Lexical Analysis

Responsibilities

Scans the input program

Removes white spaces

Removes comments

Extracts and identifies tokens

Generates lexical errors

Passes tokens to parser

Terminologies

Token: classification for a common set of strings

Examples: Identifier, Integer, Float, Operator,....

Pattern: The rules that characterize the set of strings for a token

Examples: [0-9]+

Lexeme: Actual sequence of characters that matches a pattern and has a given Token class

Examples:

Identifier: sum, data, x

Integer: 345, 2, 0, 629,....

Lexical Errors

Error Handling is localized with respect to the input source code

For example: fi (a == f(x)) … generates no lexical error in C

In what situations do errors occur?

Prefix of remaining input does not match any defined token

Possible error recovery actions:

1. Deleting or Inserting Input Characters
2. Replacing or Transposing Characters
3. Or, skip over to next separator to *ignore* problem

Overall strategy

Use one character of *look-ahead*

Perform a case analysis

1) Based on lookahead char
2) Based on current lexeme

Outcome

3) If char can extend lexeme, continue.
4) If char cannot extend lexeme, find what the complete lexeme is (upto the previous character) and return its token. Put the lookahead back into the symbol stream.

Regular expressions

ε is a regular expression, L(ε) = {ε}

If a is a symbol in ∑, then a is a regular expression, L(a) = {a}

(r) | (s) is a regular expression denoting the language L(r) ∪ L(s)

(Strings from both languages)

(r)(s) is a regular expression denoting the language L(r)L(s)

(Strings constructed by concatenating a string from the first language with a string from the second language)

(r)* is a regular expression denoting (L(r))*

(Each string in the language is a concatenation of any number of strings in the language of s)

(r) is a regular expression denoting L(r)

Regular expressions

Examples:

letter → A | B | … | Z | a | b | … | Z | _

digit → 0 | 1 | … | 9

id → letter (letter | digit)*

Regular expressions

Extensions

One or more instances: (r)+

Zero of one instances: r?

Character classes: [abc]

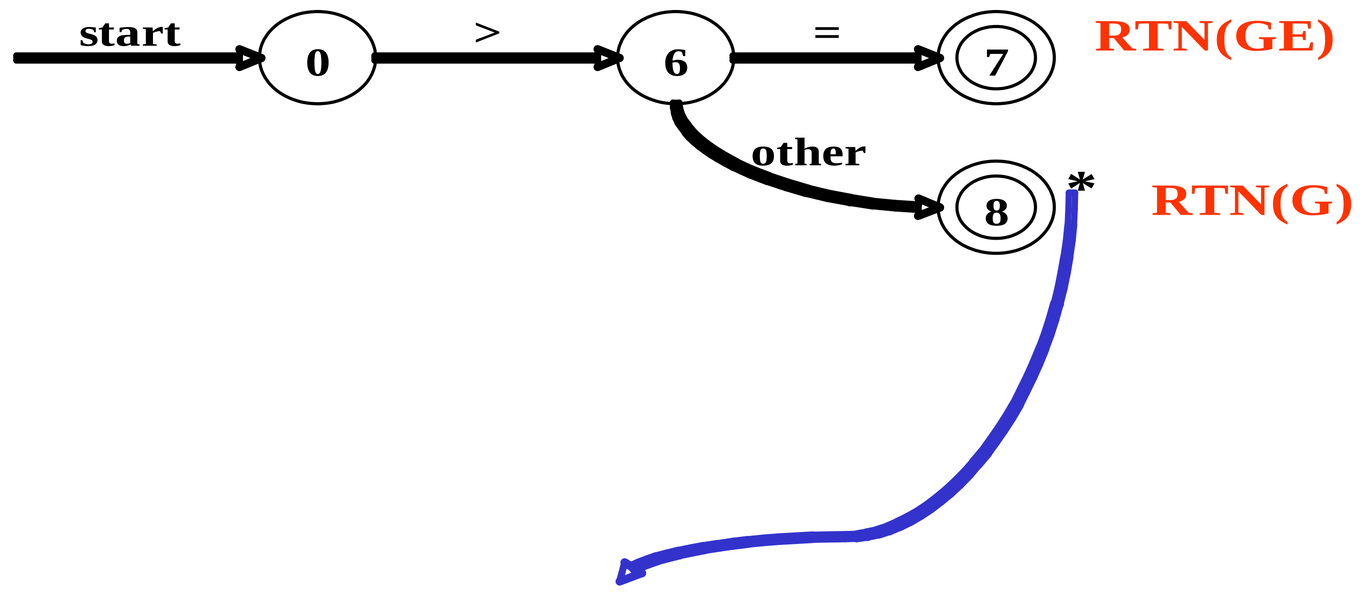
Example:
letter → [A-Za-z]

digit → [0-9]

id → letter(letter|digit)*

Transition diagrams

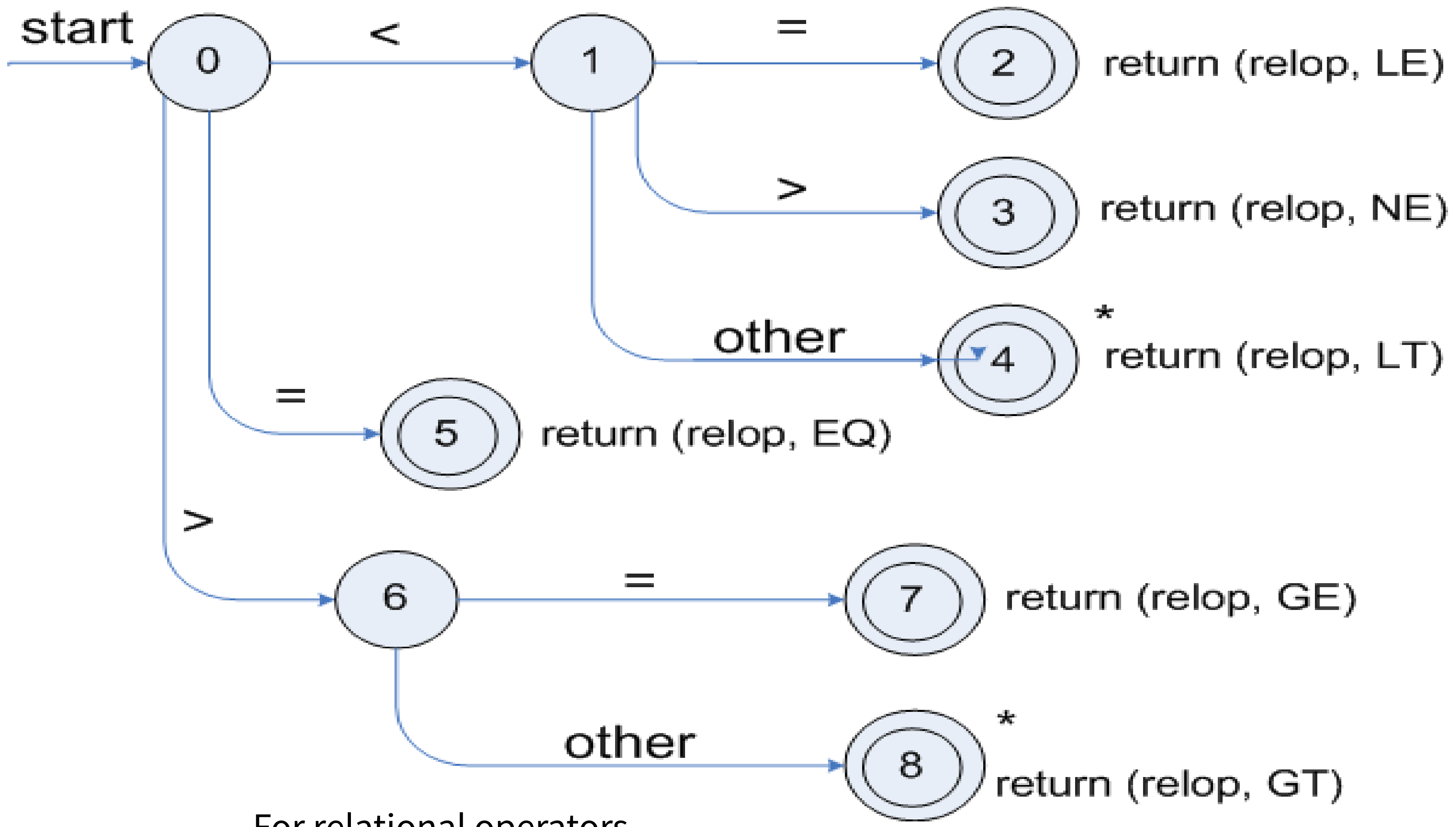
$\underline{\geq =}$



WeÕve accepted Ò>Óand have read other char that must be unread.

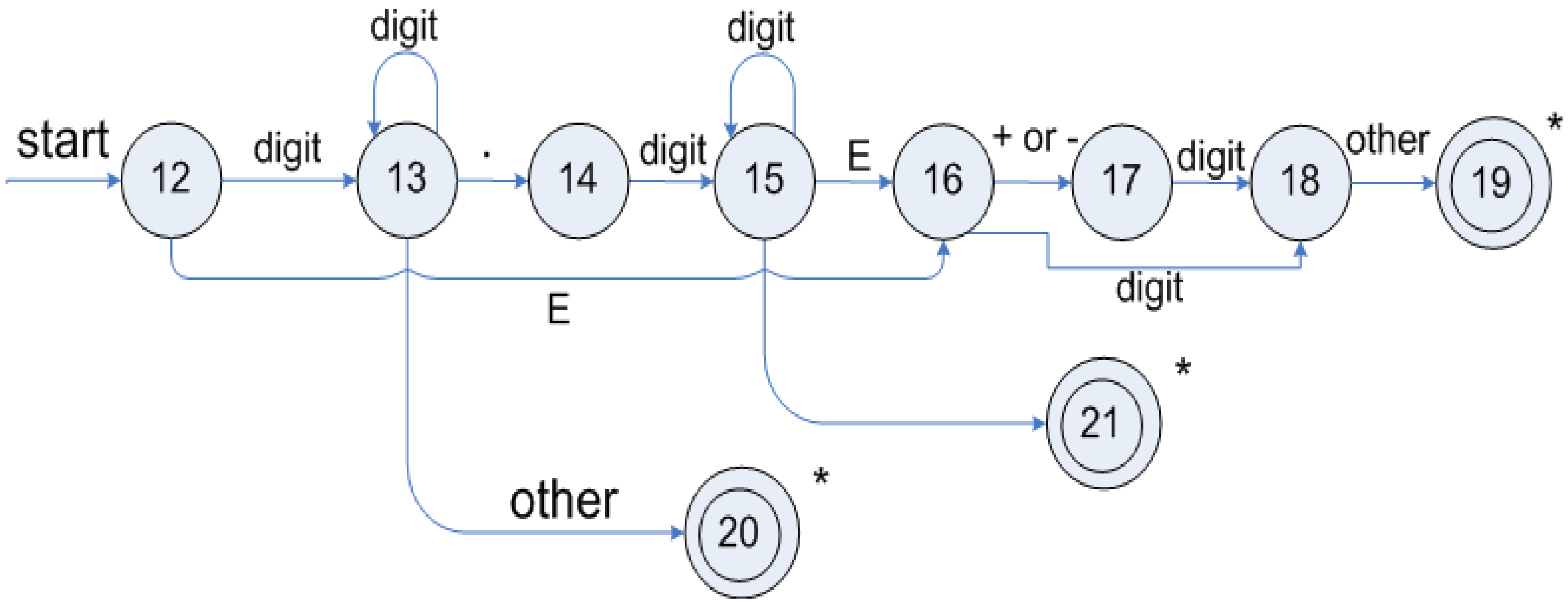
An additional character is read, that needs to be unread(return to input buffer

Transition diagrams



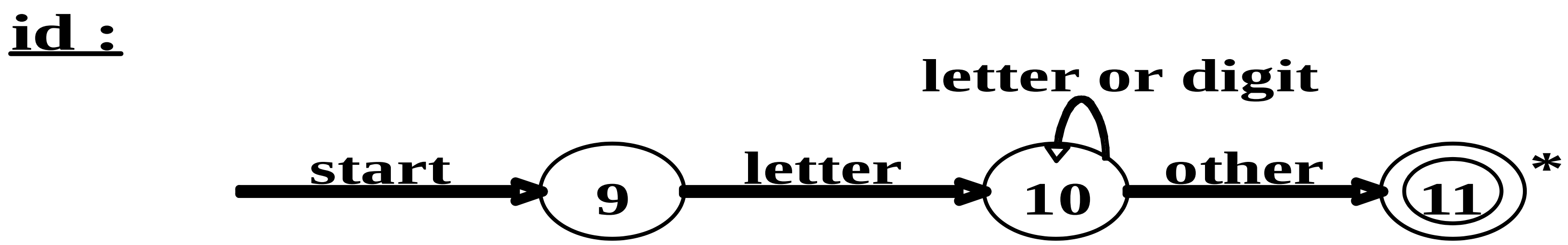
For relational operators

Transition diagrams



For unsigned numbers

Transition diagrams



For identifiers

What to do for keywords?

- Use the “Identifier” token
- After a match, lookup the keyword table
- If found, return a token for the matched keyword
- If not, return a token for the *true* identifier

Recognising Tokens

Regular expressions provide specifications for the tokens in a language

Finite automaton is used to recognise a token – implementation

A finite automaton consists of

An input alphabet Σ

A set of states S

A start state n

A set of accepting states $F \subseteq S$

A set of transitions state $\rightarrow^{\text{input}}$ state

Finite Automaton

Deterministic Finite Automata (DFA)

One transition per input per state

No ε-moves

Nondeterministic Finite Automata (NFA)

Can have multiple transitions for one input in a given state

Can have ε-moves

***Finite* automata have *finite* memory**

Need only to encode the current state

Conversion can be automated

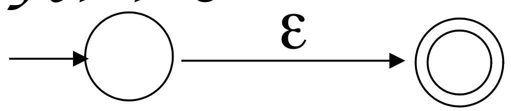
NFAs and DFAs recognize the same set of languages (regular languages)

DFAs are easier to implement

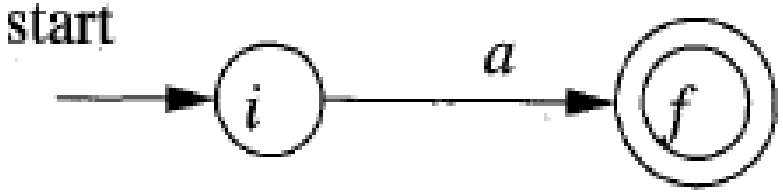
Regular Expressions to NFA McNaughton-Yamada-Thompson Algorithm

For each kind of regular expression, define an NFA

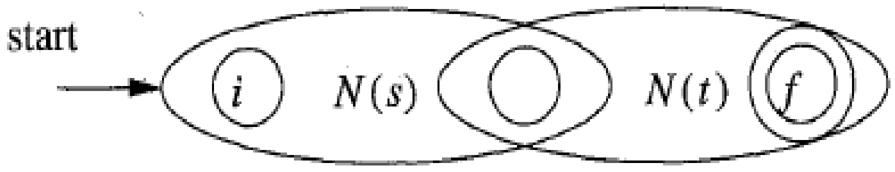
For $r=\varepsilon$



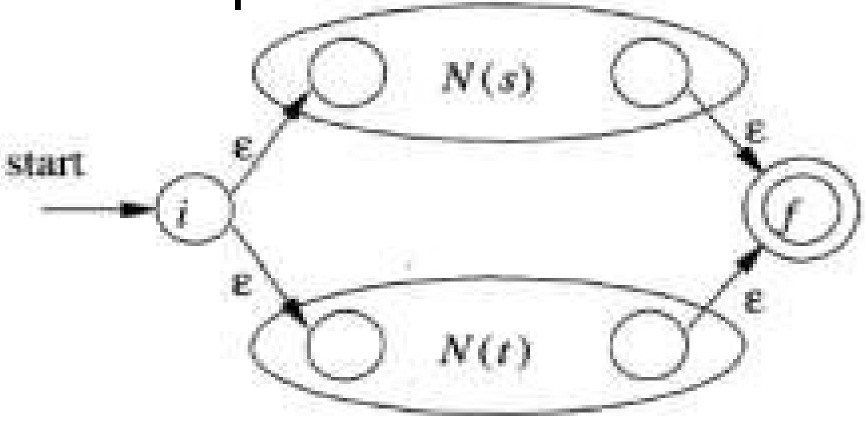
For $r=a$



For r=st

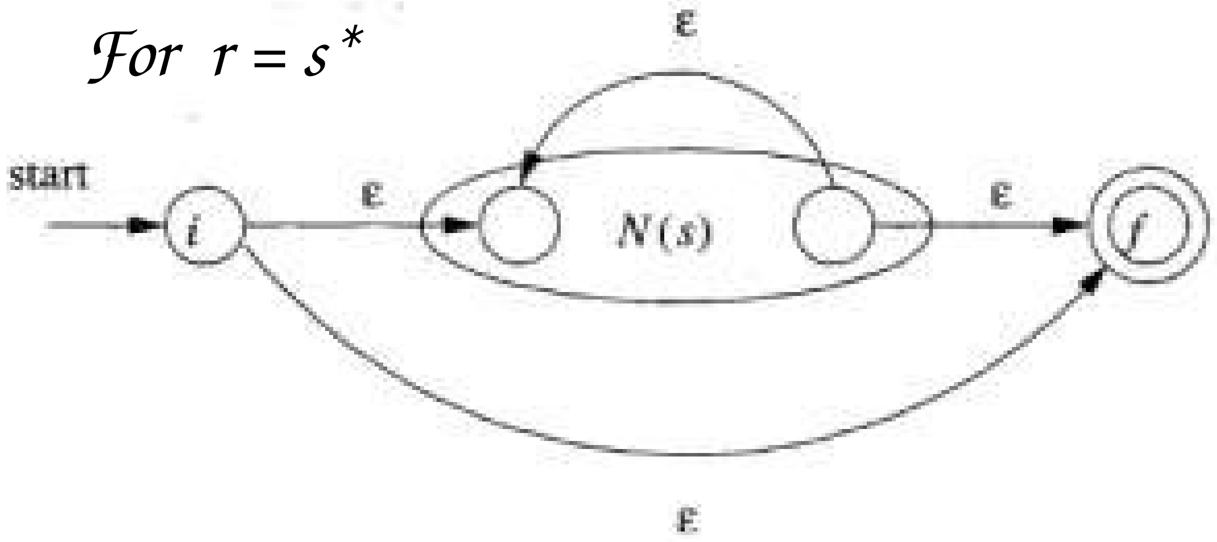


For r=s | t



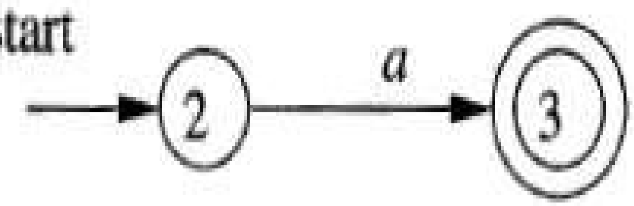
Regular Expressions to NFA
McNaughton-Yamada-Thompson Algorithm

$\mathcal{F}or \;\; r = s^*$

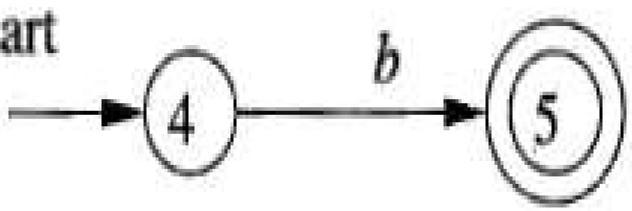


Example: (a|b)*abb

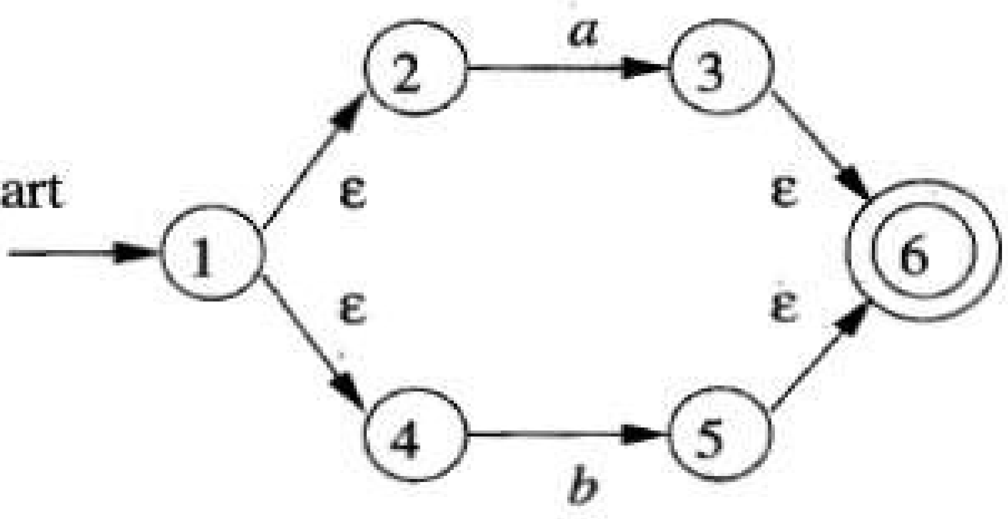
a



b

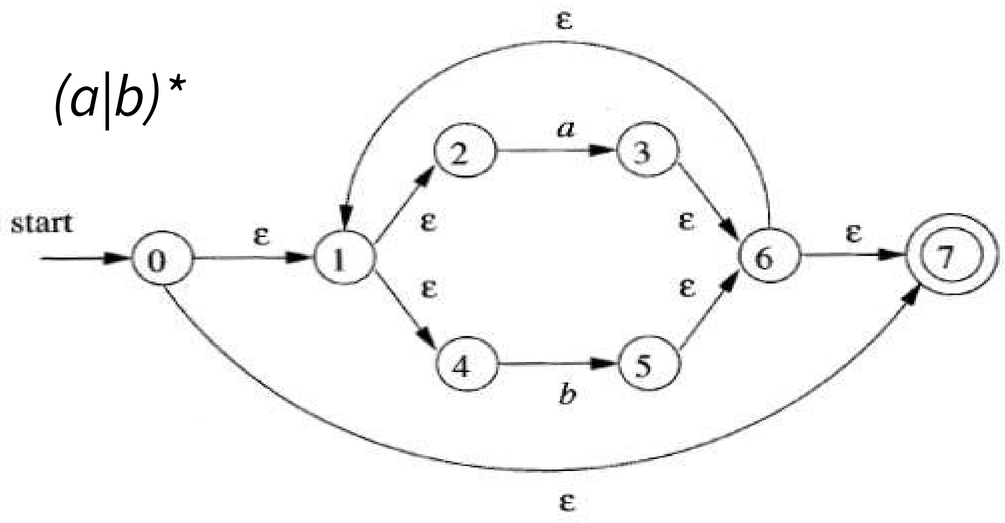


a | *b*

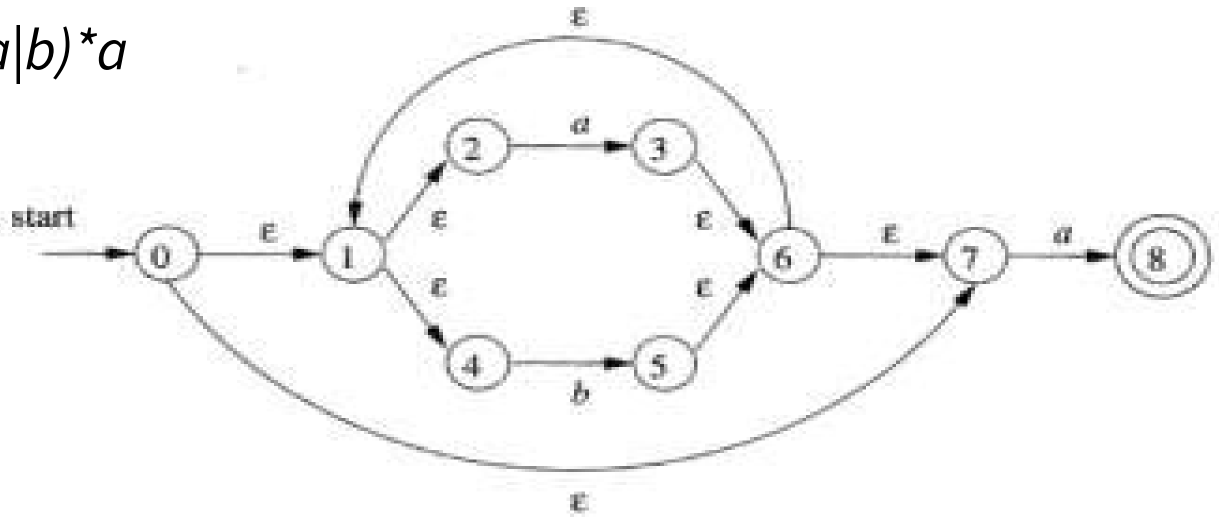


Example: (a|b)*abb

*(a|b)**

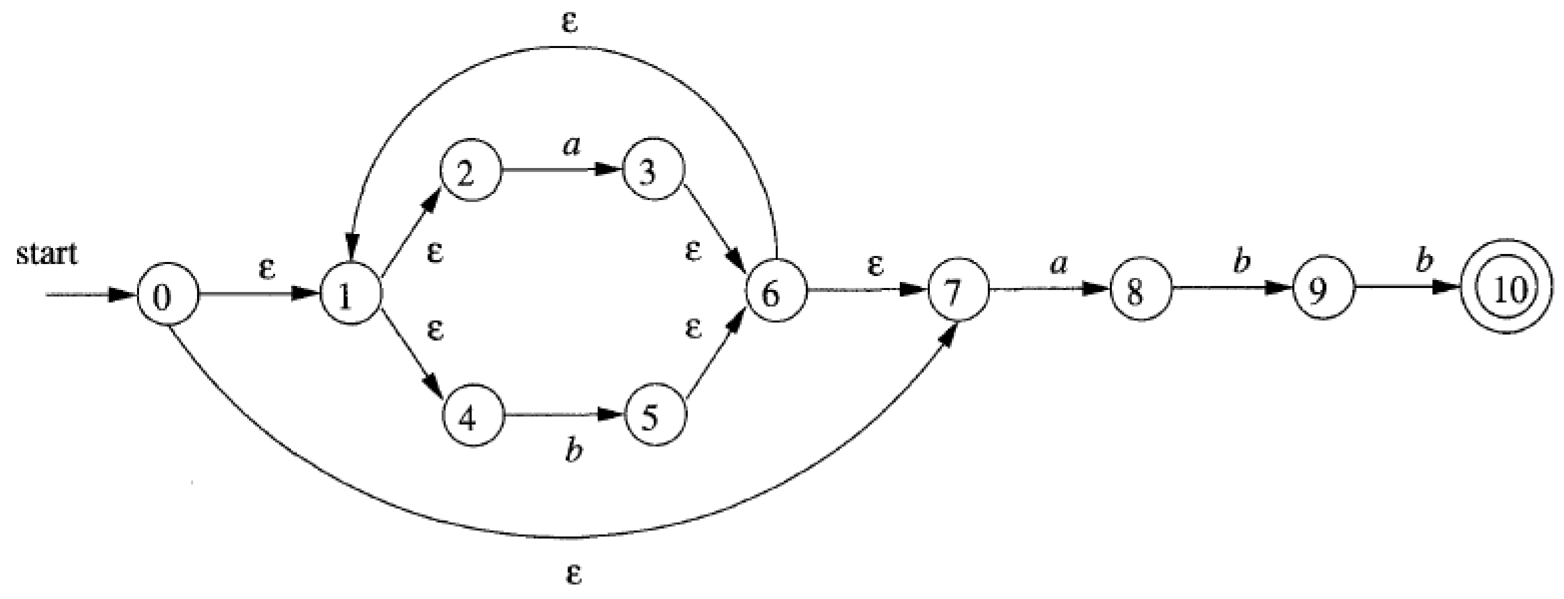


*(a|b)*a*



Example: (a|b)*abb

*(a|b)*abb*



Converting NFA to DFA

- Subset construction: each state of DFA corresponds to a set of NFA states
- For real languages NFA and DFA have approximately the same number of states (though not theoretically)

Definitions

OPERATION	DESCRIPTION
$\epsilon\text{-}closure(s)$	Set of NFA states reachable from NFA state s on ϵ-transitions alone.
$\epsilon\text{-}closure(T)$	Set of NFA states reachable from some NFA state s in set T on ϵ-transitions alone; $= \cup_{s \text{ in } T}\ \epsilon\text{-}closure(s)$.
$move(T, a)$	Set of NFA states to which there is a transition on input symbol a from some state s in T.

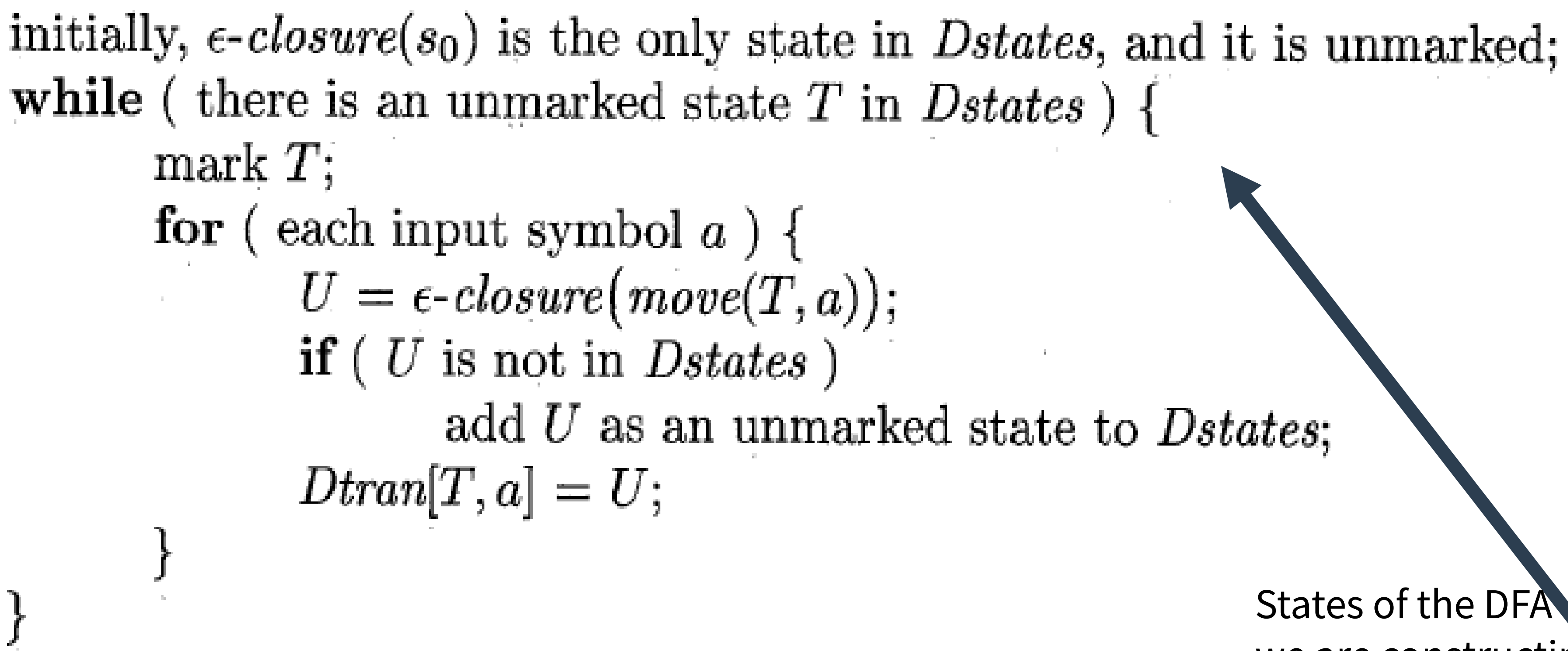
Simulating NFA

```
1)  $S = \epsilon\text{-}closure(s_0)$;
2)  $c = nextChar()$;
3)  while ( $c$ != eof ) {
4)          $S = \epsilon\text{-}closure(move(S, c))$;
5)          $c = nextChar()$;
6)  }
7)  if ( $S \cap F$ != $\emptyset$ ) return "yes";
8)  else return "no";
```

Computing ε-closure

```
Push all states of T onto stack;
Initialize ε-closure(T) to T;
while (stack is not empty) {
    pop t, the top element of the stack
    for (each state u with an edge from t to u
            labeled ε)
        if (u is not in ε-closure(T))
        { add u to ε-closure(T))
        push u onto stack
        }
}
```

Subset Constructions

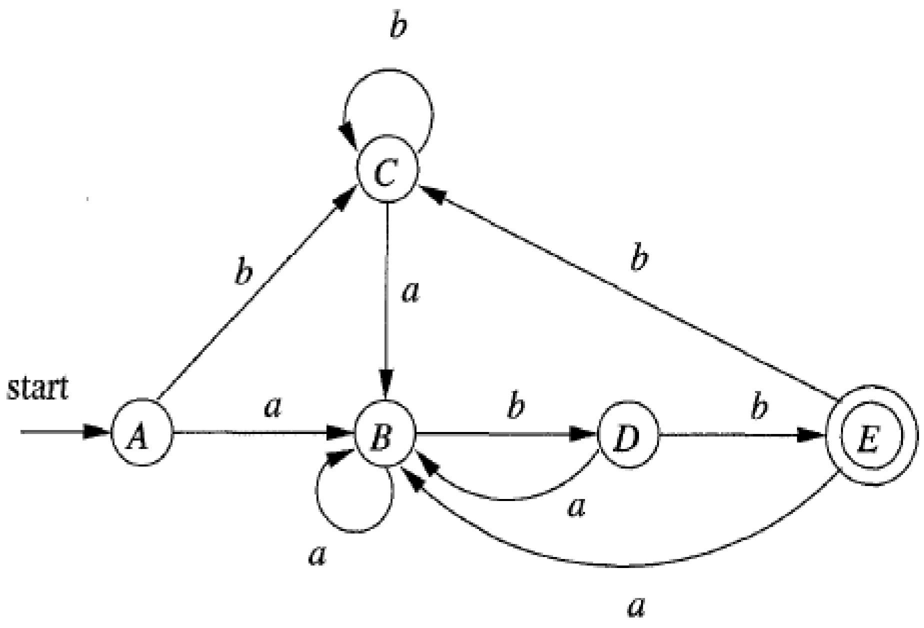


```
initially, $\epsilon$-closure($s_0$) is the only state in Dstates, and it is unmarked;
while ( there is an unmarked state $T$ in Dstates ) {
        mark $T$;
        for ( each input symbol $a$ ) {
                $U = \epsilon\text{-}closure(move(T, a))$;
                if ( $U$ is not in Dstates )
                        add $U$ as an unmarked state to Dstates;
                $Dtran[T, a] = U$;
        }
}
```

States of the DFA
we are constructing

The resulting DFA

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


start
A
B
C
D
E
a
b