Principles of Compiler Design

Chapter 2

Lexical Analysis

Objective

At the end of this session students will be able to:

- Understand the basic roles of **lexical analyzer (LA)**: Lexical Analysis Versus Parsing, Tokens , Patterns, and Lexemes, Attributes for Tokens and Lexical Errors.
- Understand the specification of Tokens: Strings and Languages, Operations on Languages, Regular Expressions, Regular Definitions and Extensions of Regular Expressions
- Understand the generation of Tokens: Transition Diagrams, Recognition of Reserved Words and Identifiers, Completion of the Running Example, Architecture of a Transition-Diagram-Based Lexical Analysis.
- Understand the basics of Automata: **Nondeterministic Finite Automata (NFA)** Versus **Deterministic Finite Automata(DFA)**, and conversion of an NFA to a DFA.
- Understand the **Javacc**, A Lexical Analyzer and Parser Generator: Structure of Javacc file, Tokens in Javacc, Dealing with whitespaces (SKIP rules),

Introduction

- The **lexical analysis phase** of compilation breaks the **text file** (program) into smaller chunks called **tokens.**
- A **token** describes a pattern of characters having **same meaning** in the source program. (such as identifiers, operators, keywords, numbers, delimiters and so on)
  - The lexical analysis phase of the compiler is often called **tokenization**
- The **lexical analyzer (LA)** reads the stream of characters making up the source program and groups the characters into meaningful sequences called **lexemes**.
- For each lexeme, the lexical analyzer produces as output a **token** of the form **{token- name, attribute-value}** that it passes on to the subsequent phase, syntax analysis. **Where,**
  - **token- name** is an **abstract symbol** that is used during syntax analysis , and
  - **attribute-value** points to an entry in the symbol table for this token.
- Information from the symbol-table entry 'is needed for semantic analysis and code generation.

Example

- For example, suppose a source program contains the assignment statement

  posit ion = initial + rate * 60

- The characters in this assignment could be grouped into the following **lexemes** and mapped into the following tokens passed on to the syntax analyzer:

1. **position** is a lexeme that would be mapped into a token **{id, 1}**, where **id** is an abstract symbol standing for **identifier** and **1** points to the **symbol table entry for position**.
   - The symbol-table entry for an identifier holds information about the identifier, such as its name and type.
2. The assignment symbol **=** is a lexeme that is mapped into the token **{=}.** Since this token needs no attribute-value, the second component is omitted.
   - We could have used any abstract symbol such as assign for the token-name , but for notational convenience we have chosen to use the lexeme itself as the name of the abstract symbol.
3. **initial** is a lexeme that is mapped into the token **{id, 2}** , where 2 points to the symbol-table entry for initial .

Contd…

4. **+** is a lexeme that is mapped into the token {+}.

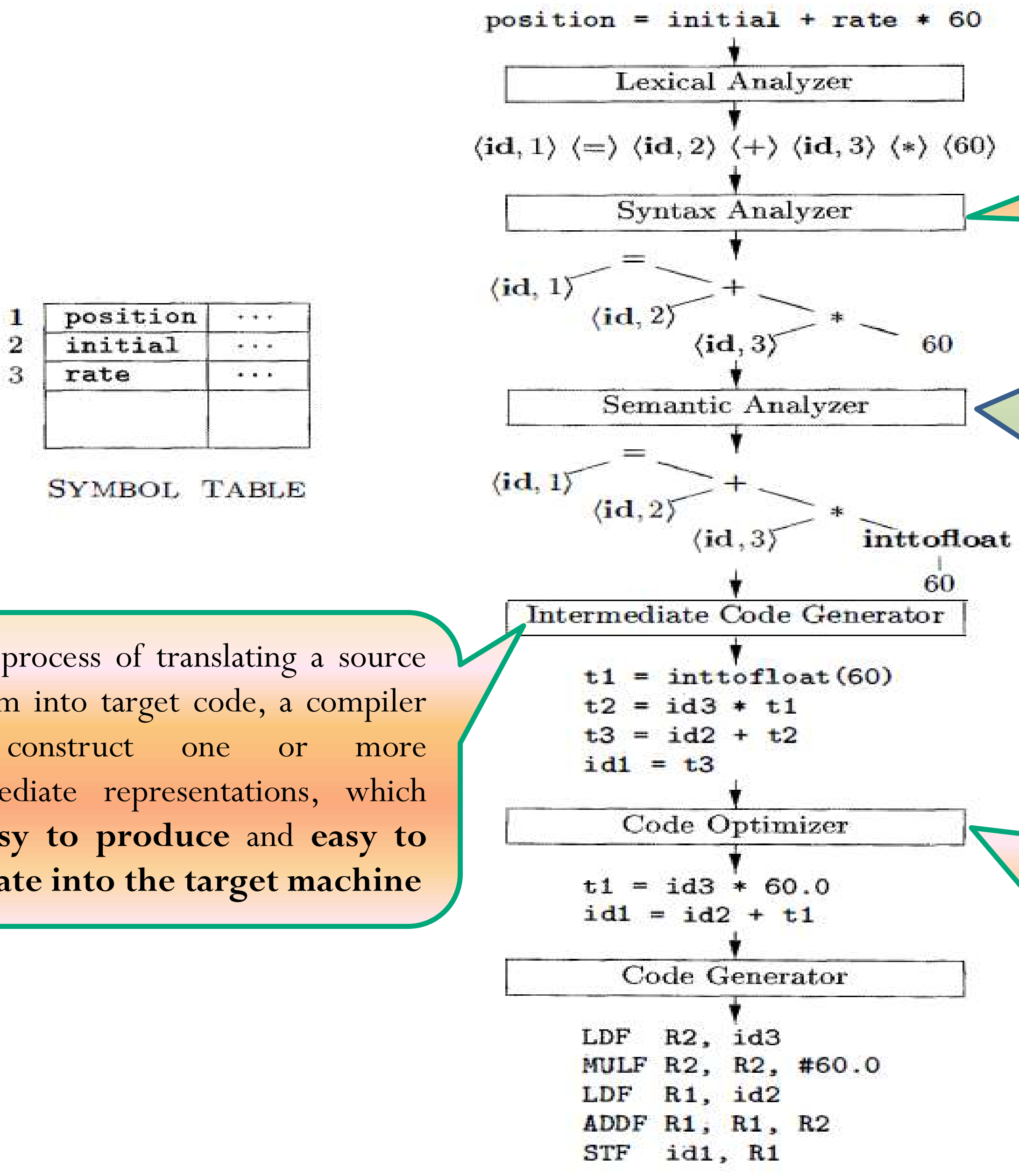
5. **rate** is a lexeme that is mapped into the token { **id, 3** }, where 3 points to the symbol-table entry for rate.

6. ***** is a lexeme that is mapped into the token {*****}.

7. **60** is a lexeme that is mapped into the token { **60** }.

☞ Blanks separating the lexemes would be discarded by the lexical analyzer.

Contd…



```
t1 = inttofloat(60)
t2 = id3 * t1
t3 = id2 + t2
id1 = t3
```

```
t1 = id3 * 60.0
id1 = id2 + t1
```

```
LDF  R2, id3
MULF R2, R2, #60.0
LDF  R1, id2
ADDF R1, R1, R2
STF  id1, R1
```

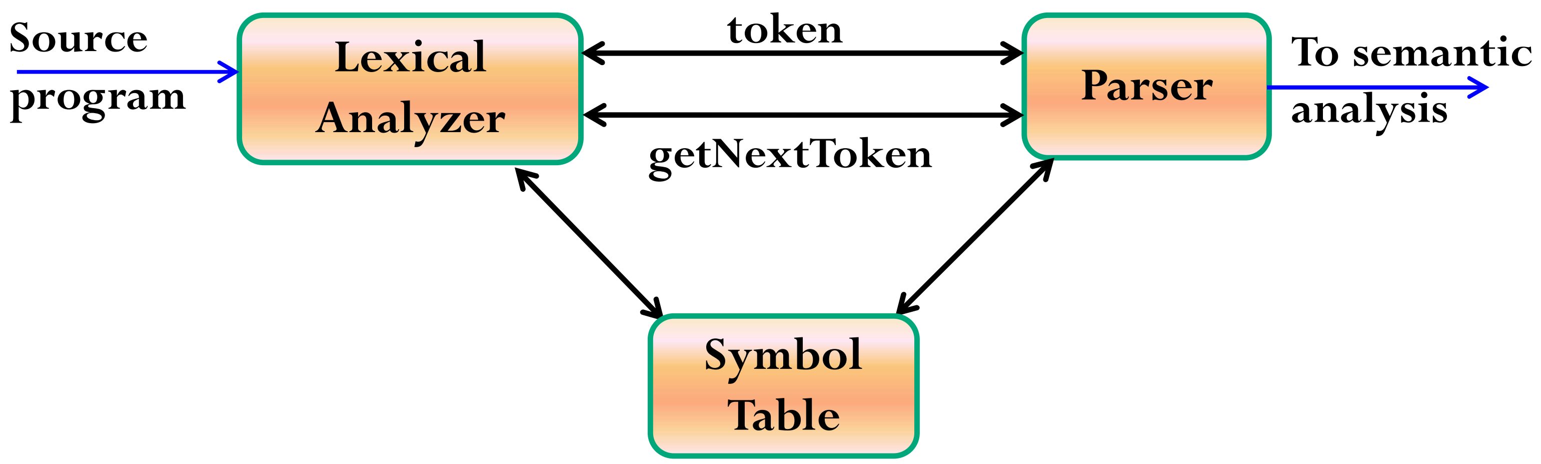
Parser uses tokens produced by the LA to create a tree-like intermediate representation that depicts the grammatical structure of the token stream.

- It uses the syntax tree and the information in the symbol table to check the source program for semantic consistency with the language definition.
- An important part of semantic analysis is type checking, where the compiler checks that each operator has matching operands

In the process of translating a source program into target code, a compiler may construct one or more intermediate representations, which are **easy to produce** and **easy to translate into the target machine**

The machine-independent code-optimization phase attempts to improve the intermediate code so that better target code will result. Usually better means faster, but other objectives may be desired, such as shorter code, or target code that consumes less power.

The Role of Lexical Analyzer



- It is common for the lexical analyzer to interact with the **symbol table**.
  - When the lexical analyzer discovers a lexeme constituting an identifier, it needs to enter that lexeme into the symbol table.
  - In some cases, information regarding the kind of identifier may be read from the symbol table by the lexical analyzer to assist it in determining the proper token it must pass to the parser.

Other tasks of Lexical Analyzer

- Since LA is the part of the compiler that reads the source text, it may perform certain other tasks besides identification of lexemes such as:
  1. Stripping out comments and whitespace (blank, newline, tab, and perhaps other characters that are used to separate tokens in the input).
  2. Correlating error messages generated by the compiler with the source program.
     - For instance, the LA may keep track of the number of newline characters seen, so it can associate a line number with each error message.
  3. If the source program uses a macro-preprocessor, the expansion of macros may also be performed by the lexical analyzer.
- Sometimes, lexical analyzers are divided into a cascade of two processes:

  A. **Scanning** consists of the simple processes that do not require tokenization of the input, such as deletion of comments and compaction of consecutive whitespace characters into one.

  B. **Lexical analysis** proper is the more complex portion, where the scanner produces the sequence of tokens as output

Lexical Analysis Vs Parsing

- There are a number of reasons why the analysis portion of a compiler is normally separated into lexical analysis and parsing (syntax analysis) phases:

1. **Simplicity of Design:-** is the most important consideration that often allows us to simplify compilation tasks.
   - For example, a parser that had to deal with comments and whitespace as syntactic units would be considerably more complex than one that can assume comments and whitespace have already been removed by the lexical analyzer.
   - If we are designing a new language, separating lexical and syntactic concerns can lead to a cleaner overall language design.
2. **Compiler efficiency is improved:-** A separate lexical analyzer allows us to apply specialized techniques that serve only the lexical task, not the job of parsing.
   - In addition, specialized buffering techniques for reading input characters can speed up the compiler significantly.
3. **Compiler portability is enhanced:-** Input-device-specific peculiarities can be restricted to the lexical analyzer.

Tokens, Patterns, and Lexemes

A lexeme is a sequence of characters in the source program that matches the pattern for a token.

Example: -5, i, sum, 100, for ; int 10 + -

Token:- is a pair consisting of a **token name** and an **optional attribute value**.

Example:

Identifiers = { i, sum }

int_Constatnt = { 10, 100, -5 }

Oppr = { +, -}

rev_Words = { for, int}

Separators = { ;, ,}

- 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 in comparison operators
- 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

Pattern is a rule describing the set of lexemes that can represent a particular token in source program

- It is a description of the form that the lexemes of a token may take.
- In the case of a keyword as a token, the pattern is just the sequence of characters that form the keyword.
- For identifiers and some other tokens, the pattern is a more complex structure that is matched by many strings.

Contd…

- One efficient but complex brute-force approach is **to read character by character to check if each one follows the right sequence of a token**.
- The below example shows a partial code for this brute-force lexical analyzer.
    - What we'd like is a set of tools that will allow us to easily create and modify a lexical analyzer that has the same run-time efficiency as the brute-force method.
    - The first tool that we use to attack this problem is Deterministic Finite Automata (DFA).

```
if (c = nextchar() == 'c') {
      if (c = nextchar() == 'l') {
              // code to handle the rest of either "class" or any identifier that starts with "cl"
       } else if (c == 'a') {
             // code to handle the rest of either "case" or any identifier that starts with "ca"
             } else {
                 // code to handle any identifier that starts with c
             }
} else if ( c = …..) {
                         // many more nested ifs for the rest of the lexical analyzer
```

Contd…

LEXEME	PATTERN	TOKEN
A1, Sum, Total	Starting with a letter & followed by letter or digit but not a keyword	ID
If Then Else	If Then Else	IF THEN ELSE
123, 123.45, 12e+07	Starting with digit followed by a digit or optional fraction and or optional exponent	NUM
<, >, <=, >=, <>, =	< or > or <= or >= or <> or =	RELOP
"lateral String"	Any sequence of character within Quotations but no quotation symbol included	"LITERAL-STRING"
Punctuation Symbol , ; ( ) { }	, ; ( ) { }	, : ( ) { }

Consideration for a Simple Design of LA

Lexical Analyzer can allow source program to be

1. **Free-Format Input:-** the alignment of lexeme should not be necessary in determining the correctness of the source program such restriction put extra load on Lexical Analyzer
2. **Blanks Significance:-** Simplify the task of identifying tokens

   E.g. Int a indicates <Int is keyword> <a is identifier>
   Inta indicates <Inta is Identifier>
3. **Keyword must be reserved:-** Keywords should be reserved otherwise LA will have to predict whether the given lexeme should be treated as a keyword or as identifier

   E.g. if then then then =else;

   Else else = then;

The above statements are misleading as then and else keywords are not reserved.

Approaches to implementation

- **Use assembly language-** Most efficient but most difficult to implement
- **Use high level languages like C**- Efficient but difficult to implement
- **Use tools like** Lex, flex, javacc… Easy to implement but not as efficient as the first two cases

Lexical Errors

- Are primarily of two kinds.

1. Lexemes whose length exceed the bound specified by the language.
   - Most languages have bound on the precision of numeric constants.
   - A constant whose bound exceeds this bound is a lexical error.
2. Illegal character in the program
   - Characters like ~, ∀, ©, ® occurring in a given programming language (but not within a string or comment) are lexical errors.
3. Un terminated strings or comments.

Handling Lexical Errors

- It is hard for a LA to tell , without the aid of other components, that there is a source-code error.
  - For instance, if the string **fi** is encountered for the first time in a java program in the context : **fi** (a == 2*4+3) a lexical analyzer cannot tell whether **fi** is a misspelling of the keyword **if** or an undeclared function identifier.
- However, suppose a situation arises in which the lexical analyzer is **unable to proceed** because none of the patterns for tokens matches any prefix of the remaining input.
  - The simplest recovery strategy is "**panic mode**" recovery.
  - We delete successive characters from the remaining input , until the lexical analyzer can find a well-formed token at the beginning of what input is left.
  - This recovery technique may confuse the parser, but in an interactive computing environment it may be quite adequate.

Contd…

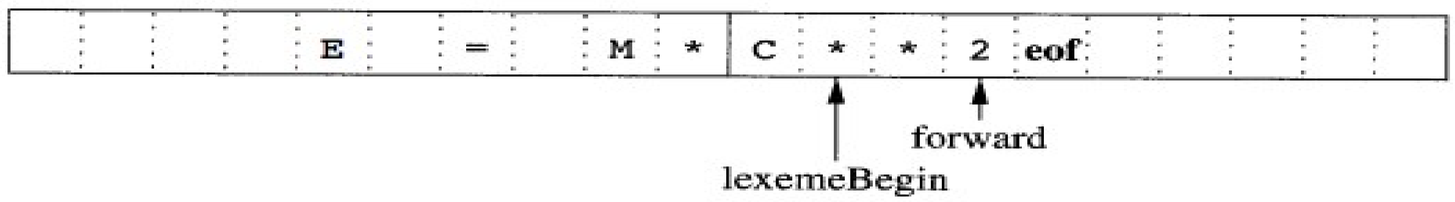
- Other possible error-recovery actions are:

1. Insert a missing character into the remaining input.
2. Replacing an incorrect character by a correct character
3. Transpose two adjacent characters(such as , fi=>if).
4. Deleting an extraneous character
5. Pre-scanning

Input Buffering

- There are some ways that the task of reading the source program can be speeded
- This task is made difficult by the fact that we often have to look one or more characters beyond the next lexeme before we can be sure we have the right lexeme
  - In C language: we need to look after -, = or < to decide what token to return
  - We shall introduce a two-buffer scheme that handles large loo-kaheads safely
- We then consider an improvement involving sentinels that saves time checking for the ends of buffers
- Because of the amount of time taken to process characters and the large number of characters that must be processed during compilation of a large source program, specialized buffering techniques have been developed to reduce the amount of overhead to process a single input character
  - An important scheme involves **two buffers** that are alternatively reloaded, as shown in the Figure next slide.

Contd…



- Each buffer is of the same size **N**, and N is usually the size of a disk block, e.g., 4096 bytes
- Using one system read command we can read N characters into a buffer, rather than using one system call per character
  - If fewer than N characters remain in the input file, then a special character, represented by eof, marks the end of the source file and is different from any possible character of the source program
- Two pointers to the input are maintained:
  1. **Pointer lexemeBegin**, marks the beginning of the current lexeme, whose extent we are attempting to determine
  2. **Pointer forward** scans ahead until a pattern match is found

Contd…

- Once the lexeme is determined, **forward** is set to the character at its right end (involves retracting)
  - Then, after the lexeme is recorded as an attribute value of the token returned to the parser, **lexemeBegin** is set to the character immediately after the lexeme just found
- Advancing forward requires that we first test whether we have reached the end of one of the buffers, and if so, we must reload the other buffer from the input, and move forward to the beginning of the newly loaded buffer

Sentinels

- If we use the previous scheme, we must check each time we advance forward, that we have not moved off one of the buffers; if we do, then we must also reload the other buffer
  - Thus, for each character read, we must make **two tests**: **one for the end of the buffer**, and **one to determine which character is read**
  - We can combine the buffer-end test with the test for the current character if we extend each buffer to hold sentinel character at the end

Contd…

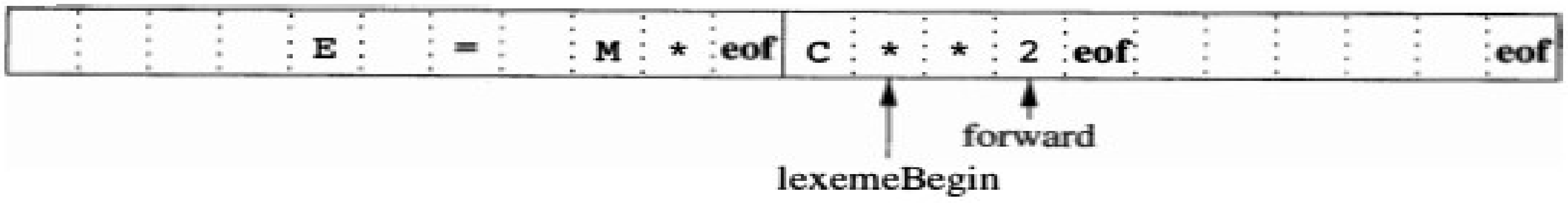


Fig. Sentinels at the end of each buffer

- The **sentinel** is a special character that cannot be part of the source program, and a natural choice is the character **eof**
  - Note that eof retains its use as a marker for the end of the entire input
- Any eof that appears other than at the end of buffer means the input is at an end
- The above Figure shows the same arrangement as Figure in slide no. 18, but with the sentinels added.

```
Switch (*forward++) {
    case eof:
      if (forward is at end of first buffer) {
                reload second buffer;
                forward = beginning of second buffer;
           }
      else if {forward is at end of second buffer) {
                reload first buffer;\
                forward = beginning of first buffer;
           }
      else /* eof within a buffer marks the end of input */
               terminate lexical analysis;
                break;
                cases for the other characters;
}
```

Specification of Tokens

- Regular expressions are an important notation for specifying lexeme patterns.
- While they cannot express all possible patterns, they are very effective in specifying those types of patterns that we actually need for tokens.
  - In theory of compilation regular expressions are used to formalize the specification of tokens
  - Regular expressions are means for specifying regular languages

Strings and Languages

- An ***alphabet*** is any finite set of symbols. Typical examples of symbols are letters , digits, and punctuation.
  - The set {0, 1 } is the binary alphabet.
  - **ASCII** is an important example of an alphabet ; it is used in many software systems.
  - **Unicode** , which includes approximately 100,000 characters from alphabets around the world, is another important example of an alphabet .

Contd…

- A string over an alphabet is a finite sequence of symbols drawn from that alphabet .
  - In language theory, the terms "sentence" and "word" are often used as synonyms for "string."
  - The length of a string **s**, usually written |**s**|, is the number of occurrences of symbols in **s**.
  - For example, **banana** is a string of length six.
  - The empty string, denoted **ε**, is the string of length zero.
- **A language is any countable set of strings over some fixed alphabet.**
  - Abstract languages like ϕ , the empty set, or {**ε**} , the set containing only the empty string, are languages under this definition.
  - So too are the set of all syntactically well-formed java programs and the set of all grammatically correct English sentences, although the latter two languages are difficult to specify exactly.
  - **Note that the definition of "language" does not require that any ":meaning be ascribed to the strings in the language.**

Terms for Parts of String

The following string-related terms are commonly used:

1. A **prefix** of string S is any string obtained by removing zero or more symbols from the end of s.
   - For example, **ban**, **banana**, and **ε** are prefixes of **banana**.
2. A **suffix** of string s is any string obtained by removing zero or more symbols from the beginning of s.
   - For example, **nana**, **banana**, and **ε** are suffixes of **banana**.
3. A **substring** of s is obtained by deleting any prefix and any suffix from s.
   - For instance, **banana**, **nan**, and **ε** are substrings of **banana**.
4. The proper prefixes, suffixes, and substrings of a string **s** are those, **prefixes**, **suffixes**, and **substrings**, respectively, of **s** that are not **ε** or not equal to **s** itself.
5. A **subsequence** of **s** is any string formed by deleting zero or more not necessarily consecutive positions of **s**.
   - For example, **baan** is a subsequence of **banana**.

Operations on Languages

The following string-related terms are commonly used:

- In lexical analysis, the most important operations on languages are union, concatenation, and closure, which are defined formally below:

1. **Union (U):-** is the familiar operation on sets.

   Example Union of L and M, **L U M = {s|s is in L or s is in M}**

2. **Concatenation :-** is all strings formed by taking a string from the first language and a string from the second language, in all possible ways , and concatenating them.

   Example Concatenation of L and M, **LM = {st|s is in L and t is in M}**

3. **Kleene closure:-** the kleene closure of a language L, denoted by **L***, is the set of strings you get by concatenating L zero, or more times.

   Example Kleene closure of L,

Note that:- L^0 , the "concatenation of L zero times," is defined to be {Ɛ} , and inductively, L^i is L^{i-1} L.

4. **Positive closure:-** is the positive closure of a language L, denoted by L^+, is the same as the Kleene closure, but without the term L^0 . That is, **Ɛ will not be in L^+** unless it is **L** in itself.

   Example Kleene closure of L,

Contd…

Let L be the set of letters {A, B , . . . , Z , a, b, . . . , z} and let D be the set of digits {0, 1, . . . 9}. We may think of L and D in two, essentially equivalent, ways.

- One way is that L and D are, respectively, the alphabets of uppercase and lowercase letters and of digits.
- The second way is that L and D are languages, all of whose strings happen to be of length one.

- Here are some other languages that can be constructed from languages L and D, using the above operators:
  1. **L U D** is the set of letters and digits - strictly speaking the language with 62 strings of length one, each of which strings is either one letter or one digit.
  2. **LD** is the set of 520 strings of length two, each consisting of one letter followed by one digit.
  3. **L^4** is the set of all 4-letter strings.
  4. **L*** is the set of ail strings of letters, including E, the empty string.
  5. **L(L U D)*** is the set of all strings of letters and digits beginning with a letter.
  6. **D^+** is the set of all strings of one or more digits.

Regular Expressions

- In theory of compilation regular expressions are used to formalize the specification of tokens
  - Regular expressions are means for specifying regular languages

Example: ID → letter_(letter|digit)*

letter →A|B|…|Z|a|b|…|z|_

digit →0|1|2|…|9

- Each regular expression is a pattern specifying the form of strings
- The regular expressions are built recursively out of smaller regular expressions, using the following two rules:

R1:- ε is a regular expression, and $L(\varepsilon) = \{\varepsilon\}$, that is , the language whose sole member is the empty string.

R2:- If **a** is a symbol in Σ then a is a regular expression, $L(a) = \{a\}$, that is , the language with one string, of length one , with **a** in its one position.

Note:- By convention, we use italics for symbols, and boldface for their corresponding regular expression.

- Each regular expression ***r*** denotes a language **L(r)**, which is also defined recursively from the languages denoted by r' s sub expressions.

Contd…

- There are **four parts** to the induction whereby larger regular expressions are built from smaller ones.
- Suppose r and s are regular expressions denoting languages L(r) and L(s) , respectively**.**
  1. **(r) |(s)** is a regular expression denoting the language **L(r) U L(s) .**
  2. **(r) (s)** is a regular expression denoting the language **L(r)L (s) .**
  3. **(r)*** is a regular expression denoting **(L (r))*.**
  4. **(r)** is a regular expression denoting **L(r) .**
- This last rule says that we can add additional pairs of parentheses around expressions without changing the language they denote.
- Regular expressions often contain **unnecessary** pairs of parentheses .
- We may drop certain pairs of parentheses if we adopt the conventions that :
  a. The unary operator * has highest precedence and is left associative.
  b. Concatenation has second highest precedence and is left associative.
  c. has lowest precedence and is left associative.

  Example:- (a) |((b) *(c)) can be represented by **a|b*c**. Both expressions denote the set of strings that are either a single **a** or are **zero or more b' s followed by one c**.

Contd…

- A language that can be defined by a regular expression is called **a regular set**.
- If two regular expressions **r** and **s** denote the same regular set , we say they are **equivalent** and write **r = s**.
  - For instance, **(a|b) = (b|a)**.
- There are a number of algebraic laws for regular expressions; each law asserts that expressions of two different forms are equivalent.

LAW	DESCRIPTION
r|s = s|r	| is commutative
r|(s|t) = (r|s)|t	|is associative
r(st) = (rs)t	Concatenation is associative
r(s|t) = rs|rt;	Concatenation distributes over |
(s|t)r= sr|tr	
Ɛr=rƐ=r	**Ɛ** is the identity for concatenation
r* = (r|Ɛ)*	**Ɛ** is guaranteed in a closure
r = r***	***** is idempotent

Table: The algebraic laws that hold for arbitrary regular expressions **r, s,** and **t**.

Regular Definitions

- If Σ is an alphabet of basic symbols, then a regular definition is a sequence of definitions of the form :

  $d_1 \rightarrow r_1$

  $d2 \rightarrow r_2$

  …

  $dn \rightarrow r_n$

Where

1. Each $\mathbf{d_i}$ is a new symbol, not in Σ and not the same as any other of the d's, and
2. Each $\mathbf{r_i}$ is a regular expression over the alphabet $\Sigma \cup \{ d_1, d_2, \ldots, d_{i-1}\}$.

Examples

(a) Regular Definition for Java Identifiers

ID $\rightarrow$ letter(letter | digit)*

letter $\rightarrow$A | B | … | Z | a | b | … | z | _

digit $\rightarrow$0 | 1 | 2 | … | 9

(b) Regular Definition for Java statements

stmt $\rightarrow$ if expr then stmt

| if expr then stmt else stmt

| ε

expr $\rightarrow$ term relop term

| term

term $\rightarrow$ id

| number

(c) Regular Definition for unsigned numbers such as 5280 , 0.0 1 234, 6. 336E4, or 1. 89E-4

digit $\rightarrow$0 | 1 | 2 | … | 9

digits $\rightarrow$digit digit*

optFrac$\rightarrow$. digts | ε

optExp$\rightarrow$ (E(+ | - | ε) digts | ε

number $\rightarrow$ digits optFrac OptExp

Extensions of Regular Expression

- Many extensions have been added to regular expressions to enhance their ability to specify string patterns.
- Here are few notational extensions:

1. **One or more instances(+):-** is a unary postfix operator that represents the positive closure of a regular expression and its language. That is , if r is a regular expression, then (r)+ denotes the language (L(r)) + .
   1. The operator + has the same precedence and associativity as the operator *****.
   2. Two useful algebraic laws, $\mathbf{r^*=r^+|\varepsilon}$ and $\mathbf{r^+=rr^*=r^*r}$.
2. **Zero or one instance(?):-** **r?** is equivalent to **r|Ɛ** , or put another way, **L(r?) = L(r) U** {**Ɛ**}.
   - The **?** operator has the same precedence and associativity as ***** and +.
3. **Character classes**. A regular expression $\mathbf{a_1|a_2|\cdots|a_n}$, where the $\mathbf{a_i}$'s are each symbols of the alphabet , can be replaced by the shorthand $\mathbf{[a_1a_2\cdots a_n]}$.

   Example:
   - **[abc]** is shorthand for **a| b | c**, and
   - **[a- z]** is shorthand for **a |b |c |··· |z**.

Example

- Using the above short hands we can rewrite the regular expression for examples a and c in slide number 29 as follows :

Examples (a) Regular Definition for Java Identifiers

ID → letter(letter | digit)*

letter →[A-Za-z_]

digit →[0-9]

(b) Regular Definition for unsigned numbers such as 5280 , 0.0 1 234, 6. 336E4, or 1. 89E-4

digit →[0-9]

digits →digit$^+$

number → digits (. digits)? (E [+ -]? digits)?

Exercises

1. Consult the language reference manuals to determine
   - A. The sets of characters that form the input alphabet (excluding those that may only appear in character strings or comments) ,
   - B. The lexical form of numerical constants, and
   - C. The lexical form of identifiers , for C++ and java programming languages.
2. Describe the languages denoted by the following regular expressions:
   - A. a(a | b) *a
   - B. ((Ɛ | a)b *)*
   - C. (a | b) *a (a | b) (a | b)
   - D. a* ba*ba*ba*.
   - E. (aa | bb) * ((ab | ba) (aa| bb) * (ab | b a) (aa | bb)*)*

Contd…

3. Write regular definitions for the following languages:

   A. All strings of lowercase letters that contain the five vowels in order.

   B. All strings of lowercase letters in which the letters are in ascending lexicographic order.

   C. All strings of binary digits with no repeated digit s.

   D. All strings of binary digits with at most one repeated digit.

   E. All strings of a ' s and b's where every a is preceded by b.

   F. All strings of a 's and b's that contain substring abab.

Recognition of Regular Expressions

1. Starting point is the language grammar to understand the tokens:

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

2. The next step is to formalize the patterns:

   digit → [0-9]
   Digits → digit+
   number → digit(.digits)? (E[+-]? Digit)?
   letter → [A-Za-z_]
   id → letter (letter|digit)*
   If → if
   Then → then
   Else → else
   Relop → < | > | <= | >= | = | <>

3. We also need to handle whitespaces:

   ws → (blank | tab | newline)$^+$

Transition Diagram

Lexemes	Token Names	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
>	relop	GT
>=	relop	GE

Table: Tokens, their patterns, and attribute values

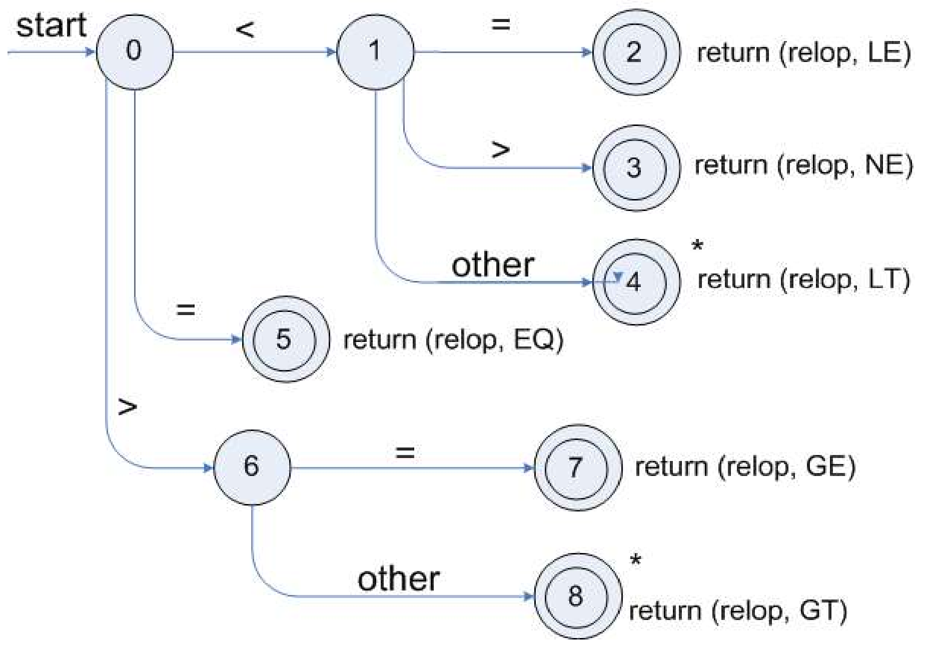


Fig. Transition diagram for relop

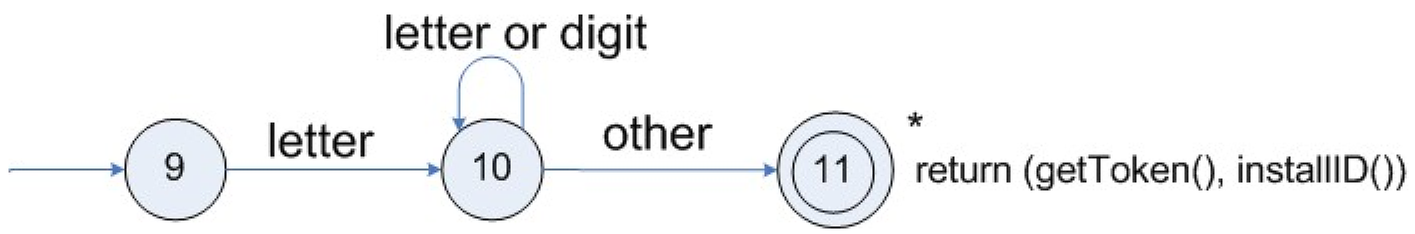


Fig. Transition diagram Transition diagram for reserved words and identifiers

Contd…

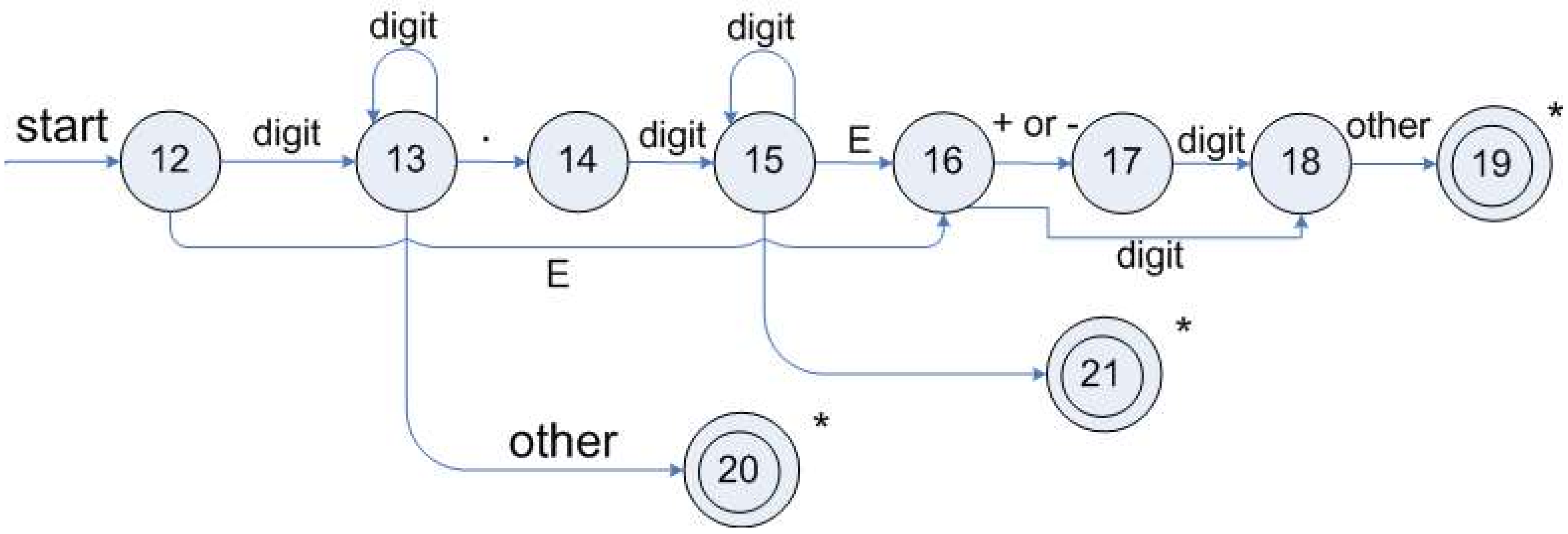


Fig. Transition diagram for unsigned numbers

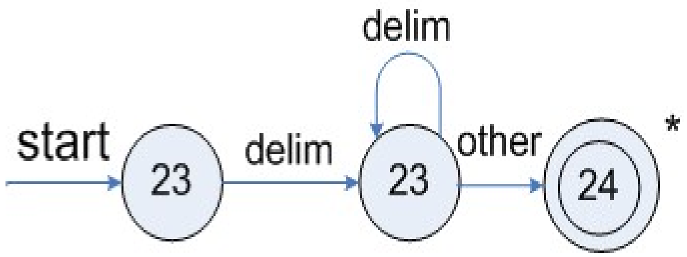


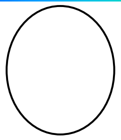
Fig. Transition diagram for white spaces where **delim** represents one or more whitespace characters

Finite Automata

- The lexical analyzer tools use **finite automata,** at the heart of the transition**,** to convert the input program into a lexical analyzer .
- These are essentially graphs, like transition diagrams, with a few differences:
  1. Finite automata are recognizers ; they simply say "yes" or "no" about each possible input string.
  2. Finite automata come in two flavors :
     - A. **Nondeterministic finite automata (NFA)** have no restrictions on the labels of their edges . A symbol can label several edges out of the same state, and ε, the empty string, is a possible label.
     - B. **Deterministic finite automata (DFA)** have, for each state, and for each symbol of its input alphabet exactly one edge with that symbol leaving that state.
- Both deterministic and nondeterministic finite automata are capable of recognizing the same languages .
  - In fact these languages are exactly the same languages , called the **regular languages**, that regular expressions can describe.

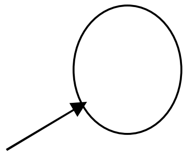
Finite Automata State Graphs

A state



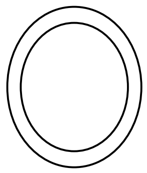
The start state

(Initial State)

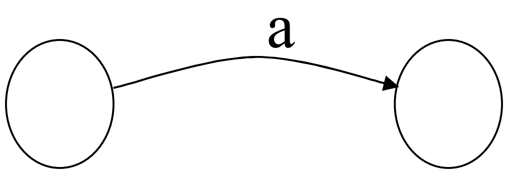


Accepting state

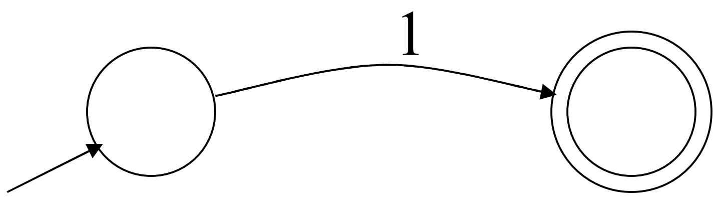
(Final State)



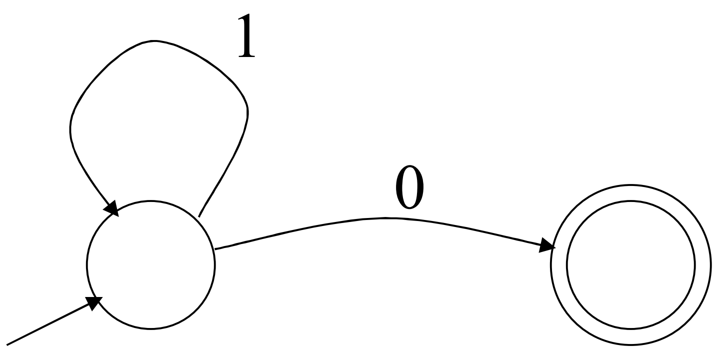
A Transition



Example 1: A finite automaton that accepts only "1"



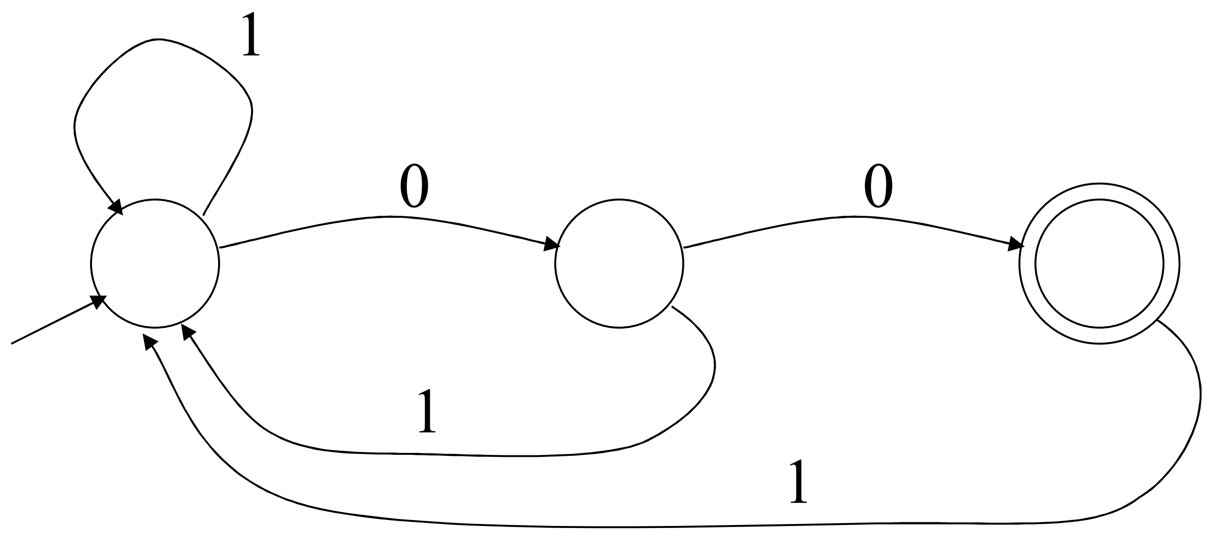
Example 2: A finite automaton accepting any number of 1's followed by a single 0



Q: Check that "1110" is accepted but "110…" is not?

Contd…

Question: What language does this recognize?

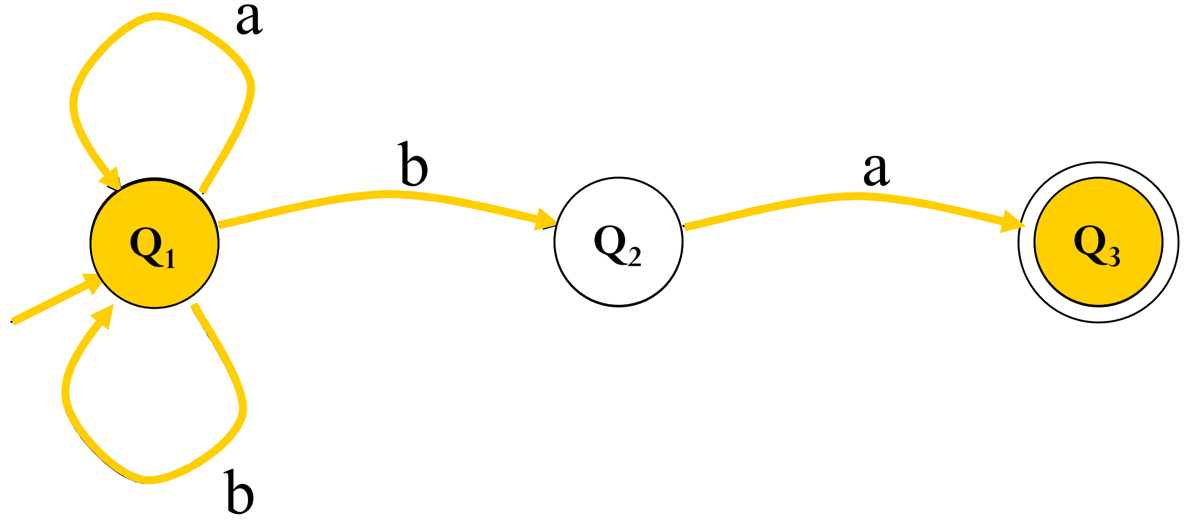


Nondeterministic Finite Automata (NFA)

- A nondeterministic finite automaton (NFA) consists of:
  1. A finite set of states **S**.
  2. A set of input symbols Σ, the input alphabet. We assume that $\boldsymbol{\varepsilon}$, which stands for the empty string, is never a member of Σ .
  3. A transition function that gives, for each state, and for each symbol in $\Sigma \cup \{\varepsilon\}$ a set of next states.
  4. A state $\mathbf{s_0}$ from **S** that is distinguished as the **start state** (or **initial state**) .
  5. A set of states ***F***, a subset of **S**, that is distinguished as the **accepting states** (or **final states**) .
- We can represent either an NFA or DFA by a **transition graph**, where the **nodes** are **states** and the **labeled edges** represent the **transition function**.
  - There is an edge labeled **a** from state **s** to state **t** if and only if **t** is one of the next states for state **s** and input **a**.
- This graph is very much like a transition diagram, except:
  1. The same symbol can label edges from one state to several different states, and
  2. An edge may be labeled by $\boldsymbol{\varepsilon}$ instead of, or in addition to, symbols from the input alphabet.

Contd…

- An NFA can get into multiple states



Input: **a b a**

Rule: NFA accepts if it can get in a final state

Transition Tables

- We can also represent an NFA by a transition table, whose **rows** correspond to **states**, and whose **columns** correspond to the **input symbols** and Ɛ.
  - The entry for a given state and input is the value of the transition function applied to those arguments.
  - If the transition function has no information about that state-input pair, we put ϕ in the table for the pair.

Example:- The transition table for the NFA on the pervious slide is represented as:

The transition table has the **advantage** **that we can easily find the transitions on a given state and input.**

Its **disadvantage** is that it takes a lot of space, when the input alphabet is large, yet most states do not have any moves on most of the input symbols.

Input State	a	b	Ɛ
Q_1	Q_1	$\{Q_1, Q_2\}$	ϕ
Q_2	Q_3	ϕ	ϕ
Q_3	ϕ	ϕ	ϕ

Acceptance of Input String by Automata

- An NFA accepts input string **x** if and only if there is **some path in the transition graph** from the start state to one of the **accepting(Final) states**, such that the symbols along the path spell out **x**.
  - Note that **ε** labels along the path are effectively ignored, since the empty string does not contribute to the string constructed along the path.

Example:- Strings **aaba** and **bbbba** are accepted by the NFA in slide 40.

- The **language defined (or accepted)** by an NFA is the set of strings labeling some path from the start to an accepting state.
  - As was mentioned above, the NFA of slide # 40 defines the same language as does the regular expression **(a|b)* ba**, that is, all strings from the alphabet {**a, b**} that end in **ba**.
  - We may use L(A) to stand for the language accepted by automaton A.

Deterministic Finite Automata (DFA)

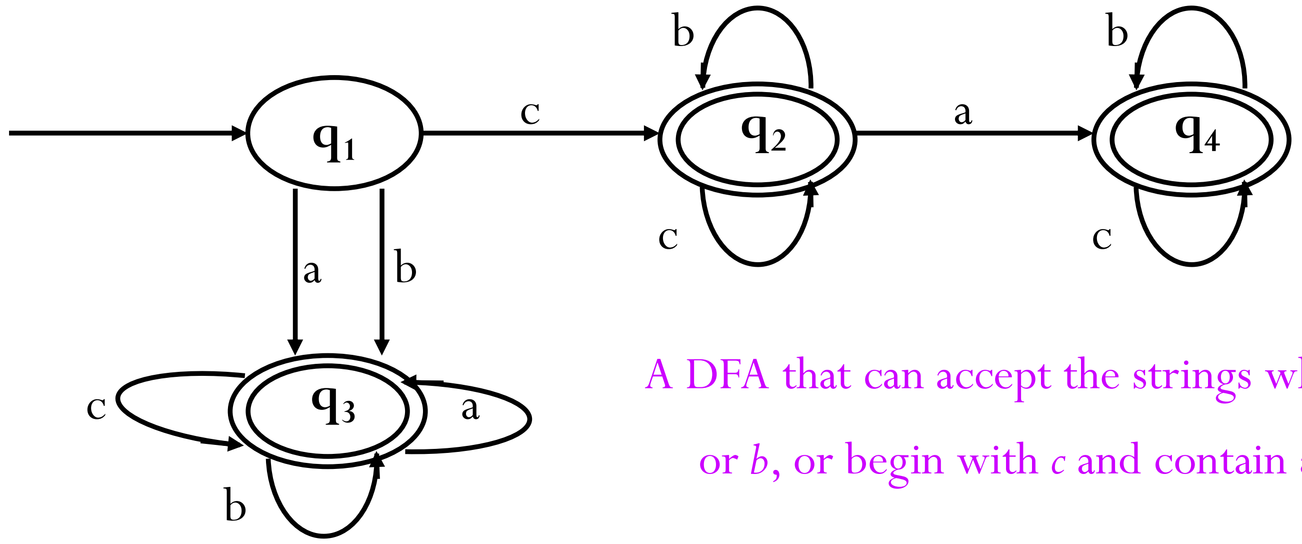
- A deterministic finite automaton (DFA) is a special case of an NFA where:
  1. There are no moves on input ε , and
  2. For each state **S** and input symbol **a**, there is exactly one edge out of **s** labeled **a**.
- If we are using a transition table to represent a DFA, then each entry is a single state.
  - we may therefore represent this state without the curly braces that we use to form sets.
- While the NFA is an abstract representation of an algorithm to recognize the strings of a certain language, the DFA is a **simple**, **concrete algorithm** for recognizing strings.
- It is fortunate indeed that every regular expression and every NFA can be converted to a DFA accepting the same language, because it is the DFA that we really implement or simulate when building lexical analyzers.
- **DFA recognize strings:-** When a string is fed into a DFA, if the DFA recognizes the string, it **accepts** the string otherwise it **rejects** the string.

Contd…

- A DFA is a collection of states and transitions:- Given the input string, the transitions tell us how to move among the states.
  - One of the states is denoted as the **initial state,** and a subset of the states are **final states**.
  - We start from the initial state, move from state to state via the transitions, and check to see if we are in the final state when we have checked each character in the string.
  - If we are, then the string is accepted, otherwise, the string is rejected
- A DFA is a quintuple, a machine with five parameters, **$M = (Q, \Sigma, \delta, q0, F)$**, where
  - **Q** is a finite set of states
  - Σ is a finite set called the alphabet
  - **δ** is a total function from (**$Q \times \Sigma$**) to **Q** known as transition function (a function that takes a state and a symbol as inputs and returns a state)
  - **q_0** an elements of **Q** is the start state, and
  - **F** is **subset of Q** called **final states**

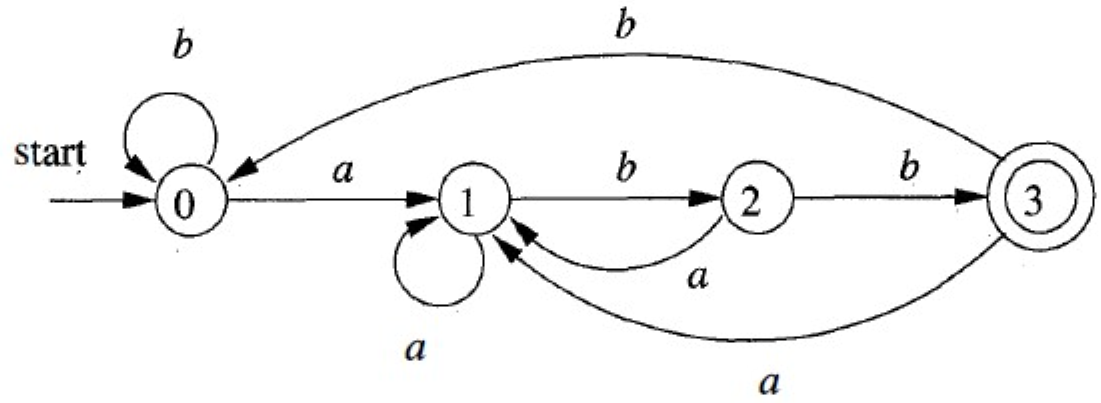
Example

1)



A DFA that can accept the strings which begin with *a* or *b*, or begin with *c* and contain at most one *a*.

2)

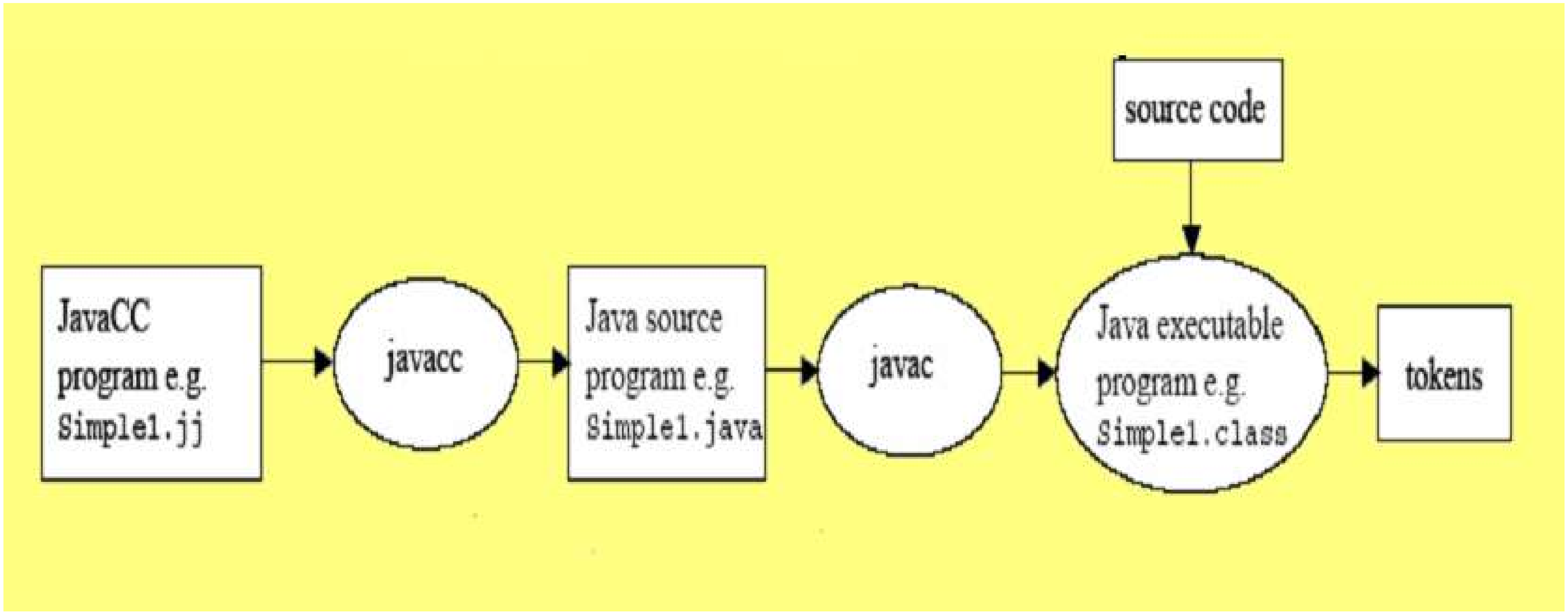


A DFA accepting **(a|b) * abb**

JavaCC- A Lexical Analyzer and Parser Generator

- **Java Compiler Compiler (JavaCC)** is the most popular Lexical analyzer and parser generator for use with Java applications.
    - In addition to this, JavaCC provides other standard capabilities related to parser generation such as tree building (via a tool called JJTree included with JavaCC), actions, debugging, etc.
- JavaCC takes a set of **regular expressions** as **input** that describe tokens
    - **Creates a DFA** that recognizes the input set of tokens, and
    - Then creates a Java class to implement that DFA
- JavaCC works with any Java VM version 1.2 or greater.
- JavaCC also creates a parser which will be explored in next chapter.

Flow for using JavaCC


source code
JavaCC program e.g. Simple1.jj
javacc
Java source program e.g. Simple1.java
javac
Java executable program e.g. Simple1.class
tokens

- JavaCC is a "top-down" parser generator.
- Some parser generators (such as yacc , bison, and JavaCUP) need a separate lexical-analyzer generator.
- With JavaCC, you can specify the tokens within the parser generator.

Structure of JavaCC File

- Input files to JavaCC have the extension ".jj"
- A sample example **simple.jj** file is shown on next slide # 50 and 51
- Several tokens can be defined in the same TOKEN block, with the rules separated by a " | ". (see example from slide # 50)
- By convention, token names are in all UPPERCASE.
- When JavaCC runs on the file simple.jj, JavaCC creates a class simpleTokenManager, which contains a method getNextToken().
- The getNextToken() method implements the DFA that recognizes the tokens described by the regular expressions
- Every time getNextToken is called, it finds the rule whose regular expression matches to the next sequence of characters in the input file, and returns the token for that rule

Contd…

- Inputfile file is used as input file for simple.jj code, as shown on slide number 52.
- When there is more than one rule that matches the input, JavaCC uses the following strategy:
  1. Always match to the longest possible string
  2. If two different rules match the same string, use the rule that appears first in the .jj file
- For example, the "else" string matches both the ELSE and IDENTIFIER rules, getNextToken() will return the ELSE token
- But "else21" matches to the IDENTIFIER token only

Simple.jj

```
PARSER_BEGIN(simple)
public class simple
{
}
PARSER_END(simple)
TOKEN_MGR_DECLS:
{
        public static int count=0;
}
SKIP:
{
        <" ">
        |<"\n">
        |<"\t">
        |<"\r">
        |<"\b">
        |<"//"(~["\n"])*"\n">
        |<"/*"> {count++;} : INNER_COMMENT
}
<INNER_COMMENT>
SKIP:
{
<"/*">{count++;}:INNER_COMMENT
}
```

```
<INNER_COMMENT>
SKIP:
{
        <~[]>
        |<"*/">
        {
         count--;
        if(count==0);  SwitchTo(DEFAULT);
        }
}

TOKEN:
{
     <ELSE:"else">
     |<FOR:"for">
     |<AND:"and">
     |<CLASS:"class">
     |<PUBLIC:"public">
     |<PROTECTED:"protected">
     |<PRIVATE:"private">
     |<DO:"do">
     |<IF:"if">
     |<WHILE:"while">
     |<INT:"int">
     |<FLOAT:"float">
     |<CHAR:"char">
     |<VOID:"void">
}
```

Contd…

```
TOKEN:
{
        <PLUS:"+">
        |<MINUS:"-">
        |<TIMES:"*">
        |<DIVIDE:"/">
        |<SEMICOLON:";">
        |<LEFT_PARENTHESIS:"(">
        |<RIGHT_PARENTHESIS:")">
        |<LEFT_SQUARE_BRACKET:"[">
        |<RIGHT_SQUARE_BRACKET:"]">
        |<LEFT_BRACE:"{">
        |<RIGHT_BRACE:"}">
        |<DOT:".">
        |<EQUAL_TO:"==">
        |<NOT_EQUAL_TO:"!=">
        |<LESS_THAN:"<">
        |<GREATER_THAN:">">
        |<LESS_THAN_OR_EQUAL_TO:"<=">
        |<GREATER_THAN_OR_EQUAL_TO:">=">
        |<ASSIGNMENT:"=">
        |<LOGICAL_AND:"&&">
        |<LOGICAL_OR:"||">
        |<NOT:"!">
        |<DOLLAR:"$">
  }
```

```
TOKEN:
{
<IDENTIFIERS:["a"-"z","A"-"Z"](["a"-"z","A"-
"Z","0"-"9"])*>
|<INTEGER_LITERAL:(["0"-"9"])+>
}
void expression():
{}
{
    term()((<PLUS>|<MINUS>) term())*
}

void term():
{}
{
    factor() ((<TIMES>|<DIVIDE>) factor())*
}

void factor():
{}
{
   <INTEGER_LITERAL>|<IDENTIFIERS>
}
```

Inputfile

```
/* this is an input file for simple.jj file
 Prepared By:
   Dileep Kumar
   Date: 02/02/2014
 ******************************************  */
public class testLA
{
   int y=7;
  float w+=2;
  char z=t*8;
  int x,y=2,z+=3;
 if(x>0)
  {
        sum/=x;
   }
  else if(x==0)
  {
    sum=x;
  }
  else {}
  }
while(x>=n)
 {
    float sum=2*3+4-6;
  }
}
```

Tokens in JavaCC

❖ When we run JavaCC on the input file simple.jj, JavaCC creates several files to implement a lexical analyzer

1. simpleTokenManager.Java: to implement the token manager class
2. simpleConstants.Java: an interface that defines a set of constants
3. Token.Java: describes the tokens returned by **getNextToken()**
4. In addition to these, the following additional files are also created:
   - ☞ simple.Java,
   - ☞ ParseException.Java,
   - ☞ SimpleCharStream.Java,
   - ☞ TokenMgrError.Java

Using the generated TokenManager in JavaCC

- To create a Java program that uses the lexical analyzer created by JavaCC, you need to instantiate a variable of type simpleTokenManager, where simple is the name of your **.jj** file.
- The constructor for simpleTokenManager requires an object of type SimpleCharStream.
- The constructor for SimpleCharStream requires a standard Java.io.InputStream.
- Thus, you can use the general lexical analyzer as follows:

```
Token t;
simpleTokenManager tm;
Java.io.InputStream infile;
tm = new simpleTokenManager(new SimpleCharStream(infile));
infile = new Java.io.FileInputstream("Inputfile.txt");
t = tm.getNextToken();
while (t.kind != simpleConstants.EOF)
{
    /* Process t */
    t = tm.getNextToken();
}
```

Thank You ...

?