**Regular Expressions: Formal Definition**

We construct REs from primitive constituents (basic elements) by repeatedly applying certain recursive rules as given below. (In the definition)

**Definition :** Let*S*be an alphabet. The regular expressions are defined recursively as follows.

**Basis :**

* 1. is a RE
1. is a RE

iii) , *a* is RE.

These are called primitive regular expression i.e. Primitive Constituents

**Recursive Step :**

If and are REs over, then so are

i) 

1. 
2. 
3. 

**Closure :** *r*is RE over only if it can be obtained from the basis elements (Primitive REs) by a finite no ofapplications of the recursive step (given in 2).

**Example :** Let = { 0,1,2 }. Then (0+21)\*(1+ *F* ) is a RE, because we can construct this expression by applying the above rules as given in the following step.

|  |  |  |
| --- | --- | --- |
| **Steps** | **RE Constructed** | **Rule Used** |
| 1 | 1 | Rule 1(iii) |
| 2 |  | Rule 1(i) |
| 3 | 1+ | Rule 2(i) & Results of Step 1, 2 |
|  |  |



|  |  |  |
| --- | --- | --- |
| 4 | (1+ ) | Rule 2(iv) & Step 3 |
|  |  |
| 5 | 2 | 1(iii) |
| 6 | 1 | 1(iii) |
| 7 | 21 | 2(ii), 5, 6 |
| 8 | 0 | 1(iii) |
| 9 | 0+21 | 2(i), 7, 8 |
| 10 | (0+21) | 2(iv), 9 |
| 11 | (0+21)\* | 2(iii), 10 |
| 12 | (0+21)\* | 2(ii), 4, 11 |

**Language described by REs :** Each describes a language (or a language is associated with every RE). Wewill see later that REs are used to attribute regular languages.

**Notation :** If*r*is a RE over some alphabet then*L*(*r*)is the language associate with*r*. We can define thelanguage *L*(*r*) associated with (or described by) a REs as follows.

1. is the RE describing the empty language i.e. *L*() *= *.
2. is a RE describing the language {} i.e. *L*() = {} .

3. , *a* is a RE denoting the language {*a*} i.e **.** *L*(a) = {a} .

4. If and are REs denoting language *L*() and *L*() respectively, then

i) is a regular expression denoting the language *L*() = *L*() ∪ *L*()

1. is a regular expression denoting the language *L*()=*L*() *L*()
2. is a regular expression denoting the language 
3. () is a regular expression denoting the language *L*(()) = *L*()

**Example :** Consider the RE (0\*(0+1)). Thus the language denoted by the RE is

*L*(0\*(0+1)) = *L*(0\*) *L*(0+1) .......................by 4(ii)

* *L*(0)\**L*(0)∪ *L*(1)
* { , 0,00,000,........} {0} {1}
* { , 0,00,000,........} {0,1}
* {0, 00, 000, 0000,..........,1, 01, 001, 0001,...............}

**Precedence Rule**

Consider the RE *ab* + *c*. The language described by the RE can be thought of either *L*(*a*)*L*(*b*+*c*) or

*L*(*ab*)*L*(*c*)as provided by the rules (of languages described by REs) given already. But these tworepresents two different languages lending to ambiguity. To remove this ambiguity we can either

1. Use fully parenthesized expression- (cumbersome) or
2. Use a set of precedence rules to evaluate the options of REs in some order. Like other algebras mod in mathematics.

For REs, the order of precedence for the operators is as follows:

1. The star operator precedes concatenation and concatenation precedes union (+) operator.
2. It is also important to note that concatenation & union (+) operators are associative and union operation is commutative.

Using these precedence rule, we find that the RE ab+c represents the language *L*(*ab*) *L*(*c*)i.e. it should begrouped as ((*ab*)+*c*).

We can, of course change the order of precedence by using parentheses. For example, the language represented by the RE *a*(*b*+*c*) is *L*(*a*)*L*(*b*+*c*).

**Example :** The RE*ab*\*+*b*is grouped as((*a*(*b*\*))+*b*)which describes the language*L*(*a*)(*L*(*b*))\**L*(*b*)

**Example :** The RE(*ab*)\*+*b*represents the language(*L*(*a*)*L*(*b*))\* *L*(*b*).

**Example :** It is easy to see that the RE (0+1)\*(0+11) represents the language of all strings over {0,1} which areeither ended with 0 or 11.

**Example :** The regular expression*r*=(00)\*(11)\*1 denotes the set of all strings with an even number of 0'sfollowed by an odd number of 1's i.e.



**Note :** The notation is used to represent the RE *rr*\*. Similarly, represents the RE *rr*, denotes *r*,and so on.

An arbitrary string over = {0,1} is denoted as (0+1)\*.

**Exercise :** Give a RE*r*over {0,1} s.t.*L*(*r*)={ has at least one pair of consecutive 1's}

**Solution :** Every string in*L*(*r*)must contain 00 somewhere, but what comes before and what goes before iscompletely arbitrary. Considering these observations we can write the REs as (0+1)\*11(0+1)\*.

**Example :** Considering the above example it becomes clean that the RE (0+1)\*11(0+1)\*+(0+1)\*00(0+1)\*represents the set of string over {0,1} that contains the substring 11 or 00.

**Example :** Consider the RE 0\*10\*10\*. It is not difficult to see that this RE describes the set of strings over {0,1}that contains exactly two 1's. The presence of two 1's in the RE and any no of 0's before, between and after the 1's ensure it.

**Example :** Consider the language of strings over {0,1} containing two or more 1's.

**Solution :** There must be at least two 1's in the RE somewhere and what comes before, between, and after iscompletely arbitrary. Hence we can write the RE as (0+1)\*1(0+1)\*1(0+1)\*. But following two REs also represent the same language, each ensuring presence of least two 1's somewhere in the string

1. 0\*10\*1(0+1)\*
2. (0+1)\*10\*10\*

**Example :** Consider a RE*r*over {0,1} such that

*L*(*r*) = { has no pair of consecutive 1's}

**Solution :** Though it looks similar to ex ……., it is harder to construct to construct. We observer that, whenever

a 1 occurs, it must be immediately followed by a 0. This substring may be preceded & followed by any no of 0's. So the final RE must be a repetition of strings of the form: 00…0100….00 i.e. 0\*100\*. So it looks like the

RE is (0\*100\*)\*. But in this case the strings ending in 1 or consisting of all 0's are not accounted for. Taking these observations into consideration, the final RE is *r* = (0\*100\*)(1+ )+0\*(1+).

**Alternative Solution :**

The language can be viewed as repetitions of the strings 0 and 01. Hence get the RE as *r* = (0+10)\*(1+).This is a shorter expression but represents the same language.

Regular Expression:

FA to regular expressions:

**FA to RE (REs for Regular Languages) :**

**Lemma :** If a language is regular, then there is a RE to describe it. i.e. if*L*=*L*(*M*)for some DFA*M*, then thereis a RE *r* such that *L* = *L*(*r*).

**Proof :** We need to construct a RE*r*such that . Since *M* is a DFA, it has a finite

no of states. Let the set of states of *M* is *Q* = {1, 2, 3,..., *n*} for some integer n. [ Note **:** if the *n* states of *M* were denoted by some other symbols, we can always rename those to indicate as 1, 2, 3,..., *n* ]. The required RE is constructed inductively.

**Notations : **is a RE denoting the language which is the set of all strings *w* such that *w* is the label of a

path from state *i* to state *j *in *M*, and that path has no intermediate state whose number is greater then *k*. ( *i* & *j* (begining and end pts) are not considered to be "intermediate" so *i* and /or *j* can be

greater than *k* )

We now construct inductively, for all *i*, *j Q* starting at *k* = 0and finally reaching *k* = *n*.

**Basis :** *k*= 0, i.e. the paths must not have any intermediate state ( since all states are numbered 1 or above). There are only two possible paths meeting the above condition **:**

1. A direct transition from state *i* to state *j*.
	* = a if then is a transition from state *i* to state *j* on symbol the single symbol *a*.
	* = if there are multiple transitions from state *i* to state *j* on symbols

.

* + = *f* if there is no transition at all from state *i* to state *j*.
1. All paths consisting of only one node i.e. when *i* = *j*. This gives the path of length 0 (i.e. the RE denoting the string ) and all self loops. By simply adding Î to various cases above we get the corresponding REs i.e.
	* = + *a* if there is a self loop on symbol *a* in state *i .*
	* = + if there are self loops in state *i* as multiple symbols

.

* + = if there is no self loop on state *i.*

**Induction :**

Assume that there exists a path from state *i* to state *j* such that there is no intermediate state whose number is

greater than *k*. The corresponding Re for the label of the path is .

There are only two possible cases **:**

1. The path dose not go through the state *k* at all i.e. number of all the intermediate states are less than

*k*. So, the label of the path from state *i* to state *j* is tha language described by the RE .

1. The path goes through the state *k* at least once. The path may go from *i* to *j* and *k* may appear more than once. We can break the into pieces as shown in the figure 7.



Figure 7

1. The first part from the state *i* to the state *k* which is the first recurence. In this path, all intermediate

states are less than *k* and it starts at iand ends at *k*. So the RE denotes the language of the label of path.

1. The last part from the last occurence of the state *k* in the path to state *j*. In this path also, no

intermediate state is numbered greater than *k*. Hence the RE denoting the language of the label of the path.

1. In the middle, for the first occurence of *k* to the last occurence of *k* , represents a loop which may be taken zero times, once or any no of times. And all states between two consecutive *k*'s are numbered less than *k*.

Hence the label of the path of the part is denoted by the RE .The label of the path from state *i* to state *j* is the concatenation of these 3 parts which is



Since either case 1 or case 2 may happen the labels of all paths from state *i* to *j* is denoted by the following RE



We can construct for all *i*, *j *{1,2,..., *n*} in increasing order of *k* starting with the basis *k* = 0 upto *k* = *n*

since depends only on expressions with a small superscript (and hence will be available). WLOG, assume

that state 1 is the start state and are the *m* final states where *ji *{1, 2, ... , *n* }, and

. According to the convention used, the language of the automatacan be denoted by the RE



Since is the set of all strings that starts at start state 1 and finishes at final state following the transition of the FA with any value of the intermediate state (1, 2, ... , *n*) and hence accepted by the automata.

Regular Grammar:

A *grammar *is right-linear if each production has one of the following three forms:

* *AcB* ,
* *Ac*,
* *A*

Where *A*, *B *( with *A* = *B* allowed) and *.* A *grammar G* is left-linear if each production has once ofthe following three forms.

*ABc* , *Ac*, *A*

A right or left-linear grammar is called a regular grammar.

Regular grammar and Finite Automata are equivalent as stated in the following theorem.

**Theorem :** A language*L*is regular iff it has a regular grammar. We use the following two lemmas to prove theabove theorem.

**Lemma 1 :** If*L*is a regular language, then*L*is generated by some right-linear grammar.

**Proof :** Let be a DFA that accepts *L*.

Let and .

We construct the right-linear grammar by letting



*N* = *Q* , and

[ Note: If , then ]

Let . For *M* to accept *w*, there must be a sequence of states such that



and

By construction, the grammar G will have one production for each of the above transitions. Therefore, we have the corresponding derivation.



Hence *w L*(*g*).

Conversely, if , then the derivation of w in G must have the form as given above. But, then the construction of *G* from *M* implies that

, where , completing the proof.

**Lemma 2 :** Let be a right-linear grammar. Then *L*(*G*) is a regular language.

Proof: To prove it, we construct a FA *M* from *G* to accept the same language.

is constructed as follows:



( is a special sumbol not in *N* )



,

For any and and is defined as



if

and , if .

We now show that this construction works.

Let . Then there is a derivation of *w* in *G* of the form



By contradiction of *M*, there must be a sequence of transitions



implying that i.e. *w* is accepted by *M*.

Conversely, if is accepted by *M*, then because is the only accepting state of *M*, the transitions causing *w* to be accepted by *M* will be of the form given above. These transitions corresponds to a

derivationof *w* in the grammar *G*. Hence , completing the proof of the lemma.

Given any left-linear grammar *G* with production of the form , we can construct from it a right-

linear grammar by replacing every production of *G* of the form with 

It is easy to prove that . Since is right-linear, is regular. But then so are

i.e. because regular languages are closed under reversal.

Putting the two lemmas and the discussions in the above paragraph together we get the proof of the theorem-

A language *L* is regular iff it has a regular grammar

**Example :** Consider the grammar



It is easy to see that *G* generates the language denoted by the regular expression (01)\*0.

The construction of lemma 2 for this grammar produces the follwoing FA.

This FA accepts exactly (01)\*1.

Decisions Algorithms for CFL

In this section, we examine some questions about CFLs we can answer. A CFL may be represented using a CFG or PDA. But an algorithm that uses one representation can be made to work for the others, since we can construct one from the other.

**Testing Emptiness :**

**Theorem :** There are algorithms to test emptiness of a CFL.

**Proof :** Given any CFL*L*, there is a CFG*G*to generate it. We can determine, using the construction described

in the context of elimination of useless symbols, whether the start symbol is useless. If so, then ; otherwise not.

**Testing Membership :**

Given a CFL *L* and a string *x*, the membership, problem is to determine whether ?

Given a PDA *P* for *L*, simulating the PDA on input string *x* doesnot quite work, because the PDA can grow its stack indefinitely on input, and the process may never terminate, even if the PDA is deterministic.

So, we assume that a CFG is given such that *L* = *L*(*G*).

Let us first present a simple but inefficient algorithm.

Convert *G* to in CNF generating . If the input string , then we need to

determine whether and it can easily be done using the technique given in the context of elimination of

-production. If , then iff . Consider a derivation under a grammar in CNF. At every step, a production in CNF in used, and hence it adds exactly one terminal symbol to the sentential form.

Hence, if the length of the input string *x* is n, then it takes exactly *n* steps to derive *x* ( provided *x* is in ).

Let the maximum number of productions for any nonterminal in is *K*. So at every step in derivation, there

are atmost *k* choices. We may try out all these choices, systematically., to derive the string *x* in . Since

there are atmost i.e. choices. This algorithms is of exponential time complexity. We now present an efficient (polynomial time) membership algorithm.

**Pumping Lemma:**

**Limitations of Finite Automata and Non regular Languages :**

The class of languages recognized by FA s is strictly the regular set. There are certain languages which are non regular i.e. cannot be recognized by any FA

Consider the language 

In order to accept is language, we find that, an automaton seems to need to remember when passing the center point between *a*'s and *b*'s how many *a*'s it has seen so far. Because it would have to compare that with the number of *b*'s to either accept (when the two numbers are same) or reject (when they are not same) the input string.

But the number of *a*'s is not limited and may be much larger than the number of states since the string may be arbitrarily long. So, the amount of information the automaton need to remember is unbounded.

A finite automaton cannot remember this with only finite memory (i.e. finite number of states). The fact that *FA* s have finite memory imposes some limitations on the structure of the languages recognized. Inductively, we can say that a language is regular only if in processing any string in this language, the information that has to

be remembered at any point is strictly limited. The argument given above to show that is non regular is

informal. We now present a formal method for showing that certain languages such as are non regular

**Properties of CFL’s**

**Closure properties of CFL:**

We consider some important closure properties of CFLs.

**Theorem :** If and are CFLs then so is 

**Proof :** Let and be CFGs generating. Without loss of generality,

we can assume that . Let is a nonterminal not in or . We construct the grammar

from and , where

,



We now show that 

Thus proving the theorem.

Let . Then . All productions applied in their derivation are also in . Hence i.e.



Similarly, if , then 

Thus .

Conversely, let . Then and the first step in this derivation must be either or

. Considering the former case, we have 

Since and are disjoint, the derivation must use the productions of only ( which are also in

) Since is the start symbol of . Hence, giving .

Using similar reasoning, in the latter case, we get . Thus .

So, , as claimed

**Theorem :** If and are CFLs, then so is .

**Proof :** Let and be the CFGs generating and respectively.

Again, we assume that and are disjoint, and is a nonterminal not in or . we construct the CFG

from and , where



We claim that

To prove it, we first assume that and . Then and . We can derive the string *xy* in as shown below.



since and . Hence .

For the converse, let . Then the derivation of *w* in will be of the form

i.e. the first step in the derivation must see the rule . Again, since and are

disjoint and and , some string *x* will be generated from using productions in ( which are

also in ) and such that .

Thus

Hence and .

This means that *w* can be divided into two parts *x*, *y* such that and . Thus .This completes the proof

**Theorem :** If*L*is a CFL, then so is *.*

**Proof :** Let be the CFG generating *L*. Let us construct the CFG from *G*

where .

We now prove that , which prove the theorem.

can generate in one step by using the production since , can generate any string in *L*.

Let for any *n* >1 we can write where for . *w* can be generated by using following steps.



First (*n*-1)-steps uses the production *SSS* producing the sentential form of *n* numbers of *S* 's. The

nonterminal *S* in the *i*-th position then generates using production in *P* ( which are also in )

It is also easy to see that G can generate the empty string, any string in L and any string for *n* >1 and none other.

Hence 

**Theorem :** CFLs are not closed under intersection

**Proof :** We prove it by giving a counter example. Consider the language .The following CFG generates *L*1 and hence a CFL



The nonterminal *X* generates strings of the form and *C* generates strings of the form , .

These are the only types of strings generated by *X* and *C*. Hence, *S* generates .

Using similar reasoning, it can be shown that the following grammar and hence it is also a CFL.



But, and is already shown to be not context-free.

Hence proof.

**Theorem :** A CFL's are not closed under complementations

**Proof :** Assume, for contradiction, that CFL's are closed under complementation. SInce, CFL's are also closed

under union, the language , where and are CFL's must be CFL. But by DeMorgan's law



This contradicts the already proved fact that CFL's are not closed under intersection.

But it can be shown that the CFL's are closed under intersection with a regular set.

**Theorem :** If*L*is a CFL and*R*is a regular language, then is a CFL.

**Proof :** Let be a PDA for *L* and let be a DFA for *R*. We construct a PDA *M* from *P* and *D* as follows



where is defined as

contains iff

and contains 

The idea is that *M* simulates the moves of *P* and *D* parallely on input *w*, and accepts *w* iff both *P* and *D* accepts. That means, we want to show that



We apply induction on *n*, the number of moves, to show that

iff

and



**Basic Case** is*n*=0. Hence , and . For this case it is trivially true

**Inductive hypothesis :** Assume that the statement is true for*n*-1.

**Inductive Step :** Let*w*=*xa*and



Let

By inductive hypothesis, and 

From the definition of and considering the *n*-th move of the PDA *M* above, we have



and 

Hence and 

If and , then and we got that if *M* accepts *w*, then both *P* and *D* accepts it.

We can show that converse, in a similar way. Hence is a CFL ( since it is accepted by a PDA *M* ) This property is useful in showing that certain languages are not context-free.

**Example :** Consider the language



Intersecting *L* with the regular set , we get





Which is already known to be not context-free. Hence *L* is not context-free **Theorem :** CFL's are closed under reversal. That is if*L*is a CFL, then so is 

**Proof :** Let the CFG generates *L*. We construct a CFG where

. We now show that , thus proving the theorem. We need to prove that

iff .

The proof is by induction on *n*, the number of steps taken by the derivation. We assume, for simplicity (and of

course without loss of generality), that *G* and hence are in CNF.

The basis is *n*=1 in which case it is trivial. Because must be either or BC with .

Hence iff 

Assume that it is true for (n-1)-steps. Let . Then the first step must apply a rule of the form and it gives

where and 

By constructing of G', 

Hence



The converse case is exactly similar

**Substitution :**

, let be a language (over any alphabet). This defines a function *S*, called substitution, on which is

denoted as - for all 

This definition of substitution can be extended further to apply strings and langauge as well.

If , where , is a string in , then



.

Similarly, for any language *L*,



The following theorem shows that CFLs are closed under substitution.

**Thereom :** Let is a CFL, and *s* is a substitution on such that is a CFL for all , thus *s*(*L*)is a CFL

**Proof :** Let*L*=*L*(*G*)for a CFG and for every , for some

. Without loss of generality, assume that the sets of nonterminals *N* and 's are disjoint.

Now, we construct a grammar , generating *s*(*L*), from *G* and 's as follows **:**

****













* consists of
	1. and
	2. The production of *P* but with each terminal *a* in the right hand side of a production replaced by everywhere.

We now want to prove that this construction works i.e. iff .

**If Part :** Let then according to the definition there is some string and for such that 

We will show that .

From the construction of , we find that, there is a derivation corresponding to the string

(since contains all productions of G but every ai replaced with in the RHS of any production).

Every is the start symbol of and all productions of are also included in .

Hence



Therefore, 



**(Only-if Part)** Let . Then there must be a derivative as follows :

(using the production of *G* include in as modified by (step 2) of the construction of .)

Each () can only generate a string , since each 's and *N* are disjoin. Therefore, we get



since 

since 



The string is formed by substituting strings for each and hence .

**Theorem :** CFL's are closed under homomorphism

**Proof :** Let be a CFL, and *h* is a homomorphism on i.e for some alphabets . consider

the following substitution S:Replace each symbol by the language consisting of the only string h(a), i.e.

for all . Then, it is clear that, *h*(*L*) = *s*(*L*). Hence, CFL's being closed under substitution must also be closed under homomorphism.