Compilers

by
Marwa Yusuf

Lecture 3
Mon. 29-3-2021

Chapter 3 (3.3 to 3.6)

Lexical Analysis

Specification of Tokens

- Regular expressions are used to specify lexeme patterns.

Strings and Languages

- **Symbols**: digits, letters, punctuation.
- ***Alphabet*** Σ: any finite set of symbols.
  - $\{0, 1\}$ binary alphabet.
  - ASCII
  - Unicode: 100,000 symbols.
- ***String*** (over an alphabet): finite seq. of symbols drawn from alphabet.
- |**s**|: length of string, number of occurrences of symbols.
- ***Empty string ε****:* the zero length string.

Strings and Languages

- ***Language***: any countable set of strings over some fixed alphabet, meaning is not a condition.
  - φ, empty set, {ε} are languages.
  - All syntactically well-formed c programs.
  - All grammatically correct English sentences.
- ***Concatenation*** of x and y (xy): appending y to x.
  - $\varepsilon s = s\varepsilon = s$
- ***Exponentiation***:
  - $s^0 = \varepsilon$
  - $s^i = s^{i-1}s$
  - $s^1 = s,\ s^2 = ss,\ s^3 = sss \ldots$

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, …, Z, a, b, …, z} – D = {0, 1, …, 9}
  - Alphabet or a language of one letter

1) $L \cup D$
2) LD
3) L^4
4) L^*
5) $L(L \cup D)^*$
6) D^+

Regular Expressions

- Item 5 + underscore describes C identifiers.
- ***Regular expressions***: all the languages built by applying prev. operators on some alphabet symbols.
- **Ex:** *letter_* (*letter_* | *digit*)*
- Each regular expression r denotes a language $L(r)$, recursively from r's sub-expressions.

Regular Expressions

- **Basis:**
  - ε is a regular expression, L(ε) is {ε}
  - If *a* is a symbol in Σ then **a** is a regular expression and L(**a**) = {*a*}.
- **Induction:** suppose *r* and *s* are regular expression with *L(r)* and *L(s)* languages:
  - (r)|(s) is a regular expression denoting the language L(r) U L(s).
  - (r)(s) is a regular expression denoting the language L(r)L(s).
  - (r)* is a regular expression denoting (L(r))*.
  - (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.
- Parenthesis can be dropped given that:
  - *, concatenation, | with this precedence order are left associative.

Regular Expressions

- $\Sigma = \{a, b\}$

  1) **a|b** denotes ……….
  2) **(a|b)(a|b)** denotes ……….
  3) **a*** denotes ……….
  4) **(a|b)*** denotes
     ………………………………………………...
  5) **a|a*****b** denotes ……….

Regular Expressions

- $\Sigma = \{a, b\}$
  1) **a|b** denotes {a, b}
  2) **(a|b)(a|b)** denotes {aa, ab, ba, bb} = aa|ab|ba|bb
  3) **a*** denotes zero or more a = {ε, a, aa, …}
  4) **(a|b)*** denotes all strings of zero or more a or b = {ε, a, b, aa, ab, bb, ba, aaa, …} = (a*b*)*
  5) **a|a*b** denotes {a, b, ab, aab, aaab, …} = (a)|(a*b)

Regular Expressions

- ***Regular set***: a language that can be defined by a regular expression.
- If *r* and *s* denote the same regular set; they are *equivalent*.

<table>
<tr><th>LAW</th><th>DESCRIPTION</th></tr>
<tr><td>r|s = s|r</td><td>| is commutative</td></tr>
<tr><td>r|(s|t) = (r|s)|t</td><td>| is associative</td></tr>
<tr><td>r(st) = (rs)t</td><td>Concatenation is associative</td></tr>
<tr><td>r(s|t) = rs|rt; (s|t)r = sr|tr</td><td>Concatenation distributes over |</td></tr>
<tr><td>εr = rε = r</td><td>ε is the identity for concatenation</td></tr>
<tr><td>r* = (r|ε)*</td><td>ε is guaranteed in a closure</td></tr>
<tr><td>r** = r*</td><td>* is idempotent</td></tr>
</table>

Regular Definitions

- Give names to some regular expressions and use them later as if symbols.
- A regular definition:

  $d_1 \to r_1$

  $d_2 \to r_2$

  …

  $d_n \to r_n$

  where:

  1) each d_i is a new symbol not in Σ and not the same as any other d;

  2) each r_i is a regular expression over the alphabet $\Sigma \cup \{d_1, d_2, \ldots, d_{i-1}\}$.

Regular Definitions

- Ex: For C identifiers:

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

  $digit \rightarrow$ 0 | 1 | … | 9

  $id \rightarrow letter\_ \; (letter\_ \mid digit)^*$

- Ex: For unsigned numbers 123, 0.0334, 5.7, 1.87E-3

  $digit \rightarrow$

  $digits \rightarrow$

  $optionalFraction \rightarrow$

  $optionalExponent \rightarrow$

  $number \rightarrow$

Regular Definitions

- Ex: For C identifiers:

  $letter\_ \rightarrow \mathrm{A} \mid \mathrm{B} \mid \ldots \mid \mathrm{Z} \mid \mathrm{a} \mid \mathrm{b} \mid \ldots \mid \mathrm{z} \mid \_$

  $digit \rightarrow 0 \mid 1 \mid \ldots \mid 9$

  $id \rightarrow letter\_ \, (\, letter\_ \mid digit \,)^*$

- Ex: For unsigned numbers 123, 0.0334, 5.7, 1.87E-3

  $digit \rightarrow 0 \mid 1 \mid \ldots \mid 9$

  $digits \rightarrow digit\ digit^*$

  $optionalFraction \rightarrow .\ digits \mid \varepsilon$

  $optionalExponent \rightarrow (\ \mathrm{E}\ (\ + \mid - \mid \varepsilon\ )\ digits\ ) \mid \varepsilon$

  $number \rightarrow digits\ optionalFraction\ optionalExponent$

Extensions of Regular Expressions

- One or more instances $(r)^+$
  - the same precedence and associativity as *
  - $r^* = r^+|\varepsilon$, $r+ = rr^* = r^*r$
- Zero or one instance $r? = r|\varepsilon$ $[L(r?) = L(r) \cup L(\varepsilon)]$
  - the same precedence and associativity as *
- Character classes
  - $a_1|a_2|\ldots|a_n = [a_1a_2\ldots a_n]$
  - if a's form a logical sequence, like all lowercase letters, all digits, can be replaced by a_1-a_n
  - $[abc] = a|b|c$ $\quad$ $[a\text{-}z] = a|b|\ldots|z$

Extensions of Regular Expressions

- Ex: For C identifiers:

  $letter\_ \rightarrow \mathrm{A} \mid \mathrm{B} \mid \ldots \mid \mathrm{Z} \mid \mathrm{a} \mid \mathrm{b} \mid \ldots \mid \mathrm{z} \mid \_$

  $digit \rightarrow 0 \mid 1 \mid \ldots \mid 9$

  $id \rightarrow letter\_ \, (\, letter\_ \mid digit \,)^*$

- becomes

  $letter\_ \rightarrow$

  $digit \rightarrow$

  $id \rightarrow$

Extensions of Regular Expressions

- Ex: For C identifiers:

  $letter\_ \rightarrow \mathrm{A} \mid \mathrm{B} \mid \ldots \mid \mathrm{Z} \mid \mathrm{a} \mid \mathrm{b} \mid \ldots \mid \mathrm{z} \mid \_$

  $digit \rightarrow 0 \mid 1 \mid \ldots \mid 9$

  $id \rightarrow letter\_ \, (letter\_ \mid digit)^*$

- becomes

  $letter\_ \rightarrow [\mathrm{A\text{-}Za\text{-}z}]$

  $digit \rightarrow [0\text{-}9]$

  $id \rightarrow letter\_ \, (letter\_ \mid digit)^*$

Regular Definitions

- Ex: For unsigned numbers 123, 0.0334, 5.7, 1.87E-3

  digit → 0 | 1 | … | 9

  digits → *digit digit**

  optionalFraction → . *digit* | ε

  optionalExponent → (E (+ | - | ε) *digits*) | ε

  number → *digits optionalFraction optionalExponent*

- becomes

  digit →

  digits →

  optionalFraction →

  optionalExponent →

  number →

Regular Definitions

- Ex: For unsigned numbers 123, 0.0334, 5.7, 1.87E-3

  digit $\rightarrow$ 0 | 1 | … | 9

  digits $\rightarrow$ *digit digit**

  optionalFraction $\rightarrow$. *digit* | ε

  optionalExponent $\rightarrow$ (E (+ | - | ε) *digits*) | ε

  number $\rightarrow$ *digits optionalFraction optionalExponent*

- becomes

  digit $\rightarrow$ [0-9]

  digits $\rightarrow$ *digit*$^+$

  ~~*optionalFraction* → . *digit* | ε~~

  ~~*optionalExponent* → (E (+ | - | ε) *digits*) | ε~~

  number $\rightarrow$ *digits* (. *digits*)? (E [+-]? *digits*)?

Recognition of Tokens

- Build a pattern matching code.
- **Ex:** Grammar of **if statement** like Pascal (**then** is explicit, = and <> are for comparison).

$$
\begin{aligned}
stmt &\rightarrow \textbf{if}\ expr\ \textbf{then}\ stmt \\
&\quad |\ \textbf{if}\ expr\ \textbf{then}\ stmt\ \textbf{else}\ stmt \\
&\quad |\ \varepsilon \\
expr &\rightarrow term\ \textbf{relop}\ term \\
&\quad |\ term \\
term &\rightarrow \textbf{id} \\
&\quad |\ \textbf{number}
\end{aligned}
$$

Recognition of Tokens

- The terminals are: **if, then, else, relop, number** and **id**

  $if \rightarrow \text{if}$

  $then \rightarrow \text{then}$

  $else \rightarrow \text{else}$

  $relop \rightarrow \; < \mid > \mid <= \mid >= \mid = \mid <>$

  $digit \rightarrow [0\text{-}9]$

  $digits \rightarrow digit^{+}$

  $number \rightarrow digits\ (\ .\ digits)?\ (\ \text{E}\ [+-]?\ digits)?$

  $letter \rightarrow [\text{A-Za-z}]$

  $id \rightarrow letter\ (\ letter \mid digit\ )^{*}$

Recognition of Tokens

- For stripping whitespace:

  $ws \rightarrow$ (**blank** | **tab** | **newline**)+

- **blank**, **tab** and **newline** are symbols for the ASCII characters.
- *ws* is a token that is NOT returned to the parser.

Recognition of Tokens

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**	

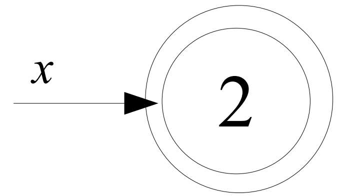
Transition Diagrams

- Convert patterns into transition diagrams (Intermediate step).
- For now, manually.
- ***States***: (circles or nodes) a condition during scanning.
- ***Edges***: directed between states, labeled by symbol(s), *forward* pointer advances according to input and edges.
- For now, assume all transition diagrams are deterministic.

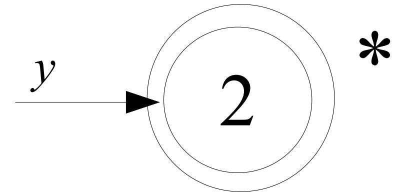
Transition Diagrams

- Some conventions:

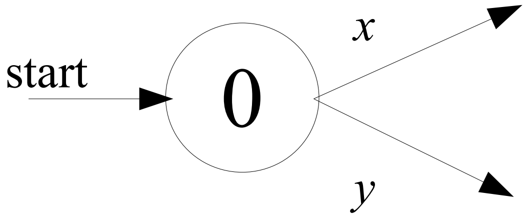
  1) Certain states are ***accepting*** or *final* (double circle): a lexeme has been found, attached to it an action (usually return token to parser).



  2) If lexeme does not include the symbol that got us to the accepting state, put a * (or more) near the accepting state.

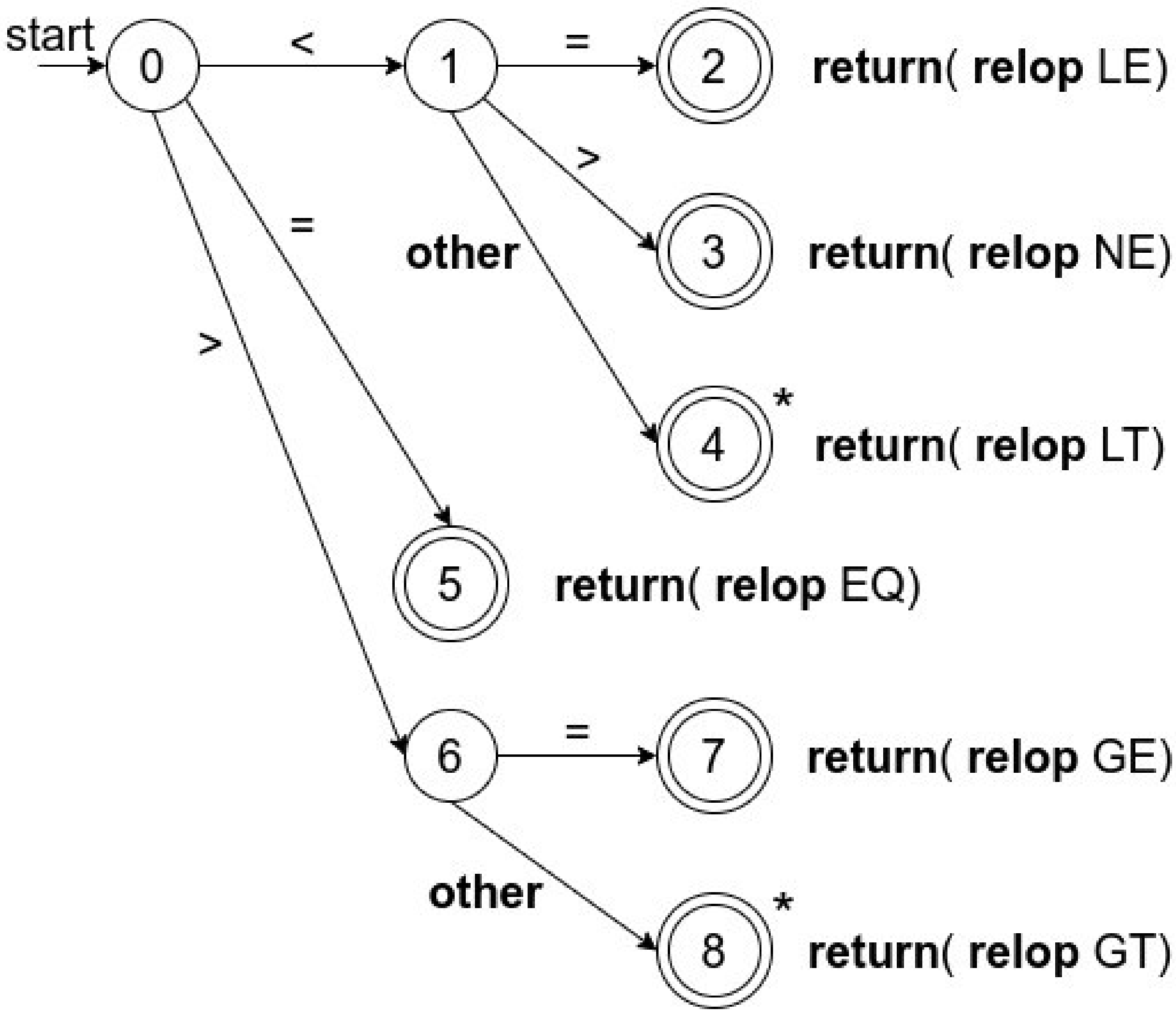


  3) One state is the ***start state*** or ***initial state***: edge labeled start coming from nowhere.



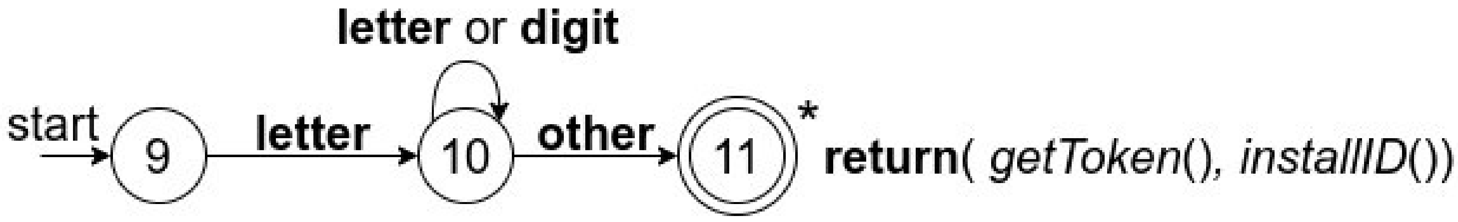
Transition Diagrams - **relop**

$$relop \rightarrow < \mid > \mid <= \mid >= \mid = \mid <>$$



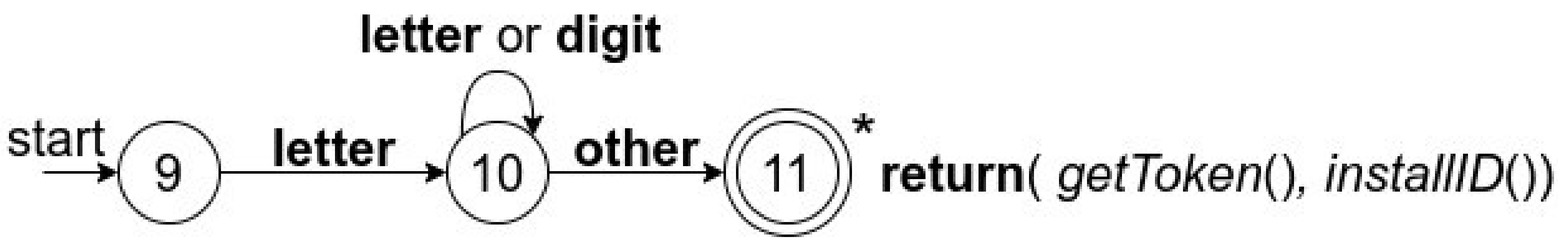
Transition Diagrams - **id**

$$id \rightarrow letter\ (\ letter\ |\ digit\ )^*$$

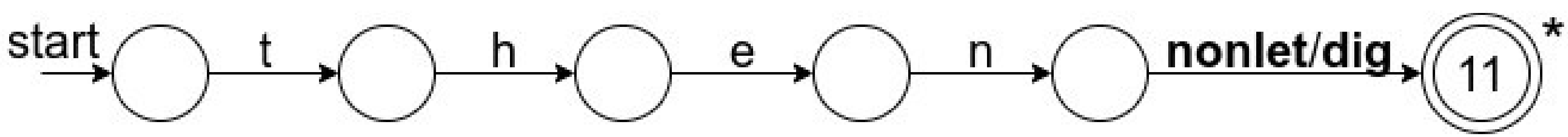


Recognition of Reserved Words and Identifiers

- 2 Approaches to differentiate reserved words from identifiers:
  1) Using symbol table initially loaded with reserved words:

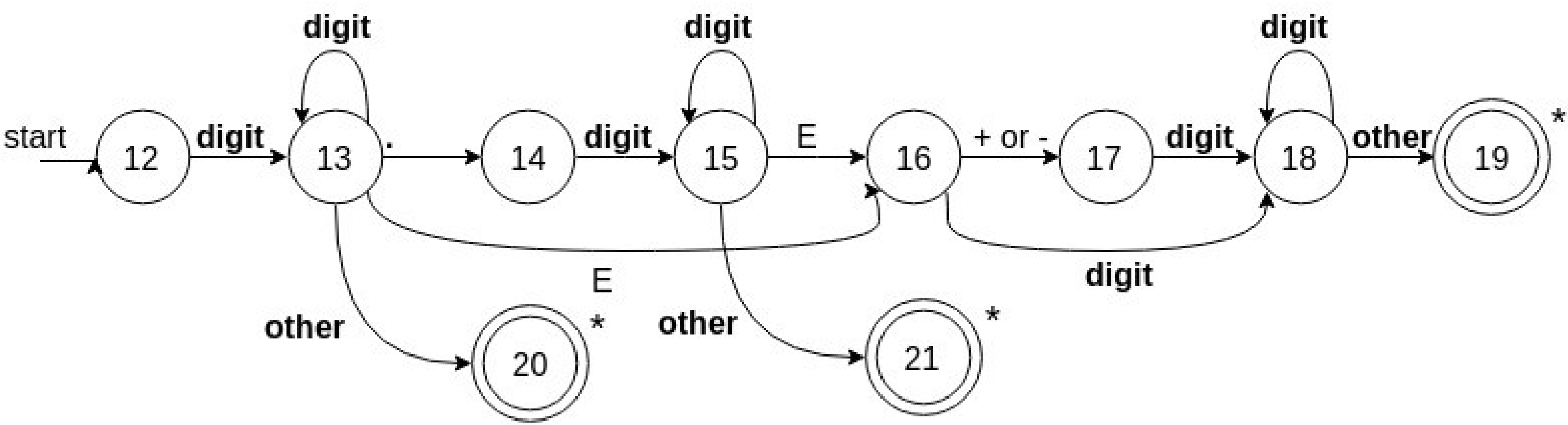


  2) Create a diagram for each keyword, with higher priority for keywords. (must check for end of word, **thenextvalue**):



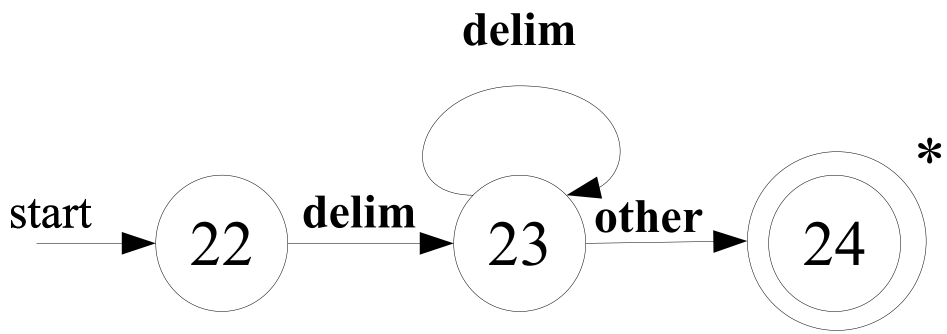
Recognizing Unsigned Number

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



Recognizing Whitespace

$ws \rightarrow$ (**blank** | **tab** | **newline**)+



Transition Diagram Based Lexical Analyzer

- Each state → a piece of code.
- A switch statement to the next state given input symbol.

Code for **relop** Transition Diagram

```
TOKEN getRelop(){
  TOKEN retToken = new(RELOP);
  while(1) { /* repeat character processing until return or
  failure */
    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);                        }
              }}
```

Transition Diagram Based Lexical Analyzer - Ways

1) Try diagrams sequentially, fail() resets *forward* and starts another diagram. A diagram for each keyword can be used this way, just try them before **id**.

2) Try diagrams in parallel. Prefer the longest prefix of the input to resolve similar prefixes.

3) Combine all diagrams into one. In prev. examples, combine all start states into one state. This is easy, cause they differ in first character. It is not always that easy.

The Lexical Analyzer Generator Lex

- Skip section 3.5 (We Use ANTLR).

Finite Automata

- Graphs like transition diagrams, but:
  1) say “yes’ or “no” about an input string.
  2) NFA: the same symbol may label multiple edges.

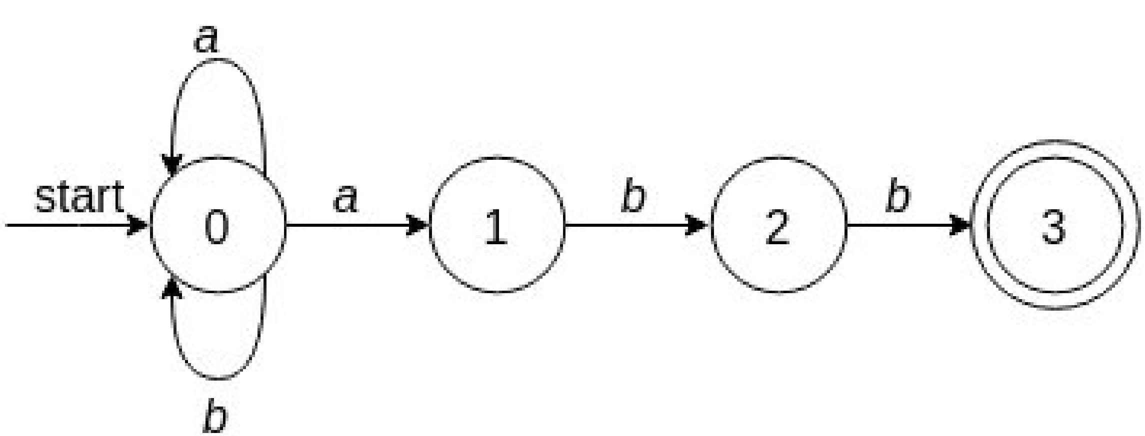
     DFA: for each state and for each symbol exactly one edge.

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 Σ U {ε} a set of *next states*.
  4) A state s_o 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*).

Transition Graph

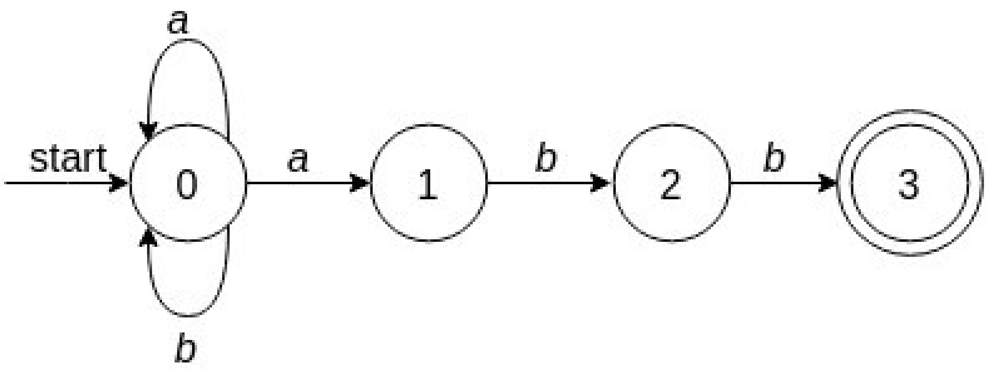
- NFA can be represented by a transition graph, similar to transition diagram, except:
  - The same symbol can label multiple edges.
  - ε can label an edge.
- Ex: $\mathbf{(a|b)^*abb}$



Transition Tables

- Rows for states and columns for symbols.
- Advantage: Easy to find transition.

  Disadvantage: large space with large input alphabet while most states have no move with all symbols.
- Ex: **(a|b)*abb**



State	*a*	b	ε
0	{0,1}	{0}	Φ
1	Φ	{2}	Φ
2	Φ	{3}	Φ
3	Φ	Φ	Φ

Acceptance of Input Strings by Automata

- An NFA *accepts x* iff there is some path from *start state* to one *accepting state s*uch that the path forms *x*.
- Ex: *aabb*

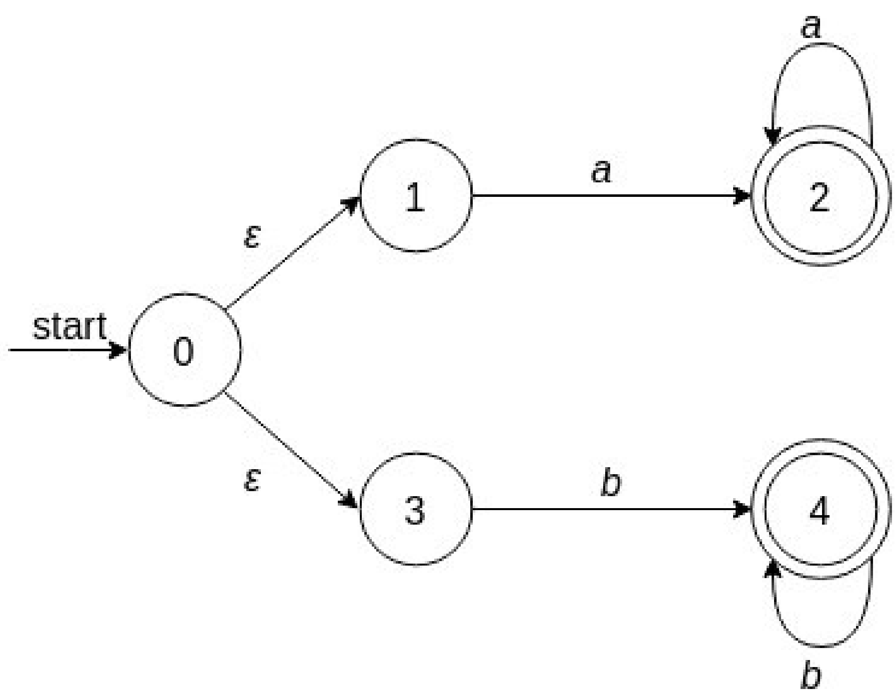
$$0 \xrightarrow{a} 0 \xrightarrow{a} 1 \xrightarrow{b} 2 \xrightarrow{b} 3$$

$$0 \xrightarrow{a} 0 \xrightarrow{a} 0 \xrightarrow{b} 0 \xrightarrow{b} 0$$

- The *language defined* (or *accepted*) by an NFA is the set of strings labeling some path from the start state to an accepting state $L(A)$.

Acceptance of Input Strings by Automata

- Ex: $L(\mathbf{aa}^*|\mathbf{bb}^*)$



$$0 \xrightarrow{\varepsilon} 0 \xrightarrow{a} 1 \xrightarrow{a} 2 \xrightarrow{a} 3$$

DFA

- Special case of 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.
- NFA is abstract, DFA is concrete algorithm.
- Every regular expression and every NFA can be converted to DFA.
- DFA is what is implemented or simulated to build lexical analyzer.

Simulating a DFA Algorithm

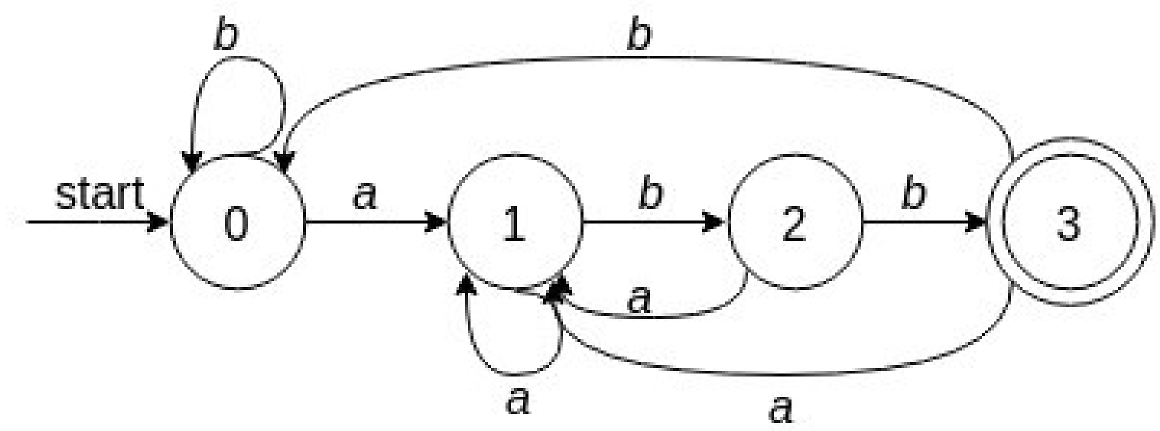
- **INPUT** : An input string x terminated by an end-of-file character **eof**. A DFA D with start state s_o, accepting states F, and transition function *move*.
- **OUTPUT** : Answer "yes" if D accepts x; "no" otherwise.
- **METHOD** : Apply the following algorithm to the input string x. The function *move(s,c)* gives the state to which there is an edge from state s on input c. The function *nextChar* returns the next character of the input string x.

Algorithm Code

```
s = s0
c = nextChar();
while ( c != eof ) {
   s = move(s,c);
   c = nextChar();
}
if ( s is in F ) return "yes";
else return "no";
```

DFA Graph

- Ex: **(a|b)*****abb**



- Given *ababb,* this DFA enters the seq. 0, 1, 2, 1, 2, 3 and returns “yes”.

$$0 \xrightarrow{a} 1 \quad \xrightarrow{b} 2 \quad \xrightarrow{a} 1 \quad \xrightarrow{b} 2 \xrightarrow{b} 3$$