# Lexical Analysis (Scanning)

Chapter 2

# Lexical Analysis (Scanning)

- Basic Ideas
  - divide character stream into tokens
  - a token is the smallest logical unit in code
  - common categories:

easy to enum.

- **keywords**: “`if`”, “`for`”, “`while`”, ...
- **special symbols**: “`+`”, “`-`”, “`=`”, “`[`”, ...

hard to enum.

- **number**: “`4`”, “`23`”, “`6.63`”, “001”, ...
- **ID**: “`a`”, “`abs`”, “`sum`”, ...

**...**

# Scanning

- Token Categories
  - common categories: keywords, special symbols, number, and ID.
  - e.g.,

```
int bigger(int a, int b)
{
  int c = 0;
  if (a > b)
    c = a;
  else
    c = b;
  return c;
}
```

# Scanning

- Define Tokens
  - define different kinds of tokens in enum

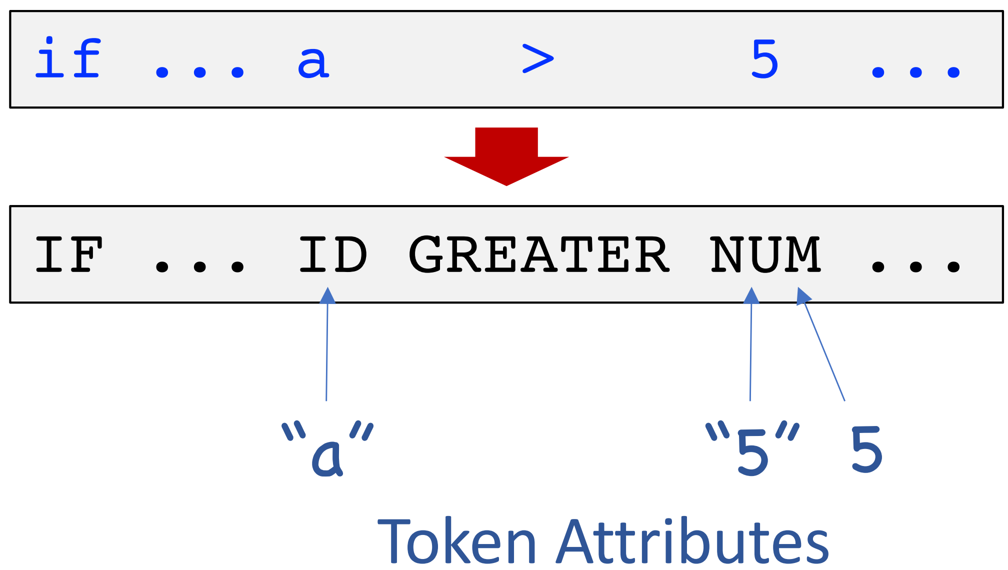
```
typedef enum
{
   IF,    // "if"
   ELSE,  // "else"
   PLUS,  // "+"
   NUM,   // "23"
   ID,    // "a"
   ...
} TokenType;
```

"Lexemes"

# Scanning

- Token Attributes
  - A token may carry attributes (e.g., *stringval*, *numberval*, ...)

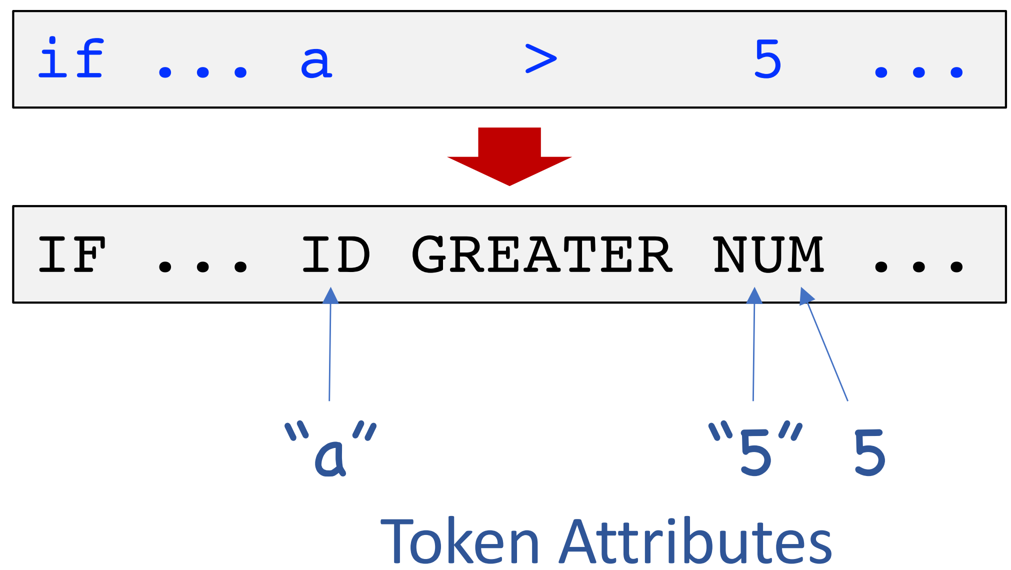
```
typedef enum
{
  IF,   // “if”
  ELSE, // “else”
  PLUS, // “+”
  NUM,  // “23”
  ID,   // “a”
  ...
} TokenType;
```



# Scanning

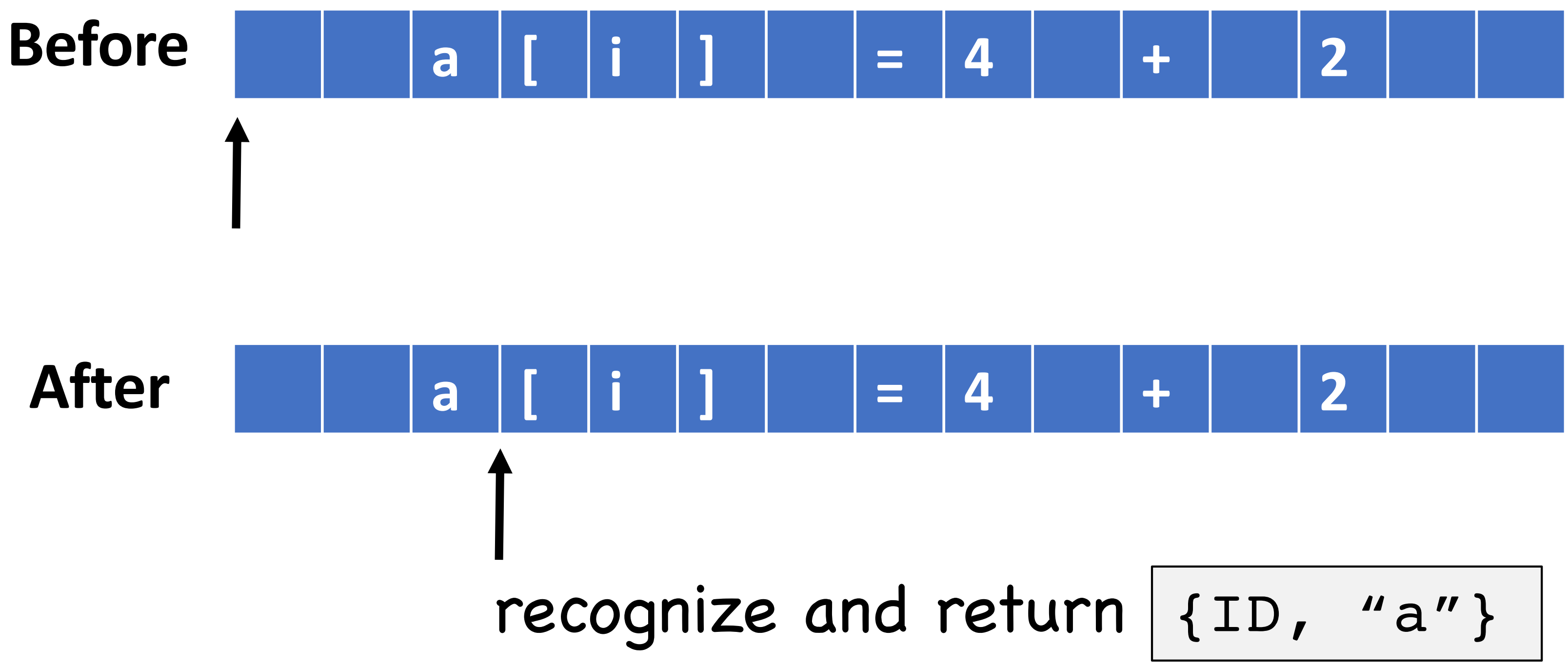
- Token Attributes
  - A token may carry attributes (e.g., *stringval*, *numberval*, ...)

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



# Scanning

- getToken()
  - scanner is often driven by the parser



# Regular Expressions

# Regular Expressions

- Basics
  - A regex $r$ represents a pattern of strings, where
  - the set of strings is called regular language $L(r)$
  - the character set is called alphabet $\Sigma$
  - Given an alphabet $\Sigma$, we can construct regex $r$:

| | | |
|---|---|---|
| if $r$ = a, a in $\Sigma$ | $L(r) = \{a\}$ | |
| if $r = \phi$ | $L(r) = \{ \}$ | empty set |
| if $r = \varepsilon$ | $L(r) = \{\varepsilon\}$ | a set w/ an empty string |

# Regular Expressions

- Operations
  - alternation “a|b”
  - concatenation “ab”
  - repetition “a*”

given regex r and s, $L(r|s) = L(r) \cup L(s)$

given regex r and s, $L(rs) = L(r)L(s)$

given regex r, $L(r^*) = \{\varepsilon\} \cup L(r) \cup L(rr) \cup L(rrr) \ldots$

# Regular Expressions

- Examples
  - What is the language of (a|b)* ?

{ε, a, b, aa, ab, ba, bb, aaa, ...}

  - What is the language of a|b* ?

{ε, a, b, bb, bbb, bbbb, ...}

Precedence: repetition > concatenation > alternation

# Regular Expressions

- Names
  - As a notational simplification

```
(0|1|2|…|9)  (0|1|2|…|9)*
```

```
digit = 0|1|2|…|9
numseq = digit digit*
```

# Regular Expressions

- Extended Regex
  - one or more repetitions `a+` = `aa*`
  - any character `.b` = `(a|b|c)b` if Σ = {a,b,c}
  - a range of characters

    `[abc]` or `[a-c]` = `(a|b|c)`

    `[acd]` = `(a|c|d)`
  - not `~(a|b)` or `[^ab]` = `c` if Σ = {a,b,c}
  - optional subexpressions

    `(a|b)?c = ac|bc|c`

# Regular Expressions

- Exercise
  - What is the regex for US zip code?

{ 92521-4120, 92508, …}

**digit = [0-9]**
**zip = digit{5}-digit{4}**

  - What is the regex for any int between 2 and 36?

**digit = [0-9]**
**zip =  [2-9] | [12]digit | 3[0-6]**

# Regular Expression

- Exercise
  - Rewrite the regex with only three core operators (concatenation/alternation/repetition)

    **(x+y)? . [^x-y]** assume $\Sigma$ = `{x,y,z}`

    **xx*y (x|y|z) z | (x|y|z)z**

  - Write the regex for strings in C programs (assume escape character \ is not allowed)

    E.g. `x = “hello, world!”;`

    **`“[^”\\]”`**

# Token Specification

# Token Specification

- Specify tokens with regex
  - Given the complexity, regex is perfect for this purpose

easy to enum.

**keywords**: "`if`", "`for`", "`while`", …

**special symbols**: "`+`", "`-`", "`=`", "`[`", …

hard to enum.

**number**: "`4`", "`23`", "`6.63`", "001", …

**ID**: "`a`", "`abs`", "`sum`", …

**…**

# Token Specification

- Numbers
  - sequence of digits "23"
  - signed numbers "-12", "+17"
  - decimal numbers "1.24"
  - scientific numbers "2.74E+2"

```
nat = [0-9]+
signedNat = [+-]? nat
decimalNum = signedNat \. nat
scientificNum = signedNat \. nat E signedNat
number = signedNat (\. nat)? (E signedNat)?
```

# Token Specification

- Reserved Words

```
reserved = if | while | do | ...
```

- Identifiers
  - Begins with a letter; contains only letters and digits

```
letter = [a-zA-Z]

digits = [0-9]

identifier = letter(letter|digit)*
```

# Example -- Identifier

Valid Identifiers

- Composed of letters, digits, and underscores
- Cannot end in an underscore
- Cannot contain two underscores in a row

A1BC_3A_B5

# Token Specification

- Comments
  - Typically are “skipped” during scanning
  - Still need to be recognized so they can be skipped

```
{this is a Pascal comment}          \{[^\}]*\}

/* this is a C/C++ comment */

// this is a C/C++ comment          //[^\n]*
```

# Comment /* ………. */

/* this is the end **** not yet **** not yet *********/

**/ * [^ *]$^*$ ( *+ [^ /] [^ *]$^*$ )$^*$ *+ /**

# Token Specification

- Ambiguity
  - Token specification may contain ambiguities
  - Existing multiple ways to interpret the same substring

| | |
|---|---|
| "if" | IF |
| "if" | identifier |

| | |
|---|---|
| "<>" | not equal |
| "<" ">" | less than, greater than |

Keyword is preferred!

Longer token is preferred!

*(principle of **longest** substring)*

# Token Specification

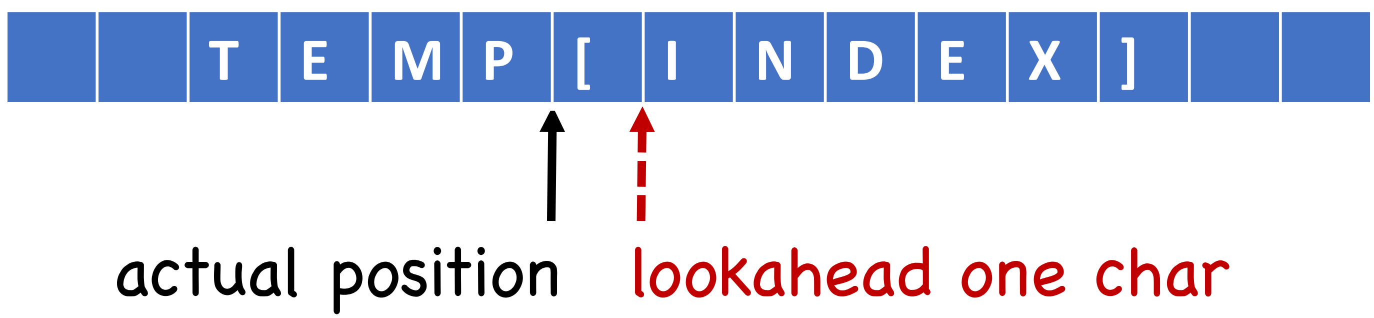
- Token Delimiters
  - Characters that imply a longer string cannot be a token
  - White spaces are delimiters
  - Comments could also be delimiters

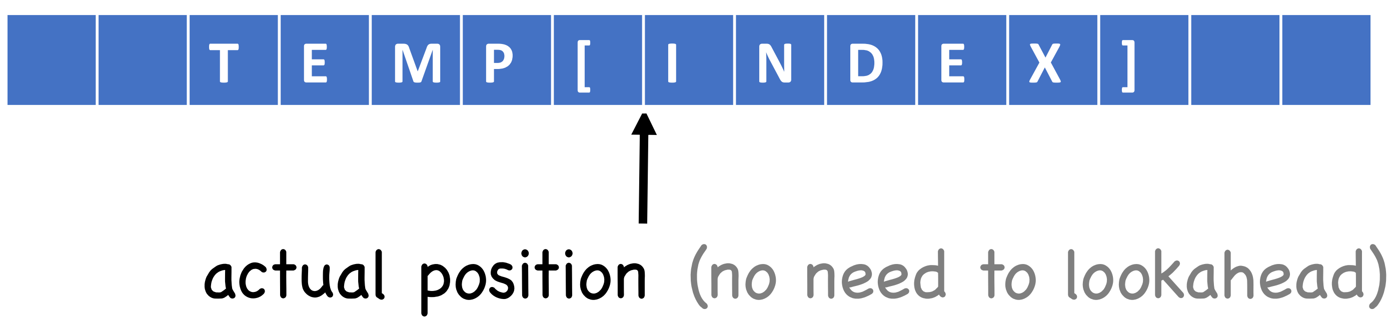
| | |
|---|---|
| *"xtemp=ytemp"* | *"="* is not part of any token |
| *"int x"* | *blank/newline/tab* are neither |
| *"do//if"* | *comments* are neither |

$$whitespace=(newline \mid blank \mid tab \mid comment)+$$

# Token Specification

- Token Delimiters
  - A delimiter ends a token, but not part of that token
  - Should not be consumed, but just be examined - Lookahead



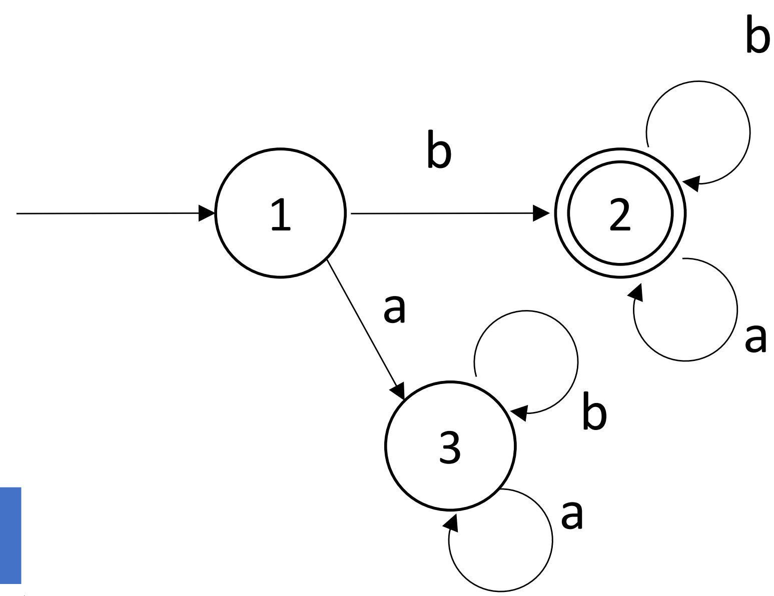
# Finite Automata

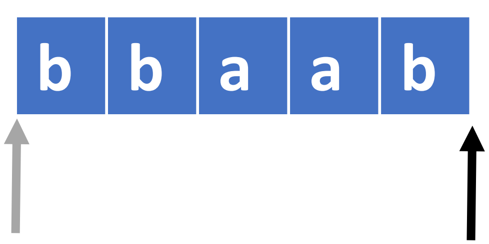
# Finite Automata

- Equivalence
  - a regex specifies a regular language
  - FA accepts a regular language
  - regex ←→ FA

`r = b(b|a)*`

`Σ = {a,b}`



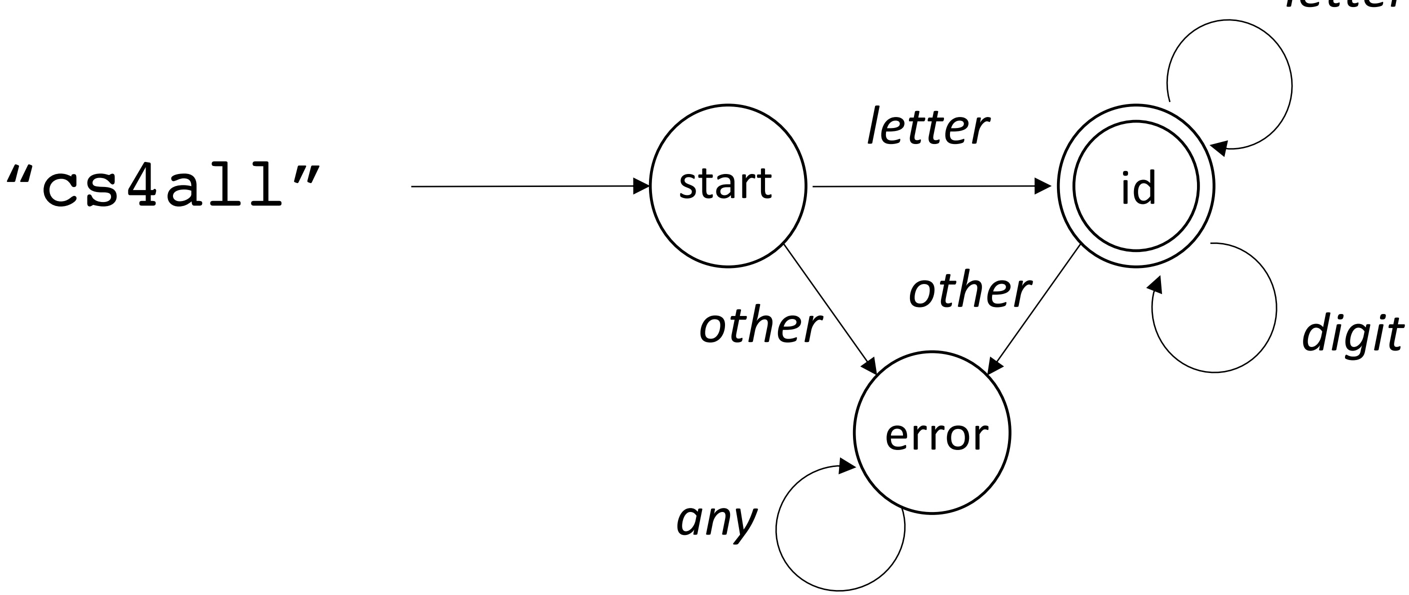




1 2 2 2 2 **2** **accept!**

# Finite Automata

- Extensions and Simplification
  - name transitions w/ regex names (also, *other* and *any*)
  - error state

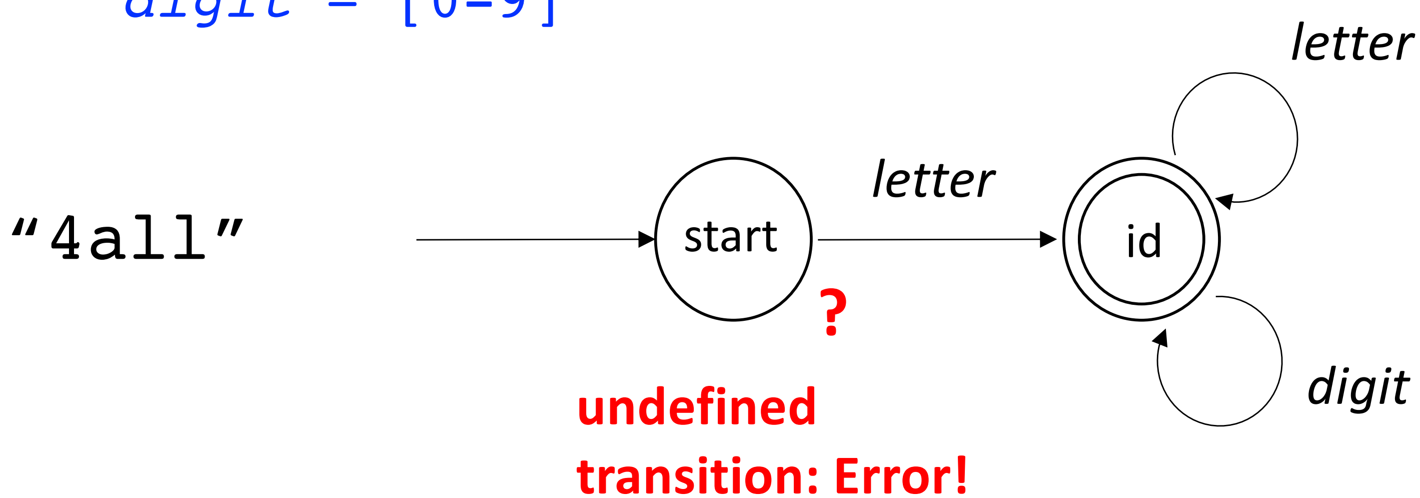
```
identifier = letter(letter|digit)*
letter = [a-zA-Z]
digit = [0-9]
```



# Finite Automata

- Extensions and Simplification
  - name transitions w/ regex names (also, *other* and *any*)
  - error state is often omitted

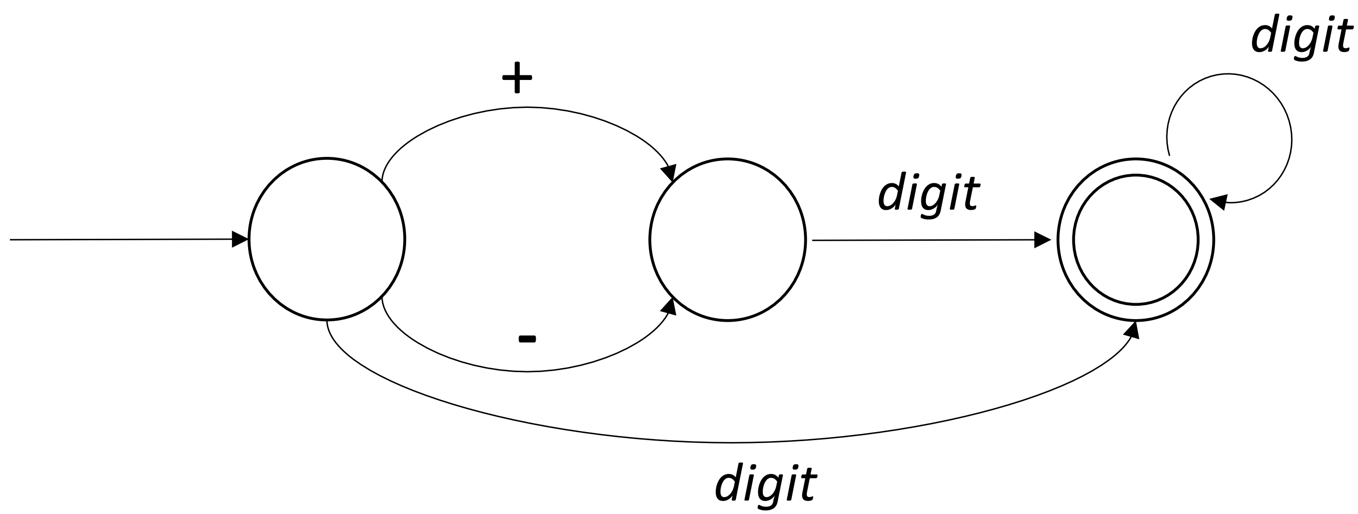
```
identifier = letter(letter|digit)*
letter = [a-zA-Z]
digit = [0-9]
```



# Finite Automata

- Exercise
  - FA for recognizing signed numbers

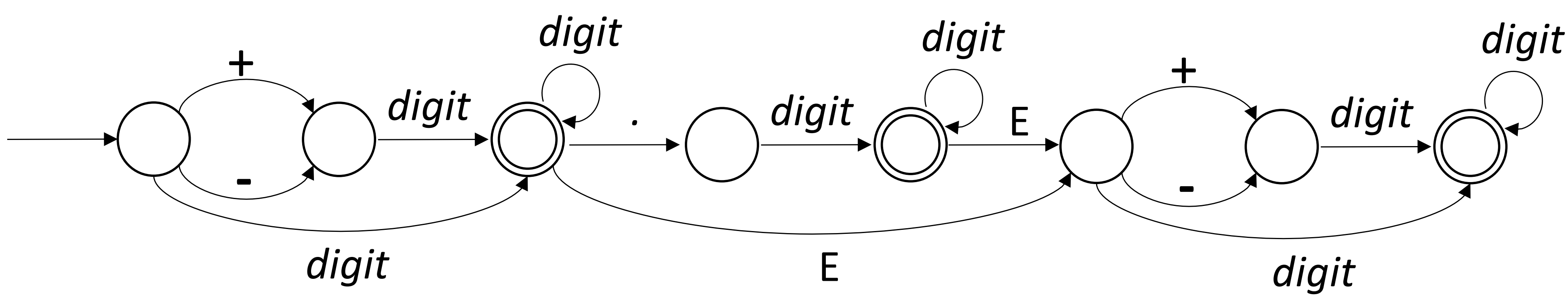
```
digit = [0-9]
nat = digit+
signedNat = (+|-)? nat
```



# Finite Automata

- Exercise
  - FA for recognizing numbers

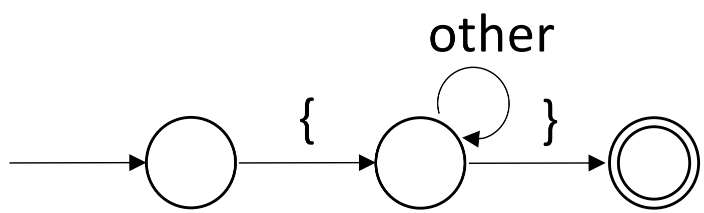
```
digit = [0-9]
nat = digit+
signedNat = (+|-)? nat
number = signedNat ("." nat)? (E signedNat)?
```



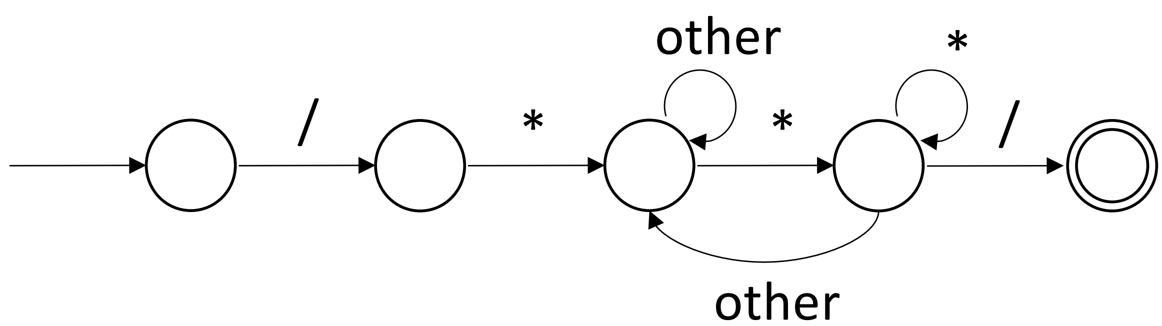
# Finite Automata

- Exercise
  - FA for recognizing comments

`\{[^\}]*\}`

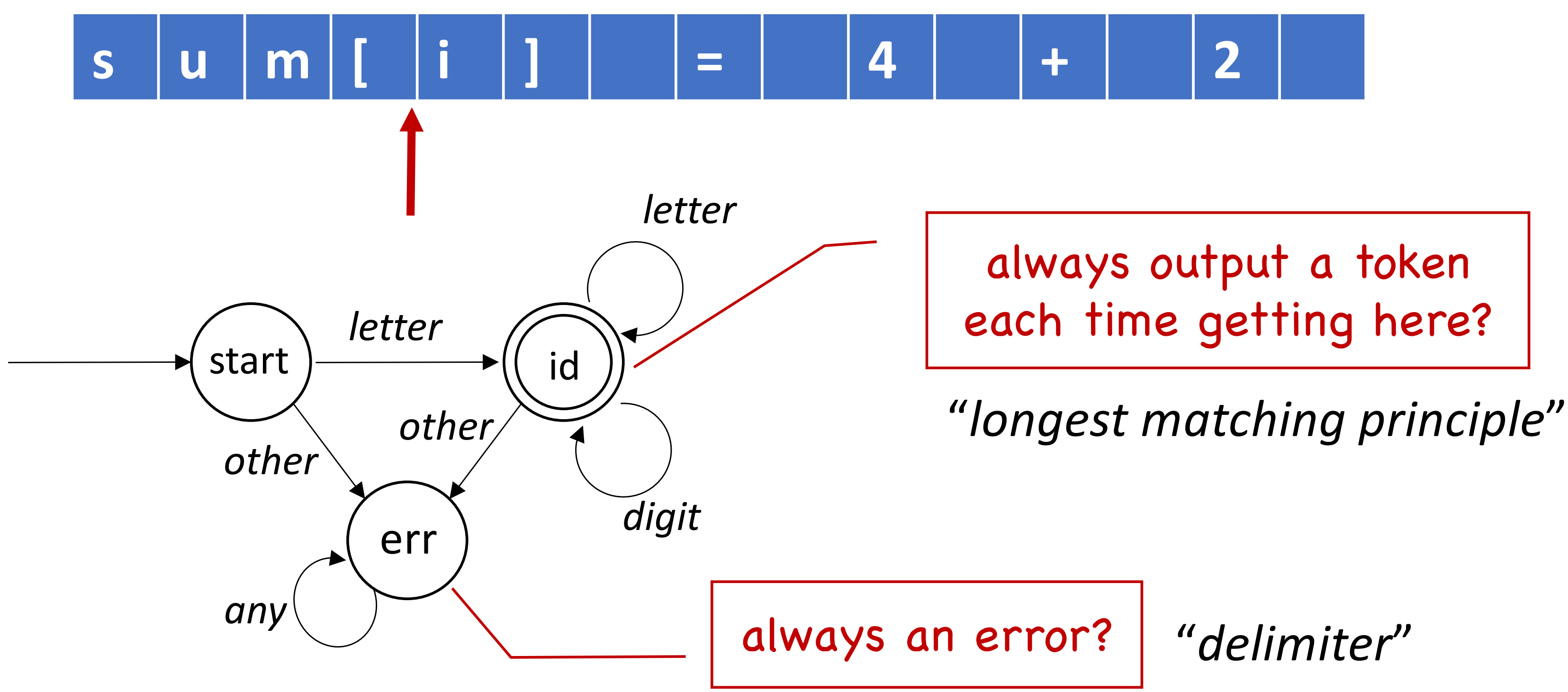



`/* hello */`



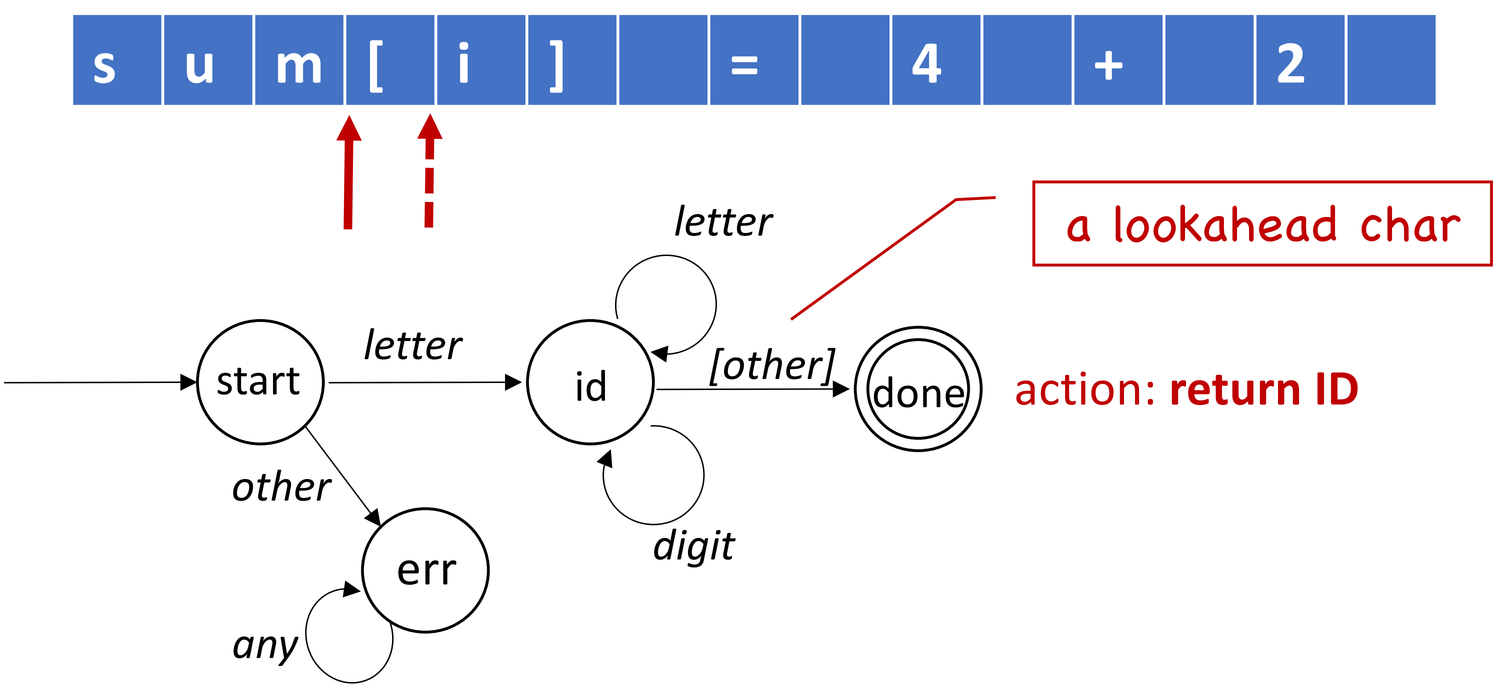
# Finite Automata

- FA Actions
  - ”normal” state: copy a character to a token buffer
  - accept state: return a token & go back to intial state
  - error state: generate an error



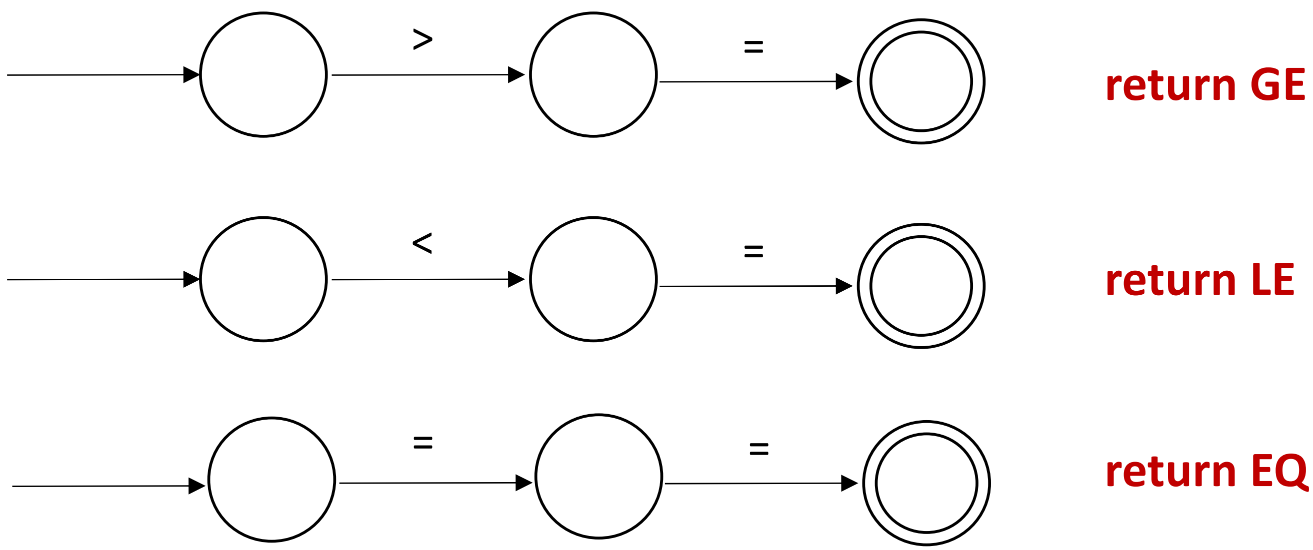
# Finite Automata

- Adjusted FA
  - "error" state becomes an accept state
  - which has no further transition edges
  - *[other]* is from lookahead



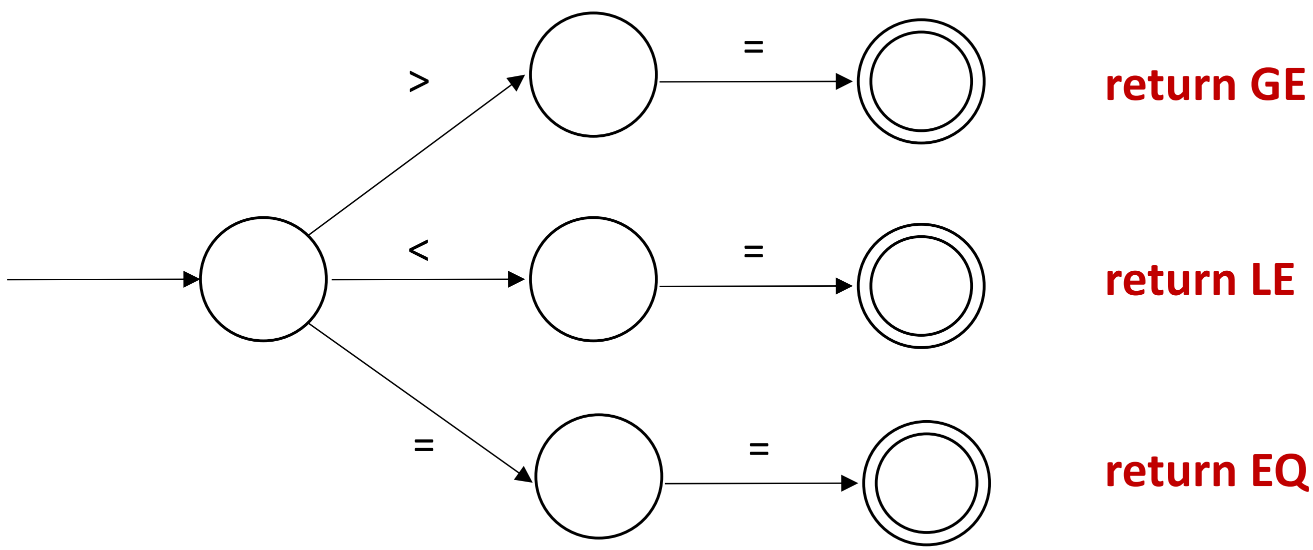
# Finite Automata

- Recognize Multiple Types of Tokens
  - cannot track the states of all different FAs - too expensive!
  - **solution**: merge different FAs



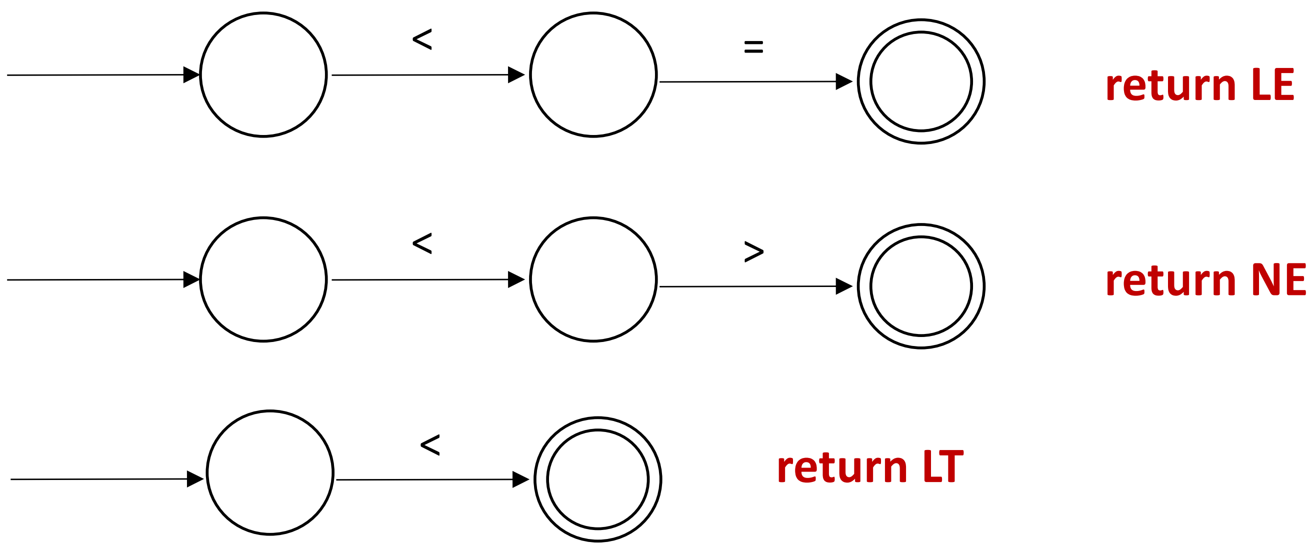
# Finite Automata

- Recognize Multiple Tokens
  - tokens starting with different characters
  - easier to merge: simply combine their starting states



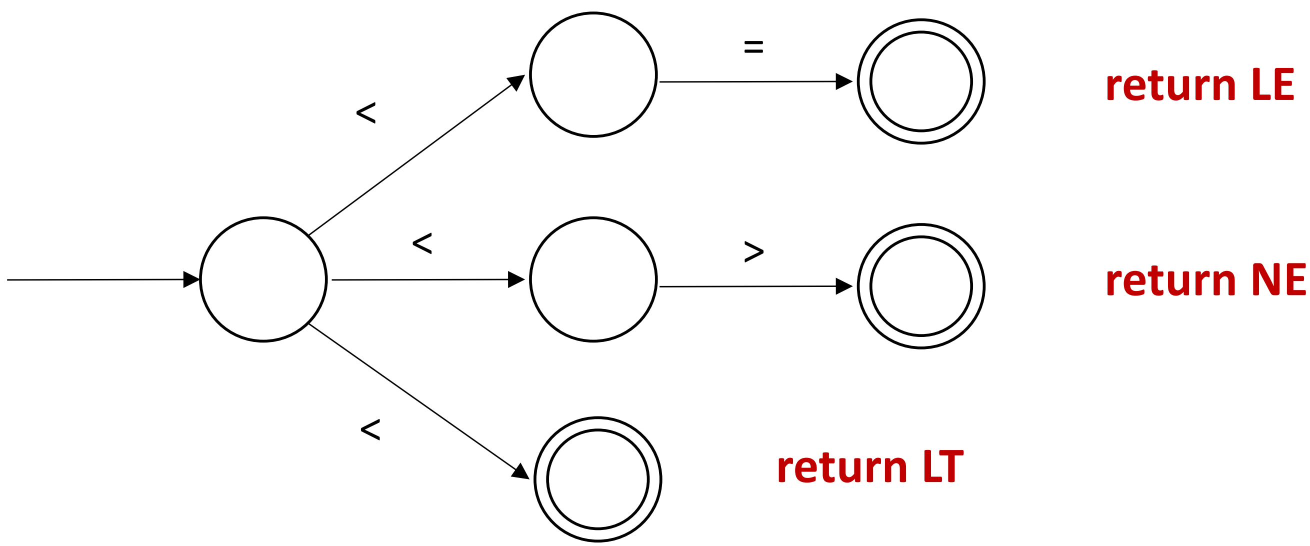
# Finite Automata

- Recognize Multiple Tokens
  - tokens starting with the same character



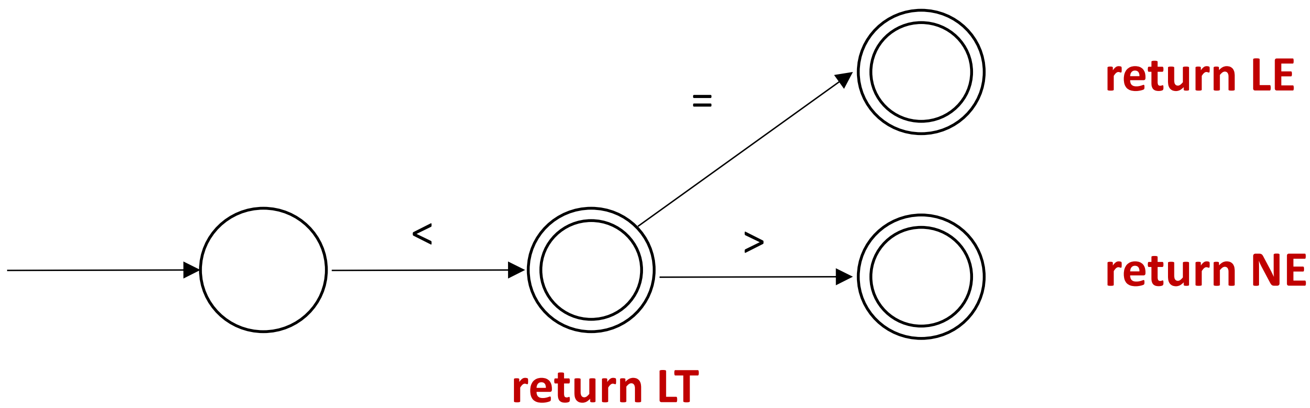
# Finite Automata

- Recognize Multiple Tokens
  - it becomes an NFA (*non-deterministic finite automaton*)
  - expensive to run an NFA!



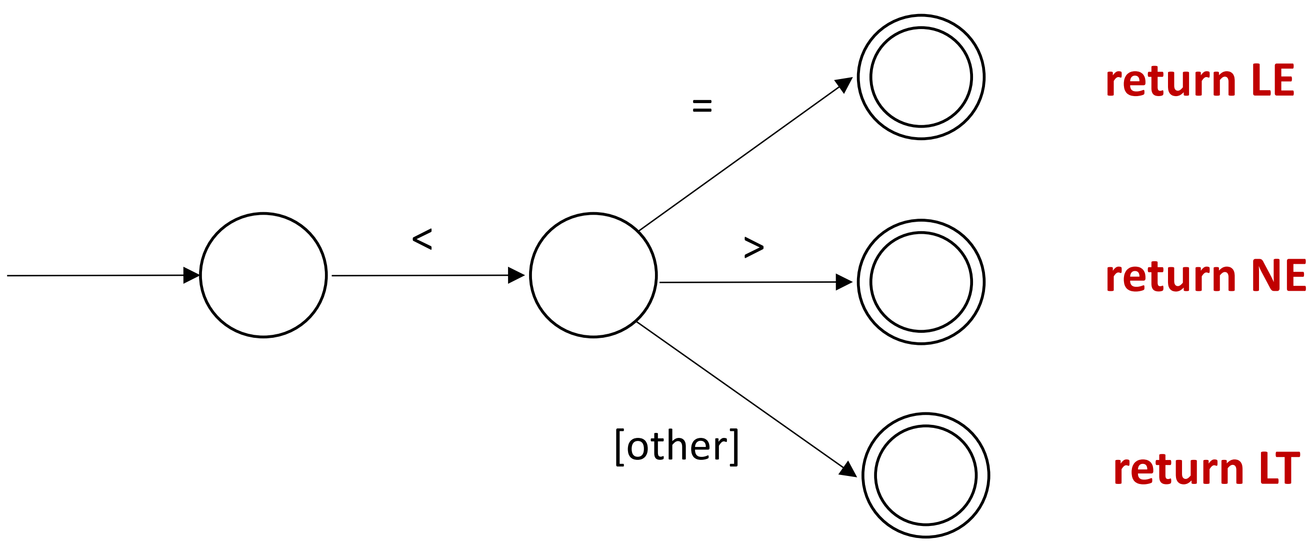
# Finite Automata

- Recognize Multiple Tokens
  - it becomes an NFA (*non-deterministic finite automaton*)
  - NFA → DFA (deterministic ...)



# Finite Automata

- Recognize Multiple Tokens
  - it becomes an NFA (*non-deterministic finite automaton*)
  - NFA → DFA (deterministic ...)
  - DFA adjustment

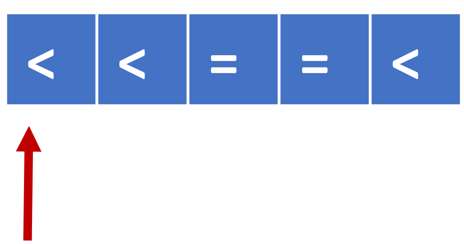


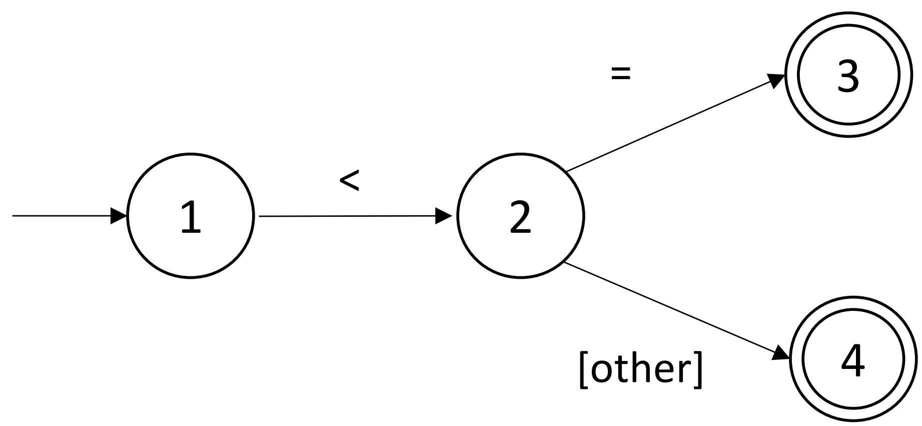
# Finite Automata

- Example:
  - Draw the FA for recognizing the following two kinds of tokens in a token string:

**LT : <**

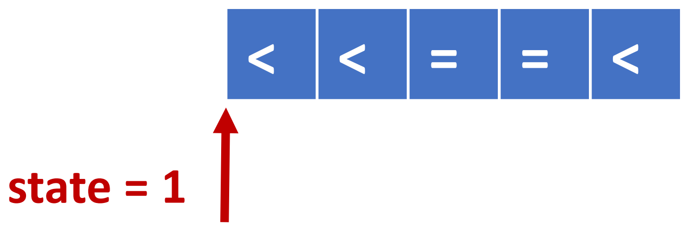
**LE : <=**

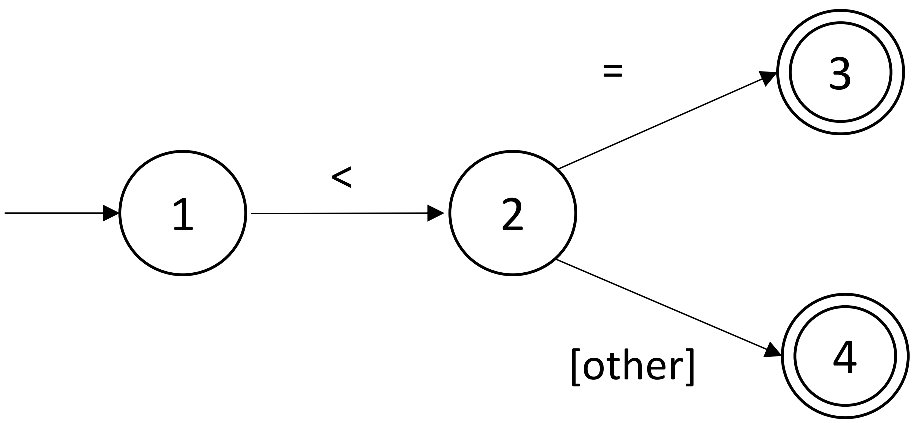


# Finite Automata

- Implementation
  - hard-coded

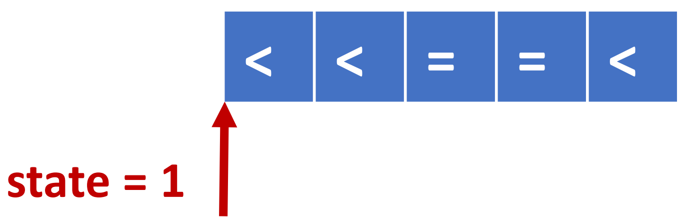


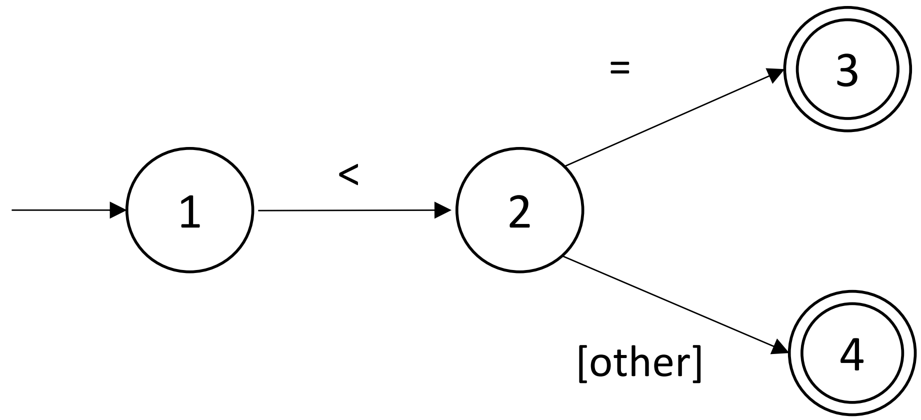




```
state = 1
while (!EOF)
{
  switch(state)
    case 1:
      if(advance() == ‘<’)
        state = 2
      else
        error & break
    case 2:
      if(advance() == ‘=’)
        state = 3
      else
        state = 4
    case 3:
      output token LE
      state = 1
    case 4:
      output token LT
      stepback()
      state = 1
}
```

# Finite Automata

- Implementation
  - transition table




state

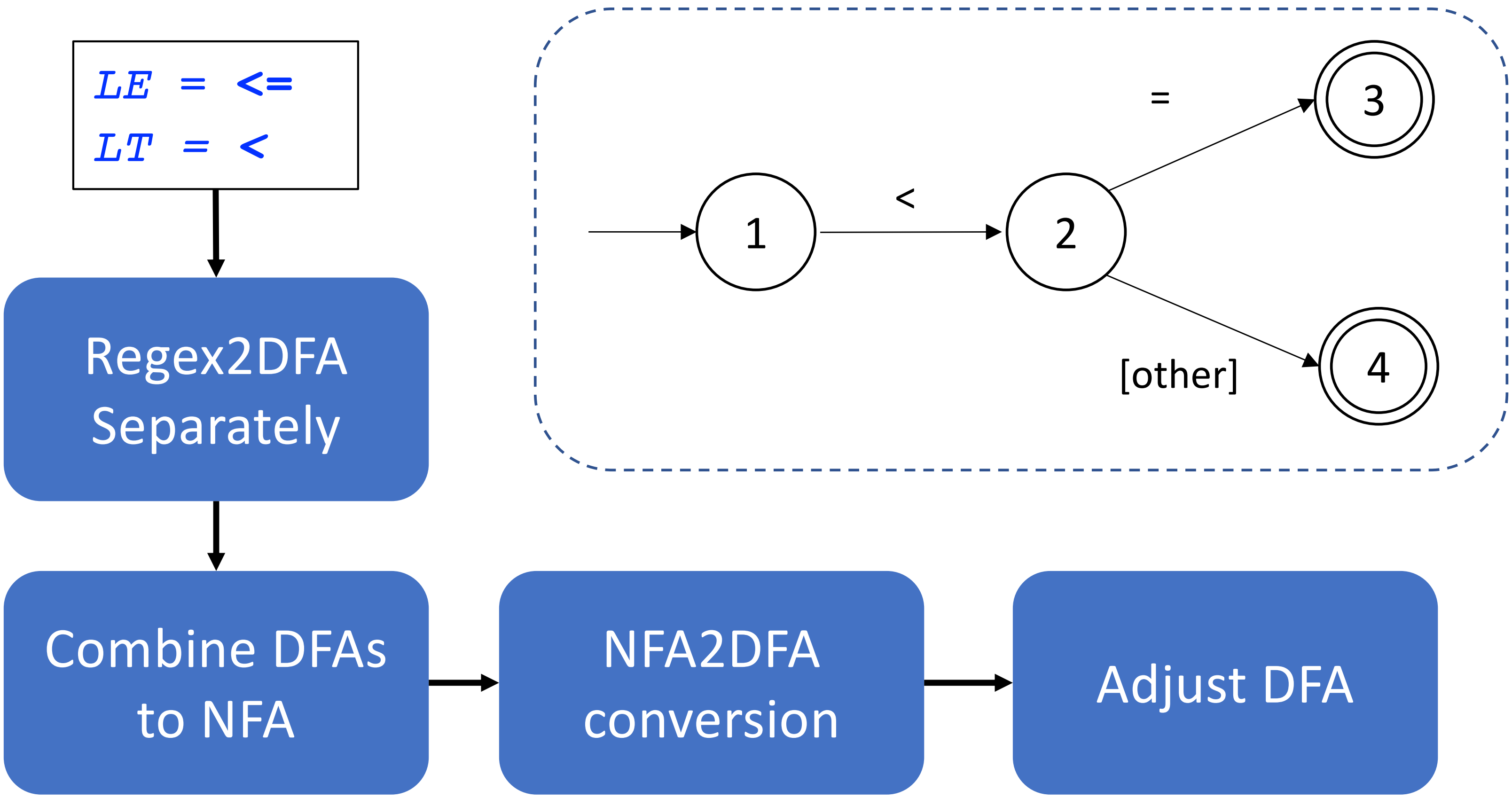
| | < | = | action |
|---|---|---|---|
| 1 | 2 | err | c = advance() |
| 2 | 4 | 3 | c = advance() |
| 3 | 1 | 1 | output LE |
| 4 | 1 | 1 | output LT; c = stepback() |
| err | - | - | print error; break |

```
state = 1
c = advance()
while (!EOF)
{
  state = Trans[state][c]
  action(state)
}
```

# Putting it All Together

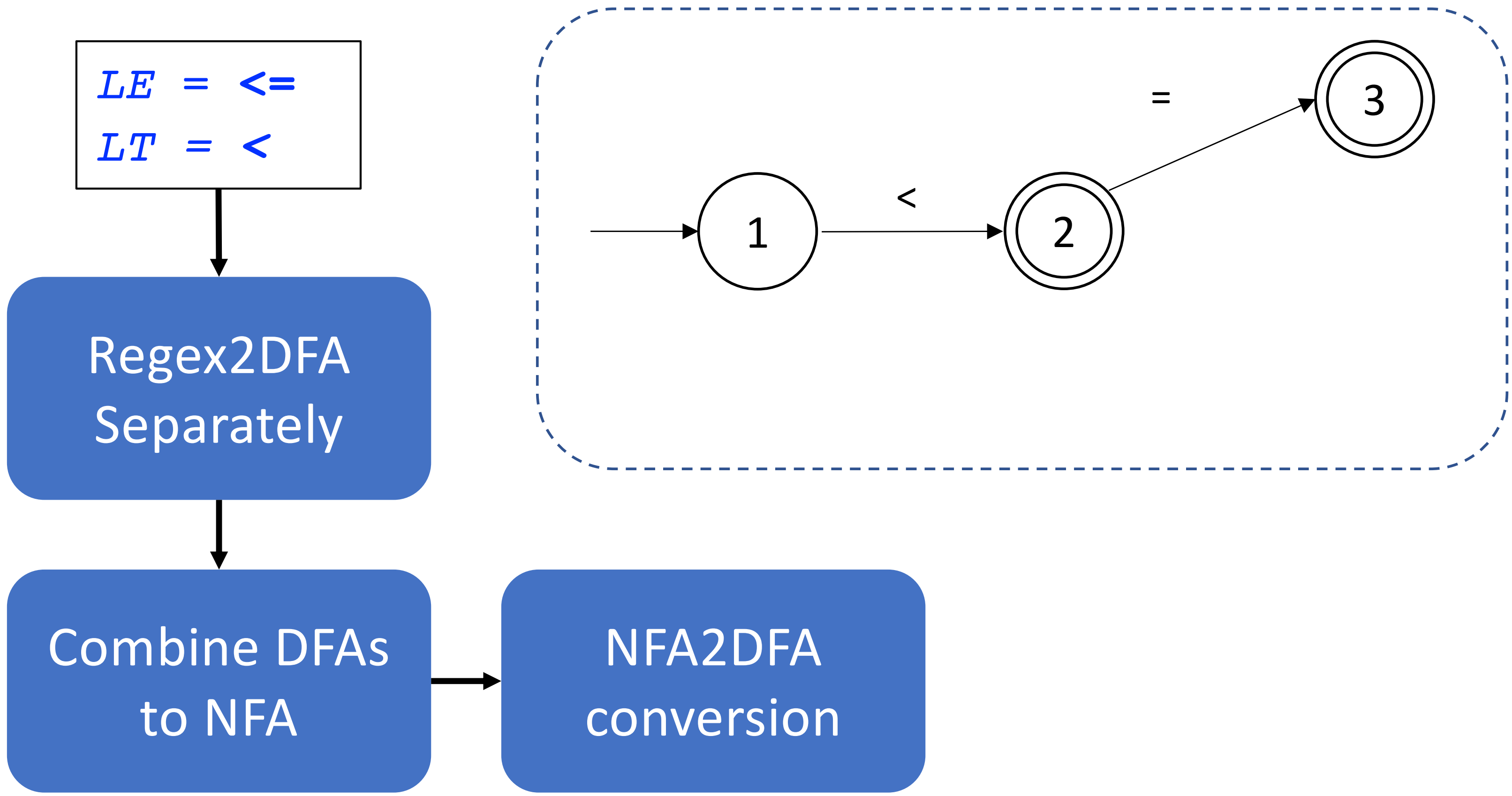
# Putting It All Together

- [Louden Ch. 2]



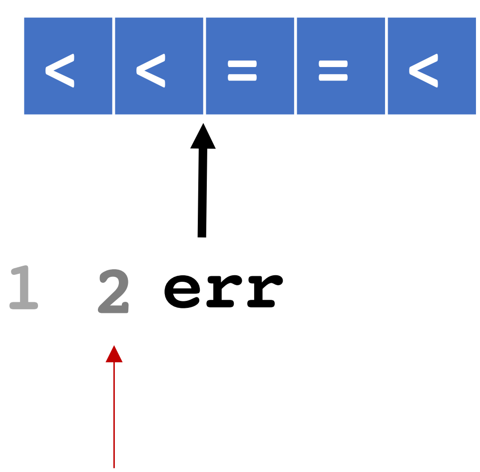
# Putting It All Together (flex)

- flex-generated scanner

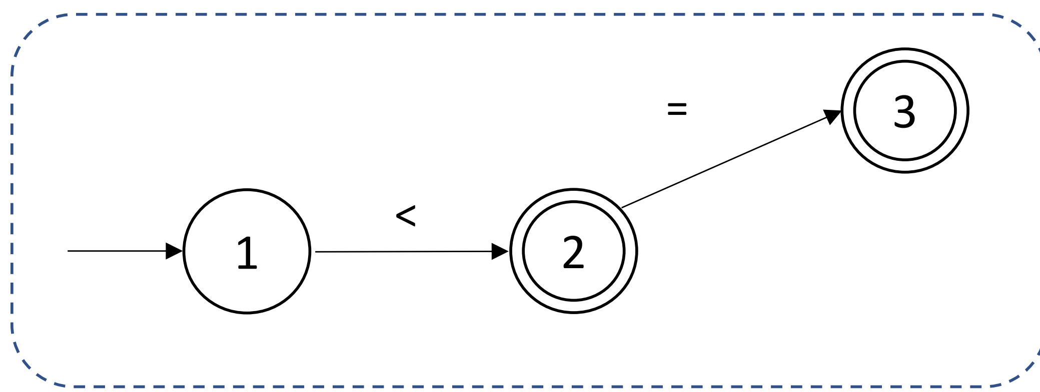


# Putting It All Together (flex)

- flex-generated scanner
  - Move forward until impossible (meeting an “error”)
  - Backtrack to find the latest accept state

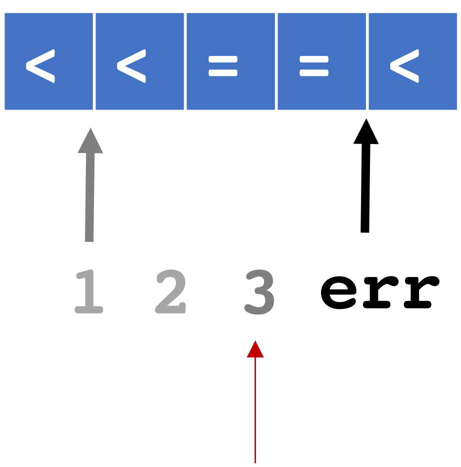



**latest accept**
**output: LT**

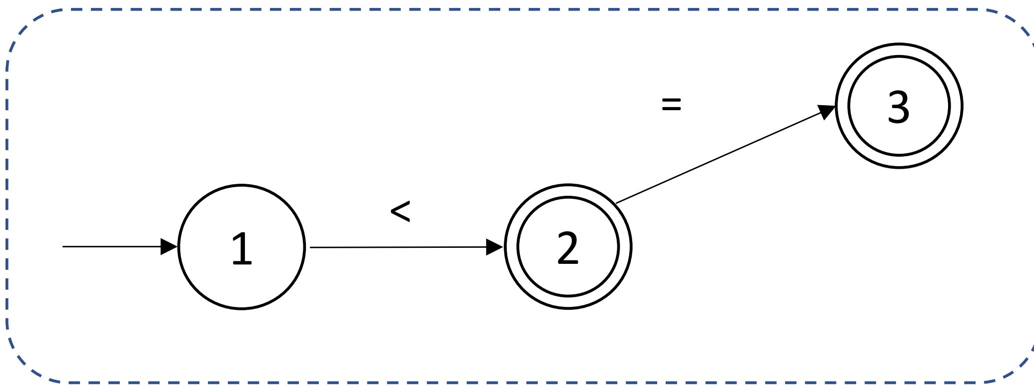


# Putting It All Together (flex)

- flex-generated scanner
  - Move forward until impossible (meeting an “error”)
  - Backtrack to find the latest accept state

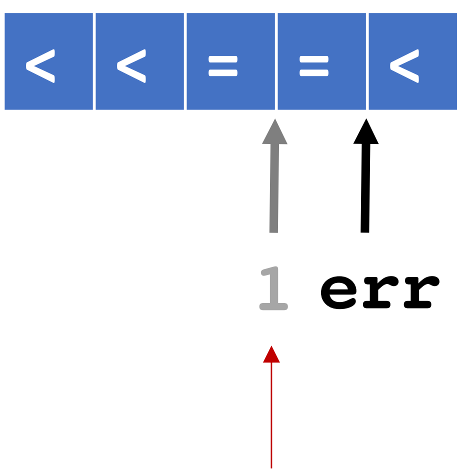


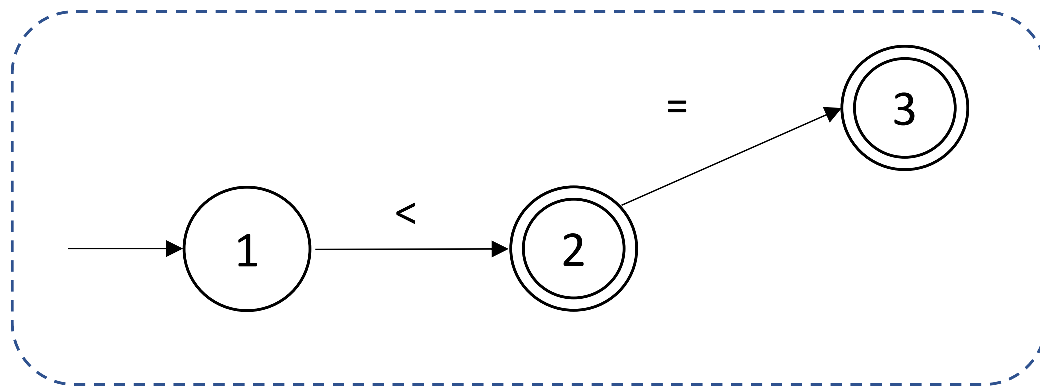

**latest accept**
**output: LE**



# Putting It All Together (flex)

- flex-generated scanner
  - Move forward until impossible (meeting an "error")
  - Backtrack to find the latest accept state




**no accept state! real error!**