Chapter 2
Scanning

한양대학교 컴퓨터공학부
컴파일러
2014년 2학기

Overview

- The scanning process
- Regular expressions
- Finite Automata
  - DFA
  - NFA

Scanning: introduction

- **Scanning** or **lexical analysis**
  - **Characters → Tokens**
- Tokens
  - Like the words in a natural language
  - Examples
    - Keywords: **if**, **while**
    - Identifiers
    - Special symbols: +, *, >=, …
- a special case of pattern matching
  - regular expressions: a standard notation for representing the patterns
  - finite automata: algorithms for recognizing patterns

The scanning process

- a[index] = 4 + 2

  → a / [/ index /] / = / 4 / + / 2

lexemes	tokens
a	identifier
[	left bracket
index	identifier
]	right bracket
=	assignment
4	number
+	plus sign
2	number

The scanning process

- Tokens
  - Reserved words
    - IF, THEN, ELSE,…
      - if, then, else, …
  - Special symbols
    - PLUS, MINUS, …
      - +, -
  - tokens for multiple strings
    - NUM
      - 123, 456, ….
    - ID
      - a, index

The scanning process

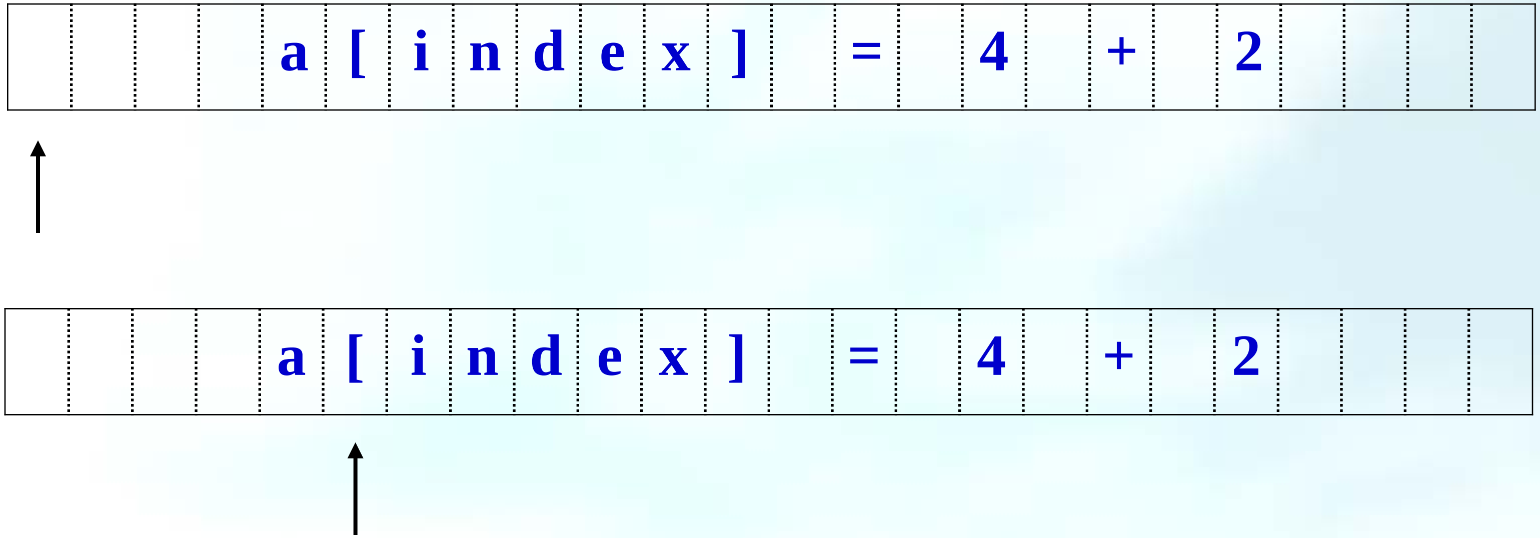
- Data structures for tokens

```
typedef struct
    {       TokenType tokenval;
             char * stringval;
            int numval;
    } TokenRecord;
```

```
typedef struct
        {           TokenType tokenval;
           union
                    { char * stringval;
                       int numval;
                    } attribute;
        } TokenRecord;
```

The scanning process

- Scanning and parsing are mixed together.
  - TokenType getToken(void);
  - This function returns the next token one by one.



				a	[	i	n	d	e	x	]		=		4		+		2				

				a	[	i	n	d	e	x	]		=		4		+		2				

The scanning process

- **Representing lexemes**
  - enumeration?
    - {if, then, else, … +, -, …, 0, 1,2, … a, b, c, …}
    - It may be appropriate for reserved words and special symbols.
    - Not appropriate for numbers and identifiers.
    - Inefficient
  - Representing using ***regular expression***

Regular expressions

- **Definitions**
  - **symbols**: characters
    - a, b, c, +, -, ...
  - **alphabet** (Σ): set of legal symbols
    - {A,B, C, …, Z, a, b, c, …, z}
  - **strings**: concatenation of symbols
    - I am a boy

- A **regular expression** *r* represents
  - a set of strings that is called the **language generated by** *r*, i.e., $L(r)$.

Regular expressions

- A ***symbol*** can be a regular exp.
  - a: $L(a) = \{a\}$, b: $L(b) = \{b\}$, …
  - **ε**: $L(\varepsilon) = \{\varepsilon\}$, **Φ**: $L(\Phi) = \{\}$

- ***Choice*** among regular exps is a regular exp.
  - $r|s$: $L(r|s) = L(r) \cup L(s)$
  - example
    - $L(a|b) = \{a\} \cup \{b\} = \{a,b\}$
    - $L(a|b|c|d) = \{a,b,c,d\}$

Regular expressions

- ***Concatenation*** of regular exps is a regular exp.
  - rs: $L(rs) = L(r)L(s)$
  - example) $L(ab) = \{ab\}$

- ***Repetition*** of a regular exp is a regular exp.
  - r^*: $L(r^*) = \{\varepsilon\} \cup L(r) \cup L(rr) \cup L(rrr)$ …
  - example) $L(a^*) = \{\varepsilon, a, aa, aaa, \ldots\}$
  - $L(a^*) = L(a)^*$
    - $L((\mathrm{a}|\mathrm{bb})^*) = L(\mathrm{a}|\mathrm{bb})^*$

Regular expressions

- Further examples
  - $(a|b)c$
    - $L((a/b)c) = L(a/b)L(c) = \{a,b\}\{c\} = \{ac, bc\}$
  - $(a|bb)^*$
    - $L((a/bb)^*) = \{ \qquad \}$

- **Precedence of operations**
  - $* \ > \ \cdot \ > \ |$
  - $a|bc^*$: $L(a/bc^*) = L(a) \cup L(b)L(c)^*$

- Names
  - (0|1|2|...|9)(0|1|2|...|9)*
  - It can be rewritten as *digit digit** where *digit* = 0|1|2|...|9.

Examples

- The set of all strings over $\{a,b,c\}$ containing exactly one b.
  - $(a|c)^*b(a|c)^*$

- The set of all strings over $\{a,b,c\}$ containing at most one b.
  - $(a|c)^* \mid (a|c)^*b(a|c)^*$
  - $(a|c)^*(b/\varepsilon)(a|c)^*$

- The set of all strings over $\{a,b\}$ consisting of a single b surrounded by the same number of a's.
  - $\{b, aba, aabaa, \ldots\}$
  - impossible

Example 2.4

- Consider the strings over the alphabet ∑ = {a, b, c} that contain no two consecutive b’s.

Example 2.5

- Consider the alphabet ∑ = {a, b, c} and the regular expression

  ((b|c)*a(b|c)*a)* (b|c)*

Extensions to regular expressions

- **+** : one or more repetitions
  - $r+ = rr^*$
  - (0|1|2|...|9)(0|1|2|...|9)* → (0|1|2|...|9)+

- **.** : any symbol in the alphabet
  - $.^*b.^*$

- **-** : a range of symbols
  - *a|b|c* → [*abc*]
  - *a|b|...|z* → [*a-z*]
  - [*a-zA-Z*]

Extensions to regular expressions

- **~, ^**: any symbol not in a given set
  - $\sim(a|b|c)$ or $[\wedge abc]$: a character that is not either a or b or c

- **?**: optional subexpressions
  - *natural* = [0-9]+
  - *signedNatural* = *natural* | + *natural* | - *natural*

    → *signedNatural* = (+|-)? *natural*

Regular expressions for PL tokens

- Reserved words
  - *reserved* = if | while | do | …
- Special symbols
  - +, =, :=, ++, …
- Identifiers
  - *letter* = [*a-zA-Z*]
  - *digit* = [0-9]
  - *identifier* = *letter*(*letter*|*digit*)*

Regular expressions for PL tokens

- Numbers
  - *nat* = [0-9]+
  - *signedNat* = (+|-)? *nat*
  - *number* = *signedNat*(“.” *nat*)? (E *signedNat*)?

- Comments
  - {this is a Pascal comment}
  - -- this is an Ada comment
  - /* this is a C comment */

Regular expressions for PL tokens

- Comments
  - {this is a Pascal comment}
    - {(~})*}
  - -- this is an Ada comment
    - --(~*newline*)*
  - /* this is a C comment */
    - *ba* …. *ab* where *b* = / and *a* = *.
    - *ba* (*b**(*a**~(*a*|*b*)*b**)**a**) *ab*
    - usually handled by ad hoc methods

Regular expressions for PL tokens

- **Ambiguity**
  - Is if a keyword or an identifier?
  - Is temp an identifier temp or identifiers te and mp?

- **Disambiguating rules**
  - **Keyword** is preferred to **identifiers**.
    - if is a keyword.
  - principle of longest substring
    - temp is an identifier temp.

Regular expressions for PL tokens

- **Token delimiters**
  - White space
    - *whitespace* = (*blank* | *tab* | *newline* | *comment*)+
    - do if, do/**/if
  - Characters that are unambiguously part of other tokens.
    - xtemp=ytemp

Regular expressions for PL tokens

- lookahead and backtrack
  - single-character lookahead
    - xtemp=ytemp
  - backtrack (more than single-character lookahead)
    - FORTRAN
      - DO99I=1,10 (loop)
      - DO99I=1.10 (assignment)

Chapter 2
Scanning
– Finite Automata –

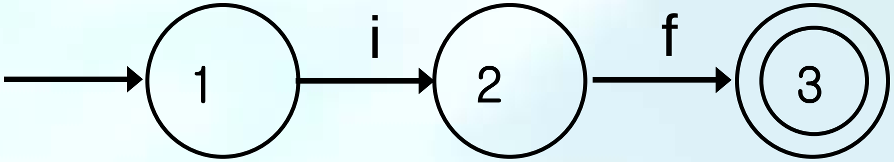
한양대학교 컴퓨터공학부

컴파일러

2014년 2학기

Finite automata

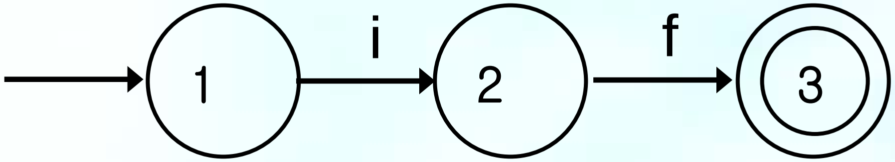
- Finite automata consists of
  - states
  - transitions (on symbols)
  - start state
  - accepting states



Finite automata

- Used for recognizing pattern represented by regular expressions

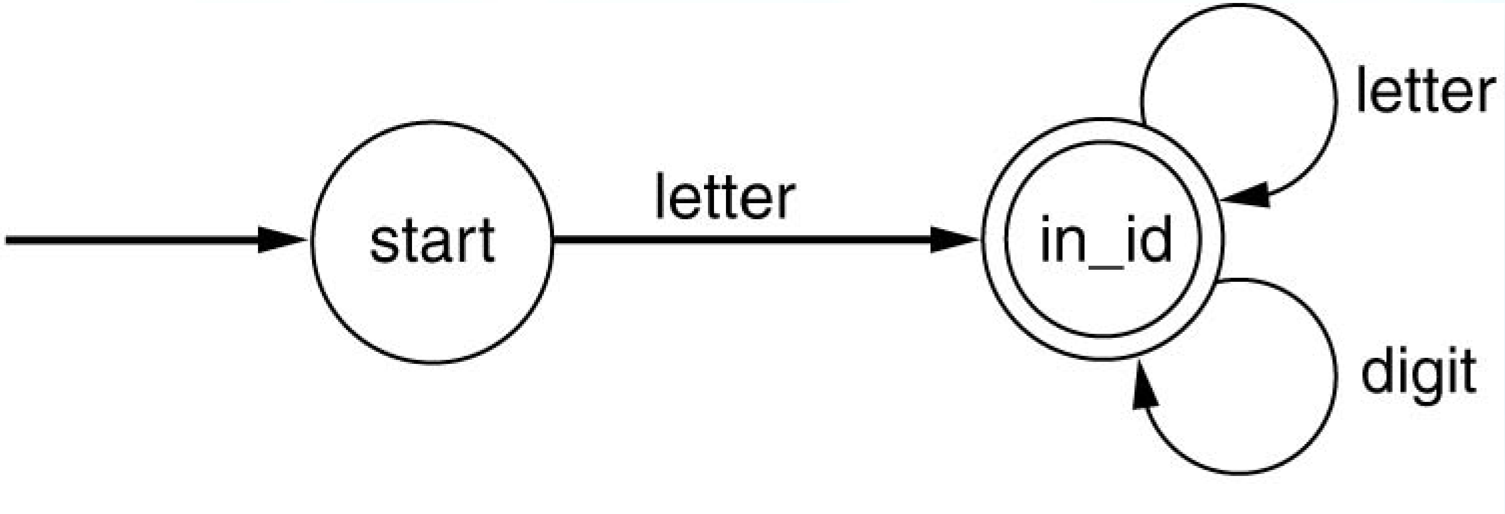
if



$$1 \xrightarrow[i]{} 2 \xrightarrow[f]{} 3$$

Finite automata

*identifier = letter(letter|digit)**

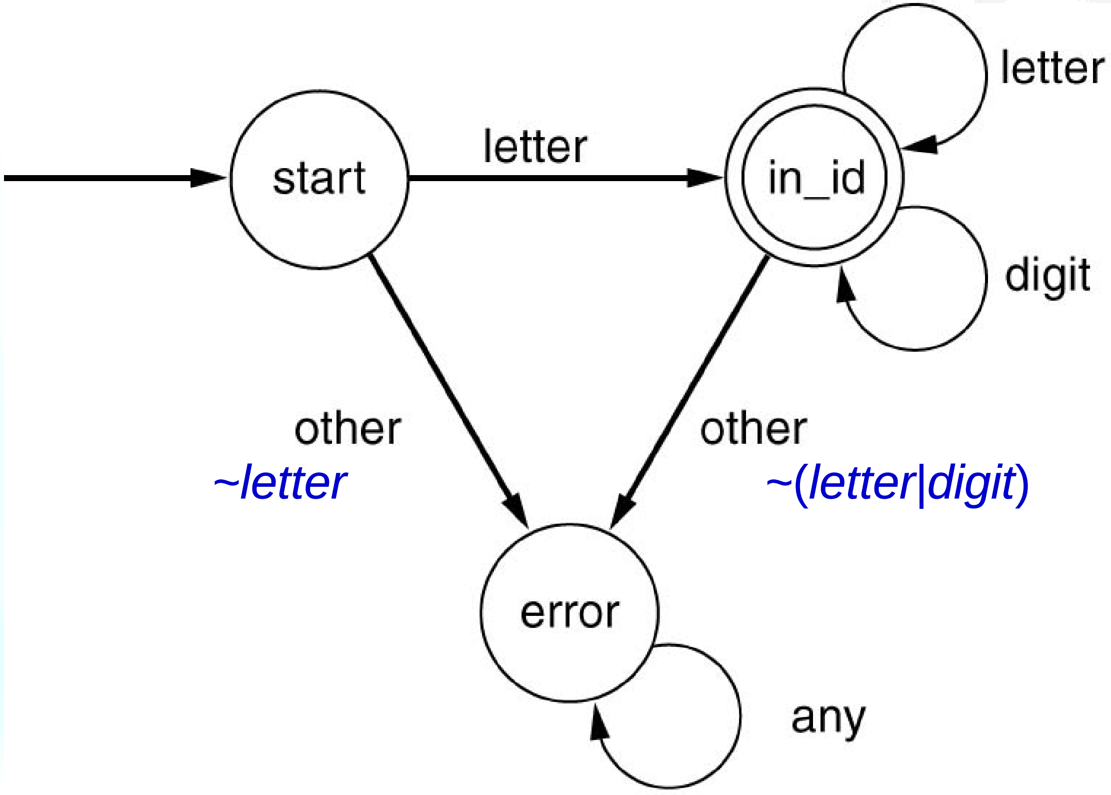


Mathematical definition of DFA

- A **DFA** M consists of an alphabet ∑, a set of states S, a transition function T: S x ∑ → S, a start state $s_0 \in S$, and a set of accepting states $A \subset S$. The language accepted by M, written L(M), is defined to be the set of strings of characters $c_1c_2 \ldots c_n$ with each $c_i \in \sum$ such that there exist states $s_1 = T(s_0, c_1)$, $s_2 = T(s_1, c_2)$, …, $s_n = T(s_{n-1}, c_n)$ with s_n an element of A.

Finite automata

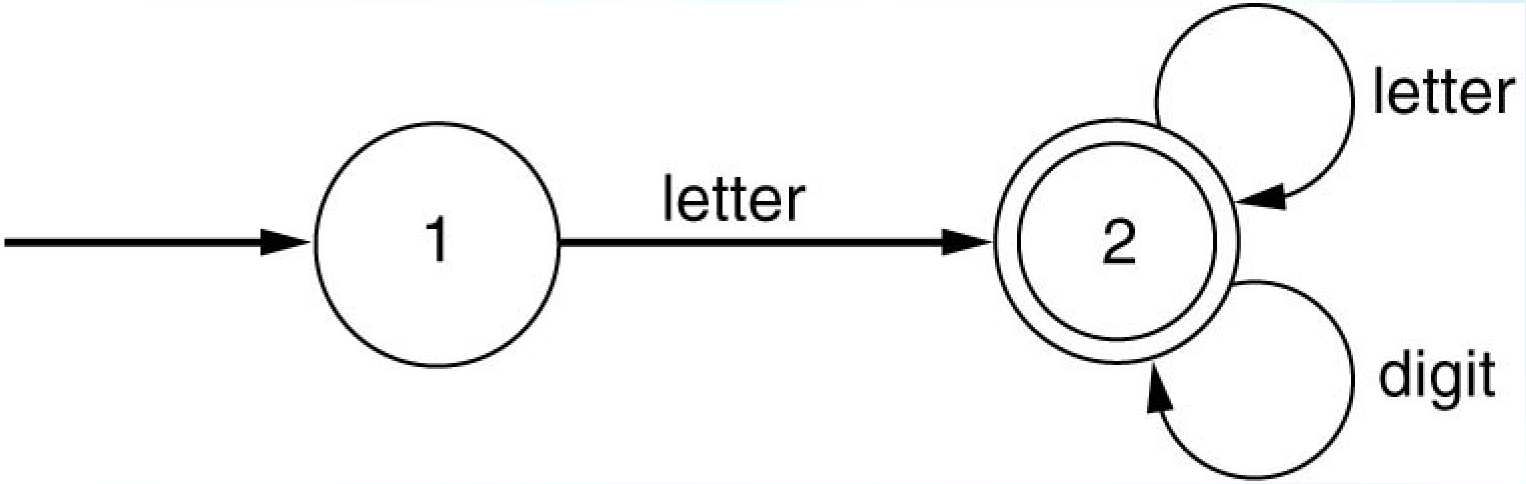
- Error transitions are not drawn.



Finite automata

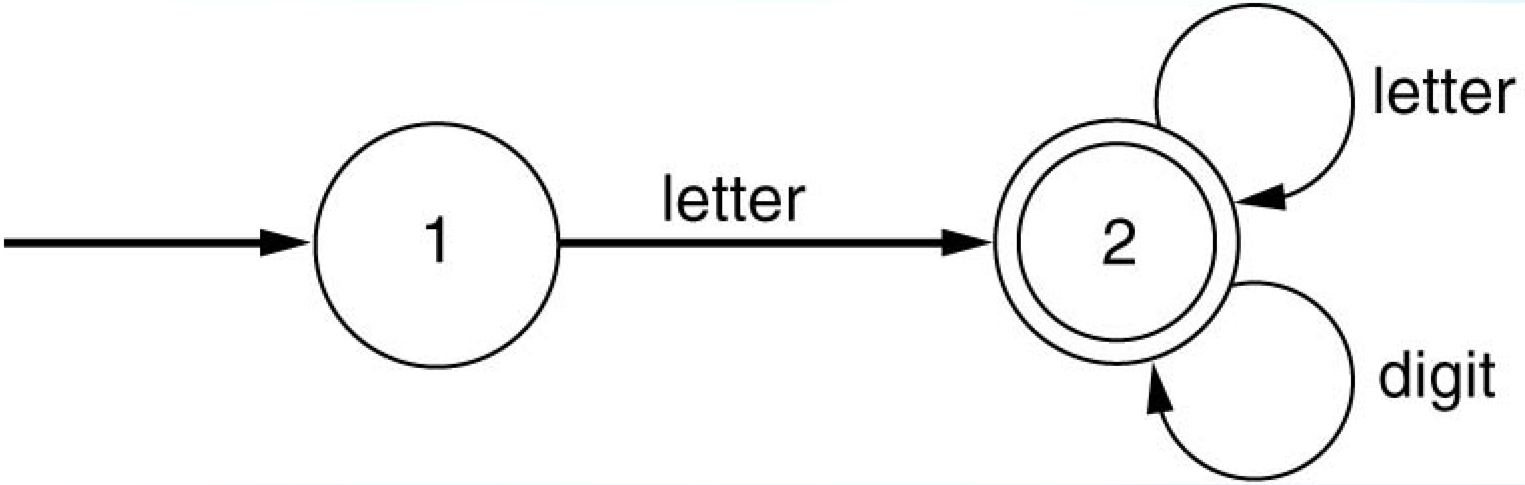
- xtemp
  - *1 → 2 → 2 → 2 → 2 → 2*

    x t e m p



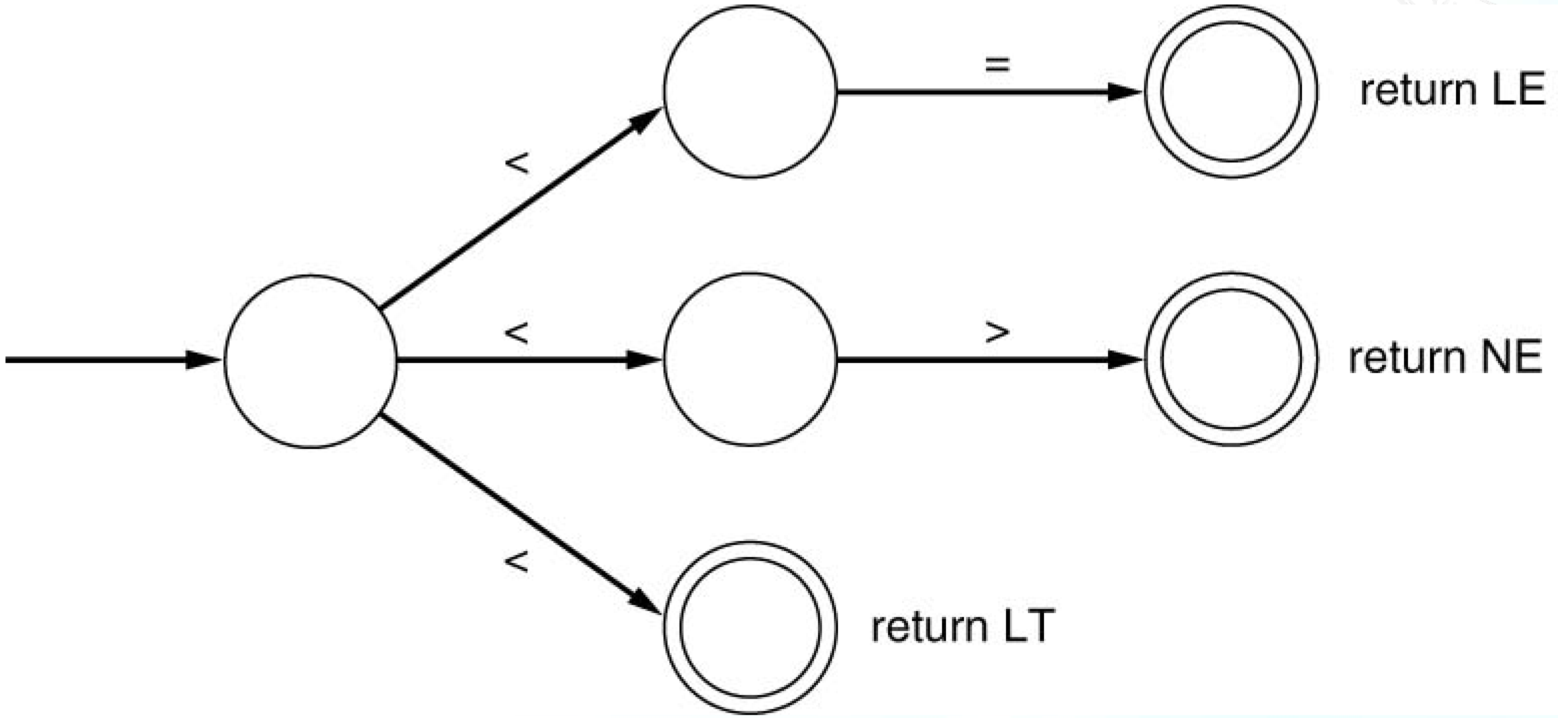
Finite automata

- DFA (deterministic finite automaton)
  - Given a state and a symbol, the next state is unique.



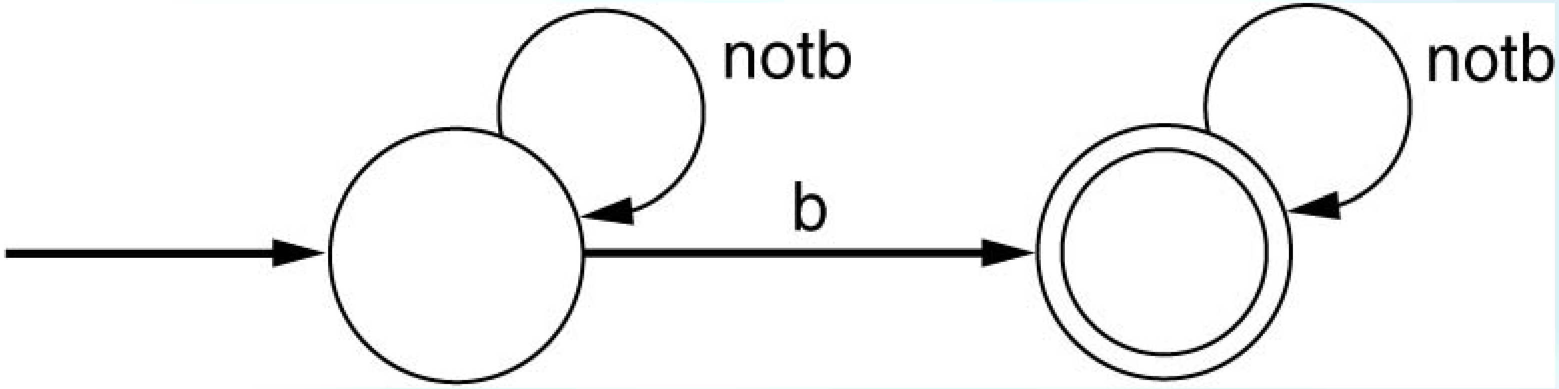
Finite automata

- NFA (nondeterministic finite automaton)
  - Given a state and a symbol, the next state is not unique.



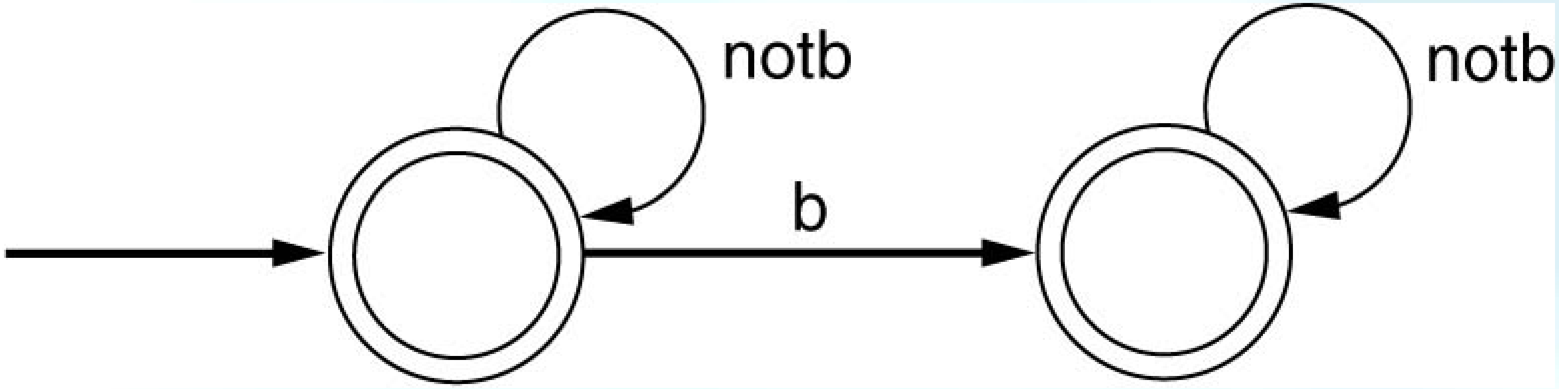
DFA

- Examples
  - The set of all strings over {*a*,*b*,*c*} containing exactly one *b*.
    - (*a*|*c*)**b*(*a*|*c*)*



DFA

- Examples
  - The set of all strings over {*a*,*b*,*c*} containing at most one *b*.
    - (*a*|*c*)* | (*a*|*c*)**b*(*a*|*c*)*
    - (*a*|*c*)*(*b*/ε)(*a*|*c*)*



DFAs for PL tokens

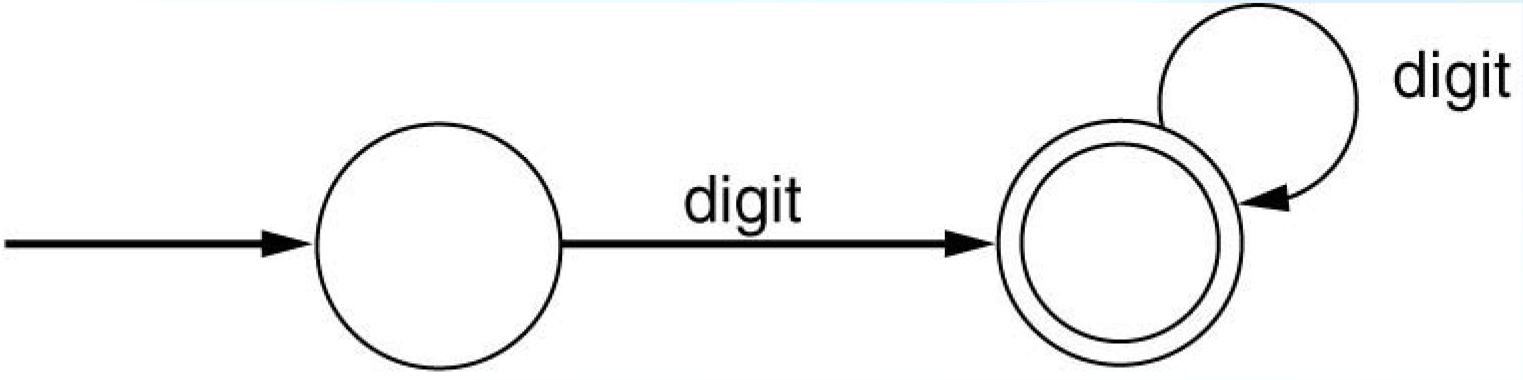
- Examples
  - *nat* = [0-9]+
  - *signedNat* = (+|-)? *nat*
  - *number* = *signedNat* (“.” *nat*)? (E *signedNat*)?

  - *digit* = [0-9]
  - *nat* = *digit*+
  - *signedNat* = (+|-)? *nat*
  - *number* = *signedNat* (“.” *nat*)? (E *signedNat*)?

DFAs for PL tokens

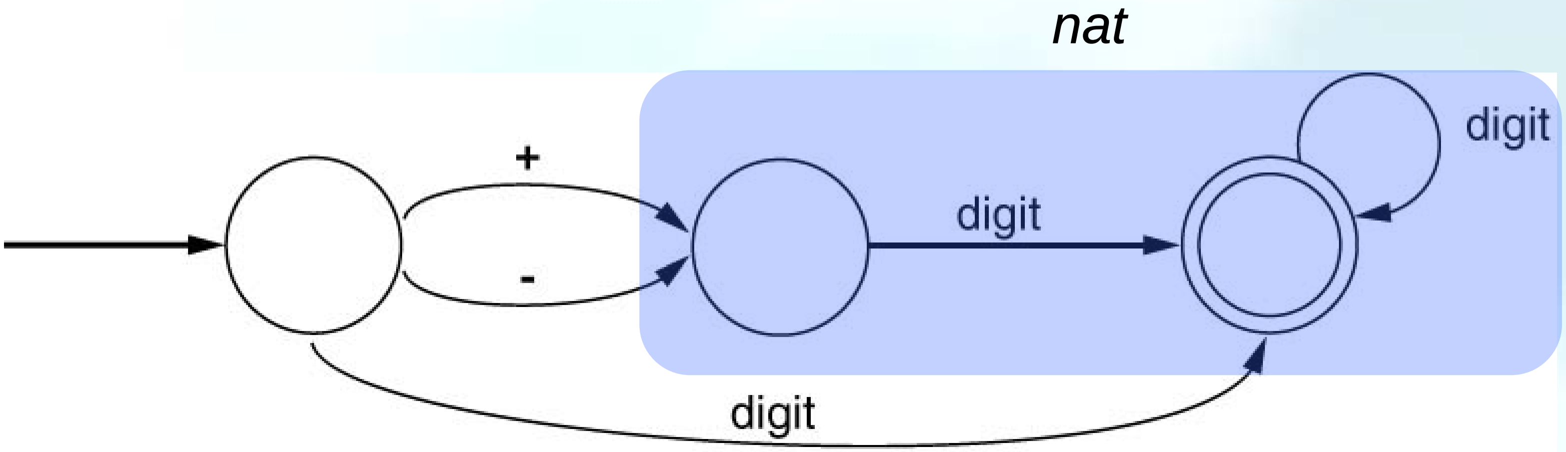
- Examples

  - *digit* = [0-9]
  - *nat = digit+*
  - *singedNat* = (+|-)? *nat*
  - *number* = *signedNat* (“.” *nat*)? (E *signedNat*)?



DFAs for PL tokens

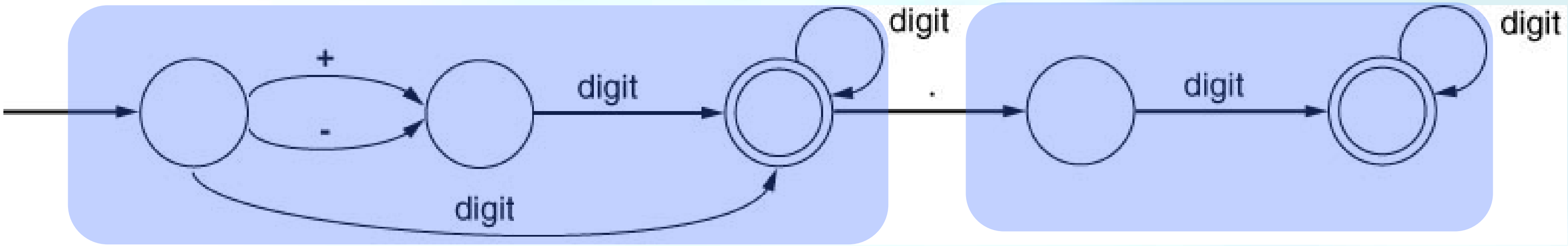
- Examples
  - *digit* = [0-9]
  - *nat* = *digit*+
  - *signedNat* = (+|-)? *nat*
  - *number* = *signedNat* (“.” *nat*)? (E *signedNat*)?



DFAs for PL tokens

- Examples
  - *digit* = [0-9]
  - *nat* = *digit*+
  - *signedNat* = (+|-)? *nat*
  - *number* = *signedNat* (“.” *nat*)? (E *signedNat*)?

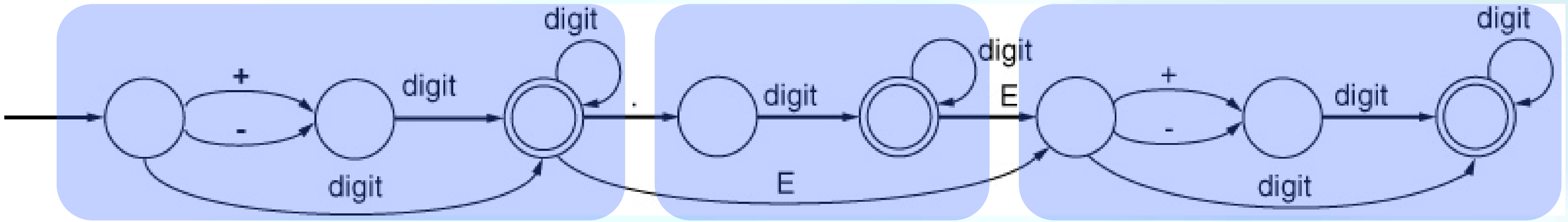
signedNat *nat*



DFAs for PL tokens

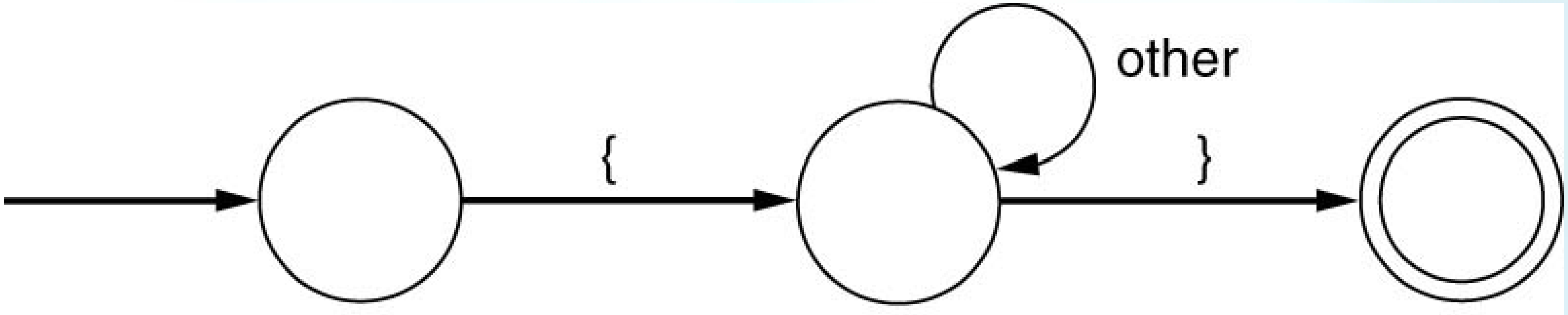
- Examples
  - *digit* = [0-9]
  - *nat* = *digit*+
  - *signedNat* = (+|-)? *nat*
  - *number* = *signedNat* (“.” *nat*)? (E *signedNat*)?

signedNat *nat* *signedNat*



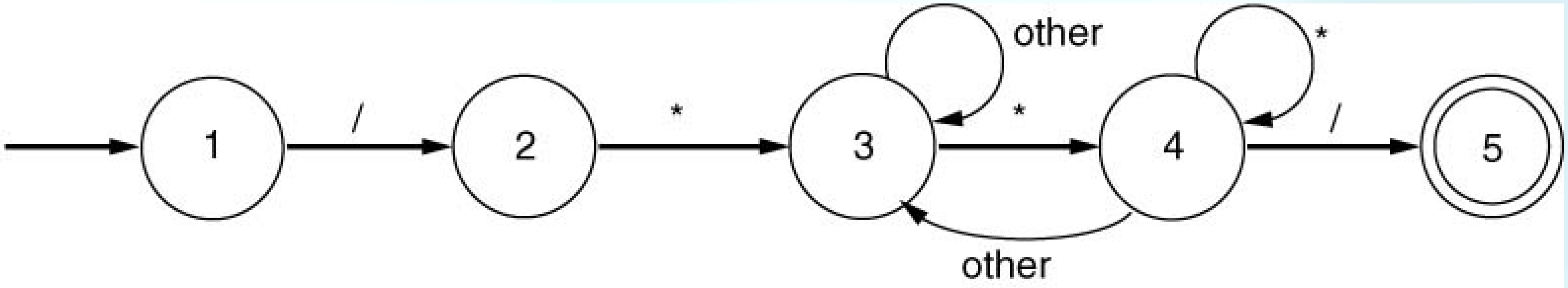
DFAs for PL tokens

- Comments
  - {this is a Pascal comment}
    - {(~})*}



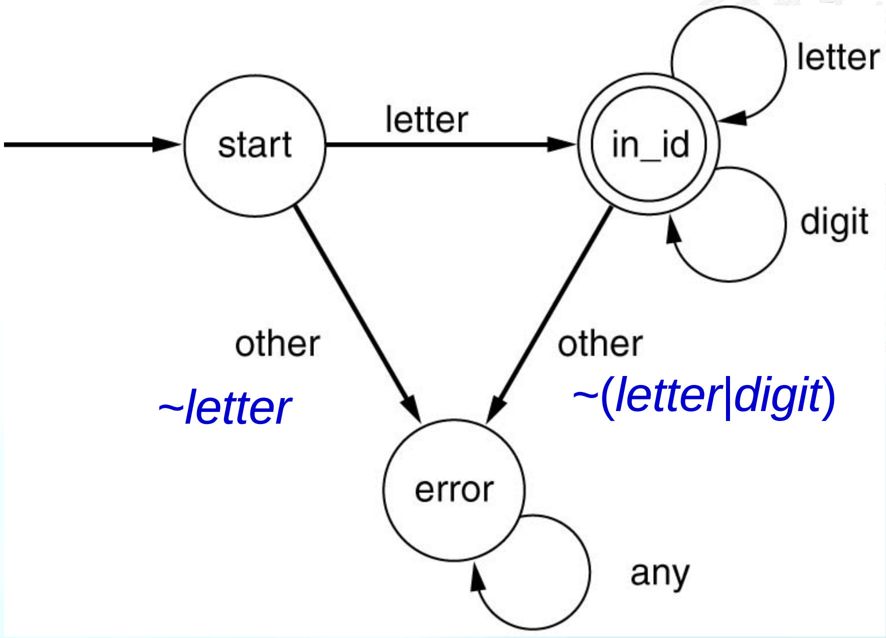
DFAs for PL tokens

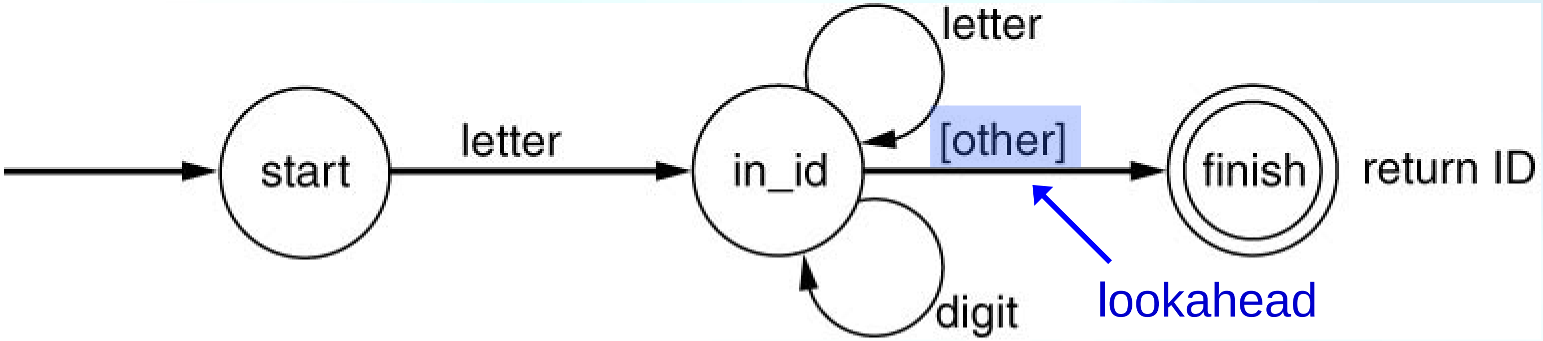
- Comments
  - /* this is a C comment */
    - $ba\ (b^*(a^*\sim(a|b)b^*)^*a^*)\ ab$



DFAs for PL tokens

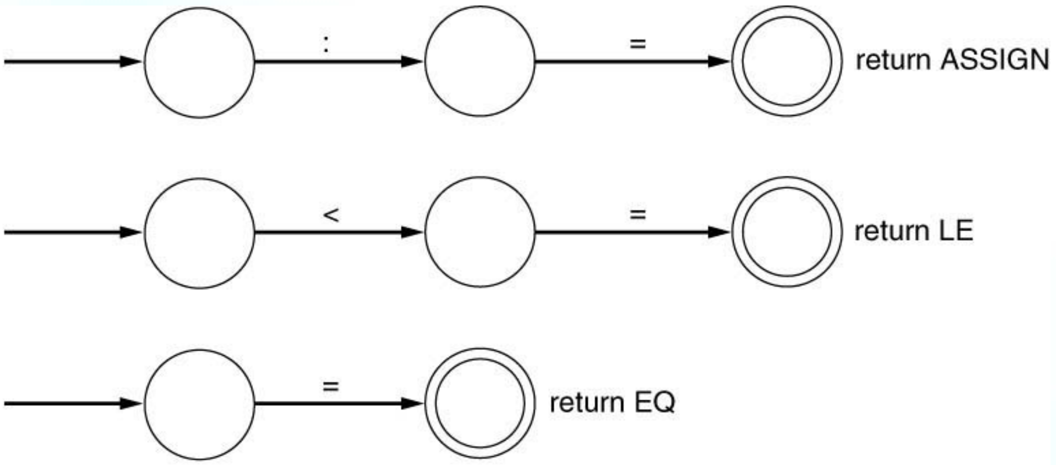
- longest substring?

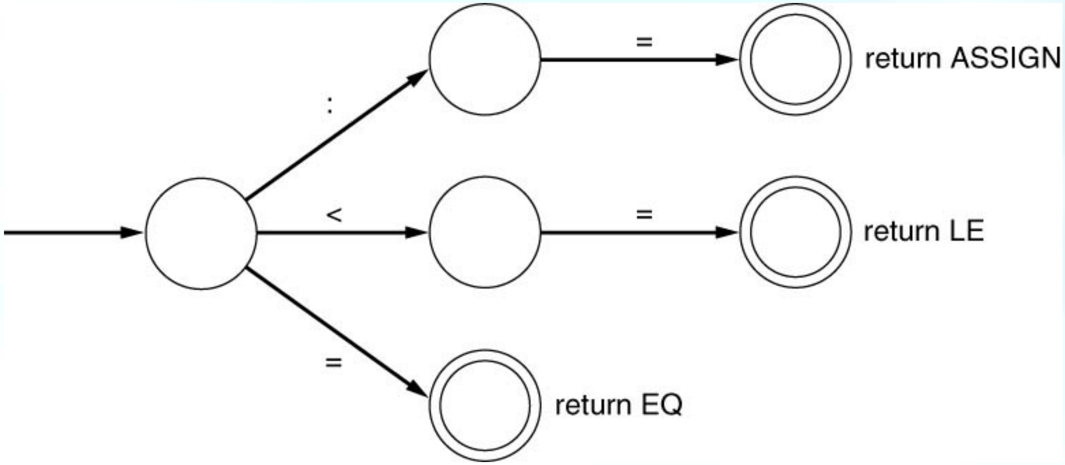




Merging DFAs

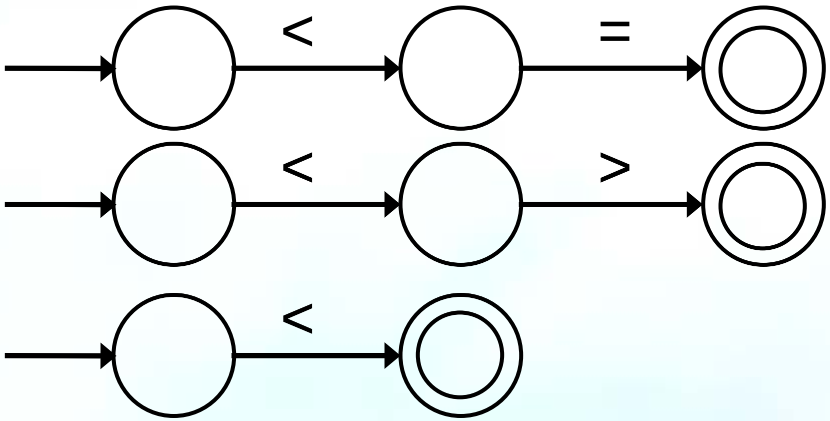
- a DFA for each token → DFA for some tokens

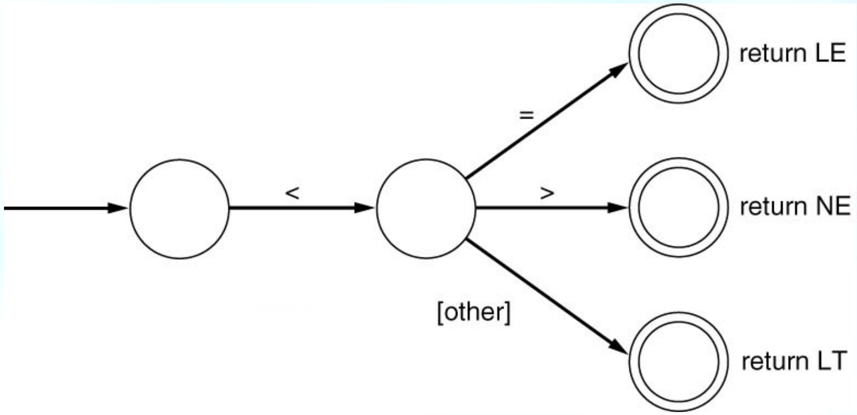




Merging DFAs

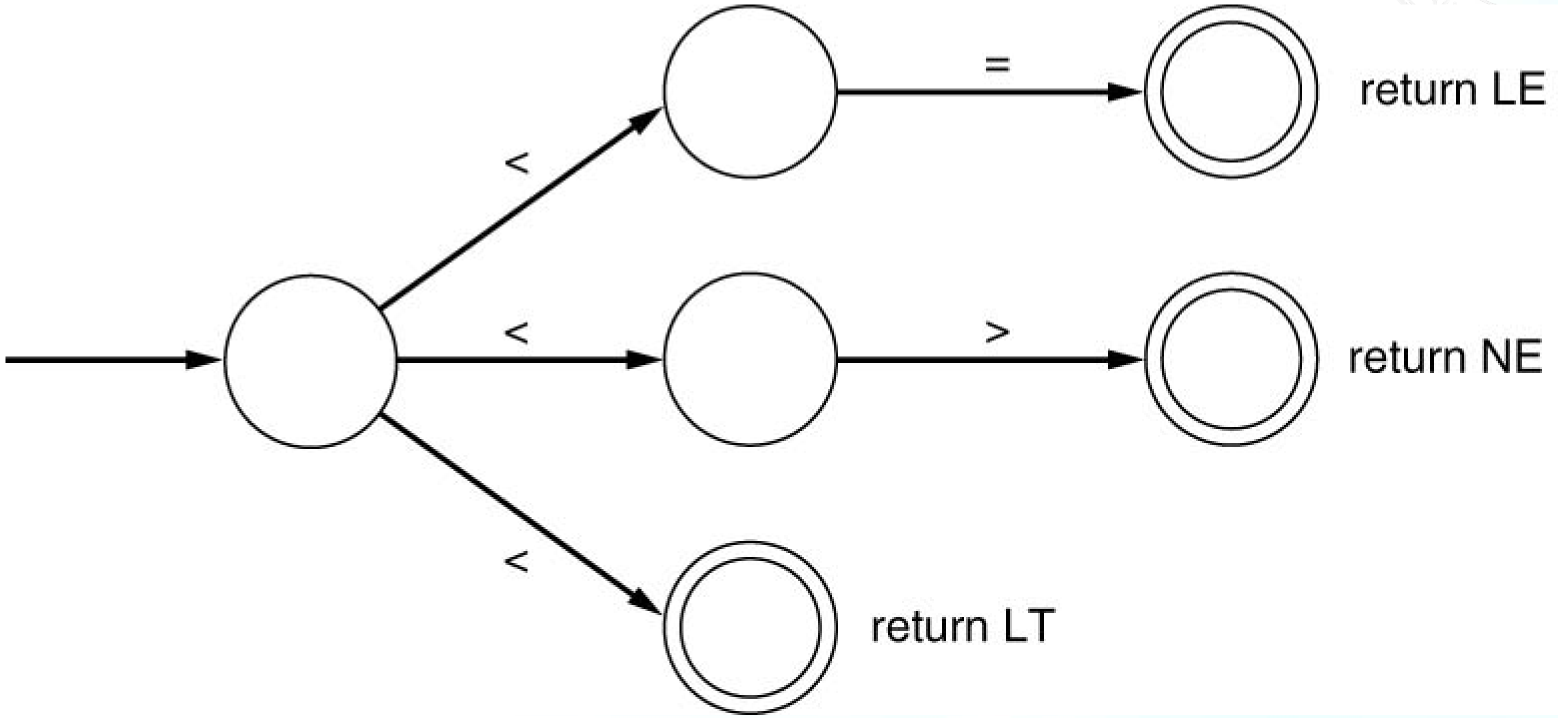
- Merging DFAs when tokens begin with the same symbol.





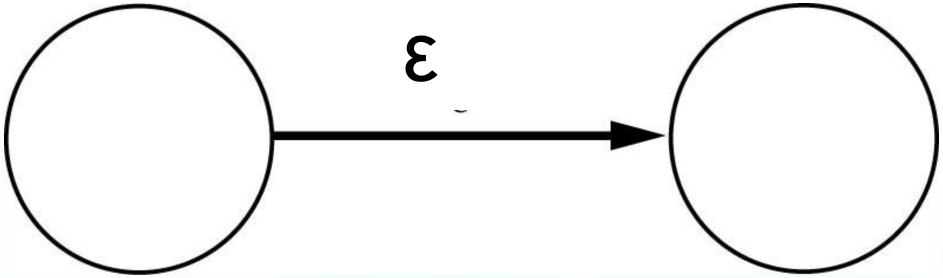
NFA

- Given a state and a symbol, the next state is not unique.



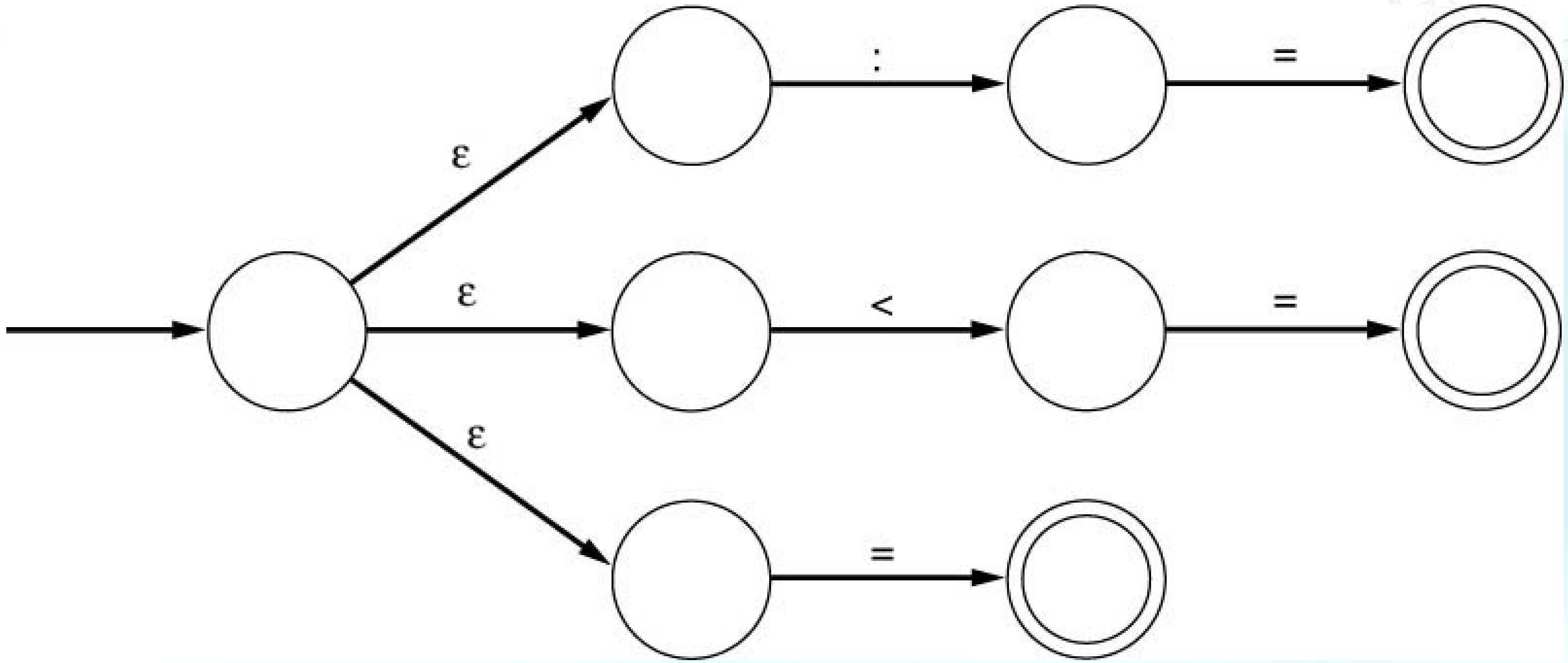
NFA

- It also includes ε-transitions.



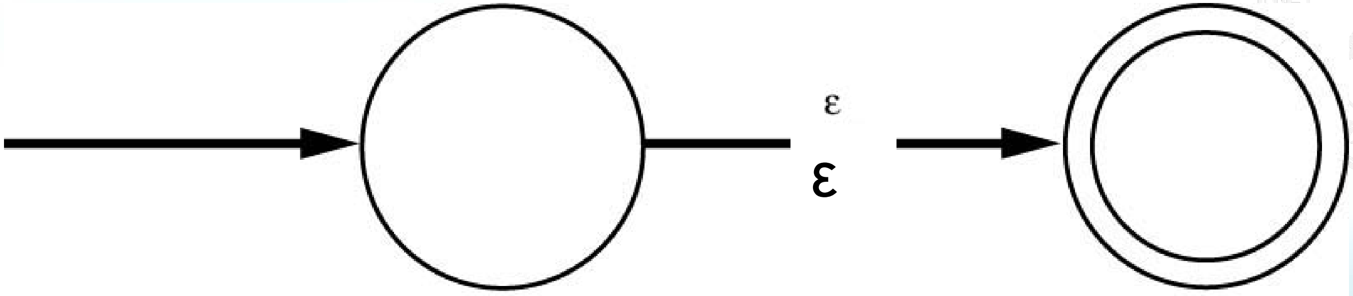
NFA

- ε-transitions makes merging automata without combining states.

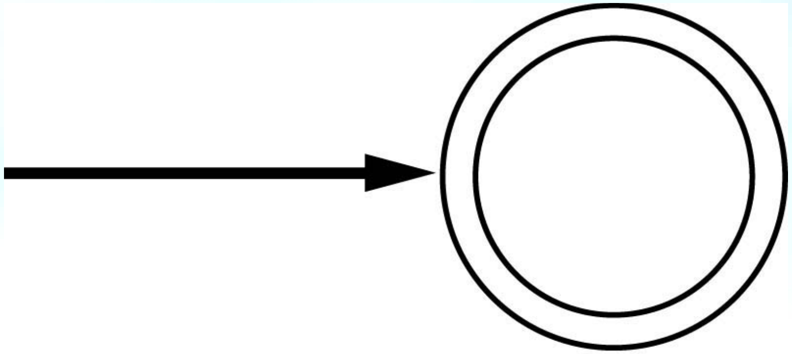


NFA

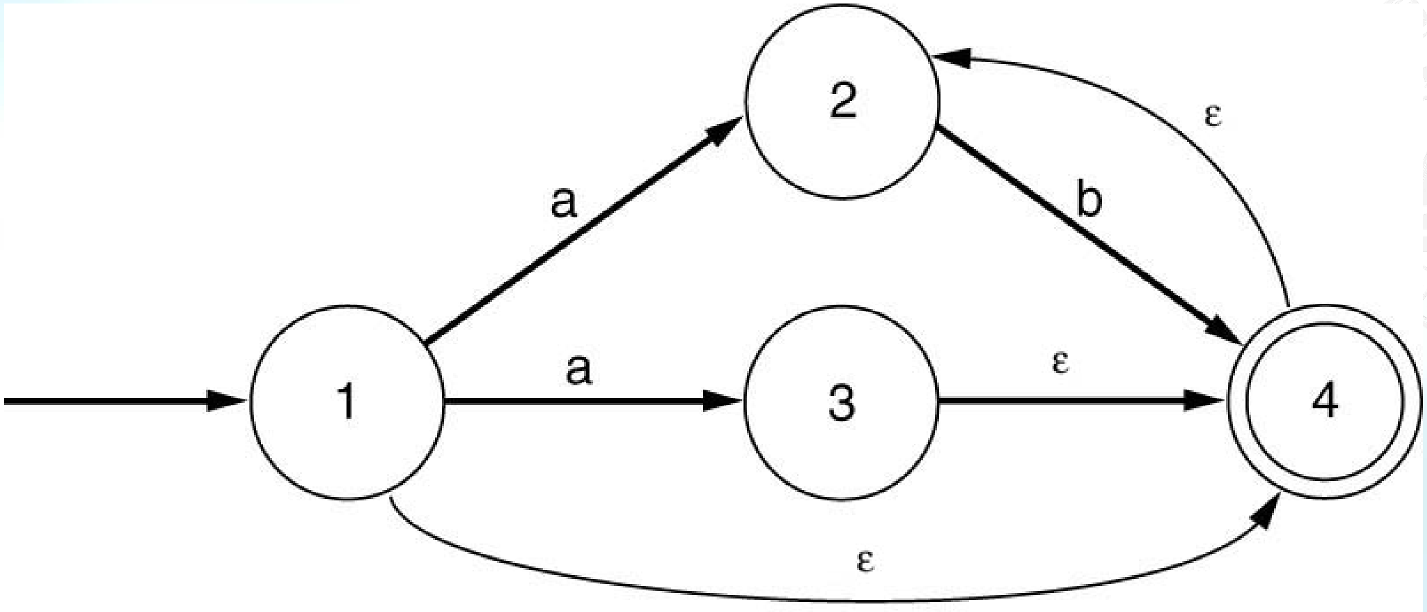
- NFA for the empty string.



- DFA for the empty string



NFA

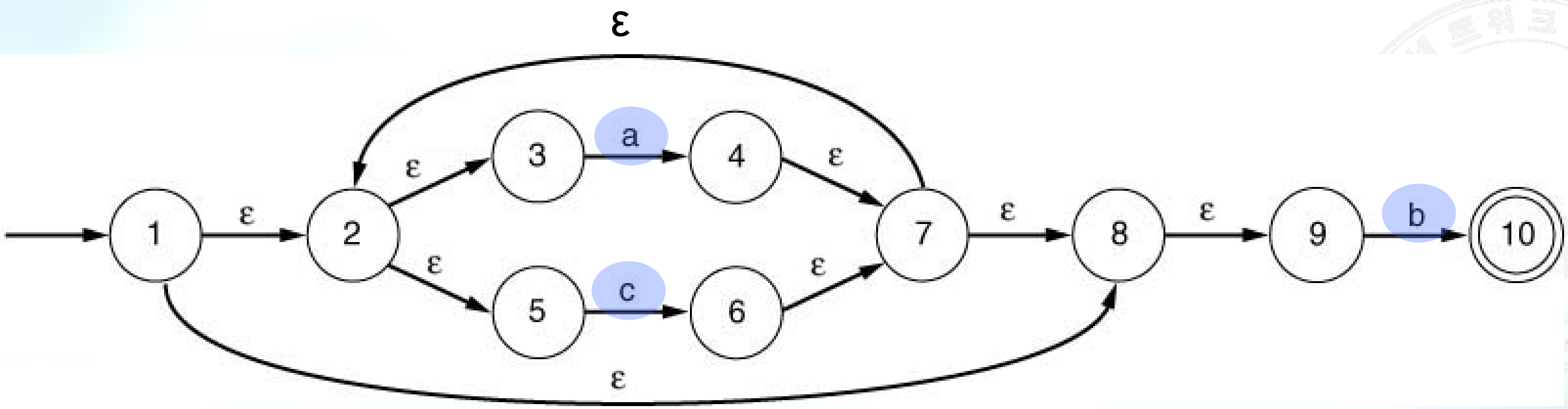


abb

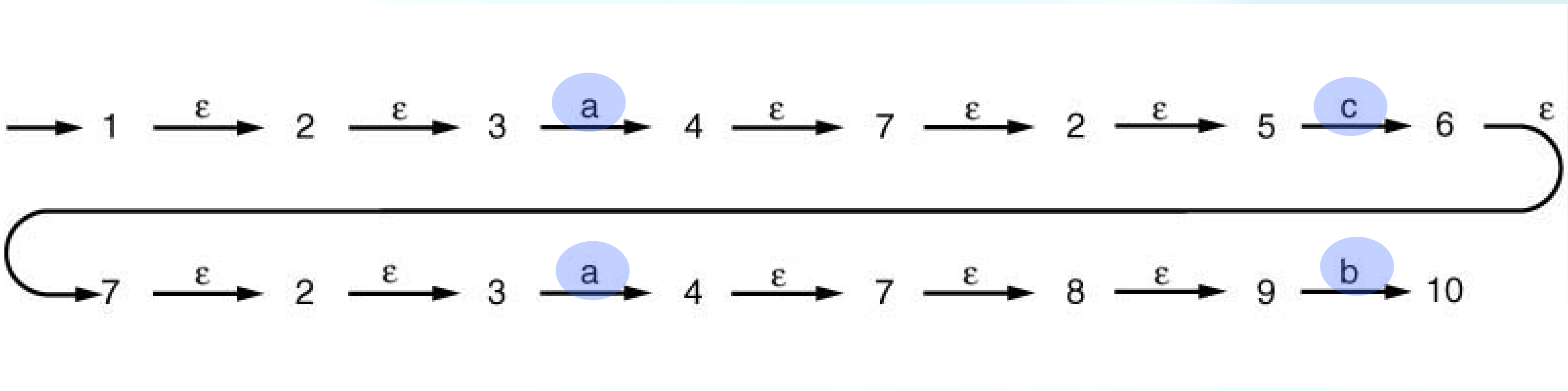
$$1 \xrightarrow{a} 2 \xrightarrow{b} 4 \xrightarrow{\varepsilon} 2 \xrightarrow{b} 4$$

$$1 \xrightarrow{a} 3 \xrightarrow{\varepsilon} 4 \xrightarrow{\varepsilon} 2 \xrightarrow{b} 4 \xrightarrow{\varepsilon} 2 \xrightarrow{b} 4$$

NFA

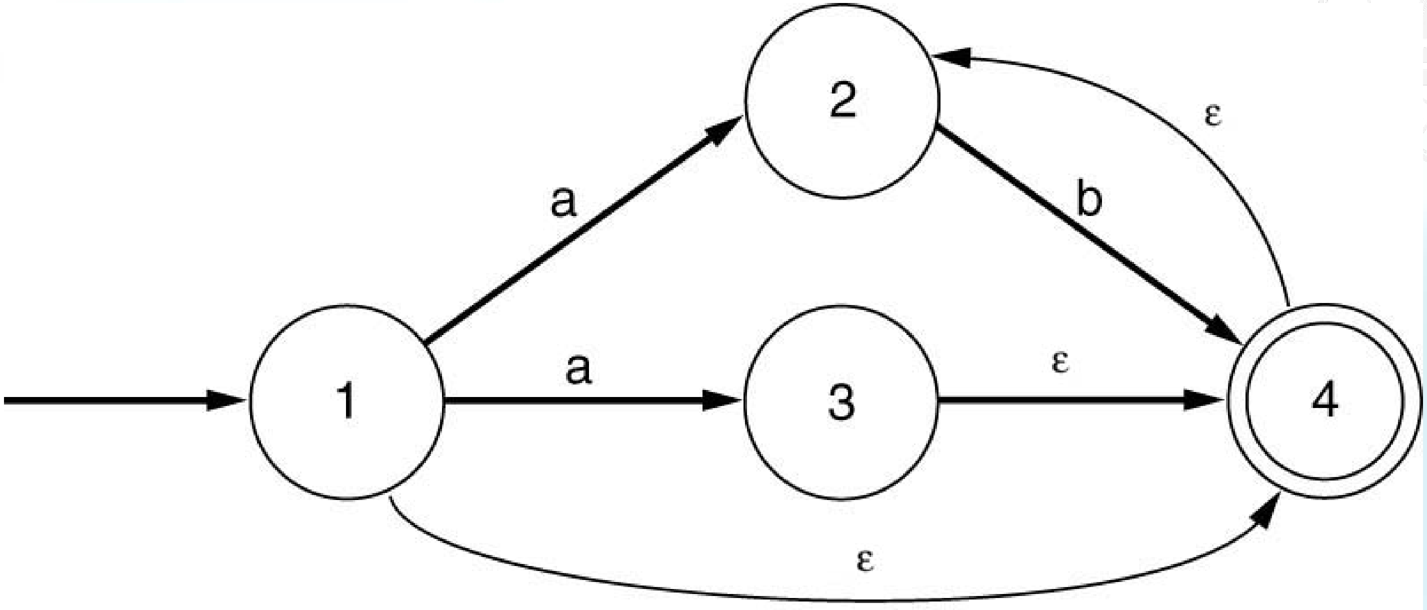

ε
1
ε
2
ε
ε
3
a
4
5
c
6
ε
ε
7
ε
8
ε
9
b
10
ε

acab


1
ε
2
ε
3
a
4
ε
7
ε
2
ε
5
c
6
ε
7
ε
2
ε
3
a
4
ε
7
ε
8
ε
9
b
10

NFA

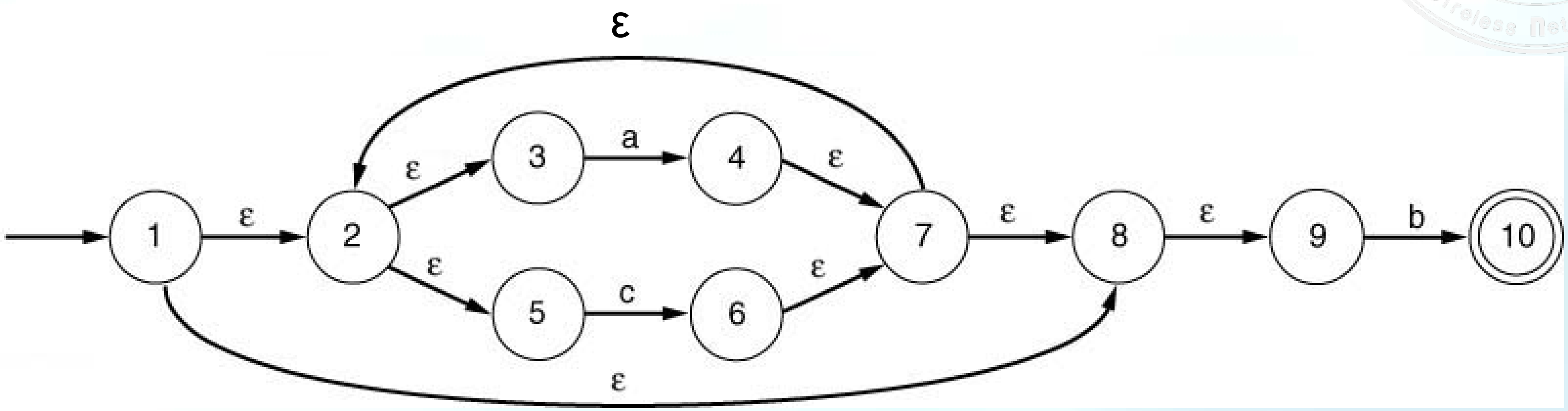
- Corresponding regular expression



ab+|ab*|b*   or   (a|ε)b*

NFA

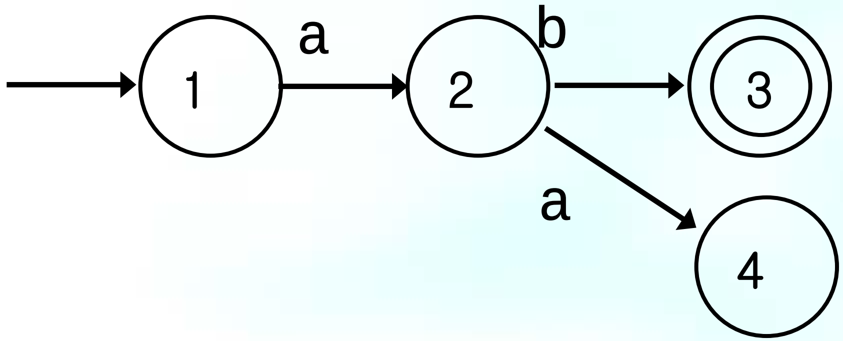
- Corresponding regular expression



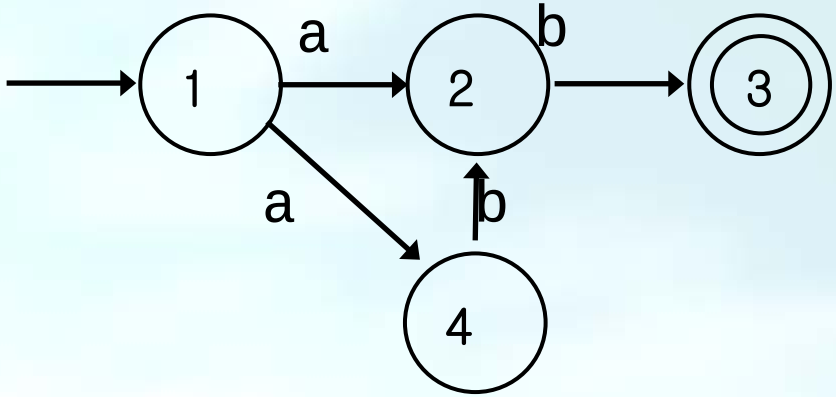
(a|c)*b

Finite Automata

- An alphabet Σ
  - the set of symbols: {a, b, … }
- a set of states S
  - normal states, a start state, a set of accepting states
- a transition function T (for every pair of each state and each symbol)
  - T: S X $\Sigma \rightarrow S$ (DFA)
  - T: S X $(\Sigma \cup \{\varepsilon\}) \rightarrow \boldsymbol{p}(S)$ (NFA)



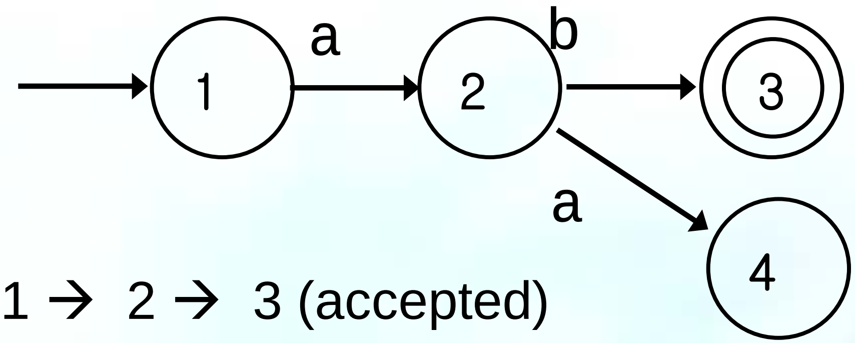
T(1,a) → 2



T(1,a) → {2,4}

Finite Automata

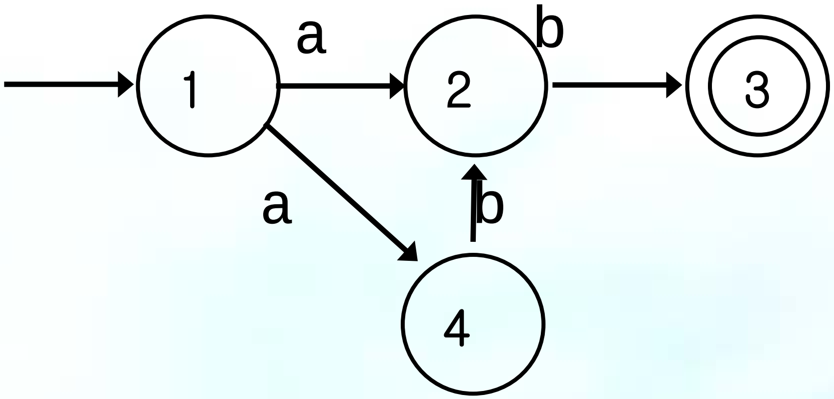
- Strings accepted by a finite automata
  - Strings that can reach one of the accepting states using transitions from the start state.
  - DFA



- ab: $1 \xrightarrow[a]{} 2 \xrightarrow[b]{} 3$ (accepted)
- aa: $1 \xrightarrow[a]{} 2 \xrightarrow[a]{} 4$ (not accepted)

Finite Automata

- Strings accepted by a finite automata
  - Strings that can reach one of the accepting states using transitions from the start state.
  - NFA



- ab: $1 \xrightarrow[a]{} \{2,4\} \xrightarrow[b]{} \{3,2\}$ (accepted)

subset construction

What if ε-transitions exist?

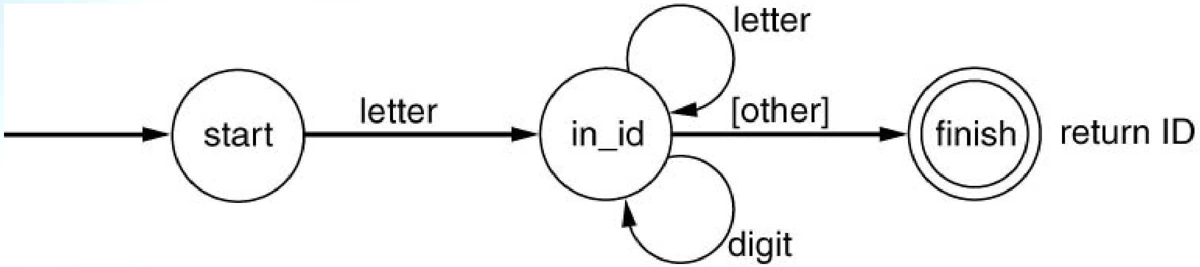
Finite Automata

- The language accepted by a finite automata
  - The set of strings accepted by the finite automata.

a DFA for all PL tokens

- It is possible to generate a DFA for each token and merging the DFAs.

Implementation of Finite Automata



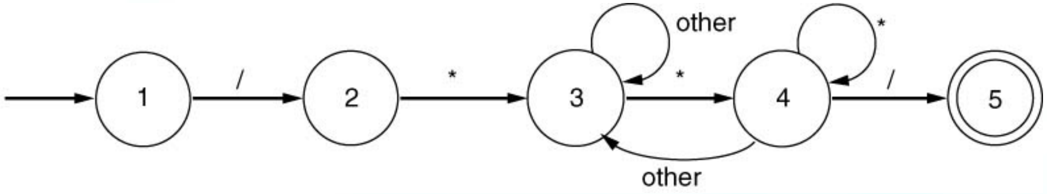
```
{ starting in state 1 }
If the next char is a letter then
  advance the input;  {now in state 2}
  while the next char is a letter or a digit do
     advance the input; { stay in state 2 }
   end while;
   accept;
else
   { error or other cases }
end if;
…
```

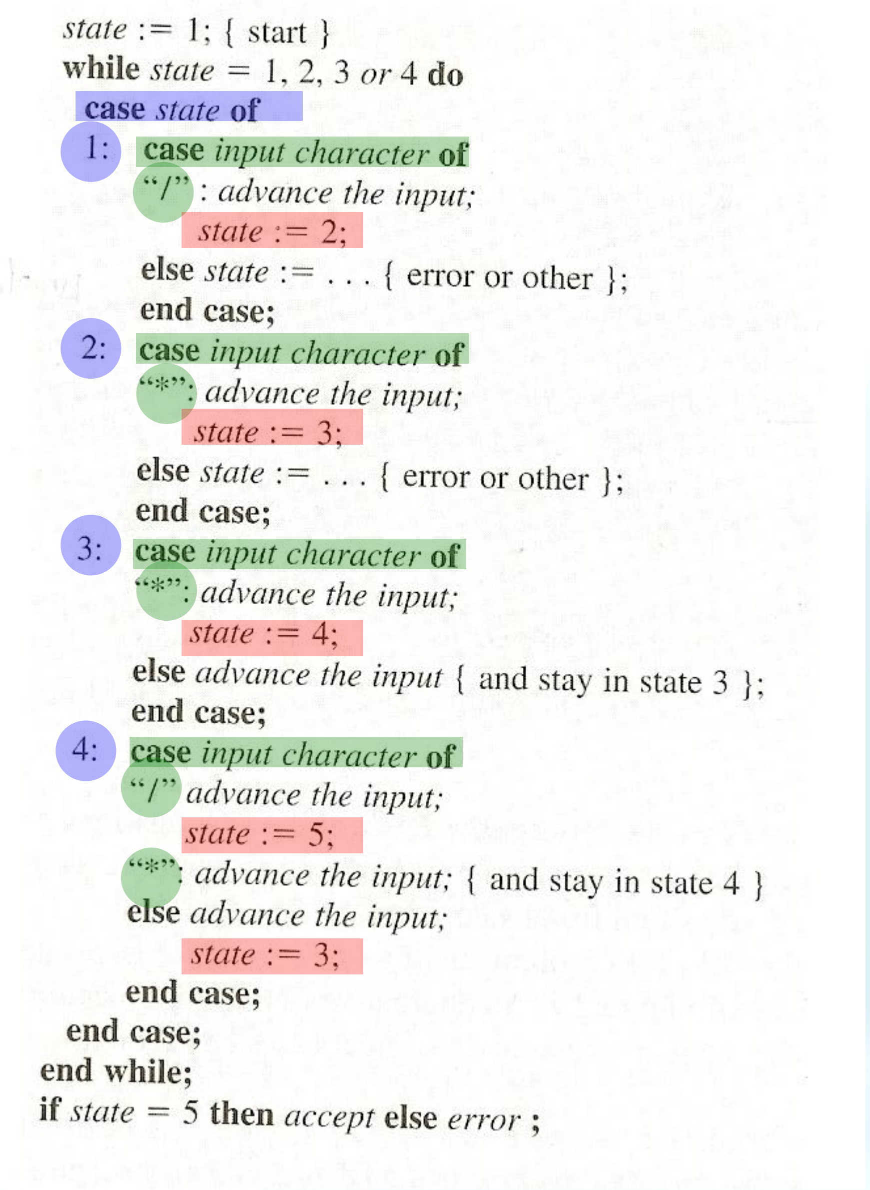
Implementation of Finite Automata

```
state := 1; { start }
while state = 1 or 2 do
   case state of
    1:  case input char of
           letter: advance the input;
                state := 2;
         else state := ERROR;
         end case;
    …
```

DFA → Code

- Using nested case
  - The DFA for C comments





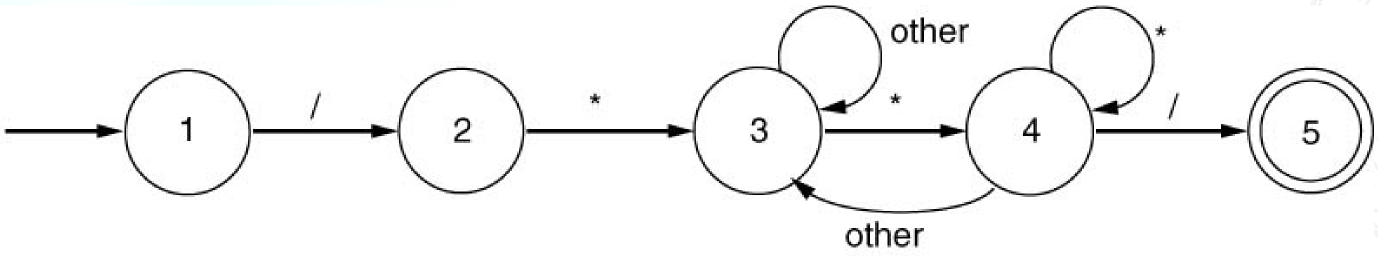
```
state := 1; { start }
while state = 1, 2, 3 or 4 do
 case state of
 1: case input character of
     "/" : advance the input;
          state := 2;
     else state := . . . { error or other };
     end case;
 2: case input character of
     "*": advance the input;
          state := 3;
     else state := . . . { error or other };
     end case;
 3: case input character of
     "*": advance the input;
          state := 4;
     else advance the input { and stay in state 3 };
     end case;
 4: case input character of
     "/" advance the input;
          state := 5;
     "*": advance the input; { and stay in state 4 }
     else advance the input;
          state := 3;
     end case;
 end case;
end while;
if state = 5 then accept else error ;
```

Implementation of Finite Automata

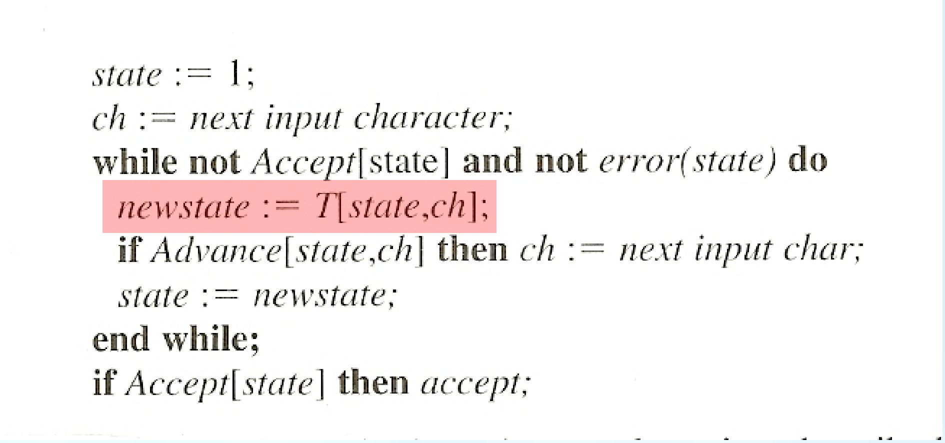
```
state := 1;
ch := next input char;
while not Accept[state] and not error(state) do
  newstate := T[state, ch];
  if Advance[state, ch] then
    ch := next input char;
  state := newstate;
end while;
if Accept[state] then accept;
```

DFA → Code

- Using a transition table



input state	/	*	other	Accepting
1	2			no
2		3		no
3	3	4	3	no
4	5	4	3	no
5				yes



```
state := 1;
ch := next input character;
while not Accept[state] and not error(state) do
  newstate := T[state,ch];
  if Advance[state,ch] then ch := next input char;
  state := newstate;
end while;
if Accept[state] then accept;
```

- Waste of space

a DFA for all PL tokens

- It is possible to generate a DFA for each token and merging the DFAs.
- However, it is not a systematic way.
- There is a more systematic way
  - Regular expression → NFA → DFA

Aho-Corasick Algorithm

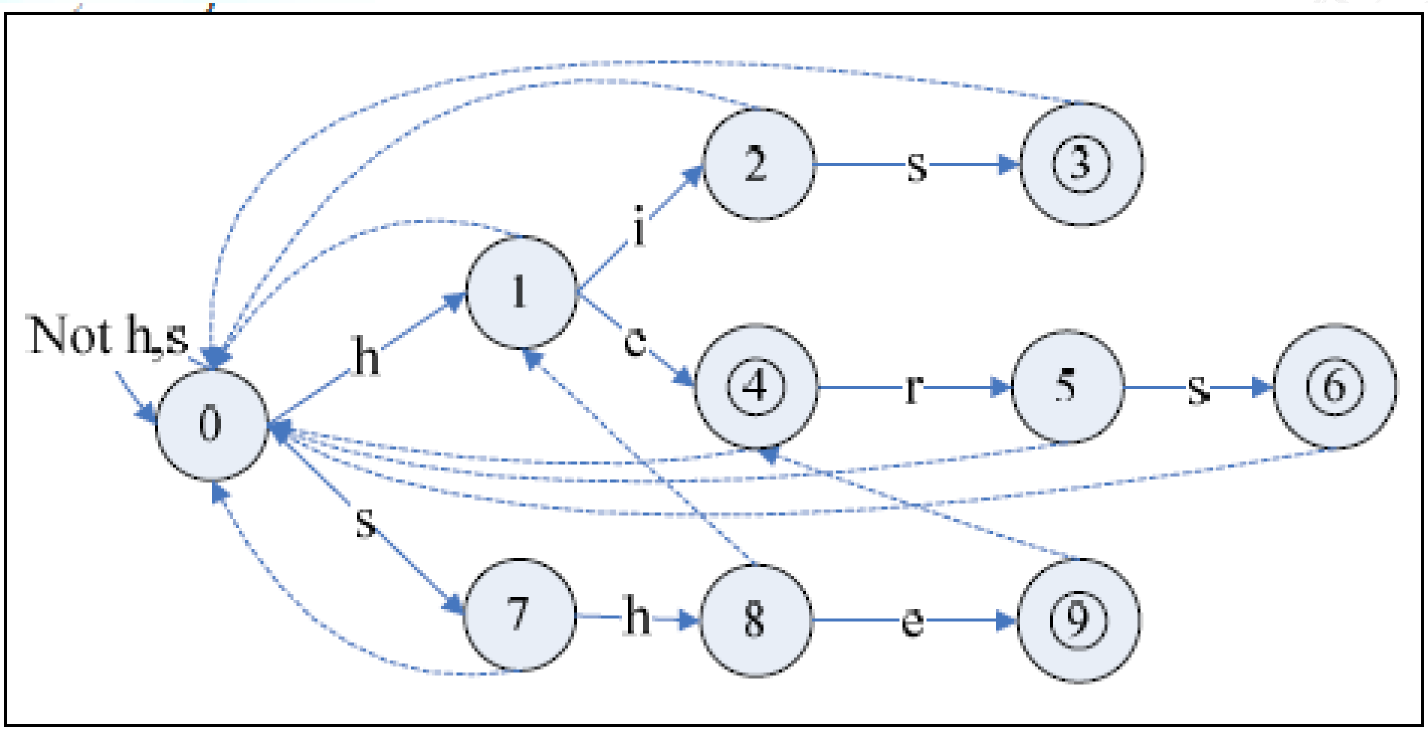


Figure 2. AC automaton for the set of keywords {he, she, his, hers}, the real line arrow represents goto function, the virtual line arrow represents failure funtion and the double circle represents output function.

Optimized AC Algorithm

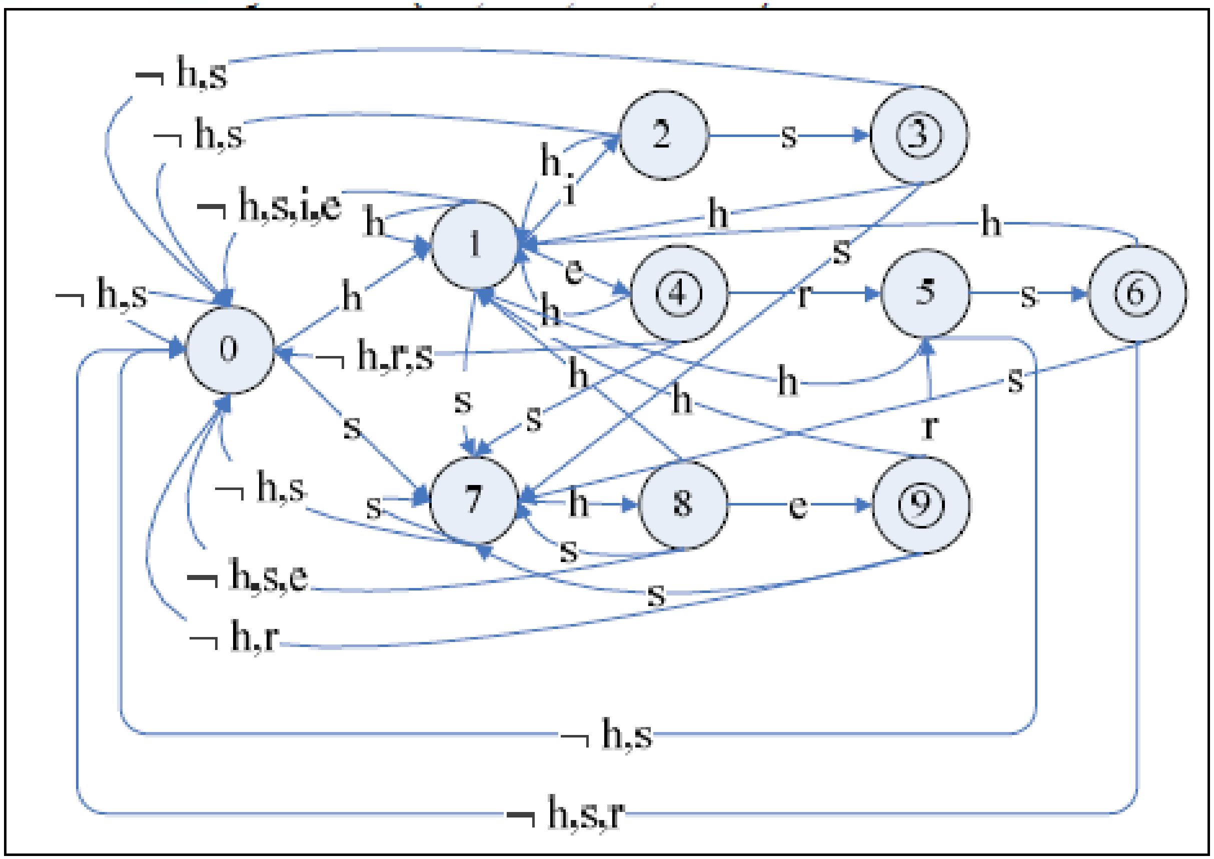


Figure 4. Optimized AC automaton for the set of keywords {he, she, his, hers}, the real line arrow represents goto function and the double circle represents output function