

# 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

|          |   |           |
|----------|---|-----------|
| <b>1</b> | <b>Propositional Logic</b>  | <b>4</b>  |
| 1.1      | Syntax . . . . .  | 4         |
| 1.2      | Semantics . . . . .   | 5         |
| 1.2.1    | Important Terminology . . . . .                                   | 7         |
| 1.3      | Proof Rules . . . . .   | 8         |
| 1.4      | Natural Deduction . . . . .                                       | 10        |
| 1.5      | Soundness and Completeness of our proof system . . . . .          | 11        |
| 1.6      | What about Satisfiability? . . . . .                              | 13        |
| 1.7      | Algebraic Laws and Some Redundancy . . . . .                      | 14        |
| 1.7.1    | Distributive Laws . . . . .                                       | 14        |
| 1.7.2    | Reduction of bi-implication and implication . . . . .             | 14        |
| 1.7.3    | DeMorgan's Laws . . . . .   | 14        |
| 1.8      | Negation Normal Forms . . . . .                                   | 14        |
| 1.9      | From DAG to NNF-DAG . . . . .                                     | 15        |
| 1.10     | An Efficient Algorithm to convert DAG to NNF-DAG . . . . .        | 19        |
| 1.11     | Conjunctive Normal Forms . . . . .                                | 23        |
| 1.12     | Satisfiability and Validity Checking . . . . .                    | 25        |
| 1.13     | DAG to Equisatisfiable CNF . . . . .                              | 26        |
| 1.14     | Tseitin Encoding . . . . .  | 26        |
| 1.15     | Towards Checking Satisfiability of CNF and Horn Clauses . . . . . | 27        |
| 1.16     | Counter example for Horn Formula . . . . .                        | 28        |
| 1.16.1   | Example . . . . .   | 28        |
| 1.17     | Davis Putnam Logemann Loveland (DPLL) Algorithm . . . . .         | 29        |
| 1.18     | DPLL in action . . . . .  | 31        |
| 1.18.1   | Example . . . . .   | 31        |
| 1.19     | Applying DPLL Algorithm to Horn Formulas . . . . .                | 32        |
| 1.20     | DPLL on Horn Clauses . . . . .                                    | 33        |
| 1.21     | Rule of Resolution . . . . .                                      | 33        |
| 1.21.1   | Completeness of Resolution for Unsatisfiability of CNFs . . . . . | 34        |
| <b>2</b> | <b>DFAs and Regular Languages</b>                                 | <b>36</b> |
| 2.1      | Definitions . . . . .   | 36        |
| 2.2      | Deterministic Finite Automata . . . . .                           | 37        |
| 2.3      | DFA Design - Example 1 . . . . .                                  | 38        |
| 2.4      | DFA Design - Example 2 . . . . .                                  | 39        |

|          |  |           |
|----------|--|-----------|
| <b>3</b> | <b>Non-Deterministic Finite Automata (NFA)</b>               | <b>41</b> |
| 3.1      | NFA Representation . . . . .                                 | 41        |
| 3.1.1    | Formalization of Non-Deterministic Finite Automata . . . . . | 42        |
| 3.1.2    | NFA into action . . . . .                                    | 42        |
| 3.2      | Equal expressiveness of DFA or NFA . . . . .                 | 43        |
| 3.2.1    | Construction of DFA from NFA . . . . .                       | 43        |
| 3.2.2    | Step Wise Conversion from NFA to DFA . . . . .               | 43        |
| 3.2.3    | Proof of Equivalence . . . . .                               | 45        |
| 3.3      | Reflective Insights . . . . .                                | 45        |
| 3.4      | Proof of Equivalence – Correctness . . . . .                 | 45        |
| 3.4.1    | Claim . . . . .  | 45        |
| 3.4.2    | Proof . . . . .  | 46        |
| 3.5      | Compiler Special Case – Lexical Analyzer . . . . .           | 47        |
| 3.6      | NFA with $\epsilon$ –edges . . . . .                         | 49        |
| 3.6.1    | Equivalence with DFA . . . . .                               | 50        |
| 3.7      | Recap . . . . .  | 52        |
| 3.7.1    | Converting $\epsilon$ -NFA to non- $\epsilon$ -NFA . . . . . | 52        |
| 3.7.2    | Correctness of Algorithm . . . . .                           | 55        |
| 3.7.3    | Intuition of the algorithm . . . . .                         | 56        |
| 3.7.4    | Extras . . . . .   | 56        |
| 3.7.5    | Significance of $\epsilon$ -edges . . . . .                  | 57        |
| 3.7.6    | Examples . . . . .   | 58        |
| 3.8      | Equivalence in Finite Automata . . . . .                     | 59        |
| 3.9      | Regular Expression . . . . .                                 | 59        |
| 3.9.1    | Introduction . . . . .                                       | 59        |
| 3.9.2    | Formal Definition of a Regular Expression . . . . .          | 59        |
| 3.9.3    | Semantics of Regular Language . . . . .                      | 60        |

# 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  $C_1, \dots, C_5$ , four slots  $S_1, \dots, S_4$ , and five days  $D_1, \dots, D_5$ . 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  $S_1, \dots, S_4$ .
- 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  $p, q, r, x, y, z, \dots$  etc. Here is a short description of propositional logic symbols:

- **Variables:** They are usually denoted by smalls ( $p, q, r, x, y, z, \dots$  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  $\perp$ . 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             | $\perp$           | 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  $[n] = \{1, \dots, n\}$ ) 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  $\perp$  are formulae.
- If  $\varphi$  is a formula, so are  $\neg\varphi$  and  $(\varphi)$ .
- If  $\varphi_1$  and  $\varphi_2$  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 strings over the alphabet  $\mathbf{Vars} \cup \{\top, \perp, \neg, \wedge, \vee, \rightarrow, \leftrightarrow, (, )\}$ .  $\mathbf{Vars}$  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  $\varphi$  of  $n$  variables is a function

$$\llbracket \varphi \rrbracket : \{0, 1\}^n \rightarrow \{0, 1\}$$

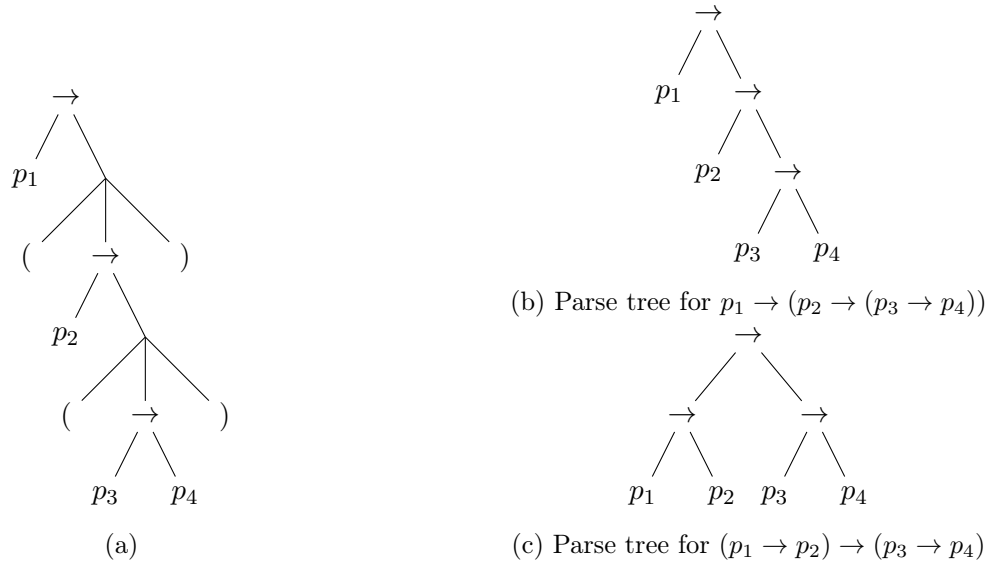


Figure 1.1: Parse trees obviate the need for parentheses.

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

|           |               |             |             |                              |             |             |                            |
|-----------|---------------|-------------|-------------|------------------------------|-------------|-------------|----------------------------|
| $\varphi$ | $\neg\varphi$ | $\varphi_1$ | $\varphi_2$ | $\varphi_1 \wedge \varphi_2$ | $\varphi_1$ | $\varphi_2$ | $\varphi_1 \vee \varphi_2$ |
| 0         | 1             | 0           | 0           | 0                            | 0           | 0           | 0                          |
| 1         | 0             | 0           | 1           | 0                            | 0           | 1           | 1                          |
|           |               | 1           | 0           | 0                            | 1           | 0           | 1                          |
|           |               | 1           | 1           | 1                            | 1           | 1           | 1                          |

(a) Truth table for  $\neg\varphi$ .      (b) Truth table for  $\varphi_1 \wedge \varphi_2$ .      (c) Truth table for  $\varphi_1 \vee \varphi_2$ .

|             |             |                                   |             |             |                                       |
|-------------|-------------|-----------------------------------|-------------|-------------|---------------------------------------|
| $\varphi_1$ | $\varphi_2$ | $\varphi_1 \rightarrow \varphi_2$ | $\varphi_1$ | $\varphi_2$ | $\varphi_1 \leftrightarrow \varphi_2$ |
| 0           | 0           | 1                                 | 0           | 0           | 1                                     |
| 0           | 1           | 1                                 | 0           | 1           | 0                                     |
| 1           | 0           | 0                                 | 1           | 0           | 0                                     |
| 1           | 1           | 1                                 | 1           | 1           | 1                                     |

(d) Truth table for  $\varphi_1 \rightarrow \varphi_2$ .      (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  $\perp$ : 0 (false) and 1 (true) are meanings, while  $\top$  and  $\perp$  are symbols.

#### Rules of semantics:

- $\llbracket \neg\varphi \rrbracket = 1$  iff  $\llbracket \varphi \rrbracket = 0$ .
- $\llbracket \varphi_1 \wedge \varphi_2 \rrbracket = 1$  iff  $\llbracket \varphi_1 \rrbracket = \llbracket \varphi_2 \rrbracket = 1$ .
- $\llbracket \varphi_1 \vee \varphi_2 \rrbracket = 1$  iff at least one of  $\llbracket \varphi_1 \rrbracket$  or  $\llbracket \varphi_2 \rrbracket$  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  $\varphi$  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 ( $\varphi$  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  $\varphi$  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  $\varphi$  is valid then  $\neg\varphi$  is UNSAT.
- **semantically entail**  $\varphi_1$  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  $\varphi$  evaluates to 1 then  $\varphi_1$  will evaluate to 1. Equivalently  $\varphi \rightarrow \varphi_1$  is valid.



- **semantically equivalent** to  $\varphi_1$  iff  $\varphi \models \varphi_1$  and  $\varphi_1 \models \varphi$ . Basically  $\varphi$  and  $\varphi_1$  have identical truth tables. Equivalently,  $\varphi \leftrightarrow \varphi_1$  is valid.
- **equisatisfiable** to  $\varphi_1$  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    | $\neg p \models p \rightarrow q$ |
| semantically equivalent | $p \rightarrow q, \neg p \vee q$ |
| equisatisfiable         | $p \wedge q, r \vee s$           |

Table 1.4: Some examples for the definitions.

**Example.** Consider the formulas  $\varphi_1 : p \rightarrow (q \rightarrow r)$ ,  $\varphi_2 : (p \wedge q) \rightarrow r$  and  $\varphi_3 : (q \wedge \neg r) \rightarrow \neg p$ . The three formulas  $\varphi_1$ ,  $\varphi_2$  and  $\varphi_3$  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  $\neg(\sum_{k=1}^4 \sum_{j=1}^5 p_{ijk} \leq 2)$ .

### 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

$$\frac{\varphi_1 \wedge \varphi_2}{\varphi_1} \wedge_{e1}$$

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  $\varphi_1$  and  $\varphi_2$  are true if  $\varphi_1 \wedge \varphi_2$  is true, so  $\varphi_1$  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  $\wedge_{e1}$  as  $\varphi_1$  is kept in the inference even when both  $\varphi_1$  and  $\varphi_2$  could be kept in inference. If we use  $\varphi_2$  in inference then the rule becomes  $\wedge_{e2}$ .

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

| Connector     | Introduction  | Elimination   |
|---------------|---|---|
| $\wedge$      | $\frac{\varphi_1 \quad \varphi_2}{\varphi_1 \wedge \varphi_2} \wedge_i$   | $\frac{\varphi_1 \wedge \varphi_2}{\varphi_1} \wedge_{e1} \quad \frac{\varphi_1 \wedge \varphi_2}{\varphi_2} \wedge_{e2}$       |
| $\vee$        | $\frac{\varphi_1}{\varphi_1 \vee \varphi_2} \vee_{i1} \quad \frac{\varphi_2}{\varphi_1 \vee \varphi_2} \vee_{i2}$             | $\frac{\varphi_1 \vee \varphi_2 \quad \varphi_1 \rightarrow \varphi_3 \quad \varphi_2 \rightarrow \varphi_3}{\varphi_3} \vee_e$ |
| $\rightarrow$ | $\frac{\boxed{\begin{array}{c} \varphi_1 \\ \vdots \\ \varphi_2 \end{array}}}{\varphi_1 \rightarrow \varphi_2} \rightarrow_i$ | $\frac{\varphi_1 \quad \varphi_1 \rightarrow \varphi_2}{\varphi_2} \rightarrow_e$   |
| $\neg$        | $\frac{\boxed{\begin{array}{c} \varphi \\ \vdots \\ \perp \end{array}}}{\neg \varphi} \neg_i$                                 | $\frac{\varphi \quad \neg \varphi}{\perp} \neg_e$   |
| $\perp$       |   | $\frac{\perp}{\varphi} \perp_e$   |
| $\neg\neg$    |   | $\frac{\neg\neg\varphi}{\varphi} \neg\neg_e$  |

Table 1.5: Proof rules.

In the  $\rightarrow_i$  rule, the box indicates that we can *temporarily* assume  $\varphi_1$  and conclude  $\varphi_2$  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  $(\varphi_1 \wedge \varphi_2) \wedge \varphi_3$ .

$$\begin{array}{c}
 \frac{\varphi_1 \wedge (\varphi_2 \wedge \varphi_3)}{\varphi_2 \wedge \varphi_3} \wedge_{e2} \quad \frac{\varphi_1 \wedge (\varphi_2 \wedge \varphi_3)}{\varphi_1} \wedge_{e1} \\
 \frac{\varphi_2 \wedge \varphi_3}{\varphi_2} \wedge_{e1} \quad \frac{\varphi_2 \wedge \varphi_3}{\varphi_3} \wedge_{e2} \\
 \frac{\varphi_1 \quad \varphi_2}{\varphi_1 \wedge \varphi_2} \wedge_i \\
 \frac{\varphi_1 \wedge \varphi_2 \quad \varphi_3}{(\varphi_1 \wedge \varphi_2) \wedge \varphi_3} \wedge_i
 \end{array}$$

## 1.4 Natural Deduction

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

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

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  $\varphi$  at a point if it occurs prior to it in the proof and if **no box enclosing that occurrence of  $\varphi$  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. | $p \vee \neg p$           | $\vee_{i1} 2$  |
| 4. | $\perp$                   | $\neg_e 3, 1$  |
| 5. | $\neg p$                  | $\neg_i 2-4$   |
| 6. | $p \vee \neg p$           | $\vee_{i2} 5$  |
| 7. | $\perp$                   | $\neg_e 6, 1$  |
| 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. | $\perp$      | $\neg_e 1, 2$ |
| 4. | $\neg\neg p$ | $\neg_i 2-3$  |

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

|    |                   |                     |
|----|-------------------|---------------------|
| 1. | $p \rightarrow q$ | premise             |
| 2. | $\neg q$          | premise             |
| 3. | $p$               | assumption          |
| 4. | $q$               | $\rightarrow_e$ 3,1 |
| 5. | $\perp$           | $\neg_e$ 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.  | $\neg p$               | $\wedge_{e1}$ 1     |
| 5.  | $\perp$                | $\neg_e$ 3,4        |
| 6.  | $p \rightarrow \perp$  | $\rightarrow_i$ 3-5 |
| 7.  | $q$                    | assumption          |
| 8.  | $\neg q$               | $\wedge_{e2}$ 1     |
| 9.  | $\perp$                | $\neg_e$ 7,8        |
| 10. | $q \rightarrow \perp$  | $\rightarrow_i$ 7-9 |
| 11. | $\perp$                | $\vee_e$ 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.  $\Sigma = \{p \rightarrow q, \neg q\} \models \neg p$ .

| $p$ | $q$ | $p \rightarrow q$ | $\neg q$ | $\neg p$ |
|-----|-----|-------------------|----------|----------|
| 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  $p \rightarrow q$  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  $\Sigma$  imply  $\varphi$ <sup>1</sup>. And to prove an implication, we can use the  $\rightarrow_i$  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. | $(p \rightarrow q) \wedge \neg q$                      | assumption          |
| 4. | $\neg p$   | 1                   |
| 5. | $((p \rightarrow q) \wedge \neg q) \rightarrow \neg p$ | $\rightarrow_i$ 3,4 |

Similarly to mimic the second row, we would like to show  $\neg p, q \vdash ((p \rightarrow q) \wedge \neg q) \rightarrow \neg p$ . 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. | $(p \rightarrow q) \wedge \neg q$                      | assumption          |
| 4. | $\neg p$   | 1                   |
| 5. | $((p \rightarrow q) \wedge \neg q) \rightarrow \neg p$ | $\rightarrow_i$ 3,4 |

|    |  |                     |
|----|--|---------------------|
| 1. | $\neg p$   | given               |
| 2. | $q$  | given               |
| 3. | $(p \rightarrow q) \wedge \neg q$                      | assumption          |
| 4. | $\neg p$   | 1                   |
| 5. | $((p \rightarrow q) \wedge \neg q) \rightarrow \neg p$ | $\rightarrow_i$ 3,4 |

|    |  |                           |
|----|--|---------------------------|
| 1. | $p$  | given                     |
| 2. | $\neg q$   | given                     |
| 3. | $(p \rightarrow q) \wedge \neg q$                      | assumption                |
| 4. | $p \rightarrow q$                                      | $\wedge_e$ 3              |
| 5. | $q$  | $\rightarrow_e$ 1,4       |
| 6. | $\perp$  | $\neg_e$ 2,5              |
| 7. | $\neg p$   | $\perp_e$ 6               |
| 8. | $((p \rightarrow q) \wedge \neg q) \rightarrow \neg p$ | $\rightarrow_i$ 3,4,5,6,7 |

|    |  |                             |
|----|--|-----------------------------|
| 1. | $p$  | given                       |
| 2. | $q$  | given                       |
| 3. | $(p \rightarrow q) \wedge \neg q$                      | assumption                  |
| 4. | $p \rightarrow q$                                      | $\wedge_e$ 3                |
| 5. | $\neg q$   | $\wedge_e$ 3                |
| 6. | $q$  | $\rightarrow_e$ 1,4         |
| 7. | $\perp$  | $\neg_e$ 2,6                |
| 8. | $\neg p$   | $\perp_e$ 7                 |
| 9. | $((p \rightarrow q) \wedge \neg q) \rightarrow \neg p$ | $\rightarrow_i$ 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.

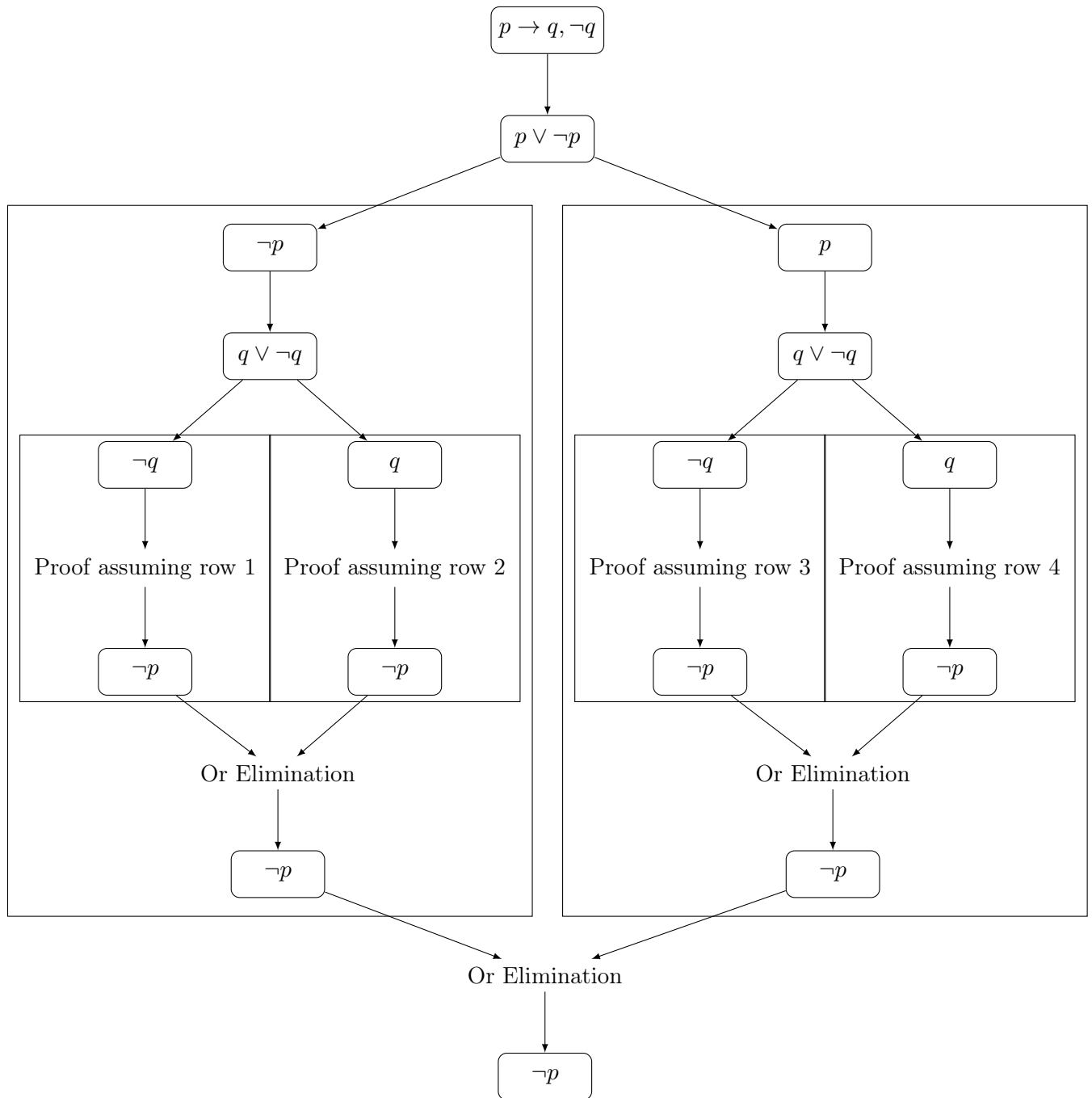
But as seen above, we were able to show a proof for the sequent  $\vdash \varphi \vee \neg \varphi$ . If we just recursively apply this property for all the variables we have, we should be able to capture every row of truth table. So combining this result, our proofs for each row of the truth table, 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  $\bigwedge_{\phi \in \Sigma} \phi \rightarrow \varphi$ .

This can be done using **structural induction** to prove the following:

Let  $\varphi$  be a formula using the propositional variables  $p_1, p_2, \dots, p_n$ . 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:

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

<sup>1</sup> $\Sigma$  semantically entails  $\varphi$  is equivalent to saying intersection of formulae in  $\Sigma$  implies  $\varphi$  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  $q$ ,

$$\not\models p \wedge q$$

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  $\phi$  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.

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

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

### 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.

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

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

### 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 (discussed in the next section).

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

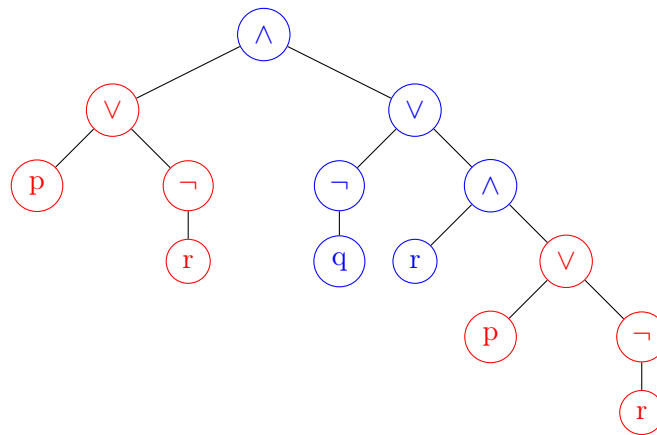
$$\neg(\phi_1 \vee \phi_2) \models (\neg\phi_1 \wedge \neg\phi_2)$$

## 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

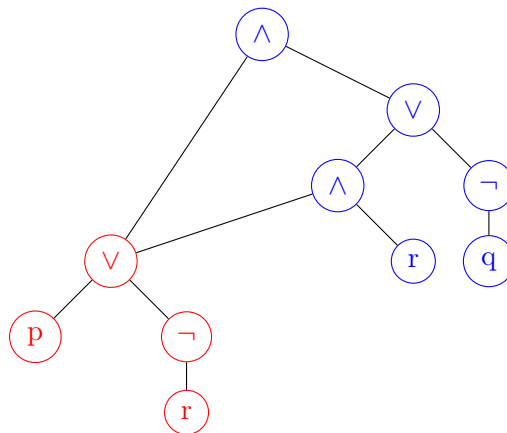
A NNF formula may be represented using its parse tree, which doesn't have any negation nodes except at the leaves, consider  $(p \vee \neg r) \wedge (\neg q \vee (r \wedge (p \vee \neg r)))$

**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**

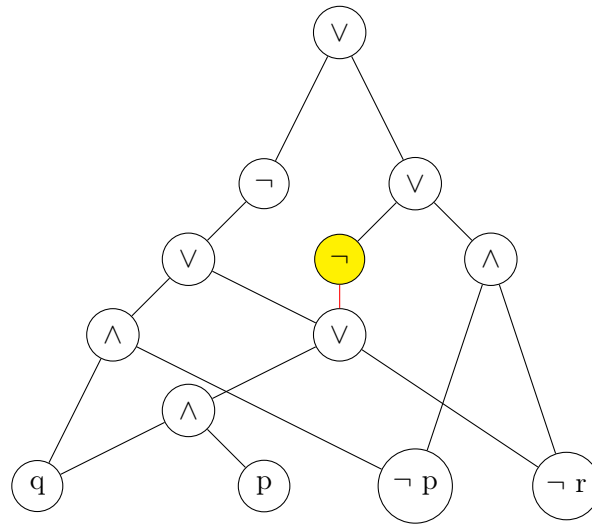


## 1.9 From DAG to NNF-DAG

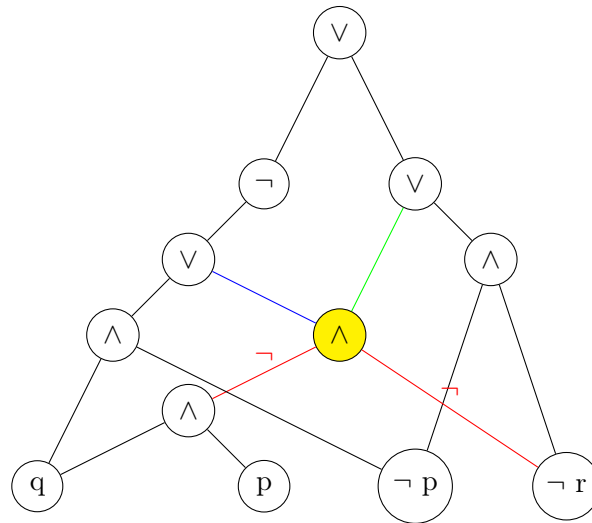
Given a DAG of propositional logic formula with only  $\vee$  ,  $\wedge$  and  $\neg$  nodes , can we efficiently get a DAG representing a semantically equivalent NNF formula ?

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





Pushing down the highlighted  $\neg$  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.,  $\neg$  pushing) only 1 of the copies and if a node wants non-negated node then take that node from the copied tree.

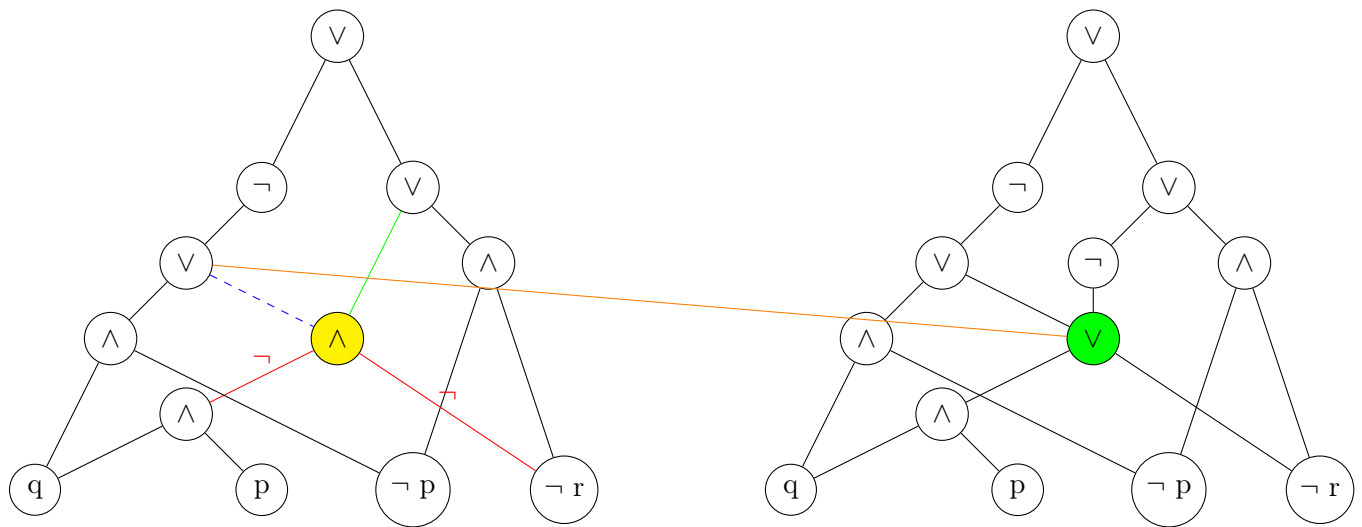


Figure 1.3: Step 1

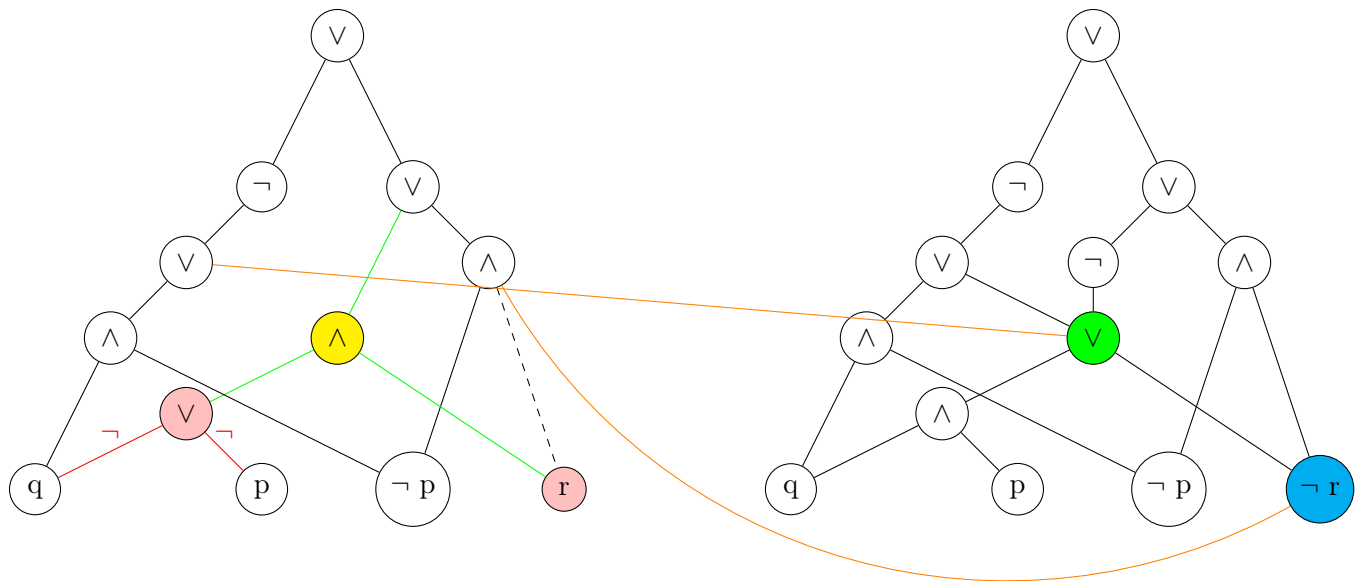


Figure 1.4: Step 2

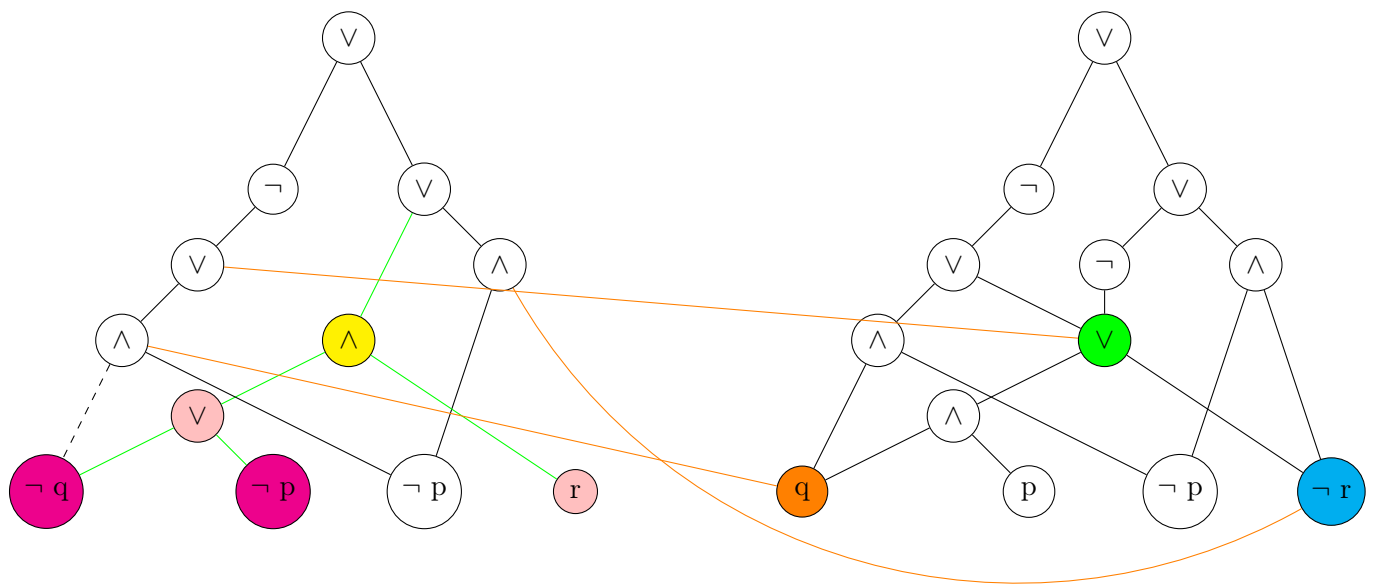


Figure 1.5: Step 3

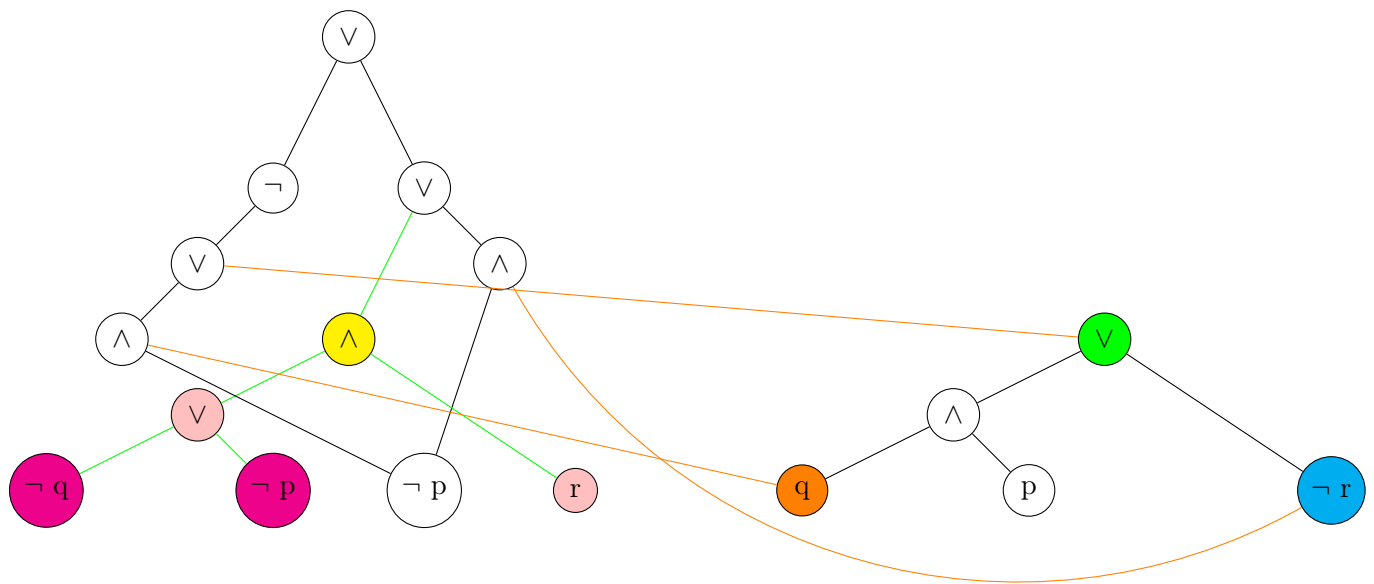


Figure 1.6: Step 4 : Remove all redundant nodes

### 1.10 An Efficient Algorithm to convert DAG to NNF-DAG

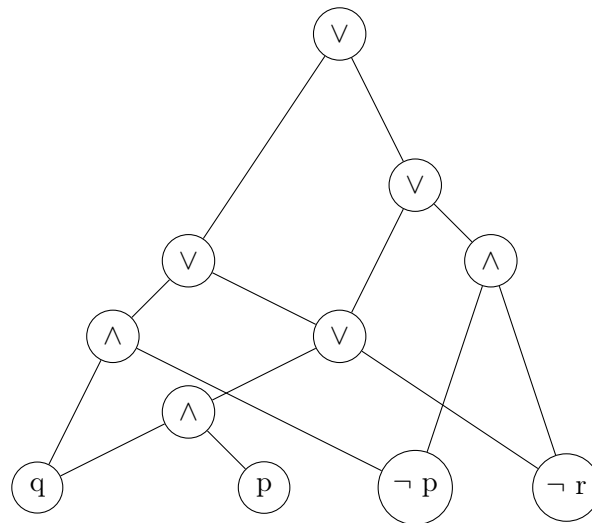


Figure 1.7: Step 1: Make a copy of the DAG and Remove all "¬" nodes except the ones which are applied to the basic variables

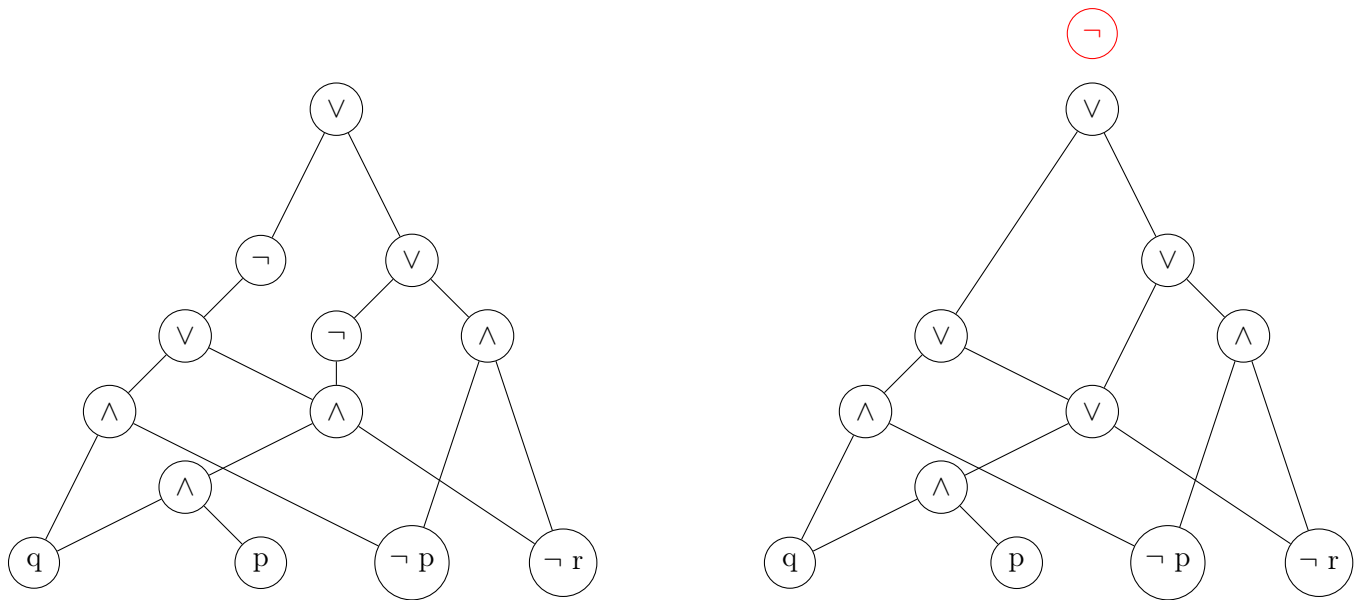
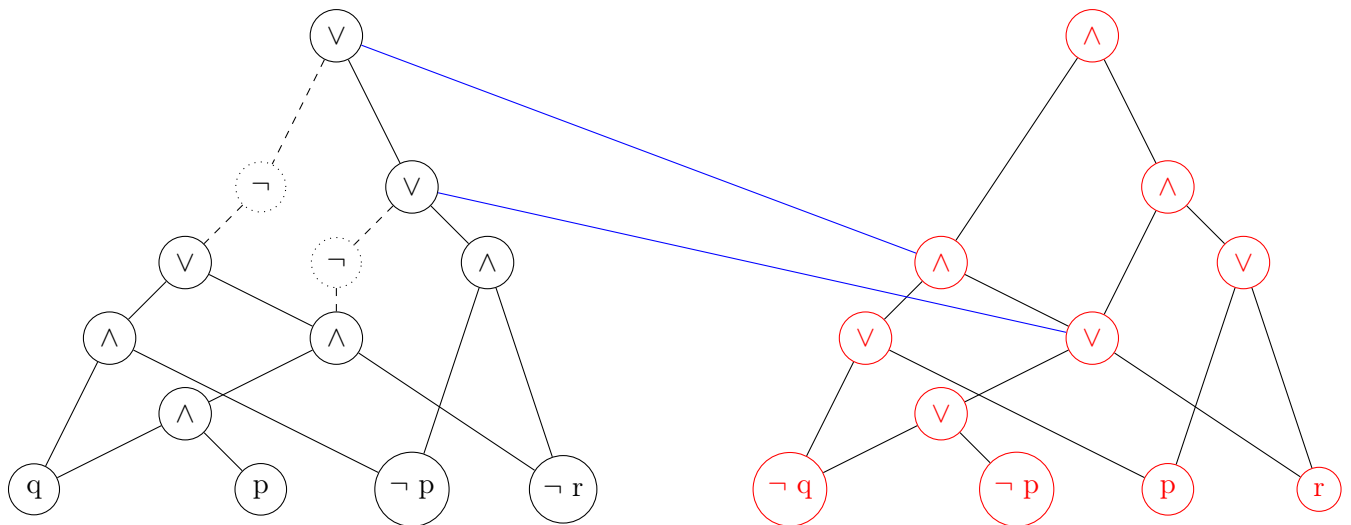
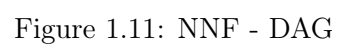
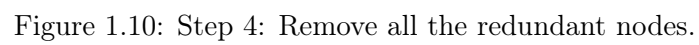


Figure 1.8: Step 2: Negate the entire DAG obtained in step 1

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



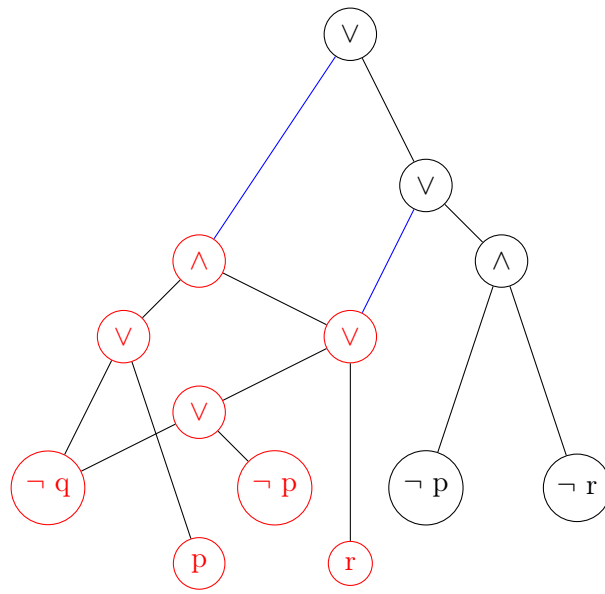


Figure 1.12: The NNF DAG (rearranged)

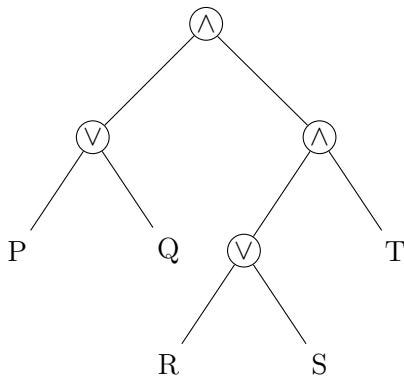
**NOTE :** The size of the NNF - DAG obtained using the above algorithm is atmost two times the size of the given DAG. Hence we have an  $O(N)$  formula for converting any arbitrary DAG to a semantically equivalent NNF - DAG.

## 1.11 Conjunctive Normal Forms

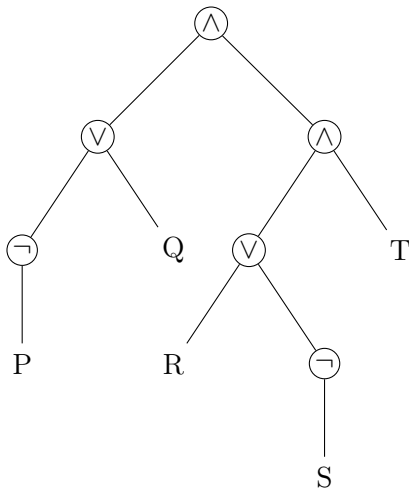
A formula is in conjunctive normal form or clausal normal form if it is a conjunction of one or more clauses, where a clause is a disjunction of variables.

**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 :  $p$  ,  $\neg p$  ,  $r$  ,  $\neg q$



- **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 :  $p \vee q \vee (\neg r)$  ,  $p \vee (\neg p) \vee (\neg r)$  -> 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  $p \wedge q \wedge (\neg r)$  ,  $p \wedge (\neg p) \wedge (\neg r)$  -> 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  $(p \vee q \vee (\neg r)) \wedge (q \vee r \vee (\neg s) \vee t)$

- **DISJUNCTIVE NORMAL FORM(DNF)**

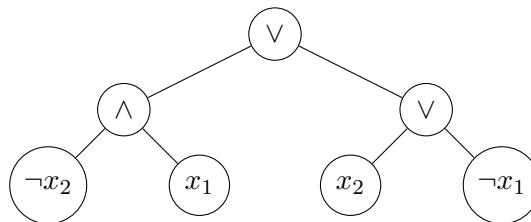
- A formula is said to be in Disjunctive Normal Form (DNF) if it is a disjunction of cubes.
- Sum of Products
- Example  $(p \wedge q \wedge (\neg r)) \vee (q \wedge r \wedge (\neg s) \wedge t)$

**Given a DAG of propositional logic formula with only  $\vee$  ,  $\wedge$  and  $\neg$  nodes , can we efficiently get a DAG representing a semantically equivalent CNF/DNF formula ?**

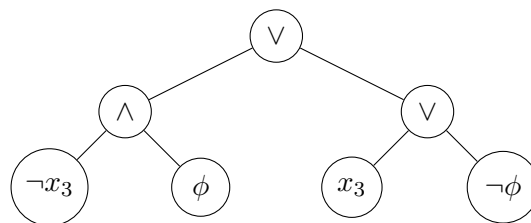
Tutorial 2 Question 2:

The Parity Function can be expressed as  $(( ( \dots (x_1 \oplus x_2 ) \oplus x_3 ) \dots \oplus x_n )$

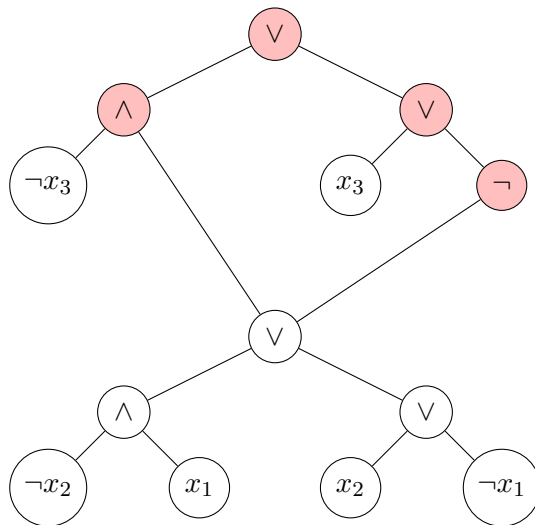
The Parse Tree for  $x_1 \oplus x_2$  is



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  $\rightarrow$  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  $p, \neg p$ . If **there is a clause which does not have both( $p, \neg p$ )**, 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  $p \vee q \vee (\neg r)$ , 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  $p \vee \neg p$  is valid and  $1 \vee \phi = 1$  for any  $\phi$ . 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  $p, \neg p$ . If there is a cube which does not have both( $p, \neg p$ ), 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 simultaneously

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

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

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

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

4. For checking the equisatisfiability of  $\phi$  and  $\phi'$ : Think about an assignment which makes  $\phi$  true then apply that assignment to  $\phi'$ .

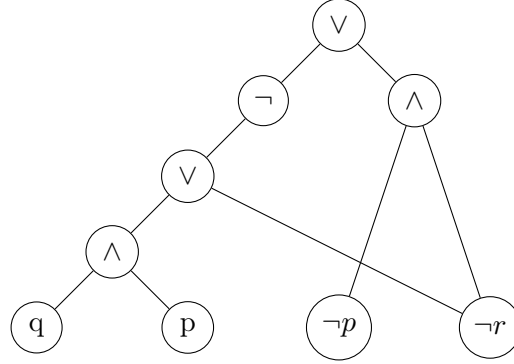
### 1.14 Tseitin Encoding

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 semantically 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 conjunction 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.

$$x_1 \wedge x_2 \implies x_3$$

$$x_4 \wedge x_3 \implies x_5$$

$$x_1 \wedge x_5 \implies \perp$$

$$x_5 \implies x_3$$

$$\top \implies x_5$$

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

---

```

1 Function HORN( $\phi$ ):
2   foreach occurrence of  $\top$  in  $\phi$  do
3     | mark the occurrence
4   while there is a conjunct  $P_1 \wedge P_2 \wedge \dots \wedge P_{k_i} \rightarrow P'$  in  $\phi$  do
5     | // such that all  $P_j$  are marked but  $P'$  isn't
6     | if all  $P_j$  are marked and  $P'$  isn't then
7       | | mark  $P'$ 
8   if  $\perp$  is marked then
9     | return 'unsatisfiable'
10  else
11    | 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

$$\begin{aligned}
 b \wedge n &\Rightarrow a & f \wedge d &\Rightarrow a & e \wedge d &\Rightarrow a \\
 e \wedge n &\Rightarrow a & p \wedge n &\Rightarrow a & d \wedge n &\Rightarrow \perp \\
 b \wedge f &\Rightarrow a & p \wedge d &\Rightarrow \perp
 \end{aligned}$$

Now we want to examine the possible behaviour of this system 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:

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

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

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 additional 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.  $\dots \wedge (\neg x_5) \wedge \dots$   
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( $\varphi$ , PA)

---

```

{ // These are the base cases for our recursion }
if  $\varphi = \top$  then
    return SAT(sat, PA)
else if  $\varphi = \perp$  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( $\varphi[l = 1]$ ,  $PA \cup \{l = 1\}$ ) { // 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( $\varphi[l = 1]$ ,  $PA \cup \{l = 1\}$ )
else
    { // This step is called Decision Step }
     $x \leftarrow \text{choose\_a\_var}(\varphi)$ 
     $v \leftarrow \text{choose\_a\_value}(\{0, 1\})$ 
    if SAT( $\varphi[x = v]$ ,  $PA \cup \{x = v\}$ ).status = sat then
        return SAT(sat,  $PA \cup \{x = v\}$ )
    else if SAT( $\varphi[x = 1 - v]$ ,  $PA \cup \{x = 1 - v\}$ ).status = sat then
        return SAT(sat,  $PA \cup \{x = 1 - v\}$ )
    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

$$(a \vee \neg b) \wedge (\neg a \vee b)$$

## 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-

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

Lets make two possible decision trees for these clauses.

PLE - Pure literal elimination      UP - Unit Propagation      D - Decision

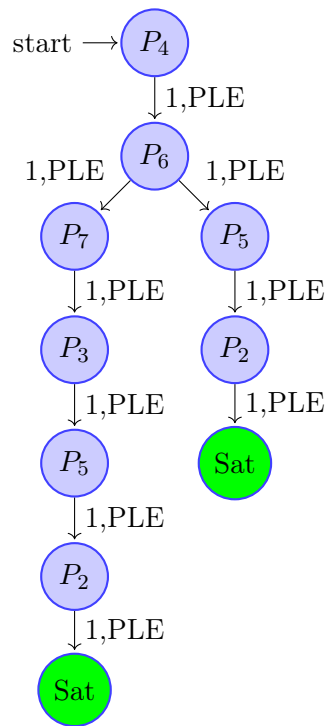


Figure 1.13: DPLL



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

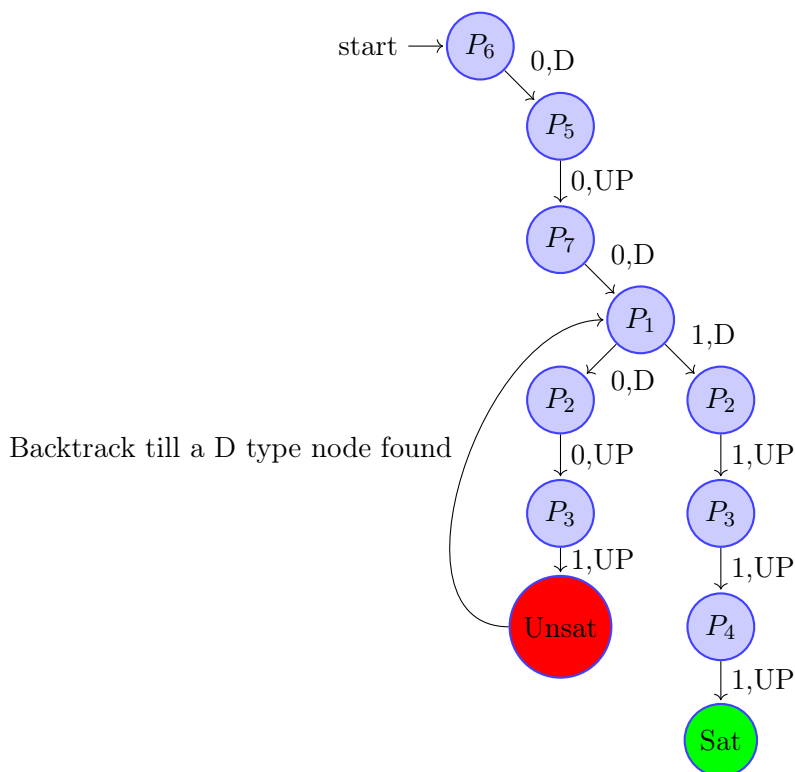


Figure 1.14: DPLL without point 2

## 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:

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

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:

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

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:

$$\frac{(\neg p_1 \vee p_2) \ (p_1 \vee \neg p_2)}{(\neg p_1 \vee p_1)} \text{ resolution}$$

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),

$$\frac{(\neg p_1 \vee \neg p_2) \ (\neg p_1 \vee p_1)}{(\neg p_2 \vee \neg p_1)} \text{ resolution}$$

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  $p_1, p_2 \dots p_m$  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 **pairs**, 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  $\{x_1, x_2, \dots, x_n\}$ ,  $\phi(x_1, x_2, \dots, x_n)$ . 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  $\phi$ .

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  $\Sigma$ . 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** ( $\cdot$ ) We can start from a string , take another string an as the name suggests concatenate them to form another string:

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

- **Identity Element** The algebra of the strings defined over the concatenation operator has the identity element :  $\varepsilon$  : **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  $\Sigma$ . 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  $\Sigma$  is countably infinite (cardinality:  $\mathbb{N}$ ), so the number of languages of  $\Sigma$  is uncountably infinite (cardinality:  $2^{\mathbb{N}}$ ).

- $\Sigma^*$  is defined to be the set of all finite strings on  $\Sigma$ , including  $\varepsilon$ . Note that  $\Sigma^* = \bigcup_{k \geq 0} \Sigma^k$ , where  $\Sigma^k$  is the set of all strings on  $\Sigma$  with exactly  $k$  letters. Note that we can prove that the number of strings for  $\Sigma^*$  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



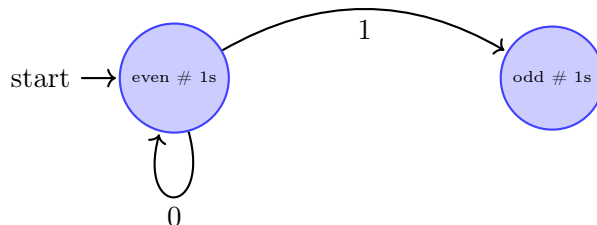
In our case we want to know whether there were even or odd number of 1s.



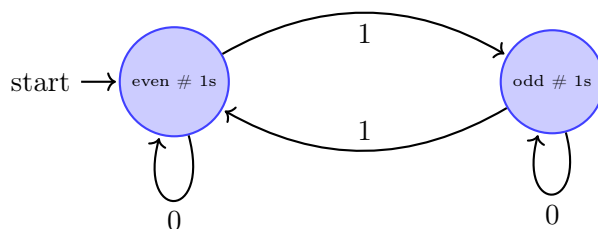
Where do we start from ? When I have seen nothing there are even number of 1s.



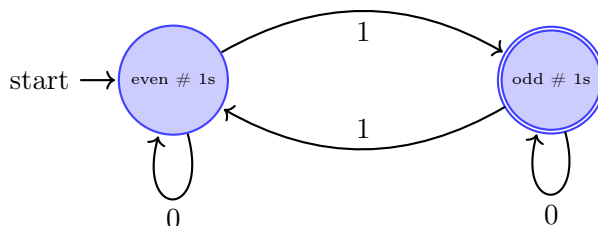
Now, suppose I receive a 0, I would remain in the same state, but if I get a 1, the parity changes.



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 our language: *is accepted*.

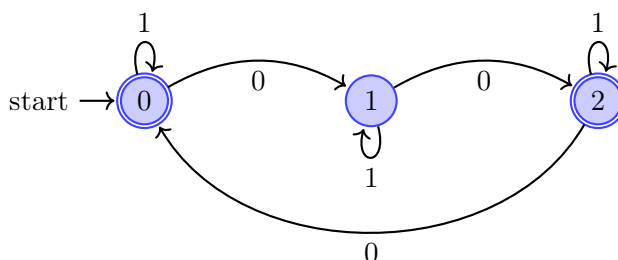


Such a formalism with **finite** states is known as **Finite Automata**.

Further, if for every string in  $\Sigma^*$ , 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  $L = \{w | w \in \Sigma^*, \text{ no. of 0's is a } 0 \bmod 3 \text{ or } 2 \bmod 3\}$



## 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  $\Sigma$  as the set of alphabets.

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

$$L = \{\omega \in \Sigma^* : n_a(\omega) \text{ is divisible by 2 and } n_b(\omega) \text{ is divisible by 3}\}$$

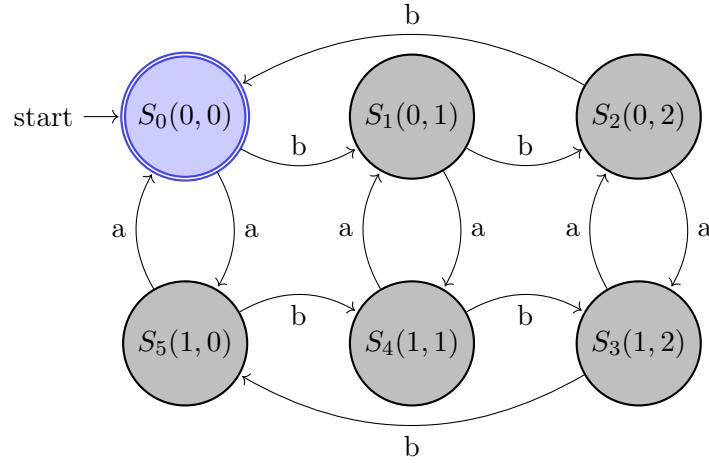


Figure 2.1: DFA for above task

$$S_0 : n_a(\mathbf{w}) \% 2 = 0 \text{ and } n_b(\mathbf{w}) \% 3 = 0$$

$$S_1 : n_a(\mathbf{w}) \% 2 = 0 \text{ and } n_b(\mathbf{w}) \% 3 = 1$$

$$S_2 : n_a(\mathbf{w}) \% 2 = 0 \text{ and } n_b(\mathbf{w}) \% 3 = 2$$

$$S_3 : n_a(\mathbf{w}) \% 2 = 1 \text{ and } n_b(\mathbf{w}) \% 3 = 2$$

$$S_4 : n_a(\mathbf{w}) \% 2 = 1 \text{ and } n_b(\mathbf{w}) \% 3 = 1$$

$$S_5 : n_a(\mathbf{w}) \% 2 = 1 \text{ and } n_b(\mathbf{w}) \% 3 = 0$$

## 2.4 DFA Design - Example 2

We will look at another example now.

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

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

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

If we try to cleverly convert the problem into a simpler one, we will observe that  $n_{ab}(\mathbf{w}) = n_{ba}(\mathbf{w})$  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:

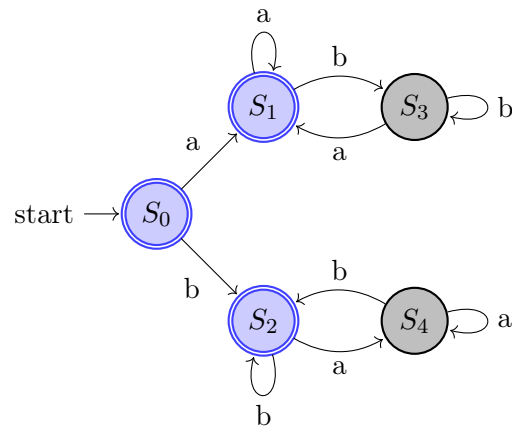


Figure 2.2: DFA: Last and First alphabets are same

## Chapter 3

# Non-Deterministic Finite Automata (NFA)

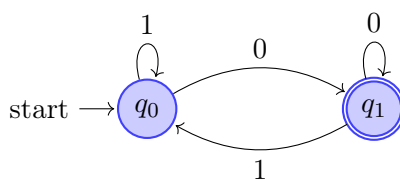
In the last few lectures, we covered the formalization of deterministic finite automata (DFA) where the transition function outputs a single state for a given input and current state. In this lecture, we will discuss non-deterministic finite automata (NFA) where the transition function can output multiple 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
- $\Sigma$  is the alphabet, a finite set of input symbols
- $q_0 \in Q$  is the initial state
- $\delta : Q \times \Sigma \rightarrow Q$  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:

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

where  $\delta$  is the transition function:

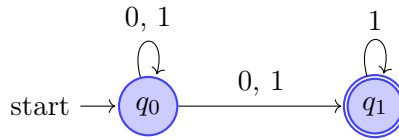
| Q     | $\Sigma$ | Q'    |
|-------|----------|-------|
| $q_0$ | 0        | $q_1$ |
| $q_0$ | 1        | $q_0$ |
| $q_1$ | 0        | $q_1$ |
| $q_1$ | 1        | $q_0$ |

### 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:

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

where  $\delta'$  is the transition function: As we can see,  $\delta'$  is a partial function whose output is a set of

| Q     | $\Sigma$ | $2^Q$          |
|-------|----------|----------------|
| $q_0$ | 0        | $\{q_0, q_1\}$ |
| $q_0$ | 1        | $\{q_0, q_1\}$ |
| $q_1$ | 1        | $\{q_1\}$      |

states instead of a single state.

### 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  $\Sigma$  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 | $\Sigma^*$ | 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 NFA gives choices at some states, it can still be represented as some equivalent DFA. Even though NFA 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 NFA is much more succinct than DFA.

We can convert a NFA 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  $(Q, \Sigma, Q_0, \delta, F)$ , construct a DFA  $(\mathcal{P}(Q), \Sigma, Q_0, \delta', F')$  with:

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

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

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.

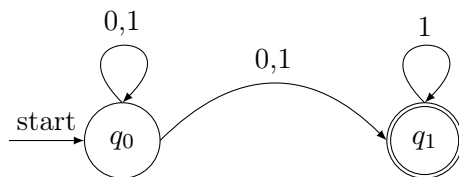
We claim that after the same characters are inputted into both the NFA and DFA, the state of the DFA is labelled the same as the set of current states of the NFA. This claim is easy to check using the definition of the  $\delta'$  function.

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( $\epsilon$ ). We represent this as:

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

that is,

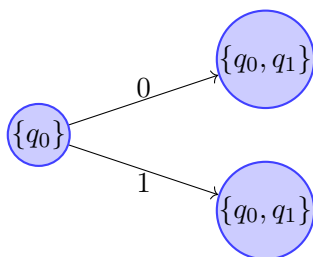
$$L(A) = \Sigma^* \setminus \{\epsilon\}$$

The transition function( $\delta$ ) table looks as follows:

| $Q$   | $\Sigma$ | $2^Q$          |
|-------|----------|----------------|
| $q_0$ | 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  $2^Q$ .

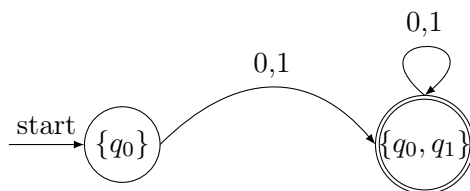
Step 1



Step 2



Final DFA



Call this DFA ‘ $A'$ ’.

One property we’ve extensively used in the conversion is:

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

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 NFA  $N$ , 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  $q \in Q$ <sup>1</sup> upon reading  $w$  if and only if DFA  $A'$  reaches state  $S \subseteq Q$  such that  $q \in S$ .

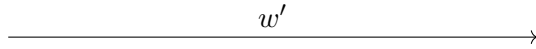
Given the condition that for all  $n \geq 0$ , and for every word  $w$ , where  $w \in \Sigma^*$  such that  $|w| = n$ , we aim to demonstrate the equivalence between the NFA  $A$ 's ability to reach state  $q \in Q$  upon reading word  $w$  and the DFA  $A'$ 's capability to reach state  $S \subseteq Q$ , 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 :



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

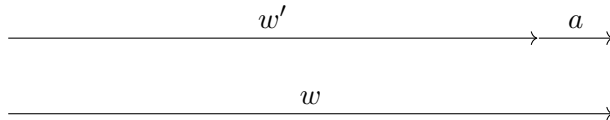
In NFA  $A$  :



In DFA  $A'$ <sup>2</sup>:



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



<sup>1</sup> $Q$  is the set of states in our original NFA  $A$ .

<sup>2</sup> $\dots q_0 \Rightarrow$  set containing initial states of the original NFA  $A$

$\dots q \Rightarrow \subseteq Q$  containing the state  $q$ , where  $Q$  is the set of states of the original NFA  $A$

$\dots \hat{q} \Rightarrow \subseteq Q$  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

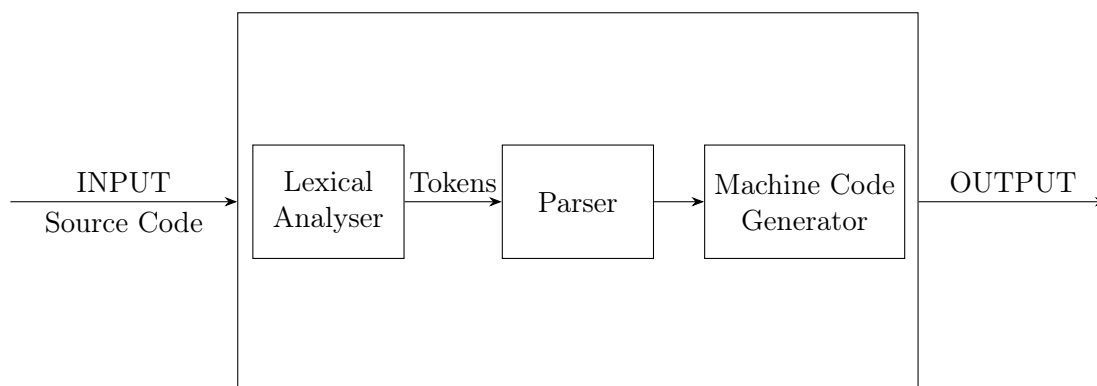


Figure 3.1: A Simple Compiler

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, it's 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 it's 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.

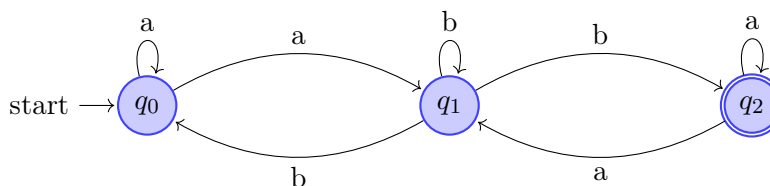
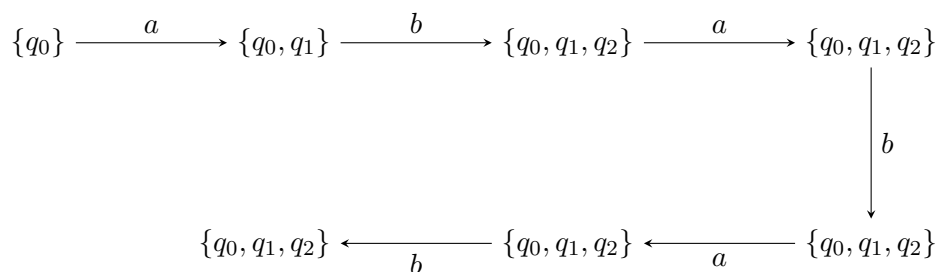


Figure 3.2: Lexical analyser Automaton

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

**Answer:**



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.<sup>3</sup>
- Here the GIVEN WORD is accepted by the GIVEN NFA<sup>4</sup>

<sup>3</sup>Size of automaton = No. of states in Automaton + No. of Transition Arrows in Automaton

<sup>4</sup> $q_2$  is contained in the last set

### 3.6 NFA with $\epsilon$ -edges

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

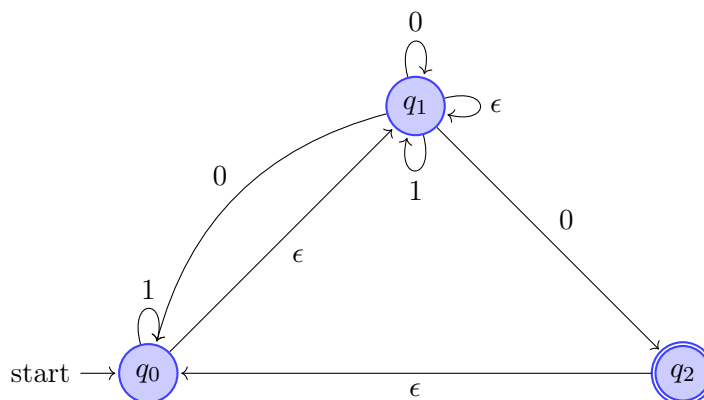


Figure 3.3: Automaton with Epsilon Edges

$\epsilon$ -edges bring non-determinism to the NFA, as you can sit on a node and take one of the  $\epsilon$ -edges possible from that node to jump to another node without consuming any letter of the input.

Figure 3.3 shows how  $\epsilon$  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  $\epsilon$  edge between  $q_0$  and  $q_1$ , but with the presence of this  $\epsilon$  edge  $10 \in L$ .

$\epsilon$ -edge also allows us to connect two automata. 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.

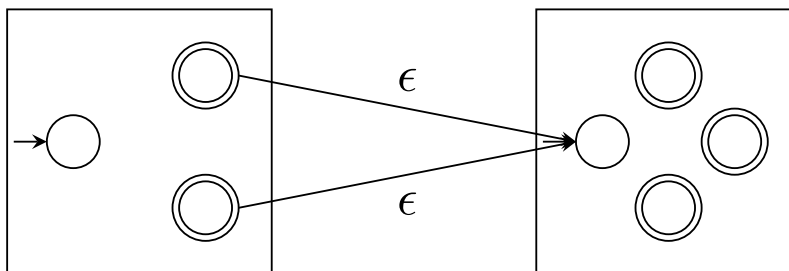


Figure 3.4:  $L_1$  accepting automaton and  $L_2$  accepting automaton connected

Now, we will try to find an equivalent DFA of this NFA having  $\epsilon$  edges. For that, we will first find an NFA without  $\epsilon$  edges which will preserve the original NFA, and this obtained NFA can then be constructed into a DFA.

Initially, just look at the  $\epsilon$ -edges only and find for each state its  $\epsilon$ -closure, which is the set of the states that we can reach from it by taking only  $\epsilon$ -edges.

From each node, we can go to every node that is present in the  $\epsilon$ -closure of that node for free. So wherever non- $\epsilon$  edges take us from these nodes in  $\epsilon$ -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

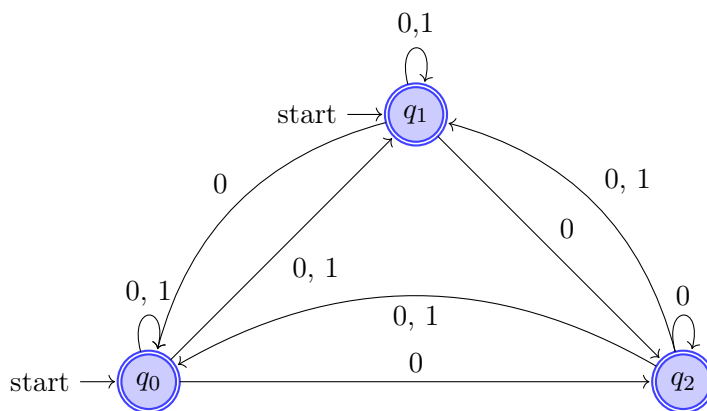


Figure 3.5: Automaton without Epsilon Edges

### 3.6.1 Equivalence with DFA

- If we can show that  $\epsilon$ -NFA have an equivalent NFA, then the equivalence of  $\epsilon$ -NFA and DFA is proved because every NFA has an equivalent DFA<sup>5</sup>.

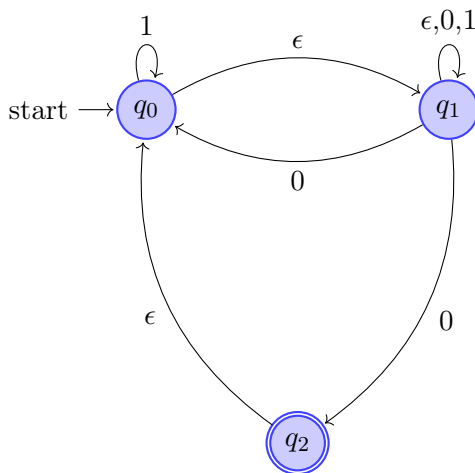


Figure 3.6: A Simple  $\epsilon$ -NFA

Focus on  $\epsilon$ -Edges of the Automaton to get the Epsilon Closure of the states in an Automaton

For Figure-3.6 Automaton:

---

<sup>5</sup>equivalence of NFA and DFA is already proved

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

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

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

Rules for converting  $\epsilon$ -NFA to equivalent NFA<sup>6</sup>:

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  $\epsilon$ -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:

---

<sup>6</sup>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  $\epsilon$ -NFA.

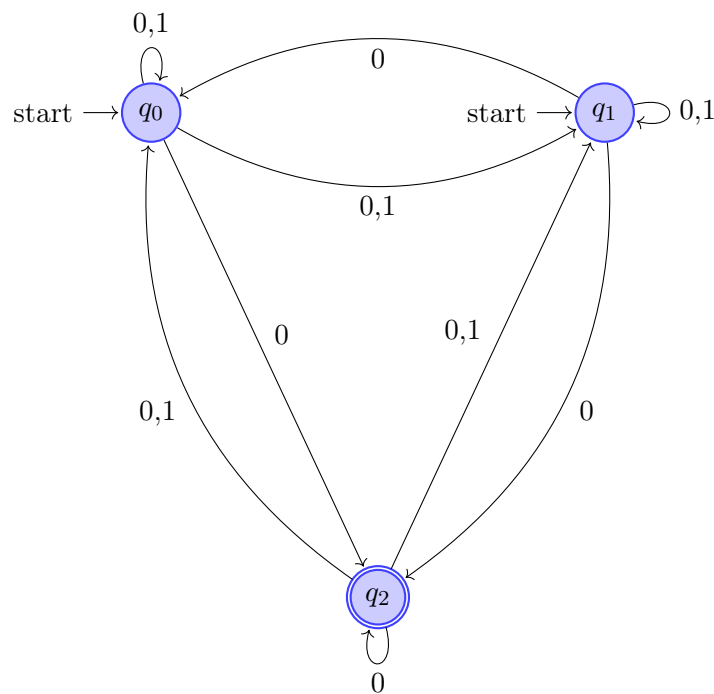


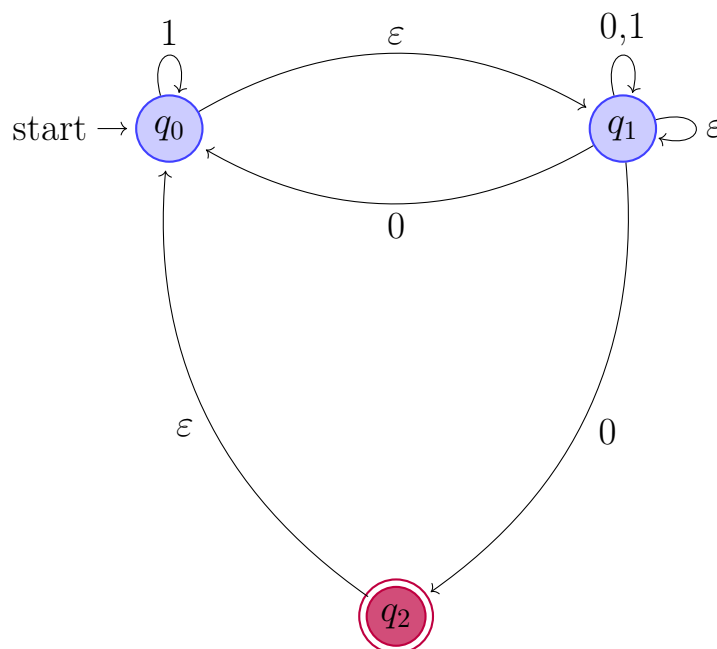
Figure 3.7: Equivalent NFA for Figure-3.6

## 3.7 Recap

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

### 3.7.1 Converting $\varepsilon$ -NFA to non- $\varepsilon$ -NFA

- **$\varepsilon$ -closure** of a node is defined as set of those nodes which can be reached from that node by traversing over  $\varepsilon$ -edges, i.e. without consuming any character from the alphabet  $\Sigma$ . Also, the node itself is trivially a part of its  $\varepsilon$ -closure.
- **$\varepsilon$ -edges** are those edges which can be traversed without consuming any character from the alphabet  $\Sigma$ , i.e. by consuming an empty string. Observe that the string "10"  $\in L$  with  $\varepsilon$ -edges but without  $\varepsilon$ -edges, string "10"  $\notin L$  where  $L$  is the Language of the NFA.

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

For example in this NFA,

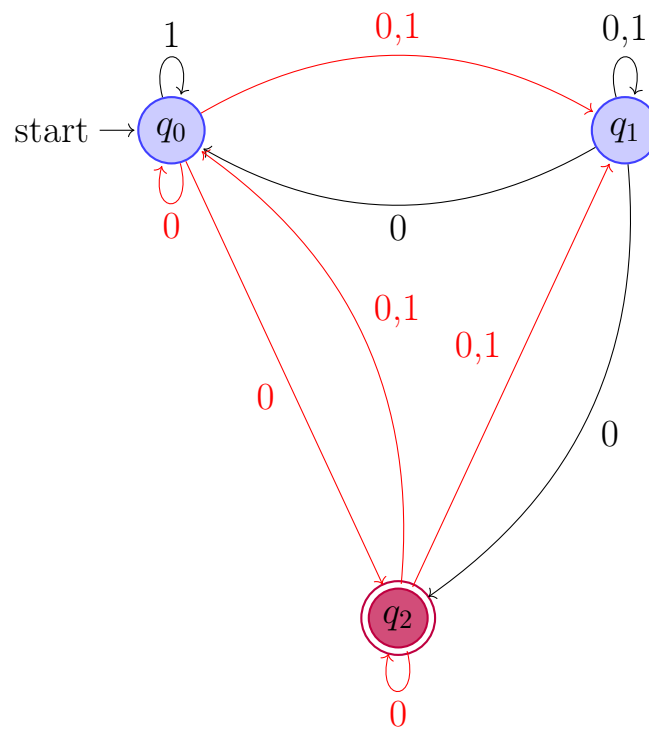
$$\varepsilon\text{-closure}(q_0) = \{q_0, q_1\}$$

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

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

Now for the algorithm to convert an NFA with  $\varepsilon$ -edges to an equivalent NFA without  $\varepsilon$ -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  $\varepsilon$ -closure (say  $S$ ) and mark all its non- $\varepsilon$ -edges as they are. Now, for each node  $q' \neq q$  in  $S$ , mark all non- $\varepsilon$ -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  $\varepsilon$ -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  $\varepsilon$ -closures contain atleast one of the accepting states of the  $\varepsilon$ -NFA. For eg., here only  $q_2$  has the accepting state  $q_2$  in its  $\varepsilon$ -closure, so only  $q_2$  is marked as accepting.
3. Starting states in the new non- $\varepsilon$ -automaton will be the same as the starting states in the original  $\varepsilon$ -automaton.

Figure 3.9: Equivalent non- $\varepsilon$ -NFA

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

- One may ask why the states which are in the  $\varepsilon$ -closures of the accepting states of the original automaton are not marked as accepting in the non- $\varepsilon$ -NFA. The answer is :- every string which reaches an accepting state  $a$  can also reach any of the states in  $\varepsilon\text{-closure}(a)$  by taking  $\varepsilon$ -edges and those strings are indeed in the language of the  $\varepsilon$ -NFA. But there are many such strings also which can reach at the states in the  $\varepsilon\text{-closure}(a)$  which are not in the language of the  $\varepsilon$ -NFA. So, if we mark those states as accepting, we are changing the language which is not our intention. For eg.,  $q_0 \in \varepsilon\text{-closure}(q_2)$  but if we mark  $q_0$  as accepting in the non- $\varepsilon$ -NFA, then the string "1" will also be included in the language of the non- $\varepsilon$ -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  $\varepsilon$ '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- $\varepsilon$ -automaton should exactly be equal to the language of the original  $\varepsilon$ -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  $\varepsilon$ -closures. So,  $\varepsilon\text{-closure}(s)$  definitely contains an accepting state in the original automaton. So, we have a path  $\varepsilon^k$  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  $s'_0 \in \varepsilon\text{-closure}(s_0)$  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  $\varepsilon\text{-closure}(s_m)$ . 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  $\varepsilon\text{-closure}(s)$  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  $\varepsilon$ -closures. So, the starting state  $s$  is an accepting state in the new automaton and thus  $w \in L(\text{new})$ .

(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  $\varepsilon^{k_i}$  from any state  $x$  means going to some state  $y \in \varepsilon\text{-closure}(x)$ . Here,  $s_1$  is the state reached after reading  $c_1$ ,  $s_2$  after reading  $c_2 \dots s_m$  after reading  $c_m$  and  $s_{m+1}$  after taking  $\varepsilon^{k_m}$ . 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  $\varepsilon\text{-closure}(s_m)$  contains accepting state  $s_{m+1}$  in the original automaton. Thus,  $w \in L(\text{new})$ .

Hence, proved.

Thus, we've shown that  $L(\text{new}) = L(\text{ori})$

### 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 automata using the step 1 of the algorithm. Also, if the path in the original automaton had trailing  $\varepsilon$ 's we weren't able to reach the same state in the new automaton but had to make the languages of the two automata exactly same so we concluded step 2 of the algorithm.

### 3.7.4 Extras

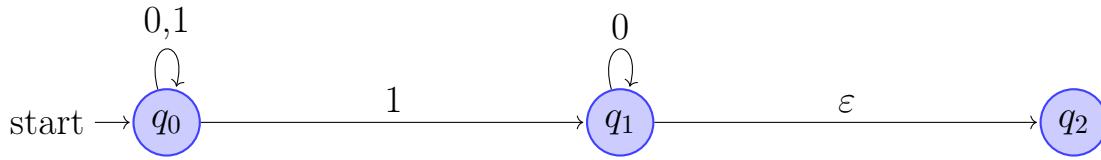
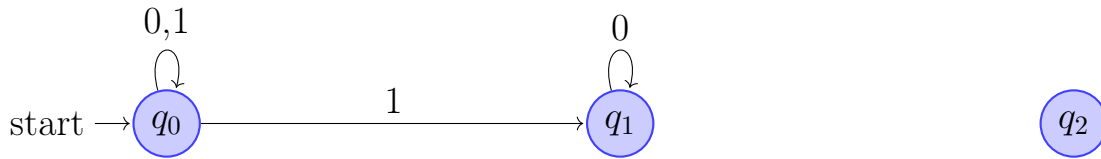
- **Language of a node  $q$**  is defined as the set of all those strings  $w \in \Sigma^*$  such that upon reading  $w$  character by character we can reach  $q$  from any of the starting states of the automaton.
- Note that in  $\varepsilon$ -automaton we could also take  $\varepsilon$ -edges in between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

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

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

- A lemma:  $L(q_i)_{\text{original}} \supseteq L(q_i)_{\text{new}}$  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 automata. If  $w \in L(q_i)_{new}$  and  $w$  is non-empty, then we can have a path with some  $\varepsilon$ '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  $\supseteq$  sign comes in is the case when the path of  $w$  has some trailing  $\varepsilon$ 's. For eg. consider this  $\varepsilon$ -NFA and its equivalent non- $\varepsilon$ -NFA,

Figure 3.10:  $\varepsilon$ -NFAFigure 3.11: Equivalent Non- $\varepsilon$ -NFA

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

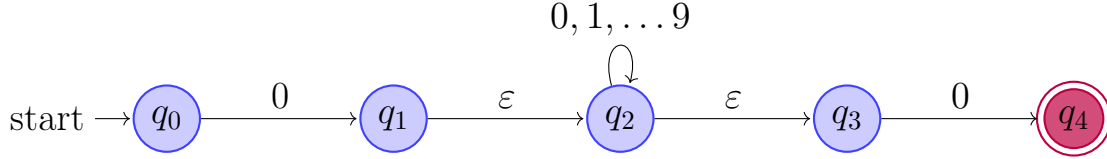
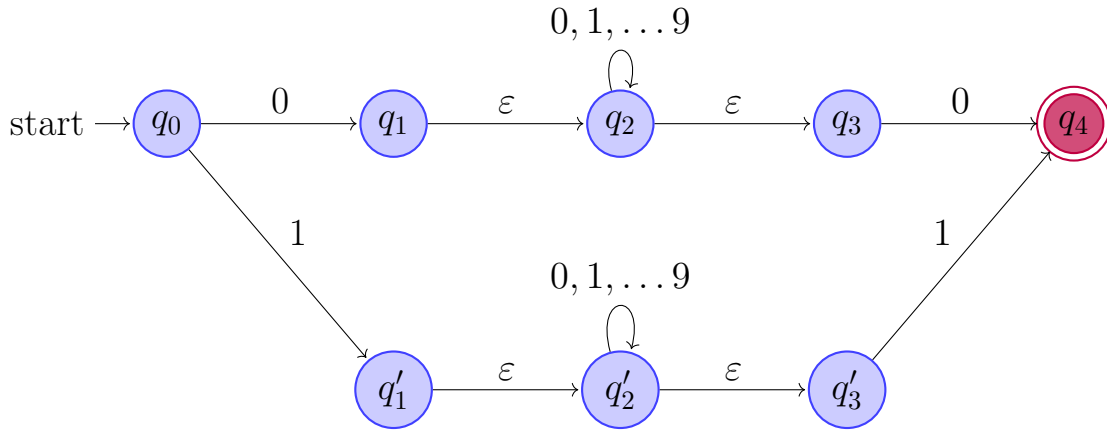
### 3.7.5 Significance of $\varepsilon$ -edges

$\varepsilon$ -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  $\varepsilon$ -edges. For eg., suppose we have to check  $w = u.v$  such that  $u$  contains an even number of 1's and  $v$  contains  $1 \bmod 3$  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  $\varepsilon$ -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,  $\varepsilon$ -edges allowed us to capture the non-determinism of the breaking point between  $u$  &  $v$ .

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

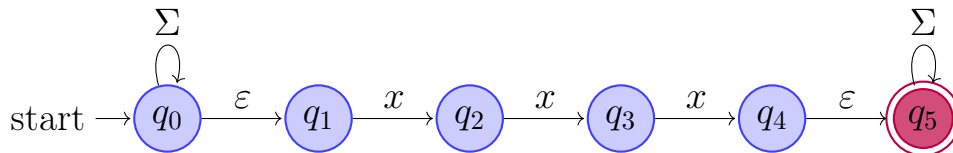
### 3.7.6 Examples

- $L = \{x \in \Sigma^* | x = u.v.w, v \in \Sigma^*, u, w \in \Sigma^+, |u| \leq 2, u=w\}$  where  $\Sigma = \{0, 1, 2 \dots 9\}$   
 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:  $\varepsilon$ -NFA for  $u = w = 0$ Figure 3.13:  $\varepsilon$ -NFA for  $u = w = 0, 1$ 

One for each  $u = w = 0, 1, 2 \dots 9$  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  $\Sigma = \{a, b, \dots z\}$   
 Here, we want to locate "xxx" in the text so the following NFA captures this language:-

Figure 3.14:  $\varepsilon$ -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 ( $\varepsilon$ -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.

$\varepsilon$ -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 Regular Expression

### 3.9.1 Introduction

In arithmetic, we can use the operations  $+$  and  $\times$  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  $\times$  symbol in algebra, the concatenation symbol  $\circ$  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.

### 3.9.2 Formal Definition of a Regular Expression

Let  $R$  be a regular expression if:

1.  $a$  for some  $a$  in the alphabet  $\Sigma$ ,
2.  $\varepsilon$ ,
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  $\varepsilon$  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.

### 3.9.3 Semantics of Regular Language

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

#### 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.

#### 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  $[e_2]$ .

#### 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  $[e_2]$ .

#### 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  $\{ab, a\}$ .

#### Order of Precedence

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

#### 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  $[e_1]$ .

#### 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  $\varepsilon$ ,  $a$ ,  $b$ ,  $ab$ ,  $ba$ ,  $aa$ ,  $bb$ , and so on.

**Further Examples**

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