Contents

1. Introduction to Automata Theory & Compiler Design

The theory of computation is a field of study that deals with the nature and limits of computation. It encompasses various mathematical and theoretical aspects related to the capabilities and limitations of computers and computational systems.

2 Regular Expressions and Languages

2.1 Regular Expressions

Regular expressions, often abbreviated as regex or regexp, are powerful and concise sequences of characters used to define search patterns. They are widely employed in various programming languages, text editors, and tools for string manipulation, pattern matching, and data validation. Here are some fundamental concepts related to regular expressions:

- **Character Classes:**
  - [ ]: Defines a character class. Matches any one of the characters inside the brackets. For example, [aeiou] matches any vowel.
- **Quantifiers:**
  - ***:** Matches zero or more occurrences of the preceding character or group.
  - +: Matches one or more occurrences of the preceding character or group.
  - ?: Matches zero or one occurrence of the preceding character or group.
- **Anchors:**
  - ^: Anchors the regex at the beginning of a line.
  - $: Anchors the regex at the end of a line.
- **Wildcards:**
  - . (dot): Matches any single character except a newline.
- **Escape Characters:**
  - \: Escapes a metacharacter, allowing it to be treated as a literal character.
- **Grouping and Capturing:**
  - ( ): Groups characters together. Also used for capturing substrings.
- **Alternation:**
  - | (pipe): Represents alternation, allowing the matching of either the expression before or after it.
- **Quantifiers:**
  - {n}: Matches exactly n occurrences of the preceding character or group.
  - {n,}: Matches n or more occurrences.

Regular Expression: b{2,}

- {n,m}: Matches between n and m occurrences.

- **Character Escapes:**
  - \d: Matches any digit (0-9).
  - \w: Matches any word character (alphanumeric + underscore).
  - \s: Matches any whitespace character (space, tab, newline).
- **Negation:**
  - [^]: Negates a character class, matching any character not inside the brackets.
- **Anchors:**
  - \b: Matches a word boundary.
  - \B: Matches a non-word boundary.
- **Modifiers:**
  - i: Case-insensitive matching.
  - g: Global matching (find all matches rather than stopping after the first).

Regular expressions provide a concise and expressive way to describe complex patterns in strings, making them a versatile tool for tasks like text searching, data validation, and text manipulation. They are supported in many programming languages, including Python, JavaScript, Java, and others, often through libraries or native language support.

Regular expressions (regex or regexp) can be understood in terms of Finite State Machines (FSMs), which are theoretical models used to represent and recognize patterns in strings. **Regular expressions and FSMs are closely related because regular expressions define patterns that can be recognized by FSMs.**

- The language accepted by finite automata can be easily described by simple expressions called Regular Expressions. It is the most effective way to represent any language.
- The languages accepted by some regular expressions are referred to as Regular languages.
- A regular expression can also be described as a sequence of patterns that defines a string.
- Regular expressions are used to match character combinations in strings. The string searching algorithm used this pattern to find the operations on a string.

In the context of regular expressions and FSMs:

Alphabet (Σ): The alphabet consists of all possible symbols that can appear in the input string. In regular expressions, these symbols are the characters or tokens that make up the strings you want to match.

States (Q): States represent different stages in the recognition process. Each state corresponds to a specific part of the pattern that has been recognized so far.

Transitions (δ): Transitions describe the changes of state based on input symbols. In the context of regular expressions, transitions represent the progression through the pattern.

Accepting States (F): Accepting states indicate successful recognition of the pattern. In the case of regular expressions, an accepting state is reached when the entire input string has been matched according to the pattern.

Now, let's consider a simple example to illustrate the connection between regular expressions and FSMs:

For instance:

In a regular expression, x* means zero or more occurrence of x. It can generate {e, x, xx, xxx, xxxx, ...}

In a regular expression, "x*" means zero or more occurrences of 'x'. It can generate the set {ε, x, xx, xxx, xxxx, ...}, where ε represents an empty string. Here's how you can interpret it in terms of a Finite State Machine (FSM):

```
Start State ---(x)---> Accepting State

    ^     |                    |

    |     |____________|

    |______(ε)
```

Additionally, the epsilon transition (ε) allows the machine to move from the start state to the accepting state without consuming any 'x', allowing for an empty string match.

the pattern "x*" means zero or more occurrences of the character 'x'. Here are some sample examples:

Pattern: a*   Matches: "", "a", "aa", "aaa", ...

Pattern: \d* (zero or more digits)   Matches: "", "0", "123", "987654", ...

Pattern: [A-Za-z]* (zero or more letters) Matches: "", "abc", "XYZ", "Hello", ...

Pattern: .* (zero or more of any character)   Matches: "", "x", "xy", "xyz", "abc", "123", ...

In a regular expression, x+ means one or more occurrences of x. It can generate {x, xx, xxx, xxxx, ...}

In terms of a Finite State Machine (FSM), the regular expression "x+" (one or more occurrences of 'x') can be represented by a simple state machine with two states: the start state and the accepting state. The transition from the start state to the accepting state occurs on each occurrence of the character 'x'. Once in the accepting state, the machine remains in that state for any subsequent occurrences of 'x'. This reflects the idea of matching one or more occurrences of 'x'.

Here is a simple illustration of the FSM for the regular expression "x+":

```
Start State ----(x)----> Accepting State
    ^                        |
    |________________|
```

This FSM will accept sequences like "x", "xx", "xxx", "xxxx", and so on, but it won't accept the empty string.

In a regular expression, the notation "x+" means one or more occurrences of the character or pattern 'x'. Here's an example of how it works:

If 'x' is a single character, like a letter or a digit, then the regular expression "x+" would match one or more occurrences of that character 'x'. For example:

Pattern: a+

Matches: "a", "aa", "aaa", "aaaa", ...

Pattern: 1+

Matches: "1", "11", "111", "1111", ...

If 'x' is a more complex pattern, the regular expression will match one or more occurrences of that entire pattern. For example:

Pattern: \d+

Matches: "0", "123", "987654", ...

Pattern: [A-Za-z]+

Matches: "abc", "XYZ", "Hello", ...

In a regular expression, ?: Matches zero or one occurrence of the preceding character or group.

In terms of a Finite State Machine (FSM), the regular expression? (question mark) indicates zero or one occurrence of the preceding character or group.

Here's a simple FSM representation:

```
Start State ---(ε)---> Accepting State ---(?)---> Optional State
    ^                        |                        |
    |________________|________________|
```

in a regular expression denotes zero or one occurrence of the preceding character or group. Here are some sample examples:

Pattern: a? Matches: "", "a"

Pattern: \d? (zero or one digit) Matches: "", "1", "9", ...

Pattern: [A-Za-z]? (zero or one letter) Matches: "", "A", "b", ...

Pattern: x? Matches: "", "x"

{n}: Matches exactly n occurrences of the preceding character or group.

In terms of a Finite State Machine (FSM), the regular expression {n} indicates exactly n occurrences of the preceding character or group. Representing this in an FSM is a bit more complex, as it involves having n transitions from the start state to an intermediate state and from the intermediate state to an accepting state.

A sample illustration is

Start State ---(x)---> Intermediate State ---(x)---> ... ---(x)---> Accepting State

For example, let's say you have the pattern a{3}, which means exactly three occurrences of the character 'a':

This FSM will accept sequences like "aaa" and only "aaa."

Here are some sample examples:

Pattern: a{3} Matches: "aaa"

Pattern: \d{2} (exactly two digits) Matches: "12", "34", ...

Pattern: [A-Za-z]{4} (exactly four letters) Matches: "abcd", "WXYZ", ...

Pattern: x{0} (exactly zero occurrences of 'x') Matches: ""

{n,}: Matches n or more occurrences.

In terms of a Finite State Machine (FSM), the regular expression {n,} indicates matching n or more occurrences of the preceding character or group. Representing this in an FSM involves having at least n transitions from the start state to an intermediate state and then having optional transitions (0 or more) from the intermediate state back to itself.

Here's a simplified illustration:

```
Start State ---(x)---> Intermediate State ---(x)---> ... ---(x)---> Intermediate State ---(x)---> Accepting State
    ^                                                                                          |
    |------------------------------------------------------------------------------ |
```

For example, let's say you have the pattern a{2,} which means matching two or more occurrences of the character 'a':

This FSM will accept sequences like "aa", "aaa", "aaaa", and so on.

ome sample examples of the regular expression {n,}, which matches n or more occurrences of the preceding character or group:

Pattern: a{2,}

Matches: "aa", "aaa", "aaaa", ...

Pattern: \d{3,} (three or more digits)

Matches: "123", "4567", ...

Pattern: [A-Za-z]{4,} (four or more letters)

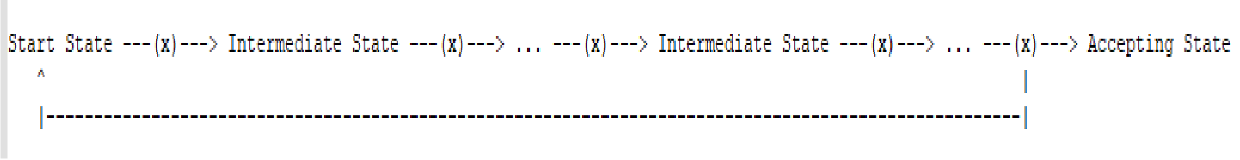
Matches: "abcd", "WXYZ", "lmno", ...

Pattern: x{0,} (zero or more occurrences of 'x')

Matches: "", "x", "xx", "xxx", ...

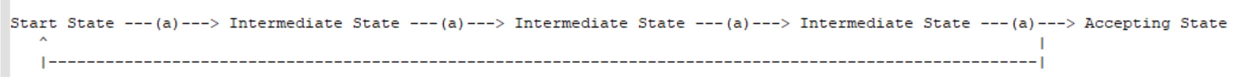
{n,m}: Matches between n and m occurrences

In terms of a Finite State Machine (FSM), the regular expression {n,m} indicates matching between n and m occurrences (inclusive) of the preceding character or group. Representing this in an FSM involves having at least n transitions from the start state to an intermediate state, optional transitions (0 or more) from the intermediate state back to itself, and at most (m-n) optional transitions from the intermediate state to an accepting state.



```
Start State ---(x)---> Intermediate State ---(x)---> ... ---(x)---> Intermediate State ---(x)---> ... ---(x)---> Accepting State
   ^                                                                                                  |
   |--------------------------------------------------------------------------------------------------|
```

For example, let's say you have the pattern a{2,4} which means matching between 2 and 4 occurrences of the character 'a':



```
Start State ---(a)---> Intermediate State ---(a)---> Intermediate State ---(a)---> Intermediate State ---(a)---> Accepting State
   ^                                                                                                           |
   |-----------------------------------------------------------------------------------------------------------|
```

This FSM will accept sequences like "aa", "aaa", and "aaaa."

sample examples of the regular expression {n,m}, which matches between n and m occurrences (inclusive) of the preceding character or group:

Pattern: a{2,4}   Matches: "aa", "aaa", "aaaa"

Pattern: \d{3,5} (between 3 and 5 digits) Matches: "123", "4567", "98765"

Pattern: [A-Za-z]{2,3} (between 2 and 3 letters) Matches: "ab", "XYZ", "lmn"

Pattern: x{0,2} (between 0 and 2 occurrences of 'x') Matches: "", "x", "xx"

2.3 Operations of Regular Expressions

Regular languages, as defined in formal language theory, can be expressed using regular expressions and recognized by finite state machines (FSMs). Let's discuss some common operations on regular languages along with examples represented in FSMs.

1. Union (L1 ∪ L2): The union of two regular languages results in a language containing all strings that belong to either language.

Example FSM:

Let L1 be the language of strings ending with "a."

Let L2 be the language of strings starting with "b."

The FSM for L1 ∪ L2 recognizes strings that either end with "a" or start with "b."

L U M = {s | s is in L or s is in M}

LUM = {ba, baaaba, bbbbaaa.. }

2. Concatenation (L1 · L2): The concatenation of two regular languages results in a language containing all possible concatenations of strings from the first language with strings from the second language.

Example FSM:

Let L1 be the language of strings containing only "0" or "1."

Let L2 be the language of strings containing only "2" or "3."

The FSM for L1 · L2 recognizes strings like "01," "32," etc.

3. Kleene Closure (L):* The Kleene closure of a regular language results in a language containing all possible repetitions (including zero) of strings from the original language.

Example FSM:

Let L be the language of strings with an even number of "a"s.

The FSM for L* recognizes strings with any number of "a"s, including an empty string, "aa," "aaaa," etc.

4. Intersection (L1 ∩ L2): The intersection of two regular languages results in a language containing only strings that are common to both languages.

Example FSM:

Let L1 be the language of strings containing "ab."

Let L2 be the language of strings containing "bc."

The FSM for L1 ∩ L2 recognizes strings that contain both "ab" and "bc."

L ⋂ M = {st | s is in L and t is in M}

5. **Complementation (¬L):** The complement of a regular language results in a language containing all strings not in the original language.

Example FSM:

Let L be the language of strings containing an even number of "0"s.

The FSM for ¬L recognizes strings with an odd number of "0"s.

These operations demonstrate how regular languages can be manipulated and combined using various set-theoretic operations. FSMs provide a graphical representation of these languages and their operations, making it easier to understand and implement algorithms for recognizing and generating strings within these languages.

Regular expressions are used for representing certain sets of strings in an algebraic fashion

1) Any terminal symbol i.e. symbols ϵ ∑ a,b,c, ^ , ϕ

Including ^ and ϕ are regular expressions.

2) The Union of two regular expressions is $R_1, R_2,$ (R_1+R_2)

also a regular expresion.

3) The Concatenation of two regular expressions $R_1, R_2, \rightarrow (R_1+R_2)$

is also a regular expression.

4) The iteration (or Closure) of a regular expressions $R \rightarrow R^*$ a^* =^,a, aa, aaa,

is also a regular expression.

5) The regular expression over Σ are precisely those obtained recursively by the application of the above rules once or several times.

Q: Write the regular expression for the language accepting all combinations of a's except the null string, over the set ∑ = {a}

The regular expression has to be built for the language

L = {a, aa, aaa, ...}

This set indicates that there is no null string. So we can denote regular expression as:

R = a+

Q: Write the regular expression for the language accepting all the string containing any number of a's and b's.

This will give the set as L = {ε, a, aa, b, bb, ab, ba, aba, bab, ...}, any combination of a and b.

The (a + b)* shows any combination with a and b even a null string.

Q: Write the regular expression for the language accepting all the string which are starting with 1 and ending with 0, over ∑ = {0, 1}.

In a regular expression, the first symbol should be 1, and the last symbol should be 0. The r.e. is as follows:

R = 1 (0+1)* 0

Q: Write the regular expression for the language starting and ending with a and having any having any combination of b's in between.

The regular expression will be:

R = a b* b

Q: Write the regular expression for the language starting with a but not having consecutive b's.

The regular expression has to be built for the language:

L = {a, aba, aab, aba, aaa, abab, ...}

The regular expression for the above language is:

R = {a + ab}*

Q: Write the regular expression for the language accepting all the string in which any number of a's is followed by any number of b's is followed by any number of c's.

As we know, any number of a's means a* any number of b's means b*, any number of c's means c*. Since as given in problem statement, b's appear after a's and c's appear after b's. So the regular expression could be:

R = a* b* c*

Q: Write the regular expression for the language over ∑ = {0} having even length of the string.

The regular expression has to be built for the language:

L = {ε, 00, 0000, 000000, ...}

The regular expression for the above language is:

R = (00)*

Q: Write the regular expression for the language having a string which should have atleast one 0 and alteast one 1.

The regular expression will be:

R = [(0 + 1)* 0 (0 + 1)* 1 (0 + 1)*] + [(0 + 1)* 1 (0 + 1)* 0 (0 + 1)*]

Q: Describe the language denoted by following regular expression

r.e. = (b* (aaa)* b*)*

The language can be predicted from the regular expression by finding the meaning of it. We will first split the regular expression as:

r.e. = (any combination of b's) (aaa)* (any combination of b's)

L = {The language consists of the string in which a's appear triples, there is no restriction on the number of b's}

Q: Write the regular expression for the language L over ∑ = {0, 1} such that all the string do not contain the substring 01.

The Language is as follows:

L = {ε, 0, 1, 00, 11, 10, 100, ...}

The regular expression for the above language is as follows:

R = (1* 0*)

Q: Write the regular expression for the language containing the string over {0, 1} in which there are at least two occurrences of 1's between any two occurrences of 1's between any two occurrences of 0's.

At least two 1's between two occurrences of 0's can be denoted by (0111*0)*.

Similarly, if there is no occurrence of 0's, then any number of 1's are also allowed. Hence the r.e. for required language is:

R = (1 + (0111*0))*

Q: Write the regular expression for the language containing the string in which every 0 is immediately followed by 11.

The regular expectation will be:

R = (011 + 1)*

Q: Write regular expressions for the following language

a) the set of all strings such that the number of 0's is odd

b) set of all strings that do not contain 1101

a) Set of all strings such that the number of 0's is odd:

To express the language of strings with an odd number of 0's, you can use the regular expression:

Regular Expression: (1*01*0)*1*

Explanation:

(1*01*0)*: Zero or more occurrences of the pattern "1", followed by "0", followed by zero or more occurrences of "1".

1*: Zero or more occurrences of "1" at the end.

This regular expression represents strings with an odd number of 0's.

b) Set of all strings that do not contain 1101:

To express the language of strings that do not contain the substring "1101", you can use the regular expression:

Regular Expression: (0|1|10|110)*

Explanation:

(0|1|10|110)*: Zero or more occurrences of the patterns "0", "1", "10", or "110".

This regular expression represents strings that can be formed by concatenating any combination of "0", "1", "10", or "110", and it ensures that the forbidden substring "1101" is not present in the string.

Q: Write regular expressions for the for the following over {0,1}*

a) the set of all strings that begin with 110

b) the set of all strings that contain 1011

c) the set of all strings that contain exactly three 1's

a) Set of all strings that begin with 110:

Regular Expression: 110.*

Explanation:

110: Specifies that the string must start with "110".

.*: Represents zero or more occurrences of any character (wildcard) after "110".

This regular expression captures strings that start with the sequence "110".

b) Set of all strings that contain 1011:

Regular Expression: .*1011.*

Explanation:

.*: Represents zero or more occurrences of any character (wildcard) before and after the pattern.

1011: Specifies the required pattern "1011".

This regular expression captures strings that contain the substring "1011" anywhere within them.

c) Set of all strings that contain exactly three 1's:

Regular Expression: 0*1(0*1){2}0*

Explanation:

0*: Represents zero or more occurrences of "0".

1: Specifies the first occurrence of "1".

(0*1){2}: Specifies exactly two occurrences of the pattern "0*1", meaning two additional occurrences of "1".

0*: Represents zero or more occurrences of "0" after the three 1's.

This regular expression captures strings that contain exactly three occurrences of the digit "1".

Q: Write regular expressions for the following languages

a) the set of all strings of 0s and 1s not containing 101 as a substring

b) the set of all strings with an equal number of 0's and 1s. such that no prefix has two more 0'S THAN 1's , nor two more 1's than 0's

c) the set of strings of 0's and 1's whose number of 0's is divisible by five and whose number of 1's is even

a) Set of all strings not containing 101 as a substring:

Regular Expression: (0|1)*((?!101)(0|1))*

Explanation:

(0|1)*: Matches any combination of 0's and 1's (including an empty string).

((?!101)(0|1))*: Utilizes negative lookahead to ensure that "101" is not present as a substring.

This regular expression captures strings that do not contain "101" as a substring.

b) Set of all strings with an equal number of 0's and 1's, without a prefix having two more 0's than 1's or two more 1's than 0's:

Regular Expression: ε|(0*10*1)*0*1*

Explanation:

ε: Represents the empty string.

(0*10*1)*: Matches pairs of 0's and 1's in any order.

0*1*: Matches any additional 0's or 1's that may follow.

This regular expression captures strings with an equal number of 0's and 1's, ensuring that no prefix has two more 0's than 1's or two more 1's than 0's.

c) Set of strings with the number of 0's divisible by five and the number of 1's being even:

Regular Expression: 0*(00000)*(1*01*01*)*

Explanation:

0*: Matches any number of 0's at the beginning.

(00000)*: Matches groups of five 0's.

(1*01*01*)*: Matches any combination of 1's, 0's, and 1's in a way that ensures the number of 1's is even.

This regular expression captures strings where the number of 0's is divisible by five, and the number of 1's is even.

Q.Give english description of the languages of the following regular expressions

a) (1+ε)(00*1)*0*

b)(0*1*)*000(0+1)*

c)(0+10)*1*

a) (1+ε)(00*1)*0*

This regular expression describes the language of strings that either start with a "1" or are empty, followed by zero or more occurrences of the pattern "00*1," and ending with zero or more "0"s. In simpler terms:

- The string may start with a "1" or be empty.
- It can then have zero or more occurrences of "001," where "0" represents zero or more 0's.
- Finally, the string may end with zero or more "0"s.

b)(0*1*)*000(0+1)*

This regular expression describes the language of strings that consist of any combination of 0's and 1's, including the empty string, followed by the substring "000," and ending with zero or more occurrences of 0's or 1's. In simpler terms:

- The string can consist of any number of 0's and 1's, including the empty string.
- It must then contain the substring "000."
- The string can end with zero or more occurrences of 0's or 1's.

c)(0+10)*1*

This regular expression describes the language of strings that consist of any combination of "0" or "10," followed by zero or more occurrences of "1." In simpler terms:

- The string can have any combination of "0" or "10."
- It must end with zero or more occurrences of "1."

<u>Regular Expression- Examples</u>

Describe the following sets as regular Expressions

1) {0, 1, 2}   0 or 1 or 2

R = 0 + 1+ 2

2) {^, ab}

R = ^ ab

3) {abb, a, b, bba}   abb or a or b or bba

R =abb + a + b + bba

4) {^, 0, 00, 000, ……….}   closure ∞ 0

R = 0*

5) {1, 11, 111, 1111, ………..}

R =1*

Identities of Regular Expressions

1) Ø+ R=R
2) ØR + ØR = Ø   ^
3) $\in R = R\emptyset = \emptyset$
4) $\in^* = \in$ and $\emptyset^* = \in$
5) R + R = R
6) $R^*R^* = R^*$
7) $RR^* = R^*R$
8) $(R^*)^* = R^*$ RR^+
9) $\in + RR^* = \in + R^*R = R^*$
10) $(PQ)^*P = P(QP)^*$
11) $(P+Q)^* = (P^*Q^*)^* = (P^* + Q^*)^*$
12) (P+Q)R =PR +QR and
    R(P + Q) =RP +RQ

An Example Proof using Identities of Regular Expressions

Prove that (1+00*1) + (1+00*1) (0+10*1)* (0+10*1) is equal to 0*1(0+10*1)*

LHS = (1+00*1) + (1+00*1) (0+10*1)* (0+10*1)

= (1+00*1) [∈+ (0+10*1)* (0+10*1)]   ∈+R*R = R*

= (1+00*1) (0+10*1)*

=(∈. (1+00*1) (0+10*1)*   ∈.R =R

=(∈+ 00*)1 (0+10*1)*

= O*1 (0+10*1)*=RHS

Design Regular Expression for the following languages over {a,b}

1) Language accepting strings of length exactly 2

2) Language accepting strings of length atleast 2

3) Language accepting strings of length atmost 2

<u>Soln</u>

1) L = {aa, ab, ba, bb}   2) L_1={aa, ab, ba,bb, aaa,…………..}

R= aa+ab+ba+bb   R= (a+b)(a+b) (a+b)*

= a (a+b)+b(a+b)   = (a+b) (a+b)

3) L_1 = {∈, a, b, aa, ab, ba, bb}

R=∈+a+b+aa+ab+ba+bb

= (∈+a+b) (∈+a+b)

2.4 Finite Automata and Regular Expression

2.4.1 From DFA to Regular Expression

Converting a Deterministic Finite Automaton (DFA) to regular expressions is a process that allows you to represent the same language in a more compact and human-readable form. This conversion is useful for various reasons, including simplifying the understanding of the language, facilitating further analysis and manipulation, and providing a more intuitive representation for certain applications.

2.4.2 Why Convert DFA to Regular Expressions:

- **Compact Representation**: Regular expressions are often more concise and easier to understand than DFAs, especially for complex languages. They provide a high-level view of the language without delving into the detailed transitions and states of the automaton.

- **Ease of Communication**: Regular expressions are widely understood and used in computer science and programming. Converting a DFA to regular expressions makes it easier to communicate the language's rules and patterns with others.
- **Simplification of Language Description**: Regular expressions allow you to describe patterns and rules more naturally and intuitively. This can be especially beneficial when dealing with complex languages that may have intricate DFA representations.

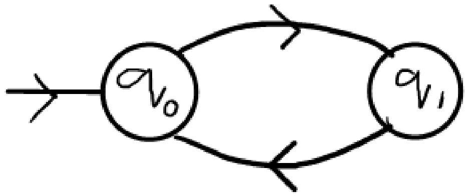
2.4.3 Different methods of DFA to Regular expression

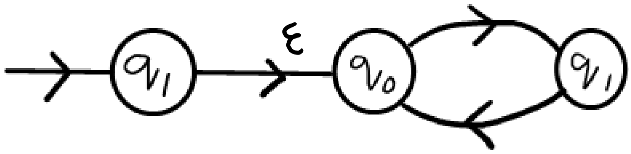
There are several methods to convert a Deterministic Finite Automaton (DFA) to a regular expression. Two common methods are the state elimination method and the use of the Arden's Theorem. Let's briefly discuss both methods:

1. **State Elimination Method**: This method involves eliminating states one by one until only the initial and final states remain. At each step, you replace two states with a regular expression that represents the language accepted by the original DFA between those states.

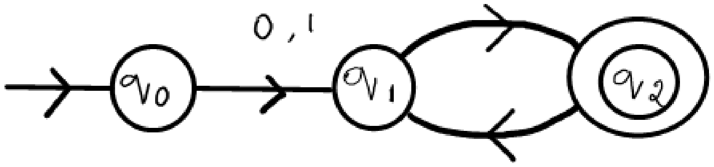
- **Identify Dead-End States**: Identify and eliminate any non-final states that have transitions only to final states.
- **State Elimination**: For each pair of states (q_i, q_j) in the remaining set, find a regular expression that represents the language accepted from q_i to q_j, excluding eliminated states.
- **Repeat**: Repeat the process until only the initial and final states remain, and you have a regular expression representing the entire language.

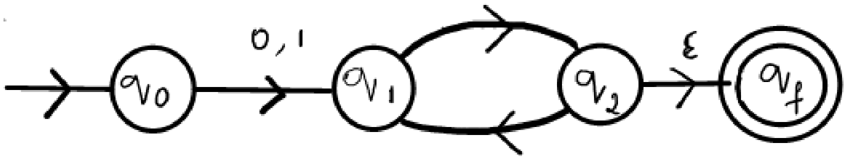
STEP 1: if there exists any incoming edge to initial state add a new initial state



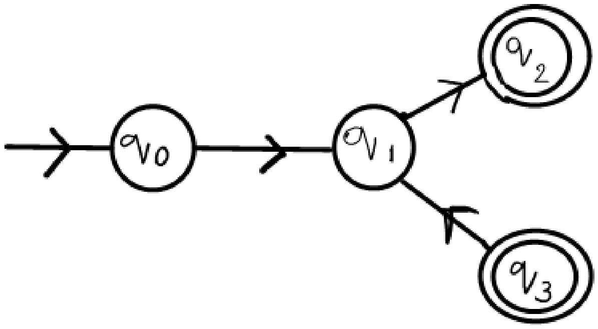


Step 2: if there exist an outgoing edge to final state create a new final state

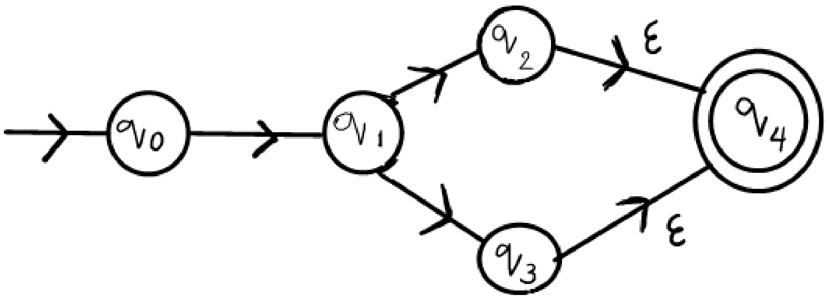




Step 3: if there exist multiple final states convert into non-final state and create a single final state

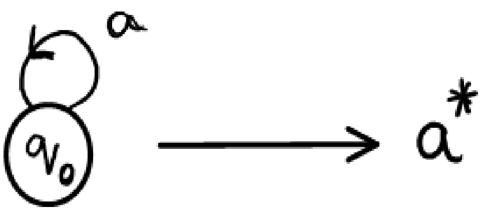


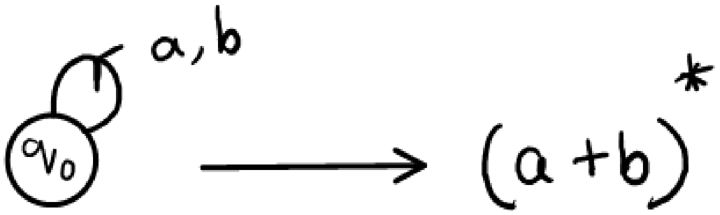
Step 4: Eliminate all intermediate states one by one in any order

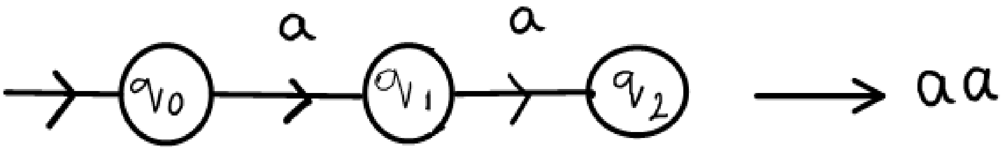


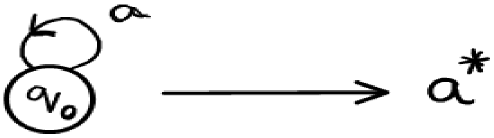
Step 4: Eliminate all intermediate states one by one in any order

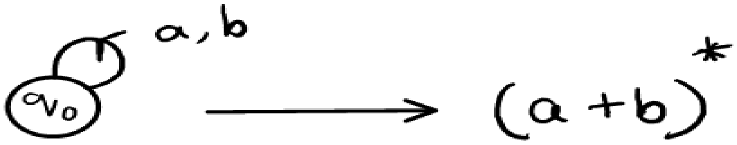
Background: converting finite automata to regular expression

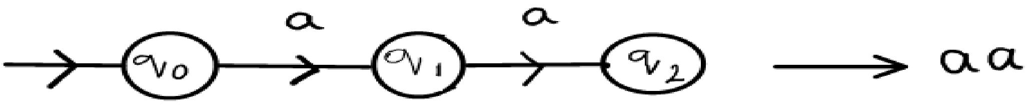




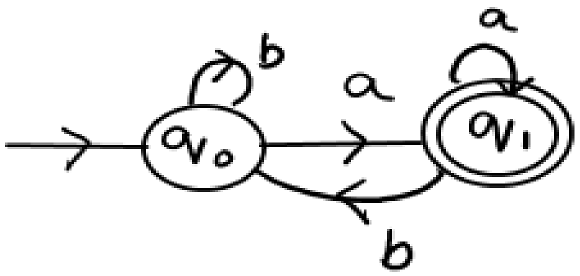




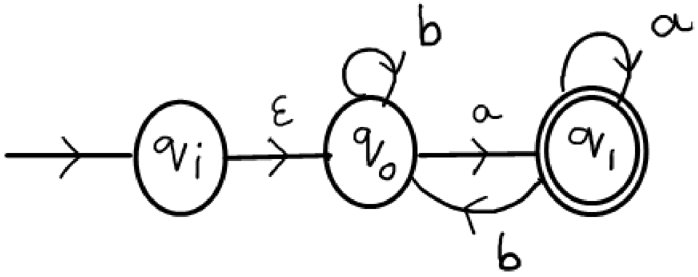




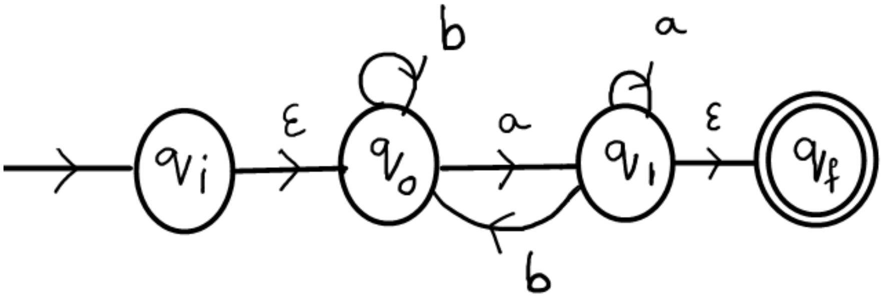
1. **Convert Following DFA to Regular expression**



Add Initial state

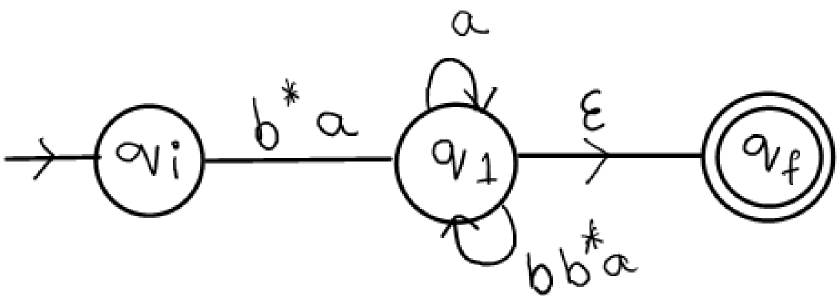


Add final state

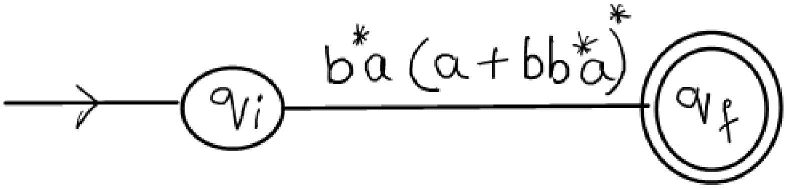


Approach 1: (from initial state)

Eliminate state q_0

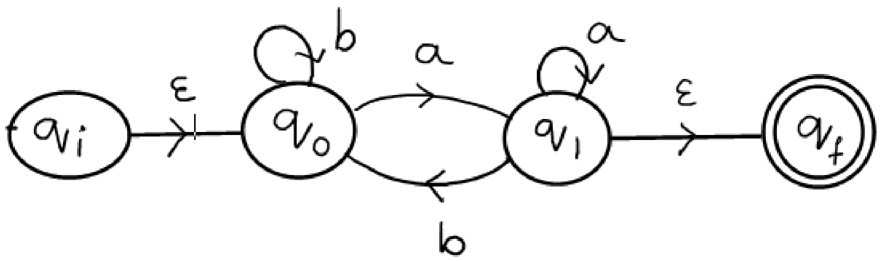


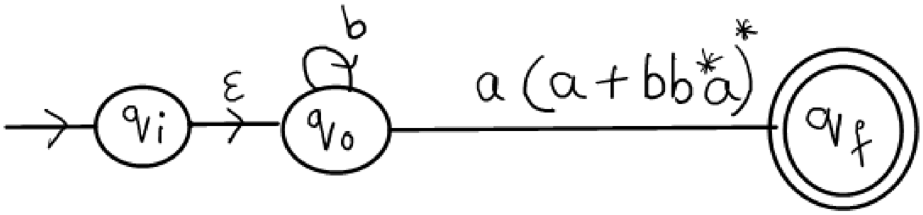
Eliminate state q_1



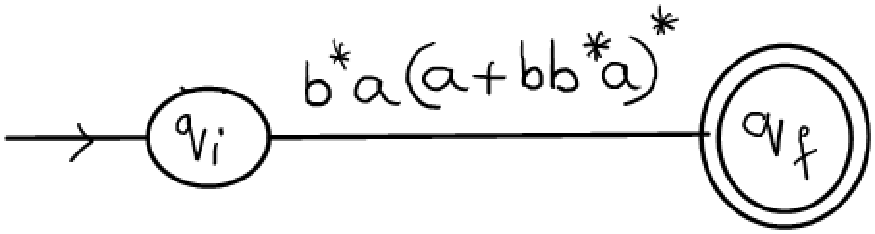
Approach 2: (from final state)

Eliminate q1

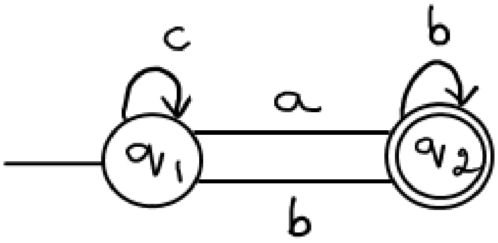




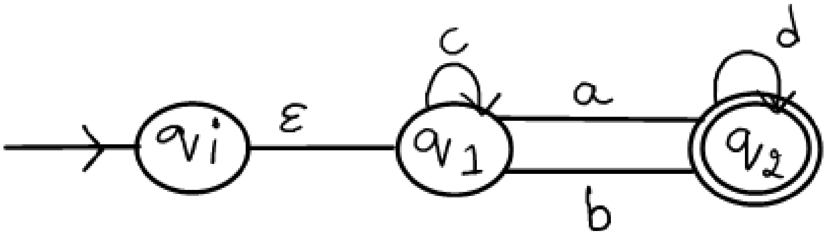
Eliminate state q_0



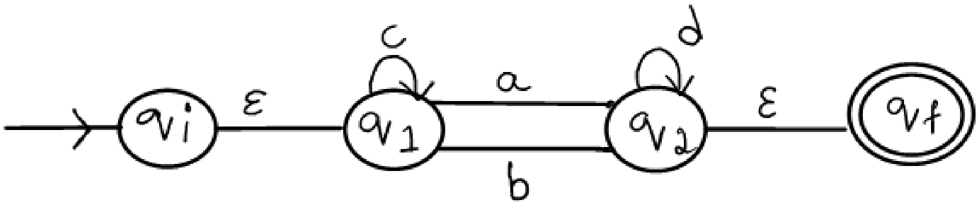
2. Construct Regular Expression using following DFA



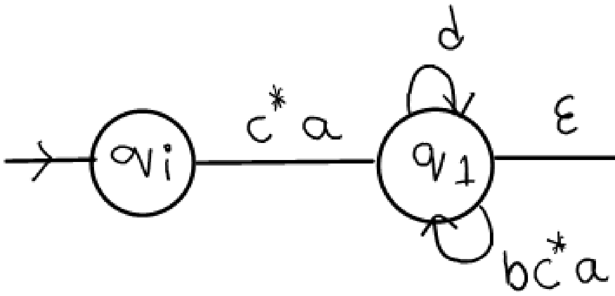
Step 1 – Initial state q1 has incoming edge. So, create a new initial state qi.



Step 2 – Final state q2 has outgoing edge. So, create a new final state.



Step 3 – Start eliminating intermediate states one after another.
Eliminate q1



Now eliminate q2.

After eliminating q2 direct path from state qi to qf having cost.

c*a(d+bc*a) * ε=**c*a(d+bc*a)***

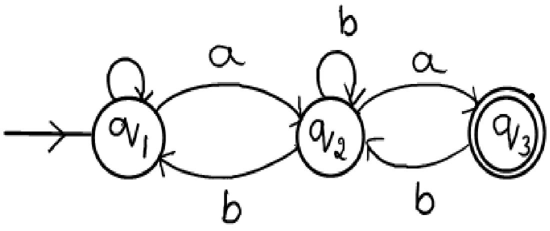
2. **Arden's Theorem Method**:

Arden's Theorem provides a way to find the solution to a system of linear equations involving regular expressions. This method can be applied to express the regular language accepted by a DFA.

Steps:

- **Set up Equations**: For each state q_i, set up an equation expressing the regular expression representing the language accepted by q_i in terms of the regular expressions of its neighbour's.
- **Solve Equations**: Use Arden's Theorem to solve the system of equations and find the regular expression for each state.
- **Combine Expressions**: Combine the regular expressions obtained for each state to get the overall regular expression for the language accepted by the DFA.

1. Design or Find Regular expression for the following NFA (Using Arden's theorem)



$q_3 = q_2 a \longrightarrow$ ①

$q_2 = q_1 a + q_2 b + q_3 b \longrightarrow$ ②

$q_1 = \varepsilon + q_1 a + q_2 b \longrightarrow$ ③

① $\rightarrow$ $q_3 = q_2 a$

$= (q_1 a + q_2 b + q_3 b) a$

$= q_1 aa + q_2 ba + q_3 ba \longrightarrow$ ④

2) $q_2 = q_1 a + q_2 b + q_3 b$ Putting value of q_3 from — ①

$= q_1 a + q_2 b + (q_2 a) b$

$= q_1 a + q_2 b + q_2 ab$

$\underbrace{q_2}_{R} = \underbrace{q_1 a}_{Q} + \underbrace{q_2}_{R} \underbrace{(b+ab)}_{P}$

$R = Q + RP$ Arden's theorem

$R = QP$

$q_2 = (q_1 a)(b+ab)^* \longrightarrow$ ⑤

$$③ \hookrightarrow q_1 = \epsilon + q_1 a + q_2 b$$

Putting value of q_2 from ⑤

$$q_1 = \epsilon + q_1 a + ((q_1 a)(b+ab)^*) b$$

$$\underbrace{q_1}_{R} = \underbrace{\epsilon}_{Q} + \underbrace{q_1}_{R} \underbrace{(a+a(b+ab)^*)}_{P} b$$

$$R = Q + RP$$

$$R = QP^*$$

$$\epsilon . R = R$$

$$q_1 = \epsilon ((a+a(b+ab)^*)b)^*$$

$$q_1 = (a+a(b+ab)^* b)^* \longrightarrow ⑥$$

Final state q_3

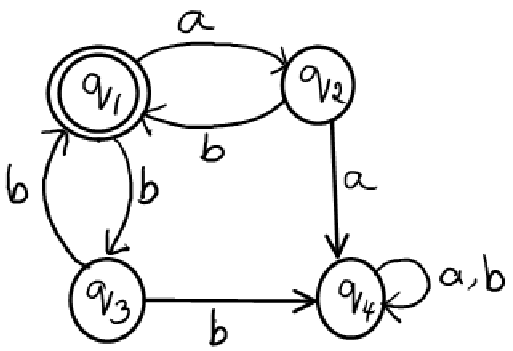
$q_3 = q_2 a$

$= q_1 a(b+ab)^*a$ putting value of q_2 from 5

$q_3 = (a+a(b+ab)^*b)^* \; a(b+ab)^*a$ putting value of q_1 from 6

= Required Regular expression for the given NFA

2. Design the Regular expression for the following DFA



$$q_1 = \epsilon + q_2 b + q_3 a \longrightarrow ①$$

$$q_2 = q_1 a \longrightarrow (II)$$

$$q_3 = q_1 b \longrightarrow (III)$$

$$q_4 = q_2 a + q_3 b + q_4 a + q_4 b \longrightarrow (IV)$$

(I) ↳ $q_1 = \epsilon + q_2 b + q_3 a$

Putting values of q_2 and q_3 from (II) and (III)

$q_1 = \epsilon + q_1 ab + q_1 ba$

$\underbrace{q_1}_{R} = \underbrace{\epsilon}_{Q} + \underbrace{q_1}_{R} \underbrace{(ab+ba)}_{P}$

$q_1 = \epsilon \, (ab+ba)^*$

$q_1 = (ab+ba)^*$

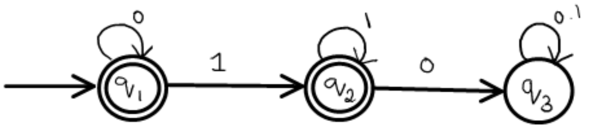
$R = Q + RP$

$R = QP^*$ Arden's Theorem

$\epsilon R = R$

3 Design Regular expression for following when there are multiple Final states

Find the Regular Expression for the following DFA



$q_1 = \epsilon + q_1 0 \rightarrow$ (I)

$q_2 = q_1 1 + q_2 1 \rightarrow$ (II)

$q_3 = q_2 0 + q_3 0 + q_3 1 \rightarrow$ (III)

Final state q_1

(I) ↳ $\underbrace{q_1}_{R} = \underbrace{E}_{Q} + \underbrace{q_1}_{R} \underbrace{0}_{P}$

$R = Q + RP$

$R = QP^*$ Arden's theorem

$ER = R$

$q_1 = E . 0^*$

$q_1 = 0^* \rightarrow$ (4)

Final state $\textcircled{q_2}$

$$q_2 = q_1 1 + q_2 1$$

$$\underbrace{q_2}_{h} = \underbrace{0^* 1}_{Q} + \underbrace{q_2}_{R}\underbrace{1}_{P} \qquad \text{Putting value of } q_1 \text{ from } \textcircled{4}$$

$$R = Q + RP$$

$$R = QP^*$$

$$q_2 = 0^* 1 (1)^*$$

h= union of both Final states

$$= 0^* + 0^* 1 1^*$$

$$= 0^* (\epsilon + 1 1^*) \qquad \epsilon + RR^* = R^*$$

$$= 0^* 1^*$$

2.3.4 Regular Expression to Finite Automata

The conversion of a regular expression to a finite automaton is a fundamental concept in formal language theory. The importance of this conversion lies in the fact that it provides a systematic way to represent and understand the relationship between regular languages (described by regular expressions) and the corresponding machines (finite automata)

The importance of converting regular expressions to finite automata:

1. **Equivalence**: The regular expression and finite automaton representations are equivalent in terms of expressive power; that is, they describe the same class of languages. This equivalence demonstrates that regular languages can be recognized by both regular expressions and finite automata.

2. **Understanding Language Recognition**: The conversion process helps in understanding how certain patterns and structures in regular expressions can be translated into the operational behavior of a finite automaton. It provides insights into the relationship between the syntactic and semantic aspects of regular languages.

3. **Algorithmic Recognition**: Finite automata are often used as the underlying mechanism for recognizing regular languages in computer science and automata theory. Algorithms for pattern matching, lexical analysis (e.g., in compilers), and string searching often involve finite automata. By converting regular expressions to finite automata, we can implement efficient algorithms for recognizing patterns in strings.

4. **Tool for Language Specification**: Regular expressions are widely used for specifying patterns in various contexts, such as text processing, search algorithms, and lexical analysis. Finite automata provide a

concrete and implementable representation of these patterns, allowing developers to build systems that recognize and process languages described by regular expressions.

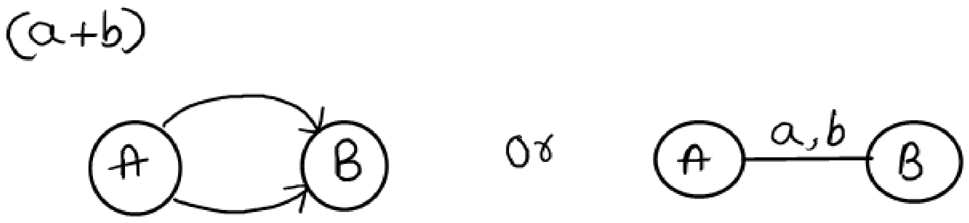
5. **Optimization and Implementation**: The conversion process can reveal opportunities for optimizing the recognition process. Understanding the finite automaton corresponding to a regular expression can lead to the elimination of redundant states or transitions, resulting in more efficient implementations.

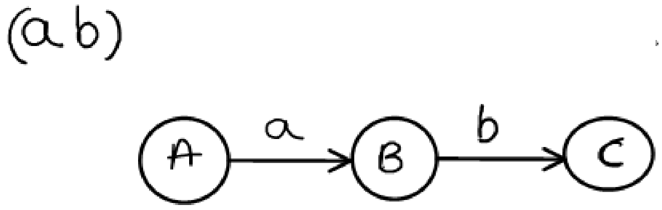
6. **Educational Purposes**: Learning about the conversion from regular expressions to finite automata is a common topic in courses on formal languages and automata theory. It serves as a foundational concept for understanding more complex language classes and machine models.

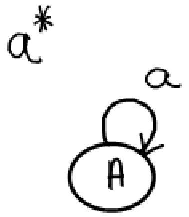
7. **Automation in Software Development:** Tools and libraries that automate the conversion from regular expressions to finite automata are valuable for software developers. These tools enable efficient implementation of language recognition mechanisms and are used in various applications, including text processing and validation.

In summary, the conversion of regular expressions to finite automata is a crucial step in the study of formal languages and provides a bridge between theoretical language descriptions and practical language recognition implementations in computer science and software development.

Important points:



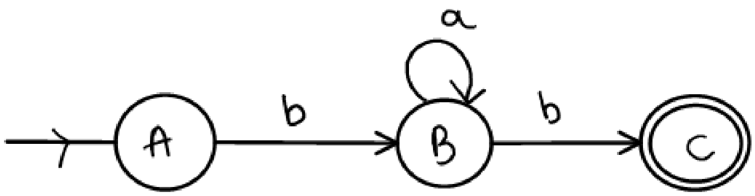




1. **Convert following Regular expression to their Finite Automata**
   a) **ba*b**
   b) **(a+b)c**
   c) **a(bc)***

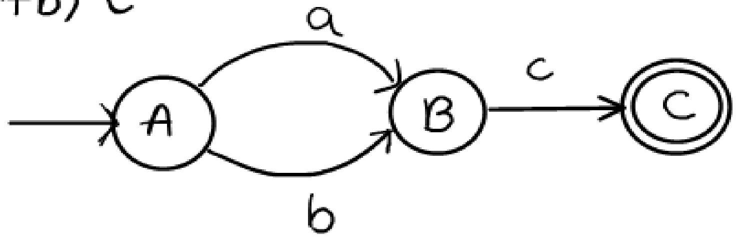
a) **b a* b :**

1) ba^*b bb, bab, baab



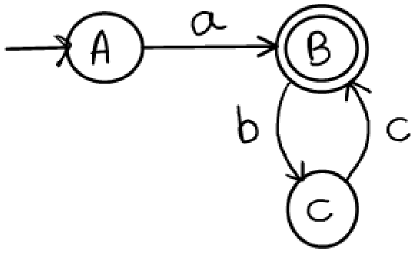
b) (a+b).c

(a+b) c

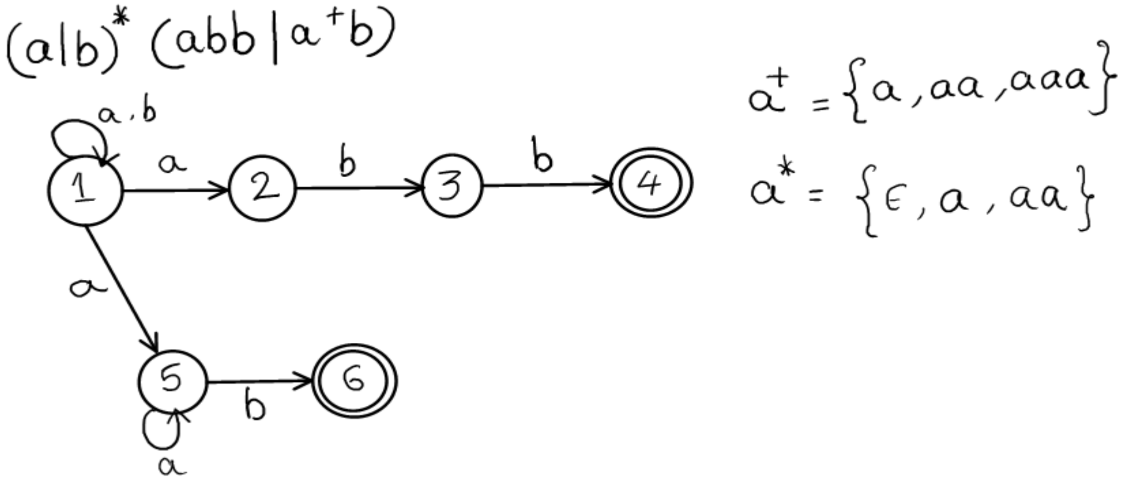


c) a(bc)*8

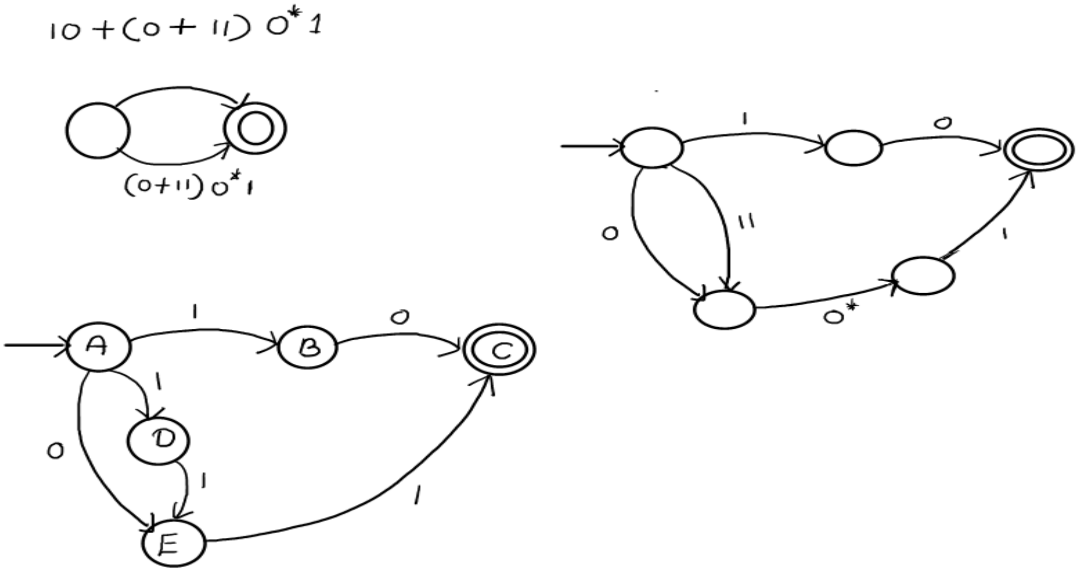
3) $a(bc)^*$ a, abc, abcbc, abcbcbc



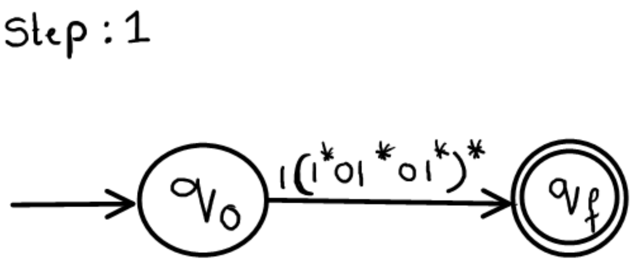
6) (a | b)* (abb|a$^+$b)

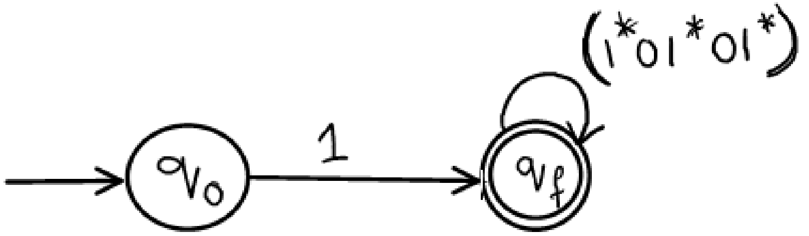


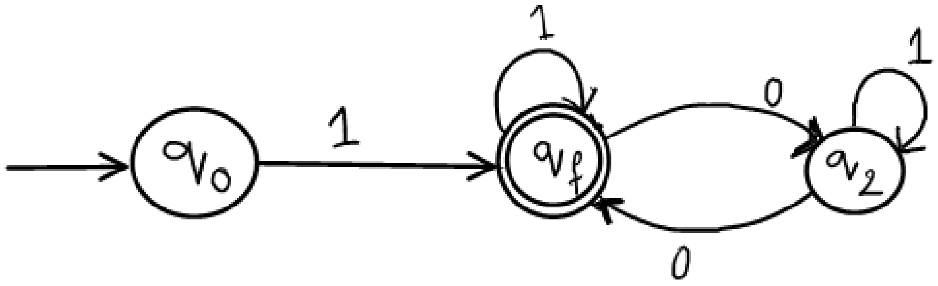
7) 10+(0+11)0*1



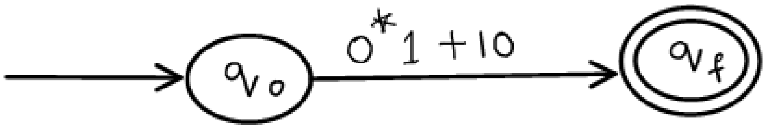
8). 1 (1* 01* 01*)*.

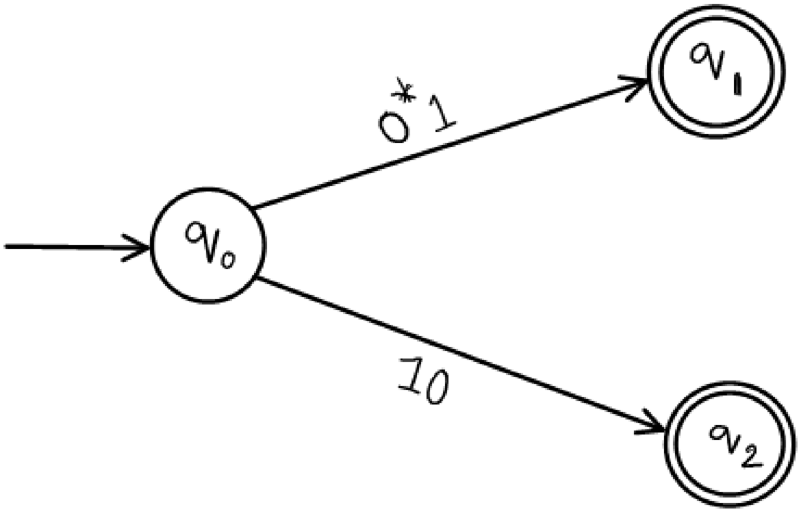


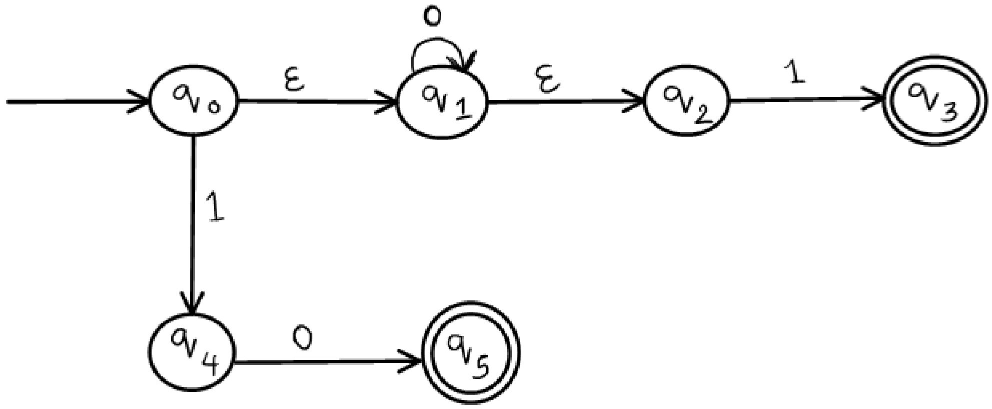


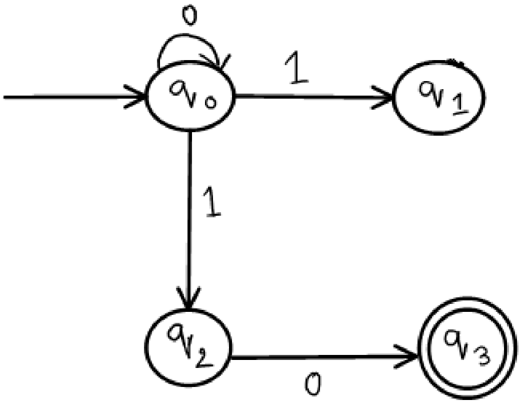


9) 0*1+10









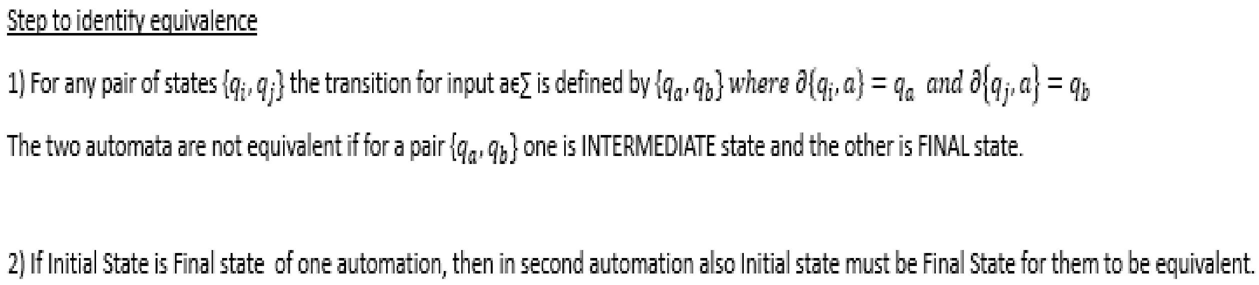
2.3.5 *Equivalence of two finite automata*

Equivalence of two finite automata means that the two automata recognize the same language. In other words, they accept the same set of strings. Checking the equivalence of finite automata is a crucial task in various areas of computer science and formal language theory. Here are some reasons why one might need to check the equivalence of finite automata:

- **Language Equivalence**: The primary reason for checking equivalence is to determine whether two finite automata recognize the same language. If two automata are equivalent, it means they accept the same set of strings. This is essential for ensuring that different implementations or representations of a language are consistent.
- **Program Verification**: In the field of software engineering, finite automata and regular languages are often used to model certain behaviors or specifications of programs. Checking the equivalence of automata can be part of program verification processes to ensure that different implementations of a system behave identically in terms of the recognized language.

- **Compiler Construction**: In compiler construction, finite automata are commonly used to model lexical analyzers (scanners). It is crucial to verify that the lexical analyzers generated by different compiler tools or components are equivalent, ensuring consistent token recognition.
- **Optimization**: Checking equivalence is relevant when optimizing finite automata. For example, if you have multiple representations of the same language, you might want to check if they are equivalent and choose the most efficient one. Minimization algorithms are often applied to reduce the size of an automaton while preserving its language.
- **Automata Learning**: In the context of machine learning and automata learning, equivalence testing is used to verify if a learned automaton is equivalent to a target automaton. This is essential for validating the correctness of the learned model.
- **Formal Language Theory**: Equivalence testing is a fundamental concept in formal language theory. It helps researchers and practitioners understand the relationships between different automata models and their expressive power.
- **Security Protocols**: Finite automata are used in security protocols to model the behavior of systems and attackers. Checking the equivalence of models is crucial in security analysis to ensure that the system and the attacker's model are consistent.
- **Database Query Optimization**: In the context of query optimization, finite automata are sometimes used to represent query languages. Checking equivalence can help optimize queries by choosing equivalent but more efficient representations.

In summary, checking the equivalence of finite automata is a fundamental and versatile concept with applications in various domains, including software engineering, formal methods, compiler construction, and security analysis. It ensures consistency and correctness in language recognition, which is essential in many computational and theoretical contexts.



Step to identify equivalence

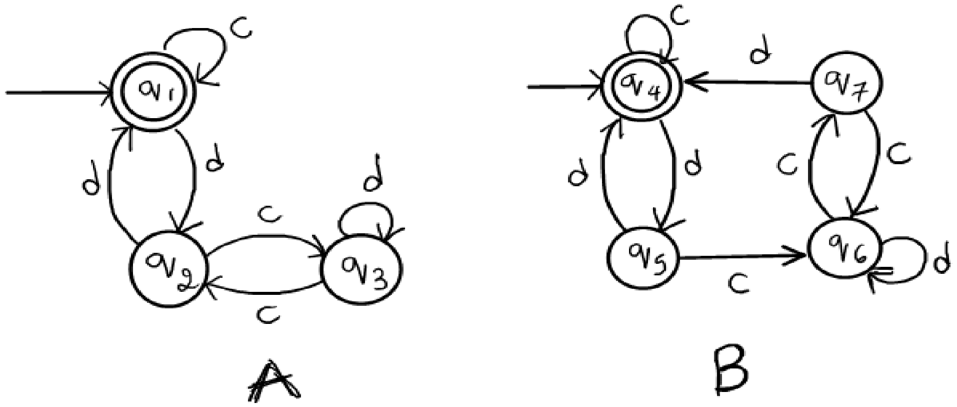
1) For any pair of states $\{q_i, q_j\}$ the transition for input a∊∑ is defined by $\{q_a, q_b\}$ $where\ \partial\{q_i, a\} = q_a\ \ and\ \partial\{q_j, a\} = q_b$

The two automata are not equivalent if for a pair $\{q_a, q_b\}$ one is INTERMEDIATE state and the other is FINAL state.

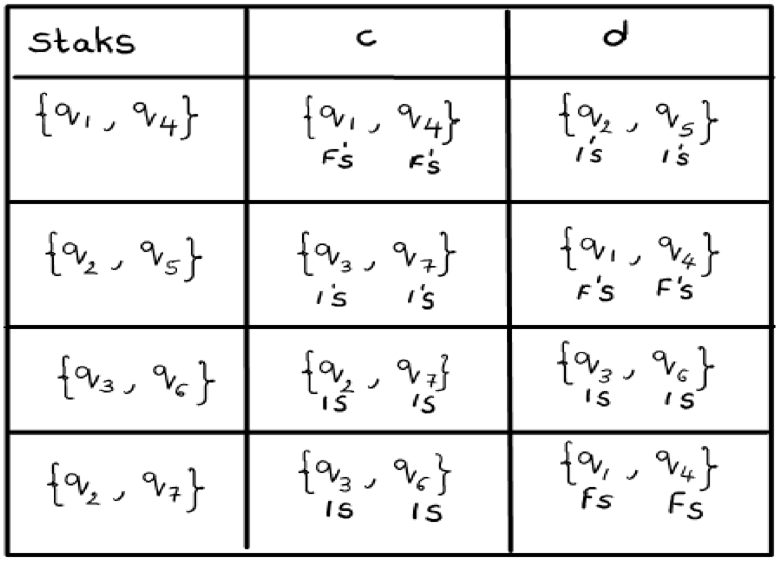
2) If Initial State is Final state of one automation, then in second automation also Initial state must be Final State for them to be equivalent.

1. **Check the two following two finite automata are equivalent or not**

Example:



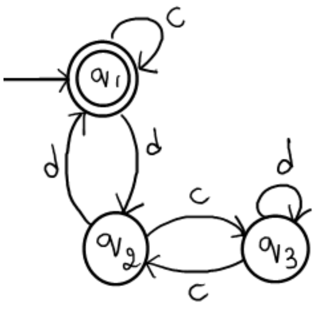
Following are the evaluation



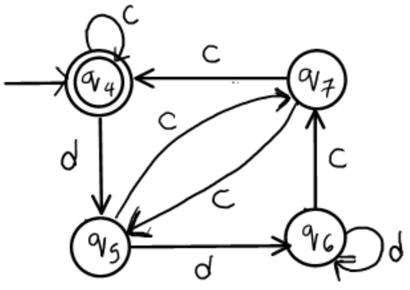
Staks	c	d
$\{q_1, q_4\}$	$\{q_1, q_4\}$ F's F's	$\{q_2, q_5\}$ I's I's
$\{q_2, q_5\}$	$\{q_3, q_7\}$ I's I's	$\{q_1, q_4\}$ F's F's
$\{q_3, q_6\}$	$\{q_2, q_7\}$ IS IS	$\{q_3, q_6\}$ IS IS
$\{q_2, q_7\}$	$\{q_3, q_6\}$ IS IS	$\{q_1, q_4\}$ FS FS

A and B are equivalent

2. Check the two following two finite automata are equivalent or not



A



B

Solution:

Staks	c	d
$\{q_1, q_4\}$	$\{q_1, q_4\}$ F's F's	$\{q_2, q_5\}$ I's I's
$\{q_2, q_5\}$	$\{q_3, q_7\}$ I's I's	$\{q_1, q_6\}$ F's I's

A and B are not equivalent

2.4 *Proving languages not to be regular*

Regular languages have several properties that distinguish them from other classes of languages. Here are some key properties of regular languages:

- **Closure under Union (OR):** If L1 and L2 are regular languages, then their union L1∪L2 is also a regular language.
- **Closure under Intersection (AND):** If L1 and L2 are regular languages, then their intersection L1∩L2 is also a regular language.
- **Closure under Complementation (NOT):** If L is a regular language, then its complement ‾L (the set of all strings not in L) is also a regular language.
- **Closure under Concatenation:** If L1 and L2 are regular languages, then their concatenation ·L2 (the set of all strings formed by concatenating a string from L1 with a string from L2) is also a regular language.
- **Closure under Kleene Star:** If L is a regular language, then its Kleene closure L∗ (the set of all strings formed by concatenating zero or more strings from L) is also a regular language.
- **Closure under String Reversal**: If L is a regular language, then its reversal LR (the set of all strings whose reversal is in L) is also a regular language.

- **Existence of a Finite Automaton:** A language is regular if and only if there exists a finite automaton that recognizes it. This is the defining characteristic of regular languages.
- **Pumping Lemma**: Regular languages satisfy the Pumping Lemma, which provides a property used in proving that certain languages are not regular.
- **Equivalence to Regular Expressions**: A language is regular if and only if there exists a regular expression that generates it. Regular expressions and finite automata are two equivalent ways to describe regular languages.
- **Decidability**: Properties of regular languages are decidable. This means that, for any statement about regular languages, there exists an algorithm that can determine whether the statement is true or false.
- These properties make regular languages a well-defined and well-behaved class of languages, and they serve as the foundation for the study of formal languages and automata theory

2.4.4 Pumping Lemma:

The Pumping Lemma typically states that for any regular language, there exists a constant (the "pumping length") such that any strings in the language of length at least p can be divided into three parts, xyz, satisfying the following conditions:

1. For each i≥0, the string xy^iz is in the language.
2. |y|>0 (i.e., y is non-empty).
3. |xy|≤p (the length of xy is at most p).

The idea is that, because the language is regular, repeating or "pumping" the middle part y should still produce strings in the language.

To use the Pumping(substring of a string repeated) Lemma(substring) to show that a language is not regular, one assumes, for the sake of contradiction, that the language is regular and then chooses a specific string in the language that violates the conditions of the Pumping Lemma. This contradiction implies that the assumption that the language is regular must be false, and therefore the language is not regular.

It's important to note that the Pumping Lemma is specific to regular languages. There are other lemmas and techniques used for proving properties of context-free languages and other language classes. The Pumping Lemma is a powerful tool, but its application requires careful reasoning and understanding of the properties of regular languages. **In layman's understanding if the substring of a string is repeated many times and if the resultant of string is available in the language then we can say that it as REGULAR.**

STEPS :

- Consider a language as regular
- Assume constant 'c' and select the 'w' string such that |w| >= C
- Divide w as XYZ
- |Y|>0
- |XY| <=C
- For i>=0 every string xy^{iz} belong to L

1. **Prove that {$a^n b^n$ | n>=0} is not Regular**

Solution:

Let us consider the language L={ ε,ab, aabb,aaabbb,aaaabbbbb,aaaaabbbbb,….}

Assume C = 6

For instance , w =aaabbb |w| >=C

$w= \underbrace{a\ a}_{x}\ \underbrace{a\ b}_{y}\ \underbrace{b\ b}_{z}$ $\qquad\qquad$ $\underbrace{a}_{x}\ \underbrace{a\ a\ b}_{y}\ \underbrace{b\ b}_{z}$

x = aa y= ab z=bb

|y|=2 |xy| =aaab =4 < C

For instance I=0 , xy^2z =aaababbb – not available in L (NO. of a's are not followed b in sequence)

For instance , w =aaabbb |w| >=C

In this x =a y=aa z=bbb C=6

|Y| = 2 >0, |XY| = 3 < C ,

When i=2

XY^0Z = aaa not in our Language L

XY^2Z = aaaaabbb not in our Language L

{$a^n b^n$} is not a regular

2. Using Pumping Lemma prove that the language A={YY|Y ∈ {0,1}* } is not Regular

Solution :

Y=ϵ: YY=ϵϵ=ϵ

Y="0" YY="00"

Y="10" YY="1010"

Consider L= {00,0000,01010,0000100001.}

C = 7

For instance , w= 0000100001|w| >=C

In this i=0, x=00 z=00001 = {0000001} is not Regular

When i=1, x=0 y=0001 z=00001 $|XY^1Z|$ = 00000100001 -not repeating properly so it is not regular

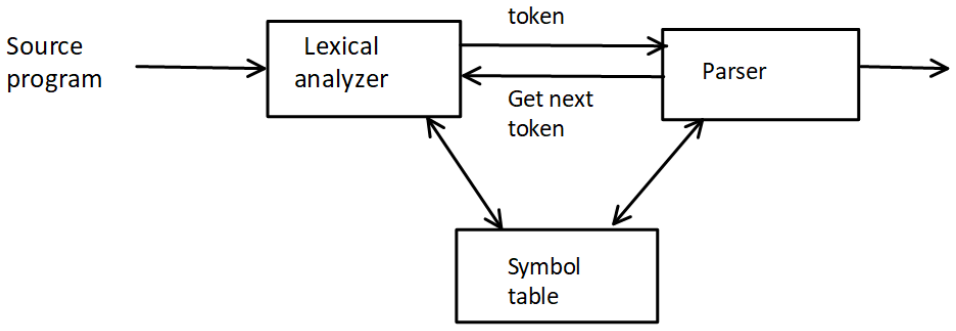
2.5 Lexical Analysis Phase and compiler design

2.5.4 Role of Lexical Analyzer

The lexical analyzer, also known as the lexer or scanner, is a crucial component of the compiler responsible for processing the input source code and generating a stream of tokens. Its role is primarily focused on the lexical analysis phase, which is the initial step in the compilation process.

- ✓ The main task of the lexical analyzer is to
  - ○ **read the input characters** of the source program,
  - ○ **group** them into **lexemes**, and
  - ○ produce as **output a sequence of tokens** for the source program.
  - ○ **stripping out comments and whitespace** (blank, newline, tab etc), that are used to separate tokens in the input.

- ✓ Parser invokes the lexical analyzer by ***getNextToken*** command
- ✓ Lexical analyzer reads the characters from input until it finds the next lexeme and produce token



Here are the key roles and functions of the lexical analyzer:

- **Tokenization:** The primary function of the lexical analyzer is to break the source code into a sequence of tokens. Tokens are the smallest meaningful units of the programming language, such as keywords, identifiers, literals, and operators. Tokenization simplifies the subsequent phases of the compiler by representing the source code in a more structured and manageable form.

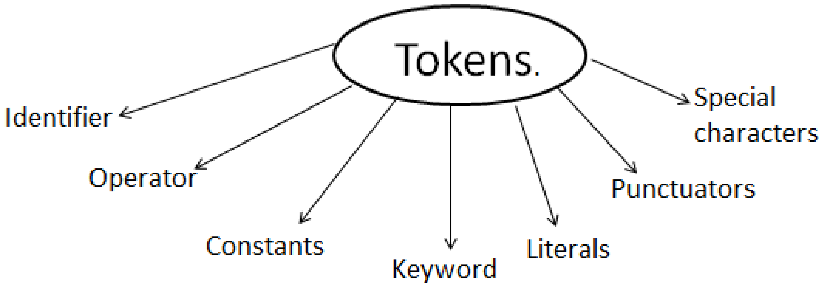
- **Ignoring Whitespaces and Comments:** The lexical analyzer filters out irrelevant elements such as whitespaces and comments from the source code. These elements are important for human readability but do not contribute to the execution of the program. Removing them reduces the complexity of the subsequent phases.

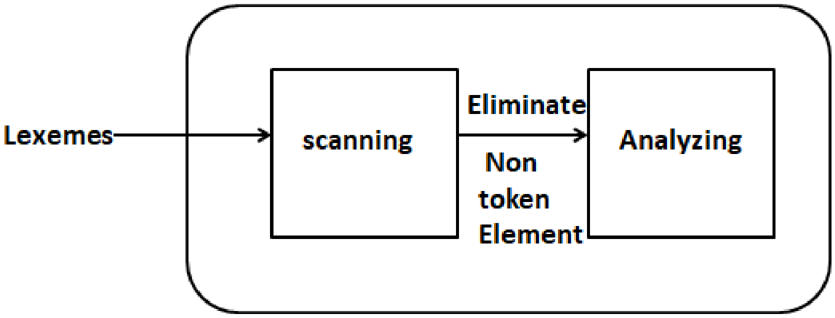
- **Error Detection**: The lexical analyzer plays a role in detecting lexical errors in the source code. Lexical errors include issues such as invalid characters or tokens that do not conform to the language's syntax. Detecting errors early in the compilation process helps provide meaningful error messages to the programmer for debugging.

- **Symbol Table Generation:** As the lexical analyzer processes the source code, it may build a symbol table. The symbol table is a data structure that keeps track of identifiers (variable names, function names, etc.) encountered in the source code along with their attributes, such as data type or memory location. This information is crucial for later phases of the compiler, particularly in semantic analysis.
- **Pattern Matching:** The lexical analyzer uses regular expressions and finite automata to perform pattern matching on the source code. Each token type is associated with a specific pattern, and the lexical analyzer identifies and categorizes tokens based on these patterns.
- **Generating Output for Parser:** The output of the lexical analyzer is a stream of tokens, which is then passed on to the next phase of the compiler, the parser. The parser uses this token stream to build a parse tree or an abstract syntax tree, representing the syntactic structure of the program.
- **Efficiency and Optimization:** Lexical analyzers are designed to be efficient, as they are the first phase in the compilation process. Techniques such as finite automata and regular expressions are employed to optimize the scanning process and reduce the time complexity of token recognition.

Lexical Analyzer

- Scans the pure HLL code line by line.
- Takes Lexemes as i/p and produces Tokens.





Sample source code :

2.5.5 Tokens, Patterns and Lexemes

Lexeme: A lexeme is a basic unit of lexical analysis and represents a sequence of characters in the source program that corresponds to a single token. Tokens are the smallest units in a programming language, and lexemes are the actual instances of these tokens in the source code.

consider the following line of code in a simple programming language:

x = 10 + y

In this line, there are several lexemes, each representing a distinct token:

x is a lexeme representing an identifier.

= is a lexeme representing the assignment operator.

10 is a lexeme representing a numeric constant.

+ is a lexeme representing the addition operator.

y is a lexeme representing another identifier.

So, in short, lexemes are the fundamental building blocks identified during lexical analysis, and they help break down the source code into meaningful units for further processing by the compiler or interpreter

Pattern: In lexical analysis, a "pattern" usually refers to a rule or a template that describes the structure of a token. Tokens are the basic building blocks of a programming language, and patterns help identify and classify these tokens in the source code.

Let's break it down in simpler terms:

Token: Think of a token as a meaningful unit in a programming language. For example, in the code int x = 5;, the tokens include the keywords int, x, =, and 5.

Pattern: A pattern is a set of rules that defines what a token looks like. For instance, a simple pattern for an integer constant could be "one or more digits." So, the pattern for the token 5 is "digit(s)."

Here's an example:

Consider the following line of code:

total = count * 10;

In this line, there are several tokens, each following a specific pattern:

Identifier (total): This follows the pattern of starting with a letter and followed by letters or digits.

Assignment Operator (=): This is a simple pattern of the equal sign.

Identifier (count): Similar to the first identifier, it follows the pattern of starting with a letter and followed by letters or digits.

Multiplication Operator (*): This is a single-character pattern representing the multiplication operation.

Integer Constant (10): This follows the pattern of one or more digits.

Semicolon (;): This is a single-character pattern representing the end of a statement.

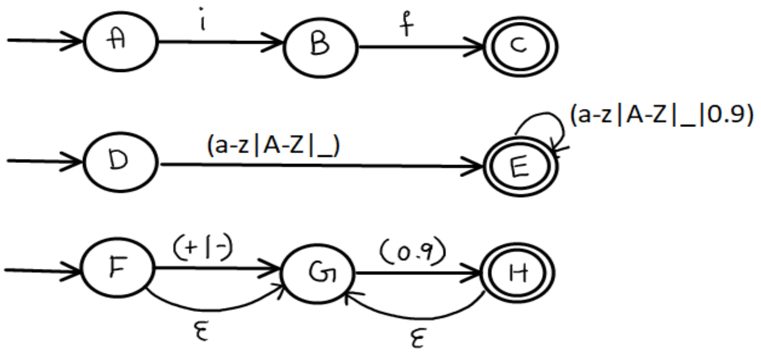
So, in lexical analysis, patterns help define the rules for recognizing and extracting different types of tokens from the source code. They are crucial for breaking down the code into meaningful elements for further processing by the compiler or interpreter

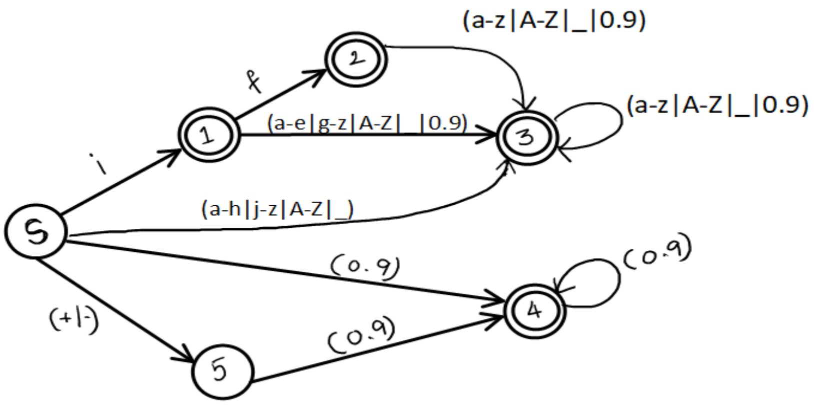
Token: It is a pair consisting of a token name and an optional attribute value. a "token" in lexical analysis is like a building block of a programming language. It's a meaningful chunk or piece of the code that represents a fundamental unit.

- The token name as an abstract symbol represents the kind of lexical unit/lexeme(keyword/identifier, operator symbol etc)
- Processed by parser

In the statement C, printf("Total =%d\n", score)

Printf ,score -> lexmes matching the pattern for token id, "Total="%d\n" is lexme matching literal





DFA

Lexical Analyzer-Tokenization:

```
int main()
{
int x , a=2,b=3,c=5;
x = a + b*c;
 Print f ("The value of x is %d", x);
return 0;
}
```

Count: 39

Tokens:

1. Keyword: int, return

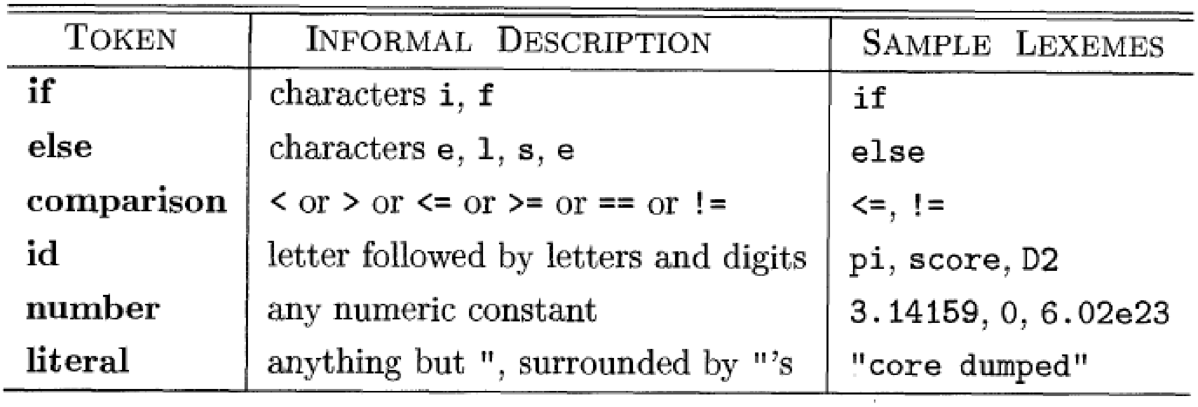
2. Identifier: main, x ,a ,b ,c , print f

3. Punctuator: (,) ,{,,,,;}

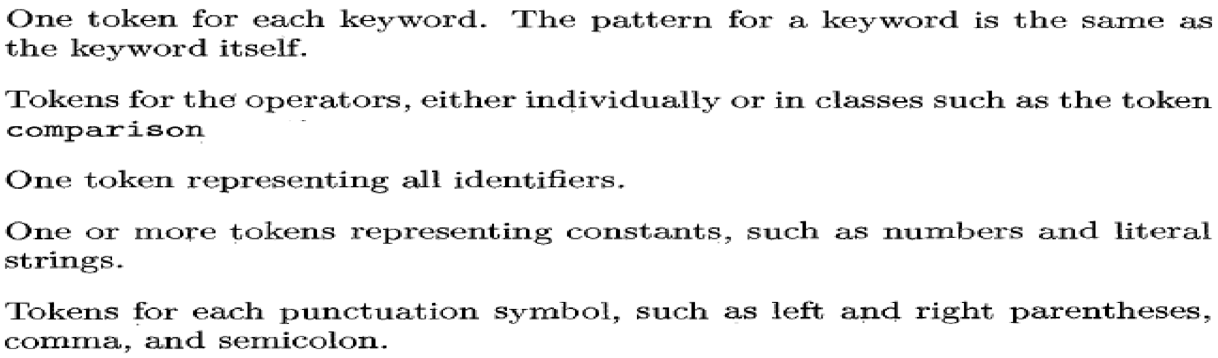
4. Operator: =,+,*

5. Constant: 2,3,5,0

6. Literal: "The value of x is %d"



TOKEN	INFORMAL DESCRIPTION	SAMPLE LEXEMES
if	characters `i`, `f`	`if`
else	characters `e`, `l`, `s`, `e`	`else`
comparison	`<` or `>` or `<=` or `>=` or `==` or `!=`	`<=`, `!=`
id	letter followed by letters and digits	`pi`, `score`, `D2`
number	any numeric constant	`3.14159`, `0`, `6.02e23`
literal	anything but ", surrounded by "'s	`"core dumped"`



One token for each keyword. The pattern for a keyword is the same as the keyword itself.

Tokens for the operators, either individually or in classes such as the token comparison

One token representing all identifiers.

One or more tokens representing constants, such as numbers and literal strings.

Tokens for each punctuation symbol, such as left and right parentheses, comma, and semicolon.

SAMPLE CODE IN PYTHON : Construction of Symbol table

```
import re
# Token types
TOKEN_INT = 'INT'
TOKEN_FLOAT = 'FLOAT'
TOKEN_IDENTIFIER = 'IDENTIFIER'
TOKEN_OPERATOR = 'OPERATOR'
TOKEN_KEYWORD = 'KEYWORD'
TOKEN_SEPARATOR = 'SEPARATOR'
# Regular expressions for token recognition
regex_patterns = [
    (r'\bint\b|\bfloat\b|\bchar\b', TOKEN_KEYWORD),
    (r'\bif\b|\belse\b|\bwhile\b', TOKEN_KEYWORD),
    (r'\breturn\b', TOKEN_KEYWORD),
    (r'\b\d+(\.\d+)?\b', TOKEN_FLOAT),
    (r'\b[a-zA-Z_]\w*\b', TOKEN_IDENTIFIER),
    (r'\+|\-|\*|\/|>|=|\(', TOKEN_OPERATOR),
    (r'\)|\{|\}|\;|\,', TOKEN_SEPARATOR),
]

# Symbol table
symbol_table = {}

def getNextToken(input_string):
    if not input_string:
        return None, input_string    # Return None for an empty string
    for pattern, token_type in regex_patterns:
        regex = re.compile(pattern)
        match = regex.match(input_string)
        if match:
            value = match.group(0)
            return {'lexeme': value, 'type': token_type}, input_string[len(value):].lstrip()
    raise Exception(f'Unexpected character: {input_string[0]}')

def parse(source_code):
    global symbol_table
    tokens = []
    while source_code:
        token, source_code = getNextToken(source_code)
        tokens.append(token)
        if token['type'] == TOKEN_KEYWORD and token['lexeme'] in ['int', 'float', 'char']:
            # Handle variable declarations and update symbol table
            var_type = token['lexeme']
            var_name, source_code = getNextToken(source_code)
            if var_name['type'] == TOKEN_IDENTIFIER:
                symbol_table[var_name['lexeme']] = var_type
            else:
                raise Exception(f'Expected identifier after {var_type}, got {var_name["lexeme"]}')

        elif token['type'] == TOKEN_KEYWORD and token['lexeme'] == 'return':
            # Handle 'return' keyword (additional parsing logic can be added here)
            return_expression, source_code = getNextToken(source_code)
            if return_expression['type'] == TOKEN_IDENTIFIER:
                if return_expression['lexeme'] not in symbol_table:
                    raise Exception(f'Undeclared identifier in return statement: {return_expression["lexeme"]}')
            # Additional parsing logic for the return statement can be added here
        elif token['type'] == TOKEN_IDENTIFIER:
            # Handle variable usage (e.g., assignments) and check symbol table
            if token['lexeme'] not in symbol_table:
                raise Exception(f'Undeclared identifier: {token["lexeme"]}')
            # Additional parsing logic for assignments, etc., can be added here
    return tokens
```

```
try:
    tokens = parse(source_code)
    print("{:<15} {:<15} {:<15}".format("Lexeme", "Token Type", "Symbol Table"))
    print("="*60)
    for token in tokens:
        lexeme = token["lexeme"]
        token_type = token["type"]
        symbol_table_entry = symbol_table.get(lexeme, '-')
        print("{:<15} {:<15} {:<15}".format(lexeme, token_type, symbol_table_entry))

    print("\nSymbol Table:")
    print("{:<15} {:<15}".format("Identifier", "Type"))
    print("="*30)
    for identifier, var_type in symbol_table.items():
        print("{:<15} {:<15}".format(identifier, var_type))
except Exception as e:
    print(f'Error: {str(e)}')
```

OUTPUT:

```
Lexeme          Token Type      Symbol Table
============================================================
int             KEYWORD         -
(               OPERATOR        -
)               SEPARATOR       -
{               SEPARATOR       -
float           KEYWORD         -
=               OPERATOR        -
3.14            FLOAT           -
;               SEPARATOR       -
int             KEYWORD         -
=               OPERATOR        -
42              FLOAT           -
;               SEPARATOR       -
if              KEYWORD         -
(               OPERATOR        -
x               IDENTIFIER      float
>               OPERATOR        -
2.0             FLOAT           -
)               SEPARATOR       -
{               SEPARATOR       -
y               IDENTIFIER      int
=               OPERATOR        -
y               IDENTIFIER      int
*               OPERATOR        -
2               FLOAT           -
;               SEPARATOR       -
}               SEPARATOR       -
else            KEYWORD         -
{               SEPARATOR       -
y               IDENTIFIER      int
=               OPERATOR        -
y               IDENTIFIER      int
+               OPERATOR        -
1               FLOAT           -
;               SEPARATOR       -
}               SEPARATOR       -
return          KEYWORD         -
;               SEPARATOR       -
}               SEPARATOR       -

Symbol Table:
Identifier      Type
```

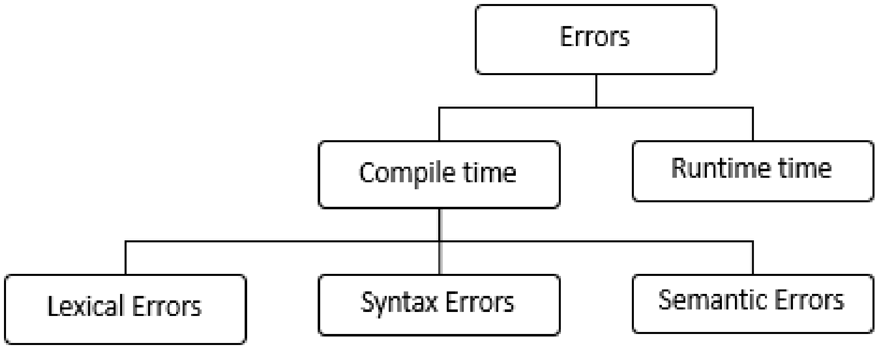
```
=============================
main                int
x                   float
y                   int
```

```
-------------------------------------------------------------------------------------------------------------------------------------------
```

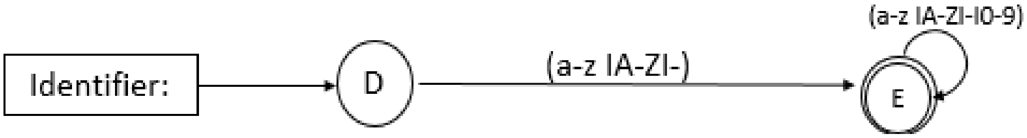
2.5.6 Lexical errors

Lexical errors, also known as lexical or scanning errors, occur during the lexical analysis phase of the compilation process when the compiler is analyzing the source code to identify and tokenize its basic components, such as keywords, identifiers, literals, and symbols. These errors are related to the structure of the code and how it conforms to the language's lexical rules. Here are some common types of lexical errors:

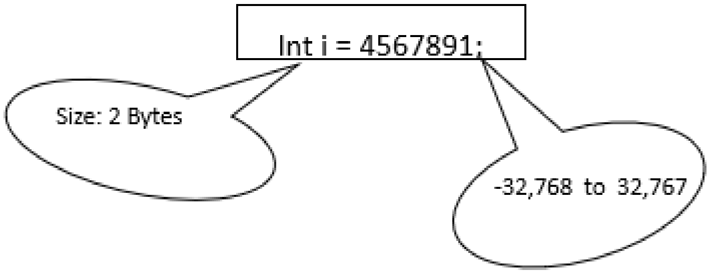
Classification of Errors:



Identifiers that are way too long



- Exceeding length of numeric constants.



- **Numeric constants which are ill-formed.**

```
Int i = 4567$91;
```

2.5.7 *Attributes for token*

In lexical analysis, tokens are the smallest units of meaning in a programming language. Each token represents a specific type of symbol or sequence of characters in the source code. Tokens are identified based on patterns defined by the lexical rules of the language. The attributes of tokens refer to additional information associated with each token type. Here are some common attributes of tokens along with examples:

- Token Type:

Example: int, if, while, +, =

- Lexeme: The actual sequence of characters in the source code that matches a particular token.

Example: For the token type int, the lexeme might be int. For the token type identifier, the lexeme might be variable Name.

- Value: For tokens representing constants (e.g., numeric literals, string literals), the actual value of the constant.

Example: For the token type integer literal, the value might be the actual numeric value like 42.

- Line Number: The line number in the source code where the token was found.

Example: If a token is found on line 5 of the source code, its line number attribute would be 5.

- Position in Line: The position of the token within the line of source code.

Example: If a token is the third element on line 5, its position in line attribute would be 3.

- Token Class: A categorization of tokens into broader classes (e.g., keywords, identifiers, operators).

Example: Token class can be Keyword for tokens like if and while, or Operator for tokens like + and =.

- Scope: The scope in which an identifier is defined (used in languages with scoping rules).

Example: In a programming language with block scope, the scope might be the block in which the identifier is declared.

- Data Type: For languages with static typing, the data type is associated with a variable or constant.

Example: For an identifier representing an integer, the data type attribute might be int.

These attributes help in building the symbol table and capturing additional information needed for subsequent phases of compilation. The specifics of token attributes can vary based on the language and the design choices made by the compiler or interpreter for that language.

2.5.8 Input Buffering

Input buffering is a process used in computer systems and programming to efficiently handle the input data, especially in the context of lexical analysis and parsing in compilers. The primary goal of input buffering is to reduce the number of system calls or I/O operations, improving the overall performance and efficiency of the system.

Here's how input buffering works:

- **Reading Input:** When a program needs to read input, it often uses system calls or I/O functions provided by the operating system. These calls are relatively expensive in terms of processing time.
- **Buffering:** Input buffering involves reading a larger chunk of data from the input source (e.g., a file or keyboard) than is immediately needed by the program. This larger chunk is stored in a buffer—a temporary storage area in memory.
- **Processing from Buffer:** The program then processes the input data from the buffer rather than making frequent I/O operations for small amounts of data. This reduces the number of system calls, which can be a significant performance improvement, especially when dealing with large volumes of input data.
- **Efficiency:** Input buffering is particularly beneficial in situations where reading data in small chunks would result in a high overhead due to frequent system calls. By reading and processing data in larger blocks, the program can make more efficient use of resources.

Example in the Context of Compilers: In the context of compilers, especially during the lexical analysis phase, input buffering is commonly employed. Here's how it works in the context of reading characters from the source code:

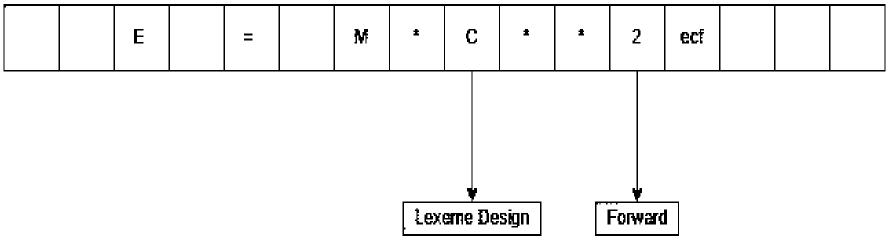
- Instead of reading one character at a time, which could lead to a large number of system calls and slow down the compilation process, a block or buffer of characters is read from the source file into memory.

- The lexical analyzer then reads characters from this buffer, and as it processes them, it advances a pointer within the buffer. When the buffer is exhausted, another block of characters is read into the buffer.
- This minimizes the overhead of reading individual characters and helps in the efficient processing of the source code.

  Specialized buffering techniques decrease the overhead required to process an input character in moving characters.

Buffer Pairs:

The input buffering mechanism consists of two buffers, each with a size of N characters, and they are reloaded alternately. Within this system, two pointers are employed: **lexemeBegin** and **forward**.



- LexemeBegin indicates the commencement of the current lexeme that is yet to be identified.
- Forward moves through the input buffer until it identifies a match for a specific pattern.
- Upon discovering a lexeme, lexemeBegin is adjusted to the character immediately following the newly identified lexeme, while forward is positioned at the rightmost character of that lexeme.
- The range between these two pointers encompasses the characters of the current lexeme. This approach enables efficient scanning and identification of tokens within the source code.

Sentinels

Sentinels In input buffering, sentinels are special markers or signals used to indicate the boundaries or endpoints of the data being processed. They help the system understand where one piece of information ends and the next one begins.

Let's imagine you're reading a book, and at the end of each chapter, there's a special symbol like "***" or "Chapter End." This symbol serves as a sentinel, telling you where one chapter ends and the next one starts. In a similar way, sentinels in input buffering help a computer system understand where one set of data or information in the input stream ends, and the next one begins.

In more technical terms, consider a scenario where you're processing a sequence of characters in a computer program. A special character, often called a sentinel, might be used to mark the end of a piece of data. For example, in strings, a null character ('\0') is commonly used as a sentinel to signify the end of the string. When the program encounters this character, it knows that it has reached the end of the current piece of data.

So, in input buffering, sentinels act like signposts, helping the system keep track of where one set of data ends and the next one starts, making it easier to read and process information.

2.5.9 Specification of Tokens

In the context of compilers and lexical analysis, a token is a sequence of characters in a source code file that represents a fundamental unit of meaning. Tokens are the building blocks of a programming language and are classified into different types, each serving a specific purpose in the syntax of the language. The specification of tokens involves defining the various types of tokens that can appear in the source code and the rules for recognizing them. Here's how the specification of tokens is typically done:

1 **Token Types:** Define the different types of tokens that can appear in the source code. Common token types include keywords, identifiers, literals, operators, punctuation symbols, and special symbols.

2 **Regular Expressions:** Use regular expressions to describe the patterns associated with each token type. Regular expressions are powerful tools for pattern matching and are used to define the syntactic structure of tokens. Each token type has a corresponding regular expression that specifies the allowed character patterns.

3. **Lexical Rules:** Specify lexical rules that define how tokens are formed from the input characters. Lexical rules describe the conditions under which a sequence of characters in the source code is recognized as a particular token type. These rules often include patterns defined by regular expressions.

4. **Token Attributes:** Define attributes associated with each token type. Token attributes provide additional information about the token, such as the value of a literal, the name of an identifier, or the specific operator represented by a token.

5. **Reserved Words:** Identify and list reserved words in the language. Reserved words are keywords that have special meaning in the programming language and cannot be used as identifiers.

6. **Error Handling:** Specify how lexical errors are handled. Lexical errors occur when the input does not match any of the defined token patterns. The specification should include rules for detecting and reporting lexical errors

Let's consider a simplified language with integer literals, identifiers, and basic arithmetic operators:

- **Token Types:**
  - INTEGER_LITERAL
  - IDENTIFIER
  - PLUS (+)

    - MINUS (-)
    - MULTIPLY (*)
    - DIVIDE (/)

- **Regular Expressions:**
    - INTEGER_LITERAL: \d+
    - IDENTIFIER: [a-zA-Z][a-zA-Z0-9]*
    - PLUS: \+
    - MINUS: \-
    - MULTIPLY: *
    - DIVIDE: \/
- **Lexical Rules:**
    - INTEGER_LITERAL: Matches one or more digits.
    - IDENTIFIER: Starts with a letter, followed by letters or digits.
    - PLUS: Matches the plus symbol.
    - MINUS: Matches the minus symbol.
    - MULTIPLY: Matches the asterisk symbol.
    - DIVIDE: Matches the forward slash symbol.
- **Token Attributes:**
    - INTEGER_LITERAL: Value of the integer.
    - IDENTIFIER: Name of the identifier.
    - PLUS, MINUS, MULTIPLY, DIVIDE: No additional attributes.
- **Reserved Words:**
    - None in this simple example.

This is a basic illustration, and in real-world scenarios, the token specification would be more comprehensive, covering a broader range of language constructs and incorporating more advanced lexical analysis techniques.

2.5.10 Recognition of Token

The recognition of tokens is a crucial step in the lexical analysis phase of a compiler. During this phase, the source code is scanned, and sequences of characters are identified as tokens based on the specifications provided in the token definition.

Finite Automata (FA) is a simple idealized machine that can be used to recognize patterns within input taken from a character set or alphabet (denoted as C). The primary task of an FA is to accept or reject an input based on whether the defined pattern occurs within the input.

There are two notations for representing Finite Automata. They are:

- Transition Table
- Transition Diagram

a general process for the recognition of tokens:

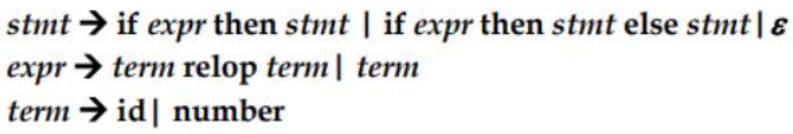
- **Lexical Analysis:** The process begins with the lexical analyzer (also known as the lexer or scanner) scanning the source code character by character.
- **Token Specification:** Referencing the token specification, the lexical analyzer uses regular expressions and rules to define the patterns associated with different token types.
- **Pattern Matching:** As characters are read from the source code, the lexical analyzer attempts to match them against the defined token patterns. Regular expressions play a crucial role in this step, as they describe the valid patterns for each token type.
- **Token Creation:** When a sequence of characters matches a token pattern, a token is created. The token includes the type of the token (e.g., identifier, keyword, operator), and in some cases, additional attributes such as the value of a literal or the name of an identifier.
- **Error Detection:** If a sequence of characters does not match any token pattern, the lexical analyzer may detect a lexical error. Error handling mechanisms are implemented to report and handle such errors, providing feedback to the programmer.
- **Token Stream:** The result of the recognition process is a stream of tokens, representing the fundamental units of meaning in the source code. This token stream is then passed on to the next phase of the compiler for further processing.

1. **Transition Table:**

   It is a tabular representation that lists all possible transitions for each state and input symbol combination.

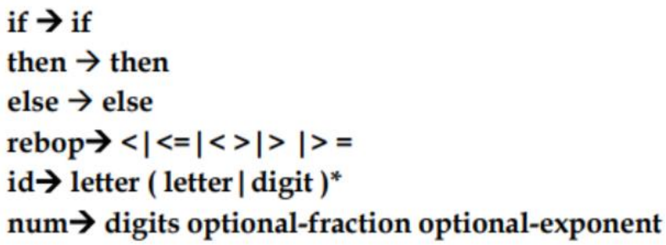
   Example:

   Assume the following grammar fragment to generate a specific language



stmt → **if** *expr* **then** *stmt* | **if** *expr* **then** *stmt* **else** *stmt* | *ε*
expr → *term* **relop** *term* | *term*
term → **id** | **number**

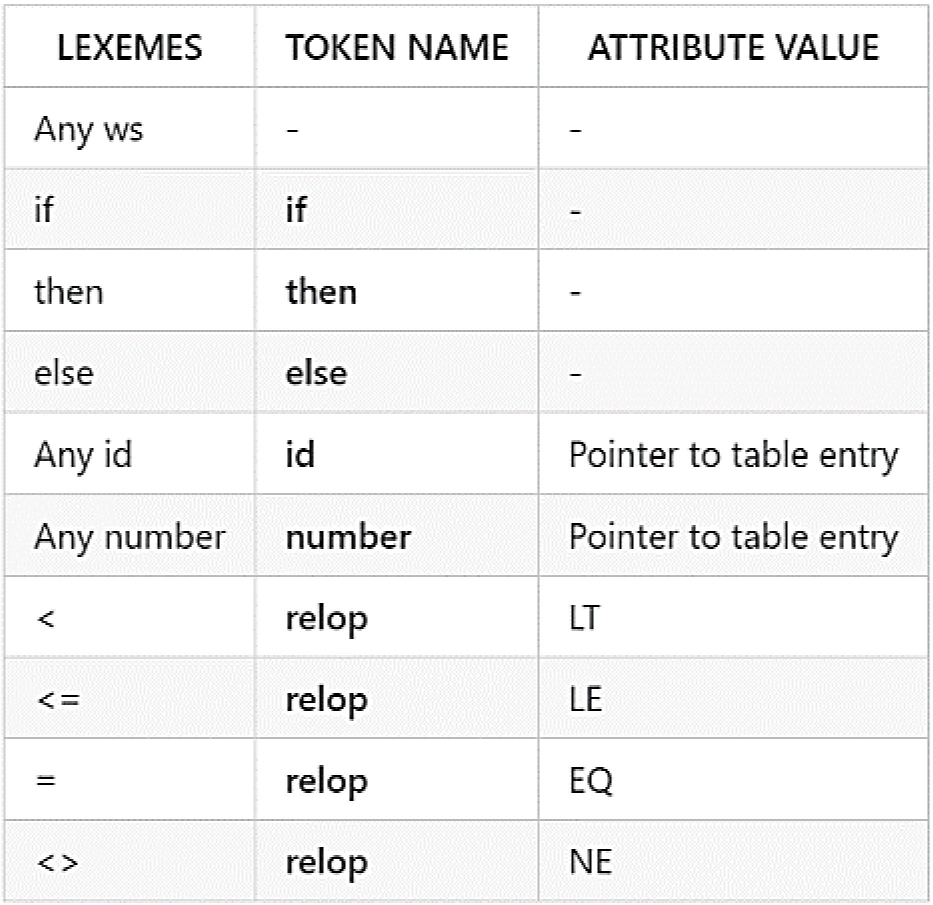
where the terminals if, then, else, relop, id and num generates sets of strings given by following regular definitions.



if → if
then → then
else → else
rebop→ < | <= | < > | > | > =
id→ letter (letter | digit)*
num→ digits optional-fraction optional-exponent

- where letter and digits are defined as - (letter → [A-Z a-z] & digit → [0-9])
- For this language, the lexical analyzer will recognize the keywords if, then, and else, as well as lexemes that match the patterns for relop, id, and number.
- To simplify matters, we make the common assumption that keywords are also reserved words: that is they cannot be used as identifiers.
- The num represents the unsigned integer and real numbers of Pascal.
- In addition, we assume lexemes are separated by white space, consisting of nonnull sequences of blanks, tabs, and newlines.
- Our lexical analyzer will strip out white space. It will do so by comparing a string against the regular definition ws, below.
- If a match for ws is found, the lexical analyzer does not return a token to the parser

It is the following token that gets returned to the parser.

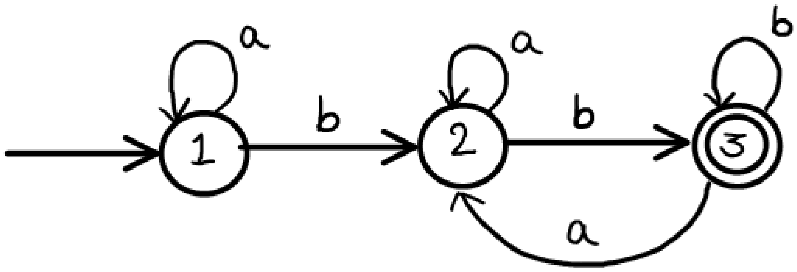


LEXEMES	TOKEN NAME	ATTRIBUTE VALUE
Any ws	-	-
if	**if**	-
then	**then**	-
else	**else**	-
Any id	**id**	Pointer to table entry
Any number	**number**	Pointer to table entry
<	**relop**	LT
<=	**relop**	LE
=	**relop**	EQ
<>	**relop**	NE

2. Transition Diagram

It is a directed labeled graph consisting of nodes and edges. Nodes represent states, while edges represent state transitions.

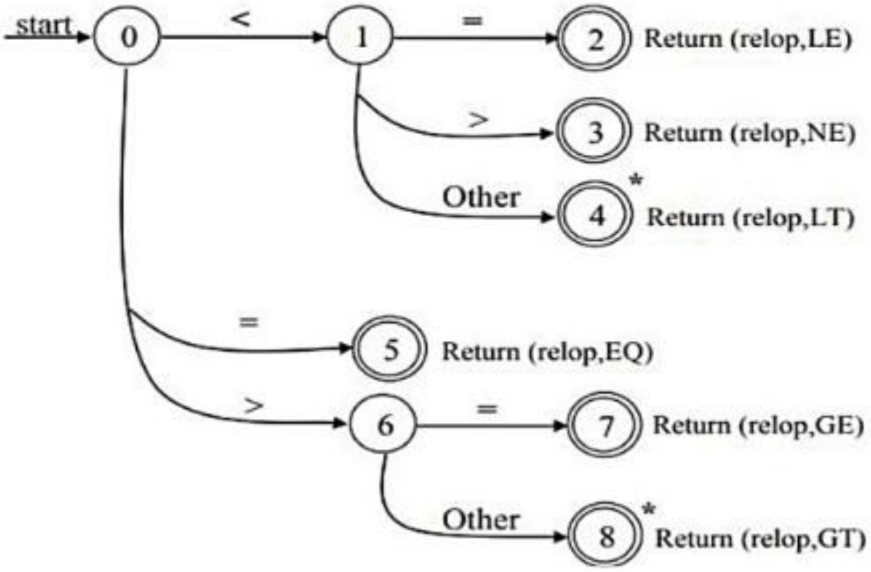
1. One state is labelled the **Start State**. It is the initial state of transition diagram where control resides when we begin to recognize a token.
2. Position is a transition diagram are drawn as circles and are called **states**.
3. The states are connected by Arrows called **edges**. Labels on edges are indicating the input characters
4. Zero or more **final states** or **Accepting states** are represented by double circle in which the tokens has been found.



the transition diagram of Finite Automata that recognizes the lexemes matching the token relop.

<u>Example:</u> A Transition Diagram for the token relation operators

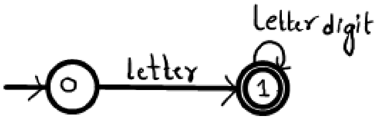
"relop" is shown in Figure below:

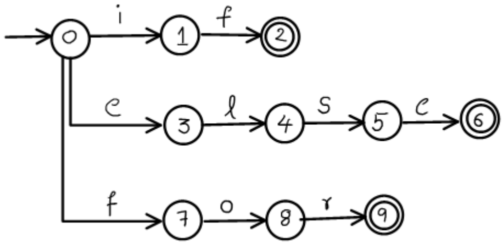


the Finite Automata Transition Diagram for the Identifiers and Keywords.

Letter ⟶ a|b|.......|z| A|B|.......|Z|
Digit ⟶ 0|1|........|9|
Id ⟶ letter(letter | digit)*

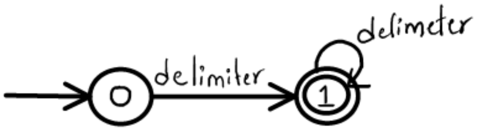
abc12





Here is the Finite Automata Transition Diagram for recognizing white spaces.

WS ⟶ delimiter (delimiter)*



Write regular expression for number

Example:- Reg. Expr for digits (int no, floating point no)

digit ⟶ 0 | 1 | …….. | 9

digits ⟶ digit (digit)*

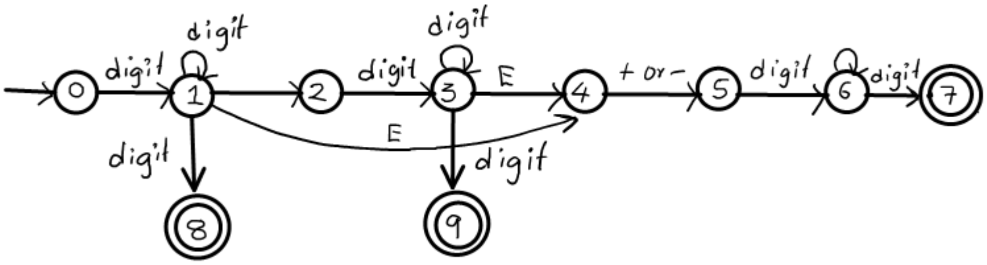
Number ⟶ digits (. digits)? (E [+ -] ? digits)

123 456
123.45
123.45 E.23
123.45 E-23
123 E+23

5. Recognition of numbers (int | flating points)

$$\text{Number} \longrightarrow \text{Digits}^{+} \; \overset{1}{(\text{digit}^{+})?} \; \overset{0}{(E\,[+-]\,?\,\text{digit}^{+})?}$$

123
123.45
123 E 20
123.45 E +2
123.45 E -2



Recognize the given token in python

```
import re

def tokenize(input_string):
    # Define regular expressions for different types of tokens
    token_patterns = [
        (r'\bif\b|\belse\b|\bwhile\b|\bfor\b', 'KEYWORD'),
        (r'\b\d+\b', 'NUMBER'),
        (r'\b[a-zA-Z_]\w*\b', 'IDENTIFIER'),
        (r'[-+*/=<>]', 'OPERATOR'),    # Updated to include comparison operators
        (r'\s', 'WHITESPACE'),
        (r'\(|\)', 'PARENTHESIS'),    # Add a pattern for parentheses
        (r':', 'COLON'),    # Add a pattern for colon
    ]

    tokens = []

    while input_string:
        for pattern, token_type in token_patterns:
            match = re.match(pattern, input_string)
            if match:
                value = match.group(0)
                tokens.append((value, token_type))
                input_string = input_string[len(value):].lstrip()
                break
        else:
            # If no match is found, raise an error or handle accordingly
            raise ValueError(f"Unable to tokenize: {input_string}")

    return tokens

# Example usage
input_string = "if x > 5 for x in range(10): print(x)"
tokens = tokenize(input_string)

for token in tokens:
    print(token)
```

OUTPUT:

```
('if', 'KEYWORD')
('x', 'IDENTIFIER')
('>', 'OPERATOR')
('5', 'NUMBER')
('for', 'KEYWORD')
('x', 'IDENTIFIER')
('in', 'IDENTIFIER')
('range', 'IDENTIFIER')
('(', 'PARENTHESIS')
('10', 'NUMBER')
(')', 'PARENTHESIS')
(':', 'COLON')
('print', 'IDENTIFIER')
('(', 'PARENTHESIS')
('x', 'IDENTIFIER')
(')', 'PARENTHESIS')
```