Computational Theory
LEGENDS

The English language consist of three different entities: letters, words, sentences. Group of letters make up words and group of words make up sentences but not any collection of letters form a valid word, and not any collection of words form a valid sentence.

This situation also exist with computer language, certain character string are recognizable words (GOTO, END, ...), and certain strings of words are recognizable commands, and certain sets of commands become a program.

To construct a general theory that unifies all these examples, it is necessary for us to adopt a definition of a “most universal language structure” that is a structure in which the decision of whether a given string of units constitutes a valid larger unit is based on explicitly stated rules.

As a first step to defining a general theory of abstract language, it is right for us to insist on precise rules. We call our study “Theory of Formal Language”, the word “Formal” used here emphasizes that it is the form of the string of symbols we are interested in, not the meaning.

We begin with only one finite set of fundamental units out of which we build structures, We shall call this the “Alphabet”. A certain specified set of strings of character from the alphabet will be called the “Language”, these strings that are permissible in the language we call “Words”, the symbols in the alphabet do not have to be Latin letters.

We shall call “Null string” to the string have no letters and denote it by a symbol “Λ”, and denote the alphabet by Greek letter capital sigma “Σ”.

\[ Σ = \{a, b, c, d, e, f, \ldots, z, ', -\} \]

We can now specify which string of these letters are valid words in our language by listing them all, as is done in dictionary, it is long list but finite list, and it makes a perfectly good definition of the language, we call this language ENGLISH-WORDS and we may write :
ENGLISH-LANGUAGE = \{ all the words (main entries) in a standard dictionary \}

If we wish to make formal definition of the language of the sentence in English, we must begin by saying that this time our basic alphabet is the entries in the dictionary, let us call this alphabet “\( \Gamma \)”, the capital gamma:

\[ \Gamma = \{ \text{the entries in a standard dictionary, plus a blank space, plus the usual punctuation marks} \} \]

In order to specify which string of elements from \( \Gamma \) produce valid word in the language ENGLISH-SENTENCE, we must rely on the grammatical rules of English, this is because we could never produce a complete list of all possible words in this language, that would have to be a list of all valid English sentences.

If we go by the rules of grammar only, many strings of alphabet letter seem to be valid words, for example “I ate three Tuesdays “, in a formal language we must allow this string, it is grammatically correct, but the meaning is silly, in other words we interested in syntax but not semantics.

The abstract language we treat will be defined in one of two, Either they will be presented as an alphabet and the complete list of all valid words, or else they will be presented as an alphabet and a set of rules defining the acceptable words.

The set of rules can be of two kinds : Either tell us how to test a string of alphabet letters that we might be presented with, or they can tell us how to construct all the words in the language by some clear procedure.

Let us consider some simple example of languages, if we start with an alphabet having only one letter :

\[ \Sigma = \{ x \} \]

we can define a language by saying that any nonempty string of alphabet character is a word :

\[ L_1 = \{ x \ xx \ xxx \ xxxx \ldots \} \]

or to write this in an alternate form :

\[ L_1 = \{ x^n \text{ for } n = 1 \ 2 \ 3 \ldots \} \]
In this language as in any other, we can define the operation of concatenation, in which two strings are written down side by side to form a new longer string, for example if we concatenate the word xxx with the word xx we obtain the word xxxxxxx, concatenation operation analo
gues to addition operation :-

\[ X^n \text{ concatenated with } x^m \text{ is the word } x^{n+m} \]

It is not always true that when two words are concatenated they produce another word in the language, for example

\[ L_2 = \{ x \ .xxx \ xxxxx \ xxxxxxx \ . . . \} \]
\[ = \{x^{\text{odd}}\} \]
\[ = \{x^{2n+1} \text{ for } n = 0 \ 1 \ 2 \ 3 \ . . . \} \]

Then a = xxx and b = xxxxx are both words in L₂, but their concatenated ad = xxxxxxxx is not in L₂.

In these examples, when we concatenate a with b we get the same word as when we concatenate b with a, we can show this by writing :

\[ ab = ba \]

but this relationship does not hold for all languages, in English when we concatenate “house” and “boat” we get “houseboat” which is indeed a word but distinct from “boathouse”, which different thing-not because they have different meanings but because they are different words.

Example :- Consider the language, let us begin with the alphabet :

\[ \Sigma = \{0 \ 1 \ 2 \ 3 \ 4 \ 5 \ 6 \ 7 \ 8 \ 9\} \]

And define the set of words :

\[ L_3 = \{\text{any finite string of alphabet letters that does not start with the letter zero }\} \]
The language $L_3$ then look like the set of all positive integers written in base of 10.

$$L_3 = \{1\ 2\ 3\ 4\ 5\ 6\ 7\ 8\ 9\ 10\ 11\ 12\ \ldots\}$$

We say “look like” instead of “is” because $L_3$ is only a formal collection of strings of symbols, and the integers have other mathematical properties, to define language $L_3$ so that include the string (word) zero, we could say:

$$L_3 = \{\text{any finite string of alphabet letters that, if it starts with a 0, has no more letters after the first}\}$$

**DEFINITION:**

We define the function “length of a string” to be number of letters in the string, for example:

$$a = xxxx \quad \text{Length}(a) = 4$$

$$\text{Length}(xxxx) = 4$$

We can now present yet another definition of $L_3$

$$L_3 = \{\text{any finite string of alphabet letters that, if it has length more than one, does not start with a zero}\}$$

This is not necessarily a better definition of $L_3$, but it does there are often different ways of specifying the same language.

There is some ambiguity in the expression “any finite string”, since it is not clear whether we intend to include the null string, to avoid this ambiguity, we shall always be more careful, the language $L_3$ does not include $\Lambda$ string, since we intended that the language should look like the integers, we wish to define a language like $L_3$ that contain $\Lambda$:

$$L_3 = \{\Lambda\ x\ xx\ xxx\ xxxx\ \ldots\}$$

$$L_3 = \{x^n \text{ for } n = 0\ 1\ 2\ 3\ 4\ \ldots\}$$

$x^0 = \Lambda$, not $x^0=1$ as mathematics.

$\Lambda$ is a word in language and its length is one, not a letter in alphabet.
**Definition :-**

Function reverse, if a is a word in some language L, the reverse(a) is the same string of letters spelled backward, called the reverse of a, even if this backward string is not a word in L.

Example :-

Reverse(143) = 341

**Definition :-**

Let us define a new language called PALINDROME over the alphabet

\[ \Sigma = \{a, b\} \]

PALIDROME = \{Λ, and all strings x such that reverse (x) = x\}

If we begin listing the elements in PALIDROME we find

PALIDROME = \{Λ, a, b, aa, bb, aaa, aba, bab, bbb, aaaa, abba, . . . \}

**Definition :-**

Given an alphabet \( \Sigma \), we wish to define a language in which any string of letters from \( \Sigma \) is a word, even the null string. This language we shall call the closure of the alphabet, and it is denoted by writing a star after the name of the alphabet :-

\[ \Sigma^* \]

And also called kleene star.

Example :-

If \( \Sigma = \{x\} \) then

\[ \Sigma^* = L4 = \{\Lambda x xx xxx . . . \} \]
Example :-

If $\Sigma = \{0, 1\}$, then

$$\Sigma^* = \{\Lambda 0 1 00 11 01 10 111 000 001 \ldots \}$$

We can think of the kleene star as an operation that makes an infinite language of string of letters out of an alphabet, when we say “infinite language” we mean infinitely many words each of finite length.

We write the words in the language in order size (words of shortest length first) and then listed all the words of the same length alphabetically.

We shall now generalize the use of the star operator to set of words, nor just sets of alphabet letters.

Definition :-

If $S$ is a set of words, then by $S^*$ we mean the set of all finite strings formed by concatenating words from $S$, where any word may be used as often as we like, and where the null string is also include.

Example :-

If $S = \{aa, b\}$

$S^* = \{\Lambda \text{ plus any word composed of factors of } aa \text{ and } b \}$

$= \{\Lambda \text{ plus all string } s \text{ of } a's \text{ and } b's \text{ in which the } a's \text{ occur in even clumps } \}$

$= \{\Lambda b aa bb aab baa bbb aaaa aabb baab bbbaa bbbb aaab aabaa \ldots \}$

The string aabaaab is not in $S^*$ since it has a clump of $a's$ of length 3.

Example :-

Let $S = \{a, ab\}$

$S^* = \{\Lambda \text{ plus any word composed of factor’s of } a \text{ and } ab \}$

$= \{ \Lambda a aa ab aaa aa aaaa aaab aaba abaa \ldots \}$
To prove that a certain is in the closure language $S^*$, we must show how it can be written as a concatenation of words from the base set $S$, from the last example, we can factoring the word abaab into its factors:

$$(ab)(a)(ab)$$

These three factors are all in the set $S$, therefore their concatenation is in $S^*$. This is the only way to factor the string into factors of $(a)$ and $(ab)$, when this happens, we say that the factoring is unique, Sometimes the factoring is not unique.

Example: consider $S = \{xx, xxx\}$, then:

$$S^* = \{ \Lambda \text{ and all strings of more than one } x \}$$

$$= \{x^n \text{ for } n = 0, 2, 3, 4, \ldots \}$$

$$= \{\Lambda xx xxx xxxx xxxxx \ldots \}$$

The word $x$ is not in the language $S^*$, the string xxxxxxx is in this closure for any of these reasons, it is:

$$(xx) (xxx) (xx) \text{ or } (xxx) (xx) (xx) \text{ or } (xx) (xx) (xxx)$$

It is important to note here that the parentheses $( )$ are not letters in the alphabet but are used for the sole purpose of demarcating the ends of factors, in case where parentheses are letters of the alphabet:

$$\Sigma = \{ x ( ) \}$$

Length of $(xxxxx) = 5$

Length of $( (xx) (xxx) ) = 9$

The alphabet has no letters, then its closure is the language with the null string as its only word symbolically we write:

If $\Sigma = \Phi$ (the empty set)

Then $\Sigma^* = \{ \Lambda \}$

This is not the same as:

If $S = \{ \Lambda \}$

Then $S^* = \{ \Lambda \}$
If we wish to modify the concept of closure to refer to only the concatenation of some (non-zero) string from a set \( S \), we use notation \( ^+ \) instead of \( ^* \), for example:

IF \( \Sigma = \{ X \} \)

Then \( \Sigma^+ = \{ x, xx, xxx, \ldots \} \)

If \( S = \{ xx, xxx \} \) then \( S^+ \) is the same as \( S^* \) except for the word \( \Lambda \), which is not in \( S^+ \), this is not to say that \( S^+ \) cannot in general contain the word \( \Lambda \). It can, but if \( S \) contains the word \( \Lambda \), in this case \( \Lambda \) is in \( S^+ \).

Example :-

\( S = \{ a, b, \Lambda \} \)

\( S^+ = \{ a, b, \Lambda, ab, ba, \ldots \} \)

What happens if we apply the closure operator twice? We start with a set of words \( S \) and look at its closure \( S^* \), and start with the set \( S^* \) and try to form its closure, which denote as:

\[
(S^*)^* = S^{**}
\]

If \( S \) is not empty, then \( S^* \) is finite, so we taking the closure of the infinite set.

**THEOREM 1** :- For any set \( S \) of strings we have \( S^* = S^{**} \)

**PROOF** :- Every word in \( S^{**} \) is made up of factors from \( S^* \), and every factor in \( S^* \) is made up of factors of \( S \), therefore, every word in \( S^{**} \) is also word in \( S^* \) so we can say:

\[
S^{**} \subseteq S^*
\]

Where the symbol "\( \subseteq \)", which means "is contained in or equal".

In \( S^{**} \) we can chose a word any one factors of \( S^* \), so:

\[
S^* \subseteq S^{**}
\]

\( S^* = \{ a \}, \quad S^{**} = \{ a, aa, aaa, \ldots \} \)

Together, these two inclusions prove that:

\[
S^* = S^{**}
\]
CHAPTER 2

Recursive Definition

Recursive Definition :- Typically a three step process :-

1- Specify some basic objects in the set.

2- Give rules for constructing more objects in the set from the ones we already know.

3- Declare that no objects except these constructed in this way are allowed in the set.

Example :-

Suppose that we trying to define the set of positive even integer, one standard way of defining this set is :-

EVEN is the set of all positive whole numbers divisible by 2

Another way we might try is this :-

EVEN is the set of all $2^n$ where $n= 1 2 3 4 \ldots$

The third method we present is tricky, by recursive definition :-

The set EVEN is defined by these three rules

Rule1 :- 2 is in EVEN

Rule2 :- if $x$ is in EVEN, then so is $x + 2$

Rule3 :- The only elements in the set EVEN are those that can be produced from the two rules above.

Suppose we want to prove that 14 is in the set EVEN. To show this using the first definition, we divide 14 by 2 and find that there is in reminder. Therefore it is in EVEN.
To prove that 14 is in $\text{EVEN}$ by the second definition, we have to somehow come up with the number 7 and then. Sense $14 = (2)(4)$ we know that it is in $\text{EVEN}$.

To prove that 14 is in $\text{EVEN}$ using the recursive definition is a lengthier process. We could proceed as follow :-

By rule 1, we know that 2 is in $\text{EVEN}$.

By the rule 2, we know that $2+2= 4$ is also in $\text{EVEN}$.

Again by rule 2, we know that since 4 has just been shown in $\text{EVEN}$ $4+2= 6$ is also in $\text{EVEN}$.

The fact that 6 is in $\text{EVEN}$ means that when we apply rule 2 we deduce that $6+2=8$ is in $\text{EVEN}$, too.

Now applying rule 2 to 8, we drive that $8+2=10$ is another number of $\text{EVEN}$.

Once more applying rule 2, this time to 10, we infer that $10+2=12$ is in $\text{EVEN}$.

And at last by applying rule 2, yet again, to the number 12, we conclude that $12+2=14$ is indeed in $\text{EVEN}$.

This is not the only recursive definition of the set $\text{EVEN}$, we might use :-

The set $\text{EVEN}$ is defined by these two rules :-

Rule 1 :- 2 is in $\text{EVEN}$

Rule 2 :- if $x$ and $y$ are both in $\text{EVEN}$, then so is :- $x + y$

Example :-

We can prove that 14 is in $\text{EVEN}$ in fewer steps :-

By rule 1 :- 2 in $\text{EVEN}$

By rule 2 :- $x=2$, $y=2$ --- $4$ is in $\text{EVEN}$.

By rule 2 :- $x=2$, $y=4$ --- $6$ is in $\text{EVEN}$

By rule 2 :- $x=4$, $y=4$ --- $8$ is in $\text{EVEN}$.

By rule 2 :- $x=6$, $y=8$ --- $14$ is in $\text{EVEN}$
This is better recursive definition of the set EVEN because it produce shorter proofs that elements are in EVEN.

Example :-

The following example is a recursive definition of positive integer :-

Rule 1 :- 1 is in INTEGERS

Rule 2 :- if x is in INTEGERS, then so is x+1.

If we want to include both positive and negative integer in the INTEGERS definition, we might use the following recursive definition:-

Rule 1 :- 1 is in INTEGERS.

Rule 2 :- if both x and y are in INTEGERS, then so are x+y and x-y.

Since 1-1=0, 0-1=-1 , 0 and -1 are include in the set INTEGERS.

Example :-

If we want recursive definition for all the positive real numbers, we could try a definition of the form :-

Rule 1 :- x is in POSITIVE.

Rule 2 :- if x and y are in POSITIVE, then so are x+y and xy.

But the problem is that there is no smallest positive real number on which to build the rest of the set, we could try :-

Rule 1 :- if x is in INTEGERS,"." Is a decimal point, and y is any finite string of digits. Even one that starts with some zero, then x.y is in POSITIVE.

But this definition has two problems, one, it doesn’t generate all real numbers ( it is not included of its infinite length ), two, the definition is not recursive.

Example :-

Consider the recursive definition of set POLYNOMIAL

Rule 1 :- any number is in POLYNOMIAL.
Rule 2 :- the variable x is in POLYNOMIAL.

Rule 3 :- if p and q in POLYNOMIAL, then so are p+q, p-q, (p), pq.

we have \(3x^2 + 7x - 9\) then :-

Rule 1 :- 3 is in POLYNOMIAL.

Rule 2 :- x is in POLYNOMIAL.

Rule 3 :- (3) (x) is in POLYNOMIAL: call it 3X.

Rule 3 :- (3x) (x) is in POLYNOMIAL :call it 3x^2.

Rule 1 :- 7 is in POLYNOMIAL.

Rule 3 :- (7) (x) is in POLYNOMIAL.

Rule 3 :- 3x^2 + 7X is in POLYNOMIAL.

Rule 1 :- -9 is in POLYNOMIAL.

Rule 3 :- 3x^2 + 7x + (-9) = 3x^2 + 7x - 9 is in POLYNOMIAL.

In fact there are several other sequence that could also produce the result.

Example :-

Observe how natural the following definition are :-

Rule 1 :- x is in L_1.

Rule 2 :- if w is any word in L_1, then xw is also in L_1.

\[L_1 = x^+ = \{x \ xx \ xxx \ldots \}\]

Or

Rule 1 :- \(\Lambda\) is in L_2.

Rule 2 :- if w is any word in L_2, then xw is also in L_2.

\[L_2 = x^* = (\Lambda \ x \ xx \ xxx \ldots )\]

Or

Rule 1 :- x is in L_2.
Rule 2 :- if $w$ is any word in $L_2$, then $xxw$ is also in $L_2$.

$L_2 = \{x^{\text{odd}}\} = \{ x \ xxx \ xxxxx \ldots \}$

An Important Language : Arithmetic Expressions (AE) :-

What form of a valid arithmetic expression that can be in one line, in a form digestible by computer, the alphabet for this language is :-

$\Sigma = \{0 1 2 3 4 5 6 7 8 9 = \ast / ( )\}$

But, the following strings (forbidden) are not allowed :-

$$( 3 + 5 ) + 6) \quad 2/(8 + 9 \quad (3 + (4 - ) 8 \quad 2) – (4$$

The most natural way of defining a valid arithmetic expression, is by using recursive definition rather than a long list of forbidden substring, the definition can be written as :-

Rule 1 :- Any number (positive, negative or zero) is in AE.

Rule 2 :- if $x$ is in AE, then so are :-

i. $(x)$

ii. $-x$

Rule 3 :- if $x$ and $y$ are in AE, then so are :-

i- $x + y$

ii- $x - y$

iii- $x * y$

iv- $x / y$

v- $x ** y$

we have call this the “most natural” definition because, it is the method we use for recognizing arithmetic expression in real life.
This definition gives us the possibility of writing $2 + 3 + 4$ which is not ambiguity, but it also gives as $2 / 3 / 4$, which is ambiguous, because it can be $2/(3 / 4)$ or $(2/3)/4$, by applying Rule 2 we can avoid this ambiguous by put parentheses.

This definition determine, the set AE in a manner useful for proving many theorems about arithmetic expression.

**Theorem 2-1**

an arithmetic expression cannot contain the character S

**Proof**

i- this character is not part of any number, so it cannot be introduce into an AE by Rule 1.

ii- If the character string x does not contain the character S, then neither x nor $-x$, so it cannot be introduced into AE by Rule 2.

iii- If neither x nor y contain the character S, then neither do any of the expression defined by Rule 3.

The character S can never get into an AE.

**Theorem 2-2**

No AE can begin or end with symbol /.

**Proof**

i- No number begin or end with this symbol, so it cannot occur in Rule 1.

ii- Any AE formed by Rule 2 must begin and end with parentheses or begin with a mines sign, so the symbol / cannot be introduce in Rule 2.

iii- By Rule 3, no clause begin or end with the symbol /.

Therefore, these rules will never introduce an expression beginning or ending with a /.
Theorem 2-3

No AE can contain the substring /.

Proof

Let us suppose that there were some AEs that contained the substring /, Let a shortest of these be a string called w.

We know that w like all words in AE, is formed by some sequence of applications of Rules 1, 2 and 3, the question is, which Rule is produce the substring w? the answer is Rule 3(iv), because, if it is Rule 3(iii), then the / must either be found in x or y part, but x and presumed to be on AE, so this mean that there is some shorter word in AE than w that contains the substring /, which contracts the assumption that w is the shortest. similarly we can eliminate all other possibilities, therefore, the last Rule used to produce w must have been 3(iv).

Now, since the / cannot have been contributed to w from the x part alone or from the y part alone, it must have been included by finding an x part that ended in a / or a y part that begin with a /.

But since both x and y are AEs, our previous theorem says neither case can happen, therefore, even Rule 3(iv) cannot introduce the substring /.

Therefore, there is no possibilities left for the last rule from which w can be constructed. Therefore, w cannot be in the set AE.

Another common use for recursive definition is to determine what expressions are valid in symbolic logic. We shall be interested in one particular branch of symbolic logic called sentential calculus or propositional calculus.

The version we shall define here uses only negation \( \neg \) and implication \( \rightarrow \) along with the phrase variables, although conjunction and disjunction could easily be added to the system.

The valid expressions in this language are called WFFs for Well-Formed Formulas.

There are other symbols sometimes used for negation \( \neg, \overline{\neg} \), and \( \sim \).

The rules for forming WFFs are:

Rule 1 :- any single Latin letter is a WFF.
Rule 2: if \( p \) is a WFF, then so are \( p \) and \( \neg p \).

Rule 3: if \( p \) and \( q \) are WFF, then so is \( p \rightarrow q \).

Some sequences of applications of these rules enable us to show that:

\[ p \rightarrow ((p \rightarrow p) \rightarrow q) \]

is a WFF.

and we can also show that:

\[ p \rightarrow \rightarrow p \rightarrow (p \rightarrow p) \rightarrow (p \rightarrow (p) \rightarrow (p \rightarrow (p))) \]

are all not WFFs.
CHAPTER 3

Regular Expression

Let as consider this language :-

\[ L = \{ \Lambda x \ xx \ xxx \ldots \} \]

From previous chapters, we presented one method for indicating this set as closure of a smaller set :-

\[ S = \{ x \} , \quad L = S^* \]

We could have written :-

\[ L = ( x )^* \]

Also we can apply the star notation to the letter instead to the set :-

\[ x^* \]

The star operator applied to letter is analogous to the star applied to a set, it represent an arbitrary concatenation of copies of that letter ( may be non at all ), since \( x^* \) is any string of x’s, \( L \) is then the set of all possible strings of x’s of any length ( including \( \Lambda \)).

Suppose that we wished to describe the language \( L \) over the alphabet :-

\[ \Sigma = \{ a \ b \} \]

Where :-

\[ L_1 = \{ a \ ab \ abb \ abbb \ldots \} \]

We could summarize the language by the English phrase “ all words of the form one a followed by some number of b’s ( may be no b )”, using star notation, we may wrote :-

\[ L_1 = \{ a \ b^* \} \quad \text{or} \quad L_1 = \{ ab^* \} \]

We can apply the kleen star to the whole word (ab) :-
\((ab)^* = \Lambda \text{ or } ab \text{ or } abab \text{ or } ababab \ldots\)

Parentheses are not letters in the alphabet of this language, so they can be used to indicate factoring without accidentally changing the words. Since the star represents some kind of exponentiation, we use it as powers are use in algebra.

The language of the expression :-

\[ a^*b^* \]

is :-

\[ L_2 = \{ \Lambda \ a \ b \ aa \ ab \ bb \ aaa \ aab \ abb \ bbb \ aaaa \ldots \} \]

So it is not equal to \((ab)^* :-\)

\[ a^*b^* \neq (ab)^* \]

Since the language defined by the expression on the right contains the word (abab), where the language defined by the expression on the left doesn’t. this caution as against thinking of the \(^*\) as a normal algebraic exponent.

We introduce another use for the plus sign, By the expression \((x + y)\) where x and y are strings of characters from an alphabet, we mean “either x or y“ this means that \((x + y)\) offers a choice, care should be taken so as not to confuse this with \(^*\) as an exponent.

Example :-

Consider the language \(T\) defined over the alphabet \(\Sigma = \{ a \ b \ c \}\)

\[ T = \{ a \ c \ ab \ cb \ abb \ cbb \ abbb \ cbbb \ldots \} \]

All the words in \(T\) begin with an \(a\) or a \(c\) and then followed by some numbers of \(b\)’s, symbolically, we may write this as :-

\[ T = \text{language } ((a+c)b^*) \]

\[ = \text{language } \text{ (either a or c then some or no b’s) } \]

Example :-

Let us consider a finite language \(L\) that contains all the strings of \(a\)’s and \(b\)’s of length “three exactly” :-
\[ L = \{ \text{aaa aab aba abb baa bab bba bbb}\} \]

The first letter of each word in the language \( L \) is either an \( a \) or a \( b \), so we write :-

\[ L = \text{language} \left( (a+b)(a+b)(a+b)(a+b)(a+b)(a+b)(a+b)(a+b) \right) \]

Or for short :-

\[ L = \text{language}((a+b)^* ) \]

If we wish to define the set of all seven-letter string of a’s and b’s, we could write \((a+b)^7\), in general, if we wish to refer to the set of all possible strings of a’s and b’s of any length whatsoever, we could write :-

\[(a+b)^*\]

This is the set of all possible strings of letters from the alphabet \( \Sigma = \{a\ b\} \) including null string.

We can define the set of word that begin with a letter a and followed any choice of an a or a b :=

\[ a(a+b)^*\]

And all the words that begins with an a and followed any choice of an a b followed by a b :-

\[ a(a+b)^*b\]

**Formal Definition of Regular Expression :-**

**Definition :-**

The set of regular expression is defined by the following rules :-

**Rule 1 :-** Every letter of \( \Sigma \) can be made into a regular expression by writing it in build face, \( \Lambda \) itself is a regular expression.

**Rule 2 :-** if \( r_1 \) and \( r_2 \) are regular expression, then so are :-

- \( (r_1) \)
- \( r_1r_2 \)
• \( r_1 + r_2 \)
• \( r_1^* \)

Rule 3: nothing else is a regular expression.

The language that can be defined by the regular expression is Regular Language, it is a language of language-definers, it is analogous to a book of the list of books, every word in the book is a book definer, the same confusion occur in every day speech, the word “French” is both a word (objective) and a language defining name.

The definition we have given for regular expression contains one subtle but important omission: the language phi (\( \Phi \)), which is the regular expression for the Null language, so we have:

\[
 r + \Phi = r
\]

and

\[
 r\Phi = \Phi
\]

Example:

Let us consider the language defined by the expression:

\[
 (a+b)^* a(a+b)^*
\]

It stands for anything, that is any string of a’s and b’s a, and come an a, and another anything, it represent the language is the set of all words over the alphabet \( \Sigma = \{a, b\} \), the only words left out are those that have only b’s and the word \( \Lambda \).

The words below are derived from this expression, which represent the word abbaab:

\[
 (\Lambda)a(bbaab) \quad \text{or} \quad (abb)a(ab) \quad \text{or} \quad (abba)a(b)
\]

The only words left out of the language defined by this expression is the word without \( \Lambda \) and strings of b’s, if we combine this language with language contains b’s, we produce the language of all strings, in the English phrase:

All words = (all strings with an a) + (all string without an a)
It should make sense to write:

\[(a+b)^* = (a+b)^*a(a+b)^* + b^*\]

Here, we have added two language-defining expression to produce an expression that define the **union** of the two language defined by the individual expressions, Notice that this use of the plus sign is far from the normal meaning of addition in the algebraic.

Example:-

The language of all words that have at least two a’s can be described by the expression:

\[(a+b)^*a(a+b)^*a(a+b)^*\]

= (some beginning)and first a(some in the middle)and second a(some at the end)

Example :-

Another expression that denote all words with at least two a’s :-

\[b^*ab^*a(a+b)^*\]

(some or no of b’s)and first a(some or no of b’s)and second a(anything)

In this set are abbbabb and aaaaa, we can write :-

\[(a+b)^*a(a+b)^*a(a+b)^* = b^*ab^*a(a+b)^*\]

Where equal sign we do not mean that these expressions are equal algebraically in the same way:-

\[X + X = 2X\]

But that they are equal because they describe the same item, and could write :-

The language \(((a+b)^*a(a+b)^*a(a+b)^*) = \) the language\((b^*ab^*a(a+b)^*) = \) all words with 

at least two a’s

We say that two expressions are equivalent, if they describe the same language.

Example :-
The language of all words that have at least one a and at least one b is somewhat trickier, if we write:

\[(a+b)^*a(a+b)^*b(a+b)^*\]

We are requiring that an a precede a b in the word, such words as ba, bbaaa are not included in this set, however, we know that either the a come before b or the b come before the a, we could define this set by the expression:

\[(a+b)^*a(a+b)^*b(a+b)^* + (a+b)^*b(a+b)^*a(a+b)^*\]

Here, we are still using the plus sign in the general sense of disjunction (or), we are taking the union of two sets, but it is more correct to think of this + as offering alternative in forming words.

There is a simpler expression that define the same language of all words over the alphabet \(\Sigma = \{a, b\}\) that contain both an a and b is therefore also defined by the expression:

\[(a+b)^*a(a+b)^*b(a+b)^* + bb^*aa^*\]

We have shown that:

\[(a+b)^*a(a+b)^*b(a+b)^* + (a+b)^*b(a+b)^*a(a+b)^* = (a+b)^*a(a+b)^*b(a+b)^* + bb^*aa^*\]

Example:

The only words that do not contain both an a and a b in them somewhere are the words of all a’s, all b’s, or \(\Lambda\), when these included we get everything. Therefore, the regular expression:

\[(a+b)^*a(a+b)^*b(a+b)^* + bb^*aa^* + a^* + b^*\]

Defines all possible strings of a’s and b’s, the word \(\Lambda\) is included in both a^* and b^*,

We can then write:

\[(a+b)^* = (a+b)^*a(a+b)^*b(a+b)^* + bb^*aa^* + a^* + b^*\]

Which is not a very clear equivalence at all, we must not misinterpret the fact that every regular expression defines some language to mean the associated language has a simple English description such as in the preceding examples.
Example :-

All temptation to treat these language-defining expression as if they were algebraic polynomials should be dispelled by these equivalences :-

\[(a+b)^* = (a+b)^* + (a+b)^*\]
\[(a+b)^* = (a+b)^* + a^*\]
\[(a+b)^* = (a+b)^*(a+b)^*\]
\[(a+b)^* = a(a+b)^*+b(a+b)^*+\Lambda\]
\[(a+b)^* = (a+b)^* ab(a+b)^*+b^* a^*\]

The last equivalence is mean that all the words that do not contain substring ab are all a’s, all b’s, Λ, or some b’s followed by some a’s, all four missing types are covered by b^* a^*.

we employ a star operator to defining infinite language, and we can represent finite language by using plus sign (union sign) alone, the language L over the alphabet Σ={a b} contains only finite list of words :-

\[L = \{abba \ baaa \ bbbb\}\]

We can represent L by the symbolic expression :-

\[L = \text{language (abba+baaa+bbbb)}\]

The language that include the null word Λ, then the expression that defines L must also employ the symbol Λ, for example :-:-

\[L = \{\Lambda \ a \ aa \ bbb\}\]

The symbolic expression for the language L is :-
L = language (Λ + a + aa + bbb)

Example:-
Let language

\[ V = \{Λ, a, b, ab, bb, abbb, bbbb, abbb, bbbb, \ldots \} \]

We can define \( V \) be the expression

\[ b^* + ab^* \]

Or

\[ (Λ + a)b^* \]

We factored \( b^* \) just as in algebra, it is because of this analogy to algebra that we have denoted out disjunction by the plus sign instead of the union sign \( U \) or the symbolic logic sign \( V \), we like it to look algebraic, but it is not :-

\[ ab = ba \] in algebra, they are the same numerical product

\[ ab \neq ba \] in formal language, they are different words

**Definition :-**

if \( S \) and \( T \) are sets of strings of letters (whether they are finite or infinite sets), we define the product set of strings of letters to be :-

\[ ST = \{ \text{all combination of a string from } S \text{ concatenated with a string from } T \text{ in that order} \} \]

Example :-

If

\[ S = \{a, aa, aaa\} \quad T = \{bb, bbb\} \]

Then
ST = \{ abb \ aabbb \ aabbb \ aaabb \ aaabbb \ aaabbb \\}

Example :-

If

\( P = \{ a \ bb \ bab \} \quad Q = \{ \Lambda \ bbbb \} \)

Then

\( PQ = \{ a \ bb \ bab \ aabbb \ bbbbbbb \ bbbbbbb \} \)

**Language Associated with Regular Expression :-**

We are now ready to give the rules for associating a language with every Regular Expression.

**Definition :-**

The following rules define the language associated with any regular expression :-

**Rule 1 :-** the language associated with the regular expression that is just a single letter is that one-letter word alone and the language associated with \( \Lambda \) is just \( \{ \Lambda \} \), a one-word language.

**Rule 2 :-** if \( r_1 \) is a regular expression associated with the language \( L_1 \) and \( r_2 \) is a regular expression associated with the language \( L_2 \), then :-

i- the regular expression \( (r_1)(r_2) \) is associated with the product \( L_1L_2 \) that is the language \( L_1 \) times \( L_2 \) :-

\[ \text{language}(r_1r_2) = L_1L_2 \]

ii- the regular expression \( r_1 + r_2 \) is associated with the language formed by the union of the sets \( L_1 \) and \( L_2 \) :-

\[ \text{language}(r_1 + r_2) = L_1 + L_2 \]
the language associated with the regular expression \((r_1)^*\) is \(L_1^*\), the Kleene closure of the set \(L_1\) as a set of words:

\[
\text{language}(r_1^*) = L_1^*
\]

this collect of rules proves recursively that there is some language associated with every regular expression, as we build up a regular expression from the rules, we simultaneously are building up the corresponding language.

**Finite Languages are Regular :-**

**Theorem :-**

If \(L\) is a finite language (have finitely many words), then \(L\) can be defined by a regular expression. In other words, all finite languages are regular.

**Proof :-**

To make one regular expression that defines the language \(L\), turn all the words in \(L\) into bold face type and insert plus sign between them.

Example :-

The regular expression that define the language

\[
L = \{\text{baa abbbba bababa}\}
\]

Is

\[
\text{baa + abbbba + bababa}
\]

Example :-

The regular expression that define the language

\[
L = \{\text{aa ab ba bb}\}
\]

Is
\[ aa + ab + ba + bb \]

another regular expression that define this language is :-

\[ (a + b)(a + b) \]

So the regular expression need not to be unique, but so what, we need only show at least one regular expression exist.

**How Hard it is to Understand a Regular Expression :-**

Example :-

Consider the regular expression bellow :-

\[ L = (a+b)^*a(a+b)^*(a+\Lambda)(a+b)^*a(a+b)^* \]

All the words in the language \( L \) must have at least two a’s on them, let us break up the middle plus sign into two cases, Either the middle factor contribute an \( a \) or else it contribute a \( \Lambda \), therefore :-

\[ L = (a+b)^*a(a+b)^*a(a+b)^*a(a+b)^* + (a+b)^*a(a+b)^*\Lambda(a+b)^*a(a+b)^* \]

The first term represent all words that have at least three a’s at them, before analyzing the second term, let us make the study that :-

\[ (a+b)^*\Lambda(a+b)^* \]

This term could be replaced by the term :-

\[ (a+b)^* \]

This would reduce the second term of the expression to :-

\[ (a+b)^*a(a+b)^* a(a+b)^* \]

This regular expression representing all words that have at least two a’s in them.

Therefore the language associated with \( L \) is the union of all strings that have three or more a’s with all strings that have two or more a’s, but since all strings with three or more a’s are themselves already strings with two or more a’s, this whole language is just the second set only.
It is possible by repeated application of the rules for forming regular expression to produce an expression in which the star operator is applied to a sub-expression that already has a star in it:

\[(a + b)^* = (a + b)^*\]

Example:

Consider the regular expression:

\[(a^*b^*)^*\]

The language defined by this regular expression is all strings that can be made up of factors of the form \(a^*b^*\), but since both the single letter \(a\) and the single letter \(b\) are words of the form \(a^*b^*\), this language contains all strings of \(a\)'s and \(b\)'s, so:

\[(a^*b^*)^* = (a + b)^*\]

CHAPTER 4

Finite automata
Introduction: To understand “finite Automata”, let us discuss the following example of a child game, pieces are set up on a playing board, dice are thrown, and a number is generated at random, depending on the number, the pieces on the board must be rearranged in a fashion completely specified by the rules, the child has no options about changing the board, everything determined by the rules. Usually, it is often some other child’s turn to throw the dice and make his move, but this hardly matters, because no skill or choice is involved, we could eliminate the opponent and have the one child move first the white pieces and then the black, whether or not white piece win the game is dependent entirely on what sequence of number is generated by the dice, not on who moves them.

Let us look at all possible positions of the pieces on the board and call them state, the game change from one state to another in a fashion determined by the input of a certain number.

For each possible number, there is one only and only one resulting state, we should allow for the possibility that after a number is entered, the game is still in the same state as it was before (the player in “jail” needs to roll double in order to get out, any other roll leaves the board in the same state).

After certain number of roll’s, the board arrives at a state that means a victory for one of the players and the game is over, we call this a final state, there might be many possible final states that result in victory for this player.

We can take another example, a child has a simple computer and wishes to calculate the sum of 3 and 4, the child writes the program, which is a sequence of instructions that are fed into the machine one at a time, each instruction executed as soon as it is read, if all goes well, the machine output number 7 and terminates execution.

We can consider the computer example similar to the board game, here the board is the computer and the different arrangement of pieces on the board correspond to the different arrangement of 1’s and 0’s in cell’s of memory, and the set of all possible dice rolls to be the letter of an alphabet, we can define a certain languages as the set of string of those letters that lead to victory, this example is instance of the general model which called “Finite Automaton”.

“Finite” because the number of possible states and number of letters in the alphabet are both finite, “Automaton” because the change of states is totally...
governed by the input, “Automaton” comes from Greek, so its correct plural is “Automata”.

Finite Automata also called “Finite Acceptor” because its sole job is to accept certain input strings and reject others.

**Definition :-**

A **finite automaton** is a collection of three things :-

1. A finite set of states, one of which is designated as the initial state, called start state, and some of which are designated as final states.
2. An alphabet Σ of possible input letters.
3. A finite set of “transitions” that tell for each state and for each letter of the input alphabet which state to go to next.

This definition is not complete, because it describes what a finite automaton but not how it works. It works by being presented with an input string of letters that it reads letter by letter at the leftmost letter. Beginning at the start state, the letter determine a sequence of states, the sequence ends when the last input letter has been read.

Example:-

Suppose that the input letter has only the two letters a and b, and we have three states x, y, and z. Let the following be rules of transmission :-

Rule 1 : From state x and input a, go to state y.
Rule 2 : From state x and input b, go to state z.
Rule 3 : From state y and input a, go to state x.
Rule 4 : From state y and input b, go to state z.
Rule 5 : From state z and any input, stay at state z.

Let x as a start state, and z as a final state.
We now have a perfectly defined finite automaton, because it fulfills all three requirements demand above states, alphabet, and transition.

Example :-

Let us exam the string “aaa” to this FA, the start state is x :-

- The first input letter of the string is a which tell us go to the state y by Rule 1.
- The second input letter of the string is a which tell us go back to the state x by Rule 3.
- The third input letter of the string is also a which again tell us go to the state y by Rule 1.

We didn’t finish up in the final state z, so we ha e unsuccessfully terminate of our run, the string “aaa” is not in the language of all strings that leave this FA in state z, the set of strings that do leave us in a final state is called “the language defined by the finite automata”.

Also we can say that the string “aaa” is :-

- Not accepted by this FA.
- Rejected by this FA.

If the set of all strings of the language are accepted by FA, we can say that :-

- The language associated with FA.
- The language accepted by this FA.
- This FA accepts this language.

If $L_1$ is contained in $L_2$ and a certain FA accepts $L_2$, then this FA accept all strings in $L_1$, However we do not say “$L_1$ accepted by this FA”, because this would mean that all the words the FA accepts are in $L_1$.

At the moment, the only job of FA does is define the language it accepts, which is a fine reason for calling it an “accepter”, or better a “language recognizer” the last term is good because the FA merely recognizes whether the input string is in its language much the same way we might recognize when we hear someone speak Russian without necessarily understand what it means.
Example:

Let us exam the input string “abba”, in the same FA:

- Start state is x.
- First input is a, by Rule 1, go to state y.
- Second input is b, by Rule 4, go to state z.
- Third input is b, by Rule 5, stay in the state z.
- Fourth input is a, by Rule 5, stay in the state z.

The input string “abba” has taken as successfully to the final state z, so, the string “abba” therefore a word in the language associated with this FA, the word “abba” is accepted by this FA.

We can now predict which strings are accepted by this FA, If the input string made up from a’s only, then we cannot move to the final state z, if the string have a letter b then we can move to the final state z, this FA will accept all strings that have the letter b in them no other strings, therefore, the language associated with this FA is the one defined by the regular expression:

\[(a+b)^*b(a+b)^*\]

The list of transition rules can be growth, it is much simpler to summarize them in a table format, each row of the table is the name of one of the state in the FA, and each column of the table is the letter of the input alphabet, the entries inside the table are the new state that the FA moves into the transition states, the transition table for the FA we have describe is:

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
</tr>
</thead>
<tbody>
<tr>
<td>Start</td>
<td>x</td>
<td>y</td>
</tr>
<tr>
<td></td>
<td>y</td>
<td>x</td>
</tr>
<tr>
<td>Final</td>
<td>z</td>
<td>z</td>
</tr>
</tbody>
</table>
We can define FA in mathematical abstract using function called *transition function* instead of transition table, where the input is a pair of state and the alphabet letter and the output is a single state, the abstract definition of an FA is then :-

1- A finite set of states $Q= \{q_0, q_1, q_2, \ldots \}$ of which $q_0$ is the start state.

2- A subset of $Q$ called the final states.

3- An alphabet $\Sigma=\{x_1, x_2, x_3, \ldots \}$.

4- A transition function $\delta$ (lowercase Greek delta) associating each pair of state and letter with a state :-

$$\delta(q_i, x_j) = x_j$$

- we can represent each state by circle.
- arrow is represent to the state which the input letter lead us to it, we can label the arrow with the corresponding alphabet letter.
- The arrow start form the state and back to the same state is called *loop*.
- Start state can be marked by *minus* sign inside or outside the circle, and the final state marked by *plus* sign.
- Start state can be labeled by *Start* word, and the final state can be labeled by *final* word.
- Start state is indicated by *arrow*, and the final state by *circle* or *Box*.

Every input string is interpret as traversing a path beginning from start state and moving among states, if the last state is a final state, then the path has ended in success. The input letters are the fuel needed for motion and dictate the direction of travel, when we are out of letters, we must stop.
Example :-

\[
\begin{align*}
\text{Start} & \quad x \quad y \\
\quad z & \quad a \\
\quad b \\
\quad a, b \\
\quad - & \quad + \\
\quad b &
\end{align*}
\]
The language accepted by this machine is the set of all strings except $\Lambda$, the regular expression for this machine is:

$$(a+b)(a+b)^* = (a+b)^*$$

Example:

One of the many FA’s that accept all words is:

![Diagram](image)

The sign $\pm$ means the start state and final state, the expression for this machine is:

$$(a+b)^*$$

**FA’s and there Languages** :-

We can look at the world of FA’s in two ways:-

- We could start with the machine and try to analyze it to see what language it accept.
- We could start with a desired language and try to construct an FA that would act as a language-recognizer or language-definer.

Example:

Let us build a machine that accept the language of all words over the alphabet $\{a, b\}$ with an even number of letters, we must define a Boolean flag called $E$, if the number of letters read is even then the value of $E$ is true, otherwise it will be false, let the initial value of $E$ is true, every time we read a letter, we reverse the value of $E$, when the input letters have run out, we check the value of $E$, if it if true then the input string is in the language, if false, it is not.
The finite automata for this language can contain should require only two states:

State 1: $E$ is true, this is the start state and the accept or final state.

State 2: $E$ is false.

The picture of the machine for this language is:

Example: -

Suppose we want to build a finite automata that accepts all the words in the language:

$$a(a+b)^*$$

the looks like this: -
- All strings that start with the letter a, and the start state is x.
- If the first letter read is a b, then we go to a dead-end state y ("dead end state" is an informal way of describing a state that no string can leave once it has entered).
- If the first letter is an a, then we go to the dead-end state z, where z is a final state.

The same language may be accepted by a four state machine, as follows:

And we can be carried further to a five-state FA as follows:
We can see that there is not a unique machine for a given language.

Example :-

Let us contemplate the possibility of building an FA that accept all words containing a triple letter, either aaa or bbb and only those words.

The machine must have starts state and final state and must have a path of three edges, with no loops to accept the word aaa, therefore, the machine look like this:-

The same FA can be used for bbb but different states, but if bbb path shared any of the same state as the aaa path, we could mix a’s and b’s and mistakenly got to + anyway, then we need two additional states only because the path could share the same final state without any problem :-
If we are moving anywhere along the a-path and we read b before the third a, we jump to the b-path in progress and vice versa :-
Example :-

Let us consider the FA pictured bellow :

![FA Diagram]

The machine accept all words with b as the third letter and reject others, the regular expression for this FA is :-

\[(aab+abb+bab+bbb)(a+b)^*\]

And

\[(a+b)(a+b)(b)(a+b)^* = (a+b)^2b(a+b)^*\]

But the second formula is not, strictly speaking a regular expression, because it uses the symbol 2, which is not included in the kit.

Example :-

Let us consider this FA, which accepts only the word **baa** :-
The language accept this FA is:

\[ L = \{ \text{baa} \} \]

Example:

This FA accept the two strings baa and ab:
Example :-

The following FA accept only null :-

Notice that the left state is both a start and final state, all words other than \( \Lambda \) go to the right state and stay.

Example :-

Consider the following FA :-
No matter which state we are in, when we read an a, we go to the right hand state, and when we read a b, we go to the left hand state. Any input string that ends in the + state must end in the letter a, and any string ending in a must end in +, therefore the language accepted by this machine is:

\[(a+b)^*a\]

Example:

The language in the previous example does not include \( \Lambda \), if we add \( \Lambda \) we get the language of all words that do not end in b, this is accepted by the FA below:

CHAPTER 5

Samer H. Ali Computer Science Dep.
Transition Graph :-

In FA, we had a unique path through the machine for every input string, we now expand definition to allow a word-label edges, and some string have no paths at all, while some have several. The difficult new machine called Transition Graph.

Transition Graph, is a collection of three things :-

1- A finite set of states, at lest one of which is designated as the start state(-), and some (may be none) of which are designated as final state(+).
2- An alphabet \( \Sigma \) of possible input letters from which input strings are formed.
3- A finite set of transitions (edge labels) that show how to go from some states to some others, based on reading specified substring of input letters (possibly even the null string).

Example :-

Let us interpret this machine as an FA, this machine accept not only does (bba) alone to get the final state, but all other input strings end up nowhere. If we read the first letter of the input string (a), we have no direction to where to do. The another problem with this machine when the input string (bbabb), the machine accept first three letters, but then something undetermined happens when read any more of the input letters.
This machine tell as if the input string if fails to be acceptable, we must got to the garbage collection state and read through the rest of the input string in the full knowledge that we can never leave there.

The two machines are equivalent and accept the same string, but they have different number of states.

**Definition :-**

When an input string that has not been completely read reaches a state that it cannot leave because there is no outgoing edge that it follow, we say that the input crashes at that state, execution then terminate and the input must be rejected.

Crash an input string in FA not possible because there is always an outgoing string from each state, and the remain letters unread.
Example :-

Let us examine the input string (abbab) on the following machine :-

We must start from state 1 and read proceed along the outgoing edge labeled (abb) to state 2 and the read input letters (abb) are consume, from state 2 move to state 3 along with Λ edge (when an edge labeled with the string Λ, it means that we can take the ride it offers free without consuming any letters from the input string), at state 3, we cannot read (aa), so we must read (a) and go to state 4, here we have a (b) left in the input string but no edge to follow, so we must crash and reject the input string (abbab).

Examples :-

These two machines are equivalent.
This machine accept nothing.

And this machine accept only \( \Lambda \) string.

It is important for us to notice that every FA is also TG, but not every TG is FA.

**Definition :-**

A Generalized Transition Graph (GTG) is a collection of three things:-

1- A finite set of states, of which at least one state is designed as start state, and some (may be none) are final state.

2- An Alphabet \( \Sigma \) of input letters.

3- Directed edges connecting some pairs of states, each labeled with a regular expression.

**Example :-**
CHAPTER 6

Kleen’s Theorem

Theorem 1:-

Any language that can be defined by one of the following :-

Regular Expression

Or   Finite Automata

Or   Transition Graph

Can be defined by other two .

Proof :-

There are three sections of our proof :-

Part 1 :- Every language that can be defined by Finite Automata can be defined by Transition Graph.

Part 2 :- Every language that can be defined by Transition Graph can be defined by Regular Expression.

Part 3 :- Every language that can be defined by Regular Expression can be defined by Finite Automata.

Proof of Part 1 :-

Every language that can be defined by Finite Automata can be defined by Transition Graph, because Finite Automata is itself Transition Graph.

Proof of Part 2 :-

In this proof must construct procedure that start with a Transition Graph and end with Regular Expression that define the same language.
If TG have more than one start state combine them to one start state as in the example :-

Will become :-

\[ \begin{array}{c}
1 \xrightarrow{b} 2 \\
3 \xrightarrow{ab} 5 \xrightarrow{aa} 4 \\
\hline
\end{array} \]
And the same for final states :-

Will become :-

\[
\begin{align*}
\text{b} & \quad \rightarrow \\ \text{aa} & \quad \rightarrow \\
\end{align*}
\]

\[
\begin{align*}
\text{aba} & \quad \rightarrow \\ \text{b} & \quad \rightarrow \\
\end{align*}
\]

\[
\begin{align*}
\text{b} & \quad \rightarrow \\ \text{aa} & \quad \rightarrow \\
\text{aba} & \quad \rightarrow \\
\end{align*}
\]
We are going to build piece by piece the regular expression that define the same language of TG, We shall change the TG into a GTG :-

If the state has more than one loop circling back to itself, replace them by one loop labeled with a regular expression :-

Similarly, suppose two states are connected by more than one edge going in the same state :-

The edges labeled $r_1$ and $r_2$ can be replaced with the edge labeled with a regular expression $r_1+r_2$ :-

And also we can eliminate state among states, for example if we have three states connected by edges labeled by regular expression or string, we can eliminate the middle state and go directly form one state to the other through edge labeled with the regular expression that is concatenation of the two labels :-
Another case:

Will become:

Another case:

Will become:
Another case of eliminating middle state:

Algorithm:

the algorithm that proves that all TG’s can be turned into RE that define the same language:
Step 1: Create a unique, un-enterable minus state and a unique, un-leaveable plus state.

Step 2: One by one, in any order, bypass and eliminate all non- and + states in the TG. A state is bypassed by connecting each incoming edge with each outgoing edge, the label of each resultant edge is the concatenation of the label on the incoming edge with the label on the loop edge if there is one and the label on the outgoing edge.

Step 3: When two states are joined by more than one edge going in the same direction, unify them by adding their labels.

Step 4: Finally, when all that is left is one edge from - to +, the label on that edge is a RE that generates the same language as was recognized by the original machine.

Example:

Consider the TG below, which accept all strings that begin and end with double letters:

![Diagram of TG](image)

The machine has one start state and two final states, we must introduce a new unique final state:
We can note that the edge from start state to state 1 is a double edge, we can travel it by an aa or bb, we replace this by the regular expression (aa+bb), and also note that there is a loop at the state 1, we can loop back to state 1 by an (a) or a (b), we can replace the double loop by a single loop labeled by regular expression (a+b) :-

The algorithm of reduction TG to produce RE doesn’t actually tell us which state of the TG must eliminate next, the order of elimination states is the left up to our own direction, but the algorithm implies that it really doesn’t matter, so let us choose state 2 for elimination, the new label of edge from state 1 to final state after eliminating state 2 will be concatenation of aa and Λ, which is only aa. The machine will be like this :-
Now, if we choose to eliminate state 3, then we will have the same path as we began, so, let us eliminate state 1, and the machine will look like this:

The only state left is state 3, after eliminating state 3, the machine will look like this:
The machine defines the same RE :-

$$(aa+bb)(a+b)^* (aa) + (aa+bb)(a+b)^* (bb)$$

Then :-

$$(aa+bb)(a+b)^* (aa+bb)$$

This RE define the language, which begin and end with double letters, the language of the machine.
Example :-

Convert this TG into RE :-

Eliminating the states in the order 1, 2, 3 gives this procession of TGs :-
Then:

And then Eliminating state 3:

\[ ab'a + (b + a'b)(a + b'b')(\Lambda + b'b') \]