Chapter 2: Lexical Analysis/Scanner

Session1

- Lexical Analyzer
- The Role of the Lexical Analyzer
- What is token?
- Tokens, Patterns and Lexemes
- Token Attributes
- Lexical Errors
- Specification of Tokens
- Recognition of Tokens

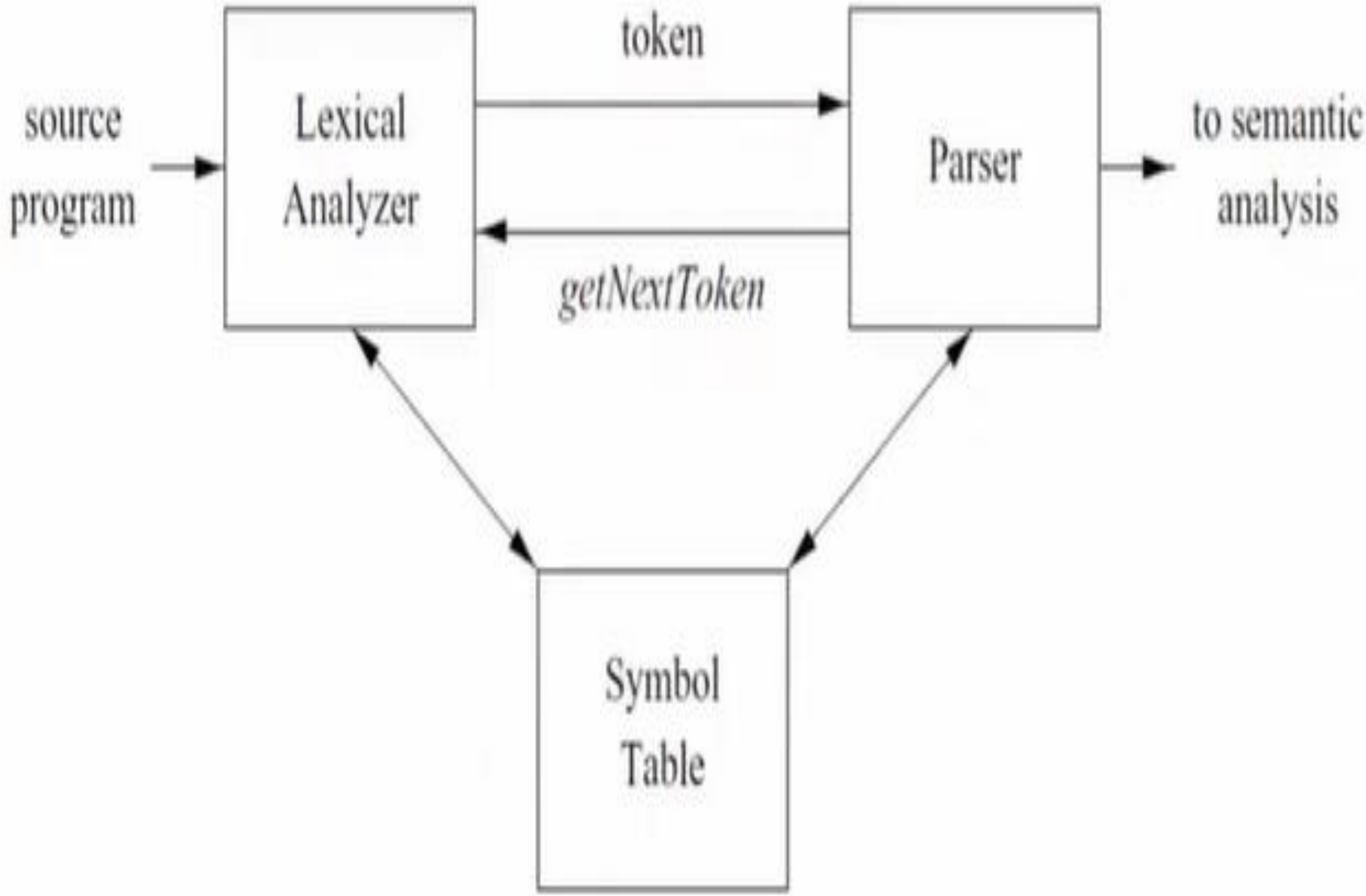
Lexical Analyzer

- **Lexical analysis** is the process of converting a sequence of characters into a sequence of tokens.
- A program or function which performs lexical analysis is called a **lexical analyzer**, **lexer**, or **scanner**.
- A **lexer** often exists as a single function which is called by a parser (syntax Analyzer) or another function.

- The tasks done by scanner are
  - Grouping input characters into tokens
  - Stripping out comments and white spaces and other separators
  - Correlating error messages with the source program
  - Macros expansion
  - Pass token to parser

Cont…

•



token
source
program
Lexical
Analyzer
Parser
to semantic
analysis
getNextToken
Symbol
Table

Cont…

- The call, suggested by the getNextToken command causes the lexical analyzer to read characters from its input until it can identify the next lexeme and produce for it the next token which it returns to the parser.
- Since the lexical analyzer is the part of the compiler that reads the source text , it may perform certain other task besides identification of lexemes.
- One such task is stripping out comment and whitespace (blank, newline, tab, and perhaps other characters that are used to separate tokens in the input)

What is token?

- Tokens correspond to sets of strings.
  - *Identifier*: strings of letters or digits, starting with a letter
  - *Integer:* a non-empty string of digits; 123, 123.45
  - *Keyword:* *“else”, “if", "begin”*
  - *Whitespace:* a non-empty sequence of blanks, newlines, and tabs
  - *Symbols:* +, -, *, / , =, <, >, ->, …
  - *Char (string) literals*: "Hello" , ‘c‘

What is token? …Example

- What do we want to do?
  - Example:

    if (i == j)

    Z = 0;

    else

    Z = 1;
  - The input is just a string of characters:

    \t if (i == j) \n \t \t z = 0;\n \t else \n \t \t z = 1;
  - Goal: Partition input string into substrings
    - Where the substrings are tokens
  - Tokens: if, (, i, ==, j,), z, =, 0, ;, else, 1, …

Tokens, Patterns and Lexemes

- Lexeme
  - Actual sequence of characters that matches a pattern and has a given token class.
  - Examples: Name, Data, x, 345,2,0,629,....
- Token
  - A *classification* for a common set of strings
  - Examples: Identifier, Integer, Float, Assign, LeftParen, RightParen,....
    - One token for all identifiers
- Pattern
  - The rules that characterize the set of strings for a token
  - Examples: [0-9]+
    - identifier: ([a-z]|[A-Z]) ([a-z]|[A-Z]|[0-9])*

Tokens, Patterns and Lexemes: Example

Token	Sample Lexemes	Informal Description of Pattern
const	const	const
if	if	if
relation	<, <=, =, < >, >, >=	< or <= or = or < > or >= or >
id	pi, count, D2	letter followed by letters and digits
num	3.1416, 0, 6.02E23	any numeric constant
literal	"core dumped "	any characters between " and "

Classifies Pattern

Actual values are critical. Info is :

1. Stored in symbol table
2. Returned to parser

Token Attributes

- More than one lexeme can match a pattern
  - The lexical analyzer must provide the information about the particular lexeme that matched.
  - For example, the pattern for token number matches both 0, 1, 934, …
- But code generator must know which lexeme was found in the source program.
  - Thus, the lexical analyzer returns to the parser a token name and an attribute value
- For each lexeme the following type of output is produced

(token-name, attribute-value)

➔ *attribute-value points to an entry in the symbol table for this token*

Example of Attribute Values

- E = M * C ** 2
  - <id, pointer to symbol table entry for E>
  - < assign_op>
  - <id, pointer to symbol table entry for M>
  - <mult_op>
  - <id, pointer to symbol table entry for C>
  - <exp_op>
  - <number, integer value 2>

Lexical Errors

- Lexical analyzer can‘t detect all errors – needs other components
- In what situations do errors occur?
  - When none of the patterns for tokens matches any prefix of the remaining input.
- However look at: fi(a==f(x)) …
  - generates no lexical error in C – subsequent phases of compiler do generate the errors
- Possible error recovery actions:
  - Deleting or Inserting Input Characters
  - Replacing or Transposing Characters
- Or, skip over to next separator to ignore problem

Specifying Tokens

- Two issues in lexical analysis.
- *How to specify tokens?*And *How to recognize the tokens giving a token specification?* (i.e. how to implement the nexttoken( ) routine)?
- How to specify tokens:
  - Tokens are specified by **regular expressions**.

- ***Regular Expressions***
  - Represent the patterns of strings in characters
  - Can not express all possible patterns, they are very effective in specifying those types of patterns that we actually need for tokens.
  - The set of strings generated by a regular expression **r** is as **L(r)**

Languages

- A language is any countable set of strings over some fixed alphabet.

Alphabet	**Language**
{0,1}	{0,10,100,1000,10000,…}
	{0,1,100,000,111,…}
{a,b,c}	{abc,aabbcc,aaabbbccc,…}
{A…Z}	{TEE,FORE,BALL…}
	{FOR,WHILE,GOTO…}
{A…Z,a…z,0…9,	{All legal PASCAL progs}
+,-,…,<,>,…}	{All grammatically correct English Sentences}
Special Languages:	Φ – EMPTY LANGUAGE
	ε – contains empty string ε only

Operations on Languages

Operation	Definition and Notation
Union of L and M	$L \cup M = \{s \mid s \text{ is in } L \text{ or } s \text{ is in } M\}$
Concatenation of L and M	$LM = \{st \mid s \text{ is in } L \text{ and } t \text{ is in } M\}$
Kleene closure of L	$L^* = \cup_{i=0}^{\infty} L^i$
Positive closure of L	$L^+ = \cup_{i=1}^{\infty} L^i$

L = {A, B, C, D } **D = {1, 2, 3}**

$L \cup D$ = {A, B, C, D, 1, 2, 3 }

L D = {A1, A2, A3, B1, B2, B3, C1, C2, C3, D1, D2, D3 }

L^2 = { AA, AB, AC, AD, BA, BB, BC, BD, CA,…,DA,… DD}

$L^4 = L^2 \ L^2$ = ??

L^* = { All possible strings of L plus $\in$ }

$L^+ = L^* - \in$

$L\ (L \cup D)$ = ??

$L\ (L \cup D)^*$ = ??

Regular Expressions

- Formal definition of Regular expression:
  - Given an alphabet Σ ,
    - (1) λ is a regular expression that denote { λ }, the set that contains the empty string
    - (2) For each a $\in \Sigma$, a is a regular expression denote {a}, the set containing the string a.
    - (3) r and s are regular expressions, then
      - o r | s is a regular expression denoting L(r) U L(s)
      - o rs is a regular expression denoting L(r) L (s)
      - o (r)* is a regular expression denoting (L (r)) *
    - Regular expression is defined together with the language it denotes.
    - Regular expressions are used to denote ***regular languages***

Regular Expressions: Example

Let $\Sigma = \{a, b\}$.

1. The regular expression $\mathbf{a}|\mathbf{b}$ denotes the language $\{a, b\}$.

2. $(\mathbf{a}|\mathbf{b})(\mathbf{a}|\mathbf{b})$ denotes $\{aa, ab, ba, bb\}$, the language of all strings of length two over the alphabet Σ. Another regular expression for the same language is $\mathbf{aa}|\mathbf{ab}|\mathbf{ba}|\mathbf{bb}$.

3. $\mathbf{a}^*$ denotes the language consisting of all strings of zero or more a's, that is, $\{\epsilon, a, aa, aaa, \ldots\}$.

4. $(\mathbf{a}|\mathbf{b})^*$ denotes the set of all strings consisting of zero or more instances of a or b, that is, all strings of a's and b's: $\{\epsilon, a, b, aa, ab, ba, bb, aaa, \ldots\}$. Another regular expression for the same language is $(\mathbf{a}^*\mathbf{b}^*)^*$.

5. $\mathbf{a}|\mathbf{a}^*\mathbf{b}$ denotes the language $\{a, b, ab, aab, aaab, \ldots\}$, that is, the string a and all strings consisting of zero or more a's and ending in b.

Regular Expressions: Algebraic Properties

LAW	DESCRIPTION
$r\|s = s\|r$	$\|$ is commutative
$r\|(s\|t) = (r\|s)\|t$	$\|$ is associative
$r(st) = (rs)t$	Concatenation is associative
$r(s\|t) = rs\|rt;\ (s\|t)r = sr\|tr$	Concatenation distributes over $\|$
$\epsilon r = r\epsilon = r$	ϵ is the identity for concatenation
$r^* = (r\|\epsilon)^*$	ϵ is guaranteed in a closure
$r^{**} = r^*$	* is idempotent

Regular Definition

- Gives names to regular expressions to construct more complicate regular expressions.
- If $\sum$ is an alphabet of basic symbols, then a *regular definition* is a sequence of definitions of the form:
  - $d_1 \rightarrow r_1$,
  - $d_2 \rightarrow r_2$,
  - $d_3 \rightarrow r_3$,
  - …
- where:
  - Each d_i is a new symbol, not in $\sum$ and not the same as any other of the d's
  - Each r_i is a regular expression over the alphabet $\sum \cup \{d_l, d_2, \ldots, d_{i-l}\}$.

Regular Definition: Example

- Example
  - **C** identifiers are strings of letters, digits, and underscores. The regular definition for the language of C identifiers.
    - *Letter_* → A | B | *C* | … | Z | a | b | … | z | _
    - *digit* → *0* | *1* | *2* | … | *9*
    - *id* → *letter*(*letter* | *digit*)*
- Unsigned numbers (integer or floating point) are strings such as 5280, 0.01234, 6.336E4, or 1.89E-4.The regular definition
  - *digit* → *0* | *1* | *2* | … | *9*
  - *digits* → *digit digit**
  - *optionalFraction* → .*digits* | ε
  - *optionalExponent* → (E(+ | - | ε) *digits*) | ε
  - *number* → *digits optionalFraction optionalExponent*

Recognition of tokens

- Recognition of tokens is the second issue in lexical analysis
  - How to recognize the tokens giving a token specification?
- Given the grammar of branching statement:

$$
\begin{array}{lcl}
stmt & \rightarrow & \textbf{if } expr \textbf{ then } stmt \\
 & | & \textbf{if } expr \textbf{ then } stmt \textbf{ else } stmt \\
 & | & \epsilon \\
expr & \rightarrow & term \textbf{ relop } term \\
 & | & term \\
term & \rightarrow & \textbf{id} \\
 & | & \textbf{number}
\end{array}
$$

- The terminals of the grammar, which are **if, then, else, relop, id,** and **number**, are the names of tokens
- The patterns for the given tokens are:

Cont…

$$
\begin{array}{rcl}
digit & \rightarrow & \texttt{[0-9]} \\
digits & \rightarrow & digit^{+} \\
number & \rightarrow & digits\ (\texttt{.}\ digits)?\ (\ \texttt{E}\ \texttt{[+-]}?\ digits\ )? \\
letter & \rightarrow & \texttt{[A-Za-z]} \\
id & \rightarrow & letter\ (\ letter \mid digit\ )^{*} \\
if & \rightarrow & \texttt{if} \\
then & \rightarrow & \texttt{then} \\
else & \rightarrow & \texttt{else} \\
relop & \rightarrow & \texttt{<} \mid \texttt{>} \mid \texttt{<=} \mid \texttt{>=} \mid \texttt{=} \mid \texttt{<>}
\end{array}
$$

Note that

$\boxed{r? \text{ is equivalent to } r|\epsilon}$

- The lexical analyzer also has the job of stripping out whitespace, by recognizing the "token" ***ws*** defined by:

$$ws \rightarrow (\ \textbf{blank} \mid \textbf{tab} \mid \textbf{newline}\ )^{+}$$

Cont…

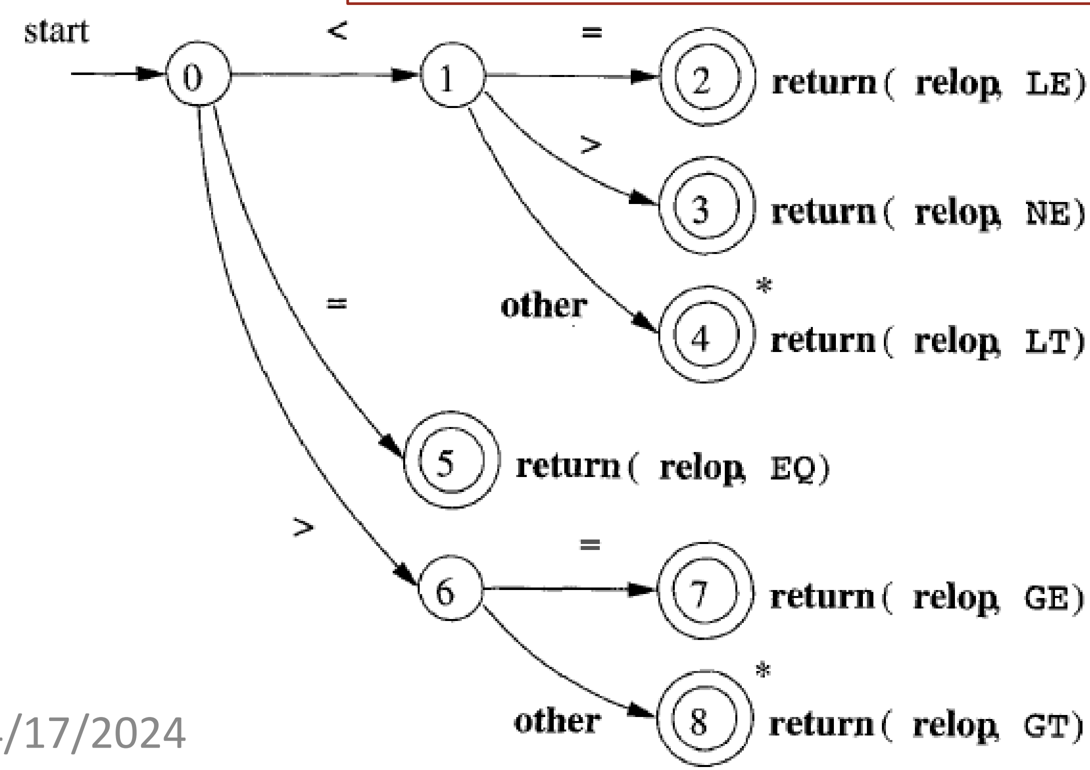
The table shows *which token name is returned to the parser and what attribute value* for each lexeme or family of lexemes.

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`
`>`	**relop**	`GT`
`>=`	**relop**	`GE`

Cont. . . . . .

- **Transition diagrams**
  - As intermediate step in the construction of a lexical analyzer, we first convert patterns into stylized flowcharts, called "*transition diagrams*"

Example Transition diagram for **relop**



* Is used to indicate that we must retract the input one position (back)

Cont. . . . . .

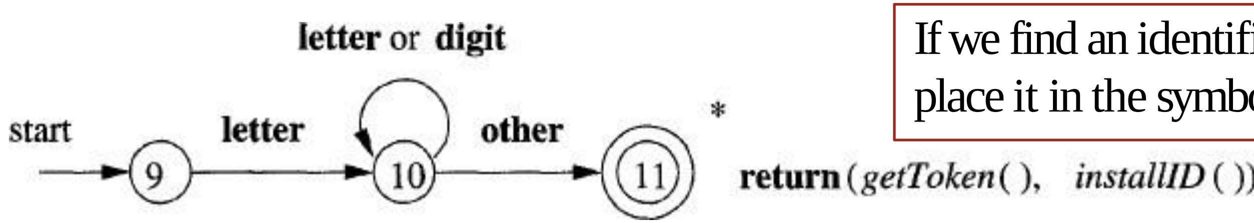
Mapping transition diagrams into C code

```
TOKEN getRelop()
{
    TOKEN retToken = new(RELOP);
    while(1) { /* repeat character processing until a return
                  or failure occurs */
        switch(state) {
            case 0: c = nextChar();
                    if ( c == '<' ) state = 1;
                    else if ( c == '=' ) state = 5;
                    else if ( c == '>' ) state = 6;
                    else fail(); /* lexeme is not a relop */
                    break;
            case 1: ...
            ...
            case 8: retract();
                    retToken.attribute = GT;
                    return(retToken);
        }
    }
}
```

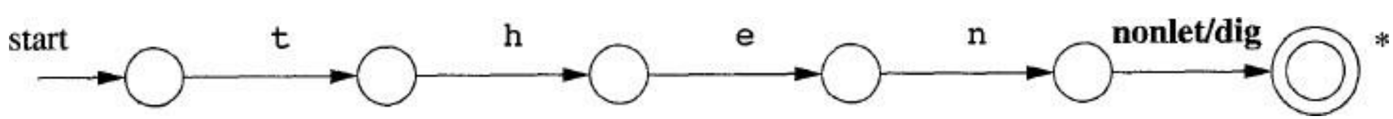
Cont…

- **Recognition of Reserved Words and Identifiers**
  - Recognizing keywords and identifiers presents a problem.
    - keywords like **if** or **then** are reserved but they look like identifiers
  - There are two ways that we can handle
    - stall the reserved words in the symbol table initially.

If we find an identifier, *installID* is called to place it in the symbol table.



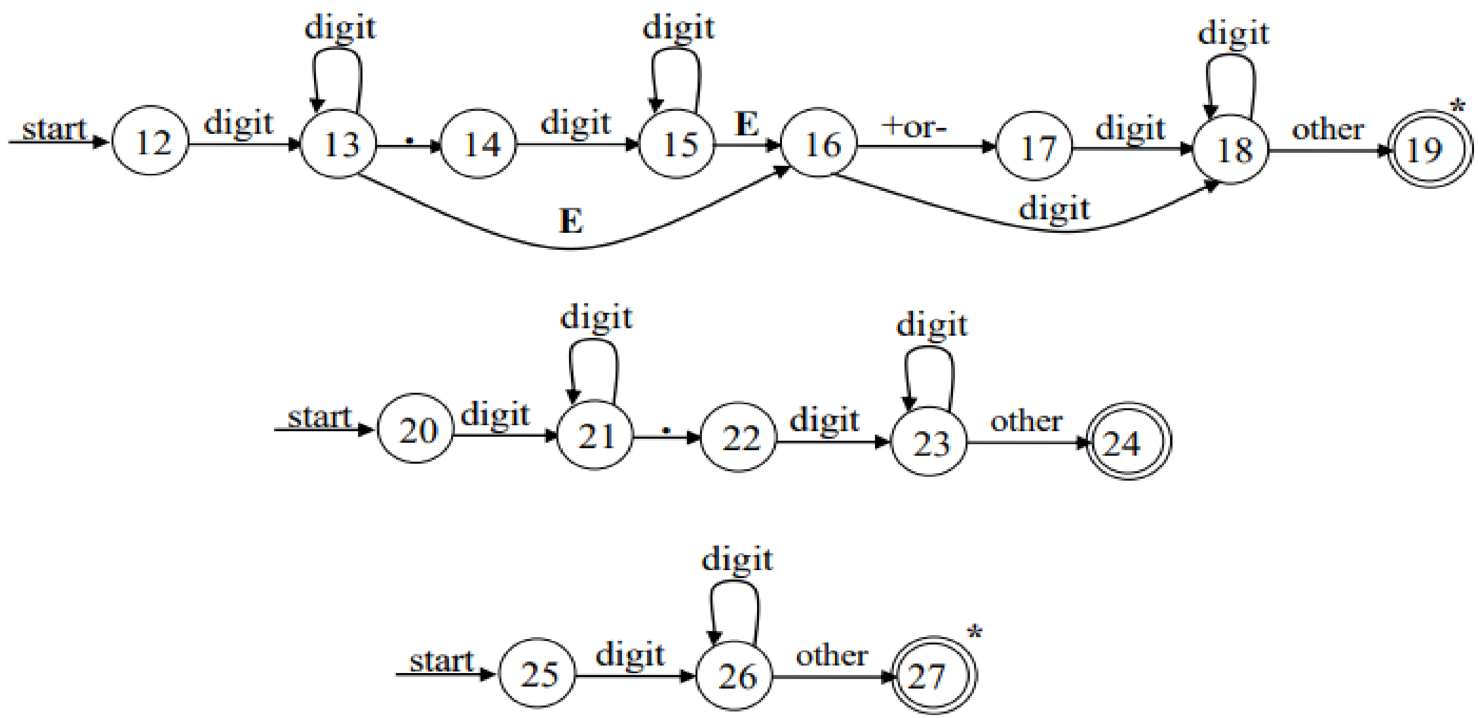
- Create separate transition diagrams for each keyword;
  - the transition diagram for the reserved word **then**



Cont…

Q1: A Transition Diagram for Unsigned Numbers

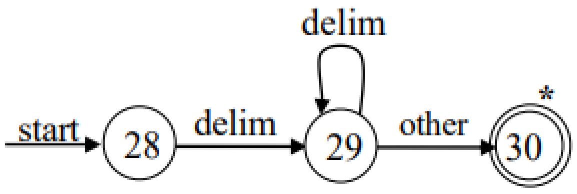
Q2. A Transition Diagram for White Space



num $\longrightarrow$ $digit^{+}(.\ digit^{+} \mid \in)(E(+ \mid - \mid \in)digit^{+} \mid \in)$

Cont…

Q2. Transition Diagram for White Space



Session2

- Finite Automata
- Nondeterministic Finite Automata
- Deterministic Finite Automata
- From Regular Expressions to Automata
- Conversion of NFA to DFA
- Design of a Lexical-Analyzer Generator
- Implementing Scanners

Finite Automata

- At the heart of the transition diagram is the formalism known as *finite automata*
- A *finite automaton* is a *finite-state transition diagram* that can be used to model the *recognition* of a *token type* specified by a *regular expression*
- Finite automata are *recognizers*; they simply say "yes" or "no" about each possible input string.
- A finite automaton can be a
  - *Non-deterministic* finite automaton (NFA)
  - *Deterministic* finite automaton (DFA)
- DFA & NFA are capable of recognizing the same languages

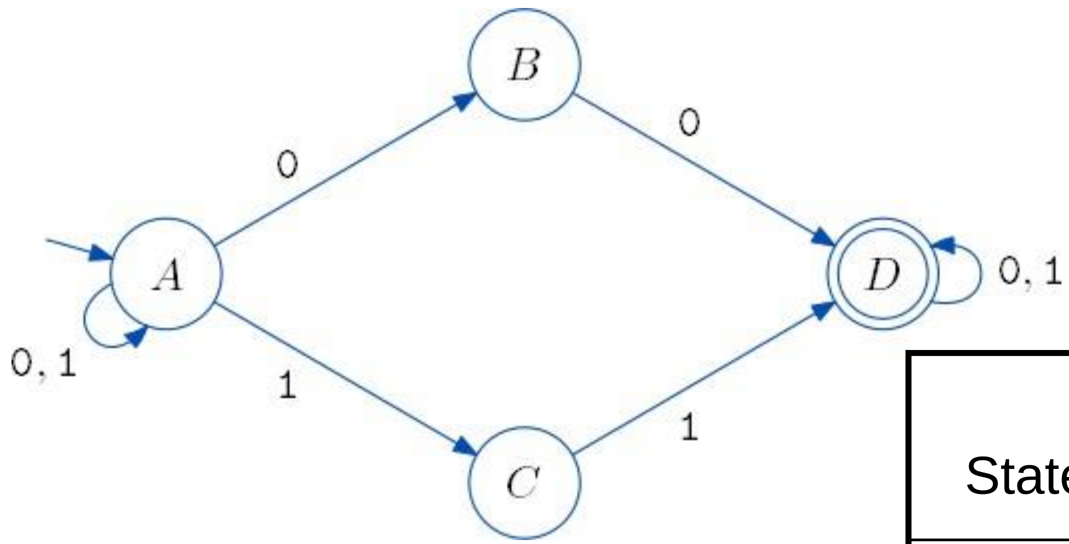
Nondeterministic Finite Automata

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 ϵ, 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 $\Sigma \cup \{\epsilon\}$ a set of *next states*.
4. A state 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*).

NFA:Transition Graph and Transition tables

- An NFA can be diagrammatically represented by a labeled directed graph called a transition graph



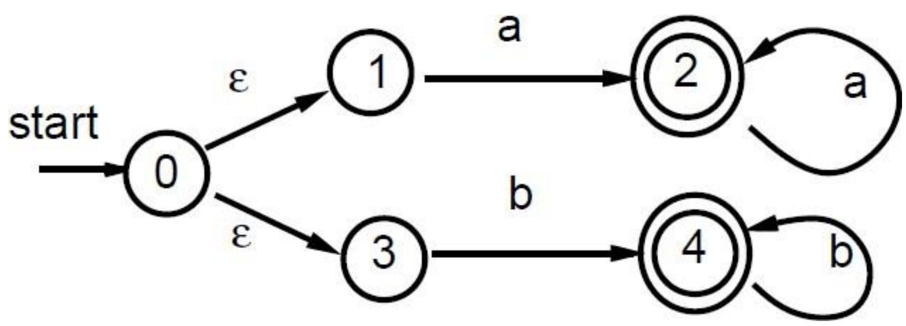
The set of states = {A,B, C, D}
Input symbol = {0, 1}
Start state is A,
accepting state is D

State	Input Symbol	
	0	1
A	{A,B}	{A,C}
B	{D}	--
C	--	{D}
D	{D}	{D}

- Transition graph shown in preceding slide can be represented by transition table on the right side

Cont…

- **Acceptance of Input Strings by Automata**
  - An NFA *accepts* an input string *str* ***iff*** there is *some* path in the finite-state transition diagram from the *start state* to some *final state* such that the *edge labels* along this path spell out *str*
  - The *language* recognized by an NFA is *the set of strings* it accepts
  - Example

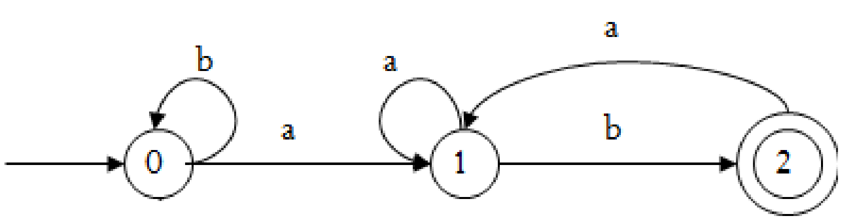


The language recognized by this NFA is $aa^*|b\,b^*$

Deterministic Finite Automata

- A deterministic finite automaton (DFA) is a special case of an NFA where:
  - There are no moves on input ε,and
  - For each state *q* and input symbol *a*, there is exactly one edge out of *a* labeled *a*.

- **Example**



The language recognized by

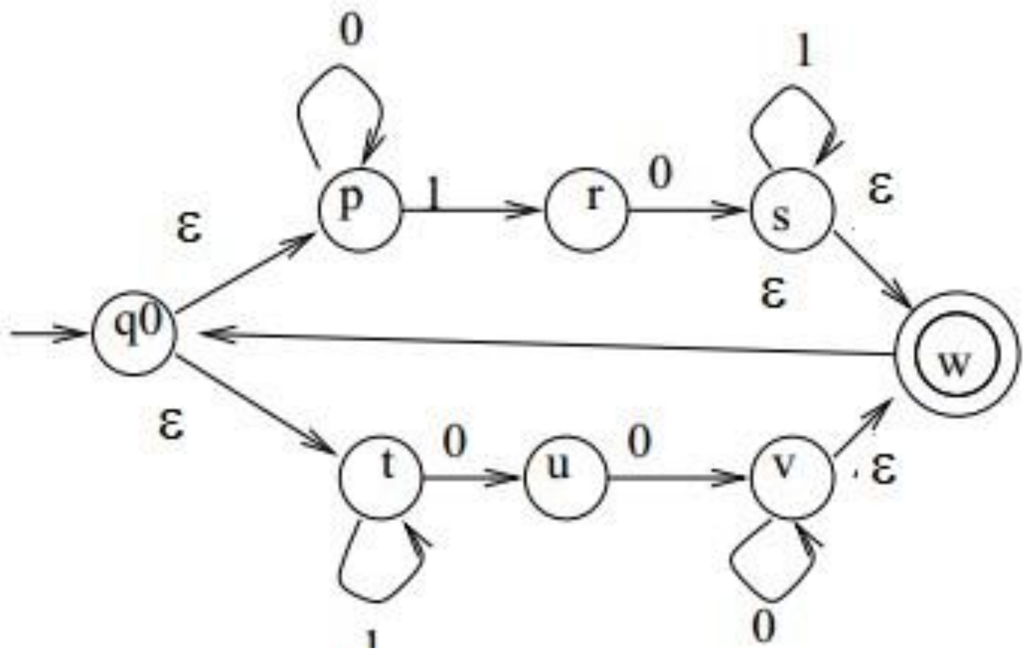
this DFA is also
(b*aa*(aa*b)+)

Computing the ε-closure (Q)

- ε-*closure*(*q*): set of states reachable from some state *q* in *Q* on ε-transitions alone.
- *move*(*q*, *c*): set of states to which there is a transition on input symbol *c* from some state *q* in *Q*

Computing the ε-closure (Q) ...Example

- Given an NFA below find ε-closure (s)



$$\varepsilon\text{-closure (S)} = \{s, w, q_0, p, t\}$$

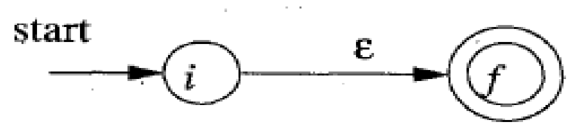
i.e. All states that can be reached through ε-transitions only

Regular Expression to NFA

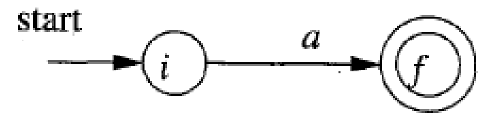
- For each kind of RE, there is a corresponding NFA
  - i.e. any regular expression can be converted to a NFA that defines the same language.
- The algorithm is syntax-directed, in the sense that it works recursively up the parse tree for the regular expression.
- For each sub-expression the algorithm constructs an NFA with a single accepting state.
- Algorithm: The ***McNaughton-Yamada-Thompson*** algorithm to convert a regular expression to a NFA.
  - **INPUT:** A regular expression r over alphabet Σ.
  - **OUTPUT:** An NFA **N** accepting L(r).

Cont…

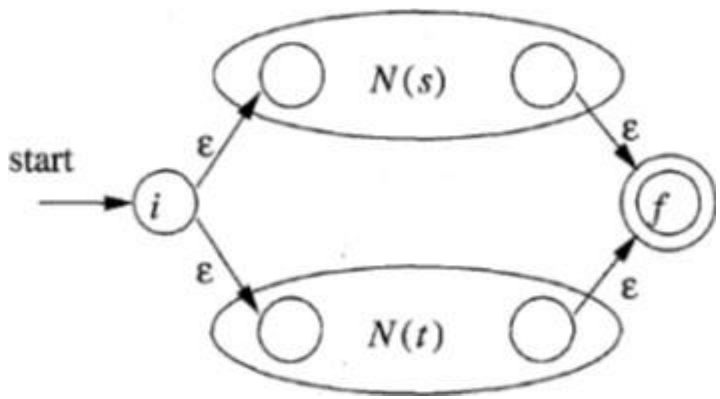
- **Method**: Begin by parsing r into its constituent sub-expressions.The rules for constructing an NFA consist of basis rules for handling sub-expressions with no operators, and inductive rules for constructing larger NFA's from the NFA's for the immediate sub-expressions of a given expression.
  - For expression e construct the NFA

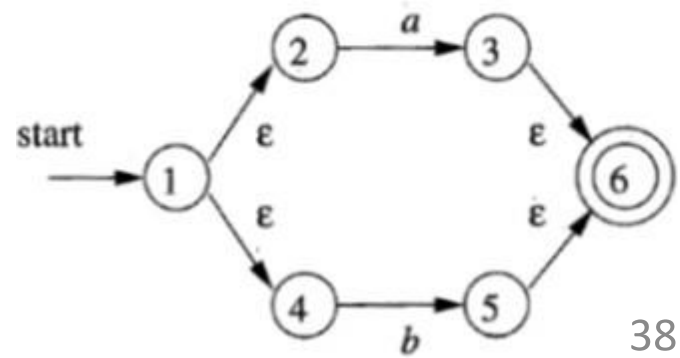


  - For any sub-expression a in C, construct the NFA



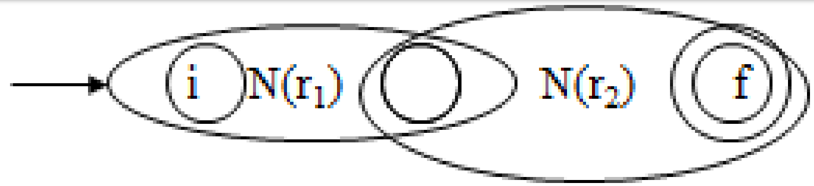
  - NFA for the union of two regular expressions
    - Ex: *a*| *b*



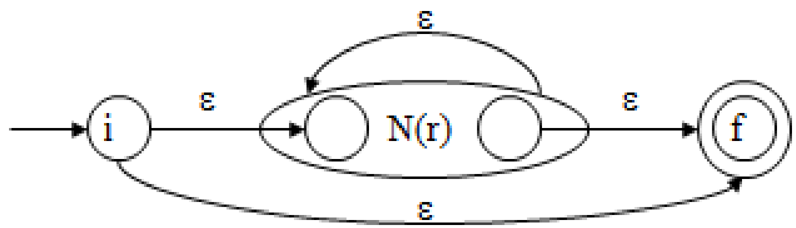


Cont…

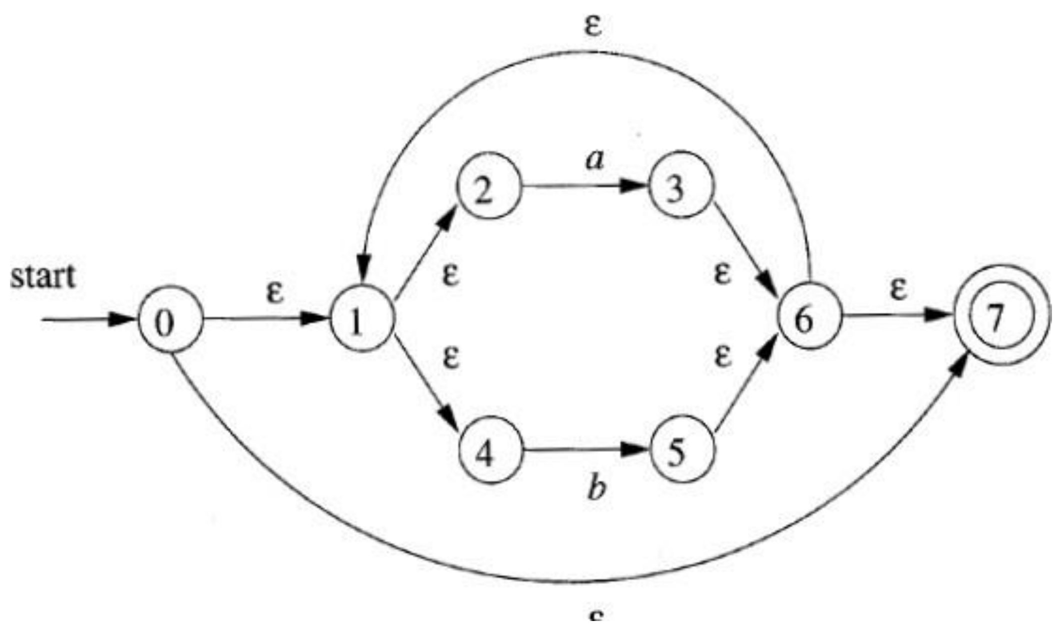
- NFA for r_1r_2



- NFA for r*



- Example: **(a|b)***

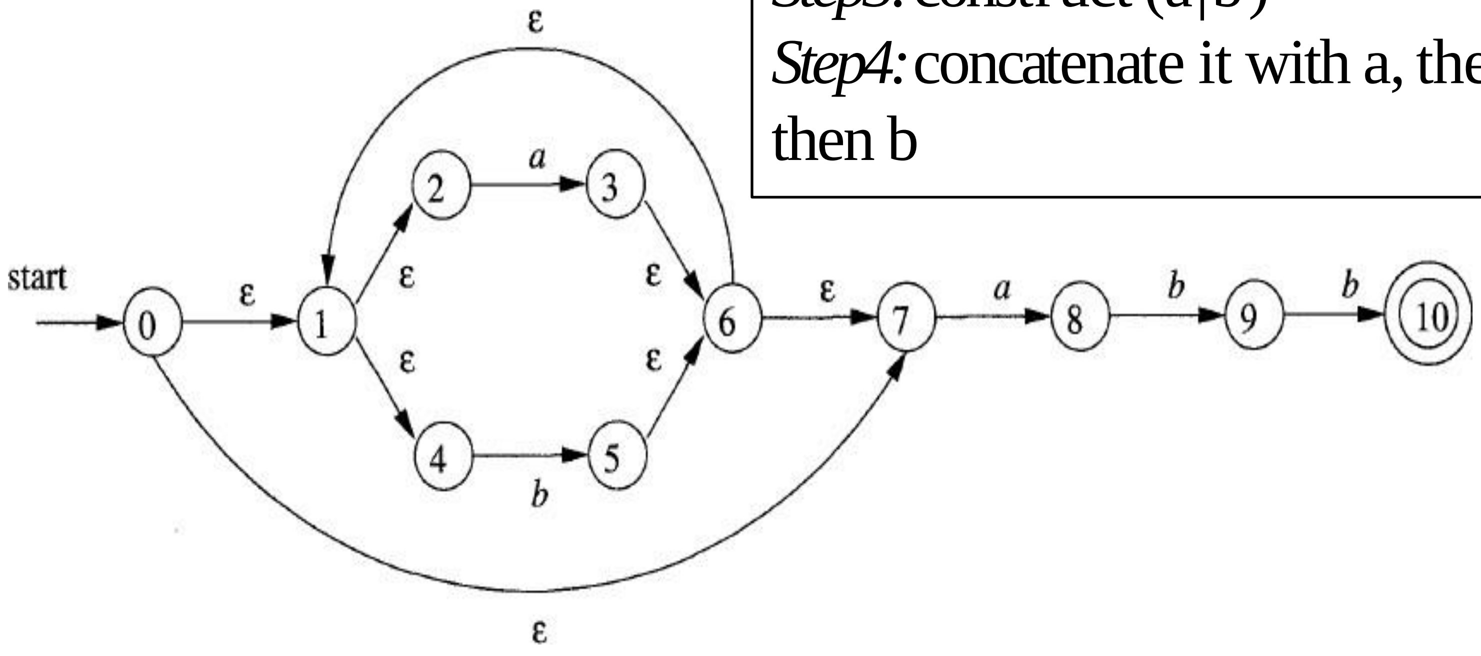


Cont…

- **<u>Example:</u>** Constructing NFA for regular expression

  r= (a|b)*abb

Step 1: construct a, b
Step 2: constructing a | b
Step3: construct (a|b)*
Step4: concatenate it with a, then, b, then b

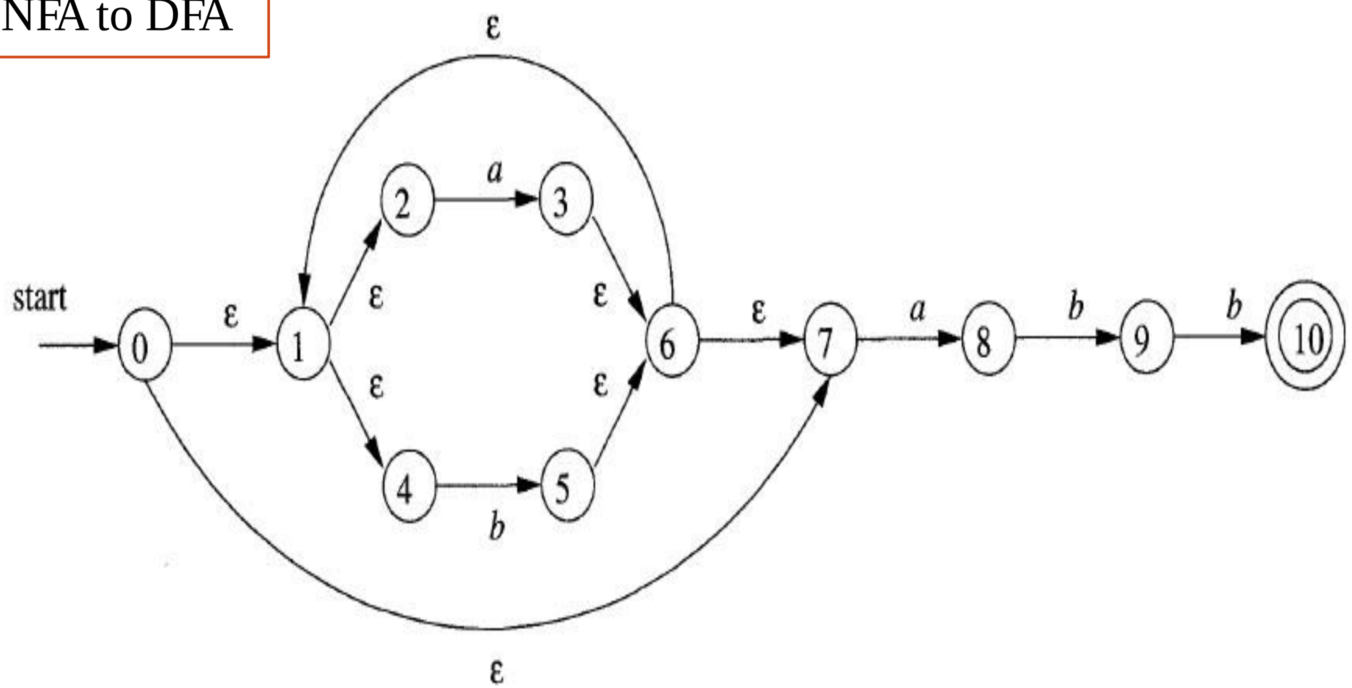


Conversion of NFA to DFA

- Why?
  - DFA is difficult to construct directly from RE's
  - NFA is difficult to represent in a computer program and inefficient to compute
  - So, RE ➔ NFA ➔ DFA
- Conversion algorithm: **subset construction**
  - The idea is that each DFA state corresponds to a set of NFA states.
  - After reading input $a_1a_2\ldots a_n$, the DFA is in a state that represents the subset Q of the states of the NFA that are reachable from the start state.
  - ***INPUT:*** An NFA N
  - ***OUTPUT:*** A DFA D accepting the same language as N.

Subset Construction Algorithm…

Example NFA to DFA



The start state A of the equivalent DFA is ε-closure(0),

- A= {0,1,2,4,7},

- Since these are exactly the states reachable from state 0 via a path all of whose edges have label ε.
    - ***Note that a path can have zero edges, so state 0 is reachable from itself by an ε -labeled path.***

Cont…

- The input alphabet is {a, b).Thus, our first step is to mark Aand compute

  Dtran[A, a] = ε -closure(move(A, a)) and

  Dtran[A, b] = ε - closure(move(A, b)) .
- Among the states 0, 1, 2, 4, and 7, only 2 and 7 have transitions on a, to 3 and 8, respectively.
  - Thus, move(A, a) = {3,8).Also, ε -closure({3,8})= {1,2,3,4,6,7,8), so we call this set B,

$$Dtran[A, a] = \epsilon\text{-}closure(move(A, a)) = \epsilon\text{-}closure(\{3, 8\}) = \{1, 2, 3, 4, 6, 7, 8\}$$

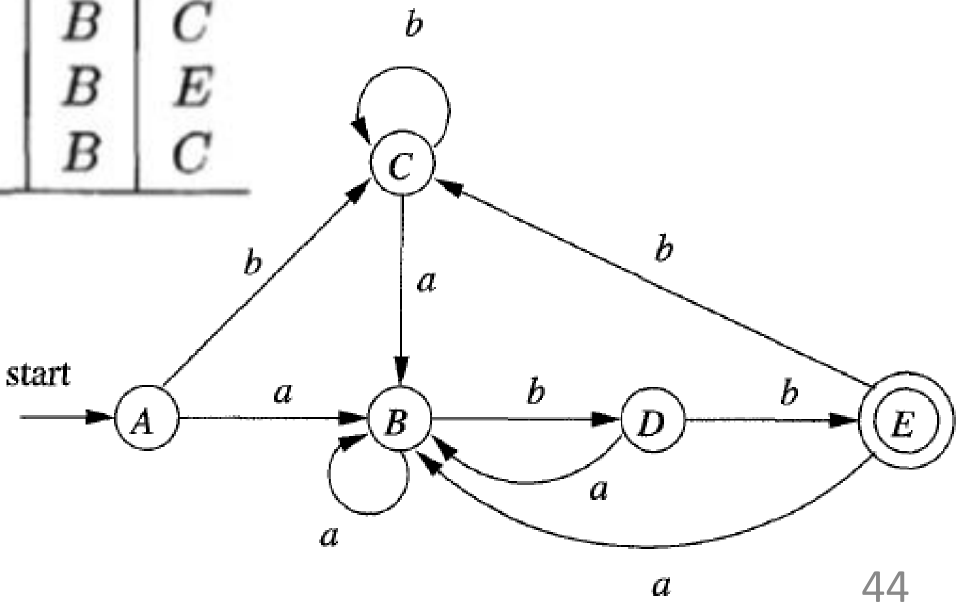
let Dtran[A, a] = B

- compute Dtran[A, b].Among the states in A, only 4 has a transition on *b*, and it goes to 5

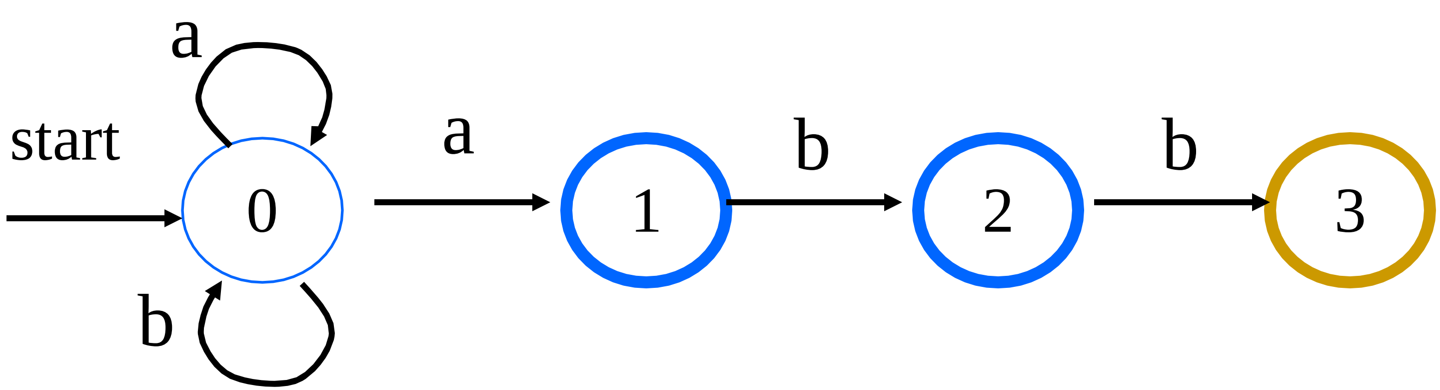
Subset Construction Algorithm…

- C = $Dtran[A, b] = \epsilon\text{-}closure(\{5\}) = \{1, 2, 4, 6, 7\}$
- If we continue this process with the unmarked sets Band C, we eventually reach a point where all the states of the DFA are marked

NFA STATE	DFA STATE	a	b
$\{0, 1, 2, 4, 7\}$	A	B	C
$\{1, 2, 3, 4, 6, 7, 8\}$	B	B	D
$\{1, 2, 4, 5, 6, 7\}$	C	B	C
$\{1, 2, 4, 5, 6, 7, 9\}$	D	B	E
$\{1, 2, 3, 5, 6, 7, 10\}$	E	B	C



NFA to DFA conversion: Example 2



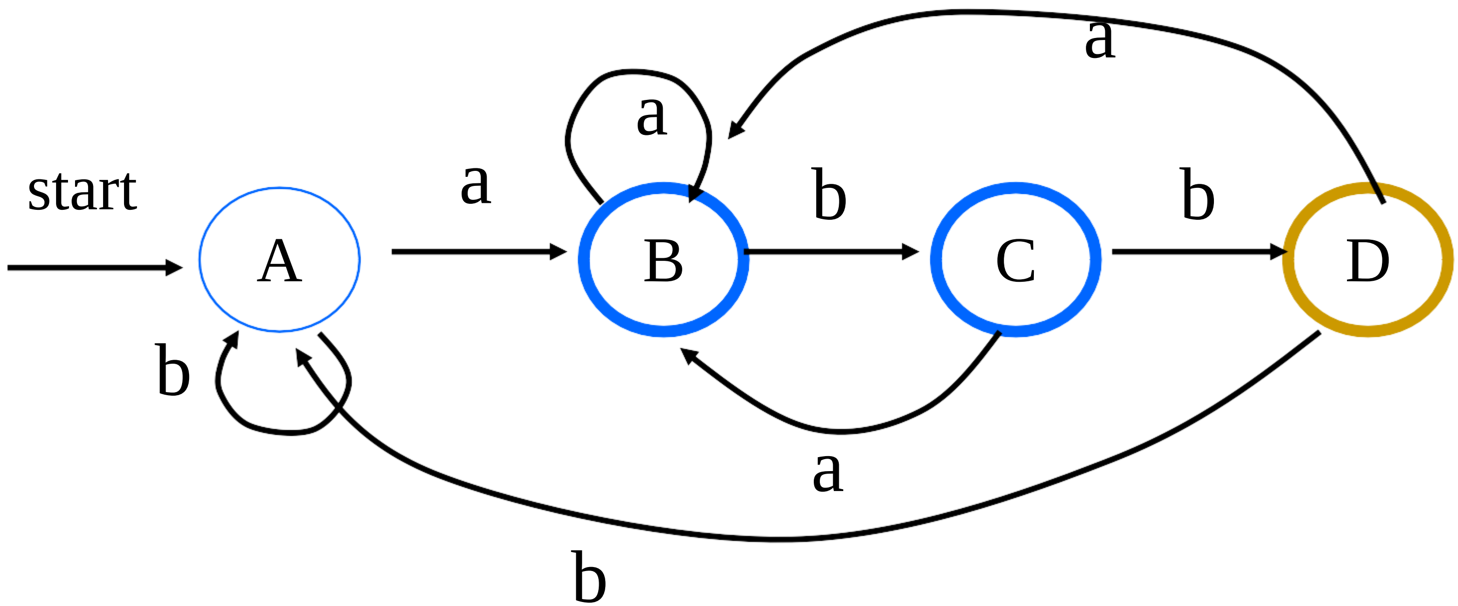
ε-closure(0) = {0}
move(0,a) = {0,1}
move(0,b) = {0}
move({0,1}, a) = {0,1}
move({0,1}, b) = {0,2}
move({0,2}, a) = {0,1}
move({0,2}, b) = {0,3}

New states

A = {0}
B = {0,1}
C = {0,2}
D = {0,3}

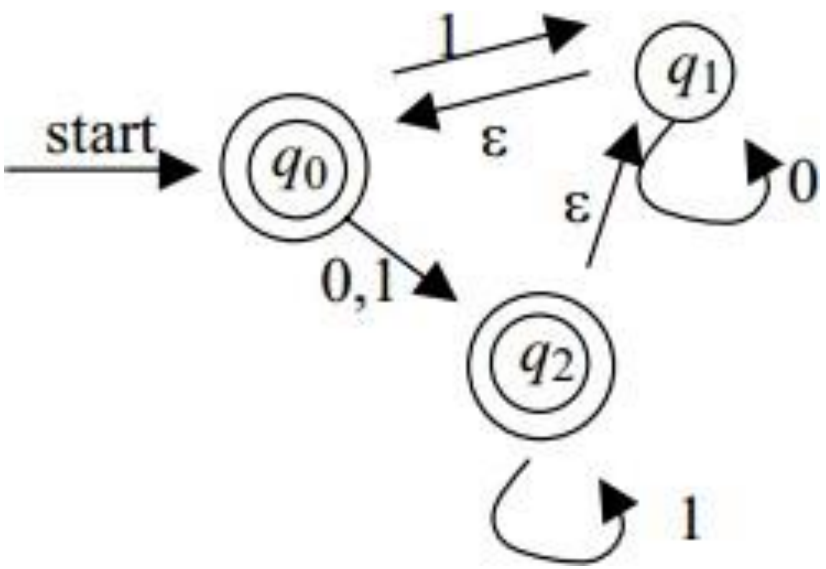
	a	b
A	B	A
B	B	C
C	B	D
D	B	A

NFA to DFA conversion: Example 2…

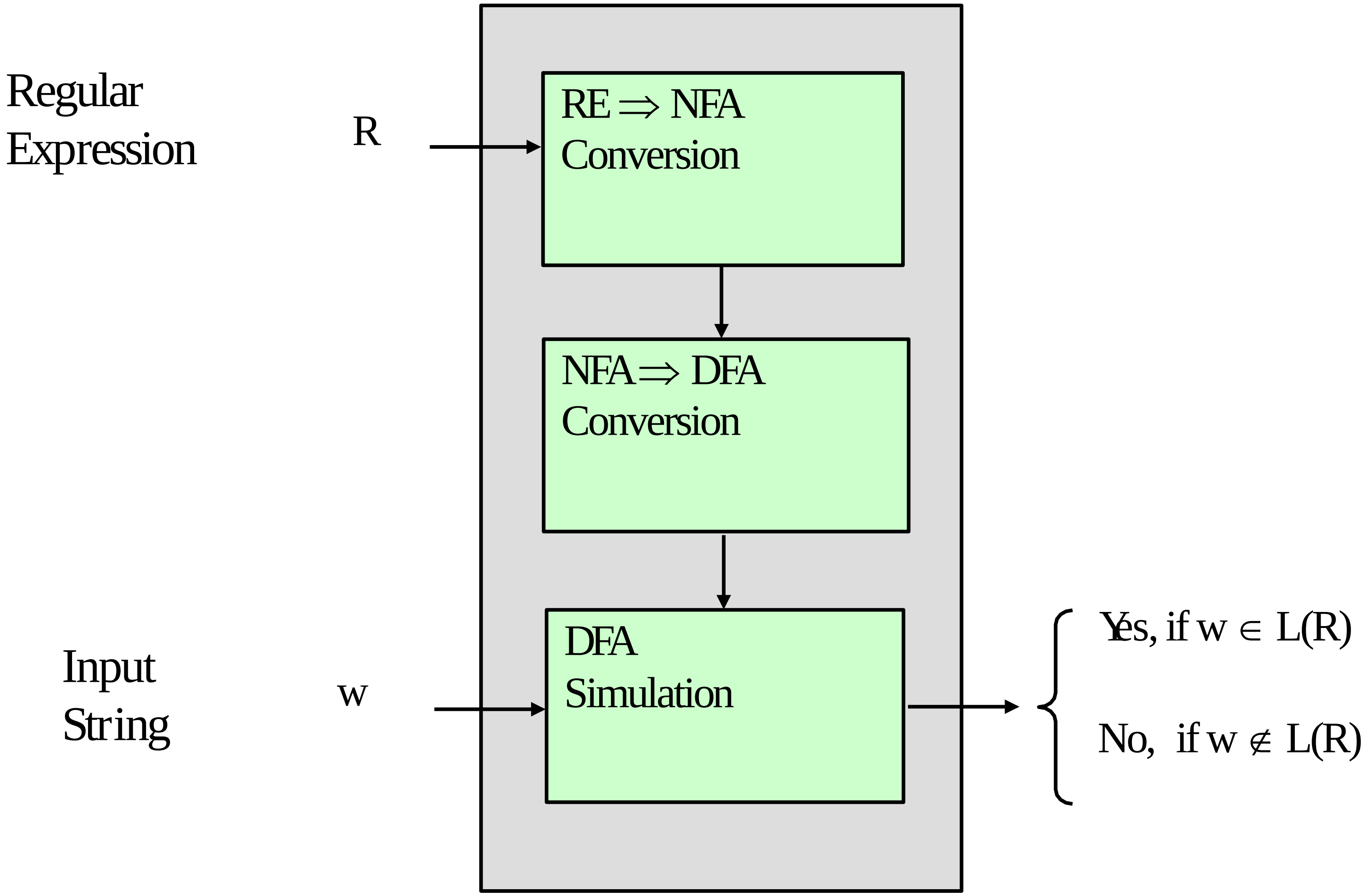


NFA to DFA conversion: Exercise

Convert the -NFA shown below into a DFA, using subset construction



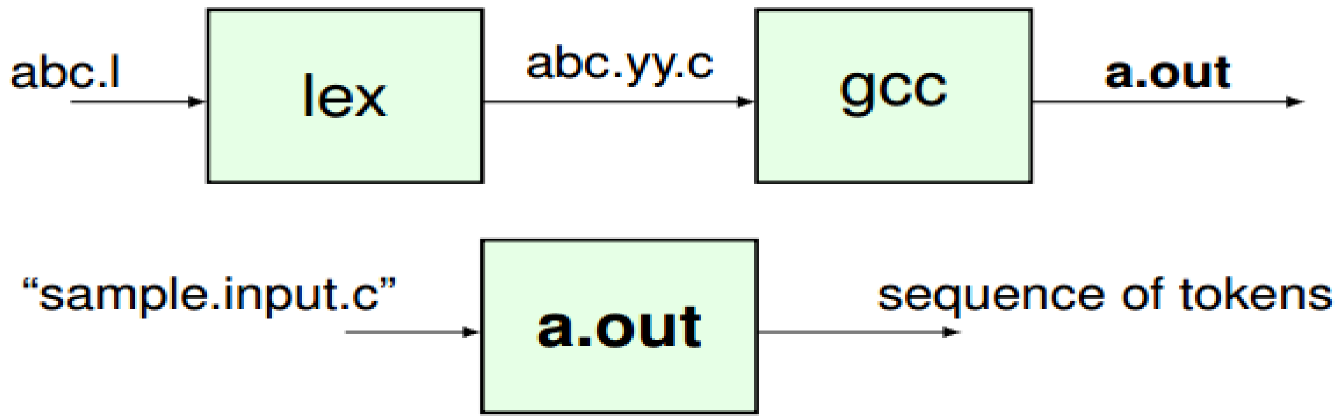
Putting the Pieces Together



Implementing the Scanner

- How to actually get the lexical analyzer?
  - **Solution 1:** to implement using a tool — Lex (for C), Flex (for C++), Jlex (for java)
    - Programmer specifies the interesting tokens using REs
    - The tool generates the source code from the given Res
  - **Solution 2:** to write the code starting from scratch
    - This is also the code that the tool generates
    - The code is table-driven and is based on finite state automaton

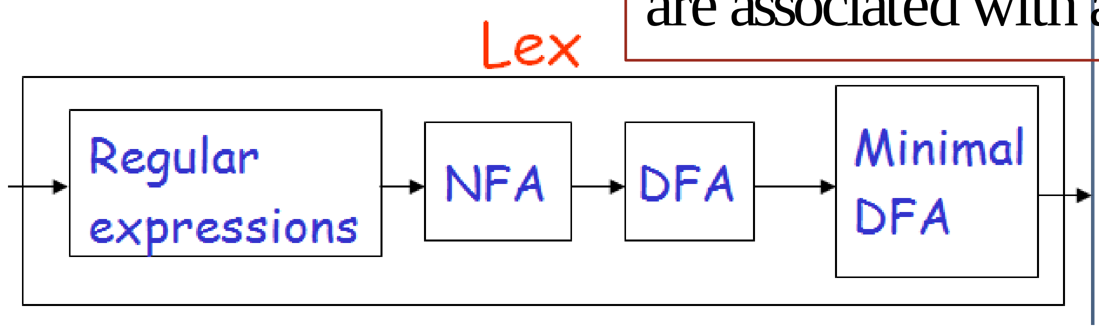
Lex: a Tool for Lexical Analysis



- Big difference from your previous coding experience
  - Writing REs instead of the code itself
  - Writing actions associated with each RE

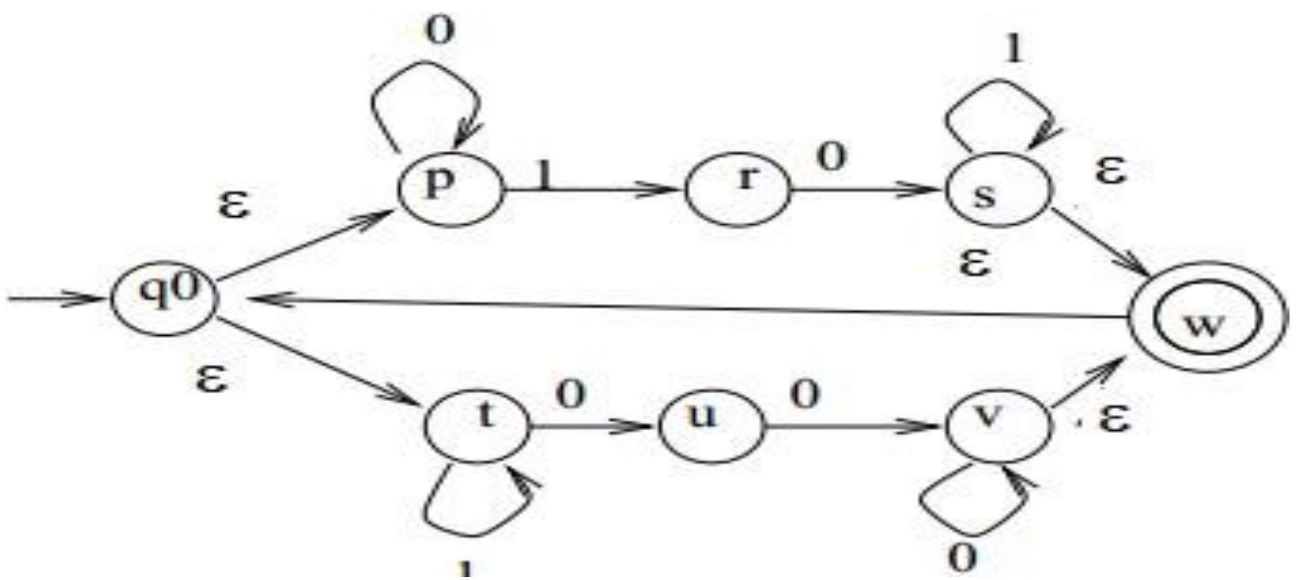
Internal Structure of Lex

The final states of the DFA are associated with actions



Assignment 1 (10%)

1. Implement DFA and NFA. The program should accept states, alphabet, transition function, start state and set of final states. Then the program should check if a string provided by a user is accepted or rejected. You can use either Java or C++ for the implementation
2. Convert the following NFA to DFA



Reading Assignment

Constructing a DFA directly from a regular expression