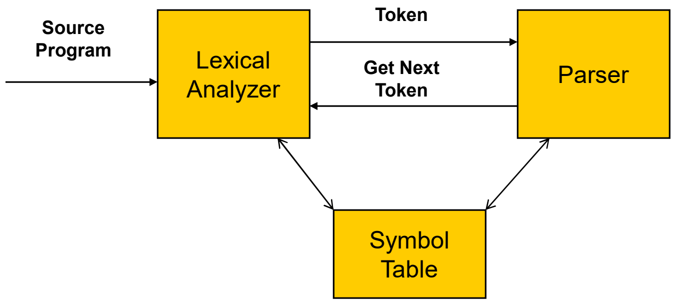
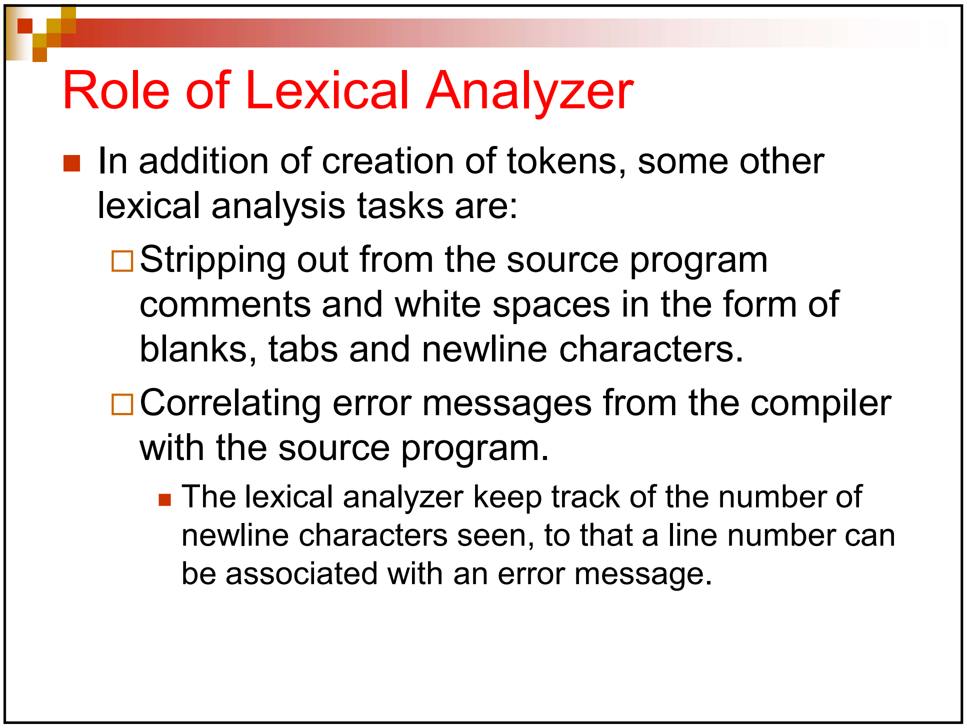
Chapter # 2 Lexical Analysis

Dr. Shaukat Ali
Department of Computer Science
University of Peshawar

Role of Lexical Analyzer

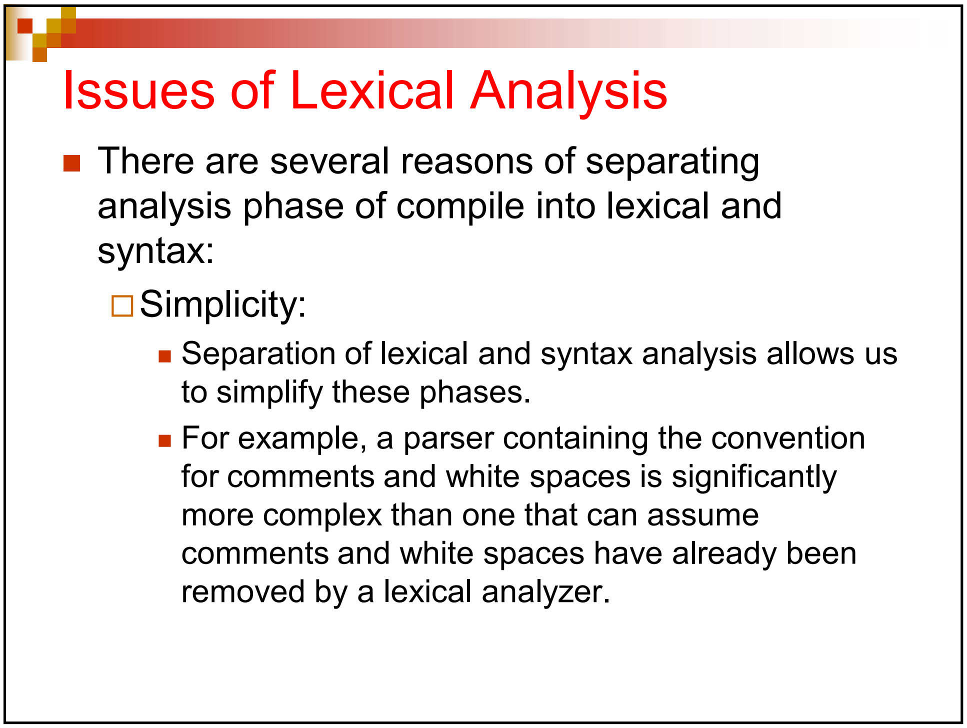
- The main task of lexical analyzer is to read source program as file of characters and produce as output a sequence of tokens that the parser uses for syntax analysis.





Role of Lexical Analyzer

- In addition of creation of tokens, some other lexical analysis tasks are:
  - Stripping out from the source program comments and white spaces in the form of blanks, tabs and newline characters.
  - Correlating error messages from the compiler with the source program.
    - The lexical analyzer keep track of the number of newline characters seen, to that a line number can be associated with an error message.



Issues of Lexical Analysis

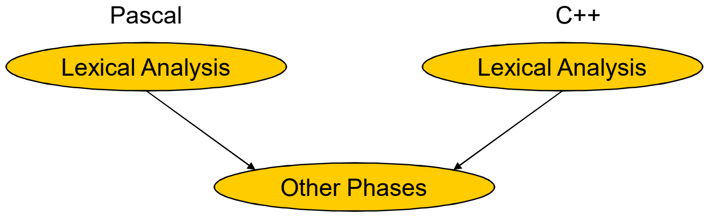
- There are several reasons of separating analysis phase of compile into lexical and syntax:
  - Simplicity:
    - Separation of lexical and syntax analysis allows us to simplify these phases.
    - For example, a parser containing the convention for comments and white spaces is significantly more complex than one that can assume comments and white spaces have already been removed by a lexical analyzer.

Issues of Lexical Analysis

- Efficiency:
  - A large amount of time is spent reading the source program and partitioning into tokens.
  - Separate lexical and syntax analyzer working in parallel (at the same time, lexical analyzer creating tokens and syntax analyzer organizes into parse trees) can significantly improve the performance.
- Portability:
  - Mostly the lexical analysis phase is different for different languages and other phases are almost the same.
    - Because different characters have different meaning in different languages.

Issues of Lexical Analysis

- Therefore, for a new language we may only require to construct the lexical analyzer and the rest of the phases of the other languages can be attached to it.



- To construct a compiler for C++ language we only have to develop the lexical analysis phase and use the other phases of pascal language.

Token, Pattern and Lexeme

- ***Token:***
  - The sequence of characters having a collective meaning is called token.
  - A token represents a class in the vocabulary of a language and is an indivisible lexical unit.
- ***Pattern:***
  - A token represents a set of strings in the input. This set of strings is describe by a rule called pattern associated with token.
  - The pattern is said to match each string in the set.

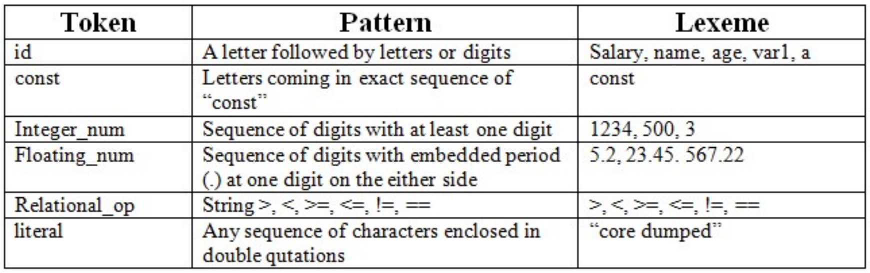
Token, Pattern and Lexeme

- Lexeme:
  - The sequence of characters in the soruce program that is matched by the pattern for a token.
  - Consider the Pascal statement:

    const pi = 3.1416;

    - Here pi is the lexeme for the token “identifier”.

Token, Pattern and Lexeme



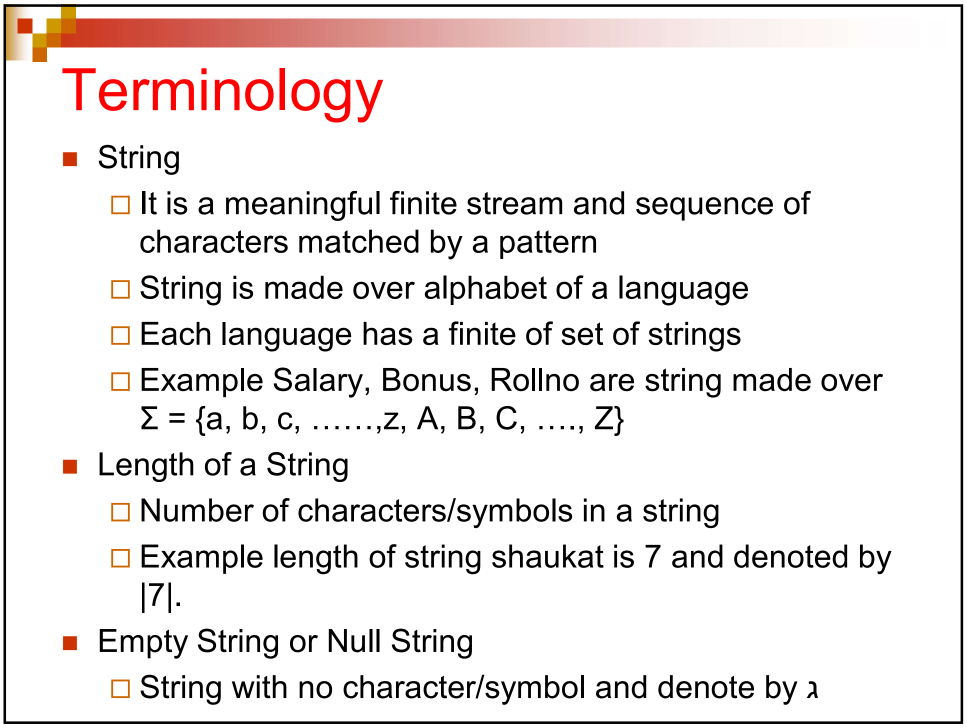
Token	Pattern	Lexeme
id	A letter followed by letters or digits	Salary, name, age, var1, a
const	Letters coming in exact sequence of "const"	const
Integer_num	Sequence of digits with at least one digit	1234, 500, 3
Floating_num	Sequence of digits with embedded period (.) at one digit on the either side	5.2, 23.45. 567.22
Relational_op	String >, <, >=, <=, !=, ==	>, <, >=, <=, !=, ==
literal	Any sequence of characters enclosed in double qutations	"core dumped"

Terminology

- Symbol
  - □ It is an abstract entity having some meaning
    - Example are letter digits etc
- Alphabets
  - □ A finite non-empty set of symbols from which string are constructed. It is denote by Greek letter Σ
  - □ Example

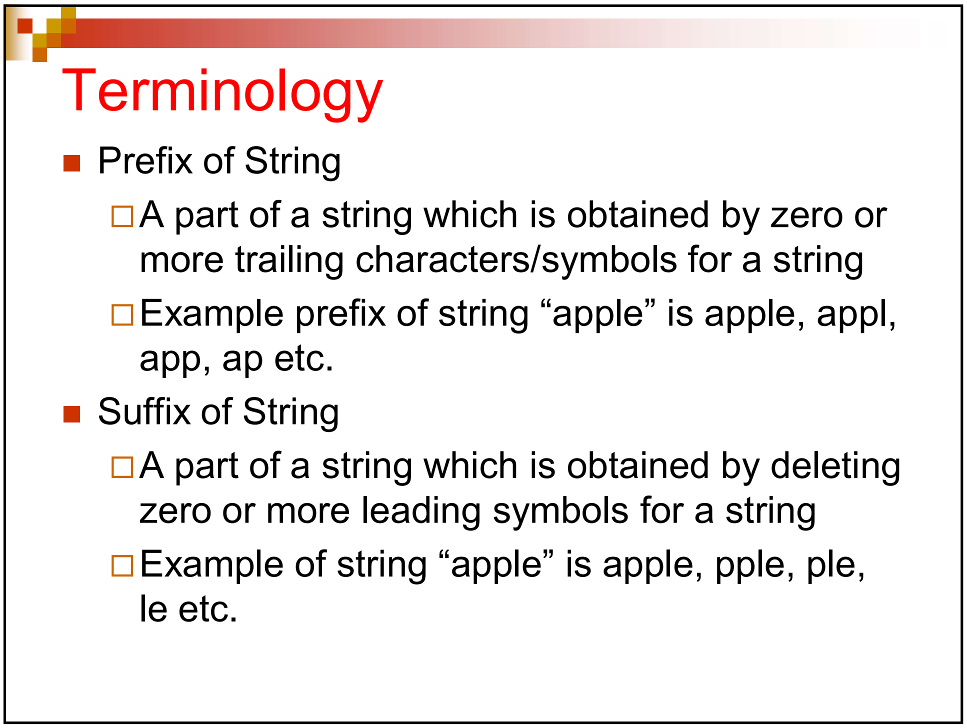
    Σ = {a, b, c, ……,z, A, B, C, …., Z}

    Σ = {0, 1, 2, ….., 9}



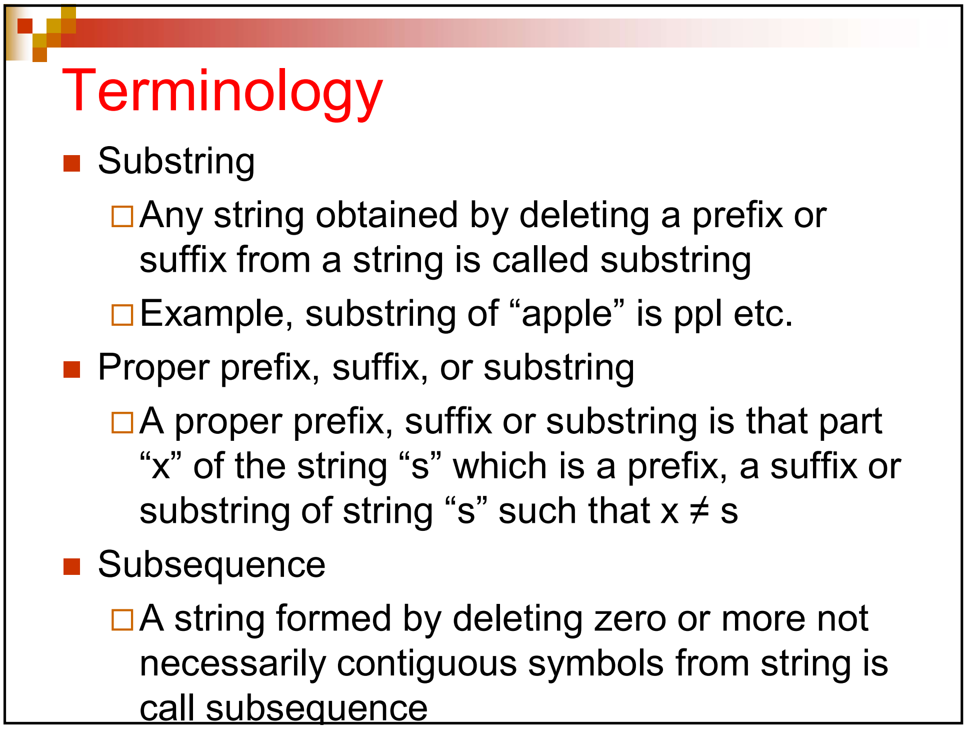
Terminology

- String
  - □ It is a meaningful finite stream and sequence of characters matched by a pattern
  - □ String is made over alphabet of a language
  - □ Each language has a finite of set of strings
  - □ Example Salary, Bonus, Rollno are string made over Σ = {a, b, c, ……,z, A, B, C, …., Z}
- Length of a String
  - □ Number of characters/symbols in a string
  - □ Example length of string shaukat is 7 and denoted by |7|.
- Empty String or Null String
  - □ String with no character/symbol and denote by ג



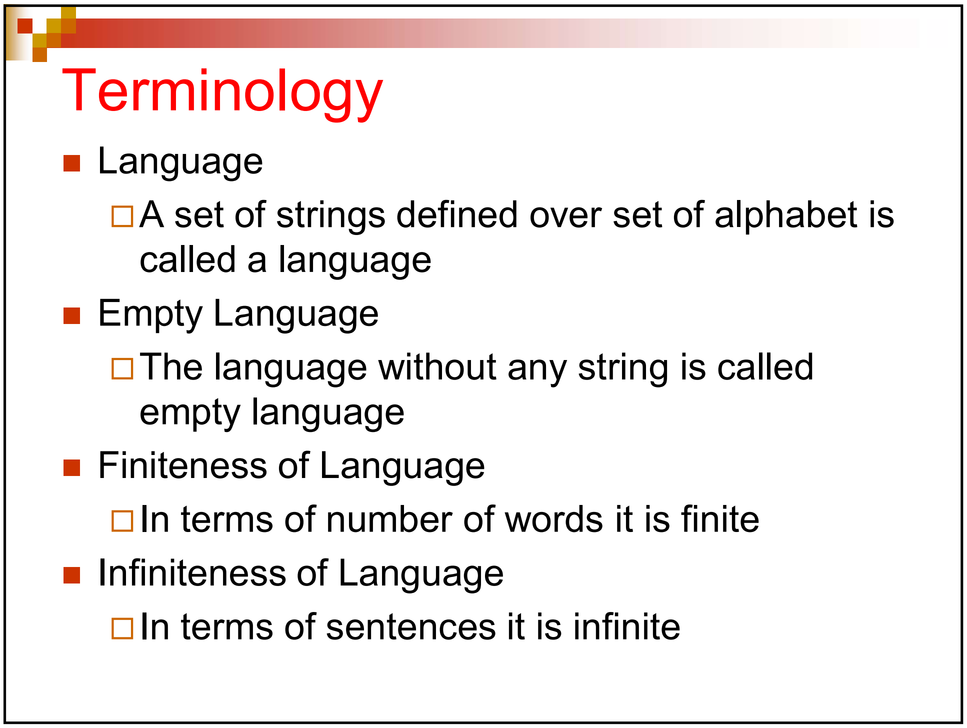
Terminology

- Prefix of String
  - □ A part of a string which is obtained by zero or more trailing characters/symbols for a string
  - □ Example prefix of string “apple” is apple, appl, app, ap etc.
- Suffix of String
  - □ A part of a string which is obtained by deleting zero or more leading symbols for a string
  - □ Example of string “apple” is apple, pple, ple, le etc.



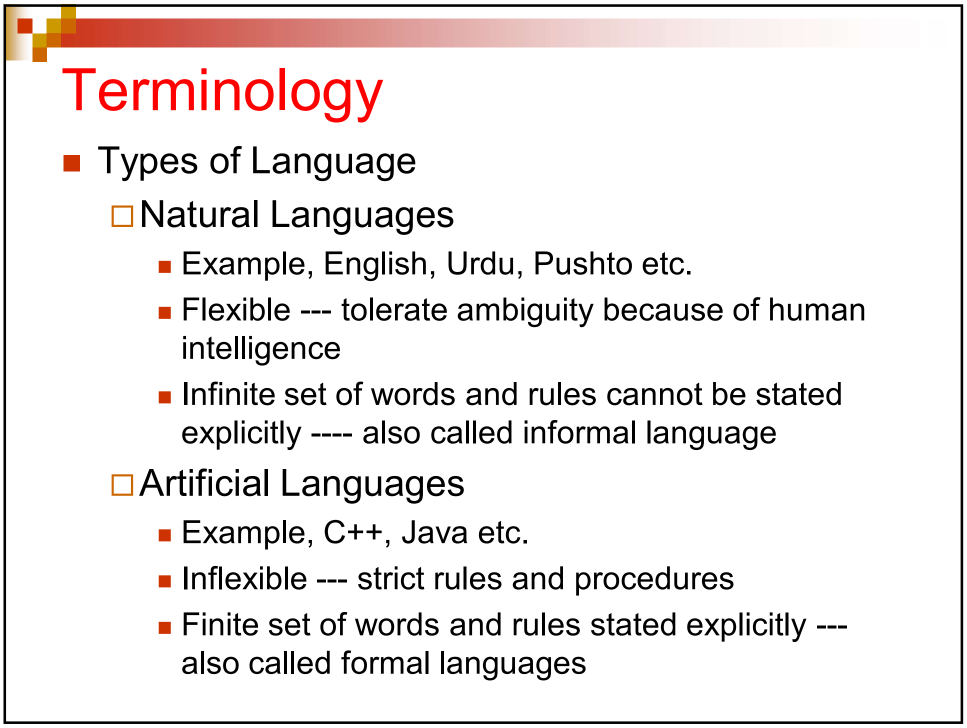
Terminology

- Substring
  - □ Any string obtained by deleting a prefix or suffix from a string is called substring
  - □ Example, substring of “apple” is ppl etc.
- Proper prefix, suffix, or substring
  - □ A proper prefix, suffix or substring is that part “x” of the string “s” which is a prefix, a suffix or substring of string “s” such that x ≠ s
- Subsequence
  - □ A string formed by deleting zero or more not necessarily contiguous symbols from string is call subsequence



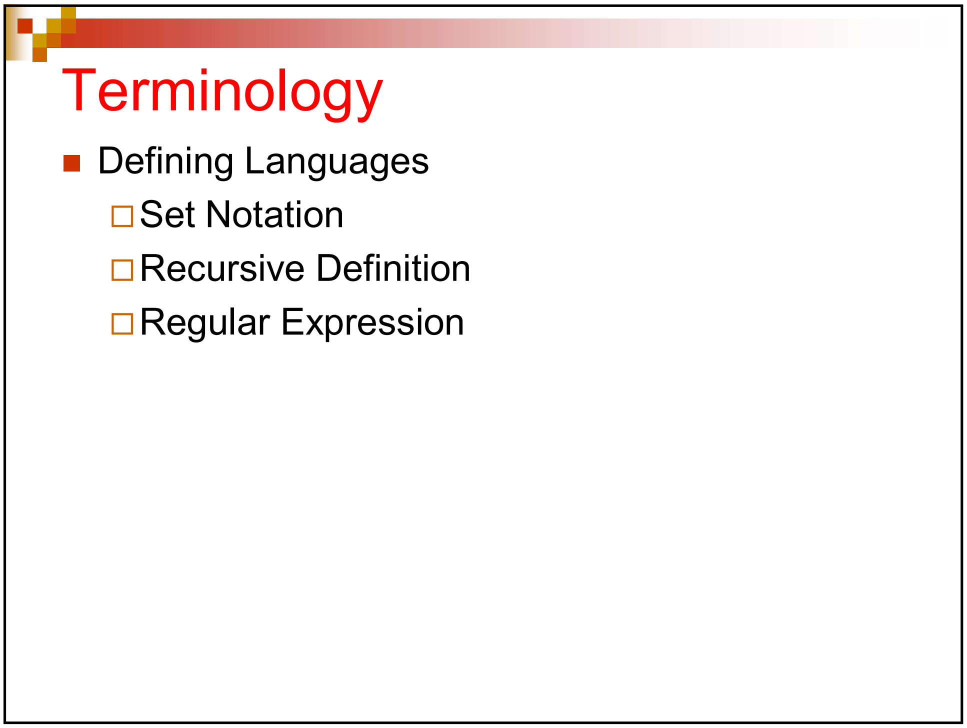
Terminology

- Language
  - □ A set of strings defined over set of alphabet is called a language
- Empty Language
  - □ The language without any string is called empty language
- Finiteness of Language
  - □ In terms of number of words it is finite
- Infiniteness of Language
  - □ In terms of sentences it is infinite



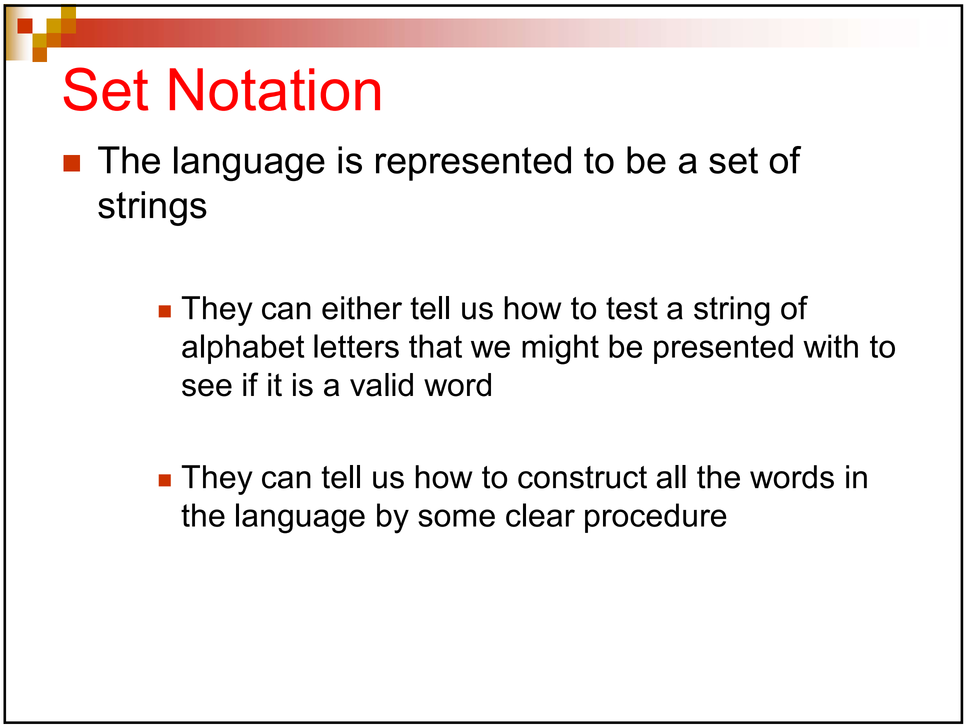
Terminology

- Types of Language
  - □ Natural Languages
    - Example, English, Urdu, Pushto etc.
    - Flexible --- tolerate ambiguity because of human intelligence
    - Infinite set of words and rules cannot be stated explicitly ---- also called informal language
  - □ Artificial Languages
    - Example, C++, Java etc.
    - Inflexible --- strict rules and procedures
    - Finite set of words and rules stated explicitly --- also called formal languages



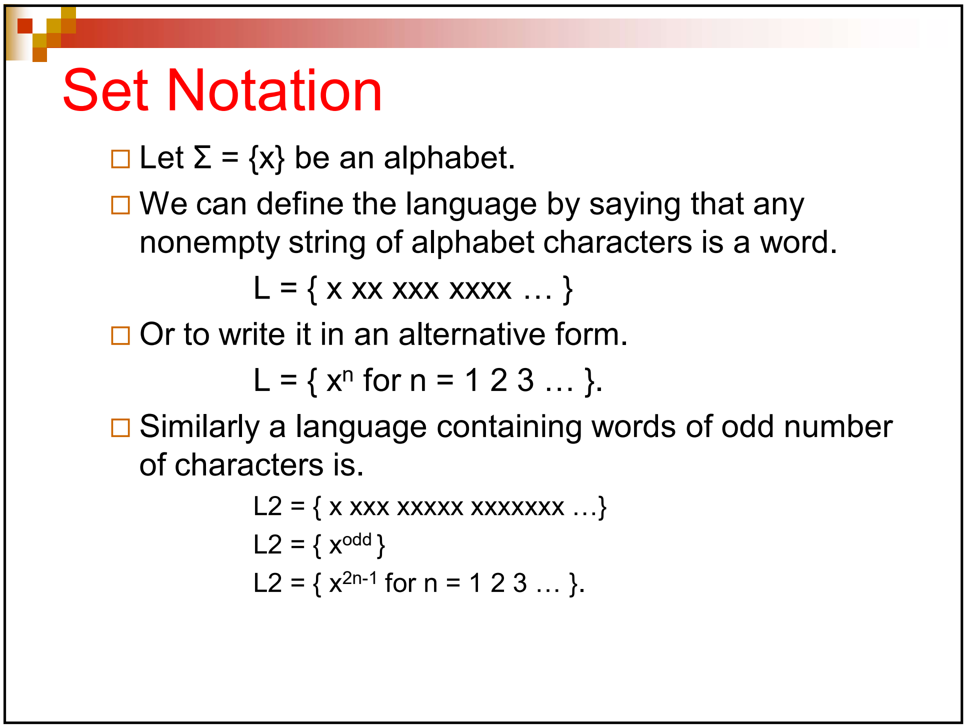
Terminology

- Defining Languages
  - □ Set Notation
  - □ Recursive Definition
  - □ Regular Expression



Set Notation

- The language is represented to be a set of strings
  - They can either tell us how to test a string of alphabet letters that we might be presented with to see if it is a valid word
  - They can tell us how to construct all the words in the language by some clear procedure



Set Notation

- □ Let Σ = {x} be an alphabet.
- □ We can define the language by saying that any nonempty string of alphabet characters is a word.

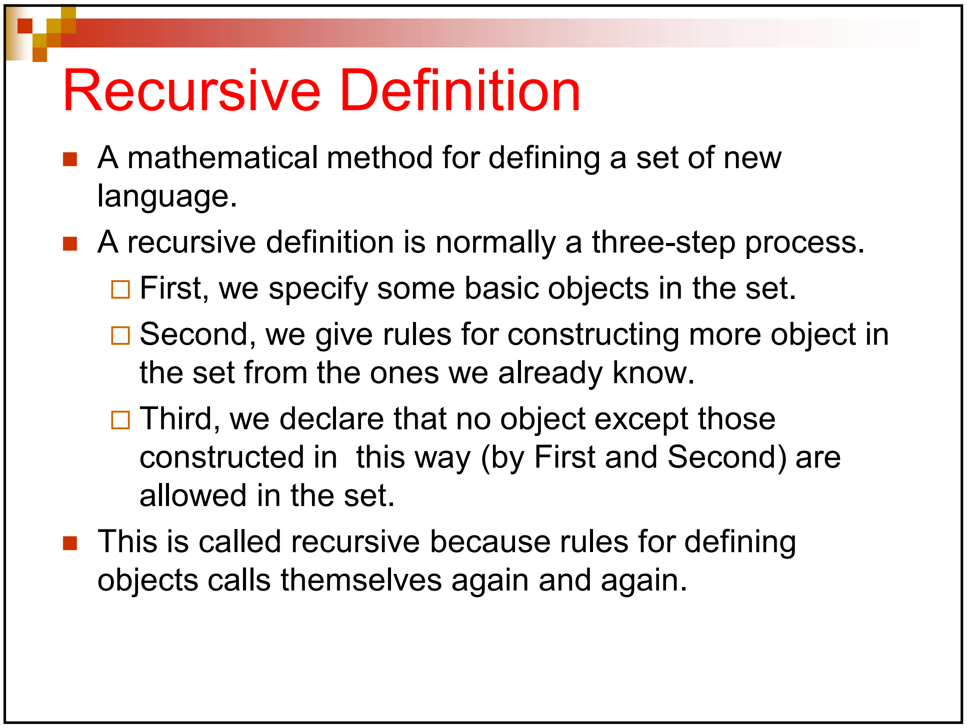
  L = { x xx xxx xxxx … }
- □ Or to write it in an alternative form.

  L = { x^n for n = 1 2 3 … }.
- □ Similarly a language containing words of odd number of characters is.

  L2 = { x xxx xxxxx xxxxxxx …}

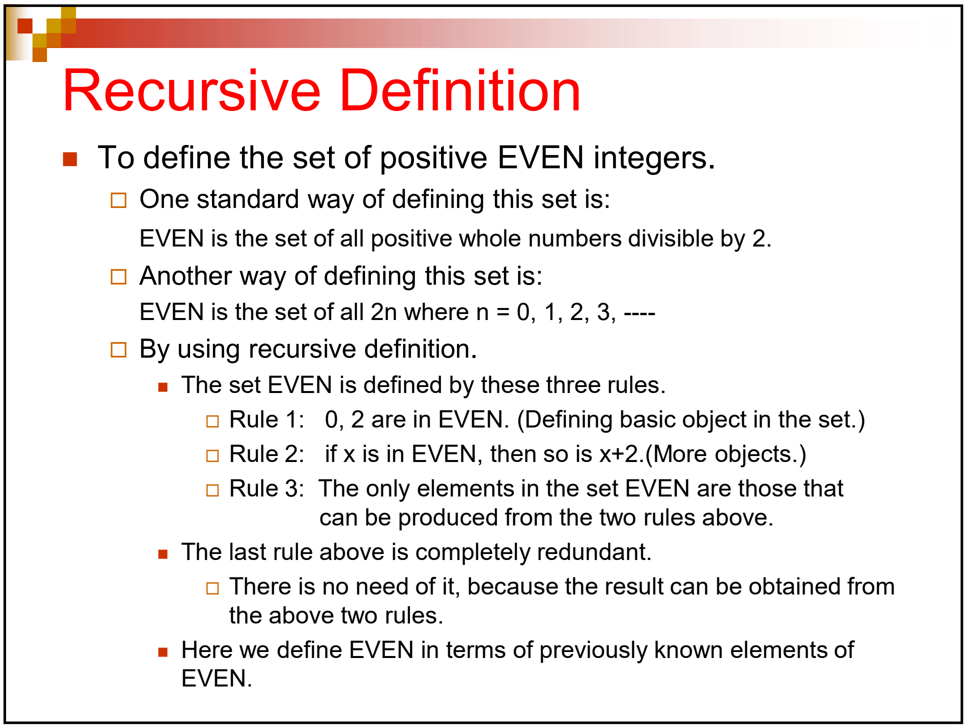
  L2 = { x^{odd} }

  L2 = { x^{2n-1} for n = 1 2 3 … }.



Recursive Definition

- A mathematical method for defining a set of new language.
- A recursive definition is normally a three-step process.
  - □ First, we specify some basic objects in the set.
  - □ Second, we give rules for constructing more object in the set from the ones we already know.
  - □ Third, we declare that no object except those constructed in this way (by First and Second) are allowed in the set.
- This is called recursive because rules for defining objects calls themselves again and again.

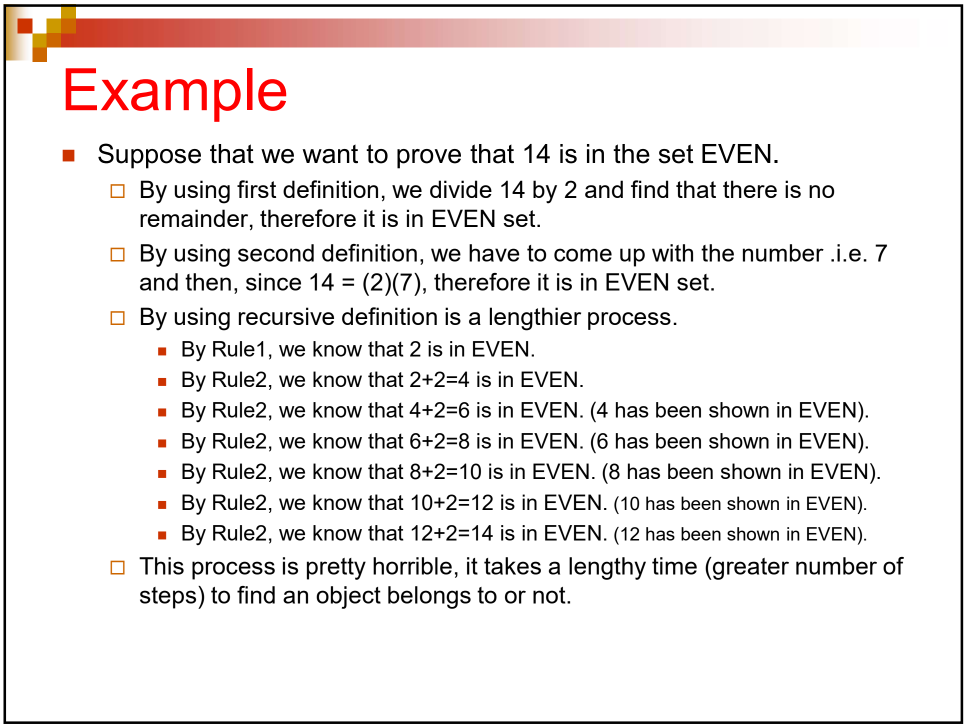


Recursive Definition

- To define the set of positive EVEN integers.
  - □ One standard way of defining this set is:

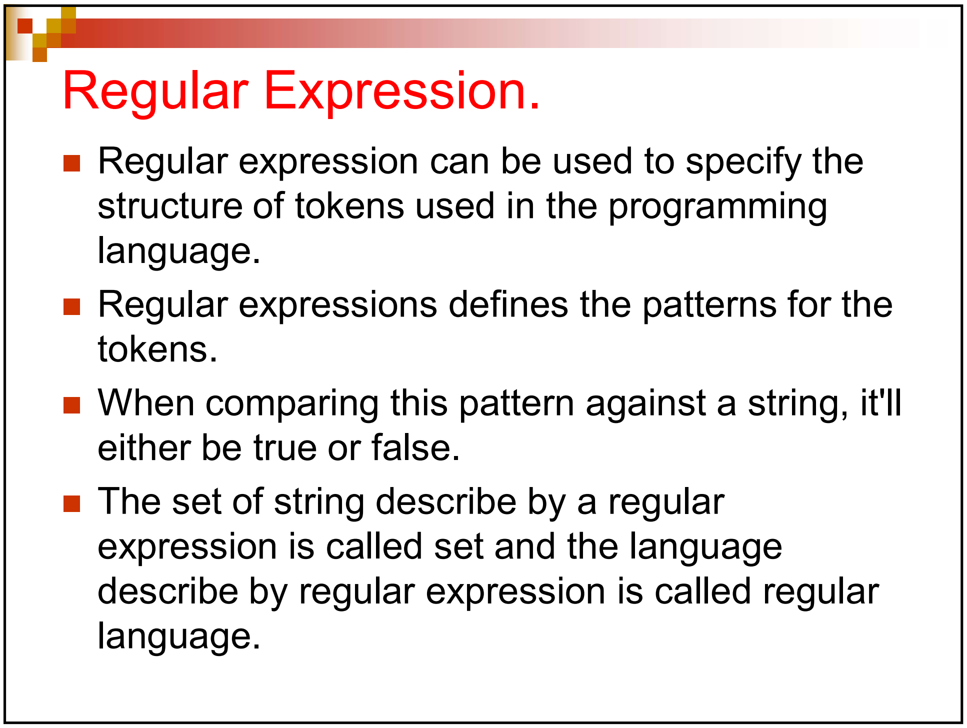
    EVEN is the set of all positive whole numbers divisible by 2.
  - □ Another way of defining this set is:

    EVEN is the set of all 2n where n = 0, 1, 2, 3, ----
  - □ By using recursive definition.
    - The set EVEN is defined by these three rules.
      - □ Rule 1: 0, 2 are in EVEN. (Defining basic object in the set.)
      - □ Rule 2: if x is in EVEN, then so is x+2.(More objects.)
      - □ Rule 3: The only elements in the set EVEN are those that can be produced from the two rules above.
    - The last rule above is completely redundant.
      - □ There is no need of it, because the result can be obtained from the above two rules.
    - Here we define EVEN in terms of previously known elements of EVEN.



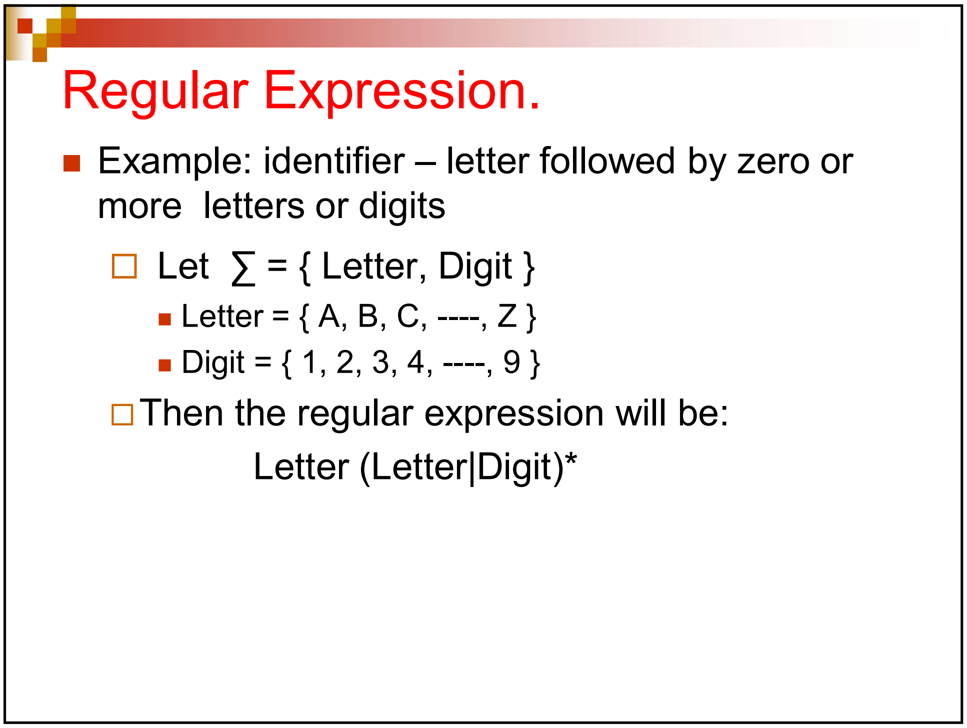
Example

- Suppose that we want to prove that 14 is in the set EVEN.
  - By using first definition, we divide 14 by 2 and find that there is no remainder, therefore it is in EVEN set.
  - By using second definition, we have to come up with the number .i.e. 7 and then, since 14 = (2)(7), therefore it is in EVEN set.
  - By using recursive definition is a lengthier process.
    - By Rule1, we know that 2 is in EVEN.
    - By Rule2, we know that 2+2=4 is in EVEN.
    - By Rule2, we know that 4+2=6 is in EVEN. (4 has been shown in EVEN).
    - By Rule2, we know that 6+2=8 is in EVEN. (6 has been shown in EVEN).
    - By Rule2, we know that 8+2=10 is in EVEN. (8 has been shown in EVEN).
    - By Rule2, we know that 10+2=12 is in EVEN. (10 has been shown in EVEN).
    - By Rule2, we know that 12+2=14 is in EVEN. (12 has been shown in EVEN).
  - This process is pretty horrible, it takes a lengthy time (greater number of steps) to find an object belongs to or not.



Regular Expression.

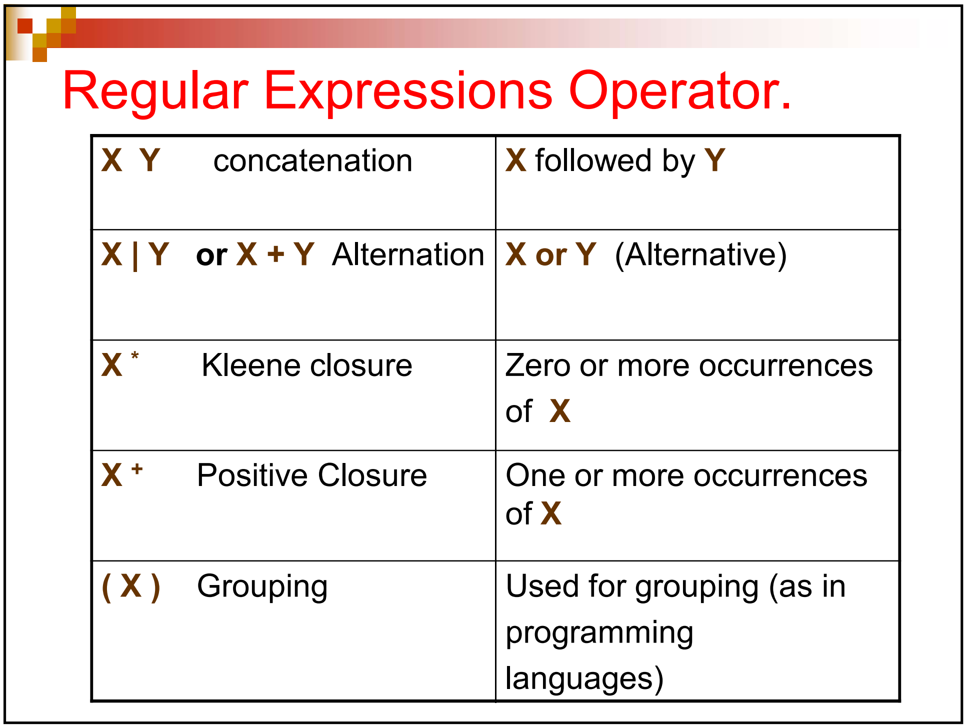
- Regular expression can be used to specify the structure of tokens used in the programming language.
- Regular expressions defines the patterns for the tokens.
- When comparing this pattern against a string, it'll either be true or false.
- The set of string describe by a regular expression is called set and the language describe by regular expression is called regular language.



Regular Expression.

- Example: identifier – letter followed by zero or more letters or digits
  - □ Let ∑ = { Letter, Digit }
    - Letter = { A, B, C, ----, Z }
    - Digit = { 1, 2, 3, 4, ----, 9 }
  - □ Then the regular expression will be:

    Letter (Letter|Digit)*



Regular Expressions Operator.

X Y concatenation	**X** followed by **Y**
X | Y **or X + Y** Alternation	**X or Y** (Alternative)
X^{*} Kleene closure	Zero or more occurrences of **X**
X^{+} Positive Closure	One or more occurrences of **X**
(X) Grouping	Used for grouping (as in programming languages)

Precedence for RE Operators

Regular Expression Operator	Analagous Arithmetic Operator	Precedence
X | Y	X + Y	lowest
X Y	X * Y	middle
X *, X +	X ^ Y	highest

- For example:

  letter letter | digit *

  letter (letter | digit) *

- Closure have higher precedence over concatenation and alternation.
- Concatenation have higher precedence over alternation.
- Alternation have the lowest precedence.

Example

- Suppose that we have set of alphabets.

  ∑ = { a, b }

- Language that could be describe by using the alphabets is as:

  L = { All words of the form one a *followed by any number of* b*'s}*

- Therefore the language can be written as:

  L = { a, ab, abb, abbb, abbbb, ---- }

- Regular Exression for this language would be:

  L = language (a b)*

  □ Or simply.

  *a b**

Example

- Describe the language defined by the regular expression.

  ab*a

- The languge will be:

  L = { all words of a's and b's that have at least two letters, that begin and end with a's }

- Similarly words in the language will be:

  L = { aa, aba, abba, abbba, abbbba, ---- }

Example

- Consider the alphabet ∑ = { a, b, c } and the language is :

  L = { a, c, ab, cb, abb, cbb, abbb, cbbb, abbbb, cbbbb }

  L = { All words of a's, b's and c's that either starts with a or c followed by any number of b's}

  □RE will be:

  (a | c) b* or (a + c) b*

Example

- Write RE that describe the language of all words that have exactly two a's.
  - b* a b* a b*
- Write RE that describe the language of all words that have at least one a and at least one b.
  - (a | b)* a (a | b)* b (a | b)*
  - ((a | b)* a (a | b)* b (a | b)*) |
    ((a | b)* b (a | b)* a (a | b)*)

RE examples

- Write RE that describe the language of all words that have odd number of a's.

  b*ab*(ab*ab*)*

- Write RE that describe the language of all words that have substring with ab.

  (a|b)*ab(a|b)*

RE Exampels

- Write RE that describe the language of all words that have even number of a’s.

$$(b|ab^*ab^*)^*$$

- Write RE that describe the language of all words that have either the second or third position form the end is a.

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

Regular Expressions Equivalences

- Two regular expressions R1 and R2 are equivalent if the language defined by R1 (i.e., the set of strings generated by regular expression R1) is equal to the language defined by R2.
- To prove equivalences for regular expressions, we use containment proofs from set theory.
  - □ That is, if S1 is the set of strings generated by regular expression R1, and S2 is the set of strings generated by regular expression R2.
  - □ We must prove that S1€ S2 and S2 € S1.
  - □ Both directions are necessary to prove equality of sets.

Example

- Let ∑ = { a, b }
  - □ S1 = { a, b, aa, bb, ab, ba }
    - Its RE will be:

      R1 = (a | b)+
  - □ S2 = { a, b, aa, bb, ab, ba }
    - Its RE will be:

      R2 = (a | b) (a | b)*
  - □ As it is clear now that:

    S1 € S2 and S2 € S1
  - □ Therefore:

    R1 = R2

Regular Definitions

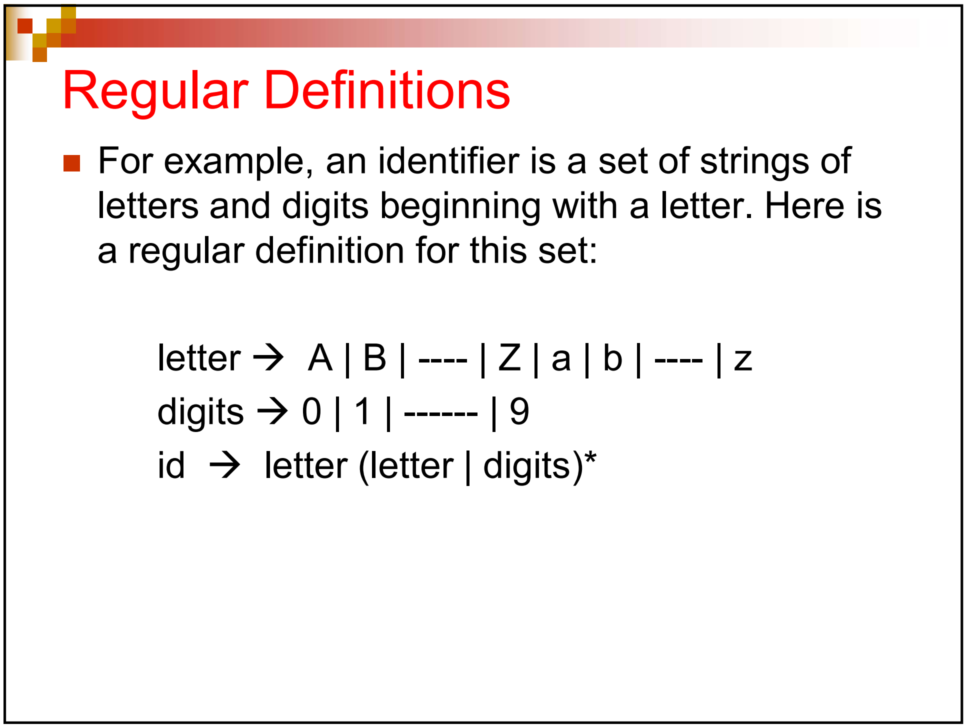
- For notational convenience, we may wish to give names to regular expressions and use these names in other regular expression as if they were the symbols.
- If ∑ is an alphabet of basic symbols, then a regular definition is a sequence of definitions of the form:

  d_1 → r_1

  d_2 → r_2

  ----

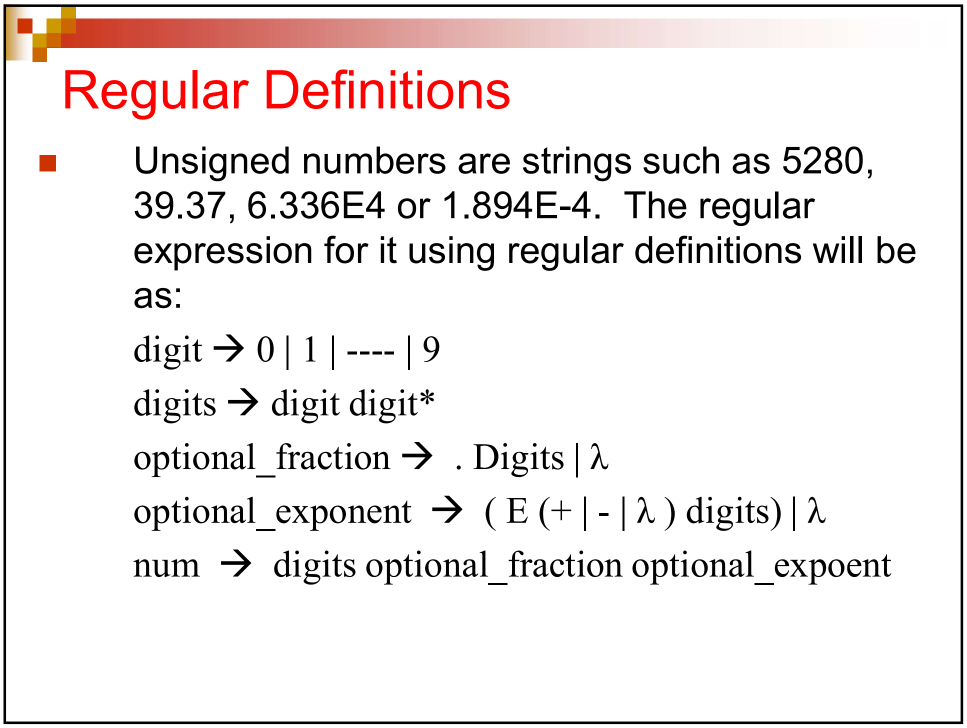
  d_n → r_n
  - □ Where each d_i is a distinct name and each r_i is a regular expression over the symbol in ∑.



Regular Definitions

- For example, an identifier is a set of strings of letters and digits beginning with a letter. Here is a regular definition for this set:

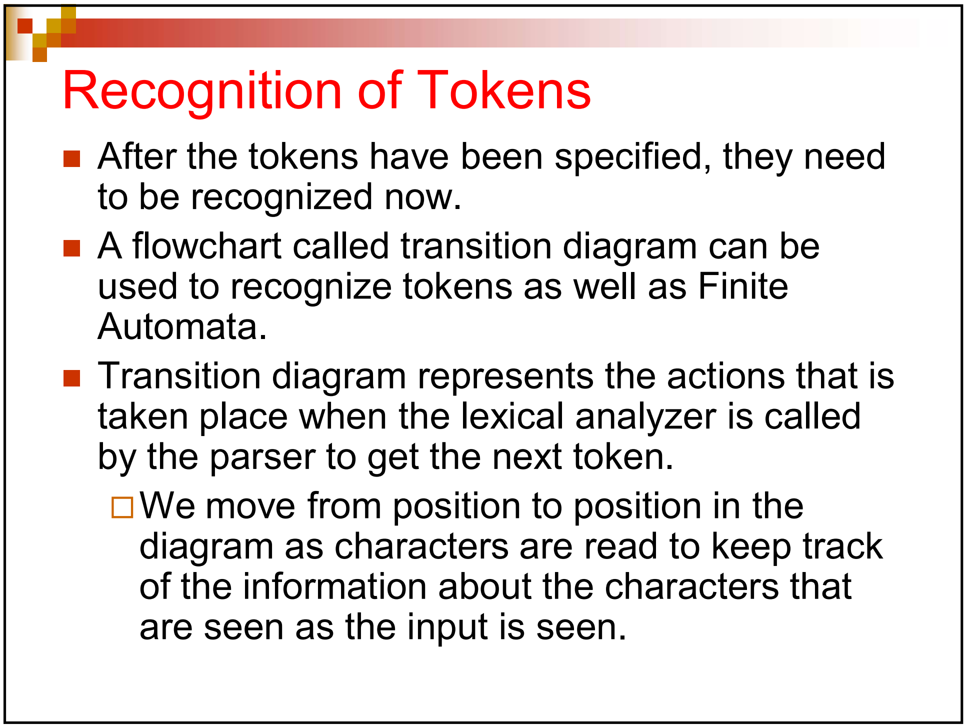
letter → A | B | ---- | Z | a | b | ---- | z
digits → 0 | 1 | ------ | 9
id → letter (letter | digits)*



Regular Definitions

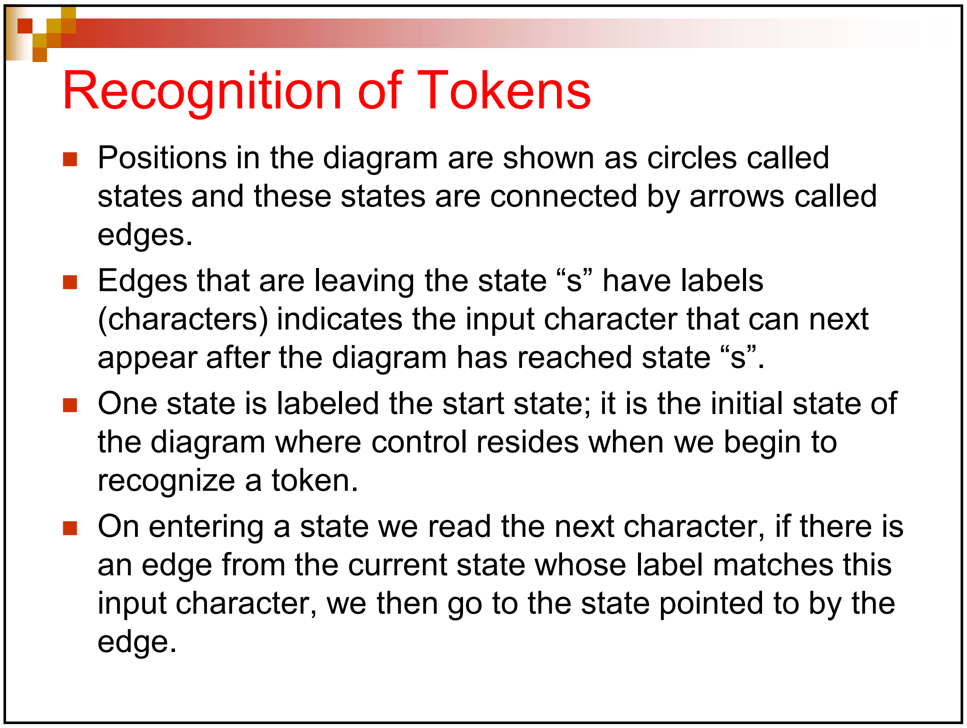
- Unsigned numbers are strings such as 5280, 39.37, 6.336E4 or 1.894E-4. The regular expression for it using regular definitions will be as:

digit → 0 | 1 | ---- | 9
digits → digit digit*
optional_fraction → . Digits | λ
optional_exponent → (E (+ | - | λ) digits) | λ
num → digits optional_fraction optional_expoent



Recognition of Tokens

- After the tokens have been specified, they need to be recognized now.
- A flowchart called transition diagram can be used to recognize tokens as well as Finite Automata.
- Transition diagram represents the actions that is taken place when the lexical analyzer is called by the parser to get the next token.
  - We move from position to position in the diagram as characters are read to keep track of the information about the characters that are seen as the input is seen.



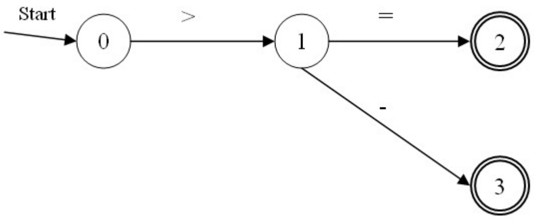
Recognition of Tokens

- Positions in the diagram are shown as circles called states and these states are connected by arrows called edges.
- Edges that are leaving the state “s” have labels (characters) indicates the input character that can next appear after the diagram has reached state “s”.
- One state is labeled the start state; it is the initial state of the diagram where control resides when we begin to recognize a token.
- On entering a state we read the next character, if there is an edge from the current state whose label matches this input character, we then go to the state pointed to by the edge.

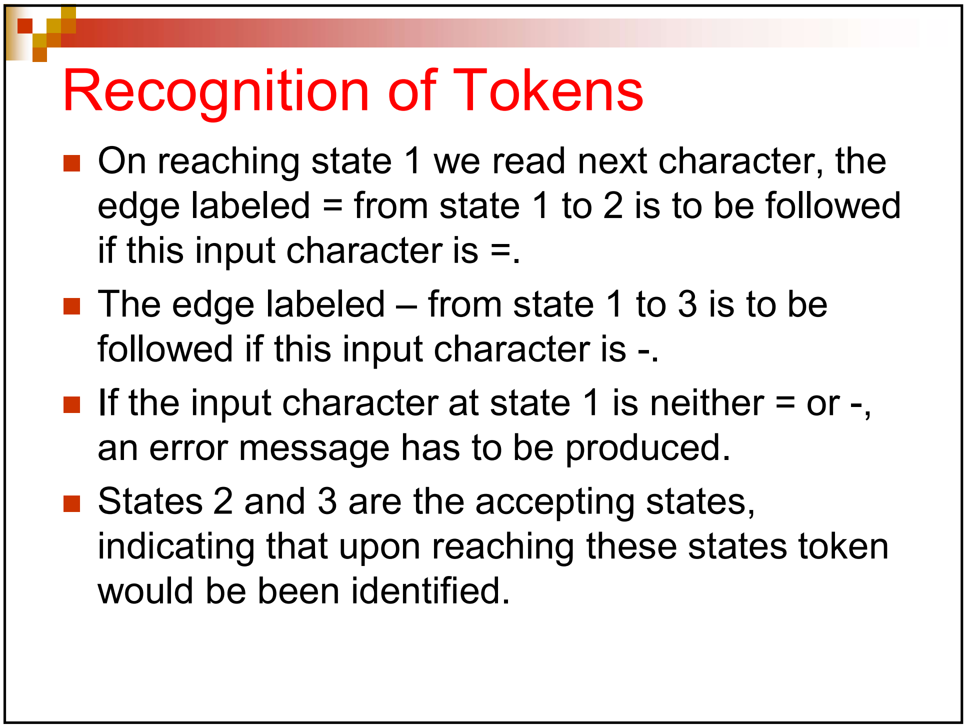
Recognition of Tokens

- One of the states is called the accepting state (final state).
- An accepting state is represented by a double circle and represents a state where a token has been recognized in the input stream.
- If the input stream does not proceeds to an accepting state, the token is not recognized and lexical analyzer displays an error message.
- For example, the transition diagram for the patterns >= and >- will be:

Recognition of Tokens

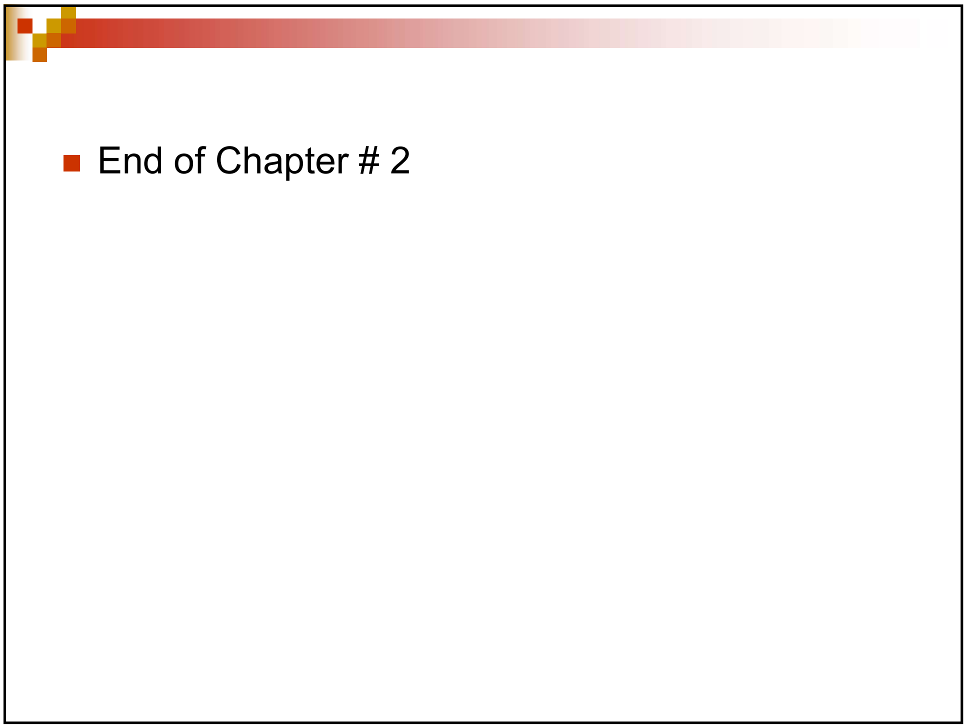


- Here 0 is the start state.
- In state 0 we read the next input character, the edge labeled > from state 0 is to be followed to state 1 if this input character is >.
- Otherwise we have failed to recognize either >= or >-



Recognition of Tokens

- On reaching state 1 we read next character, the edge labeled = from state 1 to 2 is to be followed if this input character is =.
- The edge labeled – from state 1 to 3 is to be followed if this input character is -.
- If the input character at state 1 is neither = or -, an error message has to be produced.
- States 2 and 3 are the accepting states, indicating that upon reaching these states token would be been identified.



- End of Chapter # 2