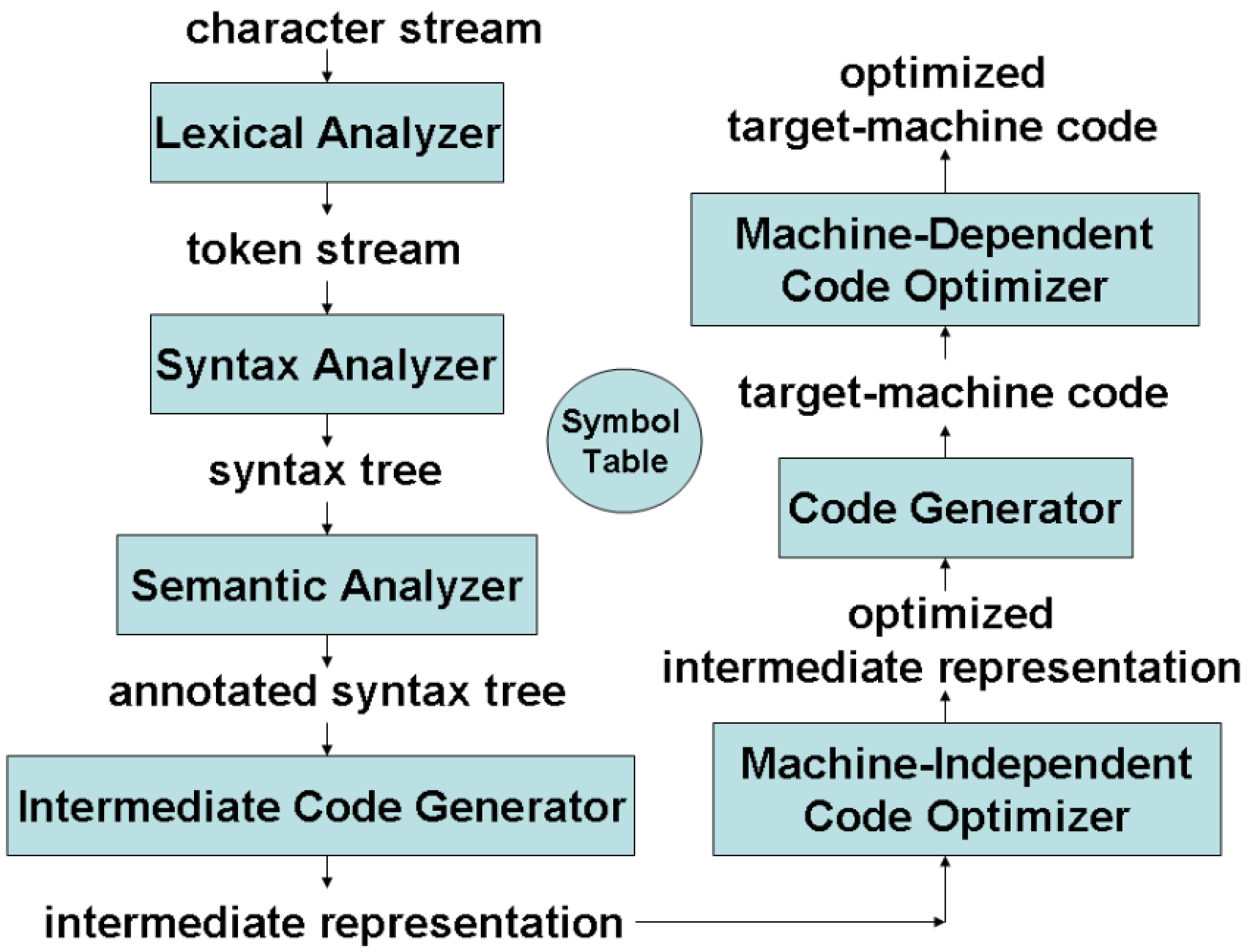
Lexical Analysis- Part 2

Recap (1)- Compiler Overview



Recap (2) – Lexical Analysis

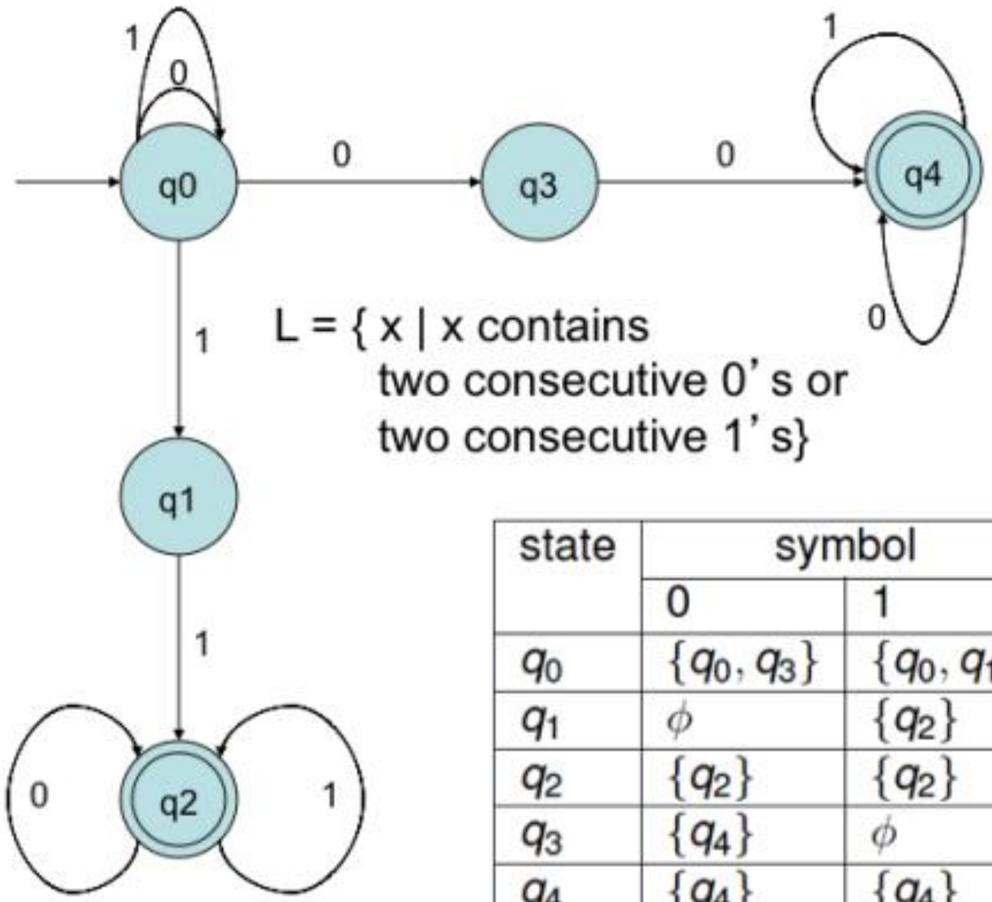
- What is lexical analysis?
- Why should LA be separated from syntax analysis?
- Tokens, patterns, and lexemes
- Difficulties in lexical analysis

- **Specification of tokens** - regular expressions and regular definitions
- **Recognition of tokens** - finite automata and transition diagrams
- **LEX** - A Lexical Analyzer Generator

Non-deterministic Finite Automata

- Finite automata that allow 0, 1 or more transitions from a state on a given symbol
- An NFA is a 5-tuple (similar to DFA), but the transition function δ is different
  - $\delta(q, a)$ = the set of all states p, such that there is a transition labelled a from q to p
  - $\delta : Q \times \Sigma \rightarrow 2^Q$
- A string is accepted by an NFA if **there exists** a sequence of transitions corresponding to the string, that leads from the start state to some final state
- Every NFA can be converted to an equivalent DFA, that accepts the same language as the NFA

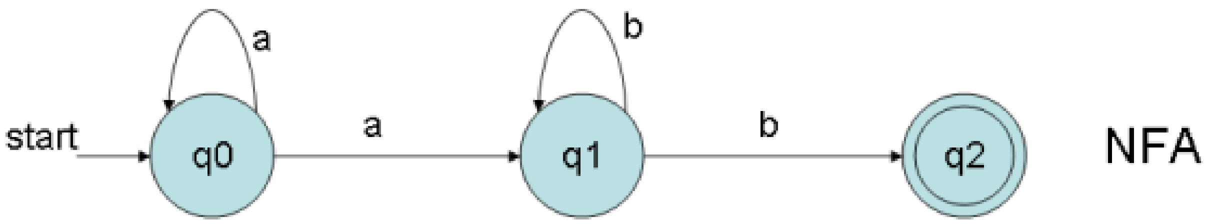
Non-deterministic Finite Automata - Example

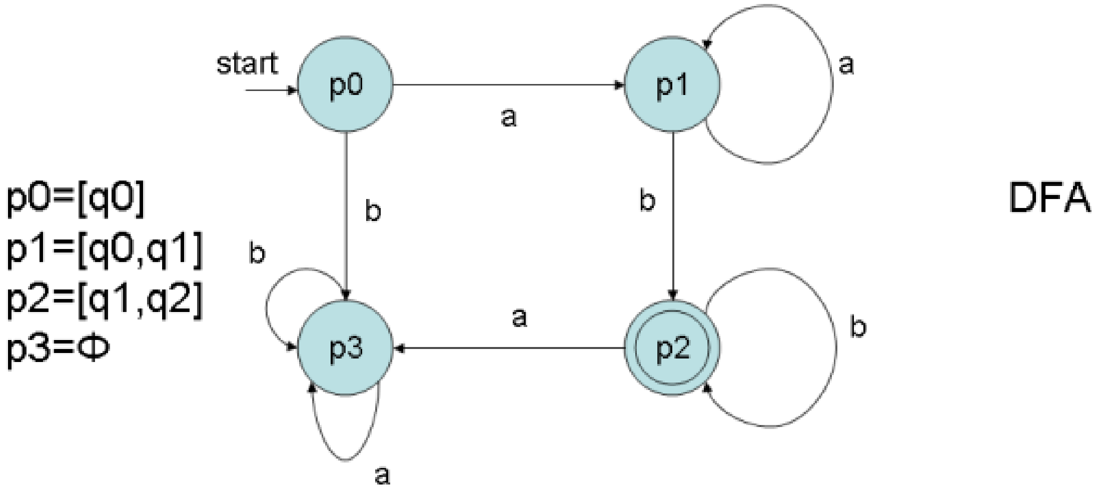


L = { x | x contains
two consecutive 0's or
two consecutive 1's}

state	symbol	
	0	1
q_0	$\{q_0, q_3\}$	$\{q_0, q_1\}$
q_1	ϕ	$\{q_2\}$
q_2	$\{q_2\}$	$\{q_2\}$
q_3	$\{q_4\}$	ϕ
q_4	$\{q_4\}$	$\{q_4\}$

Example: An NFA and its equivalent DFA





Example of NFA to DFA conversion

- The start state of the DFA would correspond to the set $\{q_0\}$ and will be represented by $[q_0]$
- Starting from $\delta([q_0], a)$, the **new states of the DFA are constructed on demand**
- Each subset of NFA states is a possible DFA state
- All the states of the DFA containing some final state as a member would be final states of the DFA
- For the NFA presented before (whose equivalent DFA was also presented)
  - $\delta([q_0], a) = [q_0, q_1]$, $\delta([q_0], b) = \phi$
  - $\delta([q_0, q_1], a) = [q_0, q_1]$, $\delta([q_0, q_1], b) = [q_1, q_2]$
  - $\delta(\phi, a) = \phi$, $\delta(\phi, b) = \phi$
  - $\delta([q_1, q_2], a) = \phi$, $\delta([q_1, q_2], b) = [q_1, q_2]$
  - $[q_1, q_2]$ is the final state
- In the worst case, the converted DFA may have 2^n states, where n is the no. of states of the NFA

Regular Expressions

- Let **Σ** be an *alphabet*. The **REs** over **Σ** and the languages they denote (or generate) are defined
  1. ϕ is an RE. L(ϕ) = ϕ
  2. ε is an RE. L(ε) = {ε}
  3. For each a ∈ Σ, a is an RE. L(a) = {a}

  4. If r and s are REs denoting the languages R and S, respectively
     1. (rs) is an RE (*denotes concatenation*)
     2. (r + s) is an RE, (denotes either *r* or *s*)
     3. (r∗) is an RE (denotes zero or more occurrences of *r*)

Regular Expressions: Examples
(Give corresponding RE)

1. L = {if, then, else, while, do, begin, end}

2. L = set of all strings of 0's and 1's, beginning with 1 and not having two consecutive 0's

3. L = $\{w \mid w \in \{a, b\}^* \wedge w \text{ ends with } a\}$

4. L = set of all strings over {a,b,c} that do not have the substring ac

Regular Expressions: Examples

1. L = {if, then, else, while, do, begin, end}

   r = if + then + else + while + do + begin + end

2. L = set of all strings of 0's and 1's, beginning with 1 and not having two consecutive 0's

   $r = (1+10)^*$

3. $L = \{w \mid w \in \{a, b\}^* \land w \text{ ends with } a\}$

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

4. L = set of all strings over **{a,b,c}** that do not have the substring **ac**

   $r = c^*(a + bc^*)^*$

Regular Languages

- The language accepted by an FSA is the set of all strings accepted by it (regular language).

- It can be shown that for every regular expression, an FSA can be constructed and vice-versa

Regular Definitions

- A regular definition is a sequence of "equations" of the form

$d_1 = r_1;\ d_2 = r_2;\ \ldots\ ;\ d_n = r_n,$

where each $\mathbf{d_i}$ is a distinct name,

and each $\mathbf{r_i}$ is a regular expression over the symbols $\mathbf{\Sigma \cup \{d_1, d_2, \ldots, d_{i-1}\}}$

Example (Identifiers and Integers)

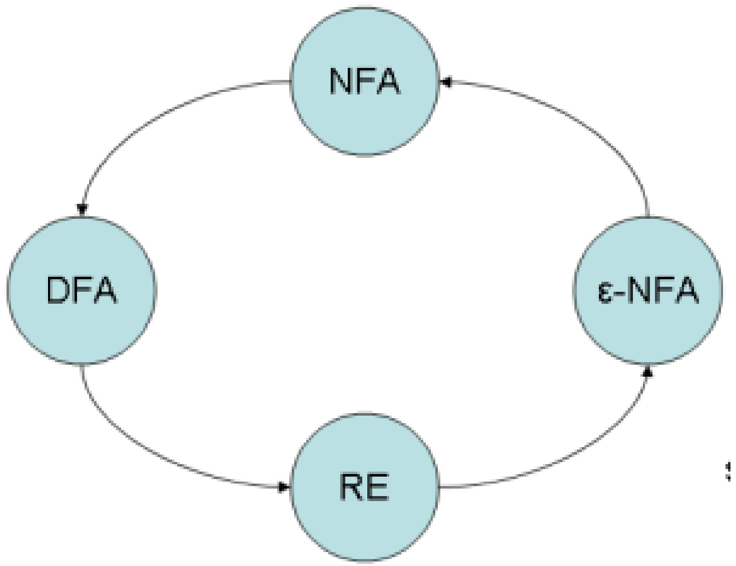
letter = a + b + c + d + e;

digit = 0 + 1 + 2 + 3 + 4;

identifier = letter(letter + digit)*;

number = digit digit*

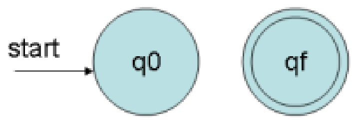
From RE to Automaton



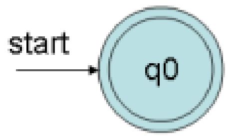
RE to NFA (Thompson's construction)

- NFA pattern for each symbol and operator

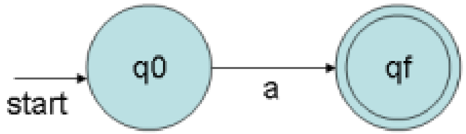
r=Φ



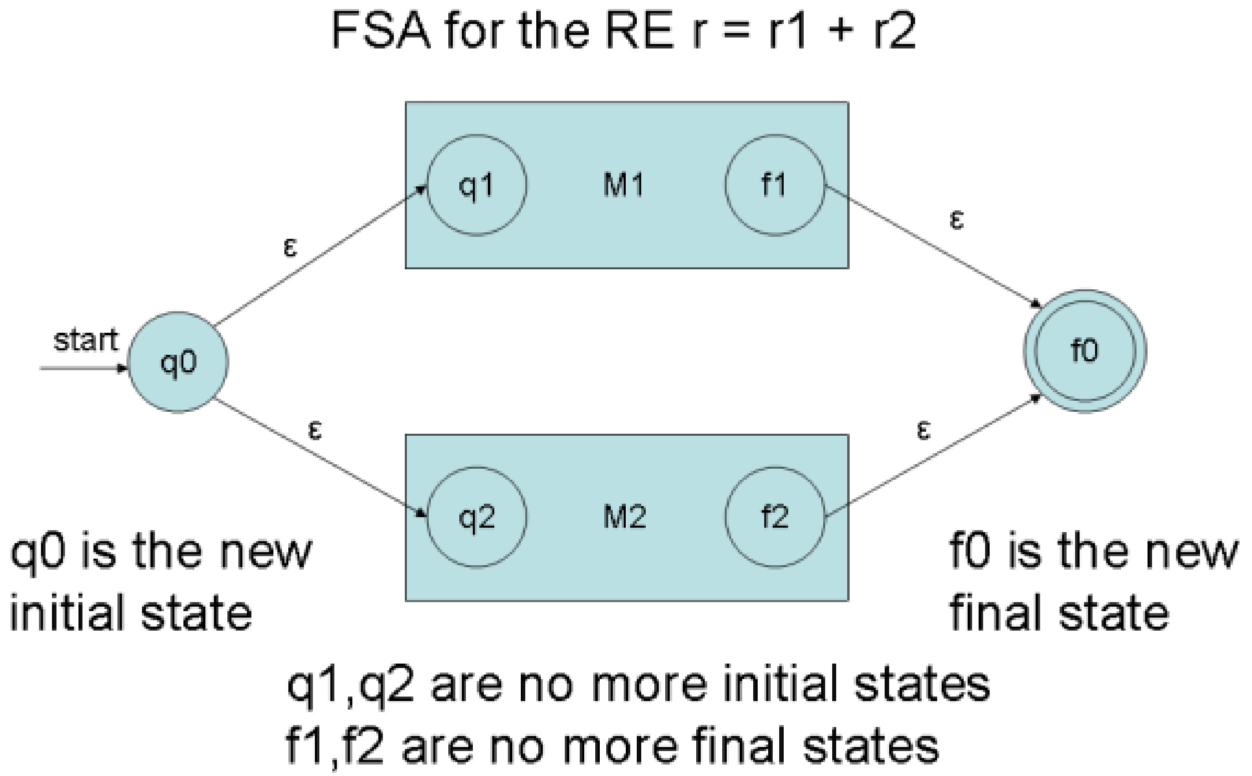
r=ε



r=a

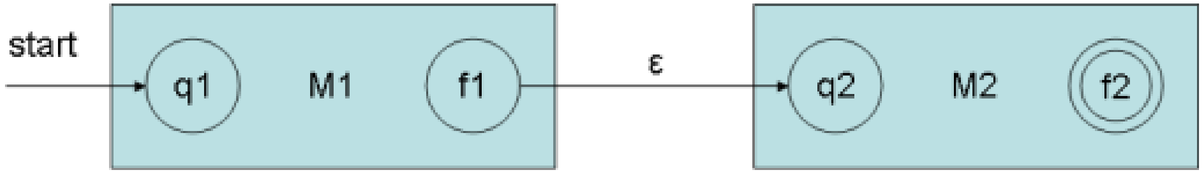


NFA pattern for $\mathbf{R = r_1 + r_2}$



NFA pattern for **R = r_1 r_2**

FSA for RE r = r1 r2

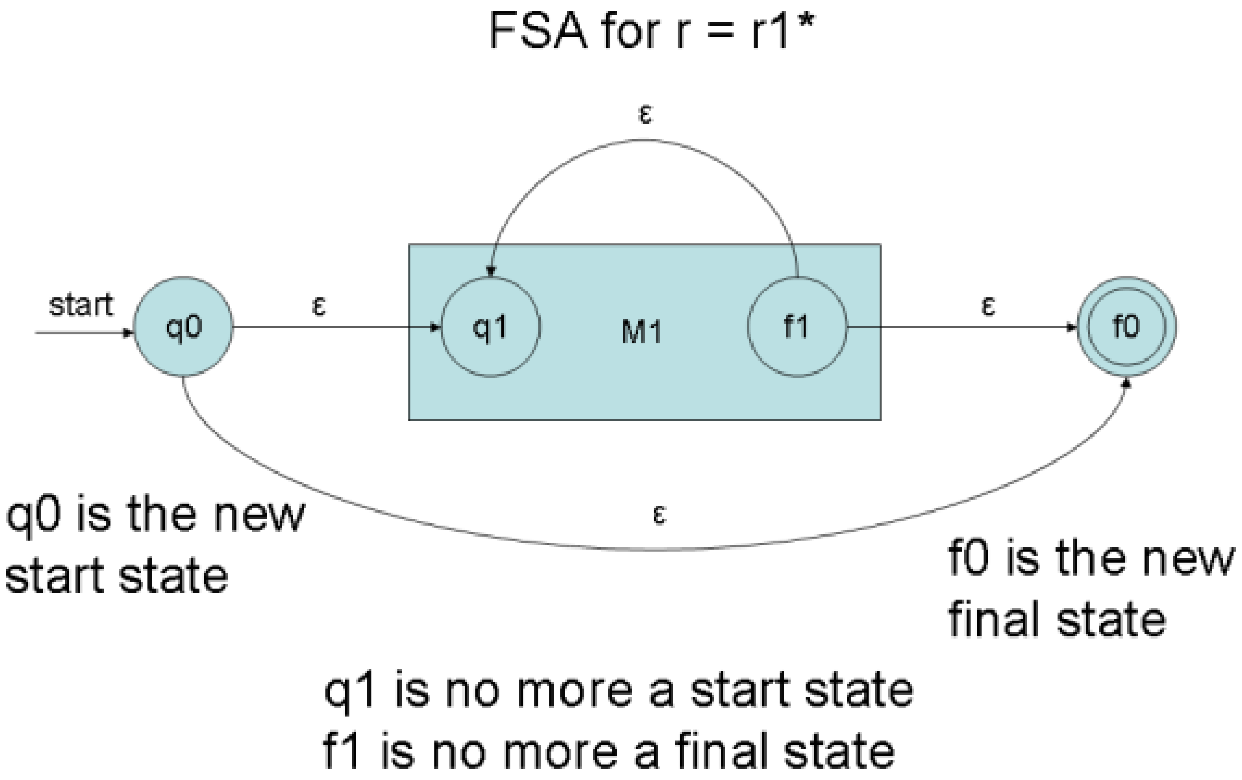


q1 is the new
start state

f2 is the new
final state

f1 is no more a final state
q2 is no more a start state

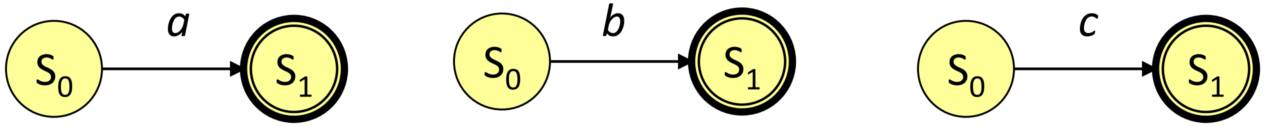
Pattern for **R = r_1^***

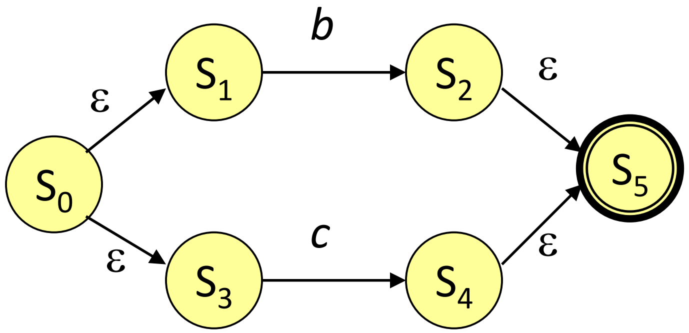


Example: Construct the NFA of *a (b|c)**

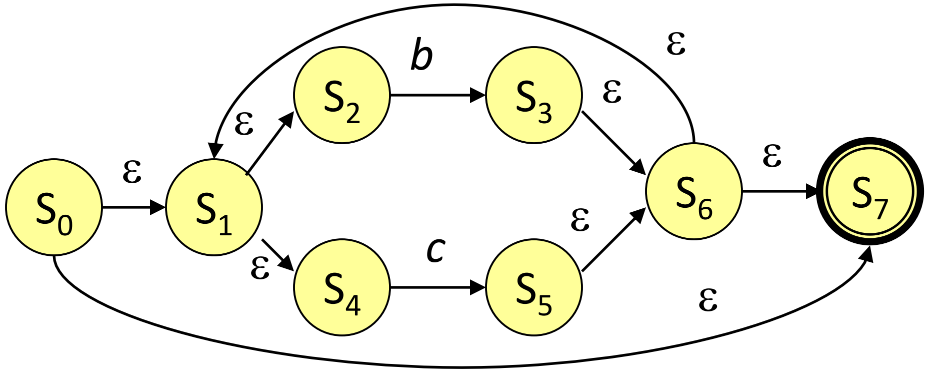
Example: Construct the NFA of ***a (b|c)****

First: NFAs for *a, b, c*



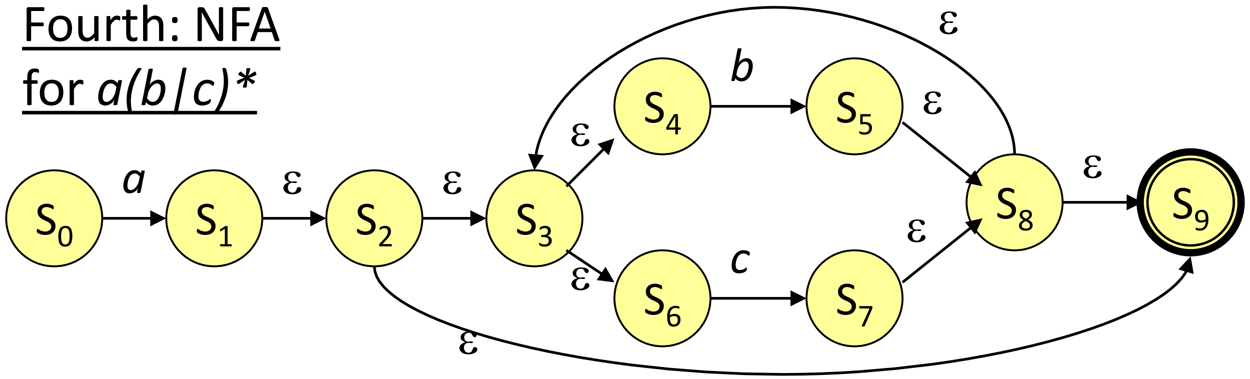


Second: NFA for *b|c*



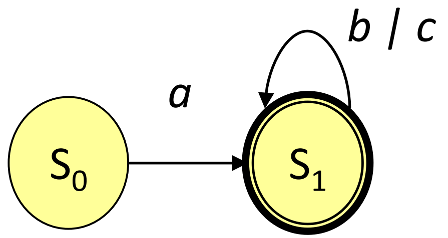
Third: NFA for *(b|c)**

Fourth: NFA for *a(b|c)**



NFA pattern for each symbol and/or operator: join them in precedence order

NFA of *a (b|c)**



Of course, a human would design a simpler one…
But, we can **automate** production of the complex one...

Transition Diagrams

- Conversion of patterns into stylized flow-charts called "transition diagrams" (**intermediate step in the construction of a lexical analyzer**).

- Let us see how to perform **conversion of regular-expression patterns to transition diagrams** by hand.

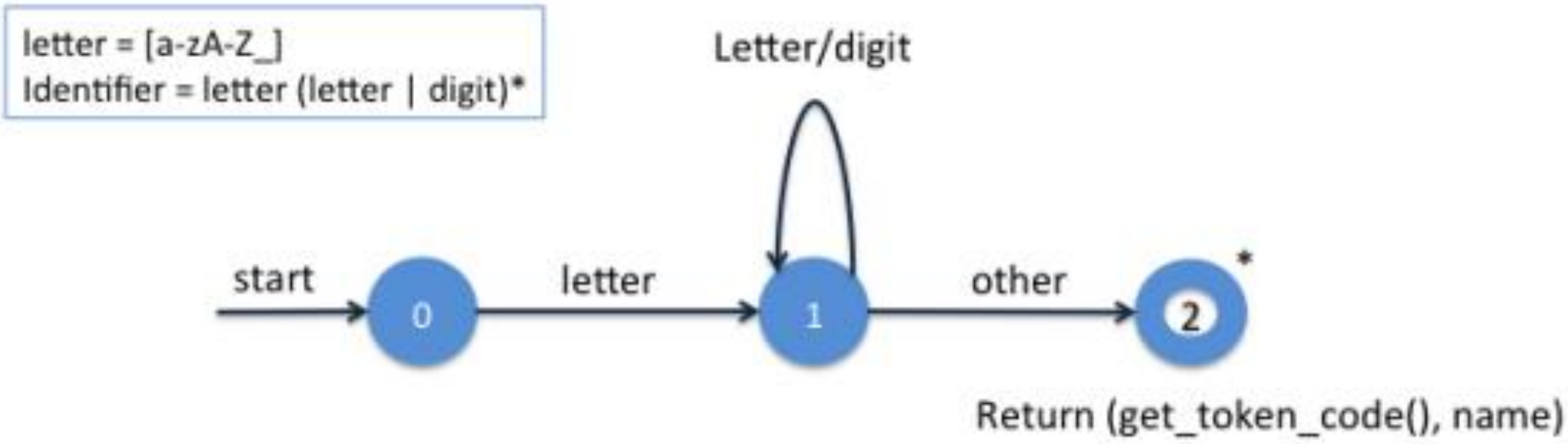
Transition Diagrams

- Transition diagrams are generalized DFAs with the following differences
  - **Edge Labels**: Edges may be labelled by a **symbol**, *a* ***set of symbols***, *or a* ***regular definition***

  - **Retracting States**: Some accepting states may be indicated as **retracting states**, indicating that the **lexeme does not include the symbol that brought us to the accepting state**

  - **Actions**: Each **accepting state** has an ***action*** attached to it, which is executed when that state is reached. Typically, such an action returns a token and its attribute value

Transition Diagram for identifier and reserved words



- “*” **indicates retraction state**
- **get_token_code()**:
  - searches a table if the name is a reserved word and returns its integer code.
  - If not a reserved word, it returns integer code of IDENTIFIER token, with **name** containing the string of characters forming the token.
  - name not relevant for reserved words

Example: A Grammar for Branching Statements

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

Grammar fragment describing a simple form of branching statements and conditional expressions

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

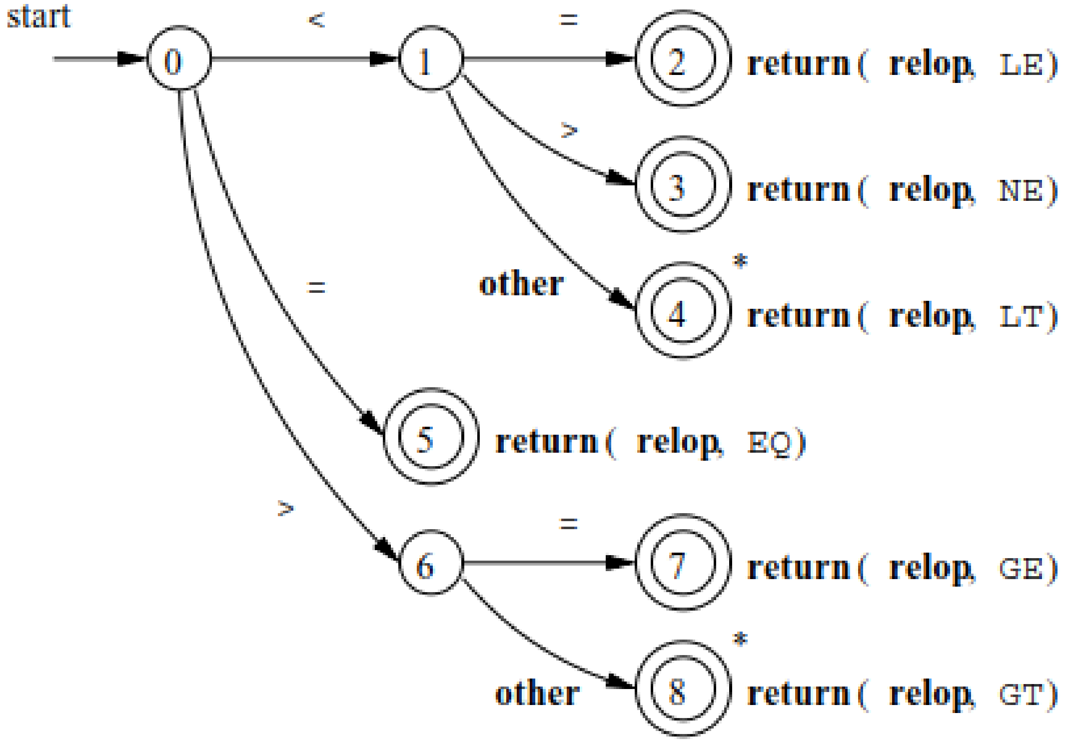
Patterns for tokens

Transition diagram for relop

relop **-> < | > | <= | >= | = | <>**

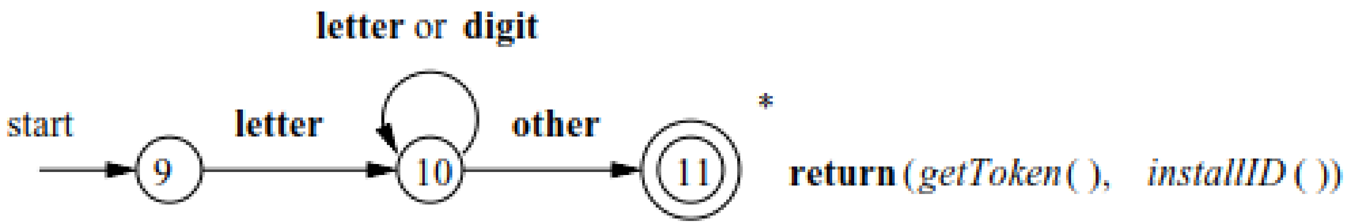
Transition diagram for relop

***relop* -> < | > | <= | >= | = | <>**



Transition diagram for identifiers (and keywords)

id* -> letter (letter | digit)



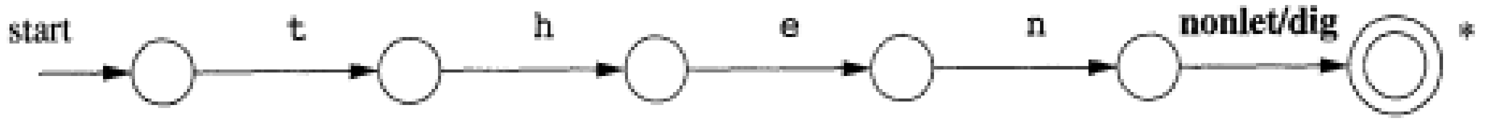
Two approaches to handle reserved keywords that look like identifiers

Approach 1:

- Reserved words in a table initially
- ***getToken()***: examines symbol table, returns what token the lexeme found represents
- ***installID()***: if **ID**, places it in the symbol table, returns pointer to the symbol table entry

Transition diagram for identifiers (and keywords)

Two approaches to handle reserved keywords that look like identifiers



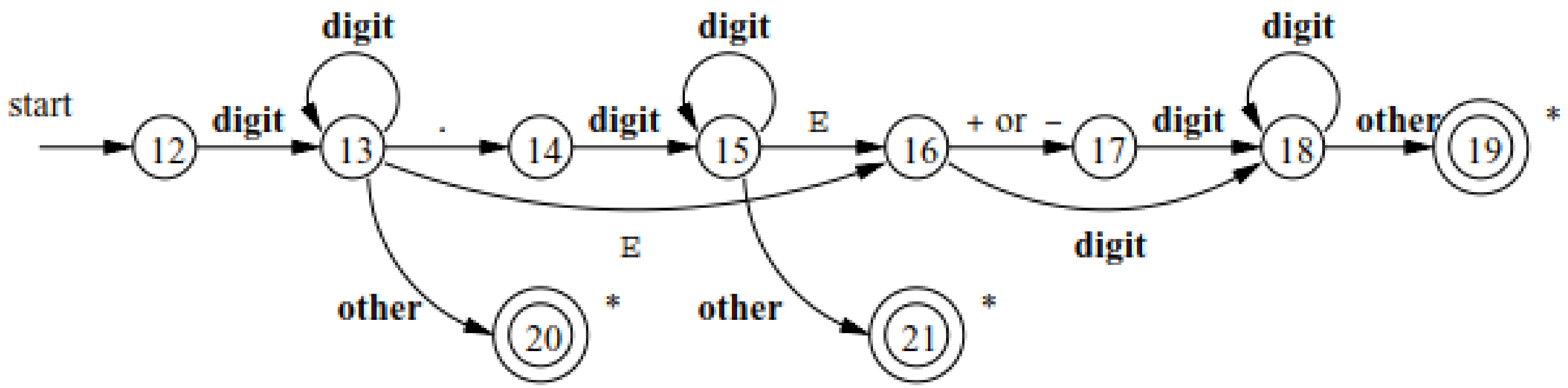
Approach 2:
- Create a separate transition diagram for each keyword
- States representing situation after each successive letter, followed by a test for a non-letter or non-digit.
- Check for non-letter or non-digit is necessary (e.g., consider "*thenextvalue*", **ID** that has "*then*" as a prefix)
- **Prioritize reserved-words** when a lexeme matches *both* **ID** and pattern for a reserved word.

Transition diagram for unsigned numbers

digit -> [0-9]
digits -> digit⁺
***number* -> digits (. digits)? (E [+-] digits)?**

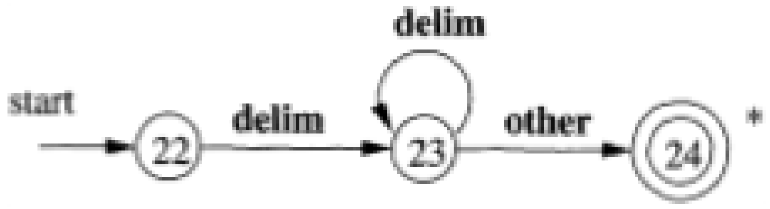
Transition diagram for unsigned numbers

digit* -> *[0-9]
***digits* -> *digit*$^{+}$**
***number* -> digits (. digits)? (E [+-] digits)?**



Stripping out white spaces

***ws* -> (blank | tab | newline)$^+$**

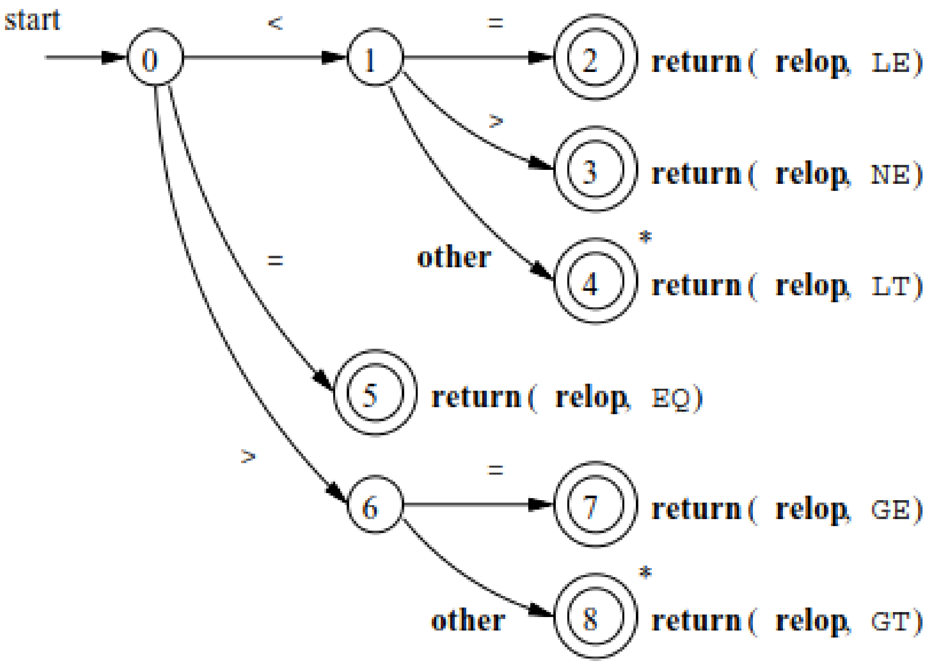


Lexical Analyzer Implementation from Transition Diagrams

- Different ways of using a collection of transition diagrams to build a lexical analyzer.

- In a **transition diagram**,
  - each state represented by a piece of code
  - Variable *state* holding information about the current state
  - *switch* based on the value of state

Example: Implementation of ***relop*** transition diagram




```
TOKEN getRelop()
{
    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);
        }
}
```

Lexical Analyzer implementation from Transition Diagrams

- What *fail()* does depends on the global error recovery strategy
- Resetting of *forward* pointer to *lexemeBegin*
  - *Allow another transition diagram to be applied from the* ***beginning of un-consumed*** *input*
- Change value of *state* to the start-state of another transition diagram that checks for another token

- If all transition diagrams are explored, *fail()* can initiate **error-correction phase**

Lexical Analyzer implementation from Transition Diagrams (Entire lexical analyzer)

- **Sequential** arrangement of transition diagrams to be tried
  - Function *fail()* resets pointer and starts the next transition diagram

- Run various transition diagrams in **parallel** feeding next input char to all of them
  - Resolving issues (lexeme match Vs. being able to consume more input)
  - **Take longest prefix** of the input that matches any pattern

- **Combine** all the transition diagrams into one
  - Allow to read input until there is no possible next state
  - Combining is easy for the considered running example (merging all initial states)
  - Combining is not trivial in general.

Lexical Analyzer Generator (Lex/flex)