].

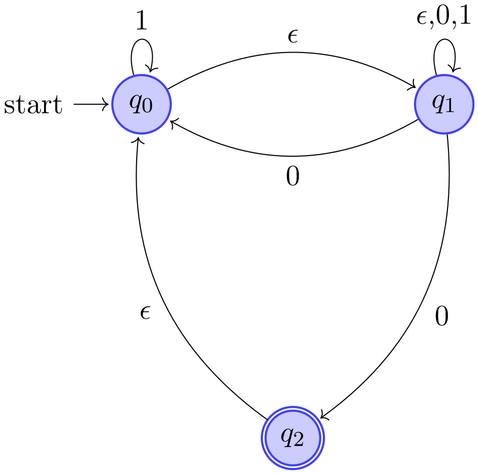


Figure 3.6: A Simple ϵ-NFA

Focus on ϵ-Edges of the Automaton to get the Epsilon Closure of the states in an Automaton

For Figure-3.6 Automaton:

[5] equivalence of NFA and DFA is already proved

$$q_0 \rightsquigarrow \{q_0,\, q_1\} \qquad \epsilon\text{-closure}_{q_0}$$

$$q_1 \rightsquigarrow \{q_1\} \qquad \epsilon\text{-closure}_{q_1}$$

$$q_2 \rightsquigarrow \{q_0\ ,\ q_1\ ,\ q_2\} \qquad \epsilon\text{-closure}_{q_2}$$

Rules for converting ϵ-NFA to equivalent NFA[6]:

1. Non-epsilon transitions for a symbol, denoted as $a \in \Sigma$, originating from any state q' within the epsilon closure of a state, say q, will also be present as non-epsilon transitions for state q in the equivalent NFA. These transitions leads to the same destination states for q in new NFA as they were for the state q' in the original epsilon-NFA.

2. The states in the epsilon closure of the initial state in the epsilon-NFA can serve as the initial state in the new NFA. However, this is not necessary, we can make an equivalent NFA where ϵ-NFA and the new NFA have same initial state(s).

3. All states within the epsilon-NFA that include the accepting state in their epsilon closure will also act as final states in the new/equivalent NFA.

For the Figure-3.6 epsilon-NFA, after applying the aforementioned rules, we obtain the following equivalent NFA:

[6] These rules will become more clearer in the next class, when we will learn about leading epsilon transitions and trailing epsilon transitions within the states of ϵ-NFA.

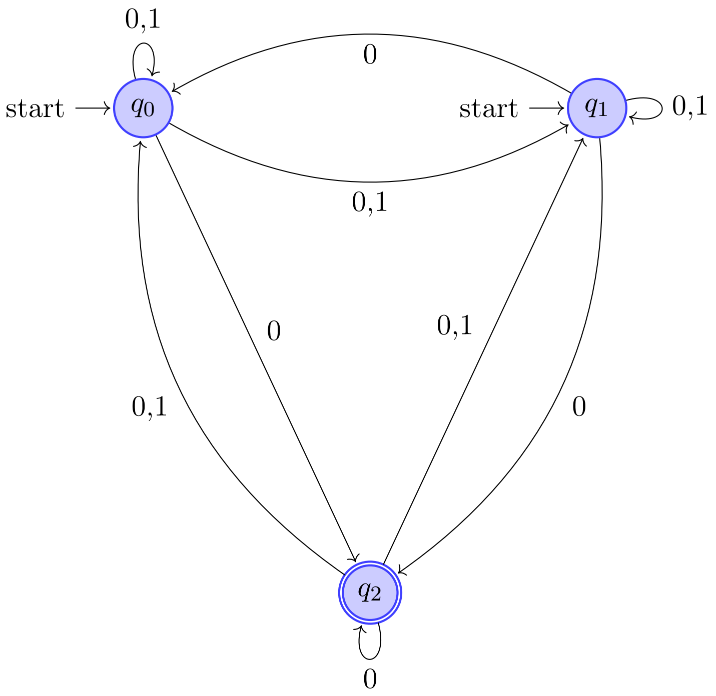


Figure 3.7: Equivalent NFA for Figure-3.6

3.7 Recap

We were trying to convert an NFA with ε edges to an equivalent NFA without ε edges in the previous lecture. We'll do that in more detail in this lecture.

3.7.1 Converting ε-NFA to non-ε-NFA

- **ε-closure** of a node is defined as set of those nodes which can be reached from that node by traversing over ε-edges, i.e. without consuming any character from the alphabet Σ. Also, the node itself is trivially a part of its ε-closure.

- **ε-edges** are those edges which can be traversed without consuming any character from the alphabet Σ, i.e. by consuming an empty string . Observe that the string "10" $\in L$ with ε-edges but without ε-edges, string "10"$\notin L$ where L is the Language of the NFA.

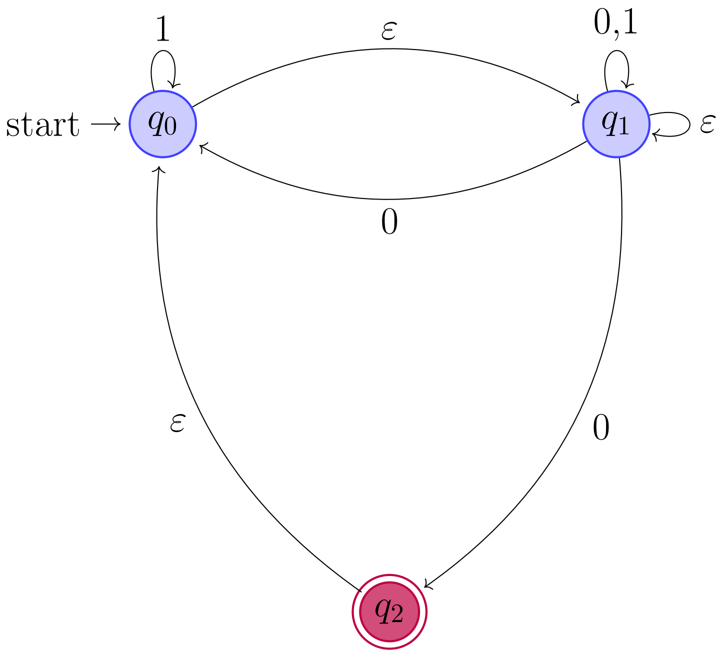


Figure 3.8: A Simple ε-NFA with alphabet $\Sigma = \{0, 1\}$

For example in this NFA,

$$\varepsilon\text{-closure}(q_0) = \{q_0, q_1\}$$

$$\varepsilon\text{-closure}(q_1) = \{q_1\}$$

$$\varepsilon\text{-closure}(q_2) = \{q_0, q_1, q_2\}$$

Now for the algorithm to convert an NFA with ε-edges to an equivalent NFA without ε-edges with the same alphabet $\Sigma = \{0, 1\}$ and with the same states, apply the following three steps:-

1. For each node q in NFA, find its ε-closure (say S) and mark all its non-ε-edges as they are. Now, for each node $q' \neq q$ in S, mark all non-ε-edges starting from q' going to q'' as extra edges starting from q going to q''. For eg., for the node q_0, the only distinct node in its ε-closure is q_1 so mark the edges $\{0, 1\}$ from q_1 to q_1, $\{0\}$ from q_1 to q_0 and $\{0\}$ from q_1 to q_2 as extra edges $\{0, 1\}$ going from q_0 to q_1, $\{0\}$ from q_0 to q_0 and $\{0\}$ from q_0 to q_2 respectively (marked in red).

2. Mark all those states as accepting whose ε-closures contain atleast one of the accepting states of the ε-NFA. For eg., here only q_2 has the accepting state q_2 in its ε-closure, so only q_2 is marked as accepting.

3. Starting states in the new non-ε-automaton will be the same as the starting states in the original ε-automaton.

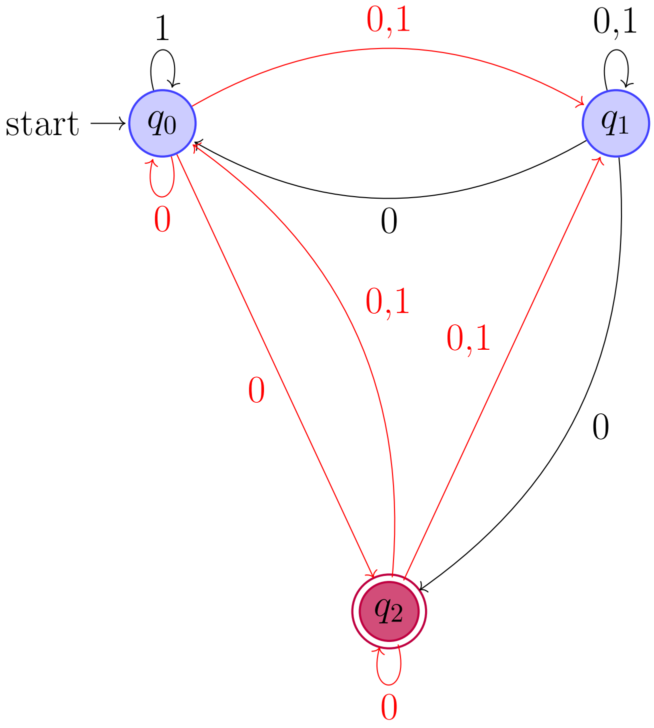


Figure 3.9: Equivalent non-ε-NFA

Subsequently, we'll use "original/ori" for the ε-automaton and "new" for the non-ε-automaton.

- One may ask why the states which are in the ε-closures of the accepting states of the original automaton are not marked as accepting in the non-ε-NFA. The answer is :- every string which reaches an accepting state a can also reach any of the states in ε-closure(a) by taking ε-edges and those strings are indeed in the language of the ε-NFA. But there are many such strings also which can reach at the states in the ε-closure(a) which are not in the language of the ε-NFA. So, if we mark those states as accepting, we are changing the language which is not our intention. For eg., $q_0 \in \varepsilon$-closure(q_2) but if we mark q_0 as accepting in the non-ε-NFA, then the string "1" will also be included in the language of the non-ε-NFA but "1" $\notin L(\varepsilon\text{-NFA})$.

3.7.2 Correctness of Algorithm

- If we say $w \in L(ori)$ it means that either w as it is $\in L(ori)$ or w with some ε's inserted $\in L(ori)$. On the other hand, if we say $w \in L(new)$ then it means that w as it is $\in L(new)$.

Now, what we mean by correctness of the algorithm is that the language of the new non-ε-automaton should exactly be equal to the language of the original ε-automaton. So we need to prove that

$$L(new) = L(ori)$$

1. Let's prove $L(new) \subseteq L(ori)$
   Let's take any arbitrary string $w = c_1c_2 \dots c_m \in L(new)$

   (a) Case 1: w is empty
   It means atleast one of the starting states s in the new automaton is an accepting state. Now, according to our algorithm (step 2), all accepting states in the new automaton are either accepting states in the original automaton as well or are those states which have atleast one accepting state in the original automaton in their ε-closures. So, ε-closure(s) definitely contains an accepting state in the original automaton. So, we have a path ε^k for some $k \geq 0$ from s to an accepting state in the original automaton and thus $w \in L(ori)$.

   (b) Case 2: w is non-empty
   It means that we have a sequence of jumps from a starting state s_0 in the new automaton to s_1 upon reading c_1 and then from s_1 to s_2 upon reading c_2 and so on till the state s_m upon reading c_m such that s_m is an accepting state. Now, according to our algorithm (step 1), for s_0 in the original automaton either we have a direct edge $\{c_1\}$ from s_0 to s_1 or we have some $s_0' \in \varepsilon$-closure(s_0) which has the edge $\{c_1\}$ from s_0' to s_1. So, effectively we can reach s_1 from s_0 in the original automaton as well by following a path $\varepsilon^k c_1$ for some $k \geq 0$. Similarly, we can have a path $\varepsilon^{k_1} c_2 \varepsilon^{k_2} c_3 \dots$ for $k_i \geq 0$ till s_m in the original automaton. Now, according to our algorithm (step 2), either s_m is an accepting state in the original automaton as well or there lies some accepting state in the ε-closure(s_m). So, finally, we can have a sequence of $\varepsilon^{k'}$ for some $k' \geq 0$ to reach to an accepting state from s_m in the original automaton and thus, $w \in L(ori)$.
   Hence, proved.

2. Let's prove $L(ori) \subseteq L(new)$
   Let's take any arbitrary string $w = c_1c_2 \dots c_m \in L(ori)$

   (a) Case 1: w is empty
   It means that there exists atleast one starting state s in the original automaton such

that ε-closure(s) contains an accepting state s'. Now, according to our algorithm (step 2), all accepting states in the new automaton are either accepting states in the original automaton as well or are those states which have atleast one accepting state in the original automaton in their ε-closures. So, the starting state s is an accepting state in the new automaton and thus $w \in L(new)$.

(b) Case 2: w is non-empty
It means that we have a path $\varepsilon^{k_0} c_1 \varepsilon^{k_1} c_2 \dots \varepsilon^{k_{m-1}} c_m \varepsilon^{k_m}$ for all $k_i \geq 0\ \forall i \in \{0, 1, 2 \dots, m\}$ from a starting state s_0 in the original automaton to an accepting state s_{m+1} where taking ε^{k_i} from any state x means going to some state $y \in \varepsilon$-closure(x). Here, s_1 is the state reached after reading c_1, s_2 after reading c_2… s_m after reading c_m and s_{m+1} after taking ε^{k_m}. Now, according to our algorithm (step 1), for s_0 in the new automaton, we have a direct edge $\{c_1\}$ from s_0 to s_1. Similarly, we'll have direct edges from s_1 to s_2 and so on. So, finally, we can reach s_m from s_0 in the new automaton. Now, according to our algorithm (step 2), s_m is an accepting state in the new automaton because ε-closure(s_m) contains accepting state s_{m+1} in the original automaton. Thus, $w \in L(new)$.
Hence, proved.

Thus, we've shown that $L(new) = L(ori)$

3.7.3 Intuition of the algorithm

- It should be clear from the correctness proof itself where we constructed paths in the new and original automatons using the step 1 of the algorithm. Also, if the path in the original automaton had trailing ε's we weren't able to reach the same state in the new automaton but had to make the languages of the two automatons exactly same so we concluded step 2 of the algorithm.

3.7.4 Extras

- **Language of a node** q is defined as the set of all those strings $w \in \Sigma^*$ such that upon reading w character by character we can reach q from any of the starting states of the automaton.

- Note that in ε-automaton we could also take ε-edges in between the characters, before the first character as well as after the last character of w and reach q so such strings would also be considered in the language of q.

- And the language of an automaton is defined as the union of the languages of all its accepting nodes. So, we have

$$L(new) = \bigcup_i L(q_i)\ \forall \text{ accepting states } q_i \text{ of the new non-}\varepsilon\text{-automaton}$$
$$L(ori) = \bigcup_j L(q_j)\ \forall \text{ accepting states } q_j \text{ of the original } \varepsilon\text{-automaton}$$

- A lemma: $L(q_i)_{original} \supseteq L(q_i)_{new}$ holds true where q_i is any node of the NFA and $L(q_i)$ is the language of that node.

- Proof: Going by the same idea as we did in Correctness proof part 1, we can prove the lemma. Like if $w \in L(q_i)_{new}$ and w is empty then q_i must be one of the starting states in the new automaton and also in the original automaton so $w \in L(q_i)_{original}$ since starting states are the same in both the automatons. If $w \in L(q_i)_{new}$ and w is non-empty, then we can have a path with some ε's in between the characters of w from some starting state s_0 to q_i in the original automaton and thus $L(q_i)_{original} \supseteq L(q_i)_{new}$ is indeed true.
  Where the $\supset$ sign comes in is the case when the path of w has some trailing ε's. For eg. consider this ε-NFA and its equivalent non-ε-NFA,

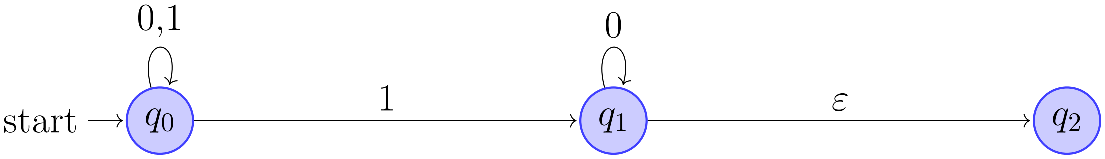


Figure 3.10: ε-NFA

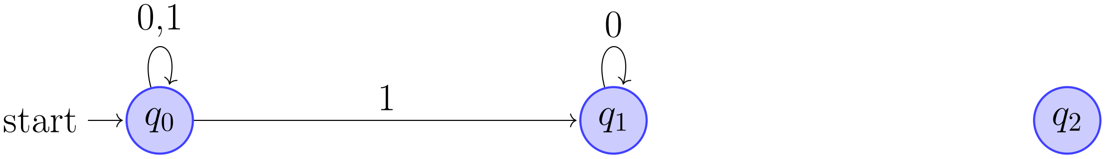


Figure 3.11: Equivalent Non-ε-NFA

$L(q_2)_{original}$ is non-empty and for instance contains the string "1" with the path "1ε" from q_0 to q_1 but $L(q_2)_{new}$ is clearly empty,i.e., the null set ϕ. This happened because the path contained a trailing ε. So, $L(q_2)_{new} \subset L(q_2)_{original}$.

3.7.5 Significance of ε-edges

ε-edges allow us to jump from one part of the NFA to another part of the NFA without consuming any additional character. Thus, if we have to do some operations sequentially we can make our life simple by using ε-edges. For eg., suppose we have to check $w = u.v$ such that u contains an even number of 1's and v contains $1\,mod\,3$ number of 1's. We can accomplish the above task by constructing two NFA's wherein the first NFA will accept all the satisfying u's and the second NFA will accept all the satisfying v's. Now, we'll just join the accepting states of first NFA to the starting states of the second NFA using ε-edges and convert the accepting states of first NFA & the starting states of the second NFA into normal intermediate states. Thus, we will get a single NFA with starting states same as the starting states of the first NFA and accepting states same as the accepting states of the second NFA. Here, ε-edges allowed us to capture the non-determinism of the breaking point between u & v.

* Any automaton containing ε-edges cannot be a DFA because ε-edges bring in uncertainty as we could choose to stay in that state or take the ε-edge.

3.7.6 Examples

- $L = \{x \in \Sigma^* | x = u.v.w, v \in \Sigma^*, u, w \in \Sigma^{+,|u| \leq 2, u=w}\}$ where $\Sigma = \{0, 1, 2 \ldots 9\}$
For instance, "0000" $\in L$ because either take $u = w = 0, v = 00$ or take $u = w = 00, v = \varepsilon$
Also, "1234" $\notin L$ because we cannot have any satisfiable u, v, w.
It's NFA will be a combination of 110 NFA's of the following form :-

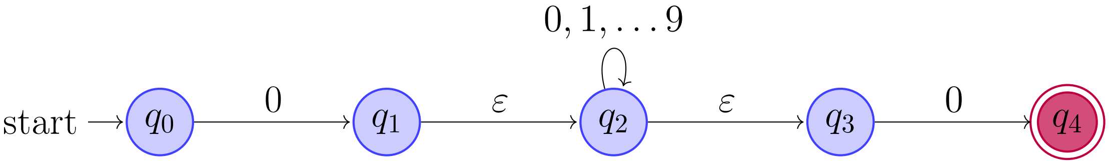


Figure 3.12: ε-NFA for $u = w = 0$

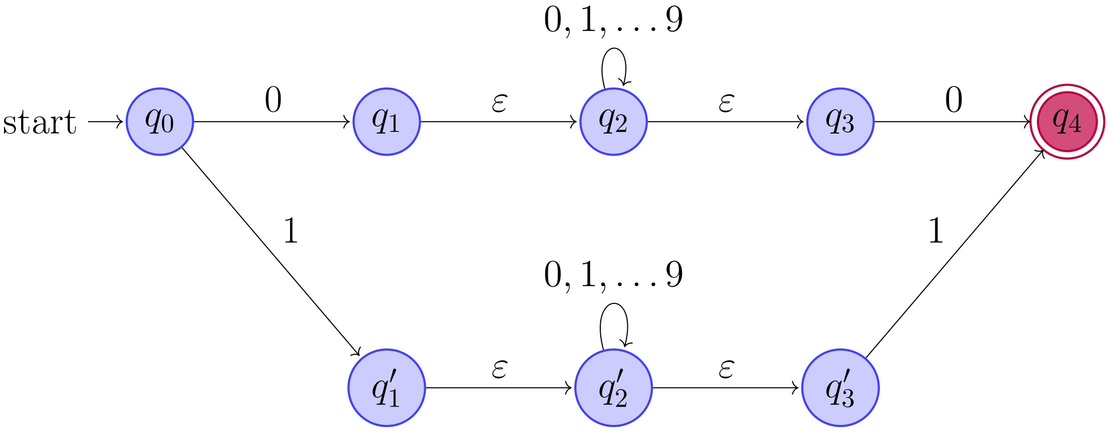


Figure 3.13: ε-NFA for $u = w = 0, 1$

One for each $u = w = 0, 1, 2 \ldots 9$ so 10 here and 100 more for $u = w = 00$ to 99. We can have separate NFA's with different possible q'_1, q'_3 states. Like instead of 0 put 1 to 9 and 00 to 99 there on the edge from q_0 to q'_1 and q'_3 to q_4 and in this way we can form the whole NFA just like in figure 6.

- $L = .^*xxx.^*$ where $\Sigma = \{a, b, \ldots z\}$
Here, we want to locate "xxx" in the text so the following NFA captures this language:-

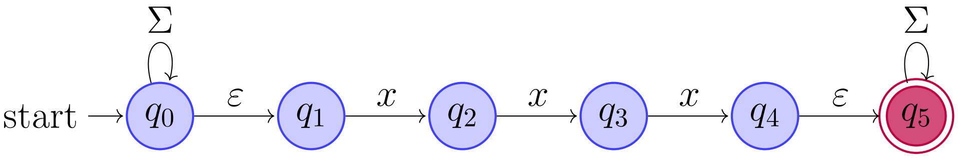


Figure 3.14: ε-NFA for searching "xxx"

- **Takeaway task :** Think about how KMP implicitly constructs NFA for pattern matching.

3.8 Equivalence in Finite Automata

In our exploration of finite automata, we've encountered deterministic finite automata (DFA), non-deterministic finite automata without epsilon transitions (NFA), and non-deterministic finite automata with epsilon transitions (ε-NFA). Surprisingly, despite their apparent differences, these three models are fundamentally equivalent in terms of computational power.

DFA, characterized by their deterministic nature, are particularly useful in scenarios where determinism is crucial, such as in hardware design where predictability is paramount.

ε-NFA, on the other hand, introduce a high level of abstraction by allowing transitions without consuming input symbols, which can simplify the representation of certain languages and aid in conceptual clarity.

NFA without epsilon transitions also provide a similarly high level of abstraction, allowing for flexibility in modeling complex systems and languages.

3.9 DFA definition

Any DFA can be completely represented by the tuple DFA $= (Q, \Sigma, q_0, \delta, F)$ where,

- Q is the set of all states
- Σ is the alphabet
- q_0 is the starting state
- $\delta : (Q \times \Sigma) \to Q$ is the transition function
- F is the set of final or accepting states

3.9.1 Combinations of DFAs

We can represent the combination of two DFAs defined on the same alphabet Σ, DFA$_1 = (Q, \Sigma, q_0, \delta, F)$ and DFA$_2 = (Q', \Sigma, q'_0, \delta', F')$ by another DFA,

$$\text{DFA}_3 = \left(Q \times Q', \Sigma, \left(q_0, q'_0\right), \hat{\delta}, \hat{F}\right) \tag{3.1}$$

where $\hat{\delta}\left((q, r), a\right) = (\delta(q, a), \delta'(r, a))$ and $\hat{F}$ can be defined according to the required operation on the DFAs.

For example, if $Q = \{q_0, q_1\}$ and $Q' = \{r_0, r_1, r_2\}$ and $F = \{q_1\}$ and $F' = \{r_1, r_2\}$ then for the **intersection** of these Automata, $\hat{F} = \{(q_1, r_1), (q_1, r_2)\}$ and for the **union** of the Automata, $\hat{F} = \{(q_1, r_0), (q_1, r_1), (q_1, r_2), (q_0, r_1), (q_0, r_2)\}$. Similarly we can define the **complement** operation by taking $\hat{F} = Q - F$.

With these basic rules in place, we can go on to define more complicated combinations like $(DFA_1 \cap DFA_3) \cup (DFA_2 \cap DFA_3) - (DFA_1 \cap DFA_2)$

We now have another way to tell if a language is a subset of another language. To tell if $L_1 \subseteq L_2$, we have to show that $L_1 \cap L_2^{\complement} = \phi$. The problem thus reduces to showing that in the DFA defined by $DFA_1 \cap DFA_2^{\complement}$ there is no path from the start node to any accepting state.

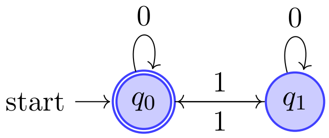


Figure 3.15: Two-State Automaton

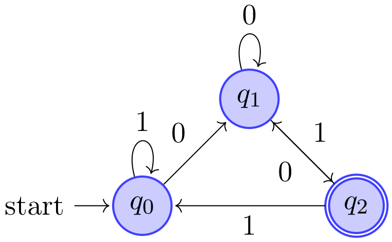


Figure 3.16: Three-State Automaton

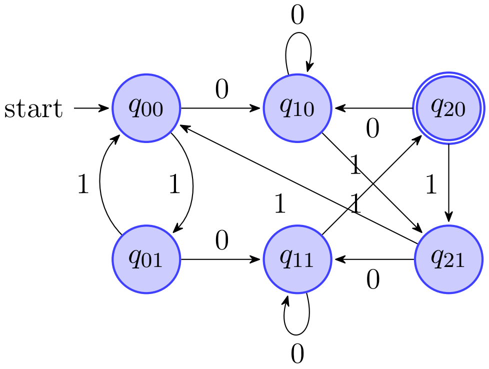


Figure 3.17: Intersection of the two automata

3.9.2 Combinations of NFAs

For two NFAs, the union and intersection is defined in a similar way with the only difference being the transition function since it now returns a set of states instead of a single state.

$$\hat{\delta}\left((q,r),a\right) = \delta(q,a) \times \delta'(r,a) \tag{3.2}$$

However, for NFAs, just flipping the accepting and non-accepting states won't give us the complement.
Thus to take the intersection or union of two NFAs, we can follow a similar approach to DFAs but for complementation, the NFA must first be converted to a DFA and then complemented to give the actual complement of the original NFA.

3.9.3 Closure Properties

Given L_1, L_2 are two regular languages over the alphabet $\Sigma = \{a, b\}$, $\overline{L_1}$, $L_1 \cup L_2$, $L_1 \cap L_2$ are also regular languages (because their corresponding DFAs can be constructed by the combination of the original DFAs).

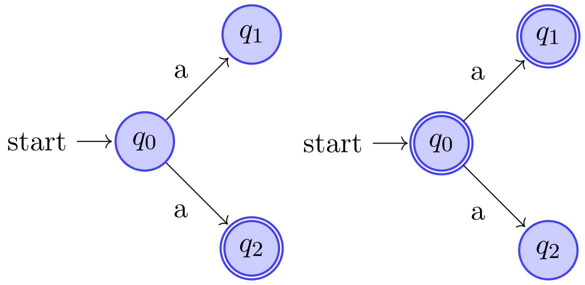


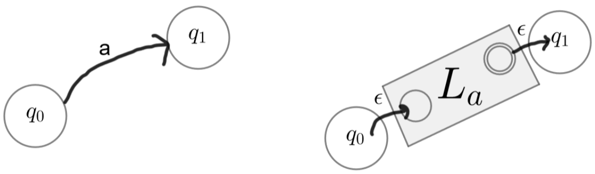
Figure 3.18: These NFAs are not complements of each other

3.10 Substitution

We will start with an example.
Consider two alphabets $\Sigma_1 = \{a, b\}$ and $\Sigma_2 = \{0, 1, 2\}$ and languages $L_1 = a^{*b^*}$ defined on Σ_1^* and $L_a = 0^{*(1+2)^{*1^*}}$ and $L_b = 1^{*(0+2)^*}$ defined on Σ_2^*.
Now we define a set subst(L_1, L_a, L_b) as

$$\text{subst}(L_1, L_a, L_b) = \left\{ w \in \Sigma_2^{* \mid \exists u \in L_1 = \alpha_1\alpha_2 \ldots \alpha_k \text{ such that } w \in L_{\alpha_1} L_{\alpha_2} \ldots L_{\alpha_k}} \right\} \tag{3.3}$$

Or equivalently, subst$(L_1, L_a, L_b) = \bigcup\limits_{u = \alpha_1\alpha_2 \ldots \alpha_k \in L_1} L_{\alpha_1} L_{\alpha_2} \ldots L_{\alpha_k}$
Intuitively, the substitution operation is to replace each letter by a language.
Diagrammatically, it is represented as replacing each edge by an entire automaton and connecting the initial and accepting states to the original states by ϵ edges.



Using this operation, we can easily prove that if L_1 and L_2 are regular languages then $L_1 \cdot L_2$ is also regular, since $L = \{a \cdot b\}$ is regular then subst(L,L_1,L_2) will also be regular.

3.10.1 Infinite Languages

If we have a DFA with n states and there exists some string of length greater than n which is accepted by the DFA, then by the pigeon-hole principle there must exist some state q in the DFA such that there is a cycle with q in it. Suppose that the accepting string is $u \cdot v \cdot w$, where v is the string that starts and ends at the same state, then the strings $u \cdot w$, $u \cdot v^2 \cdot w$, $u \cdot v^{* \cdot w}$ are also accepted in the DFA. Hence the language formed by the string is of infinite length.

Chapter 4

Regular Expressions

4.1 Introduction

In arithmetic, we can use the operations + and × to build up expressions such as $(5 + 3) \times 4$. Similarly, we can use the regular operations to build up expressions describing languages, which are called regular expressions. An example is: $(0 \cup 1)0^*$. The value of the arithmetic expression is the number 32. The value of a regular expression is a language. In this case, the value is the language consisting of all strings starting with a 0 or a 1 followed by any number of 0s. We get this result by dissecting the expression into its parts. First, the symbols 0 and 1 are shorthand for the sets $\{0\}$ and $\{1\}$. So $(0 \cup 1)$ means $(\{0\} \cup \{1\})$. The value of this part is the language $\{0, 1\}$. The part 0^* means $\{0\}^*$, and its value is the language consisting of all strings containing any number of 0s. Second, like the × symbol in algebra, the concatenation symbol ∘ often is implicit in regular expressions. Thus $(0 \cup 1)0^*$ actually is shorthand for $(0 \cup 1) \circ 0^*$. The concatenation attaches the strings from the two parts to obtain the value of the entire expression. Regular expressions have an important role in computer science applications. In applications involving text, users may want to search for strings that satisfy certain patterns. Regular expressions provide a powerful method for describing such patterns. Utilities such as `awk` and `grep` in UNIX, modern programming languages such as Perl, and text editors all provide mechanisms for the description of patterns by using regular expressions.

4.2 Formal Definition of a Regular Expression

Let R be a regular expression if:

1. a for some a in the alphabet Σ,
2. ε,
3. $\emptyset$,
4. $(R_1 \cup R_2)$, where R_1 and R_2 are regular expressions,
5. $(R_1 \cdot R_2)$, where R_1 and R_2 are regular expressions, or
6. (R_1^*), where R_1 is a regular expression.

In items 1 and 2, the regular expressions a and ε represent the languages $\{a\}$ and $\{\varepsilon\}$, respectively. In item 3, the regular expression $\emptyset$ represents the empty language. In items 4, 5, and 6, the expressions represent the languages obtained by taking the union or concatenation of the languages R_1 and R_2, or the Kleene star of the language R_1, respectively.

4.3 Semantics of Regular Language

The semantics of regular language involves understanding the structure and meaning of expressions within the language.

4.3.1 Atomic Expressions

Consider an atomic expression $[a]$, where a represents a single letter. In this context, $[a]$ denotes the language consisting of only the string a. However, it's important to note that the a within $[a]$ is not considered a letter of the alphabet; rather, it signifies a language comprising a single letter string.

4.3.2 Union Operation

For expressions e_1 and e_2, $[e_1 + e_2]$ represents the union of the languages denoted by $[e_1]$ and $[e_2]$. In simpler terms, $[e_1 + e_2]$ encompasses all strings that belong to either $[e_1]$ or $[e_2]$.

4.3.3 Concatenation Operation

When considering e_1 concatenated with e_2, denoted as $[e_1.e_2]$, it signifies the concatenation of languages represented by $[e_1]$ and $[e_2]$. This operation results in a language consisting of all possible combinations of strings where the first part belongs to $[e_1]$ and the second part belongs to $[e_2]$.

4.3.4 Example

Let's illustrate with an example: $(ab) + a$. This expression represents the union of the language containing the string ab and the language containing the string a, resulting in $\{ab, a\}$.

4.3.5 Order of Precedence

In the semantics of regular language, the order of precedence for operations is as follows: $*$ (Kleene star) $>$ $\cdot$ (concatenation) $>$ $+$ (union).

4.3.6 Kleene Star

The Kleene star operation $[e_1^*]$ denotes the union of zero or more concatenations of $[e_1]$. In other words, $[e_1^*]$ encompasses all possible strings that can be formed by concatenating any number of strings from $[e_1]$.

4.3.7 Example

For an expression $e_1 = a + b$, the language denoted by $[e_1]$ is $\{a, b\}$. Therefore, $[(a + b)^*]$ represents the set of all possible strings comprising as and bs, including the empty string ε, a, b, ab, ba, aa, bb, and so on.

4.3.8 Further Examples

- $[a^* + b^*] = \{u \in \Sigma^* \mid u = a^n$ or $u = b^n$ for some $n \geq 0\}$
- $[a^* \cdot b^*] = \{u \in \Sigma^* \mid u = a^n b^m$ and $n \geq 0$ and $m \geq 0\}$
- $[(a^* \cdot b^*)^*]$ represents the set of all strings containing any number of occurrences of strings composed of as followed by bs.
- $0^*10^* = \{w \mid w$ contains a single $1\}$
- $\Sigma^*1\Sigma^* = \{w \mid w$ has at least one $1\}$
- $\Sigma^*001\Sigma^* = \{w \mid w$ contains the string 001 as a substring$\}$
- $1^*(01^+)^* = \{w \mid$ every 0 in w is followed by at least one $1\}$
- $(\Sigma\Sigma)^* = \{w \mid w$ is a string of even length$\}$
- $(\Sigma\Sigma\Sigma)^* = \{w \mid$ the length of w is a multiple of $3\}$
- $01 \cup 10 = \{01, 10\}$
- $0\Sigma^*0 \cup 1\Sigma^*1 \cup \{0,1\} = \{w \mid w$ starts and ends with the same symbol$\}$
- $(0 \cup \varepsilon)1^* = 01^* \cup 1^*$
- $(0 \cup \varepsilon)(1 \cup \varepsilon) = \{\varepsilon, 0, 1, 01\}$
- $1^*\emptyset = \emptyset$
- $\emptyset^* = \{\varepsilon\}$ The star operation puts together any number of strings from the language to get a string in the result. If the language is empty, the star operation can put together 0 strings, giving only the empty string.

4.4 Kleene's Theorem

In the last lecture, we introduced regular expressions and saw how they can be used to represent languages. In this lecture, we explore one of the most fundamental theorems of Automata Theory, Kleene's Theorem.

Kleene's Theorem: In terms of expressive power,

Regular Expressions $\equiv$ NFAs with ϵ edges $\equiv$ NFAs without ϵ edges $\equiv$ DFAs

In previous lectures, we have already proved the equivalence of NFAs with ϵ edges, NFAs without ϵ edges and DFAs. So in order to prove Kleene's Theorem, it suffices to show the equivalence between Regular Expressions and NFAs with ϵ edges.

To prove: Regular Expressions $\equiv$ NFAs with ϵ edges
i.e., L(Reg. Ex.) = L(NFAs with ϵ edges)

The proof can be divided into two subparts

1. L(Reg. Ex.) $\subseteq$ L(NFAs with ϵ edges)
2. L(Reg. Ex.) $\supseteq$ L(NFAs with ϵ edges)

We prove each of these parts separately and then combine them to get the required result.

4.4.1 Part 1: L(Reg. Ex.) $\subseteq$ L(NFAs with ϵ edges)

Let us consider the following example.

Suppose our alphabet is $\Sigma = \{0, 1\}$
and our regular expression is $((0.1)^*+(1.0)^*)^*.(1.1+0.0)^*$

We wish to construct an NFA with ϵ edges that accepts the same language as our regular expression.

To do this, we construct the parse tree of the regular expression and use a bottom up approach starting from the leaves to construct an NFA for each node of the parse tree.

The parse tree of the given regular expression is as follows.

[.· [.∗ [.+ [.∗ [.· [.0] [.1]]] [.∗ [.· [.1] [.0]]]]] [.∗ [.+ [.· [.1] [.1]] [.· [.0] [.0]]]]]

We start from the leaves, in this case the nodes labelled 0 and 1. Consider the leaf 0. It represents a language with only one string, i.e., L(0) = {0}. It can be represented by the NFA

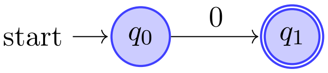


Figure 4.1: NFA that accepts only {0}

Similarly leaf 1 can be represented by the NFA

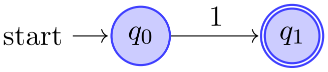


Figure 4.2: NFA that accepts only {1}

Now we have NFAs for each of the leaf nodes. Let us see how to construct the NFAs for the rest of the nodes from these.

Suppose we have the NFAs of regular expressions **a** and **b**, and we want to obtain the NFA of **a.b**. Since **.** is simply the concatenation operator, the new NFA should accept all strings that satisfy both the NFAs in sequence. Such an NFA can be constructed by connecting the final state of **a** and the starting state of **b** (and marking them as regular states) by an ϵ edge. This results in a new NFA whose starting state is the starting state of **a** and final state is the final state of **b**. The language of this NFA is simply **a.b**.

In out example, given the NFAs of **0** and **1** we want to construct the NFA of **0.1**. Following the above procedure, we obtain the NFA

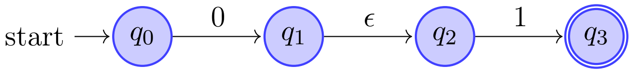


Figure 4.3: NFA for **0.1**

From the parse tree, the next node we need to construct an NFA for is **(0.1)***.

The language accepted by **a*** is simply the set of strings in which strings in the language of **a** are concatenated with themselves multiple (possibly 0) times. To obtain the NFA for **a***, we take the NFA of **a**, introduce a new state that is both starting and accepting, and draw ϵ edges from the original final state to the new state and from the new state to the original starting state. We also mark the original starting and accepting states as normal states. The correctness of this procedure can be proved by induction.

Applying this to **0.1**, we obtain the NFA for **(0.1)***

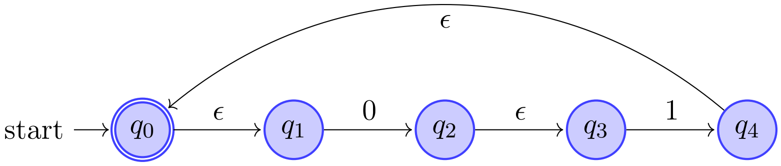


Figure 4.4: NFA for **(0.1)***

Similarly we can construct the NFA for **(1.0)***

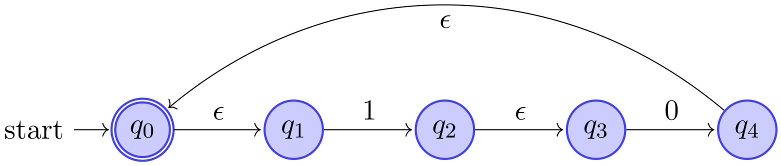


Figure 4.5: NFA for **(1.0)***

Now the next step in our recursive contruction of parse tree nodes is to obtain the NFA for **(0.1)*** + **(1.0)***.

The language accepted by **a+b** is simply the union of the language accepted by **a** and the language accepted by **b**. So in order for a string to be accepted by the NFA of **a+b**, it must be accepted by either the NFA of **a** or the NFA of **b**. Hence the required NFA can be obtained by introducing a new start state, drawing ϵ edges from it to the starting states of the original NFAs, and marking the original NFA starting states as normal states.

Applying this procedure to $\mathbf{(0.1)}^*$ and $\mathbf{(1.0)}^*$, we get the NFA of $\mathbf{(0.1)}^* + \mathbf{(1.0)}^*$

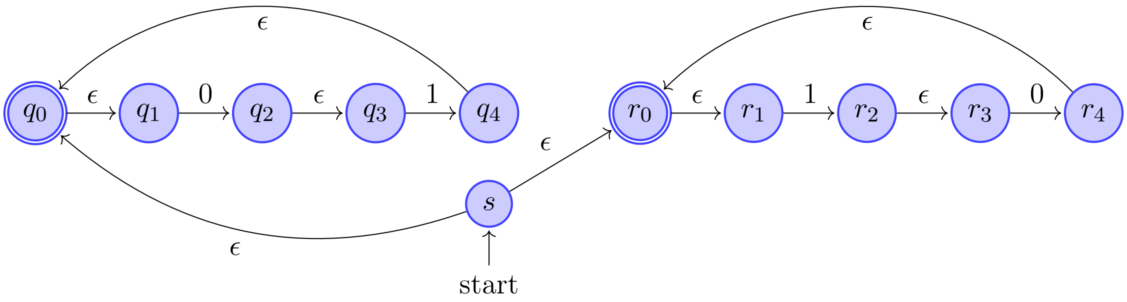
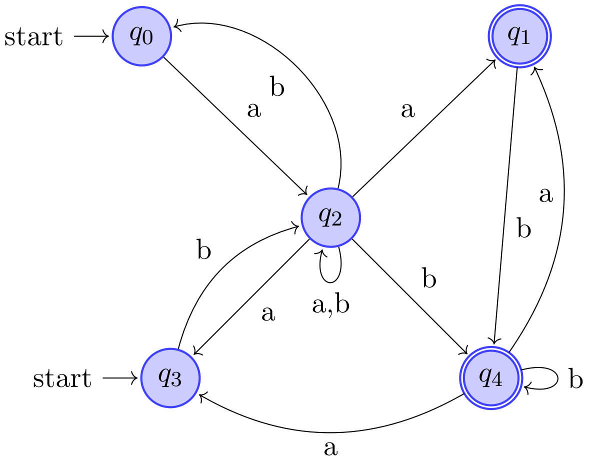


Figure 4.6: NFA for $\mathbf{(1.0)}^*+\mathbf{(0.1)}^*$

Using the same procedures, we can construct NFAs for all the remaining nodes of the parse tree. We observe that the size of the NFA is linear in the size of the original regular expression.

4.4.2 Part 2: L(Reg. Ex.) $\supseteq$ L(NFAs with ϵ edges)

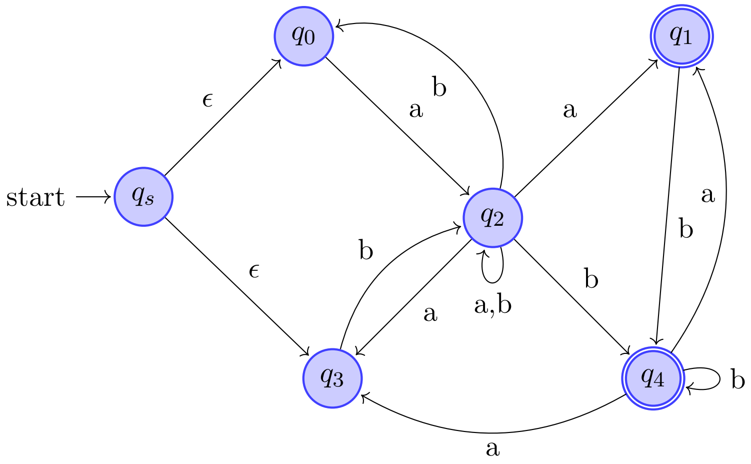
Once again we consider an example NFA with ϵ edges and convert it to a regular expression. Now we have $\Sigma = \{a, b\}$ and the following NFA



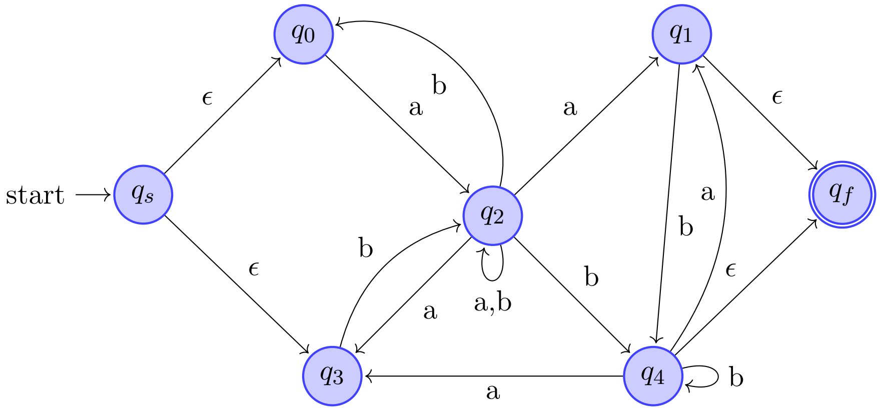
Our strategy to obtain a regular expression is to gradually reduce the 5 state automaton to one with fewer states, and keep repeating this process until we end up with a regular expression. To perform this reduction, we relax a condition on NFAs; we now allow edges to be labelled by regular expressions. An edge labelled by a regular expression simply means a path in which we consume a string satisfying the regular expression.

Algorithm

Step 1: Create a new starting state and get rid of all the old ones. Connect this new state to the old starting states using ϵ edges.



Step 2: Create a new final state and get rid of the old ones. Connect all the old final states to the new state using ϵ edges.



Observe that our NFA now has one single initial state with no edges leading back to it, and one single accepting state with no edges going out of it.

Step 3: Systematically remove all the states except for the inital and accepting states.

Suppose we choose state q_2 to be removed first. This state was facilitating some strings' paths from the initial to the accepting state. In order to remove q_2, we must first create alternate paths not involving q_2 that allow these strings to reach the final state. To do this, we follow the approach below,

1. Choose an incoming transition of q_2. In our example, suppose we choose the transition from q_0 to q_2 on consuming a.

2. Now consider all the outgoing edges of q_2, say from q_2 to q_x. Our goal is to facilitate transitions from q_0 to q_x without involving q_2. This can be done by introducing direct transitions from q_0 to q_x and labelling them with the regular expressions formed by concatenating $label(q_0 - q_2)$ and $label(q_2 - q_x)$, and then removing the incoming edge.

   Applying step 2 to q_2,

   Observe that q_2 has a self loop labelled a,b, which can be replaced by regular expression **a+b**. This means that while concatenating $label(q_0 - q_2)$ and $label(q_2 - q_x)$, we must also account for the regular expression **(a+b)*** in between the labels.

   (a) Outgoing edge: q_2 to q_0 on consuming b

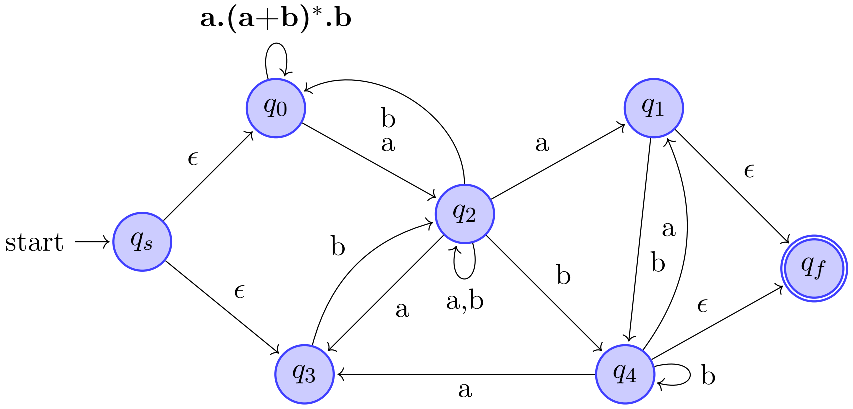


Figure 4.7: (a)

   (b) Outgoing edge: q_2 to q_1 on consuming a

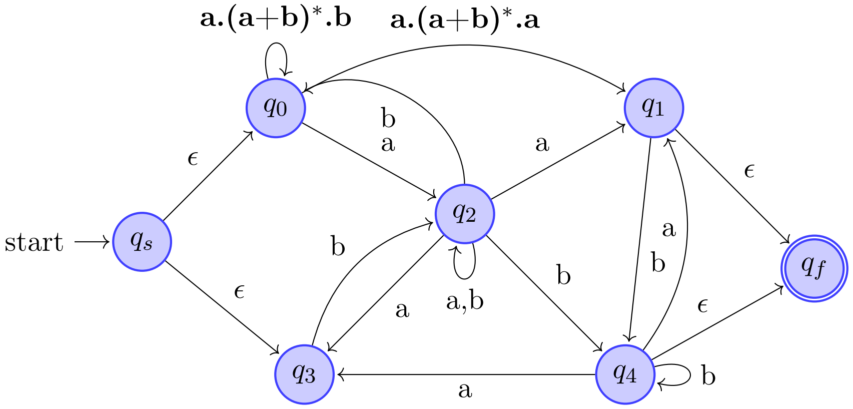


Figure 4.8: (b)

(c) Outgoing edge: q_2 to q_3 on consuming a

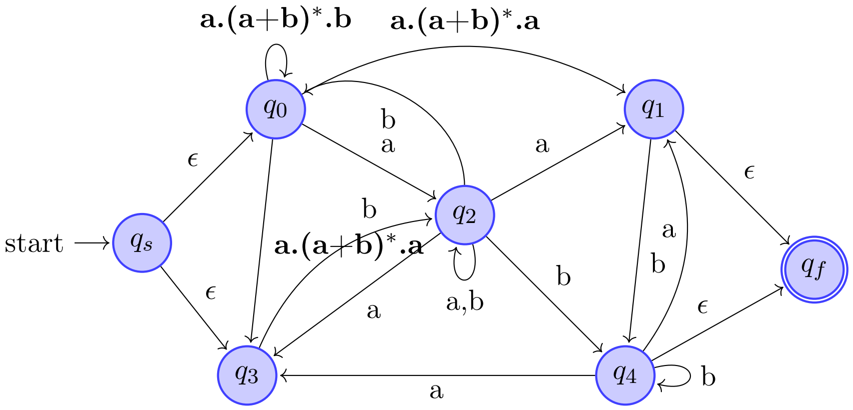


Figure 4.9: (c)

(d) Outgoing edge: q_2 to q_4 on consuming a

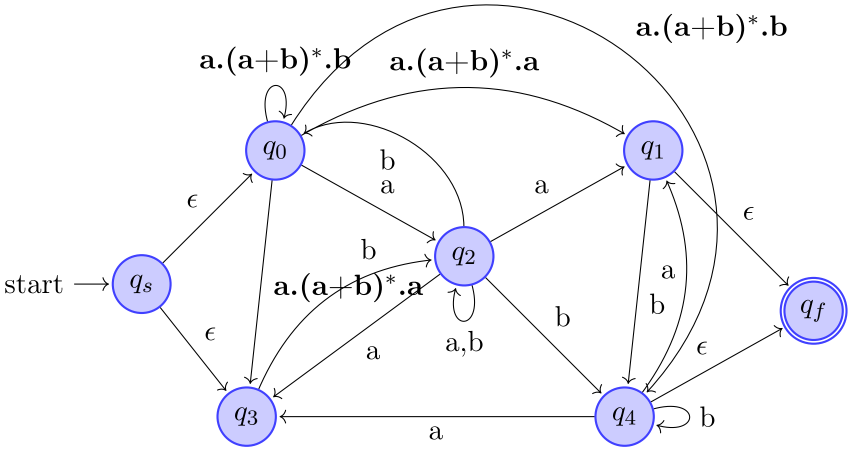


Figure 4.10: (d)

The NFAs corresponding to applying step 2 to each of the above transitions are illustrated on the next page.

Now that we have added the above four transitions, strings from q_0 do not need q_2 to reach the final state. So we can remove the incoming edge from q_0 to q_2 (see figure 11). Now we repeat the same procedure for all incoming edges of q_2. After doing this, q_2 has no incoming edges and hence cannot be part of the path of any string. So it does not contribute to our NFA and can be removed along with all its outgoing edges.

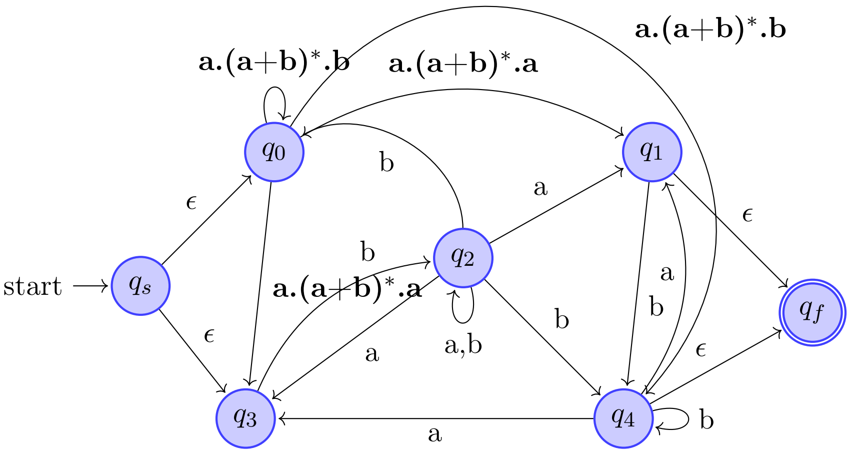
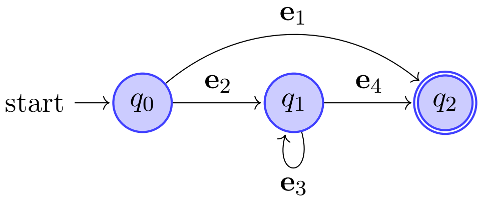


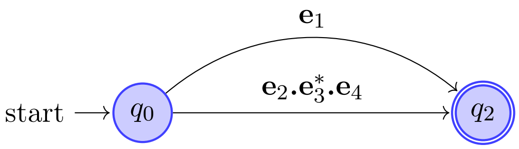
Figure 4.11: NFA after removing the incoming edge

We repeat these steps for each remaining non-initial non-accepting state of the NFA, and finally obtain the required regular expression.

To see how exactly the NFA becomes a regular expression, we illustrate the pre-final step for an arbitrary NFA.

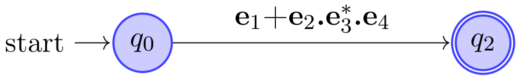


Eliminating the intermediate state, we get



Combining the two transitions,

So we obtain the regular expression $\mathbf{e}_1+\mathbf{e}_2.\mathbf{e}_3^*.\mathbf{e}_4$.



4.4.3 Checking Subsethood of Languages

We return to the example from the previous lecture, proving the equivalence of the regular expressions $(\mathbf{a}^*\mathbf{b}^*)^*$ and $(\mathbf{a}+\mathbf{b})^*$. Now we can leverage Kleene's Theorem and convert both expressions into their equivalent DFAs, and then check if both DFAs accept the same language or not. To check this, we can check each of the following conditions separately,

1. $\mathrm{L(DFA_1)} \subseteq \mathrm{L(DFA_2)}$
2. $\mathrm{L(DFA_1)} \supseteq \mathrm{L(DFA_2)}$

and then combine their results to get

$\mathrm{L(DFA_1)} = \mathrm{L(DFA_2)}$

But how do we check if a language $\mathrm{L_1}$ is a subset of another language $\mathrm{L_2}$ from just their DFAs? We use the following algorithmic procedure.

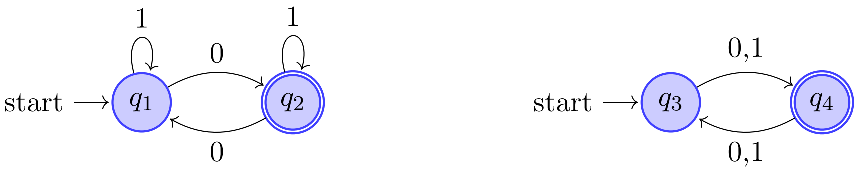
We say that $\mathrm{L_1} \subseteq \mathrm{L_2}$ when every string in $\mathrm{L_1}$ is also present in $\mathrm{L_2}$. So we can say

$$L_1 \subseteq L_2 \equiv L_1 \cap (\Sigma^{*\setminus L_2)=\phi}$$

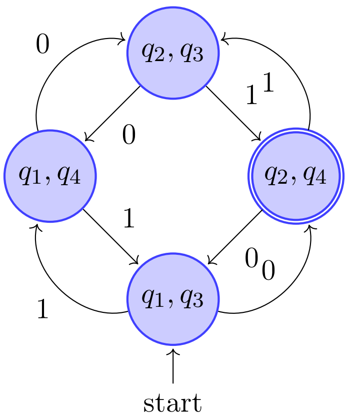
The language $\Sigma^{*\setminus L_2}$ is just the set of all strings not accepted by L_2. So to obtain its DFA, all we have to do is invert the acceptance status of each state (accepting states become normal states and vice versa). Also since L_2 is a regular language, $\Sigma^{*\setminus L_2}$ (denoted by L_2^c or $\bar{L_2}$) is also a regular language.

Now given the DFAs of L_1 and L_2^c, we wish to construct the DFA of $L_1 \cap L_2^c$. This new DFA should accept only those strings that are accepted by both L_1 and L_2^c.

We achieve this by running both DFAs simultaneously on the same string, and checking if we end up in a pair of accepting states. This is analogous to taking the Cartesian product of both transition functions. Consider the following example:



Then the DFA representing their intersection is given by
Since the initial condition we wanted to satisfy was $L_1 \cap L_2^c = \phi$, we need a way of checking this from the DFA. If the language of a DFA is ϕ, this means the DFA does not accept any string, meaning there should be no path from an initial to an accepting state.



This completes the systematic approach to determine subsethood.

Chapter 5

DFA Minimisation

5.1 Minimum States in a DFA

So, far we have dealt with DFA, NFA without ϵ, NFA with ϵ and Regular Expressions. We have also seen that all of them are equivalent and inter-convertible and represent regular languages.
Consider the language which consists of all strings which are terminated by one. The regular expression for this will be: $(0+1)^*1$. Here is a 2-state DFA for the same language:

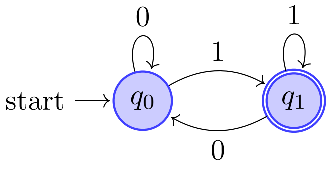


Figure 5.1: 2-state DFA for the language

The above DFA has two states q_0 and q_1. q_0 is reached when the last seen letter was 0 (also at the start). q_1 is reached when the last seen letter was 1 (also, it is an accepting state).
We can even construct a 4-state DFA for the same language. Here is an example:

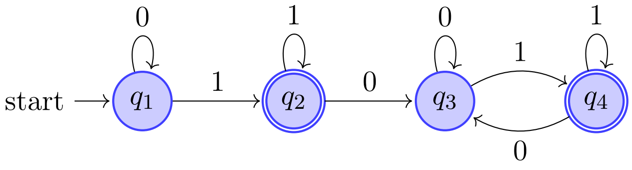


Figure 5.2: 4-state DFA for the language

The above DFA has four states q_1, q_2, q_3, q_4.

- q_1: Last letter 0 and no 1s so far
- q_2: Last letter 1 and no 101 seen so far
- q_3: Last letter 0 and more than one 1s seen so far

- q_4: Last letter 1

One can construct many more 4-state DFAs for the same language. Above is another example.

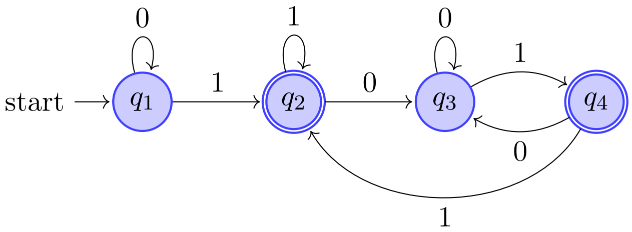


Figure 5.3: Another 4-state DFA for the same language

A natural question which comes to our mind is that can we construct another 2-state DFA for this language. One can find through trial and error, that this is not possible.
Is one state DFA possible for this language? Let us assume that this is possible. Two cases arise. If that state is accepting, then it will also accept ϵ, which is not possible. If that state is not accepting, then the language will be empty. We have arrived at a contradiction. Hence, one state DFA is not possible for this language.
Therefore, for this language, the minimum states in any DFA can be 2. We also observe that number of such 2-state DFAs is 1. So, we will now claim that for every language, the number of minimal DFAs is 1 and try to prove this. Before we prove this, we will introduce the notion of indistinguishability.

5.2 Indistinguishability

Two states of a DFA q_i and q_j are considered indistinguishable, if $\forall w \in \Sigma^*$, we start with q_i, process w and reach q_i' and we start with q_j, process w and reach q_j', then either $q_i' \in F$ and $q_j' \in F$ or $q_i' \notin F$ and $q_j' \notin F$, where F is the set of all final states of the DFA. So, we are basically changing the start states and checking whether we reach the same type of states or not through the same string.
This relation is denoted by $\equiv$. It has the following properties:

- It is **reflexive**. It is clear to see that every state is indistinguishable to itself, as it will reach a particular state on seeing w. (Due to the nature of a DFA)
- Also, it is clear to see that this relation is **symmetric**.
- This relation is also **transitive**. We can prove this by contradiction. Let us assume that $(q_i \equiv q_j) \wedge (q_j \equiv q_k)$ but $q_i \not\equiv q_k$. Then $\exists w$ such that $q_i' \in F$ and $q_k' \notin F$, where q_i' and q_k' are the states we reach from q_i and q_k respectively on seeing w. From the equivalence of q_i and q_j, we have $q_j' \in F$ but from the equivalence of q_j and q_k, we have $q_j' \notin F$, where q_j' is the state that we reach from q_j on seeing w. Hence, we have arrived at a contradiction. Therefore, this relation is transitive.
- A relation which is reflexive, symmetric and transitive, is **equivalent**. Thus the states of the DFA, on which this relation is defined can be partitioned into equivalence classes.

In the above example (Figure: 5.2), q_1 and q_3 belong to the same equivalence class, and q_2 and q_4 also belong to another equivalence class. Let us now try to construct a 2-state DFA from the above 4-state DFA example (Figure: 5.2). We will choose, one element each from both of the equivalence classes. Let us take q_1 from the first class and q_4 from the second class.

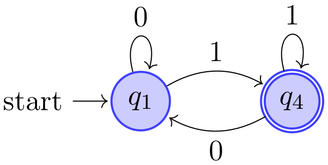


Figure 5.4: 2-state DFA constructed from the 4-state DFA

From q_1, if we see a 0, we land at q_1 itself. If we see a 1, we would have landed at q_2, but since q_2 and q_4 are equivalent, we replace q_2 by q_4. Similarly, from q_4, if we see a 1, we remain at q_4, but if we see a 0, we would have landed at q_3, but since q_1 and q_3 are equivalent, we replace q_3 by q_1. Also, q_2 and q_4 both were acceptable earlier, now q_4 is acceptable.

So, through these equivalence classes, we have minimized our 4-state DFA into a 2-state DFA, and also this 2-state DFA is structurally the same as the previous one (Figure: 9.27), thus again making us think that the claim that there is a single minimal DFA for every language might be correct.

Another interesting thing we observe is that an accepting state cannot be indistinguishable from a non-accepting state. We can take $w = \epsilon$, and observe that the states we reach from this pair of states do not satisfy the definition of indistinguishability relation. However, an initial state and a non-initial state can belong to the same equivalence class. (For example, above q_1 and q_3 belonged to the same class.)

Now, several important questions arise. How can we find the equivalence classes of this relation? When we will come to know that we cannot compress our DFA further (by compress, we mean reducing the number of states of the DFA)? How can we prove our claim that the minimal DFA will be unique?

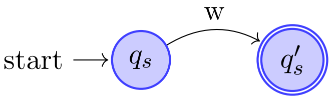
We will try to answer all these questions subsequently. Let us start with the easiest one.

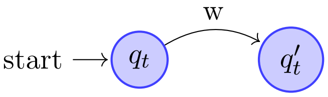
5.3 Equivalence classes of Indistinguishability relation

Now, we will develop an algorithm to find the equivalence classes of this relation.

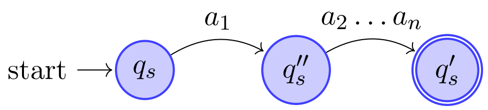
Firstly, we should keep in mind our previous observation that an accepting state cannot be indistinguishable from a non-accepting state. That is $q_i \not\equiv q_j \ \forall q_i \in F$ and $q_j \in (Q \backslash F)$, where Q is the set of all states of the DFA.

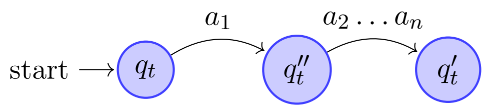
Suppose, we find two states q_s and q_t which are distinguishable. Thus there exists a string w such that q_s leads to an accepting state but q_t leads to a non-accepting state.



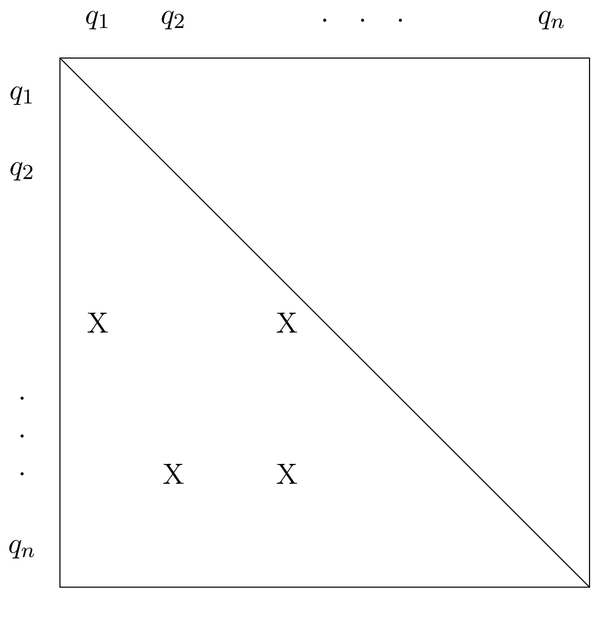


Now, w can be decomposed into $a_1a_2 \ldots a_n$, where $|w| = n$.

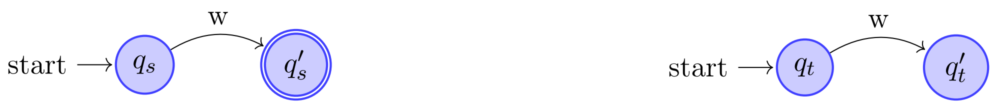




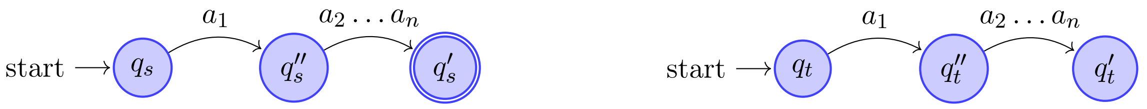
From, the above figures, one can observe that q_s'' and q_t'' are also distinguishable, where $w' = a_2 \dots a_n$ is the string which is making them distinguishable.
Thus, we can observe that for all states q_i and q_j such that $q_i \not\equiv q_j$ and for all $a \in \Sigma$, such that q_s on seeing a lands at q_i and q_t on seeing a lands at q_j then q_s will be distinguishable with q_t, where q_s and q_t are two states of the DFA.(We are basically extending $w' = a + w$.)
Therefore using a distinguishable pair, we have found another one. This is the basis of our algorithm. We initialize our set with all the pairs, where one state belongs to the set of accepting states and other state does not belong to the set of non accepting states. And then through the above step, we keep on increasing the size of this set. (Note that this algorithm is not exponential, because there are only $\binom{n}{2}$ pairs possible, and we do need to check an already visited pair.)



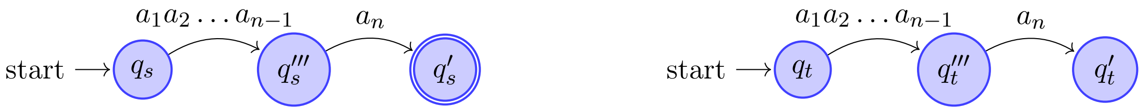
We stop the process, when no more crosses can be inserted.
Now, it is quite natural for us to ask the question that whether our algorithm is correct or not i.e. can we still find a pair of distinguishable states, which are not detected even after our algorithm finishes?
Proof: Let us assume that there are two states q_s and q_t which are distinguishable but not recognized by our algorithm. By definition, there exists a string w such that q_s leads to an accepting state on seeing w and q_t reaches a non-accepting state.



Now w can be written as $a_1a_2 \dots a_n$, where $|w| = n$. And also assume that we reach q''_s and q''_t on seeing a_1 from q_s and q_t respectively. It is clear that, if our algorithm has not detected q_s and q_t as distinguishable, then it would not even have detected q''_s and q''_t as a distinguishable pair. (Because, if it would have done, then the next step would have been to make q_s and q_t as distinguishable.)



Now, we will inductively move forward our algorithm. Thus q'''_s and q'''_t would not also be detected as distinguishable after our algorithm finishes. But clearly, this is not possible, because our algorithm initializes the set of pairs of {accepting, non-accepting} states and then it's first step is to move backwards. So, it would have marked q'''_s and q'''_t as distinguishable in the first step itself.



We have achieved contradiction. Therefore, we can safely conclude that our algorithm terminates and is also correct.

Now, once our algorithm detects all pairs of distinguishable states, we can choose equivalence classes of the indistinguishability relation. (Then, we can choose one representative from each class and move forward with our proof of existence of a unique minimal DFA.)

It is worth noting that distinguishability is not an equivalent relation. It is not even reflexive. It is also not transitive. But, it is indeed symmetric. And also, it proved out to be very useful for finding these equivalence classes.

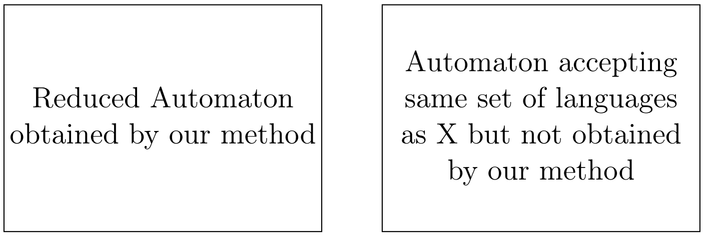
5.4 Further Analysis

We have developed an algorithm which can shrink the size of a given DFA. But is that the best we can do? Can the size of the DFA be further reduced? Are there many DFAs which are minimal or just one unique? The following sections will try to answer these questions.

Note: We are assuming that there is no redundant nodes in final DFA obtained, i.e. those nodes which can not be reached from starting state by any string. If there is one such, we can always remove it to get an equivalent DFA.

5.4.1 Optimality of Acquired DFA

Let's say we have an automaton X, which is the reduced automaton obtained by picking one state from each of the equivalence classes of the set of states, and Y is an automaton obtained by some other method for the same language.



Reduced Automaton obtained by our method

Automaton accepting same set of languages as X but not obtained by our method

Automaton X Automaton Y

We wish to show that the number of states of Y is at least as many as X.
Let's define a new term "**Language of State**". Given a DFA A and a state s, we define L_s^A as the set of all finite strings which will end in any of the accepting state if we start from s. Readers should be able to see that when we say that two states are distinguishable, we mean that there Language is different. Similarly, when we say that the two states are indistinguishable, we are actually asserting that the language of the two states is same.

Let say we started with the automaton D. After termination the resultant automaton was A. Let S_A represent the set of distinct such languages of nodes of A. Since A has all states indistinguishable from every other state, the $|S_A|$ = no of states.
Claim: Given any DFA B, equivalent to A,

$$\forall L \in S_A(\exists S(L_S^B \equiv L)), \text{ where S represent a state of B}$$

Proof: If L $\in$S$_A$, then by definetion it must be the language of some node in A, say s. Since all the nodes of A are ir-redundant, so let's say string w is one such string through which we can reach this node starting from the starting state of A. Now run the same string w on the automaton B. Say we reach the state S. Then $L \equiv L_s^A \equiv L_S^B$, because if say string x is present in former and not in latter then, string $w.x$ will be an accepting string in A and not in B, and vice-versa. Thus $L_S^B \equiv L$.
For every language in S_A, we must have at least one node in B. Since one node has a specific language, this gives us a lower bound to number of nodes, $|S_A|$.
A achives the lower bound of states, hence it is the(?) minimum state DFA which represent the same regular language represented by D.

5.4.2 Uniqueness of Acquired DFA

So far we have discussed about proving the optimality of the automaton A, which is the output of our algorithm. We defined an important notion of "Language of State" and proved a very important result which can be summarized as follow **Given two DFAs A and B where both represent some particular regular language, then for any string w, the states reached in A and B by reading it will have same language,i.e.** $L_{s_A}^A \equiv L_{s_B}^B$. Now we will try to answer that if there are mutiple DFAs which are optimal or just one.
Let's consider two automatons, one is our DFA A which has been proved to be optimal and another DFA C which is a equivalent DFA and is also optimal(given). We will show that C has to be isomorphic to A.

Since C is an optimal DFA, it can not be further shrunk. That implies two things: No redundant nodes, no pair of nodes is indistinguishable. That means that the language of the nodes of C is distinct and unique. Since A and C are both optimal, both must have same number of states, and all the nodes of both of them are reachable, and all the nodes of both of them have unique languages(unique over the domain of single DFA not collectively).
Consider any node s of A, take a string w which take us to it, run it over B, say we reach the node S, then s and S have same language. Since the nodes of B are distinguishable, so S does not depend on w.

This can be used to define an injection from the nodes of A to the nodes of B. It is injective function because of distinguishability of the nodes of A as well as of B. The function is also **bijective** because the number of nodes of A and B are both finite and equal, making it both injective and surjective, thus bijective.

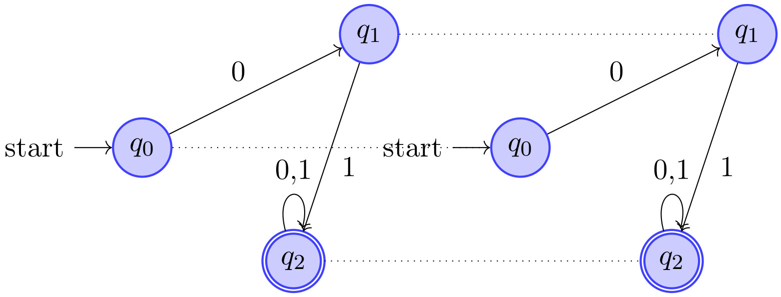


Figure 5.9: Two DFAs with corresponding states connected by dotted lines.

Claim:
As per this function, starting state of A will be mapped to starting stata of B. . This is because the languages of these two states is essentially the language of there respective DFA which is same(given).
Claim:
An accepting state will be mapped to an accepting state of B. . This is because any state which has ϵ in it's language is an accepting state by the definition of "language of state", similarly an accepting state will have ϵ in it's language. So an accepting state of A mapped to whatever state of B, that state must have ϵ in it's language making it an acceptable state of B.
Claim:
Consider Two states of A, S_1^A and S_2^A. Suppose that they are mapped to S_1^B and S_2^B respectively. Then if there is an α labelled edge going from S_1^A to S_2^A, there must be an α labelled edge going from S_1^B to S_2^B also. Consider any string w that takes us to S_1^A, we can see that $w.\alpha$ will take us to S_2^A. Now since S_1^A is mapped to S_2^A and S_1^B is mapped to S_2^B we can say that "any" string that takes us to one in A will take us to the image of it when run on B(Why? proved above:)). Thus w takes us to S_1^B and $w.\alpha$ takes us to S_2^B. Because a DFA is deterministic, there should be α labelled edge going from S_1^B to S_2^B.(Food for thought: Is mentioning determinism important here?)
Isomorphism of labelled graphs:
An isomorphism is a vertex bijection which is both edge-preserving and label-preserving.
Isomorphism of DFAs:
An isomorphism is a vertex bijection which is both edge-preserving and label-preserving, where im-

age of starting state is starting state and image of a accepting state is an accepting state.
All these three claims shows that A and B are same automatons.

This tool can be used to check wether two different regular expressions represent the same language or different languages. Just convert them into DFAs, and then to the minimal DFAs. Check for the isomorphity of the two DFAs if they are isomorphic, then the regular expressions are equivalent otherwise not. (Isomorphism $\iff$ equivalence)

5.5 From states to words

Till now we were talking about languages of states and equivalence of two states. Now we will extend this notion for words.

5.5.1 Language of word

Consider a language L. we define Language of word w, $[w]$ as set of all strings x such that $w.x \in L$. Now we will define a relation "$\tilde{}_L$ ". If languages of two words *w1* and *w2* is same for L, then we say that w1 is related to w2. This is a reflexive, symmetric and transitive relation. (Basically, if set 1 and set 2 are same and then it is also true for the other way round that is set 2 and set 1 are same. If set 1 and set 2 are same and if set 2 and set 3 are same then set 1 and set 3 will also be same.) Thus we have defined an equivalence relation over words, where the equivalence classes depends on language L.

5.5.2 Relation between states of minimal DFA and equivalence classes for $\tilde{}_L$

If a string w ends up in a state of DFA, say s, then the language of the state is equivalent to $[w]\tilde{}_L$. This can be easily proved by using definition of the "language of state" and "language of word". If a string $w.x \in$ L, then if you run the $w.x$ on DFA, you first reach that state using w then x takes to some accepting state. Hence x $\in L_s^{DFA}$. Also if x $\in L_s^{DFA}$, then essentially $w.x$ will end in a accepting state because w ends in state s.
A particular equivalence class represent a particular language and a state represent a particular language. Since for a minimal DFA, all the states represent distinct language, we can define a bijection from equivalence classes to the state of minimal DFA, where a equivalence class is mapped to that state which represent the same language as that of any string in that equivalence class.

5.6 Setting up the Parallel

We defined the Nerode equivalence relation on a language L over the alphabet Σ. This relation states that $\forall w_1, w_2 \in \Sigma^*$, $w_1 \sim_L w_2$ iff $\forall x \in \Sigma^*$, $(w_1 \cdot x \in L \iff w_2 \cdot x \in L)$

If the language L is regular, then a minimized DFA $A = (Q, \Sigma, \delta, q0, F)$ can also be defined for the language L.

For this language L, w_1 and w_2 are two words in Σ^* such that $w_1 \sim_L w_2$. Let q_i and q_j be two states $\in Q$ such that

$$q_i \rightarrow \text{state reached in A after reading } w_1$$
$$q_j \rightarrow \text{state reached in A after reading } w_2$$

Since $w_1 \sim_L w_2$, it follows that for all $x \in \Sigma^*$, the state reached in DFA A after reading $w_1 \cdot x$ and the state reached in DFA A after reading $w_2 \cdot x$ will either both belong to F or both belong to $Q \setminus F$. Otherwise, one word would be accepted by L while the other would not, leading to a contradiction in the definition of the equivalence relation.

Therefore, by the definition of indistinguishability, states q_i and q_j can be concluded to be indistinguishable.However, in a minimized DFA, all distinct pairs of states are distinguishable. Consequently, the only states in A that can be indistinguishable are those where the state is compared with itself.

This demonstrates that if two words/strings belong to the same equivalence class with respect to L, then both strings will end up in the same state $q \in Q$ in the minimized DFA.

Till now, we have defined two equivalence relations $\sim_L$ over the language L and $\equiv$ over the states of a DFA characterizing a language L. We aim to define a relation between the number of equivalence classes of both these relations.

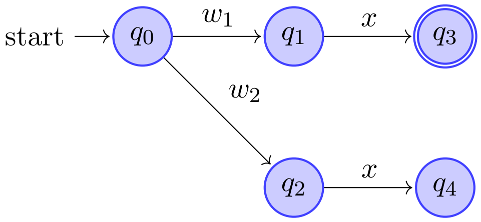
5.6.1 Can $| \sim_L | > | \equiv |$?

NO. We will in fact prove below that $| \sim_L | \leq | \equiv |$

Let there be strings w_1,w_2 s.t.

$$w_1 \in i^{th} \text{ equivalence class of } \sim_L$$
$$w_2 \in j^{th} \text{ equivalence class of } \sim_L$$
$$\text{where } i \neq j$$

Since they belong to different equivalence classes, we can say $\exists$ x $\in \Sigma^*$ s.t. $w_1 \cdot x \in L$ and $w_2 \cdot x \notin L$ (or can be vice-versa, does not matter).



- Here, q_1 represents the equivalence class of w1 while q_2 represents the equivalence class of w2.
- By the definition of indistinguishability, q1 and q2 are distinguishable states as there exists a letter in the alphabet which leads them to another pair of distinguishable states.

- Doing this for every pair of equivalence classes, we find that the states representing the distinct equivalence classes are all distinguishable from each other leading us to the following relation.

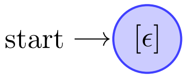
$$| \sim_L | \leq | \equiv |$$

.i.e. the number of indistinguishability equivalence classes is atleast the the number of Nerode equivalence classes

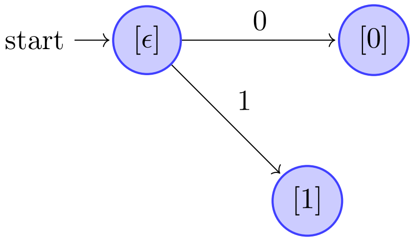
5.6.2 Can$| \sim_L | < | \equiv |$?

In order to inspect this , we are going to construct a DFA using $\sim_L \subseteq \Sigma^* \times \Sigma^*$ relation which will then accepts the language L. Let here, $\Sigma = 0, 1$

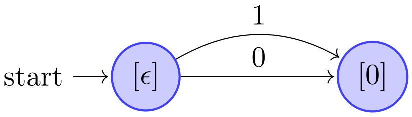
- A state denoting $[\epsilon]$ is made and is also denoted as the start state.



- Then we pick a letter from the alphabet and look the transitions from each of the existing states. If the next word already lies in the equivalence class of one of the existing states, we draw the arrow representing the particular transition otherwise a new state is made representing the equivalence class of the newly made word.

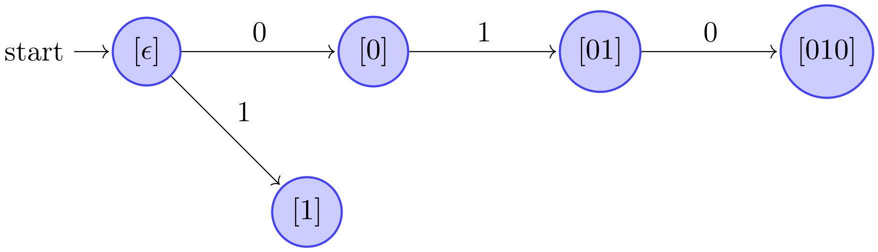


Here, 1 does not lie in the equivalence class of 0, i.e., [0].



If 1 lies in [0], then the automata will look like this.

- By repeatedly applying point 2, the automata can keep on expanding like as below



- However, this construction is guaranteed to converge because we have already proved that $| \sim_L | \leq | \equiv |$. It is known that the cardinality of equivalence classes of the indistinguishability relation is equal to the number of states in the minimal DFA representing the language L.Since the number of such states is finite, it follows that $| \equiv |$ is also finite.Given that $| \sim_L | \leq | \equiv |$, it can be deduced that $| \sim_L |$ is finite. Consequently, this guarantees the convergence of the algorithm in question.[If the above algorithm does not converge, that means new states will keep on forming contradicting the $| \sim_L | \leq | \equiv |$ identity.]

- Finally, all those states whose equivalence classes have words that are accepted by the language L are marked as accepting states.

Hence, the aforementioned algorithm successfully allows us to construct a finite state automata(DFA) which accepts only the words that are accepted by the language L. Note that here the number of states in the new automata is equal to $\sim_L$.
Now, let us assume that $| \sim_L | < | \equiv |$ holds. This implies that our newly constructed automata is the minimized DFA for the language L.
However, in the last lecture we had proved that the DFA constructed by $\equiv$ relation is minimized one and is unique. Hence, it leads to a contradiction,making our assumption wrong.

Thus proved that

$$| \sim_L |= | \equiv |$$

Note: The Nerode equivalence relation,unlike the indistinguishability relation,can be defined for any language $L \subseteq \Sigma^*$, irrespective of the fact that whether the language is regular or not.

5.7 Myhill-Nerode Theorem

L is regular if and only if $\sim_L$ has a finite number of equivalence classes.

This theorem provides an exact characterization of a regular language, unlike the Pumping Lemma. While the Pumping Lemma does not guarantee that L is regular if it holds, here, if $\sim_L$ has a finite number of equivalence classes, the language L must be regular. The proof of this theorem can be found here.

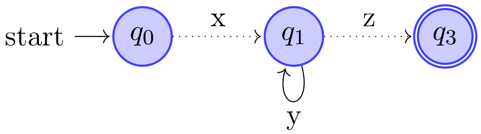
Chapter 6

Pumping Lemma for Regular Languages

If L is a regular language, then there exists a finite DFA with a minimum number of states (say p) that recognizes it. Let's consider a string w of length at least p that is accepted by this DFA. By the Pigeonhole Principle, we can deduce that when the DFA processes the string, it must revisit at least one state, thereby implying the existence of a loop.

6.1 Example

Consider string $w \in L$, where L is a regular Language. Suppose the first state in DFA which is revisited again on processing string w is q_1.



Hence for any $i \geq 0$ $xy^iz \in L$
The length of substring xy cannot exceed the size of the DFA. Hence,

$$|xy| \leq n$$

where n is the length of DFA
Also $|y| \geq 1$ as it should be possible to go back to the same state

6.2 Formal Statement of Pumping Lemma

For any regular language L, the following holds

$$\exists\, p > 0\, \forall w,\, (w \in L) \cap (|w| \geq p)\, \exists\, x, y, z\, (w = x.y.z)\, \forall n\, (n \geq 0) \Rightarrow xy^nz \in L$$

Contrapositive:

$$\forall\, p > 0\, \exists\, w\, (w \in L) \cap (|w| \geq p)\, \forall\, x, y, z\, (w = x.y.z) \exists\, n (n \geq 0) \cap xy^nz \notin L$$

If the above formula holds true then L is not a regular language.

- If a language L is regular then Pumping Lemma holds true for L but if Pumping Lemma holds true for Language L then it does not necessarily mean that L is regular.
- But if Pumping Lemma does not hold true for Language L then L is not regular.

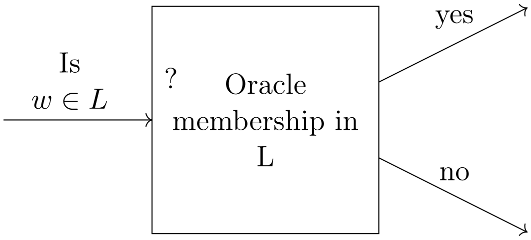
6.3 Pumping Lemma as Adversarial Game

Game between an adversary (who wants to show that the language L is not regular) and a believer (who believes that the language is regular) is as follows:

- The believer chooses an integer $p > 0$ and claims this is the count of states in the DFA that she believes recognizes the language.
- Adversary chooses a string $w \in L$ such that $|w| > p$.
- Believer then splits w into three parts $w = xyz$, where $|xy| \leq p$ and $|y| > 0$
- Adversary now chooses an integer $n \geq 0$ such that $xy^nz \notin L$ thereby winning the game. If the adversary can't do this then they lose.

6.4 If a language L is regular and the number of states in DFA for L is n, is L infinite?

Given the DFA for language L, if there is a path from the initial state to a state forming a cycle, and from that state there's a path to an accepting state, then L is infinite because the DFA can endlessly loop within this cycle, generating an infinite number of accepted strings.
Instead if only a black box is provided, which can determine whether a given string w is in L or not.



Test all strings w, such that $n < |w| \leq 2n$ for membership in L. If we find even one string which returns 'yes' on testing then L is infinite. (Since pumping lemma can be applied to such a string, and it can also be pumped to show that an infinite sequence of strings are in L).
If all strings w, ($n < |w| \leq 2n$) return 'no' on testing then we claim that there are no strings in L of length more than $2n$, and thus there are only a finite number of strings in L.
Proof: On applying the pumping lemma for such w repeatedly, we can obtain strings of length between n and $2n$, which again belong to L. This is because, for $w = xyz$, removing loop y every time decreases the length by at least 1. This is a contradiction to our assumption that there are no strings of length between n and $2n$ in L.
Hence L is infinite if it contains atleast one string of length at least $n + 1$.

6.5 Example

Consider the Language $L = \{(0+1)^{*01^k \mid k\ is\ prime\}}$ with alphabet $\sum = \{0, 1\}$
The application of the Pumping Lemma is independent of the initial state chosen such that length of string remains $\geq p$ (pumping length)
In the case of this language, the Pumping Lemma can indeed be applied to the substring 1^k, where k is any prime number.
Now we can break $w = 1^k$ into $w = xyz$ such that $y \neq \epsilon$ and $|xy| \leq n$. Let $|y \models m$ ($m > 0$), then $|xz \models k - m$
Consider the string a$=xy^{k-m}z$

$$|a \models |xz| + (k-m)|y| = (m+1)(k-m)$$

As length of string a is not a prime (it has factors $m+1$,$k-m$), we have $a \notin L$
Hence Language L is not regular.
Reverse Language
If Language L is regular then its reverse language L^R is also regular.
To construct automaton for L^R from that of L swap initial and final states and reverse the edges.

Chapter 7

Pushdown Automata

7.1 Pushdown Automata for non-regular Languages

Thus far, the finite automata we have studied cannot be used to represent non-regular languages. One such example of a non-regular language is the language of balanced parentheses.Let it be L.

> **Proving why L is not a regular language** *It is known that intersection a language L with any regular language preserves the regularity of the original language.i.e. the new language formed after intersection will hav ethe same regularity as the original language L.*
> *Therefore, considering $L \cap (^*)^*$ results in $(^n)^n$ which by applying the Pumping Lemma can be shown that that it is not a regular language, it can be thus concluded that the language of balanced parenthesis is also not a regular language.*

So if we can equip the finite state automata with more structures like a stack, it will be abl to accept the non-regular languages too.

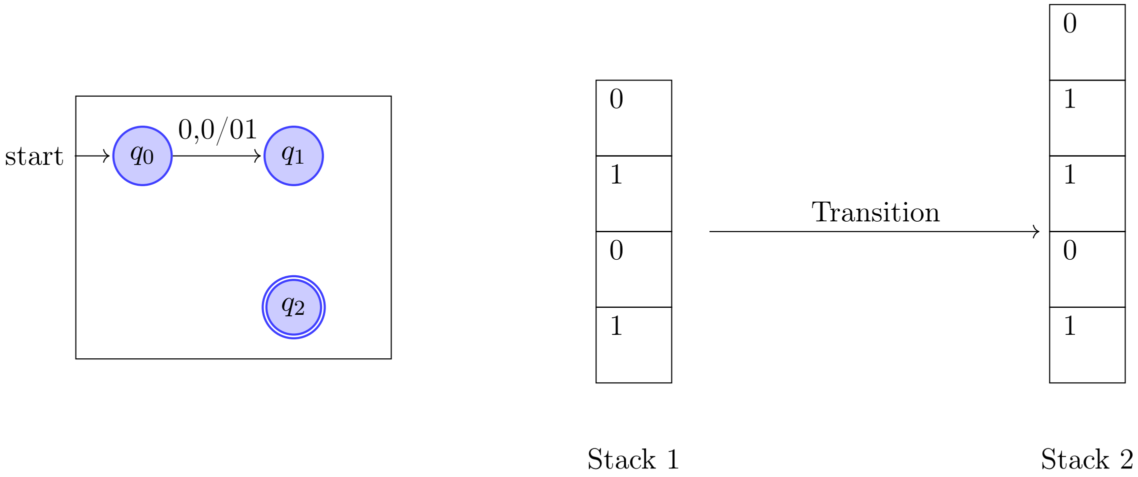


Figure 7.1: Automaton and Stacks

7.2 Description

PushDown Automata (PDA) are essentially NFAs equipped with a stack that is maintained on an alphabet that is different from the alphabet used for state transition. PDAs have greater accepting power than regular NFAs and can accept non-regular languages as well.

7.2.1 Conventions

Just like we described a NFA L as $L(Q, \Sigma, q_0, \delta, F)$, we describe PDAs as characterized by

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

where-

- Q is the set of all states in the PDA
- Σ is the alphabet accepted by the PDA
- Γ is the set of symbols that are pushed/popped from the stack associated with the PDA
- q_0 is the starting state of the PDA. There may be more than one starting state for the PDA.
- Z_0 is the starting symbol in the stack. This is so that a symbol can be popped on the first transition.
- δ is the transition function for the PDA. While the transition function for regular NFA was of the form $\delta : Q \times \Sigma \cup \{\epsilon\} \rightarrow 2^Q$, the transition function for PDAs is characterized by $\delta : Q \times \Sigma \cup \{\epsilon\} \times \Gamma \rightarrow 2^{Q \times T^*}$, where Γ^* is the set of strings made by the alphabet Γ.
- F is the set of final states of the PDA

A single transition on a PDA from state q_1 to q_2 is represented by $x, a/ba$ where $x \in \Sigma$ is the original symbol and may be ϵ, a is the top element of the stack and is popped off (it cannot be ϵ), and ba is the string that is pushed onto the stack after a is popped off. The final top element of the stack is b after this transition takes place.
This places a restriction on PDAs as compared to NFAs, that is, for any transition to occur, the top element of the stack also has to match the top element specified in the transition.

7.2.2 Example

Let us characterize the non regular language $\{0^n 1^n | n \geq 0\}$ through a PDA with a finite number of states, as shown in Figure 7.2. We define $\Gamma = \{z_0, a\}$ as our stack alphabet with our PDA starting with only z_0 in the stack. a will represent a counter for the number of zeros in our string.
Notice that if a string deviates from the form $0^m 1^n$ the automaton goes to trap. For every 0 in the string a is pushed onto the stack and hence the $l(S) = m + 1$ where $l(S)$ is the length of stack. For strings following the given structure, if $m = n$, then the string ends at an accepting state. If $m > n$, the string goes to trap and if $m < n$, the string ends the run at a non accepting state.
We reiterate that a string is accepted if and only if the run does not halt before the string is completed and the final state is an accepting state.

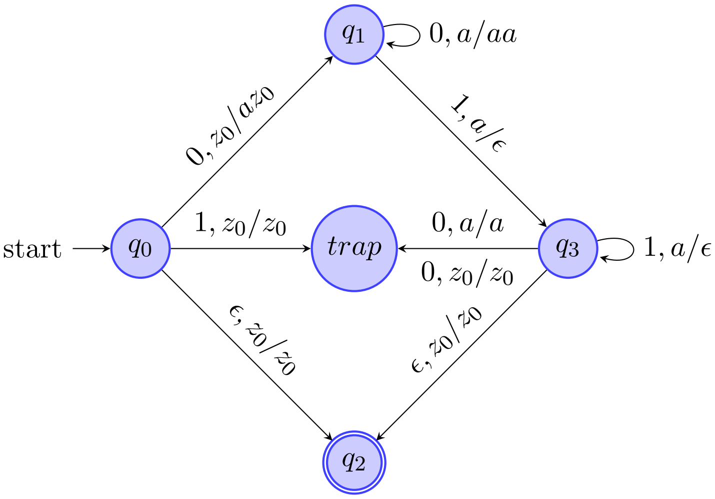


Figure 7.2: Automaton accepting $\{0^n 1^n | n \geq 0\}$

7.2.3 Example

Now let us extend our previous automaton to accept the language $\{0^n 1^m | n \geq m \geq 1\}$ as shown in Figure 7.3. Herein, we introduce a different notion of string acceptance, that is, if the stack becomes empty upon completion of the run, we can say that the string is accepted. This is called **acceptance through empty stack**.

7.3 Acceptance through empty stack

We introduce a new way of characterizing string acceptance in PushDown Automata wherein there is no need to specify accepting states explicitly. Here, a string is accepted if it satisfies the following constraints for atleast one possible run on the PDA-

1. String should not halt in between its run, that is, if any symbol is left in the run on the string, there must exist atleast one entry in the transition function δ corresponding to any of the possible current states, the next character and any of the possible stack tops.

2. Upon completion of the run, either the stack should be naturally empty or through a series of ϵ transitions, should be able to make its stack empty for the string to be accepting.

3. There are no constraints on the current state and the string can be accepted through any state $q \in Q$.

This gives a new method of acceptance. We represent the language accepted by this method over an automaton A as $N(A)$ and the language accepted by our original method as $L(A)$.
However, we do not yet know if the two methods of acceptance are equally powerful. We shall prove this claim by showing the equivalence of the two methods over PDAs by transforming one into the other and vice versa.

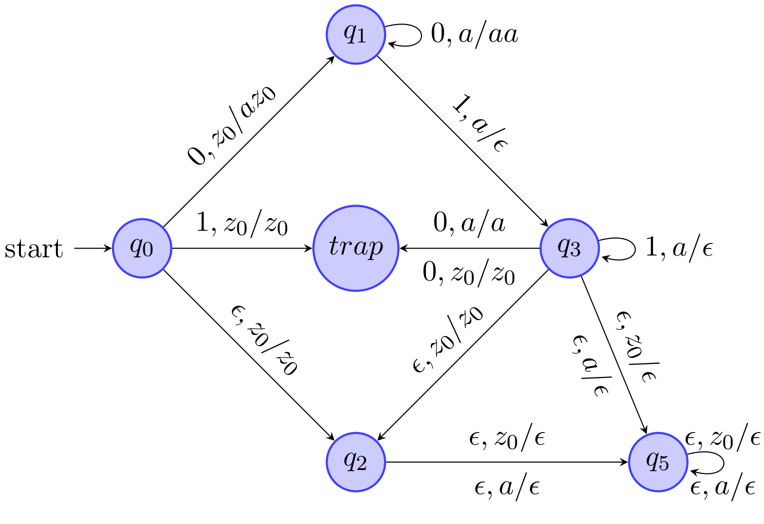


Figure 7.3: Automaton accepting $\{0^n 1^m | n \geq m \geq 0\}$

7.3.1 For any automaton A, $L(A)$ is not necessarily equivalent to $N(A)$

We must observe a subtlety, if a language L is accepted through PDAs through both methods, the automata that accept these need not be the same. This is easily shown through the previous two examples.

- In Figure 1, $L(A) = \{0^n 1^n | n \geq 0\}$ but $N(A) = \Phi$ as the stack never gets empty.
- In Figure 2, $N(A) = \{0^n 1^m | n \geq m \geq 1\}$ but $L(A) = \Phi$ as their are no accepting states.

So we want to prove that-

1. If $L(A_1)$ is the language accepted by a PDA A_1, there exists another PDA A_2 such that $L(A_1) = N(A_2)$.
2. If $N(A_1)$ is the language accepted by a PDA A_1, there exists another PDA A_3 such that $N(A_1) = L(A_3)$.

We will show the existence of both of these by giving a construction which when applied to A_1 gives us A_2 and A_3 respectively.

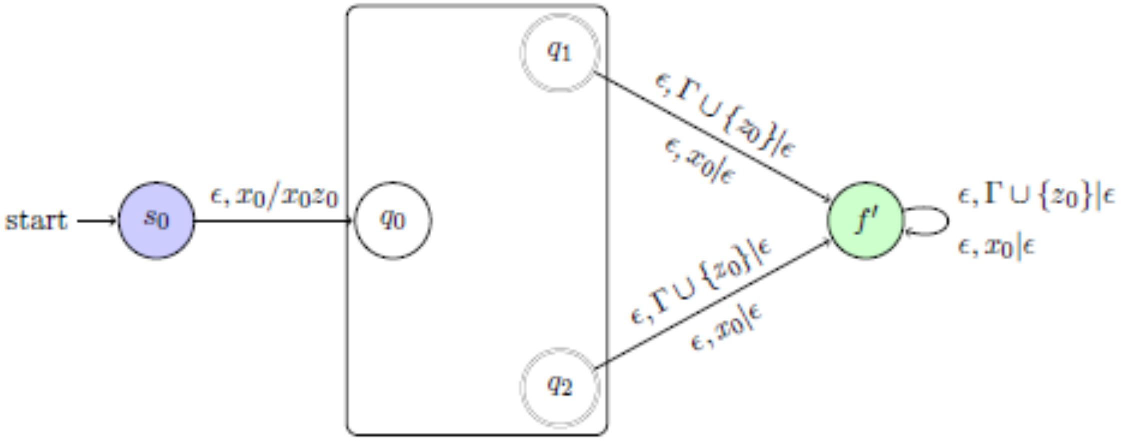
7.3.2 Construction for A_2 such that $L(A_1) = N(A_2)$

We need to modify A_1 to A_2 without making any assumptions about its structure except that it is a PDA, and make it such that it satisfies two conditions for the equivalence of the languages-

1. Rejection - $N(A_2)$ should not accept any string s that is not a part of $L(A_1)$
2. Acceptance- If a string s is a part of $L(A_1)$ it must also be a part of $N(A_2)$

We shall assume A_1 has only one starting state, and if not we can simply add extra ϵ-transitions to reach any start state from our original start state.

Before this start state, we append a new start state s_0 and have our stack start with $x_0 \notin \Gamma \cup \{z_0\}$ and connect it to start with an ϵ-edge pushing z_0x_0 to the stack. This is absolutely identical to the original stack with the exception of an extra x_0 at the bottom of the stack.
We will convert A_1 to A_2 without modifying the basic structure of A_1 such that for any accepting state in A_1, we add ϵ-transitions to a new final state f', which does not take in any input word and just pops the stack (x_0 or z_0 or $\gamma \in \Gamma$) without performing any additions onto it.



Acceptance

If string s is accepted in A_1 it will reach $f \in F$ on atleast one of its possible runs on A_1. If the stack has atleast one element upon reaching f, the string can go to f' and there the stack can get emptied through ϵ-transitions from f' to itself, thus putting s in $N(A_2)$ as well.
If the stack became empty when the string s ran on A_1 we will now have x_0 as the only element left in the stack (as no transition in the original automaton can pop x_0 because $x_0 \notin \Gamma \cup \{z_0\}$). Our automaton proceeds to f' by popping x_0 and is accepted as stack is emptied.

Hence, acceptance is proved as any string in $L(A_1)$ is also present in $N(A_2)$.

Rejection

A run of string s is rejected in $L(A_1)$ if-

1. Run successfully completes but the string is not in an accepting state
2. Run does not complete successfully because of empty stack
3. Run does not complete successfully because of no valid transitions

We show that our modifications can correctly reject all such cases.
If the run does not successfully complete on A_1 due to an empty stack, upon running on A_2, the stack will only have x_0 and as $x_0 \notin \Gamma \cup \{z_0\}$, there is no transition with x_0 as top, leading to the run halting and string being rejected.
If the run does not successfully complete on A_1 due to no valid transition, there will not be any valid transitions in A_2 as well because the stack is the same except for an extra x_0 at the bottom.

If the run successfully completes but the string is not in an accepting state, then the stack never becomes empty because x_0 will always be present.

Hence, rejection is also proved as no string rejected by $L(A_1)$ is accepted by $N(A)$.

7.4 From Empty Stack PDA to Final State PDA

We will now see how to construct a PDA with final state acceptance from a PDA with empty stack acceptance.
Consider PDA P_N = (Q, Σ, Γ, δ_N, q_0, Z_0) and N(P_N) be the set of strings that the PDA accepts by empty stack .

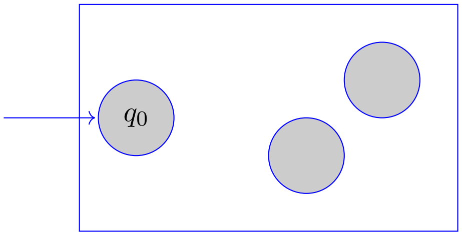
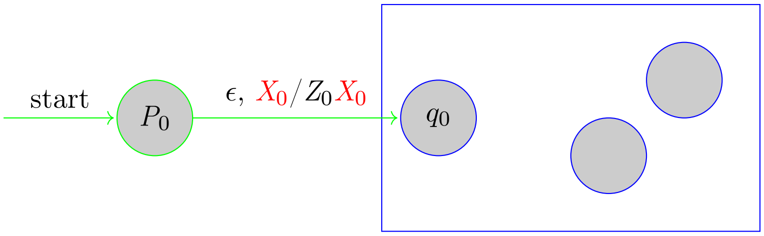


Figure 7.4: P_N - PDA with empty stack Acceptance

7.4.1 Construction

Now, Consider a construction to P that involves

- Addition of new stack variable $X_0 \notin \Gamma$ as the bottom of the stack.
- Addition of a new start state P_0 and an ϵ - transition from this new start state on reading X_0 on the stack to q_0 and Z_0 as top of stack.



- Introducing a new final state P_f and ϵ-transitions from all states in Q such that on reading X_0 on the stack transition happens to state P_f and an empty stack.

After the construction, the new automaton P_F will be $(Q \cup \{p_0, p_f\}, \Sigma, \Gamma \cup \{X_0\}, \delta_F, p_0, X_0, \{p_f\})$ whose acceptance is by final state.

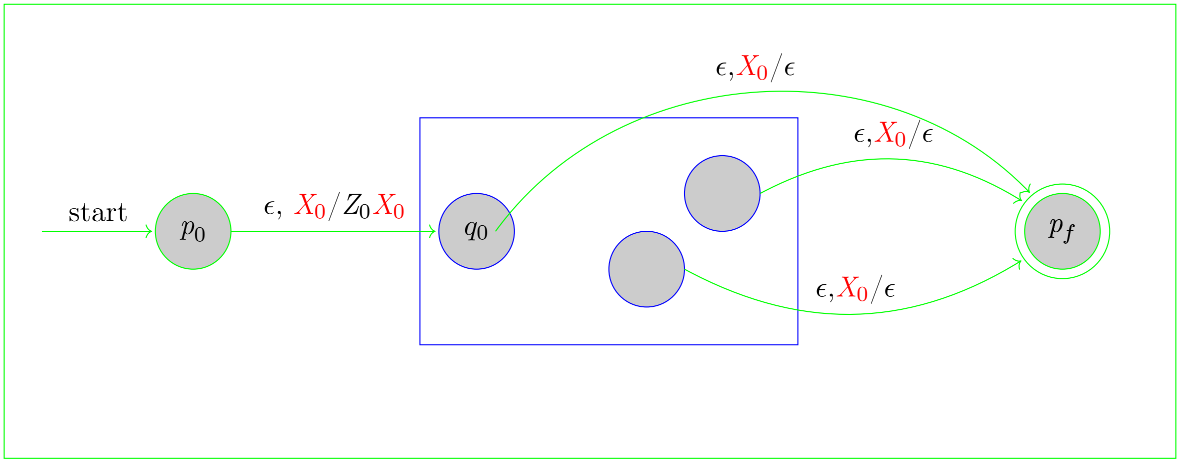


Figure 7.5: P_F (After the construction)

7.4.2 Required to show $N(P) = L(P_F)$

- First, Let us show that if a string w is accepted by P then it will also be present in $L(P_F)$

  If w is accepted by P, then the stack in P must have been emptied after consuming w, which implies that stack in P_F now has X_0 at its top. Now , taking the ϵ-transition by reading X_0 will lead us to p_f which is an accepting state of P_F.Hence w will also be present in $L(P_F)$.

- Now, Let us show that if w is in $L(P_F)$ then it will also be present in $N(P)$

  In the PDA $L(P_F)$, the only transitions that can be used to reach p_f state are ϵ transitions on reading X_0. Since w is present in $L(P_F)$, it must have taken one of these transitions and read X_0 at the top of the stack which implies that stack in P must have been emptied by w. Hence, w would also be accepted by P.

Finally one can say that,
Acceptance of PDA using final states $\equiv$ Acceptance of PDA using empty stack.
i.e For every PDA A whose acceptance is by final states there exits a PDA A' whose acceptance is by empty stack and vice-versa.

7.5 Context Free Languages

Languages accepted by the PDAs are called Context Free Languages. These are strict superset of regular languages.

7.5.1 DPDA and NPDAs

A PDA is considered deterministic if, from a given state and upon reading a specific symbol from the top of the stack, there exists only one possible transition for a given input and a PDA is considered non-deterministic if there exists either no transition or many transitions possible for an input.

The language that can be represented using DPDAs can also be represented using the NPDAs.i.e

$$DPDA \subseteq NPDA$$

But, there are certain languages which can be represented only by NPDAs but not DPDAs.Here is an example,

$$L = \{ww^R \mid w \text{ is in } (0+1)^{*\}}$$

If we try to represent L using a DPDA by pushing a copy of input symbol seen on to stack, one has to make a guess every time whether input has reached end of string w or not so that popping can be done to check the palindrome nature of the string. Hence non-determinism has to be involved to represent L.
Hence,

$$DPDA \subset NPDA$$

7.5.2 Build-up and emptying of the stack of a PDA

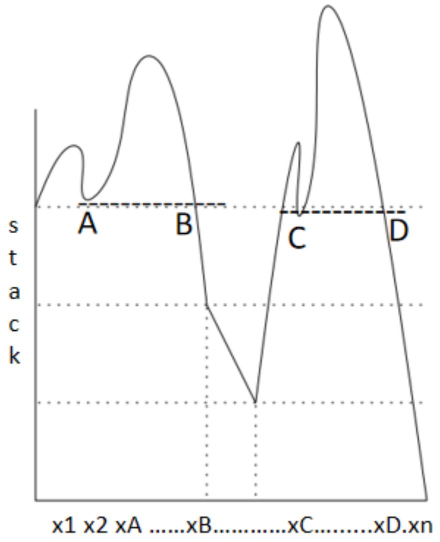


Figure 7.6: Behaviour of stack with the input being processed

In the above Figure, if we observe the part of graph where strings xA...xB and xC....xD are read, the behaviour is independent of the content of the stack before xA but just depends on the state and top of the stack when the symbol xA is being read.
Hence, a context-free language can be fragmented into some finite no.of sets (these sets need not be finite) each defined by (*top of stack*,*state of automaton*)

7.6 Acceptance by PDA

Let us begin by taking the example of the following PDA with $\Gamma = \{Z_0, X\}$ as the stacks symbols and Z_0 as the initial stack state

The language accpeted by this automata is $N(A) = \{0^n1^n | n \geq 1\}$

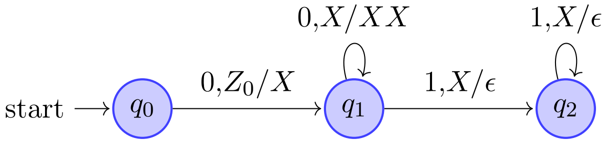


Figure 7.7: PDA A

7.6.1 Run on the PDA

Let us have a look at the run of the string 0011 on this PDA

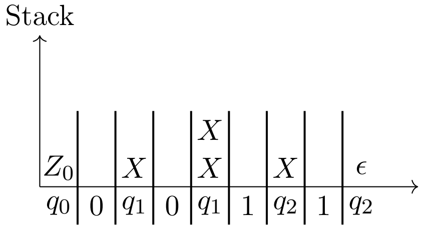
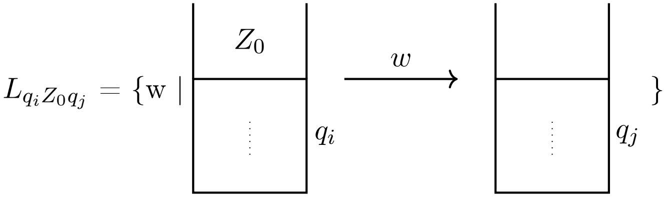


Figure 7.8: Run on 0011

We can see how the string 0011 starts with the state q_0 and Z_0 initially on the top of the stack and ends with an empty stack. Let us define a mathematical notation for the language that the string 0011 belongs to.

$L_{q_iZ_0q_j}$ = set of all strings that start at the state q_i with Z_0 on the top of the stack and end at the state q_j with Z_0 popped off the stack and the remaining stack unaltered

Note: The stack may have risen or fallen during the transitions but finally only Z_0 is popped off the stack and the remaining stack is unaltered



For PDA A we can say:

- 011 $\in L_{q1Xq_2}$
- 1 $\in L_{q_2Xq_2}$
- 0011 $\in L_{q_0Z_0q_2}$
- and many more ...

Now lets try to define a language on PDA A, N(A) such that it starts at state q_0 and an initial stack containing Z_0 and empties it.

$$N(A) = L_{q_0Z_0q_0} \cup L_{q_0Z_0q_1} \cup L_{q_0Z_0q_2}$$

We took the union of languages that started at q_0 emptied the stack and ended at any of the states present in the PDA A.

7.6.2 Recurrence Relations on Languages

Lets take a look at the individual transitions of PDA A

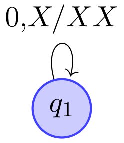


Figure 7.9: self loop on q_1

Now what can we say about the language $L_{q_1Xq_2}$

If we consider a string w such that 0 is the first letter of w then this transition is undertaken and we have added another X to the stack. So now w would have to pop two X from the stack and arrive at the state q_2 for w to belong to the language $L_{q_1Xq_2}$. This can be written as

$$L_{q_1Xq_2} \supseteq 0.L_{q_1Xq_0}.L_{q_0Xq_2} \cup 0.L_{q_1Xq_1}.L_{q_1Xq_2} \cup 0.L_{q_1Xq_2}.L_{q_2Xq_2}$$

Now if we consider the transition

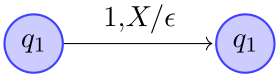


Figure 7.10: self loop on q_1

We can say

$$L_{q1Xq_2} \supseteq 1$$

We can keep doing this for all transitions from q_1 and get a super set recurrence relation for $L_{q_1Xq_2}$

7.7 More on recurrence relations

Talking about a general PDA with n states and k stack symbols n^2k languages of the form $L_{q_iS_jq_k}$ can be defined and considering individual transitions super set recurrence relations can be obtained for them. For example

$$L_1 \supseteq 1 \cup 0.L_2.L_3 \cup \dots$$

$$\vdots$$

$$L_{n^2k} \supseteq 0.1 \cup 0.L_1.L_3 \cup \dots$$

7.7.1 Smallest and Largest Languages

Consider the relation as follows

$$L_1 \supseteq 0.1 \cup 0.L_1.1$$

Notice how both Σ^* and $0^n1^nn \geq 1$ are both languages that satisfy this relation but Σ^* is largest language that satisfies this relation and $0^n1^nn \geq 1$ is the smallest(can be proven smallest by induction if 0^i1^i belongs to L_1 then so does $0^{i+1}1^{i+1}$)

We will be particularly interested in finding these smallest languages satisfying the relation

NOTE: Σ^* is always a solution(largest) of the super set recurrence relation since it contains all strings

7.7.2 Context Free Grammar

Above equations imply that wherever we see a word of the language $L_{q_1Xq_2}$, we can replace it with any word from the languages on the RHS. Thus we can write:

$$L_1 \rightarrow 0L_2L_3 \mid 0L_4L_1 \mid 0L_1L_5 \mid 1$$

where $L_1 = L_{q_1Xq_2}$, $L_2 = L_{q_1Xq_0}$, $L_3 = L_{q_0Xq_2}$, $L_4 = L_{q_1Xq_1}$ and $L_5 = L_{q_2Xq_2}$.
Similarly, we can do the same for the languages $L_i \forall i \in \{1, 2, \ldots, n^2k\}$. This will form a **context free grammar**.

WIth proper reductions, we can say that the context free grammar generated for the language $L = \{0^n1^n \mid n \geq 1\}$ is:

$$S \rightarrow 0S1 \mid 01$$

We observe that the universal language $\Sigma^{*=L((0+1)^*)}$ also satisfies the context free grammar, but the language $L = \{0^n1^n \mid n \geq 1\}$ is the minimal language that satisfies the context free grammar.

Chapter 8

Context Free Grammer

8.1 Context-free Grammar

Definition: A context-free grammar (CFG) is a formal grammar whose production rules can be applied to a nonterminal symbol regardless of its context. In particular, in a context-free grammar, each production rule is of the form $V \rightarrow (V \cup T)^*$. Where V is set of Non-Terminal and T is set of Terminals.

Formally, a context-free grammar can be represented as follows -

$$G = (V, \Sigma \cup \{\epsilon\}, P, S)$$

where
V - is the set of **non-terminals**. These are symbols that can be replaced or expanded.
$\Sigma \cup \{\epsilon\}$ - is the set of **terminals**. These are symbols that cannot be replaced or expanded further.
P - are the set of production rules
S - is the start symbol ($S \in V$)

Again, a grammar is said to be the Context-free grammar only if every production is in the form of

$$G \rightarrow (V \cup T)^{*, \quad \text{where } G \in V}$$

Let's consider a context-free grammar defined as follows:

$$S \rightarrow A.S \,|\, \epsilon$$
$$A \rightarrow A1 \,|\, 0A1 \,|\, 01$$

Here, S and A are non-terminal symbols representing languages. ϵ is a special symbol representing an empty string and is in the language S. The symbols 0 and 1 are terminal symbols and are in the language A. The set of strings that can be generated using a context-free grammar is called a *context-free language.* This language of **A** contains strings that are of the form 01 or A1 or 0A1, by just substituting all possible strings of A in the form recursively

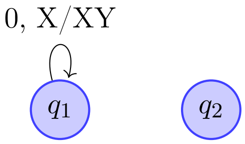
$$A \rightarrow 01 \mid 011 \mid 0011 \mid 0111 \mid 00111 \mid ...$$

Thus, Language derived from **A** can be intutively be told as

$$0^i1^j \text{ , where } j \geq i \geq 1$$

8.1.1 PDA to CFG

In the previous lecture, we talked about the language accepted by a Pushdown Automata (PDA), starting from a state q_i, with a symbol X on the top of the stack, and reaching a state q_j, with the stack remaining the same, except the fact that the top element, X, was popped off.
Such a language is denoted by $L_{q_i X q_j}$, where q_i is the start state, X is the top element of the stack, and q_j is the final state. Given certain transitions in a PDA, we could also come up with recurrence relations for such languages. For eg., if the following states and transitions were given to us, and we want to find $L_{q_1 X q_2}$



We can say that, for all states q_i in the automaton,

$$L_{q_1 X q_2} \supseteq 0 \cdot L_{q_1 X q_i} \cdot L_{q_i Y q_2}$$

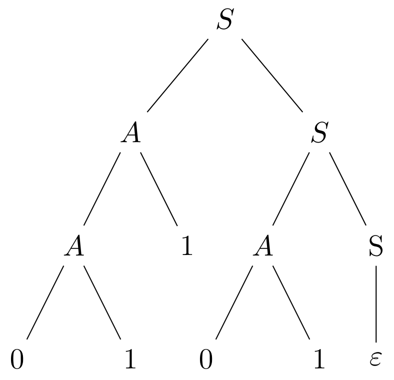
If we denote the languages by $L_1, L_2 \dots L_{n^2 k}$, where n is the number of states and k is the number of stack symbols, then we can write the above equation as a recurrence relation for all languages. For example,

$L_1 \rightarrow 1 \mid 0 \cdot L_2 \cdot L_3 \mid 0 \cdot L_4 \cdot L_1 \mid \dots$
$L_2 \rightarrow \dots$
$\vdots$
$L_{n^2 k} \rightarrow \dots$

This set of recurrence relations is called a Context-free grammar. Hence, given a PDA, we can write the language accepted by the PDA by an empty stack as a CFG.

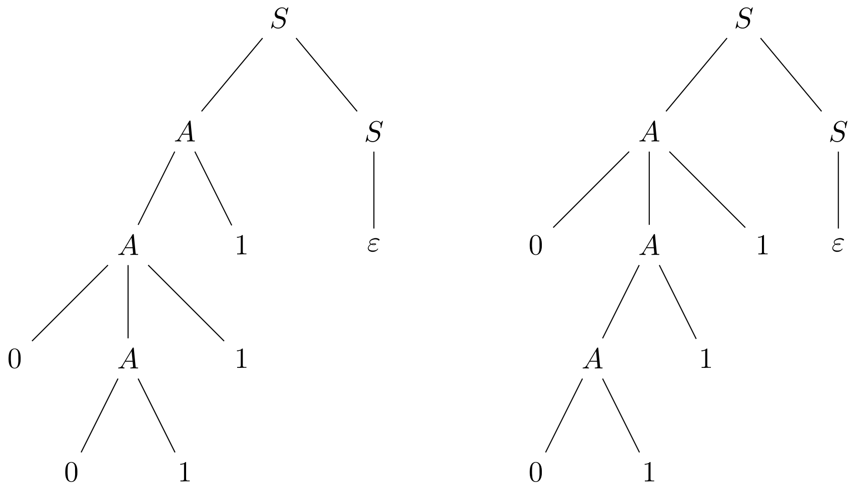
8.2 Parse Trees and Ambiguous Grammars

Suppose we have the string 01101 and we want to check whether this lies in L_S.
First, we will apply the rule $S \rightarrow A \cdot S$, since the rule $S \rightarrow \varepsilon$ is not useful yet. Now, we further see $A \cdot S$ recursively. For S, $S \rightarrow \varepsilon$ is not useful since 01101 does not lie in L_A. In this manner, we further recursively break up our string until we reach terminals and check whether the sub-parts can be further be made useful. In this manner, we can create parse/derivation trees for a string, given a grammar. The leaves in these trees are our terminals. This is shown in disgram below.



As we can see, the leaves form 01101ε, from left to right, which is exactly the string we wanted. Hence, $01101 \in L_S$, using the production rules in the grammar.
While drawing the trees, we must ensure that the root is the start symbol, and the node at which we want to split our tree, we are using production rules with that value in the left hand side. For example, if we want to split the tree at an A, we must only use production rules of the form $A \rightarrow \ldots$.

For this particular string and grammar, only one parse tree is possible. Let us now consider the string $00111 \in L_S$.



As we can see, these are two completely different, and correct parse trees for the string in the same grammar. Such grammars, where there exist multiple parse trees for some strings, are called ambiguous grammars. Hence, the grammar we have defined is indeed ambiguous.

Are there CFLs for which every CFG representing it are ambiguous grammars? Yes, there indeed are such CFLs. Such languages are called inherently ambiguous languages. If there exists even one grammar for the CFL which is unambiguous, then the language is an unambiguous language as well. Is L_S inherently ambiguous or is it an unambiguous language?

The answer is that it is not inherently ambiguous. The intuition is that, we must force our grammar to first match up all the zeros in the strings with ones, and following that, add the terminating ones.

Our current grammar puts no such restriction. We could first add zeros and ones to the start and end, add ones to the end, and then go back to adding zeros and ones to the start and end. This leads to multiple parse trees. An example of a CFG representing L_S, which is unambiguous is

$S \to C \cdot S \mid \varepsilon$
$C \to A \cdot B$
$A \to 0 \cdot A \cdot 1 \mid 01$
$B \to 1 \cdot B \mid \varepsilon$

As we can see, A does the job of adding zeros and ones to the start and end, B does the job of creating strings of ones, and C does the job of concatenating A and B.
This is important, because every programming language is specified using CFGs, for which we can build compilers and interpreters, and it is important for these CFGs to be unambiguous since we do not want multiple interpretations of a program.

8.2.1 CFG to PDA conversion

Consider the Context-free grammar defined as follows

$$\begin{aligned} S &\to AS \mid \epsilon \\ A &\to A1 \mid 0A1 \mid 01 \end{aligned}$$

Where,
Starting Symbol: S
Production Rules: $S \to AS, \quad S \to \epsilon, \quad A \to A1, \quad A \to 0A1, \quad A \to 01$
Non-terminals: $\{S, A\}$
Terminals: $\{0, 1\}$

Now Consider the following 2 alphabets,

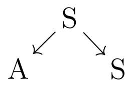
$$\Sigma = \{0, 1\}, \quad \Gamma = \{S, A, X_0, X_1\}$$

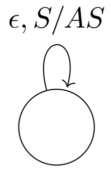
where Σ represents the alphabet of the PDA which we are going to construct and Γ represents the alphabet of the stack.
Note that we have exactly same number of symbols as total number of terminals and non-terminals in our CFG each symbol corresponding to each terminal or non-terminal.

We will start constructing our PDA by adding some edge(s) to a single node, pushing or popping some letter from the stack while constructing derivation tree/ Parse tree by using each production rule.

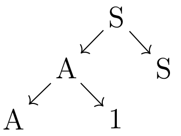
Step 1:
We initialize our stack by pushing S as S is our starting symbol and we represent it as follows ,Where left means bottom of the stack and right is it's top
Stack = S,

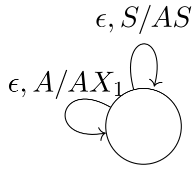
Step 2:
We used the rule: $S \rightarrow AS$ here and added the edge $\epsilon, S/AS$ into our single node as we can replace S to AS anywhere if we want to as it is defined in our CFG's production rules. Also now we pop S from our stack and push S,A (here A and S are from Γ).
Stack = S,A,



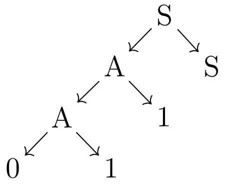


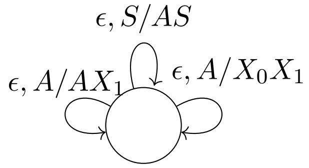
Step 3:
We used the rule: $A \rightarrow A1$ here and added the edge $\epsilon, A/AX_1$ (here X_1 is just the stack symbol corresponding to the terminal '1') into our single node as we can replace A to A1 anywhere while forming a string. Also now we pop A from our stack and push X_1,A (here A and X_1 are from Γ).
Stack = S,X_1**,A,**



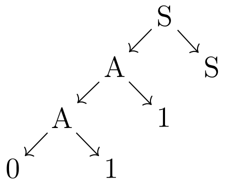


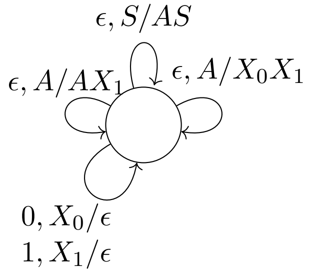
Step 4:
We used the rule: $A \rightarrow 01$ here and added the edge $\epsilon, A/X_0X_1$ (here X_1,X_0 are just the stack symbol corresponding to terminals 1 and 0) into our single node. Also now we pop A from our stack and push X_1,X_0 (here X_0 and X_1 are from Γ).
Stack = S,X_1**,**X_1**,**X_0**,**



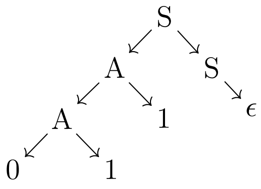


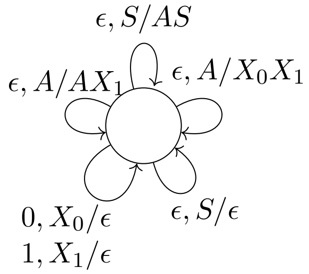
Step 5:
Now as we know that when terminals come on top of the stack and we read it from the top of the stack and pop it off and if a non terminal is seen then we apply any of the given production rules. So for this pop operation we now add the edges $0, X_0/\epsilon$ and $1, X_1/\epsilon$ into our single node. So we pop all the terminals until we see a non-terminal.
Stack = S,





Step 6:
We now use the rule: $S \rightarrow \epsilon$ here and added the edge $\epsilon, S/\epsilon$ into our single node. Also now we pop S from our stack and push ϵ that is stack becomes empty again.
Stack =





Now this stack helps in doing the DFS of the derivation tree, we first visit all the nodes on left and then all the nodes on right.
So to conclude we can say that given any arbitrary Context-free grammar we can easily construct PDA out of it by using some simple steps as described above which recognizes the exact same language as the given CFG.

8.3 Cleaning Context-free Grammar

Consider the Context-free grammar defined as follows

$$\begin{array}{lcl} S & \rightarrow & ABS \mid 0A1 \\ A & \rightarrow & 1A0 \mid D \mid 01 \mid \epsilon \\ B & \rightarrow & 1B \mid BB0 \\ C & \rightarrow & A0 \mid 01 \\ D & \rightarrow & S \mid 0AD \end{array}$$

Where,
Starting Symbol: S
Non-terminals: $\{S, A, B, C, D\}$
Terminals: $\{0, 1, \epsilon\}$

8.3.1 Eliminating Useless Symbols

We can eliminate the production rules for the non-terminal symbol "C" from our context-free grammar (CFG). Here's the reasoning:

- The starting symbol "S" can generate strings using symbols from "A," "B," and itself.
- While "B" depends on itself and "A" depends on itself and "D," we ultimately find that "D" relies on "A," "S," and itself.
- Therefore, knowing the strings accepted by "C" is unnecessary for generating strings from "S." In other words, even without the production rules for "C," "S" can still generate valid strings by appropriately utilizing "A," "B," and "D."

We can further simplify the grammar by identifying and removing another redundant non-terminal symbol: "B". Analyzing the production rules for "B", we observe two key characteristics:

- **Absence of Terminal Productions:** None of "B"'s production rules generate strings consisting solely of terminal symbols.
- **No Introduction of Useful Non-Terminals:** Additionally, "B"'s rules do not introduce any other non-terminal symbols that could potentially lead to terminal strings through subsequent productions in the parse tree.
- So the language generated by the non-terminal symbol "B" is empty (as it cannot produce any finite length string by terminating at some point), we can safely remove all production rules that involve "B" from the context-free grammar.

Now that we've removed the unnecessary rules, here's the updated CFG:

$$\begin{array}{lcl} S & \rightarrow & 0A1 \\ A & \rightarrow & 1A0 \mid D \mid 01 \mid \epsilon \\ D & \rightarrow & S \mid 0AD \end{array}$$

8.3.2 Eliminating ϵ-Productions

For removing epsilon just see where you can apply the rule $A \rightarrow \epsilon$.
We can add one more rule in all that places where A appears by replacing it with ϵ.
So after doing these changes we won't need the production rule $A \rightarrow \epsilon$ in our CFG.
Updated CFG:

$$\begin{array}{lcl} S & \rightarrow & 0A1 \mid 01 \\ A & \rightarrow & 1A0 \mid D \mid 01 \mid 10 \\ D & \rightarrow & S \mid 0AD \mid 0D \end{array}$$

8.3.3 Eliminating Unit Productions

Similar to the above property we can do the same thing here also.
Let's say we have a production rule like this $X \rightarrow Y$ where X, Y are non-terminals then we can add one more rule in all that places where "X" appears by replacing it with "Y".

Let's first remove the rule $D \rightarrow S$ by doing the changes described above,

$$\begin{array}{lcl} S & \rightarrow & 0A1 \mid 01 \\ A & \rightarrow & 1A0 \mid D \mid S \mid 01 \mid 10 \\ D & \rightarrow & 0AD \mid 0AS \mid 0D \mid 0S \end{array}$$

As we can see that after doing this modification to our CFG we now have two more production rules $A \rightarrow S$ and $A \rightarrow D$ so let's remove them also,

$$\begin{array}{lcl} S & \rightarrow & 0A1 \mid 0D1 \mid 0S1 \mid 01 \\ A & \rightarrow & 1A0 \mid 1D0 \mid 1S0 \mid 01 \mid 10 \\ D & \rightarrow & 0AD \mid 0AS \mid 0DD \mid 0DS \mid 0SD \mid 0SS \mid 0D \mid 0S \end{array}$$

After doing this we now have at least one non-terminal symbol on the right hand side of any rule or only terminals.

8.3.4 Reducing further to get at least two non-terminals on the right hand side or a single terminal

Just replace the terminals 0 and 1 with their corresponding symbols "X_0" and "X_1" in the CFG and add corresponding new rules.

$$\begin{array}{lcl} S & \rightarrow & X_0AX_1 \mid X_0DX_1 \mid X_0SX_1 \mid X_0X_1 \\ A & \rightarrow & X_1AX_0 \mid X_1DX_0 \mid X_1SX_0 \mid X_0X_1 \mid X_1X_0 \\ D & \rightarrow & X_0AD \mid X_0AS \mid X_0DD \mid X_0DS \mid X_0SD \mid X_0SS \mid X_0D \mid X_0S \\ X_0 & \rightarrow & 0 \\ X_1 & \rightarrow & 1 \end{array}$$

8.3.5 Reducing further to get exactly two non-terminals on the RHS of any production rule which has >2 non-terminals on it's RHS

Let us say our production rule has 3 non-terminals on it's right hand side then we can reduce it to exactly 2 non-teminals as follows,

$$S \rightarrow X_0AX_1 \text{ can be reduced to } S \rightarrow X_0S_1 \text{ and } S_1 \rightarrow AX_1$$

Note that the CFG after doing this modification will accept exactly same strings as accepted by it initially because now in the parse tree $S \rightarrow X_0AX_1$ will just take one step more, it first expands to $S \rightarrow X_0S_1$ and then to $S_1 \rightarrow AX_1$. However, the final string generated remains unchanged.
So it turns out that for any number of non-terminal symbols we can do this inductively to get exactly 2 non-terminals on RHS of any production rule.
Updated CFG:

$$\begin{array}{lcl} S & \rightarrow & X_0S_1 \mid X_0X_1 \\ S_1 & \rightarrow & AX_1 \mid DX_1 \mid SX_1 \\ A & \rightarrow & X_1A_1 \mid X_0X_1 \mid X_1X_0 \\ A_1 & \rightarrow & AX_0 \mid DX_0 \mid SX_0 \\ D & \rightarrow & X_0D_1 \mid X_0D \mid X_0S \\ D_1 & \rightarrow & AD \mid AS \mid DD \mid DS \mid SD \mid SS \\ X_0 & \rightarrow & 0 \\ X_1 & \rightarrow & 1 \end{array}$$

The CFG written above is said to be in it's Chomsky normal form (defined below).

8.3.6 Chomsky Normal form (CNF)

A Context-free grammar is said to be in it's Chomsky normal form if it does not contain any useless symbols, no epsilon productions ($X \rightarrow \epsilon$), no unit productions ($X \rightarrow Y$) and all production rules are either of the form $A \rightarrow BC$, where A, B, C are non-terminals or of the form $D \rightarrow u$, where D is a non-terminal and u is a terminal.

Every Context-free grammar can be expressed in it's Chomsky normal form by application of various rules as described in section 2 which accepts the same set of strings which is accepted by the original CFG.

8.4 Pumping Lemma for Context-free Languages

Let G be the Chomsky normal form of any Context-free grammar having n non-terminals. Consider derivation tree of a word "s" having length greater then 2^n as shown in the figure (8.1) below.

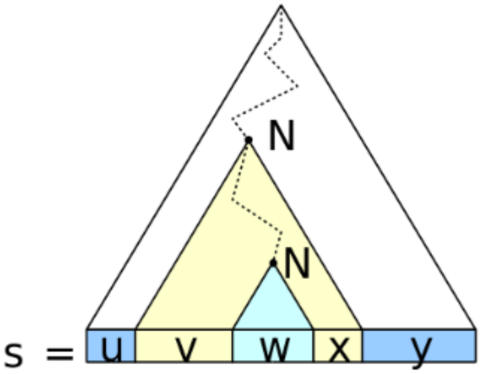


Figure 8.1: Derivation tree of a word of length $> 2^n$

Now due to the restricted nature of the CNF form of CFG (can contain maximum 2 symbols in it's RHS) the derivation tree will be a binary tree.
Now a binary tree of height n can have maximum of 2^n leaves but here we are taking $|s| > 2^n$ so the height of derivation tree must be $> n$. But we are having only n non-terminals in our CFG so by pigeon hole principle there must exist some non-terminal "N" which repeats atleast twice.
We split up s into 5 parts as shown in the figure (8.1),

$$s = u.v.w.x.y$$

- Using the property of derivation trees (of CFG) we can say that whatever I can generate from first "N" can also be generated by the second "N" and vice-versa.
- So after encountering a "N" in our derivation tree we can choose to generate a new "N" or terminate after some productions by choosing the "N" which doesn't produce another "N".
- So, to formally state pumping lemma for CFL we need to have some constraints like the sub-derivation tree of "N" should not have repetition of any non-terminal that is $|vwx| \leq 2^n$.
- Also as "N" is a non-terminal therefore $|w| \geq 1$.

8.4.1 Lemma:

If a language L is context-free, then there exists some integer $n \geq 1$ (number of non-terminals in G) such that every string s in L that has a length of 2^n or more symbols (i.e. with $|s| \geq 2^n$) can be written as

$$s = uvwxy,$$

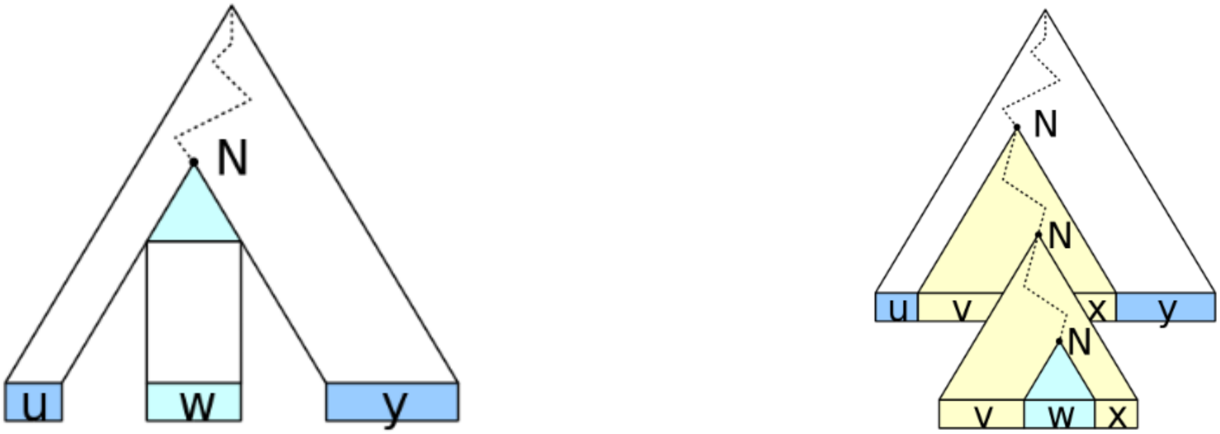


Figure 8.2: Examples of some possible derivation trees from figure (8.1)

with sub strings u, v, w, x, and y, such that:

1. $|w| \geq 1$
2. $|vwx| \leq 2^n$, and
3. $u.v^i.w.x^i.y \in L$ for all $i \geq 0$.

Example: Consider a language $L = \{0^n 1^n 2^n | n \geq 0\}$ and $\Sigma = \{0, 1, 2\}$. Now we need to prove that L can not be a CFL.

Suppose it was a CFL having n distinct non-terminals, we define $k = 2^n$ and take a very large string $w = 0^{2k} 1^{2k} 2^{2k} \in L$.
Now as $|vwx| \leq 2^n$ or $|vwx| \leq k$ so v.w.x can either lie completely in 0^{2k}, 1^{2k} or 2^{2k} or in $0^{2k}1^{2k}$ or $1^{2k}2^{2k}$.

In any case we can can pump the string to get a word which is not in L (any case will lead to increase some but not all alphabets in that word so it won't be in L) but it should be in L, therefore L is not a CFL.

8.5 Closure Properties of CFL

In this section we will look at closure properties of Context Free Languages.

8.5.1 Union & Concatenation

Context Free Languages are closed under Union.

Let L_1, L_2 be two Context Free Languages.
Their Context Free Grammars will be like,

$S_1 = ...$, and $S_2 = ...$

To represent $L_1 \cup L_2$, we can just represent its grammar as $S = S_1|S_2$.
Hence, $L_1 \cup L_2$ will be Context Free Language.

Also to represent $L_1.L_2$, we can just represent its grammar as $S = S_1S_2$.
Hence, $L_1.L_2$ will be Context Free Language.

8.5.2 Intersection

Context Free Languages are not closed under intersection.

Lets see this with an example,
Consider languages $L_1 = \{0^n1^n2^k|n, k \geq 0\}$ and $L_2 = \{0^k1^n2^n|n, k \geq 0\}$.

Both L_1 and L_2 are Context Free Languages.
$L_1 \cap L_2 = \{0^n1^n2^n|n \geq 0\}$, which we saw above is not a Context Free Language.
Hence, Context Free Languages are not closed under intersection.

8.5.3 Complement

Suppose CFLs were closed under complementation. Then for any two CFLs L_1, L_2, we have $\overline{L_1}$ and $\overline{L_2}$ are CFLs. Then, since CFLs are closed under union, $\overline{L_1} \cup \overline{L_2}$ is CFL.
Then, again by hypothesis, $\overline{\overline{L_1} \cup \overline{L_2}}$ is CFL. i.e., $L_1 \cap L_2$ is a CFL i.e., CFLs are closed under intersection. which is Contradiction! Thus CFLs are not closed under Complement.
Another example let L = {x | x not of the form ww} is a CFL.
But $\overline{L} = \{ww|w \in \{a, b\}\}$ which is not a CFL. Thus CFLs are not closed under complement.

8.5.4 Substitution

Context Free Languages are closed under Substitution, this means that replacing any symbol with a Language and the result will still be a CFL.
Given grammar G: $S \to 0S0 \mid 1S1 \mid \varepsilon$ Substituting h: $0 \to aba$ and $1 \to bb$
Rules of G' such that $L(G') = L(h(L(G)))$:

$$S \to X0SX0 \mid X1SX1 \mid \varepsilon$$

$$X0 \to aba$$

$$X1 \to bb$$

Thus, CFLs are closed under substitution.

Chapter 9

Turing Machine

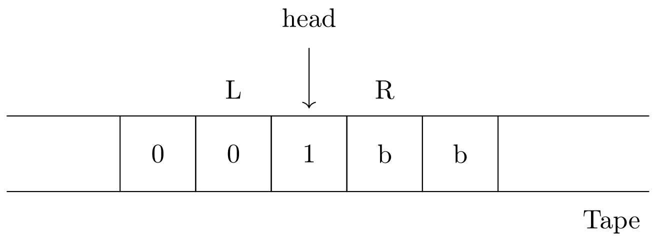


Figure 9.1: Turing Machine

Turing machines are a fundamental model of computation that can simulate the logic of any computer algorithm, regardless of complexity. They are composed of:

1. **Tape:** The machine has an infinitely long tape divided into cells, each of which can contain a symbol from a finite alphabet.

2. **Sybmols:** The tape cells contain symbols from a set. b is used to represent ’black’ cell (kind of like NULL).

3. **Head:** The machine has a head that can read and write symbols on the tape and move the tape left or right one cell at a time.

4. **Transition function:** The machine uses a transition function that dictates the machine’s actions based on its current state and the symbol it reads on the tape. Actions include writing a symbol, moving the tape left or right, and changing the state.

   - $0/1, L$: Read 0, replace with 1, Move to the left.
   - $0/1, S$: Read 0, replace with 1, stay there.
   - $0/1, R$: Read 0, replace with 1, Move to the right.
   - $-/0, S$: no matter what’s present at head, replace with 0, stay there.
     This action is similar to pushing into a stack.
   - $0/b, L$: here b is the blank symbol kept in every cell initially(means the tape cell is free).
     This action is similar to poping from a stack.

We can minic a stack by the following interconversion of moves:

1. Pushing into the stack is equivalent to moving to the right and writing the symbols to be pushed on the tape.
2. Popping off the stack is equivalent to moving to the left and writing black symbols to the tape.

The problems which can be solved using a deterministic turing machine along with a polynomial number of cells in the tape compared to the length of the input are in P and those which can be solved similarly but using non-deterministic turing machine are in NP.

9.1 Configuration of the Turing Machine

At any point when the Turing machine is running, we need to know which cell is the tape head and which state the Turing machine is in. If the finite string we gave as **input** is $X_1X_2.....X_n$ and the **tape head** at an instant is X_i, and the turning machine is in **state 'q'**. Then, we **split the input string into two parts**, the left is from the first element of the string till the element before the tape head and the right part is from the tape head to the last element.
The **configuration** is represented as $\mathbf{X_1X_2...X_{i-1}qX_iX_{i+1}...X_n}$.

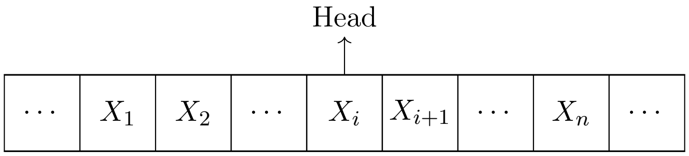


Figure 9.2: Turing Machine Tape

Let us understand more through an example:
Suppose we have been given the string **011010$\not b$0** (where $\not b$ is blank), the tape head at an instant is 1 at the third position, and the turning machine is in state q_5. Therefore, the configuration is **01q$_5$1010$\not b$0**.

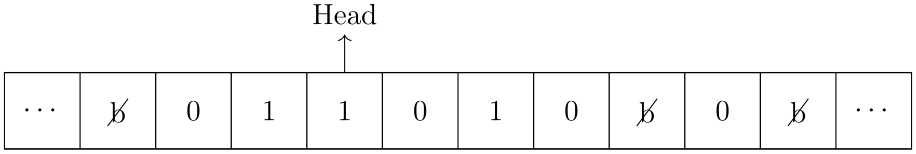


Figure 9.3: Example1 - Turing Machine Tape

Note: Here the '$\not b$' before the first 0 and the '$\not b$' after the last zero are to represent the blanks initialized in the tape and aren't part of the input string.

Now let us see what happens:

- The Turing Machine **scans cell '1'** (the tape head)
- The machine replaces the cell's content with 0 according to the transition **1/0, L**

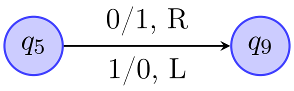


Figure 9.4: Example - Turing Machine State Transition

- The machine transitions to state $\mathbf{q_9}$
- The tape head is **shifted to the left**
- Therefore, our updated configuration is: $\mathbf{0q_9 1010\not{b}0}$.

For simplicity, let us call the left part of the input string α and the right part β and say the state is q_i, so we can represent the configuration as $\boldsymbol{\alpha}\mathbf{q_i}\boldsymbol{\beta}$

If the configuration changes from $\boldsymbol{\alpha_0}\mathbf{q_0}\boldsymbol{\beta_0}$ to $\boldsymbol{\alpha_1}\mathbf{q_1}\boldsymbol{\beta_1}$ and then to $\boldsymbol{\alpha_2}\mathbf{q_2}\boldsymbol{\beta_2}$.
This is depicted as

$$\boldsymbol{\alpha_0}\mathbf{q_0}\boldsymbol{\beta_0} \vdash \boldsymbol{\alpha_1}\mathbf{q_1}\boldsymbol{\beta_1} \vdash \boldsymbol{\alpha_2}\mathbf{q_2}\boldsymbol{\beta_2}$$

If $\boldsymbol{\alpha_0}\,\mathbf{q_0}\,\boldsymbol{\beta_0} \vdash^{*} \boldsymbol{\alpha_i}\,\mathbf{done}\,\boldsymbol{\beta_i}$, we have reached the end of the computation.
Note: '*' on $\vdash$ indicates multiple steps.

`Let us look at another example to understand halting:`

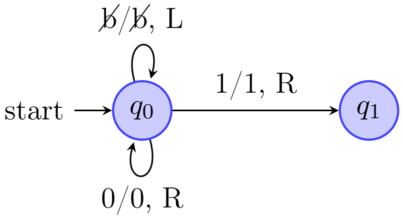
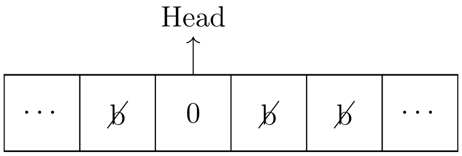


Figure 9.5: Example 2 - Turing Machine State Transition Diagram

Let us look at the computation of the string **0** where the initial configuration is $\not{b}\,\mathbf{q_0}\,\mathbf{0}$.



···	$\not{b}$	0	$\not{b}$	$\not{b}$	···

Figure 9.6: Example 2 - Turing Machine Tape (1)

Now the Turing machine scans the cell '0', replaces it with '0', moves right (to a blank cell), and stays in the same state q_0. So now the configuration becomes $\mathbf{0}\,\mathbf{q_0}\,\not{b}$.

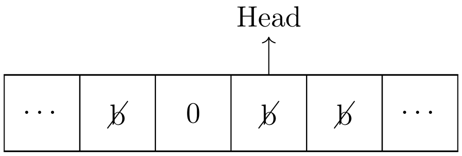


Figure 9.7: Example 2 - Turing Machine Tape (2)

This time, the Turing machine scans the cell ‘$\not b$’, replaces it with ‘$\not b$’, moves left (to ‘0’), and stays in the same state q_0, because of which the configuration changes to $\boldsymbol{\not b}\,\mathbf{q_0}\,\mathbf{0}$.

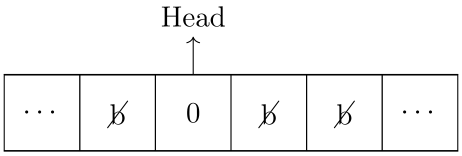


Figure 9.8: Example 2 - Turing Machine Tape (3)

As we can see, the **machine does not halt** on this input string.

Now, consider the input string **1** where the initial configuration is $\boldsymbol{\not b}\,\mathbf{q_0}\,\mathbf{1}$.

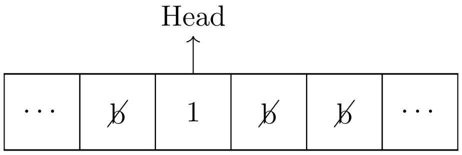


Figure 9.9: Example 3 - Turing Machine Tape (1)

The Turing machine scans the cell ‘1’, replaces it with ‘1’ moves right, and transitions to state q_1. Hence the configuration switches to $\mathbf{1}\,\mathbf{q_1}\,\boldsymbol{\not b}$.

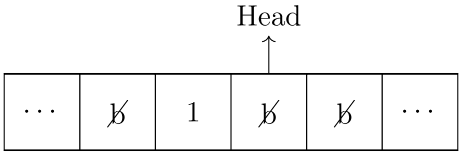


Figure 9.10: Example 3 - Turing Machine Tape (2)

As we know from the transition diagram of the Turing machine, we can't go to any other state of q_1. So the input string **1** halts on this machine whereas **0** doesn't.

We can notice that certain strings halt when processed by a Turing Machine, while others keep don't. This concept helps us define what it means for a Turing Machine to "accept" a string.
In essence, **Turing machines provide us with a way to describe different languages**.

9.2 Language described by a TM

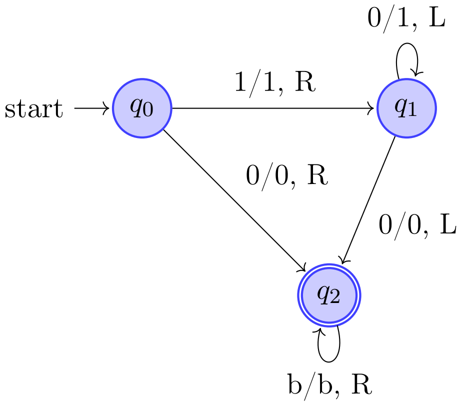
- As described above, we can use a TM to describe a set of strings based on termination. This will form the language of a TM. [1]
- We can also talk about language a TM accepts using the notion of accepting and non-accepting states. In this case, we will accept a string if the TM ever enters an accepting state during its operation on the string as input.
- However, we will primarily be interested in acceptance by halting.

9.2.1 Inter conversion between acceptance by final state and by halting:

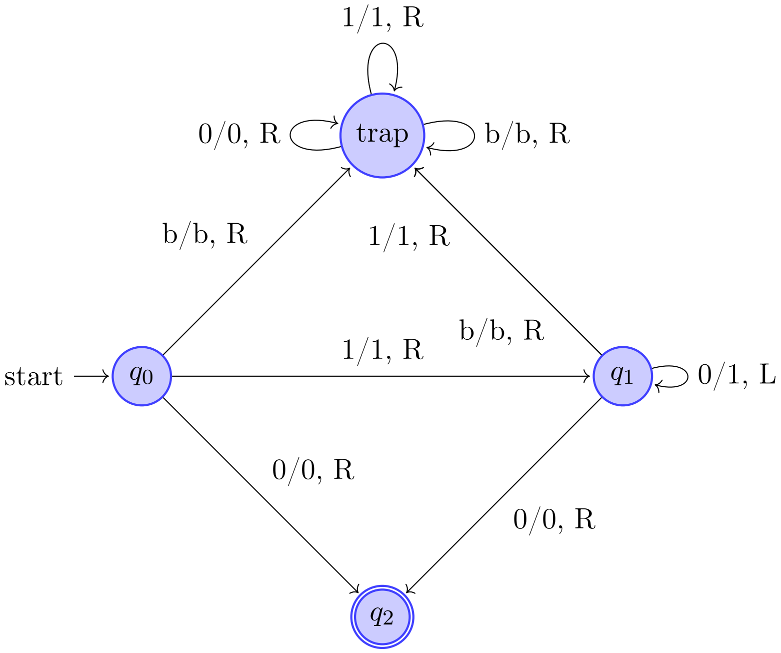
To summarize, the following are the two notions of acceptance of a string by a Turing Machine(in both cases, the input is the standard string input described above):

- **Acceptance by Halting:** All those strings are accepted for whom the Turing Machine halts its operation in finite time. We will primarily be interested in this kind of acceptance.
- **Acceptance of Final State:** All those strings are accepted for whom during the operation of the Turing Machine, a final state is reached at some point in the journey.

It is not very difficult to convert the notion of acceptance by final state to that by halting. Once you reach a final state, move on to a state whose only job is to finish parsing (similar to emptying a stack with a PDA). Moreover, we must also ensure that the TM does not halt in a non-accepting state. First, let us consider when a TM would halt on a non-accepting state:- when the TM has no outgoing transition for a particular action/input:



[1] As a side note, consider the string **0 1 0**. The behavior of termination of TM described above using this string as input depends on where you start the tail from. So, we will adopt a standard to describe the language described by a TM by assuming that the string we have taken as input lies on the tail and head is at the leftmost character of the string. If the TM halts taking this configuration as the starting configuration, then we say that the string lies in the language of the TM.



So, the process for conversion seems straightforward now: just remove the transitions going out of the final state (in this case, we removed the loop of the final state), and if some string halts on reaching some other state, make the state non-halting using trap state.

A slight note: The typical notion for acceptance by final state involves accepting the string only if after reading the entire string, you reach an accepting state. However, we have a minor modification that if you reach a final state at some point in the journey, read the rest of the string and stay in the same state. If this modification had been considered in previous things we studied, such as DFA, then it would have created a problem since that would mean all strings with their prefix as the smallest string accepted by the DFA would be accepted (there can be other strings also, but at least these would be accepted). In the case of a TM, this does not cause a problem due to 2 reasons:

- A TM can move left as well as right on the tape.
- A TM not only parses the string but also overwrites it.

So, in the case of a TM, it is not necessary that strings formed by concatenating something with an accepting prefix will be accepted.

9.3 Languages accepted by a TM

Previous lecture we saw that a Turing Machine (TM) can accept a string in two ways :- 1) Acceptance by reaching a Final State, or 2) Acceptance by Halting, out of which the latter is more commonly used. We also established that these two methods are equivalent and one can be inter-converted to other. Let's formally define these two methods :-

- $L(M) \rightarrow$ Set of all strings ω such that M enters a Final State while processing ω
- $H(M) \rightarrow$ Set of all strings ω such that M halts while processing ω

9.4 Multiple Tapes in a TM

Previously we just have seen simple TMs with a single tape. But if we have multiple tapes for a single TM, does that increase the computational power for that TM ?

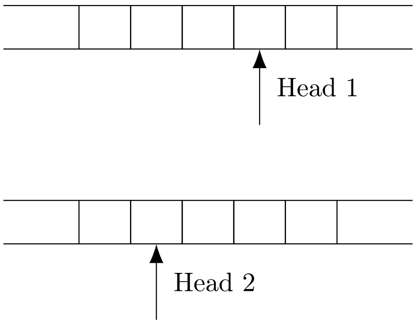


Figure 9.11: A TM with 2 Tapes

Note that we can move heads of individual tapes (here Head 1 and Head 2) independently.
Back to our question, actually as long as there are **finite** number of tapes computational power doesn't increase, i.e. we can construct another TM that functions the same as this TM but with a single tap system.

9.4.1 Converting a Multiple Tape TM to a Single Tape TM

Simultaneous Transition in Multiple Tapes

Let's take a 2 tape TM (call it M_1) example given below (Figure 9.12). Head 1 is the head for Tape T_1 and Head 2 is for Tape T_2. Set of alphabets of T_1 and T_2 is $\{0, 1, b\}$. b is the blank symbol.

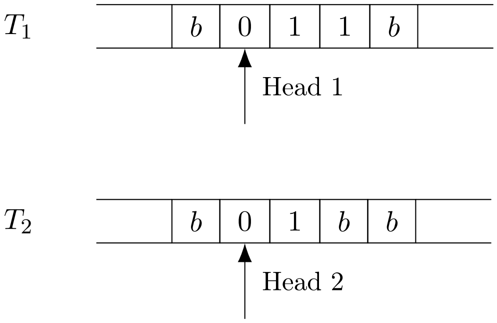


Figure 9.12: A TM with 2 tapes

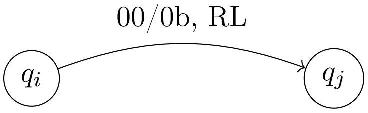


Figure 9.13: A Single Tape TM Transition

The Single Tape TM (call it M_2) will have transitions like as shown in Figure 9.13. The transition will look like of the form :-

$$\alpha\beta/\gamma\delta,\ PQ$$

Where

- α is the symbol read at Head 1 and β is for Head 2

- γ is the symbol to be overwritten at Head 1 and β is for Head 2
- P is the movement of head in T_1 (i.e. Right or Left) and Q is for T_2

After the transition in Figure 9.13, M_1 goes from Figure 9.12 to 9.15. The changes that happen in each individual tape is shown at Figure 9.14.

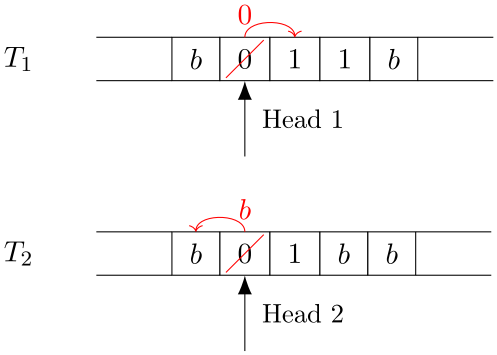


Figure 9.14: Changes

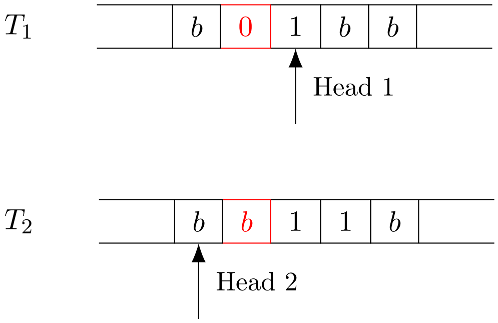
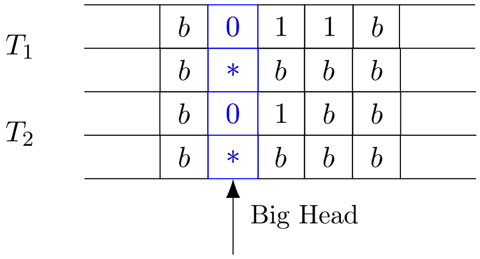


Figure 9.15: After transition

Structure of the Single Tape



T_1		b	0	1	1	b	
		b	$*$	b	b	b	
T_2		b	0	1	b	b	
		b	$*$	b	b	b	

Figure 9.16: A Single Big Fat Tape

This single tape (call it T) in Figure 9.16 consists of two rows for each of the individual tapes (in Fig. 9.12):-

- The first row consists of the elements (alphabets) of that tape (e.g. T_1) in the same order as present in T_1
- The second row consists of two types of characters (b or $*$). b is the usual blank symbol, but $*$ is a special symbol which is present where currently the head (Head 1) is pointing at T_1

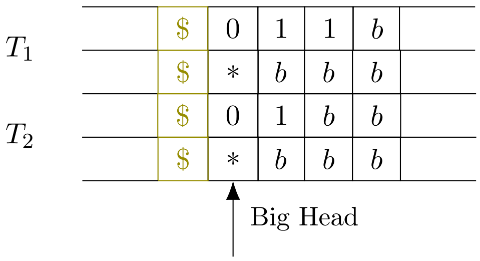
Figure 9.16 is the single tape representation of Fig. 9.12. In Fig. 9.12, Head 1 in T_1 is pointing at 0 hence the $*$ is placed just below 0 in T. Same happens for T_2.
Here an entire column serves as a single alphabet for this Big Tape. This column is the tuple of alphabets in that column. In Fig. 9.16, the head of T (calling it "Big Head") is pointing at $(0, *, 0, *)$. That column is marked in blue.

Construction of this Tape

We align the left ends of each individual tape (here T_1 and T_2) and fill each element from individual tapes one-by-one, all aligned row to row. We then mark the Left End of our Big Tape and mark the rest of the tape as blank (b).

Left End of the Single Tape

Every TM Tape must have a left end as by convention, we have to put the head at the left end of that tape. The left end of this Single Tape looks like a column full of $ as shown in Fig. 9.17:-



T_1		$	0	1	1	b	
		$	$*$	b	b	b	
T_2		$	0	1	b	b	
		$	$*$	b	b	b	

Figure 9.17: Left End in Single Tape

In Fig. 9.17, the Big Head is actually placed at the position where it should be placed when this TM is started.

Finding * in the Single Tape

So we have know the position of $*$ in T at the very beginning but we don't know that in the middle of processing. For that, there is another set of transition states (Fig. 9.18) that helps in knowing when all the $*$ have been known:-

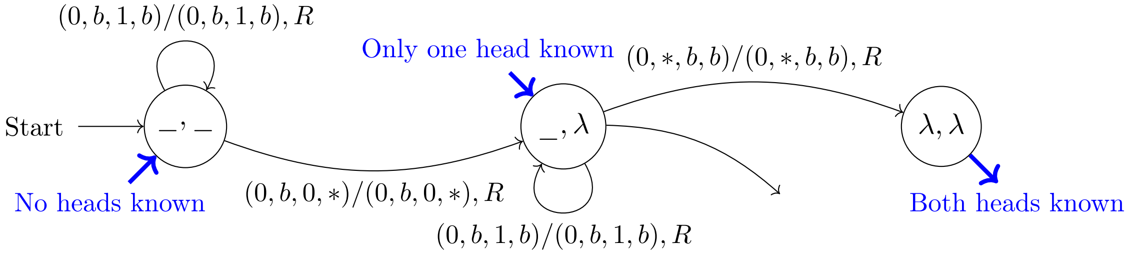

$(0, b, 1, b)/(0, b, 1, b), R$
Only one head known
$(0, *, b, b)/(0, *, b, b), R$
Start
$\_, \_$
$\_, \lambda$
λ, λ
No heads known
$(0, b, 0, *)/(0, b, 0, *), R$
Both heads known
$(0, b, 1, b)/(0, b, 1, b), R$

Figure 9.18: The $*$ finder graph

All the transitions are performed on Big Tape T and this graph always starts with **Big Head on the Left End** of T.
They are of the form "$(\alpha_1, \beta_1, \gamma_1, \delta_1)/(\alpha_2, \beta_2, \gamma_2, \delta_2)$, **R**". In this graph we just move the Big Head on the Single Tape **only Right**!
Here some of states are:-

- "_, _" state means both heads are not yet known. "_" at i^{th} position means that the i^{th} head is not known. We always start at this state.

- "$\_, \lambda$" state means exactly one head is known (and it is the second one). "λ" at i^{th} position means that the i^{th} head is known.
- "λ, λ" state means both heads are now known. We exit this graph as soon as we get to this state (here it is in case of 2 heads, for i heads the exit state is "$(\lambda, ..., \lambda)$", λ for i times. Similar logic can be applied for the appropriate Start state).

Once we exit from this "λ, λ" state, some other machine (possibly another TM) will start from the very position Big Head will be at then and then go towards the left end and do appropriate operations whenever it encounters states with even one $*$.
Note:- The number of transitions in this "$*$ finder" graph will be finite. The number of $(\alpha, \beta, \gamma, \delta)$ tuples are $|\Sigma_1| \times 2 \times |\Sigma_2| \times 2$ where Σ_i is the set of alphabets of T_i. Hence the number of "$(\alpha_1, \beta_1, \gamma_1, \delta_1)/(\alpha_2, \beta_2, \gamma_2, \delta_2)$, R" and hence total transitions are also finite.

9.4.2 Working of the Single Tape TM

Using the tools we created in the previous subsection, we can now convert the multi-tape transition diagram Fig. 9.13 into something that our Big Tape T (Fig. 9.16) can work on (see Fig. 9.19):-

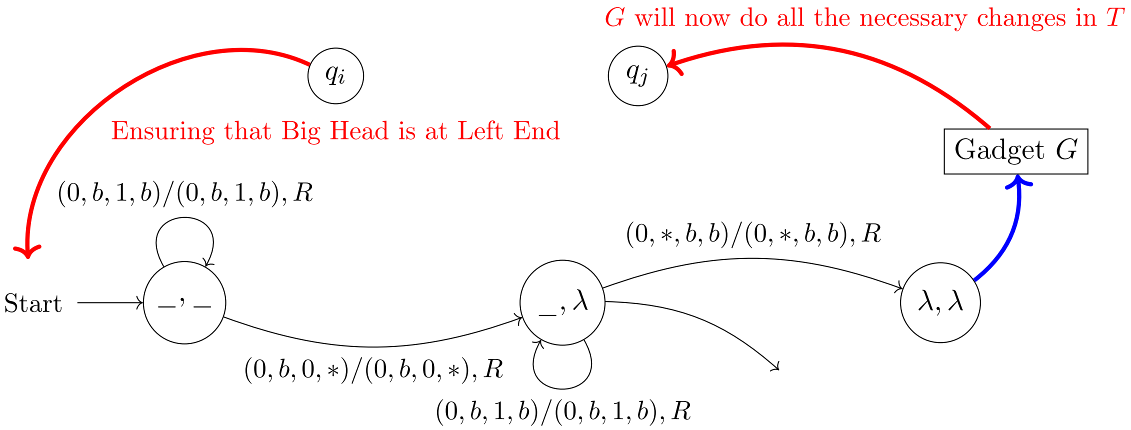

G will now do all the necessary changes in T
q_i
q_j
Ensuring that Big Head is at Left End
Gadget G
$(0, b, 1, b)/(0, b, 1, b), R$
$(0, *, b, b)/(0, *, b, b), R$
Start
$\_, \_$
$\_, \lambda$
λ, λ
$(0, b, 0, *)/(0, b, 0, *), R$
$(0, b, 1, b)/(0, b, 1, b), R$

Figure 9.19: The Single Tape transitions

So, from a configuration of Big Tape q_i, we start from the state where none the heads of individual tapes are passed by. Once we reach the final state we have passed by all the heads, we pass the control to a special gadget G. G will now traverse all the elements on the Big Tape from the current position of the Big Head till it hits the Left End ($\$, \$, \$, \$$). Whenever it encounters even one $*$, it will do appropriate operations on that column. This is equivalent to moving particular head(s) on individual tapes. And hence we have successfully reached q_j in an equivalent manner as the multiple tapes did.
Remember that the goal of the "$*$ finder" graph is not to reach till the 'right end' of the Big Tape. It just reaches till the state where it had passed by all the $*$ once and then the Gadget makes backward journey to the Start state.

What if there are multiple heads on an individual tape ?

We saw the example where each individual tape has a single head. In case there are more than one heads on single/multiple individual tape(s), then also our above method checks out.
The reason is that the states of our "$*$ finder" graph depends on the total number of heads on all individual tapes and not on the number of tapes at all (as long as it is finite).
"_" at i^{th} position (in the tuple representation of the state) means that the i^{th} head is not known. It doesn't mean that "the head of the i^{th} tape is not known". Similarly for "λ".

9.5 Non Deterministic Turing Machines-NTM

Now we move on to a different type of Turing machines, ones with non-deterministic transitions. For simplicity let us consider the NTM to have a single tape.

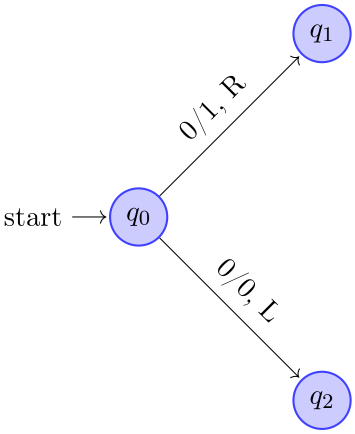


Figure 9.20: A Non Deterministic Turing Machine N

A string is accepted by an NTM if it halts for some set of choices made. The definition of acceptance is thus similar to that for an NPDA that accepts strings via halting. Now, a deterministic Turing machine (DTM) can be constructed which accepts the same set of strings as an NTM for every NTM. We first see an example for this.
Say the NTM N begins at the configuration bq_00101. Upon reading 0 there are two choices that can be made both of which are given by Figure 9.20. Now we draw a tree to represent all possible runs of this input and see if from the tree a deterministic Turing machine can be constructed.

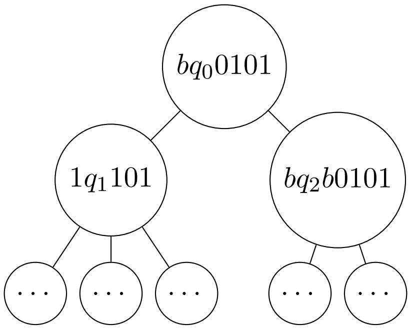


Figure 9.21: A tree showing various configurations of an NTM for a given string

We can see that if there exists a path from the root to a leaf, then the string is accepted by the NTM and the path taken constitutes a possible set of transitions that lead to halting of the TM and thus acceptance of the string. This is because a node is a leaf iff there are no transitions defined for the NTM from that configuration. We can document BFS on this tree and the moment we reach a level where a node is a leaf we can stop. This can be done by a DTM with three tapes -

1. The first tape holds the input.
2. The second tape holds the particular configuration of the NTM represented by the node.
3. The third tape holds the queue of nodes encountered during BFS.

Everytime a configuration is “popped” off the front of the queue and added to the second tape, the next possible configurations are added to the end of the queue if they exist or the TM halts if the configuration itself is a leaf in the tree. The rules leading to different configurations in the NTM can be loaded as transitions in the DTM and this can be done because the number of possible transitions in the NTM is finite. In the scheme below, α_i, β_i represent strings in $\{0, 1\}^*$.

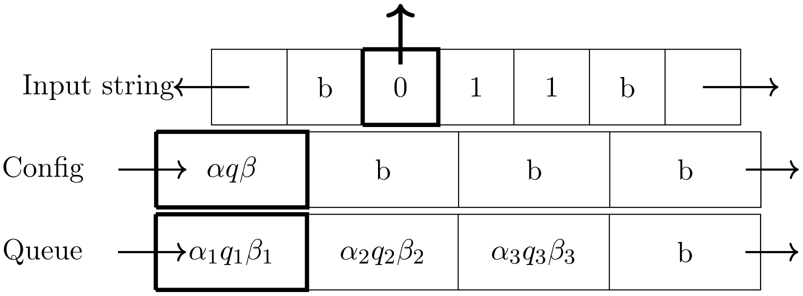


Figure 9.22: A DTM to simulate an NTM

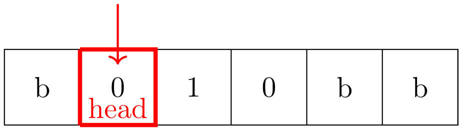
Figure 9.22 shows how an equivalent DTM constructed for any given NTM looks like. The second tape is where the next possible configurations are determinded and then pushed on the third tape which is the queue, all the while the first tape contains the string itself.

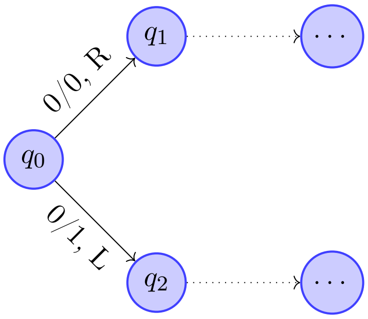
Thus an NTM has the same expressive power as a DTM unlike in the case of pushdown automata.

9.6 Turing Machine Subroutines

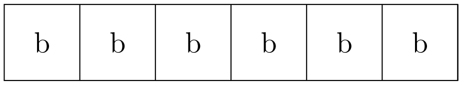
Suppose we want to develop an algorithm, perhaps a sophisticated one. A wise man would suggest we break the algorithm down into sub-tasks to eventually complete the required functionality. The same goes for the Turing Machine. We should be able to convert a single-tape Turing machine into sub-machines. Each sub-machine must perform a separate job, only to be joined together in the end.

Let's take a simple example of a non-deterministic Turing machine with the following states and transitions:

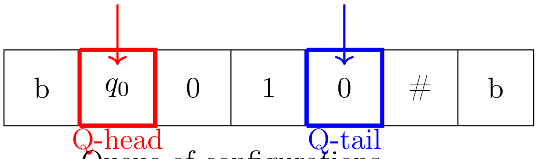




Let's divide the work into deterministic sub-machines in the following way.
Let us introduce two new tapes in addition to our input tape, one storing the current configuration of the NTM and one storing the queue of configurations of the NTM. For the above example, the initial state of the two other tapes would be:

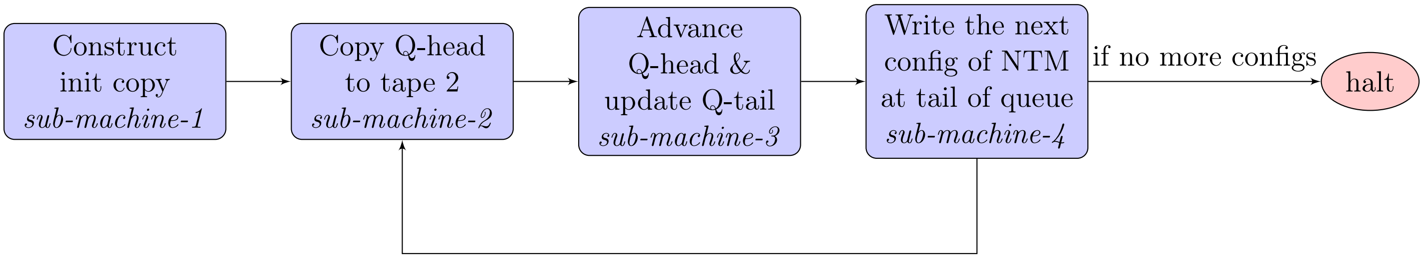


Initial Configuration



Queue of configurations

The configurations of the tape are similar to the $\alpha q \beta$ configuration of a general TM tape. The # in the queue tape allows us to separate different configurations in the queue, similar to storing multiple strings on a single tape! We break the machine into sub-machines as follows.


Construct init copy *sub-machine-1*
Copy Q-head to tape 2 *sub-machine-2*
Advance Q-head & update Q-tail *sub-machine-3*
Write the next config of NTM at tail of queue *sub-machine-4*
if no more configs
halt

Let us break this down. We are taking an NTM and changing our tapes according to the transitions.

From a particular state, there may be multiple transitions as this is an NTM. We will save all of these possible configurations into our third tape and consider all of them one by one. This is demonstrated by the flow chart demonstrated above. According to our construction, as long as there are paths to go, the DTM does not halt. If and only if there are no paths in the NTM, would our machine halt. That means this machine would accept the same words as our NTM under consideration.

We had already studied how to convert multiple-tape machines into a single-tape TM before (by separating tapes with some character # on a single tape). Thus, we can merge all of this to show the equivalence of an NTM with a DTM.

9.7 Restricted Turing Machines

The next question we ponder upon is whether we can restrict DTMs and still have the same expressive power.

9.7.1 Turing Machines with Semi-infinite Tapes

A semi-infinite tape has no cells to the left of its head position. We shall show that a semi-infinite tape can be used to simulate a Turing machine, whose tape is infinite in both directions. The trick behind the construction is to use two tracks on the semi-infinite tape. The upper track represents cells $X_0, X_1, ...$, where X_0 is the initial head position. The lower tracks represents cells $X_{-1}, X_{-2}, ...$ to the left of the initial position. We also place an endmarker $*$ on the leftmost cell of the lower track, to prevent the head of the semi-infinite tape from falling off.

X_0	X_1	X_2	$\cdots$
$*$	X_{-1}	X_{-2}	$\cdots$

Figure 9.23: Semi-infinite tape

Let the semi-infinite tape Turing Machine be represented by

$$M = (Q, \Sigma \times \{B\}, \Gamma, \delta, q_0, [B, B], F)$$

Let the original Turing Machine by represented by

$$M' = (Q', \Sigma, \Gamma', \delta', q_2, B, F')$$

Q : The states of M are $\{q_0, q_1\} \cup (Q' \times \{U, L\}$. U and L tell whether M is scanning a symbol on the upper or the lower track. In other words, U and L tell whether head of M' is at or to the right of its initial position, or to the left of its initial position.

Γ : The tape symbols of M has all the pairs of symbols from Γ', that is, $\Gamma' \times \Gamma'$. Additionally for every symbol X in Γ', there is a pair $[X, *]$ in Γ

δ : The transitions of M are as follows:

1. $\delta(q_0, [a, B]) = (q_1, [a,^{*],R)}$ This is the first move which puts $*$ in the lower track of the leftmost cell, and moves right because it cannot move left or remain stationary
2. $\delta(q_1, [X, B]) = ([q_2, U], [X, B], L)$ for any X in Γ'. This establishes the initial conditions of M' as q_2 was the initial state of M' and by moving left, we get back to the initial head.
3. If $\delta'(q, X) = (p, Y, D)$, then for every Z in Γ',
   - $\delta([q, U], [X, Z]) = ([p, U], [Y, Z], D)$ and,
   - $\delta([q, L], [Z, X]) = ([p, L], [Z, Y], \bar{D})$

   where $\bar{D}$ is the direction opposite to D
4. If $\delta'(q, X) = (p, Y, R)$, then
$$\delta([q, L], [X, *]) = \delta([q, U], [X, *]) = ([p, U], [Y, *], R)$$
5. If $\delta'(q, X) = (p, Y, L)$ then,
$$\delta([q, L], [X, *]) = \delta([q, U], [X, *]) = ([p, U], [Y, *], L)$$

F : The accepting states F are the states in $F' \times U, L$

By induction on the number of moves made by M', we can show that M will mimic the ID of M' on its own tape (basically the reverse of lower track followed by the upper track). Also M enter an accepting state exactly when M' does. Thus $L(M) = L(M')$

9.7.2 Two Stack Machine

We know what a Turing machine can accept languages not accepted by a PDA with one stack, but it turns out that a PDA with two stacks can accept any language that a Turing machine accepts. We'll prove that if a language is accepted by a Turing Machine, then it is accepted by a two stack machine.

`PROOF:` Our two stack machine S can simulate a Turing Machine M by holding what is to the left of the head in one stack and what is at and to the right of the head in the other stack, except for the infinite string of blanks beyond the leftmost and the rightmost non-blanks. Our two stack machine S will do the following:

1. Both the stacks have a bottom-of-stack marker, and are considered empty when they only contain this marker.

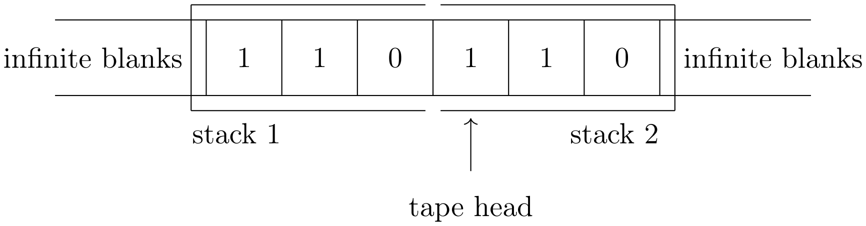


Figure 9.24: Visualizing the two stacks

2. When S enters the start state of M, the first stack is empty, since M has only blanks to the left of the cell pointed to by the tape head. The second stack holds ω, since ω appears at and to the right of the cell pointed to by the tape head.

3. S simulates a move of M as follows:

   - S knows the state M is in, in its finite control. It also knows the tape symbol X scanned by M's head. It is the top of S's second stack. If the second stack has only the bottom-of-stack marker, then S interprets the symbol scanned by M as a blank.
   - If M replaces X by a Y and moves right, then S pushes Y onto its first stack, since Y is now to the left of M's head and it pops X off the second stack. However, if the second stack is empty, then it is not changed. If the first stack is empty and Y is a blank, then it is not changed.
   - If M replaces X by a Y and moves left, then S pops the top of the first stack, say Z, then replaces X by ZY on the second stack. If the first stack is empty, then S must push BY on the second stack and not change the first one.

4. S accepts if the new state of M is accepting. Otherwise, S simulates another move of M.

9.7.3 Counter Machines

A counter machine can be regarded as a multi-stack machine with the restriction that the only allowed stack symbols are Z_0 (bottom-of-the stack marker) and X.

1. Initially Z_0 is on each stack

2. We can replace Z_0 only by a string of the form X^iZ_0 for some $i \geq 0$

3. We may replace X only by X^i for some $i \geq 0$

There's another way to interpret a counter machine:

1. Counters hold any non-negative integer.

2. In one move, counter machine can

   - Change state
   - Add or subtract 1 from any counter independently. However, a counter cannot become negative, so we cannot subtract 1 from a counter that is 0.

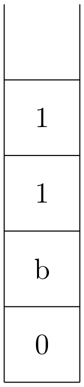


Figure 9.25: Stack from previous machine

We'll use the second interpretation.
In our earlier representation of stacks, the stacks would look something like:
We can write this as a number in base 3 (or any suitable base). So we obtain two numbers in base 3 from the two stacks. In our new counter stacks, we need to mimic stack operations using operations on numbers as follow:

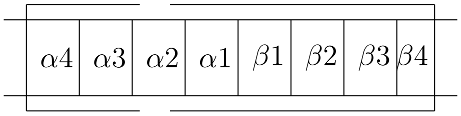
- Popping from the stack can be implemented using repeated subtraction
- Pushing onto the stack can be implemented using repeated addition

9.8 Simulators for Turing Machine

In this lecture, we will see the simulation of the turing machine as a two stack machine as well as two counter machine.

9.8.1 Turing Machine as Two Stack Machine

A single tape doubly infinite turing machine can be simulated using two stacks. To do so, we will break the turing machine into two parts and each part is represented using a stack. Suppose that the turing machine is in the state αqβ, then the basic idea of representing the turing machine using two stacks is represented in the following figure:



Let us consider the part before q i.e., α as stack 1 (s1) and the part after q i.e., β as stack 2 (s2). It is considered from the figure that the next symbol turing machine will process is β1. After processing β1, if the finite control(head) moves to the left, then the length of the string β will increase by 1 (s2.push()) and the length of α will decrease by 1 (s1.pop()), else if the finite control moves to the right, then the length of the string β will decrease by 1 (s2.pop()) and the length of α will increase by 1 (s1.push()). The simulation of the turing machine in form of two stacks can be understood easily in the following way:

1. Pop the top element in s2 –> **s2.pop()**
2. Replace the popped element $\beta 1$ by new symbol (say $\beta 1$') –> **s2.push(β1')**
3. If the head moves right, pop $\beta 1$' from s2 and push it in s1 –> **s1.push(s2.top())** and then **s2.pop()**
4. If the head moves left, pop $\alpha 1$ from s1 and push it in s2 –> **s2.push(s1.top())** and then **s1.pop()**

We will repeat the above state until we reach to an halting state and hence, we can say that the language that is accepted/computation performed by a turing machine can also be done by a specific two stack machine.

In the following section, we will see that if a language can be represented/computation performed using two stack machine, then it can also be represented using three counter (further we will reduce it to two counters). Ultimately, it means that a turing machine can be simulated using two/three counters.

9.8.2 Simulation of Two Stack Machine using Three Counters

To simulate the two stack machine using counters, we consider that the symbols in the tape can only be read (we won't write in the tape). Consider there are three counters. A counter can store any non-negative integer.

Following are the three operations that are allowed to be done on a counter:
1. Check if the counter contains zero.
2. Increment counter value by 1.
3. Decrement counter value by 1.

In the previous section, we saw that the turing machine can be simulated using two stacks. To further simulate two stacks using counters, **first counter represent stack s1, second counter represent stack s2 and the third counter is the temporary counter (auxiliary counter) which is used in doing the computations.**

Suppose there are **t** unique symbols. Let us represent these **t** symbols using 1,2,3,...,t.

Representing Stack using a Counter:- Let, there are k symbols overall (may be unique or some symbols may be same as others) in the stack s1. Then, we can represent the elements on the stack in the following manner (Only 4 elements of the stack are shown below but it contains k elements):

0	1	b	1
1	3	4	k

The bottom of the stack will be most significant and top of the stack is least significant. Above sequence is represented in base **k+1**. . For the figure above, the value inside the counter 1 will be :

$$\text{counter 1 value} = 1 * (k+1)^{k-1} + 3 * (k+1)^{k-2} + 4 * (k+1)^{k-3} + ... + k * (k+1)^0$$

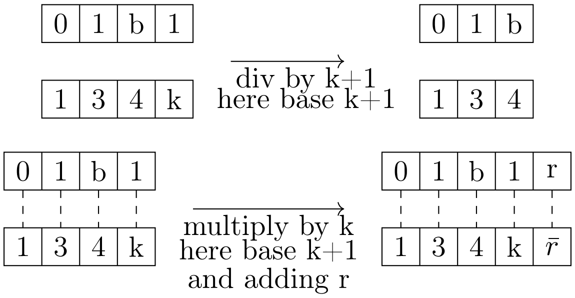
Simulating push and pop operation of the stack using counters:- Popping up the element from the top of the stack is means we are dividing the value inside the counter by **k+1**.To do so, we start decreasing the value in the counter 1 by **k+1** and increase the value in the temporary counter (counter 3 we assumed) one by one (starting from zero). We do so until we reach 0 (If initial value in stack was divisible by k+1, in the end we will found that the value inside counter 1 is zero, otherwise it will be equal to (intial value) mod (k+1). After doing so, the value that we shall found inside the counter three will be equal to (intial value)/(k+1) and we copy this value into the counter 1 for further computations.

$$\text{final value} = \text{(initial value)/(k+1)}$$

Similarly, pushing an element say **r** in stack means we are changing the value of counter in following way:

$$\text{final value} = \text{(k+1) * (initial value) + r}$$

We can do so in the following manner. For the current value in the counter 1, we will keep decreasing the value inside counter 1 by 1 until it becomes 0 and incrementing the value inside the counter 3 by (k+1). Then we just increase the value inside counter 3 by **r** step by step. Then we copy this value inside the counter 1 and find out that this value is same as written above in the expression.



Hence, in this manner we can perform the multiplication and division operation using counters.

Finding top of the stack:- Finding top of stack in two stack machine is same as finding the remainder after dividing the current value inside the counter 1 by (k+1) and this we have already discussed above in case of division operation.

9.9 Stimulating a computer using a TM

When simulating a computer with a Turing Machine (TM), we can replicate the essential components of a computer system such as registers and memory. Registers, which facilitate fast computation, can be represented by one tape, while memory, responsible for storing instructions and data, can be represented by another tape. Additionally, we can allocate a third tape to serve as the Instruction Counter, which keeps track of the current instruction being executed.
0.1 cm

By utilizing these three tapes, we can coordinate the flow of instructions and data within the TM simulation. The Instruction Counter tape guides the TM to execute the appropriate instruction from the memory tape, while the data required for computation is retrieved from the same memory tape. Computation itself is performed using the contents of the register tape.

This arrangement effectively replicates the operation of a computer system within the confines of a TM. Each tape serves a specific purpose in managing the flow of information and executing instructions, enabling the TM to emulate the behavior of a computer. Below, we provide an example illustrating how such a TM can be structured and utilized to simulate a computer system.

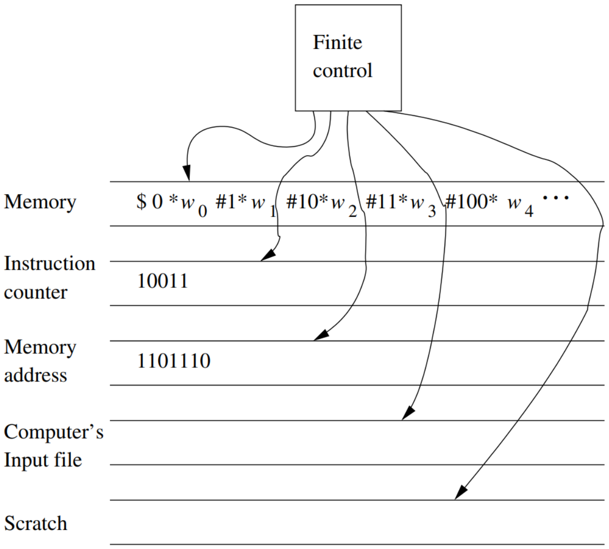


Figure 9.26: An example of stimulating a computer using a TM

9.10 Representing a Turing Machine as a Natural Number

Take a look at the following Turing machine, with the set of tape symbols $\Gamma = \{0, 1, b\}$ with their respective IDs = {1,2,3}, also {L,R} = {1,2}.

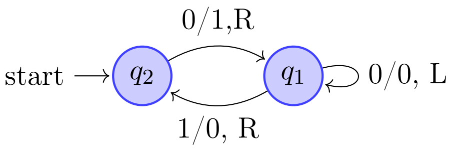


Figure 9.27: Transition diagram of a Turing machine

Now, we try to represent the no. of states, starting state, tape alphabet size, blank alphabet ID, and transition set, all using unary 1s, and 0s as separators.

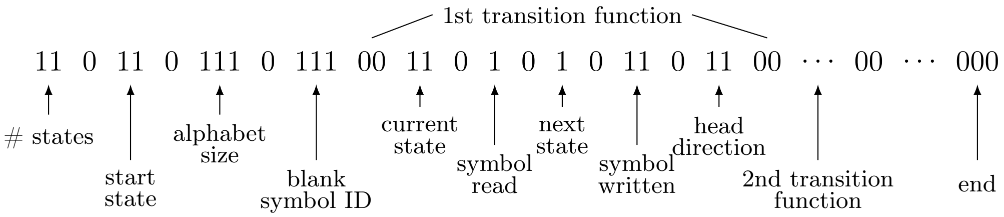


Figure 9.28: Binary representation of a Turing machine

By using the process shown in Fig 9.28, we can write any Turing machine as a binary. Therefore, Set of all Turing machines ⊆ Set of Natural Numbers. Previously, we showed that any computation that a computer does, can be reduced to a Turing machine. But, the set of languages all Turing machines can accept itself is countable! So, computers are not as powerful as you think.

Food for thought: now that we have obtained the binary representation of a Turing machine, what if we pass it as an input to another Turing machine, what if we make a Turing machine process itself?

9.10.1 Mapping Natural Numbers to Turing Machines

We can map every natural number to a Turing machine such that all Turing machines are mapped, as follows. For every natural number n,

- If there exists a Turing machine which is mapped to the bit-string corresponding to n, as per the mapping shown before, then map n to that Turing machine. (Note that since the mapping taught in the previous class was one-one, there can be atmost one Turing machine mapped to n.)
- If there does not exist such a Turing machine mapped to n, then map n to the Turing machine M (Fig. 9.29) which does not change the input on the tape.

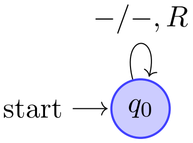


Figure 9.29: A Turing Machine M which does not halt and does nothing to the tape

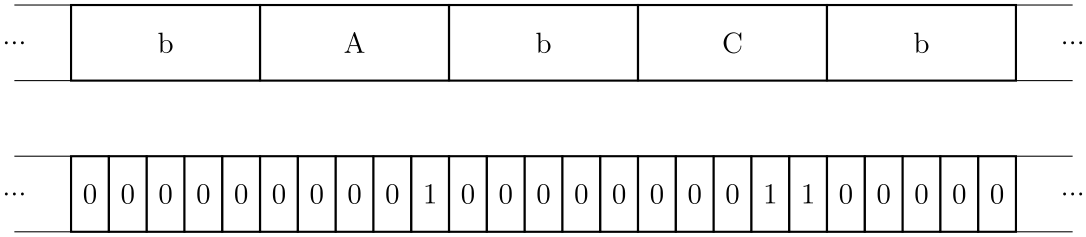
Note that this mapping from natural numbers to the set of Turing machines is not injective, that is, many natural numbers can represent the same Turing machine in this mapping.

9.10.2 Encoding Input Strings as Natural Numbers

We can map every input string to a unique natural number (bit-string) as follows. As an example, we will consider $\Gamma = \{A, B, \cdots, Z, b\}$ as the tape alphabet (including the blank symbol b).

- One way to encode strings as natural numbers is by representing each symbol in the tape alphabet Γ by a different count of the unary symbol (say 1), called the ID of the symbol. We also require a marker symbol (say 0) to separate different symbols. In this example, we can represent each tape symbol like $A = 1$, $B = 11$, $C = 111$, $b = 1^{27}$ (1 repeated 27 times), $\cdots$. Then, AbC will be encoded by $101^{27}0111$.
- Another way is using $\lceil \log_2 k \rceil$ bits to represent each symbol in the tape alphabet Γ where $k = |\Gamma|$ (which is finite). In this example, we can use $\lceil \log_2 27 \rceil = 5$ bits to represent each tape symbol like $b = 00000$, $A = 00001$, $B = 00010$, $C = 00011, \cdots$ (Fig. 9.30)

  One advantage of this encoding is that all tape symbols have encoded bit-strings of the same length, and hence we do not require markers unlike when tape symbols are encoded in unary ID's. So, we can create an Turing machine equivalent to M which reads $\lceil \log_2 k \rceil$ bits together before doing a transition equivalent to the transition taken by M.



··· b A b C b ···

··· 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 1 1 0 0 0 0 0 ···

Figure 9.30: Encoding string as a natural number (bit-string)

Therefore, the relevant part of the input tape (the part of the tape which starts with the first non-blank symbol and ends with the last non-blank symbol) can be represented by a unique natural number. Hence, we have constructed a one-one mapping from the set of input strings to the set of natural numbers.

Now, we can construct the onto mapping from the $\mathbb{N}$ to the set of input strings similar to the mapping from $\mathbb{N}$ to the set of Turing machines. For every natural number n,

- If there exists an input string w which is mapped to the bit-string n, as per the mapping given above, then map n to that input string w. (Once again, since the mapping was one-one, there can atmost be one such string w.)
- Else, map n to the blank input tape.

Note that this mapping is onto but not injective like the mapping from $\mathbb{N}$ to the set of Turing machines.

9.11 Recursively enumerable languages

Halting language: The halting language of a Turing Machine M is defined to be the set of strings for which the Turing machine M halts from the initial configuration q_0w, that is,

$$H(M) = \{w \mid q_0w \text{ configuration eventually causes } M \text{ to halt}\}$$

Recursively enumerable language: A set of strings is said to be recursively enumerable iff $\exists$ a Turing machine M such that $L = H(M)$.

For example, the empty language ϕ is recursively enumerable since we can construct a Turing machine M (Fig. 9.29) which does not halt on any string, that is, $H(M) = \phi$. In M, $-$ stands for all tape symbols, and hence M will not halt on reading any symbol.

9.11.1 Enumerating the strings of a recursively enumerable language

Given a recursively enumerable language $L \subseteq \Sigma^{*=\{0,1\}^*}$, there exists a Turing machine M with $\Gamma = \{0, 1, b\}$, such that $H(M) = L$. We will construct a Turing machine M' which will output the strings in L on an output tape T_{out} separated by a marker (say #). Take $\Gamma' = \{0, 1, b, \#\}$.

Let the Turing machine M' have 4 sub-machines M_1, M_2, M_3, M_4 and 5 tapes T_1, T_2, T_M, T_C, T_{out}. (Fig. 9.31)

- Tape T_1 contains the input string w that will be checked to determine whether it is part of L or not. This string will be incremented/decremented by M_1 at the end of each iteration, as per the order of traversal of the $w - n$ plane.
- Tape T_2 contains the maximum number of steps n upto which M_3 will run on string w, to check whether w halts in n steps or not. This number will be incremented/decremented by M_2 at the end of each iteration, as per the traversal.
- Tape T_M is used by M_3 to mimic the Turing machine M on the input string w. w is copied onto this tape at the beginning of each iteration.

- Tape T_C contains the number of steps c done by M_3 on input w until then. c is initialised to 0 at the start of each iteration, and incremented by M_4 after each step of M_3. At the beginning of each step, c is compared with n. If they are equal and M_3 has halted at this step, then w is written onto the output tape T_{out} separated by $\#$, else it is not written. Then, M' goes to the next iteration.

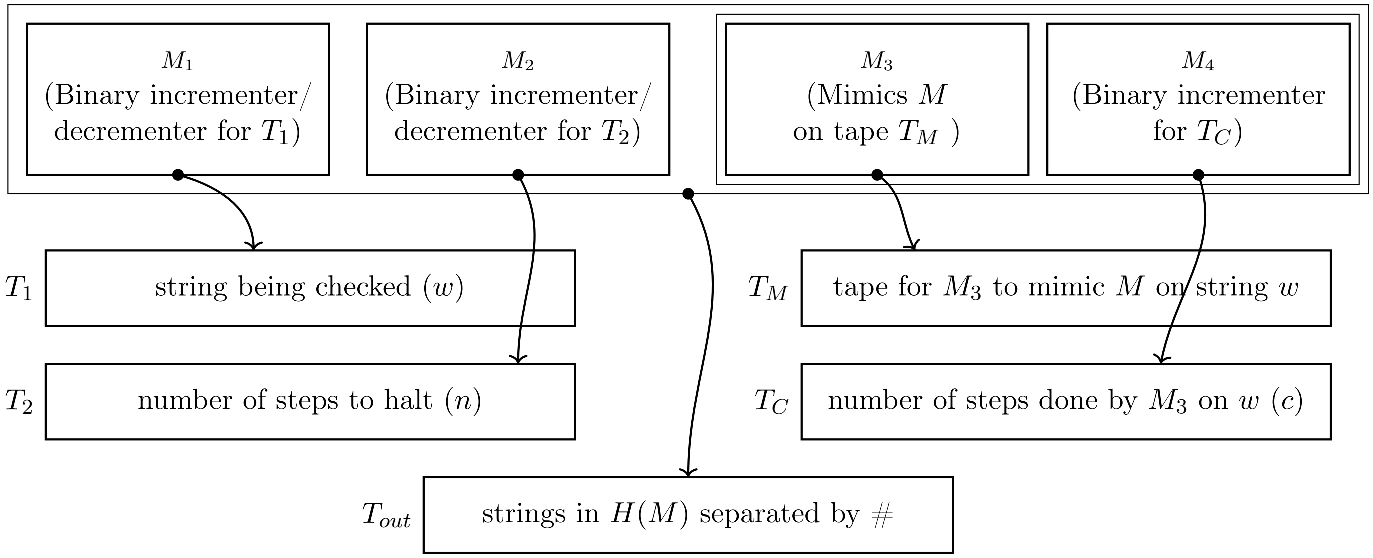

M_1
(Binary incrementer/ decrementer for T_1)
M_2
(Binary incrementer/ decrementer for T_2)
M_3
(Mimics M on tape T_M)
M_4
(Binary incrementer for T_C)
T_1
string being checked (w)
T_2
number of steps to halt (n)
T_M
tape for M_3 to mimic M on string w
T_C
number of steps done by M_3 on w (c)
T_{out}
strings in $H(M)$ separated by $\#$

Figure 9.31: Turing machine M' to enumerate strings in $H(M)$

M_1 and M_2 update w on T_1 and n on T_2 according to the diagonal traversal of all points with non-negative integer coordinates in $w - n$ plane, as shown in Fig. 9.32.

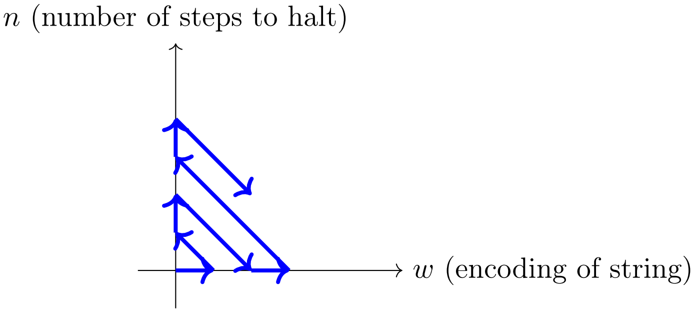


Figure 9.32: Traversing all points in the x-y plane with non-negative integer coordinates

Every string $w \in L$ will have a non-negative finite number of steps n after which M halts, since $w \in L = H(M)$. Hence, every string w in L will have a point with non-negative integer coordinates (w', n) corresponding to it, where w' is the encoding of the string and n is the number of steps for M to halt on w.

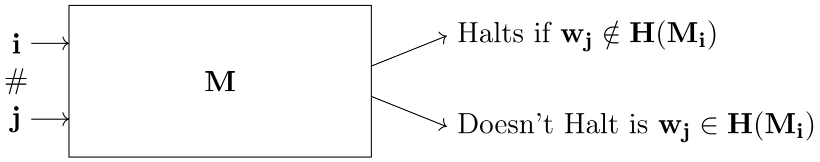
Since all the points in the $w-n$ plane with non-negative integer coordinates are traversed, all words in the language are written on the output tape T_{out} and since M does not halt for all the strings which are not in the language, they will not be written on T_{out}. Hence, M enumerates exactly all the strings in L.

9.11.2 Is $\overline{\mathbf{L_u}}$ a R.E. Language?

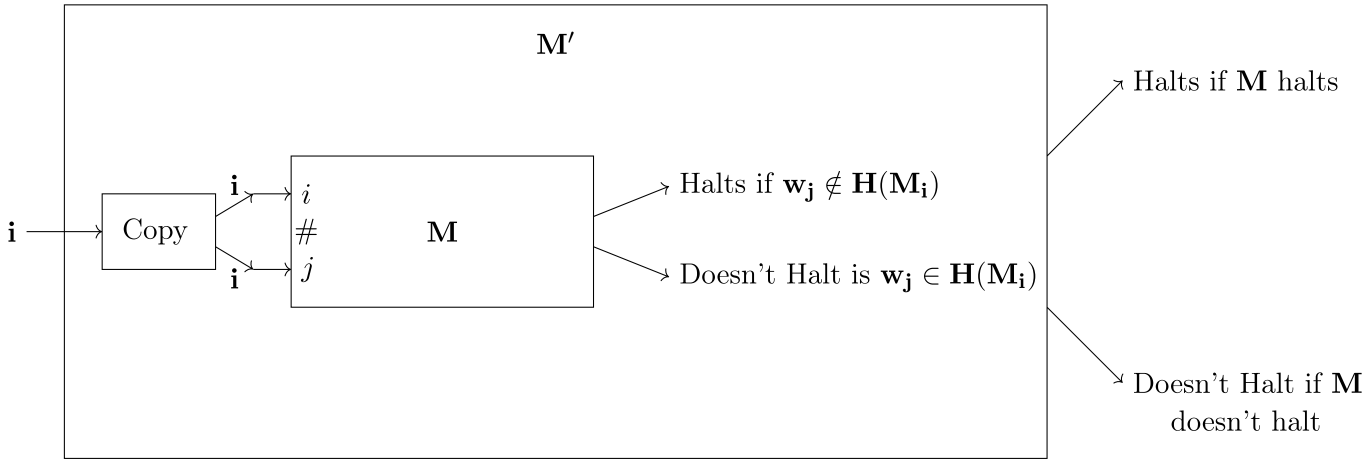
Note that, exact mathematical complement of $\mathbf{L_u}$ contains all non-halting tuples and strings which do not represent a (Machine, String) tuple. But we not interested in the invalid tuples.
Hence, we define $\overline{\mathbf{L_u}}$ not as the normal complement of $\mathbf{L_u}$ but as follows
$\overline{\mathbf{L_u}} := \{\mathbf{i\#j} \mid \mathbf{M_i}$ doesn't halt on $\mathbf{w_j}, i.e, \mathbf{w_j} \notin \mathbf{H}(\mathbf{M_i})\}$

Now we want to see whether $\overline{\mathbf{L_u}}$ is R.E. or not.
Claim: It is not a R.E. Language.
We will try to prove this claim by reducing $\overline{\mathbf{L_u}}$ to $\mathbf{L_d}$ which is not R.E.
Intuition: If there's a Turing Machine for $\overline{\mathbf{L_u}}$, we will use it to construct a Turing Machine for $\mathbf{L_d}$, but we know there does not exist a Turing Machine for $\mathbf{L_d}$, hence we can argue that there does not exist a Turing Machine for $\overline{\mathbf{L_u}}$.

Proof: Suppose there exists a Turing Machine $\mathbf{M}$ such that $\mathbf{H}(\mathbf{M}) = \overline{\mathbf{L_u}}$ and runs as follows


i
#
j
M
Halts if $\mathbf{w_j} \notin \mathbf{H}(\mathbf{M_i})$
Doesn't Halt is $\mathbf{w_j} \in \mathbf{H}(\mathbf{M_i})$

Now we construct a new bigger Turing Machine $\mathbf{M'}$ which uses $\mathbf{M}$ in it.


M′
i
Copy
i
i
i
#
j
M
Halts if $\mathbf{w_j} \notin \mathbf{H}(\mathbf{M_i})$
Doesn't Halt is $\mathbf{w_j} \in \mathbf{H}(\mathbf{M_i})$
Halts if M halts
Doesn't Halt if M doesn't halt

Note that the language $\mathbf{L_d}$ is the halting language of the above machine $\mathbf{M'}$ because:
If $\mathbf{i} \in \mathbf{L_d} \implies \mathbf{w_i} \notin \mathbf{H}(\mathbf{M_i}) \implies \mathbf{M}$ halts on $\mathbf{i\#i} \implies \mathbf{M'}$ halts on $\mathbf{i}$. ($\mathbf{M'}$ halts if $\mathbf{M}$ halts)
If $\mathbf{i} \notin \mathbf{L_d} \implies \mathbf{w_i} \in \mathbf{H}(\mathbf{M_i}) \implies \mathbf{M}$ doesn't halt $\mathbf{i\#i} \implies \mathbf{M'}$ doesn't halt $\mathbf{i}$. ($\mathbf{M'}$ doesn't halt if

M doesn't)
(**Note:** $\mathbf{j} = \mathbf{i}$ for **M** inside $\mathbf{M'}$, and so **M** Halts if $\mathbf{w_i} \notin \mathbf{H}(\mathbf{M_i})$ and vice-versa)

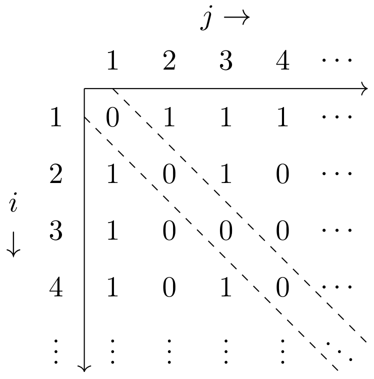
Hence, we can say $\mathbf{H}(\mathbf{M'}) = \mathbf{L_d}$. But we know, as $\mathbf{L_d}$ is not Recursively Enumerable, $\mathbf{M'}$ does not exist. Therefore, **M** does not exist, contradicting our assumption.
Hence, $\overline{\mathbf{L_u}}$ **is not a Recursively Enumerable Language**

9.11.3 Construction of a language which is not recursively enumerable

Recall that the existence of a language which is not recursively enumerable can be shown by proving that the set of Turing machines is countable and the set of languages is uncountable. Here, we give a construction for one such language.

Since we have shown that each Turing machine and each input string can be encoded by a natural number each, we can construct a well-defined table (A) in which the entry in the i-th row and j-th column (A_{ij}) is 0 if the turing machine M_i does not halt on the string w_j and 1 if it does. (Fig. 9.33)

The i-th row of the table represents the halting language of the Turing machine M_i.
$H(M_i) = \{w_j \mid M_i \text{ halts on } w_j\} = \{w_j \mid A_{ij} = 1\}$



$$j \rightarrow \qquad i \downarrow$$

$$\begin{array}{c|ccccc}
 & 1 & 2 & 3 & 4 & \cdots \\
\hline
1 & 0 & 1 & 1 & 1 & \cdots \\
2 & 1 & 0 & 1 & 0 & \cdots \\
3 & 1 & 0 & 0 & 0 & \cdots \\
4 & 1 & 0 & 1 & 0 & \cdots \\
\vdots & \vdots & \vdots & \vdots & \vdots & \ddots
\end{array}$$

Figure 9.33: Table A for acceptance of w_j by M_i

Consider the diagonal language $L^* = \{w_i \mid M_i \text{ does not halt on } w_i\} = \{w_i \mid A_{ii} = 0\}$
This is the language obtained by flipping the 0s and 1s on the diagonal of the table.

Can there exist a k such that $H(M_k) = L^*$? No. This can be proved by checking the presence of w_k in $H(M_k)$ and L.

$w_k \in H(M_k) \implies A_{kk} = 1$ (M_k halts on w_k) $\implies w_k \notin L$
$w_k \notin H(M_k) \implies A_{kk} = 0$ (M_k does not halt on w_k) $\implies w_k \in L$

Hence, for all k, w_k is part of exactly one of the languages $H(M_k)$ and L. Thus, $\forall k \in \mathbb{N},\ L^* \neq H(M_k)$. Since all Turing machines are encoded into the form M_i, we can say that L^* is not the halting language for any Turing machine, and hence L^* is not a recursively enumerable language.

9.11.4 Undecidable Languages

Consider the language $L^* = w_i \mid M_i$ doesn't halt on w_i i.e we have to deal the diagonal entries in the table, if there's an entry 0 in the diagonal i.e the string exists in L^*, i.e to get L^* just flip all the entries in the table i.e swapping 0's and 1's ,then take all the corresponding strings of entries 1 along diagonal into L^*.

Now the Question – Is L^* = H(M_k) for some M_K??
Ans:NO
It's just the contradiction principle,for suppose assume there exists k satisfying that assingment ,consider the position(k,k) if the entry in that position say 1,i.e machine M_k gets halted after finte steps for input string w_k, i.e $w_k \in$ H(M_k) but w_k doesn't belong to L^* as w_k gets halted on machine M_k in finite steps i.e $w_k \notin L^*$ — contradiction ,similarly if u say the position is 0 ,follow the same process u will land in a contradiction.

We call this type of languages as **undecidable languages**, which we will discuss detailed in the further classes.

9.11.5 Universal language

To prove that $\overline{L_d}$ is a recursively enumerable language we ended up constructing a 'Universal turing machine' which takes in a string of the form $w_i\#w_j$ and can simulate a run of the string w_j on the turing machine M_i. We can also define a Universal language denoted by L_u as the strings accepted by this Universal turing machine.

$$L_u = \{w_i\#w_j \,|\, w_j \in H(M_i)\}$$

A turing machine to accept $\overline{L_d}$ could just be one that takes in $w_i\#w_j$ and apart from running the Universal turing machine also runs a check to see if $i = j$.

9.11.6 Complement of Universal Language

The Universal language is recursively Enumerable since the Universal turing machine accepts it. What about its complement? Note that the notion of complement is not the usual definition here. The complement of L_u denoted by $\overline{L_u}$ would be

$$\overline{L_u} = \{w \,|\, w \notin \{w_i\#w_j| i, j \in N\} \text{ or } w_j \notin H(M_i)\}$$

However we are only interested in a version of $\overline{L_u}$ restricted to elements of the type $w_i\#w_j$. Thus the set becomes.

$$\overline{L_u} = \{w_i\#w_j \,|\, w_j \notin H(M_i)\}$$

Now lets see if $\overline{L_u}$ is Recursively enumerable. Assume the existence of such a turing machine say M. Such a turing machine can be used to accept the language L_d. For a given input i construct $w_i\#w_i$ and run it on M.

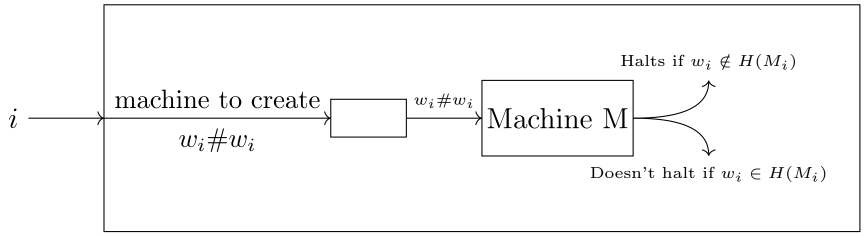
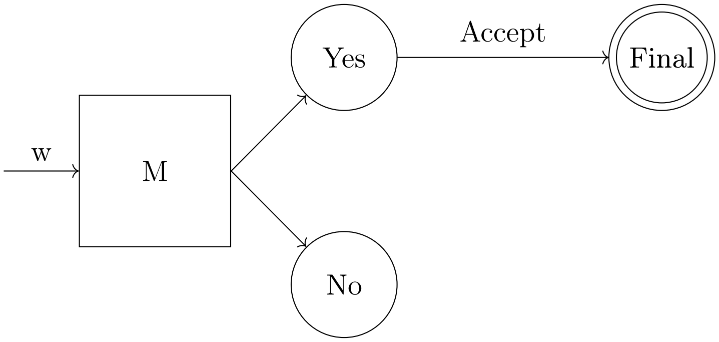

i
machine to create
$w_i\#w_i$
$w_i\#w_i$
Machine M
Halts if $w_i \notin H(M_i)$
Doesn't halt if $w_i \in H(M_i)$

Figure 9.34: Machine to accept L_d given M

9.12 Recursive Languages

Let there be a turing machine M that halts on all inputs i.e. $H(M) = \Sigma^*$ and a language L. But there are two halting states in the turing machine, one for accepting (Yes) and one for rejecting (No). If $w \in L(M)$, then M halts on w at the accepting state. If $w \notin L(M)$, then M halts on w at the rejecting state.
This turing machine can be converted to a turing machine that accepts by final state. This is done by adding a transition from the accepting halting state to the final state.



Definition: A language L is recursive if $\exists$ a turing machine M that halts on all inputs $(H(M) = \Sigma^*)$ and $L = L(M)$.

Claim: A language L is recursive iff $\overline{L}$ is recursive.
Proof: Instead of the transition from the "Yes" state to the "Final" state, we can have a transition from the "No" state to the "Final" state to now accept the strings in $\overline{L}$. So, if L is recursive, then $\overline{L}$ is also recursive and vice versa. □

9.12.1 Recursive or not?

Question: Is L_d recursive?
Answer: No, L_d is not recursive. Consider a word $w \in L_d$. We know that L_d is not recursively enumerable, so no turing machine exists that can halt on w and accept it. Hence it is impossible to construct a turing machine M that halts on all inputs.

Question: Is L_u recursive?

Answer: No, L_u is not recursive. Let us first see if $\overline{L_u}$ is recursive. The same arguments as that of the previous question can be made by considering a word $w \in \overline{L_u}$ and we know that $\overline{L_u}$ is not recursively enumerable. Hence $\overline{L_u}$ is not recursive and hence L_u is also not recursive.

9.12.2 Complementation of Recursive languages

As we saw clearly Recursively enumerable languages are not closed under complementation. A clear example is L_d and $\overline{L_d}$. What about recursive languages?
Its easy to see that they are closed under complementation. Assume a recursive language L_k exists and has a turing M_k such that $L(M_k) = L_k$. What could happen if a word from $\overline{L_k}$ is fed into the machine. We would get a no output. So to get a machine such that $L(M_{k'}) = \overline{L_k}$ we just need to input our string into M_k and then complement the output[2]. Since this can be easily done with a turing machine we can conclude that $\overline{L_k}$ is recursive.

9.12.3 Recursive → Recursively Enumerable

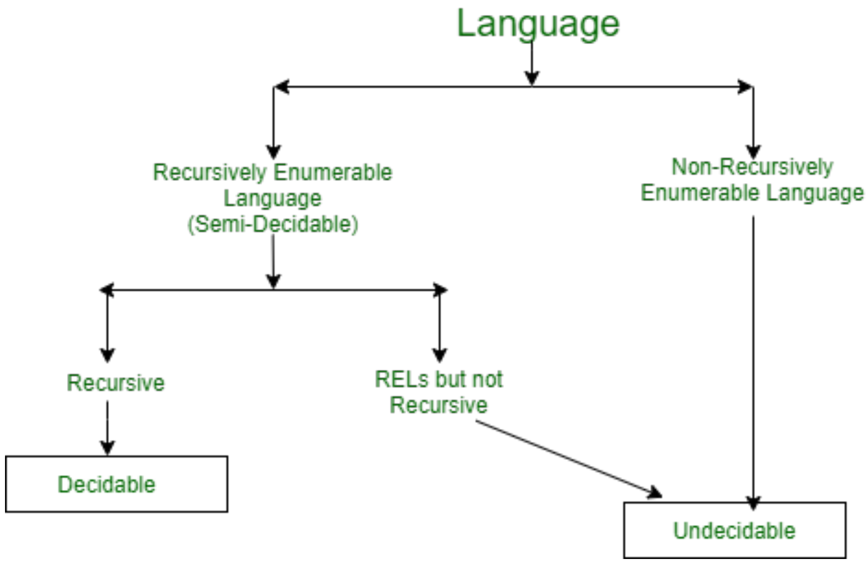
It is easy to see that if a language is recursive then it is also recursively enumerable since a turing machine which accepts by outputting yes can just be converted into one to accept the same language via halting. Whenever we get a no output we can make the turing machine go into an infinite loop and terminate for yes inputs. But the otherway is not so obvious.
As we saw recursive languages are closed under complementation. Thus $\overline{L_d}$ cannot be recursive. If it was recursive then its complement (ie) L_d will also be recursive. This means that $\overline{L_d}$ is also recursively enumerable but we have proved that it is not. We thus have a contradiction and L_d is not recursive. However it was recursively enumerable.
Thus being a recursive language is a stronger condition than a language being recursively enumerable.

9.13 Decidable languages

A language is sais to be **decidable** if it is recursive, else it is said to be undecidable. We also have semidecidable languages which are recursively enumerable but not recursive such as $\overline{L_d}$
Why do we care about decidable languages? As said above we care about languages such that we can determine if an arbitrary word is a member of it in finite time. This is because our computers cannot run for infinite time for us to get results. The existence of undecidable languages tells us that regardless of our model we will be unable to solve some problems involving such languages. This is an inherent problem in computation which cannot be fixed.
Definitions:

- A language L is decidable if it is recursive.
- A language L is undecidable if it is not recursive.
- A language L is semi-decidable if either L or $\overline{L}$ is recursively enumerable.

[2] Make our yes state a no state and vice versa


Language
Recursively Enumerable Language (Semi-Decidable)
Non-Recursively Enumerable Language
Recursive
RELs but not Recursive
Decidable
Undecidable

9.14 Rice Theorem

Decidability and Its Implications Decidability is a fundamental concept in computability and formal language theory. A decision problem is considered decidable if there exists an algorithm that can provide a correct yes or no answer for every input instance of the problem within a finite amount of time or in other words a language is said to be decidable if there exists a halting turing machine M which halts on every input in the language.

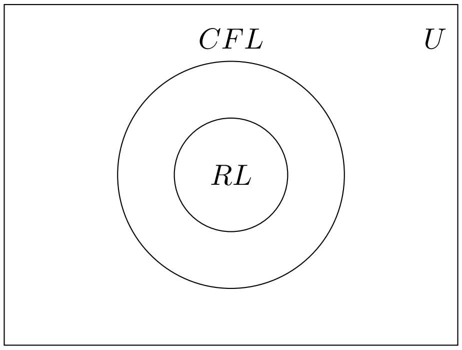


Figure 9.35: Relation between sets of Languages

9.14.1 Classification of Properties

In the study of formal languages, properties can be classified as either **trivial** or **non-trivial** based on how they partition the universal set U of all recursively enumerable languages.

Trivial properties are those for which either every language in U has the property or no language in U has it. Formally, a property P is trivial if $P = U$ or $P = \emptyset$, where P is the set of all languages in U

that satisfy the property. These properties do not provide meaningful information for distinguishing among languages because they apply universally or not at all.

Non-trivial properties are characterized by the existence of at least one language $L_1 \in U$ that has the property and at least one language $L_2 \in U$ that does not have the property. Formally, a property P is non-trivial if $\exists L_1 \in P$ and $\exists L_2 \in U \setminus P$. This distinction means that non-trivial properties can effectively differentiate between languages, making them particularly valuable for theoretical studies and practical applications in computational linguistics and automata theory.

9.14.2 Decidability of Non-Trivial Properties

Having established the distinction between trivial and non-trivial properties of languages, we now turn to a fundamental question in the theory of computation: *Is a given non-trivial property P, which is a strict subset of U, decidable?*

It is important to note that for a property to be considered non-trivial, it must be a strict subset of U (i.e., $P \subset U$ and $P \neq U$, $P \neq \emptyset$). This distinction excludes trivial properties, as they do not provide meaningful information for computational decisions.

Question: Given a Turing machine M, is the language recognized by M, denoted $L(M)$, a member of the property P? This question is central to understanding the decidability of properties of languages.

For example, consider the non-trivial property of regularity. The question becomes: Given a Turing machine M, is $L(M)$ regular?

The question posed earlierâwhether the language $L(M)$ recognized by a given Turing machine M belongs to a non-trivial property P is, in general, undecidable. This implies that no universal algorithm exists that can determine, for every conceivable Turing machine M, whether its recognized language $L(M)$ exhibits the property P.

For instance, while we might be able to verify the regularity of languages generated by certain types of automata, extending this verification to languages recognized by arbitrary Turing machines introduces problems that transcend the capabilities of algorithmic decision-making.

9.14.3 Proof of Rice's Theorem

Theorem : *For every non-trivial property P of U, the problem of identifying whether given M, is $H(M) \in P$, is undecidable.*

Proof :

Consider a non-trivial property P $\implies \exists L_1$ such that $L_1 \in P$ and $\exists L_2$ such that $L_2 \notin P$. Let M_1 and M_2 denote the turing machines such that $H(M_1) = L_1$ and $H(M_2) = L_2$, so from here we can say that $\exists M_2$ such that $H(M_2) \notin P$. (Remember this, we will use this later!)

We will prove this theorem by contradiction. Assume if possible that the problem is decidable, i.e. there exists a checker M'_P that is a turing machine that halts on all inputs and either accepts or rejects the given input based on whether or not it satisfies our property. Our turing machine looks as follows :

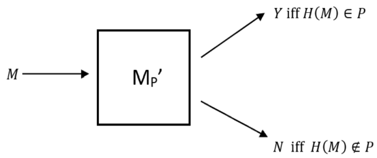


Figure 9.36: turing machine that checks if language of M satisfies the property

Consider the language $L = \phi$. Either $L \in P$ or $L \notin P$. If we have a machine for P (M'_P), we can construct a machine for P' ($M'_{\bar{P}}$) by simply swapping the Y and N outputs in M'_P. Hence we can assume without loss of generality that the empty language is in P (i.e. satisfies the property).

Now that we have established this, we will proceed to show contradiction. The idea is to construct a larger machine such that it does something we know can't be done. Consider the following machine $\mathcal{M}$ that takes as an input M_i and w_j and outputs Y if M_i doesn't halt on w_j and N if M_i halts on w_j. Given M'_P we will show step by step how we can construct such a machine.

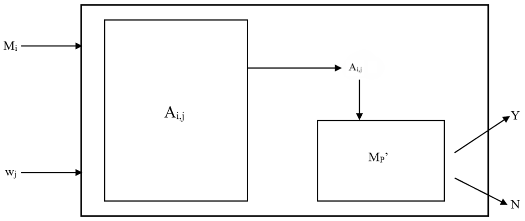


Figure 9.37: Larger machine $\mathcal{M}$ that detects whether M_i doesn't halt on w_j

The machine $A_{i,j}$ has two tapes - its input tape and a scratch tape. It takes the following sequence of steps :
1. Write w_j onto Tape 2 (scratch tape)
2. Mimics actions of M_i on w_j on Tape 2
3. Mimics actions of M_2 on Tape 1 which has the real input of $A_{i,j}$

Now consider the outputs of $A_{i,j}$ -

- If M_i halts on w_j, $H(A_{i,j})$ is exactly the halting language of M_2, i.e. the set of strings on which $A_{i,j}$ halts is exactly the set of strings for which M_2 halts or $H(A_{i,j}) = H(M_2)$.
- If M_i doesn't halt on w_j, Step 2 above will never terminate. Hence, $A_{i,j}$ will never halt regardless of input on Tape 1. So the language is empty, i.e. $H(A_{i,j}) = \phi$.

We started off by saying that $L = \phi \in P$ and $H(M_2) \notin P$. So passing $A_{i,j}$ into M'_P will give output:

- Y iff $H(A_{i,j}) \in P \iff H(A_{i,j}) = \phi \iff M_i$ doesn't halt on w_j.

- N iff $H(A_{i,j}) \notin P \iff H(A_{i,j}) = H(M_2) \iff M_i$ halts on w_j.

This means that the overall machine $\mathcal{M}$ takes M_i and w_j as input and returns Y if M_i doesn't halt on w_j and N if M_i halts on w_j. This means that $\mathcal{M}$ halts for every string in $\Sigma*$ and has halting language $H(\mathcal{M})$ as $\bar{L}_u$. We know that such a machine isn't possible because $\bar{L}_u$ is not recursive, hence we have a contradiction. There is nothing invalid about the construction of $A_{i,j}$, that means that our original assumption regarding the existence of M'_P was false.

Thus we are done, we have shown that for any non-trivial property P, given M the problem of determining whether $H(M) \in P$ is undecidable.

Applications of Rice's Theorem One of the many practical applications of Rice's Theorem is in the field of operating systems, particularly in the context of process management. Consider the problem of determining whether a process will lead to a crash in an operating system. According to Rice's Theorem, this is undecidable because the property P â whether the process will crash or not â is a non-trivial property of the computation performed by the process.

- **Property** P: A process crashes the operating system.
- **Implication**: It is undecidable to conclusively predict from the process code alone whether it will crash the system, as this involves predicting whether the execution trace of the process will enter a state that causes the crash.

This application is particularly significant because it highlights the limitations of static analysis tools in ensuring software reliability and security. Despite advances in software testing and verification, Rice's Theorem provides a fundamental limit on what can be achieved through purely algorithmic means.

Post Correspondence Problem (PCP) Problem : *Given two lists of strings $S_1 = [u_1, u_2, ...u_k]$ and $S_2 = [v_1, v_2, ...v_k]$ over some alphabet Σ. Does there exist some sequence $l = [i_1, i_2, ...i_m]$ such that $u_{i_1}u_{i_2}...u_{i_m} = v_{i_1}v_{i_2}...v_{i_m}$?*

Turns out that this problem is also undecidable. Refer to Section 9.4 of the book Introduction to Automata Theory, Languages and Computation by Hopcroft, Motwani and Ullman for a detailed proof of the same.

Chapter 10

First Order Logic

10.1 Introduction

Why FOL?
We began our discussion on logic by studying Propositional Logic, which we realised had a limitation that we could only have finite sets of assignments.

We then discussed some ways by which we could overcome this limitation, like PDAs and TMs. First-Order Logic is another way to overcome that limitation. In fact, Propositional logic is zeroth-order logic, FOL is first-order logic, and the most powerful is second-order logic, which is equivalent to Turing Machines.

FOL versus Propositional Logic
FOL is really a more capable version of propositional logic. While every propositional logic formula constitutes a First-Order Logic formula, the latter encompasses additional expressive power and complexity.
Let's look at some generalisations of FOL on propositional logic that make it more capable:

1. The variables in First-Order Logic can take on values from *any* underlying set, as opposed to the limited domain {true, false} of variables in propositional logic.
   This set could possibly be infinite, like $\mathbb{N}$.

2. FOL allows constants which are specific elements of the underlying set, which provides flexibility over Propositional Logic, where $\top$ and $\bot$ were the only constants.

3. Unlike the functions in propositional logic, which took inputs from a set and produced either true or false as output, the functions in FOL take inputs from an underlying set, and can also produce outputs belonging to a set (not just true/false).

10.1.1 Notation

Variables: $x, y, z, \ldots$
Variables, often denoted by lowercase letters at the end of the alphabet, represent the elements of an underlying set. Here the underlying set can be anything. For eg. it could be $\mathbb{N}$ (Infinite set of natural numbers) or it could be just {T, F}, a finite set. The latter was the case in propositional logic.

Constants: $a, b, c, \ldots$
Constants, often denoted by lowercase letters at the start of the alphabet, are specific elements of the same underlying set defined above.

Function symbols: $f, g, h, \ldots$
Function symbols are used to represent functions that take some (one or many) elements from the underlying set as arguments and produce an element from the underlying set.
Arity of function is the # of arguments to the function
0-ary functions are constants as they don't take any arguments

Relation (predicate) symbols: $P, Q, R, \ldots$
Predicate symbols are used to express properties or relations between the elements from the underlying set. It takes these elements as arguments and outputs True/False. Its range is boolean.
Again the arity of predicate is the # of arguments

Fixed symbols:
These are carried over from propositional logic: $\wedge, \vee, \neg, \rightarrow, \leftrightarrow, (,)$
The new thing that is introduced in FOL are the quantifiers $\exists$ (Existential), $\forall$ (Universal).

Equality in FOL: '=' is a special binary predicate called the *equality symbol.* It returns either true or false. Semantically, this predicate defines a binary identity relation (more on this in upcoming lectures).
As it is, the availability of the equality symbol provides us a higher expressive power as compared to FOL without the equality symbol. In all our discussions on FOL, we will assume access to the equality symbol, unless specified otherwise.

Syntax

Two classes of syntactic objects comprise first order logic : *terms* and *formulas.*

- **Terms:** Terms are expressions that denote objects or elements in the domain of discourse. They can be variables, constants, or the result of applying function symbols to other terms. Terms represent individual objects and do not involve any logical operations. A term evaluates to something from the underlying set. Formally, terms are defined as follows:

  1. Every variable is a term.
  2. If f is an m-ary function, $t_1, \ldots, t_m$ are terms, then $f(t_1, \ldots, t_m)$ is also a term.

  Examples of terms are: $x, a, h(f(x), g(y))$.
  Observe that constants are also terms since they are 0-ary functions.

- **Formulas:** Formulas are expressions that represent statements or propositions about objects in the domain of discourse. They can involve logical connectives, quantifiers, and predicate symbols. A formula evaluates to either true or false. Formulas are of two types:

  1. **Atomic Formulas:** These are statements that consist of predicate symbols applied to terms. Formally,

     (a) If R is an n-ary predicate, $t_1, \ldots, t_n$ are terms, then $R(t_1, \ldots, t_n)$ is an atomic formula.

(b) As a special case, $t_1 = t_2$ is an atomic formula.

For example: $P(x), Q(x, y), R(f(x), g(y)), x = y$

2. **Compound Formulas:** These are formed by combining atomic formulas using logical connectives like "and" ($\wedge$), "or" ($\vee$), "not" ($\neg$), implication ($\rightarrow$), and bi-implication ($\leftrightarrow$). For example: $(P(x) \wedge Q(y))$, $\neg R(z)$, $(P(x) \rightarrow Q(y))$

Formally, the rules for forming formulas are listed below.

1. Every atomic formula is a formula.
2. If φ is a formula, so are $\neg\varphi$ and (φ).
3. If φ_1 and φ_2 are formulas, so is $\varphi_1 \wedge \varphi_2$.
4. If φ is a formula, so is $\exists x\, \varphi$ for any variable x.

The fixed symbols $\wedge$, $\neg$ and $\exists$ are sufficient to define all formulas in FOL. These are called the *Primitive* fixed symbols. The other fixed symbols are used solely for convenience, and formulas with other fixed symbols are definable in terms of formulas with primitive symbols.

$$\begin{aligned} \varphi_1 \vee \varphi_2 & \neg(\neg\varphi_1 \wedge \neg\varphi_2) \\ \varphi_1 \rightarrow \varphi_2 & \neg\varphi_1 \vee \varphi_2 \\ \varphi_1 \leftrightarrow \varphi_2 & (\varphi_1 \rightarrow \varphi_2) \wedge (\varphi_2 \rightarrow \varphi_1) \\ \forall x\, \varphi & \neg(\exists x\, \neg\varphi) \end{aligned}$$

Summary of Syntactic Foundations

We have now formally defined the rules for the construction of formulas within the framework of propositional and first-order logic. This includes:

- Definitions and rules for constructing well-formed formulas (WFFs).
- Notations and conventions specific to our logical framework.

10.1.2 FOL formulas as strings

Alphabet

The alphabet over which strings (formulas) are constructed in logic includes:

- **Set of Variable Names:** For example, $\{x_1, x_2, y_1, y_2\}$.
- **Set of Constants, Functions, and Predicates:** For example, $\{a, b, f, =, P\}$.
- **Fixed Symbols:** These include logical operators and quantifiers: $\{\neg, \vee, \wedge, \rightarrow, \leftrightarrow, \exists, \forall\}$.

Well-Formed Formulas

A well-formed formula (WFF) is a string formed according to specific syntactic rules, typically detailed in previous discussions or slides. For instance:

- A **valid example** of a WFF in this framework is:

$$\forall x_1(\forall x_2(((x_1 = a) \vee (x_1 = b)) \wedge \neg(f(x_2) = f(x_1))))$$

This formula follows all syntactic rules for quantifiers, operators, and function application.

- An **invalid example**, which does not adhere to the syntactic rules, is:

  $$\forall(\forall x_1(x_1 = ab)\neg()x_2)$$

  This string is not well-formed due to improper use of quantifiers and logical operators. $x - 1 = ab$ is a clear fault as there is no operator present between two constants.

Representation Using Parse Trees
Well-formed formulas can be represented using parse trees, which visually depict the syntactic structure of the formula. Consider the rules detailed on the previous slide as production rules in a context-free grammar:

Vocabulary

- A part of alphabet is of special importance .It is given below.
- **Set of Constants, Functions, and Predicates:** For example, $\{a, b, f, =, P\}$.

Let's consider the smallest vocabulary needed to construct the following formula:

$$\forall x_1\,(\forall x_2\,(((x_1 = a) \vee (x_1 = b)) \wedge \neg\,(f(x_2) = f(x_1))))$$

The smallest vocabulary required for this formula includes:

- Constants: a, b
- Function symbols: f
- Predicate symbols: $=$
- The answer is $\{a, b, f, =\}$.

10.1.3 Free variables in a formula

Free variables are the variables that occur in a formula without being bound by a quantifier. Essentially these are the variables that a user has the freedom to assign values to.

In propositional logic, all the variables in a formula were "free". In FOL, due to the introduction of quantifiers, you cannot assign values to some variables at will (in particular to the quantified variables). These variables are assigned values by the semantics of the formula. Thus everytime you apply a quantifier, you are potentially reducing a free variable.
Note that when we say you can "assign values to a variable at will", the values assigned must always belong to the underlying set.

Let free(φ) denote the set of free variables in φ.

- If φ is an atomic formula, free(φ) = $\{x \,|\, x \text{ occurs in } \varphi\}$
- If $\varphi = \neg\psi$ or $\varphi = (\psi)$, free(φ) = free(ψ)
- If $\varphi = \varphi_1 \wedge \varphi_2$, free($\varphi$) = free($\varphi_1$) ∪ free($\varphi_2$)

- If $\varphi = \exists x\, \varphi_1$, $\text{free}(\varphi) = \text{free}(\varphi_1) \setminus \{x\}$

Let's look at an example. What is $\text{free}((\exists x\, P(x,y)) \wedge (\forall y\, Q(x,y)))$?

$\text{free}((\exists x\, P(x,y)) \wedge (\forall y\, Q(x,y)))$
$= \text{free}((\exists x\, P(x,y))) \cup \text{free}(\forall y\, Q(x,y))$
$= \text{free}(P(x,y)) \setminus \{x\} \cup \text{free}(Q(x,y)) \setminus \{y\}$
$= \{x,y\} \setminus \{x\} \cup \{x,y\} \setminus \{y\}$
$= \{x,y\}$

Does this mean we can assign any value we want to x and y wherever they appear in the function? No, since x is a free variable only with respect to $(\forall y\, Q(x,y))$, and y is free only with respect to $(\exists x\, P(x,y))$.

If a formula φ has free variables $\{x,y\}$, we denote it as $\varphi(x,y)$.
A formula with no free variables is called a **sentence.** For example, $\exists x \forall y\, f(x) = y$ is a sentence. Sentences form a very important class of formulae in FOL.

Every free variable in a formula must be assigned a specific value from the underlying domain to make the formula meaningful. The choice of the value to assign is up to the user.

Bound variables in a formula
Bound variables in a formula in FOL are variables that are quantified in the formula. These variables are said to be "bound" because their meaning or interpretation is determined by the quantifier that binds them.

Let $\text{bnd}(\varphi)$ denote the set of bound variables in φ.

- If φ is an atomic formula, $\text{bnd}(\varphi) = \emptyset$
- If $\varphi = \neg\psi$ or $\varphi = (\psi)$, $\text{bnd}(\varphi) = \text{bnd}(\psi)$
- If $\varphi = \varphi_1 \wedge \varphi_2$, $\text{bnd}(\varphi) = \text{bnd}(\varphi_1) \cup \text{bnd}(\varphi_2)$
- If $\varphi = \exists x\, \varphi_1$, $\text{bnd}(\varphi) = \text{bnd}(\varphi_1) \cup \{x\}$

Let us look at an example. Consider the same formula as before: what is $\text{bnd}((\exists x\, P(x,y)) \wedge (\forall y\, Q(x,y)))$?

$\text{bnd}((\exists x\, P(x,y)) \wedge (\forall y\, Q(x,y)))$?
$= \text{bnd}((\exists x\, P(x,y))) \cup \text{bnd}(\forall y\, Q(x,y))$
$= \text{bnd}(P(x,y)) \cup \{x\} \cup \text{bnd}(Q(x,y)) \cup \{y\}$
$= \emptyset \cup \{x\} \cup \emptyset \cup \{y\}$
$= \{x\} \cup \{y\}$
$= \{x,y\}$

Just like in the free case, x and y are not bound throughout the formula - x is bound with respect to $(\exists x\, P(x,y))$, and y is bound with respect to $(\forall y\, Q(x,y))$.
Hence, $\text{free}(\varphi)$ and $\text{bnd}(\varphi)$ are not complements!

10.1.4 Substitution in First Order Logic (FOL)

Introduction to Substitution

Substitution in logical formulas involves replacing every free occurrence of a variable x in a formula φ with a term t, ensuring that the free variables in t remain free in the resulting formula.

Conditions for Substitution

A term t is said to be **free for** x **in** φ if no free occurrence of x in φ is within the scope of a quantifier ($\forall y$ or $\exists y$) that quantifies any variable y occurring in t.

Example and Definition

Consider the formula:

$$\varphi \equiv \exists y R(x, y) \vee \forall x P(z, x)$$

and a term $t = f(z, x)$.

Analyzing Substitution in Parts of a Formula:

Consider a composite formula divided into two parts:

$$\varphi \equiv \exists y R(x, y) \vee \forall x P(z, x)$$

In this formula:

- In the first part, $\exists y R(x, y)$, x is a free variable.
- In the second part, $\forall x P(z, x)$, z is a free variable, and x is bound.

Implications for Substitution:

While substituting, we must ensure that the variables within the term being substituted do not become bound in the process. This means:

- Substituting any term in the first subpart where the free variable is y could be problematic if the substituted term contains y as a free variable. Doing so could inadvertently bind y within the scope of its own quantifier, which violates the conditions for a free substitution.
- Similarly, substituting any term for z in the second subpart that includes x as a free variable is incorrect, as it could result in x becoming bound by the universal quantifier in that subpart.
- Here, $f(z, x)$ is free for x in φ, but $f(y, x)$ is not, because substituting $f(y, x)$ for x would place y, which is quantified in φ, within its own quantifier's scope.

Procedure for Substitution:

1. Identify where the variable to be substituted is free in the formula.
2. Assess whether the free variables within the substituting term will remain free post-substitution.
3. If substituting the term causes any originally free variable to become bound, then the substitution is not valid in that context.

By following these steps, we ensure the logical integrity of the formula remains intact after substitution.

Substitution Notation
The result of substituting t for x in φ, denoted $\varphi[t/x]$, is a formula obtained by replacing each free occurrence of x in φ by t, provided that t is free for x in φ.
Given the formula φ defined above, the substitution of $f(z,x)$ for x is written as:

$$\varphi[f(z,x)/x] \equiv \exists y R(f(z,x),y) \vee \forall x P(z,x)$$

Intuition For Substitution
Consider the formula:

$$\varphi \equiv \exists y R(x,y) \vee \forall x P(z,x)$$

While the following is not a technically correct analogy, especially when you evaluate a first-order logic formula on a structure with an infinite universe, you can go through it to get an intuition of how bound and free variables work. Think of the following code as having z and x as arguments. These are the free variables in our formula. Now consider R(x,y) as c++ bool function .That has two arguments x and y .Now consider the part $\exists y R(x,y)$.

```
bool result = false;
for (auto y : underlying_set) {
    if (R(x, y)) {
        result = true;
        break;  // exit loop
    }
}
```

So here since y is a bound variable its value is itself being put the subformula we do not have freedom to put its value.Now the argument x is not given by subformula to R we need to fetch it from user for $R(x,y)$ to run.

Now let us have a look at the second subformula $\forall x P(x,z)$.Here P(x,z) is a boolean function which takes two argument x and z. Its corresponding c++ code is

```
bool result1 = true;
for (auto x : underlying_set) {
    if (P(x, z)==false) {
        result1 = false;
        break;  // exit loop
    }
}
```

We can clearly see the value of x is being provided by subformula to P but on the other had we need to provide it with value of z.

Finally our formula returns the truth value of $result \vee result1$..
So if we want to substitute x by some term $f(x,z)$.We cannot put this where x is being provided by subformula .We can only substitute it in places where x is given by user.

Secondly when we put it where z was being provided by subformula then we made an error as the term had z as free variable ,now it loses control over its free variable as now it will be provided by subformula.So we should be cautiuos of such errors.

Given the formula φ defined above, the substitution of $f(z, x)$ for x is written as:

$$\varphi[f(z, x)/x] \equiv \exists y R(f(z, x), y) \vee \forall x P(z, x)$$

.

So its corresponding code will look like.

```
bool result = false;
for (auto y : underlying_set) {
    if (R(f(x,z), y)) {
        result = true;
        break;  // exit loop
    }
}

bool result1 = true;
for (auto x : underlying_set) {
    if (P(x, z)==false) {
        result1 = false;
        break;  // exit loop
    }
}
```

Clearly the x in P is not touched.
Once again, the above is not a technically correct way of understanding the effect of substitution. It is only provided for you to get an intuition by an analogy with something you are probably familiar with (programming in C++, refactoring, scoping of variables etc.)