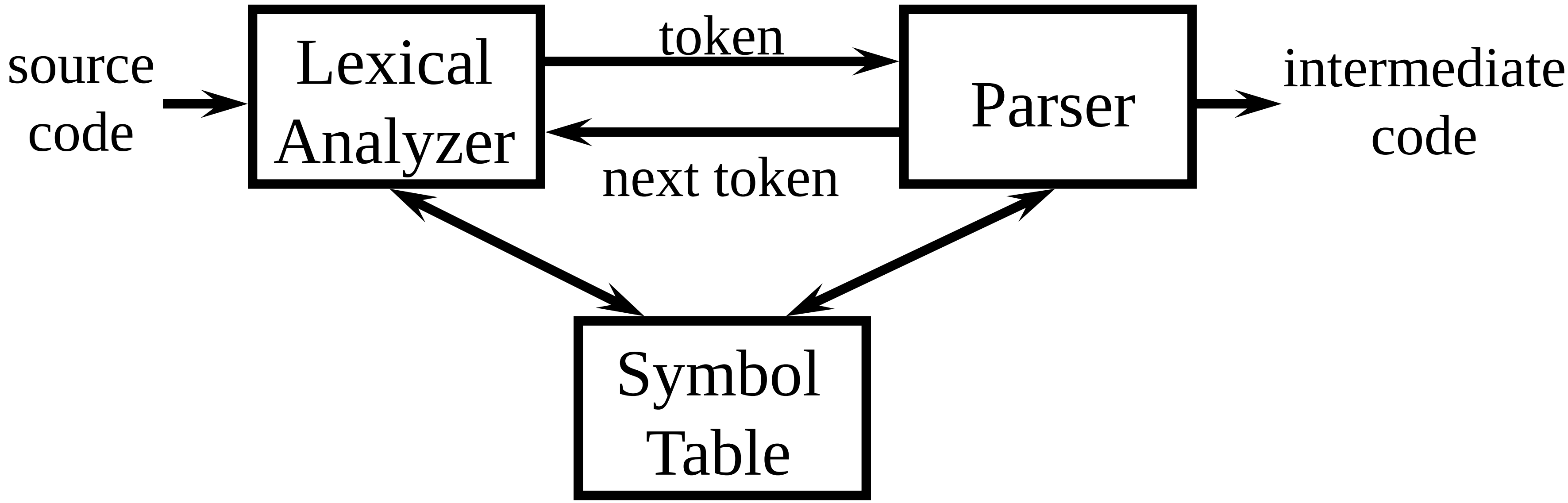
Lexical Analysis

Contents

- Introduction to lexical analyzer
- Tokens
- Regular expressions (RE)
- Finite automata (FA)
  - deterministic and nondeterministic finite automata (DFA and NFA)
  - from RE to NFA
  - from NFA to DFA
- Flex - a lexical analyzer generator

Introduction to Lexical Analyzer



Tokens

- Token (language): a set of strings
  - if, identifier, relop
- Pattern (grammar): a rule defining a token
  - if: if
  - identifier: letter followed by letters and digits
  - relop: < or <= or = or <> or >= or >
- Lexeme (sentence): a string matched by the pattern of a token
  - if, Pi, count, <, <=

Attributes of Tokens

- Attributes are used to distinguish different lexemes in a token
  - < if, >
  - < identifier, pointer to symbol table entry >
  - < relop, ‘=’ >
  - < number, value >
- Tokens affect syntax analysis and attributes affect semantic analysis

Regular Expressions

- ε is a RE denoting $\{\varepsilon\}$
- If $a \in$ alphabet, then a is a RE denoting $\{a\}$
- Suppose r and s are RE denoting L(r) and L(s)
  - (r) | (s) is a RE denoting L(r) $\cup$ L(s)
  - (r) (s) is a RE denoting L(r)L(s)
  - $(r)^*$ is a RE denoting $(L(r))^*$
  - (r) is a RE denoting L(r)

Examples

- $a \mid b$ — $\{a, b\}$
- $(a \mid b)(a \mid b)$ — $\{aa, ab, ba, bb\}$
- a^* — $\{\varepsilon, a, aa, aaa, \ldots\}$
- $(a \mid b)^*$ — the set of all strings of *a*’s and *b*’s
- $a \mid a^*b$ — the set containing the string *a* and all strings consisting of zero or more *a*’s followed by a *b*

Regular Definitions

- Names for regular expressions

  $d_1 \to r_1$

  $d_2 \to r_2$

  ...

  $d_n \to r_n$

  where r_i over alphabet $\cup \{d_1, d_2, ..., d_{i-1}\}$

- Examples:

  letter $\to$ A | B | ... | Z | a | b | ... | z

  digit $\to$ 0 | 1 | ... | 9

  identifier $\to$ {letter} ({letter} | {digit})*

Notational Shorthands

- One or more instances

  $(r)^+$ denoting $(L(r))^+$

  $r^* = r^+ \mid \varepsilon$

  $r^+ = r\, r^*$

- Zero or one instance

  $r? = r \mid \varepsilon$

- Character classes

  [abc] = a | b | c

  [a-z] = a | b | ... | z

  [^a-z] = any character except [a-z]

Examples

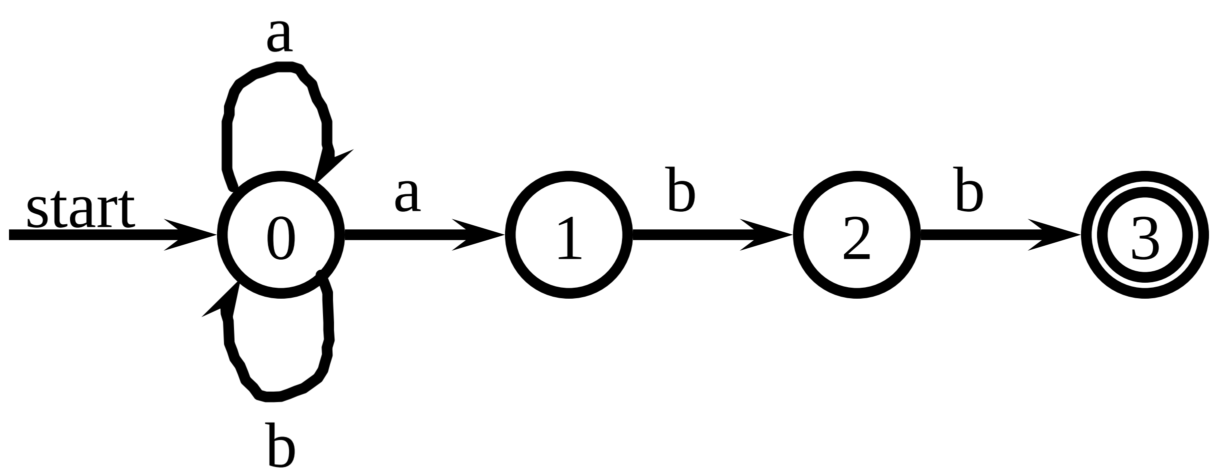
delim	→	[\t\n]
ws	→	{delim}+
letter	→	[A-Za-z]
digit	→	[0-9]
id	→	{letter}({letter}|{digit})*
number	→	{digit}+(.{digit}+)?(E[+\-]?{digit}+)?

Nondeterministic Finite Automata

- An NFA consists of
  - A finite set of *states*
  - A finite set of *input symbols*
  - A *transition function* (or *transition table*) that maps (state, symbol) pairs to sets of states
  - A state distinguished as *start state*
  - A set of states distinguished as *final states*

Transition Diagram

$(a \mid b)^*abb$



An Example

- RE: $(a \mid b)^*abb$
- States: {0, 1, 2, 3}
- Input symbols: {a, b}
- Transition function:
  (0,a) = {0,1}, (0,b) = {0}
  (1,b) = {2}, (2,b) = {3}
- Start state: 0
- Final states: {3}

Acceptance of NFA

- An NFA *accepts* an input string *s* iff there is *some* path in the transition diagram from the *start state* to some *final state* such that the edge labels along this path spell out *s*

An Example

$(a \mid b)^{*}abb$

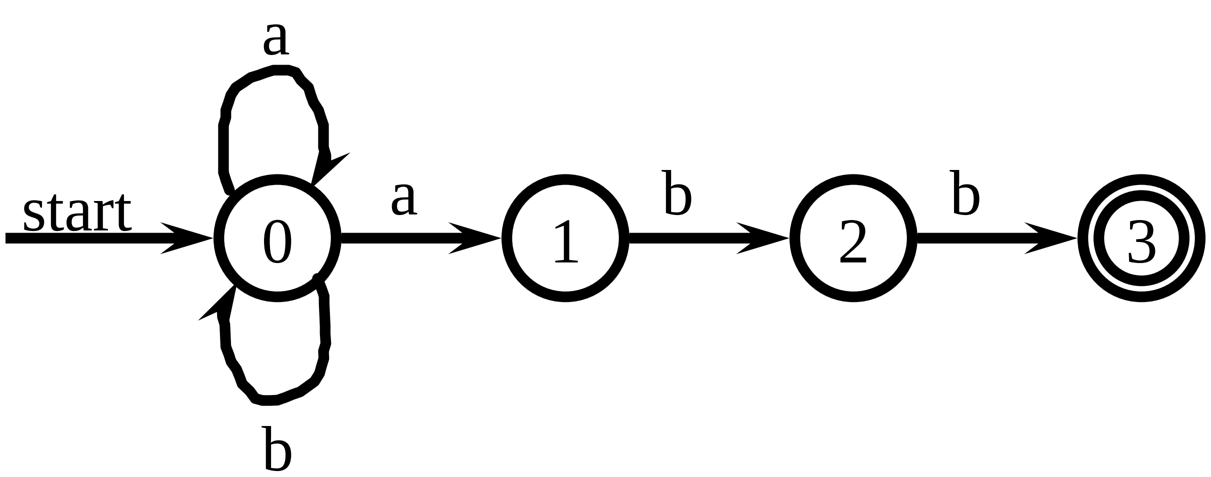


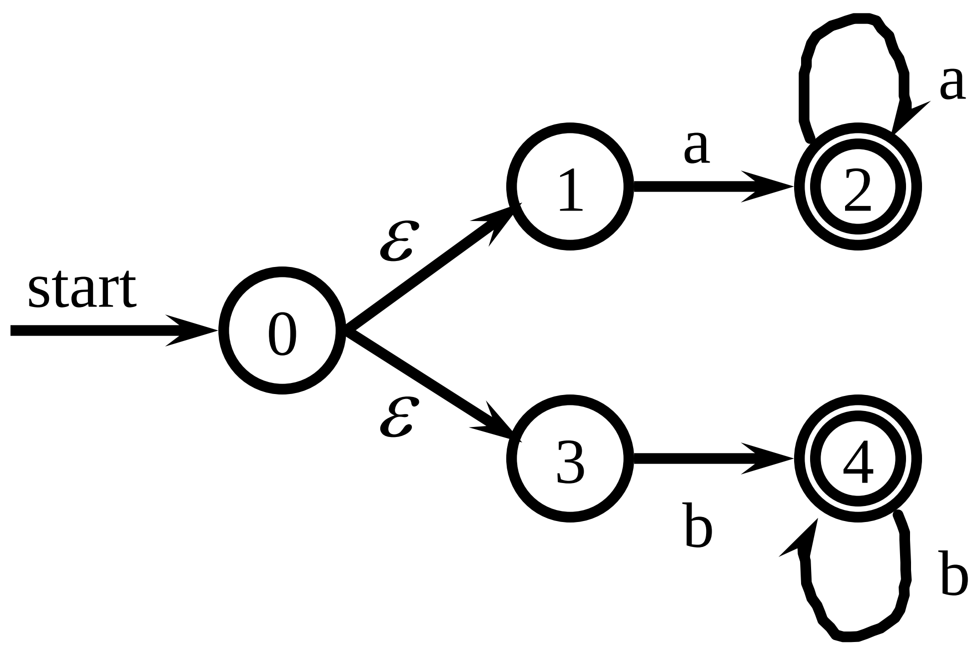
abb
aabb
babb
aaabb
ababb
baabb
bbabb
…

abb: $\{0\} \xrightarrow[a]{} \{0, 1\} \xrightarrow[b]{} \{0, 2\} \xrightarrow[b]{} \{0, 3\}$

aabb: $\{0\} \xrightarrow[a]{} \{0, 1\} \xrightarrow[a]{} \{0, 1\} \xrightarrow[b]{} \{0, 2\} \xrightarrow[b]{} \{0, 3\}$

Transition Diagram

$aa^* \mid bb^*$

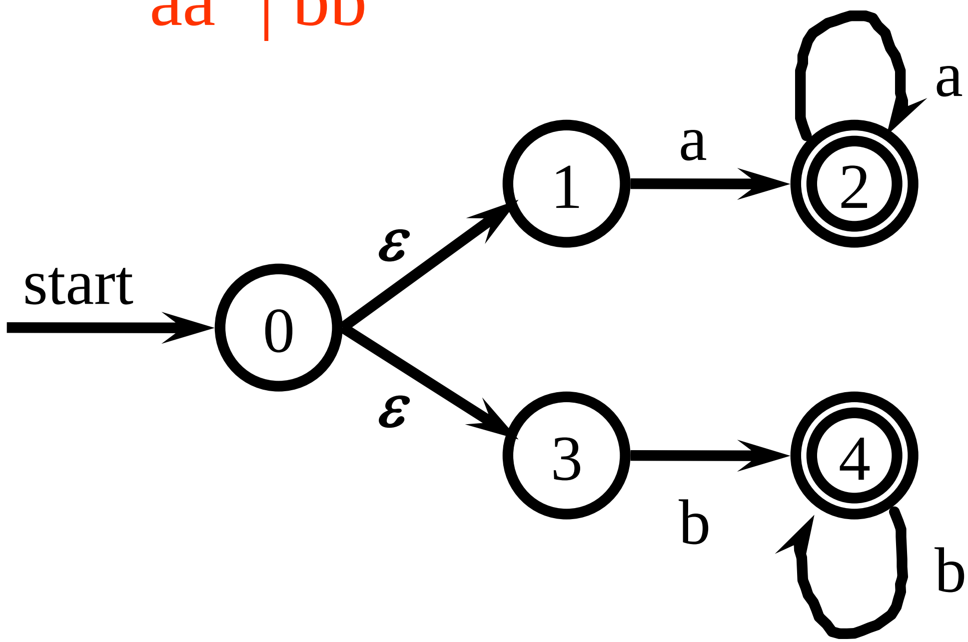


Another Example

- RE: $aa^* \mid bb^*$
- States: {0, 1, 2, 3, 4}
- Input symbols: {a, b}
- Transition function:
  $(0, \varepsilon) = \{1, 3\}$, $(1, a) = \{2\}$, $(2, a) = \{2\}$
  $(3, b) = \{4\}$, $(4, b) = \{4\}$
- Start state: 0
- Final states: {2, 4}

Another Example

$aa^* \mid bb^*$



aaa: $\{0\} \xrightarrow[\varepsilon]{} \{0, 1, 3\} \xrightarrow[a]{} \{2\} \xrightarrow[\varepsilon]{} \{2\} \xrightarrow[a]{} \{2\} \xrightarrow[\varepsilon]{} \{2\} \xrightarrow[a]{} \{2\} \xrightarrow[\varepsilon]{} \{2\}$

Simulating an NFA

Input. An input string ended with **eof** and an NFA with start state s_0 and final states F.
Output. The answer "yes" if accepts, "no" otherwise.

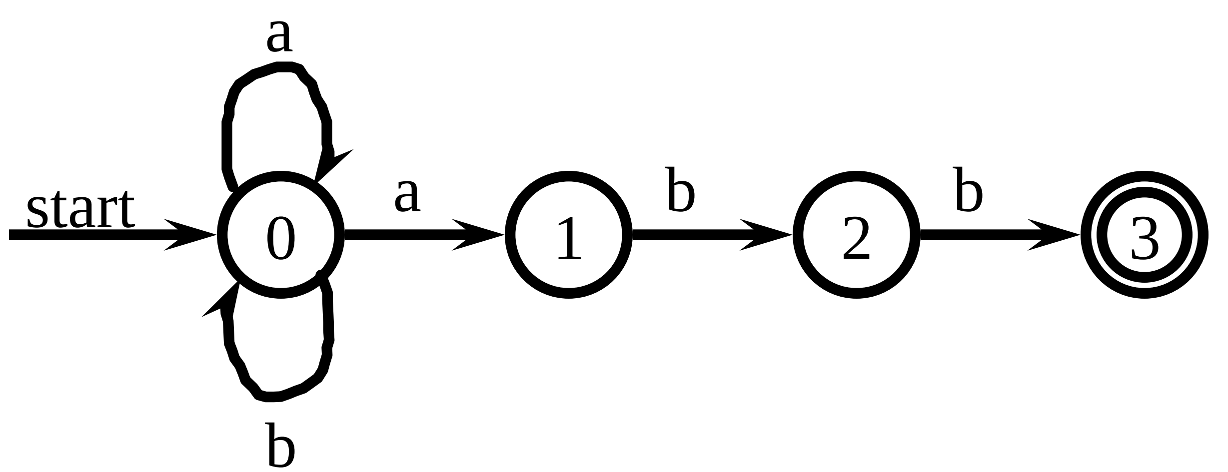
```
begin
  S := ε-closure({s0});
  c := nextchar;
  while c <> eof do begin
    S := ε-closure(move(S, c));
    c := nextchar
  end;
  if S ∩ F <> ∅ then return "yes"
  else return "no"
end.
```

Operations on NFA states

- ♠ *move*(*s*, *c*): set of NFA states reachable from NFA state *s* on input symbol *c*
- ♠ *move*(*S*, *c*): set of NFA states reachable from some NFA state *s* in *S* on input symbol *c*
- ♠ *ε-closure*(*s*): set of NFA states reachable from NFA state *s* on *ε*-transitions alone
- ♠ *ε-closure*(*S*): set of NFA states reachable from some NFA state *s* in *S* on *ε*-transitions alone

Transition Diagram

$(a \mid b)^*abb$



An Example

$(a \mid b)^* abb$

bbababb

$S = \{0\}$
$S = move(\{0\}, \mathbf{b}) = \{0\}$
$S = move(\{0\}, \mathbf{b}) = \{0\}$
$S = move(\{0\}, \mathbf{a}) = \{0, 1\}$
$S = move(\{0, 1\}, \mathbf{b}) = \{0, 2\}$
$S = move(\{0, 2\}, \mathbf{a}) = \{0, 1\}$
$S = move(\{0, 1\}, \mathbf{b}) = \{0, 2\}$
$S = move(\{0, 2\}, \mathbf{b}) = \{0, 3\}$
$S \cap \{3\} <> \varnothing$

bbabab

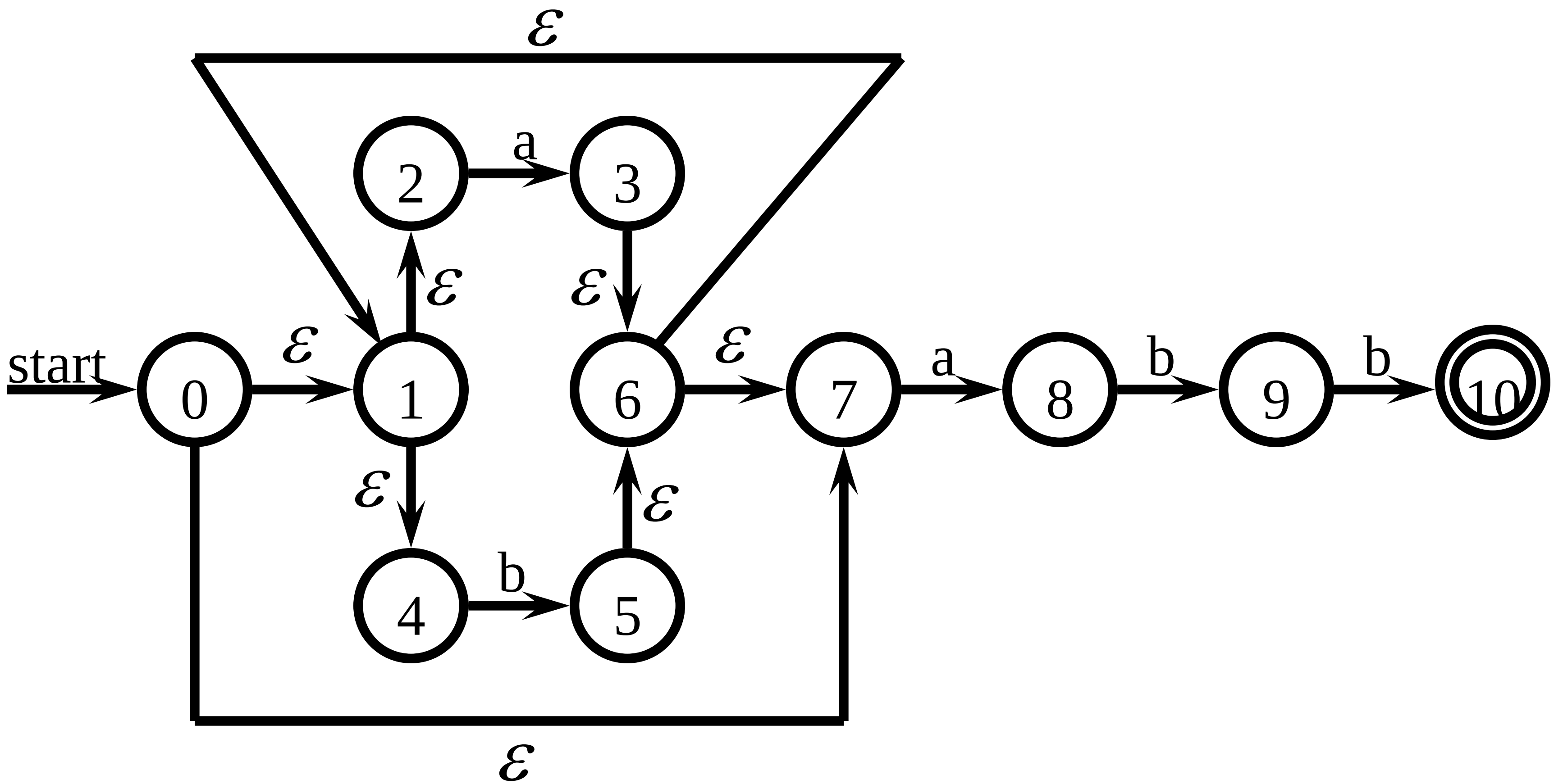
$S = \{0\}$
$S = move(\{0\}, \mathbf{b}) = \{0\}$
$S = move(\{0\}, \mathbf{b}) = \{0\}$
$S = move(\{0\}, \mathbf{a}) = \{0, 1\}$
$S = move(\{0, 1\}, \mathbf{b}) = \{0, 2\}$
$S = move(\{0, 2\}, \mathbf{a}) = \{0, 1\}$
$S = move(\{0, 1\}, \mathbf{b}) = \{0, 2\}$
$S \cap \{3\} = \varnothing$

Computation of ε-closure

Input. An NFA and a set of NFA states *S*.
Output. $T = \varepsilon\text{-}closure(S)$.
begin /* A DFT along the ε-transitions */
  push all states in *S* onto *stack*; $T := S$;
  while *stack* is not empty **do begin**
    pop *t*, the top element, off of *stack*;
    for each state *u* with an edge from *t* to *u* labeled ε **do**
      if *u* is not in *T* **do begin**
        add *u* to *T*; push *u* onto *stack*
      end
  end;
  return *T*
end.

An Example

(a | b)*abb



An Example

bbabb

$S = \varepsilon\text{-}closure(\{0\}) = \{0,1,2,4,7\}$

$S = \varepsilon\text{-}closure(move(\{0,1,2,4,7\}, \mathbf{b}))$

$= \varepsilon\text{-}closure(\{5\}) = \{1,2,4,5,6,7\}$

$S = \varepsilon\text{-}closure(move(\{1,2,4,5,6,7\}, \mathbf{b}))$

$= \varepsilon\text{-}closure(\{5\}) = \{1,2,4,5,6,7\}$

$S = \varepsilon\text{-}closure(move(\{1,2,4,5,6,7\}, \mathbf{a}))$

$= \varepsilon\text{-}closure(\{3,8\}) = \{1,2,3,4,6,7,8\}$

$S = \varepsilon\text{-}closure(move(\{1,2,3,4,6,7,8\}, \mathbf{b}))$

$= \varepsilon\text{-}closure(\{5,9\}) = \{1,2,4,5,6,7,9\}$

$S = \varepsilon\text{-}closure(move(\{1,2,4,5,6,7,9\}, \mathbf{b}))$

$= \varepsilon\text{-}closure(\{5,10\}) = \{1,2,4,5,6,7,10\}$

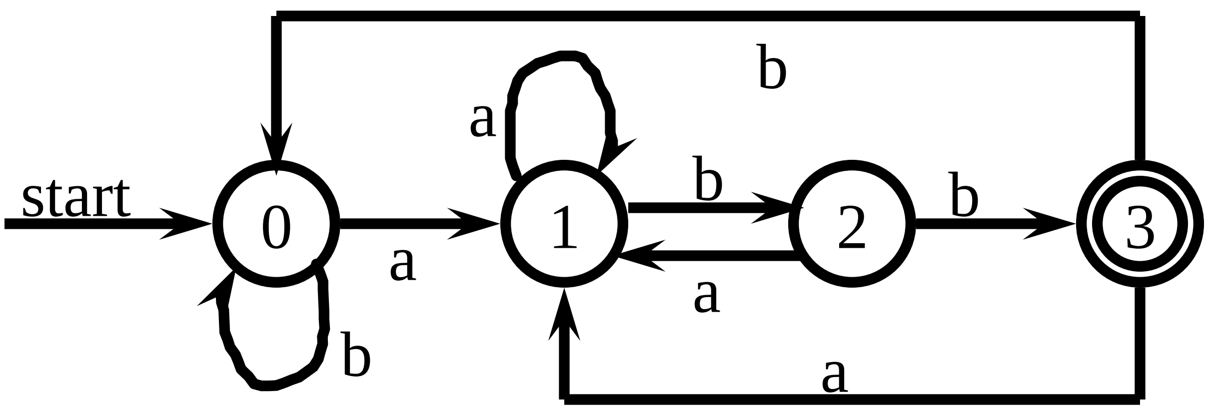
$S \cap \{10\} <> \emptyset$

Deterministic Finite Automata

- A DFA is a special case of an NFA in which
  - *no* state has an ε-transition
  - for each state *s* and input symbol *a*, there is *at most one* edge labeled *a* leaving *s*

Transition Diagram

$(a \mid b)^{*}abb$



An Example

- RE: $(a \mid b)^{*}abb$
- States: {0, 1, 2, 3}
- Input symbols: {a, b}
- Transition function:
  (0,a) = 1, (1,a) = 1, (2,a) = 1, (3,a) = 1
  (0,b) = 0, (1,b) = 2, (2,b) = 3, (3,b) = 0
- Start state: 0
- Final states: {3}

Simulating a DFA

Input. An input string ended with **eof** and a DFA with start state s_0 and final states F.
Output. The answer "yes" if accepts, "no" otherwise.

begin
  $s := s_0$;
  $c := nextchar$;
  while c <> **eof do begin**
    $s := move(s, c)$;
    $c := nextchar$
  end;
  if s is in F **then return** "yes"
  else return "no"
end.

An Example

$(a \mid b)^{*}abb$

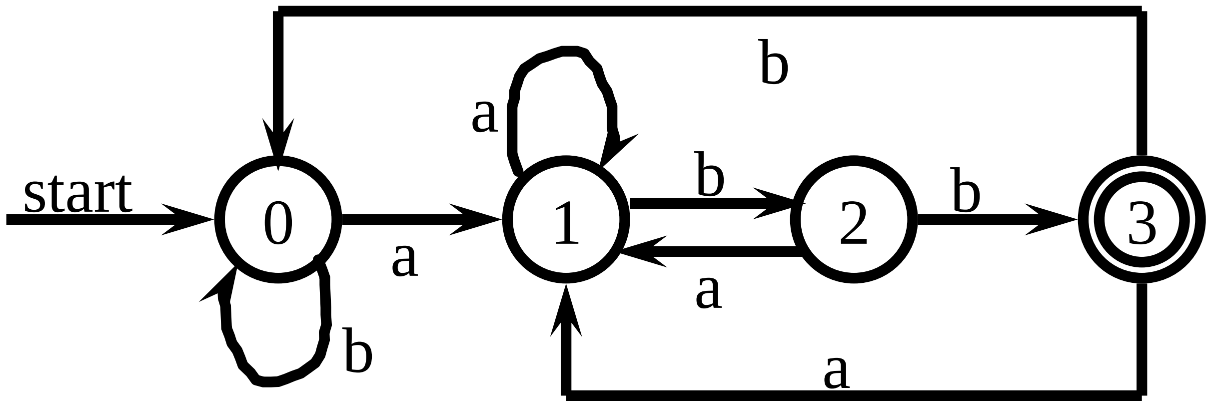


abb: $0 \xrightarrow[a]{} 1 \xrightarrow[b]{} 2 \xrightarrow[b]{} 3$

aabb: $0 \xrightarrow[a]{} 1 \xrightarrow[a]{} 1 \xrightarrow[b]{} 2 \xrightarrow[b]{} 3$

共勉

子貢曰：貧而無諂，富而無驕，何如。

子曰：可也，未若貧而樂，富而好禮者也。

子貢曰：詩云：「如切如磋，如琢如磨。」

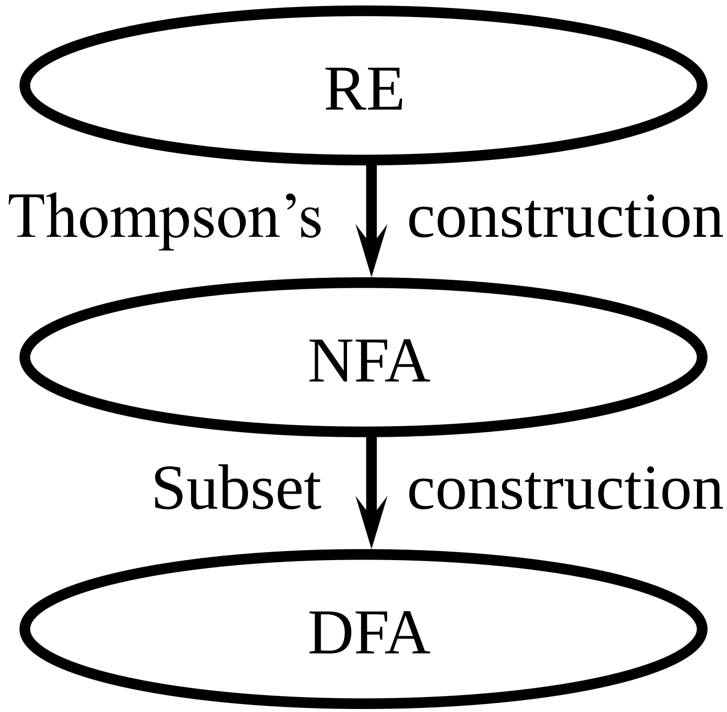
其斯之謂與。

子曰：賜也，始可與言詩已矣；

告諸往而知來者。

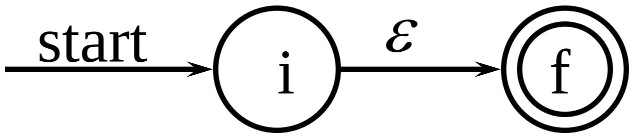
-- 論語

Lexical Analyzer Generator

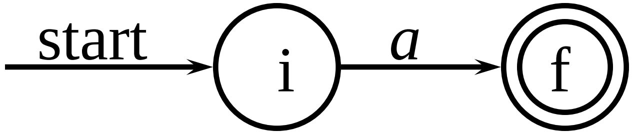


From a RE to an NFA

- Thompson's construction algorithm
  - For ε , construct



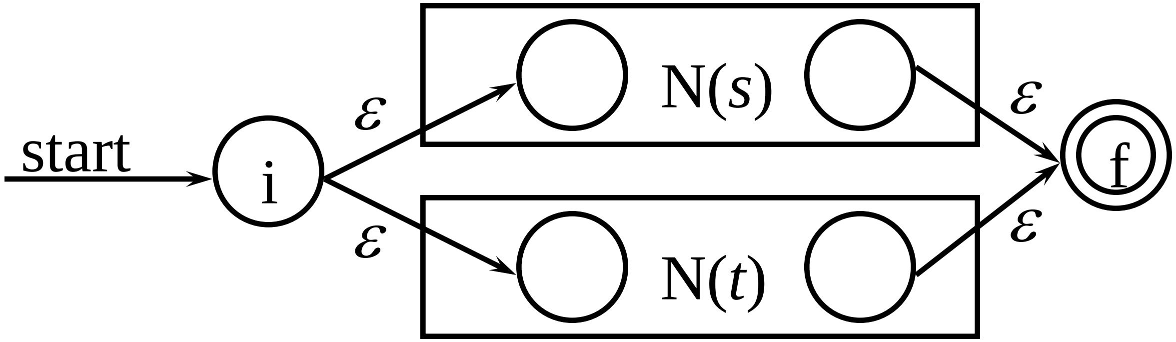
  - For a in alphabet, construct



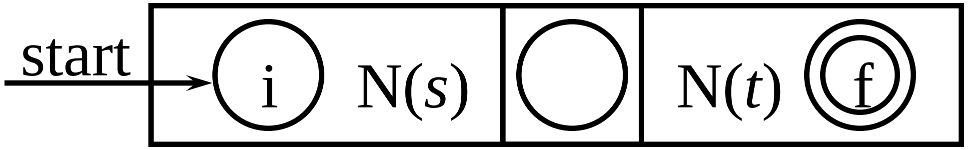
From a RE to an NFA

– Suppose N(*s*) and N(*t*) are NFA for RE *s* and *t*

- for *s* | *t*, construct

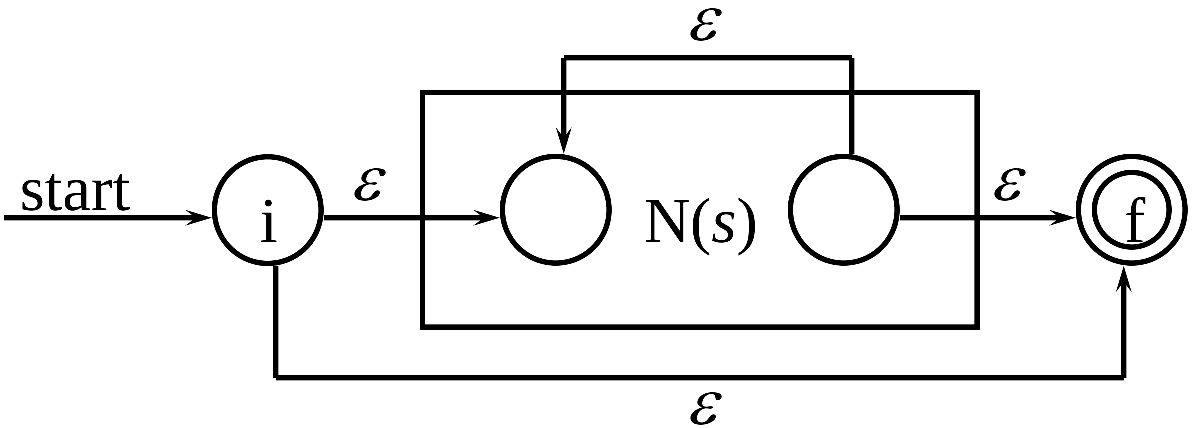


- for *st*, construct



From a RE to an NFA

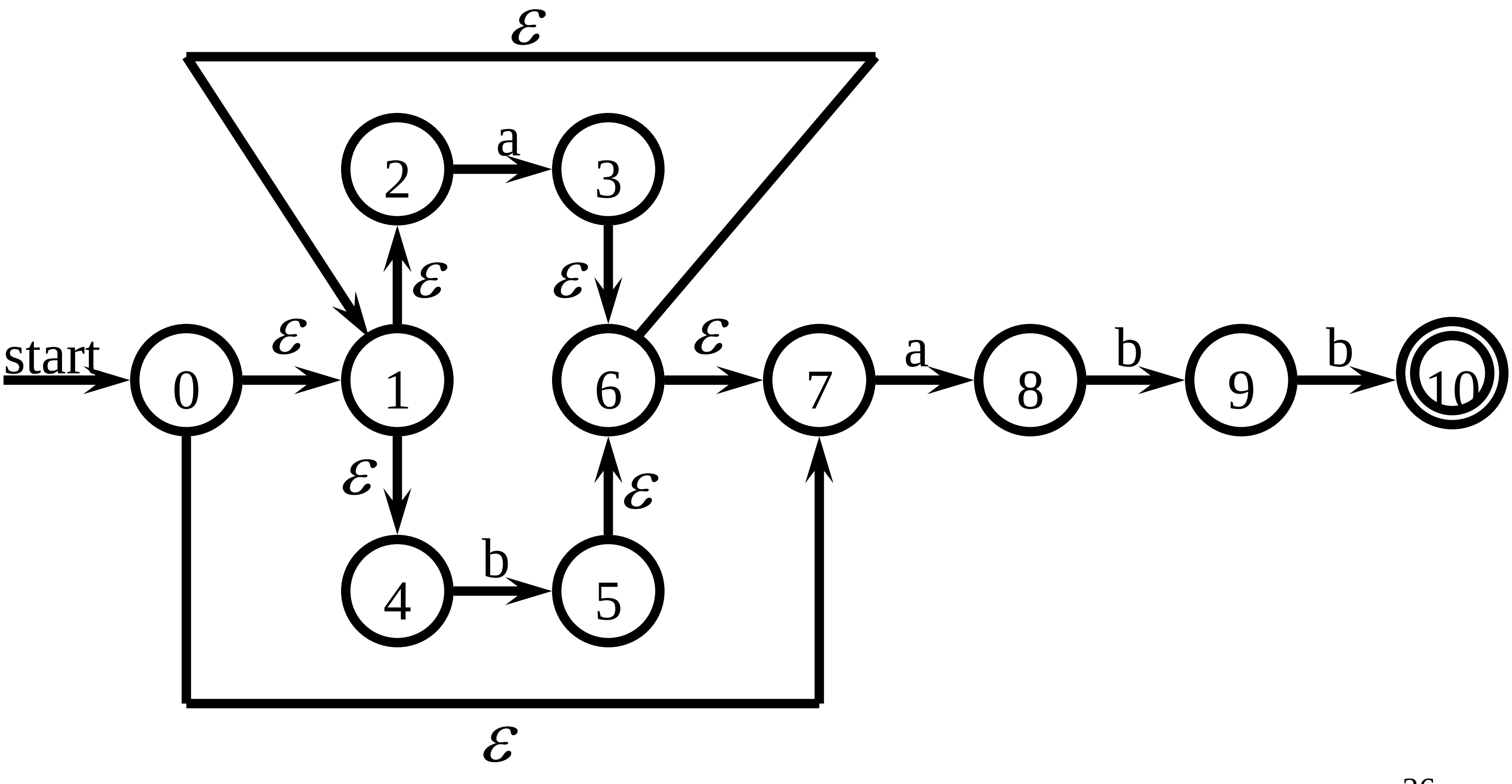
- for s^*, construct



- for (s), use $N(s)$

An Example

(a | b)*abb



From an NFA to a DFA

a set of NFA states $\equiv$ a DFA state

- Find the initial state of the DFA
- Find all the states in the DFA
- Construct the transition table
- Find the final states of the DFA

Subset Construction Algorithm

Input. An NFA N.

Output. A DFA D with states *Dstates* and trasition table *Dtran*.

begin
  add ε-*closure*(s_0) as an unmarked state to *Dstates*;
  while there is an unmarked state T in *Dstates* **do begin**
    mark T;
    for each input symbol a **do begin**
      U := ε-*closure*(*move*(T, a));
      if U is not in *Dstates* **then**
        add U as an unmarked state to *Dstates*;
      Dtran[T, a] := U
    end
end.

An Example

ε-*closure*({0}) = {0,1,2,4,7} = A

ε-*closure*(*move*(A, *a*)) = ε-*closure*({3,8}) = {1,2,3,4,6,7,8} = B

ε-*closure*(*move*(A, *b*)) = ε-*closure*({5}) = {1,2,4,5,6,7} = C

ε-*closure*(*move*(B, *a*)) = ε-*closure*({3,8}) = B

ε-*closure*(*move*(B, *b*)) = ε-*closure*({5,9}) = {1,2,4,5,6,7,9} = D

ε-*closure*(*move*(C, *a*)) = ε-*closure*({3,8}) = B

ε-*closure*(*move*(C, *b*)) = ε-*closure*({5}) = C

ε-*closure*(*move*(D, *a*)) = ε-*closure*({3,8}) = B

ε-*closure*(*move*(D, *b*)) = ε-*closure*({5,10}) = {1,2,4,5,6,7,10} = E

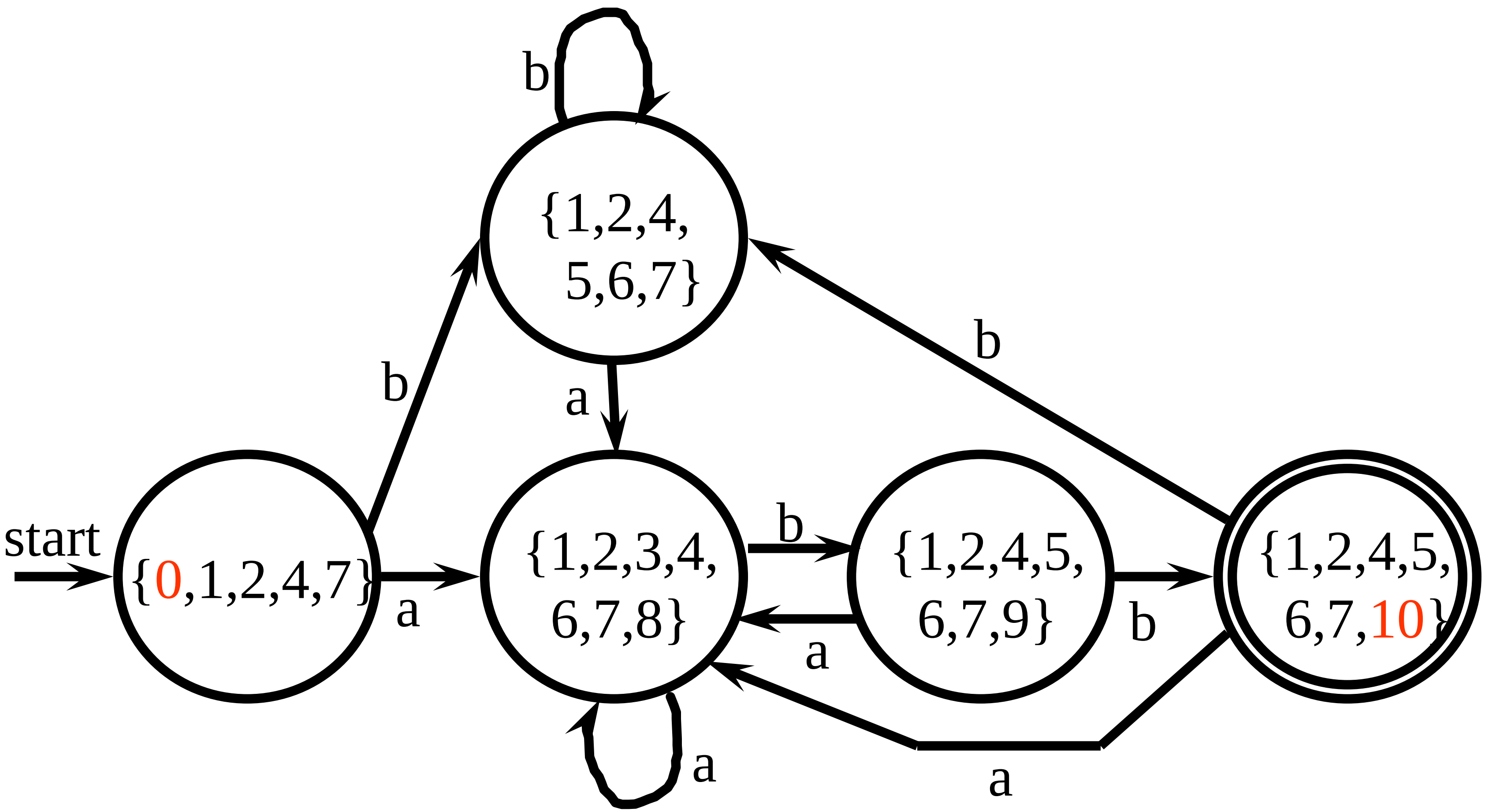
ε-*closure*(*move*(E, *a*)) = ε-*closure*({3,8}) = B

ε-*closure*(*move*(E, *b*)) = ε-*closure*({5}) = C

An Example

State	Input Symbol	
	a	*b*
A = {0,1,2,4,7}	B	C
B = {1,2,3,4,6,7,8}	B	D
C = {1,2,4,5,6,7}	B	C
D = {1,2,4,5,6,7,9}	B	E
E = {1,2,4,5,6,7,10}	B	C

An Example

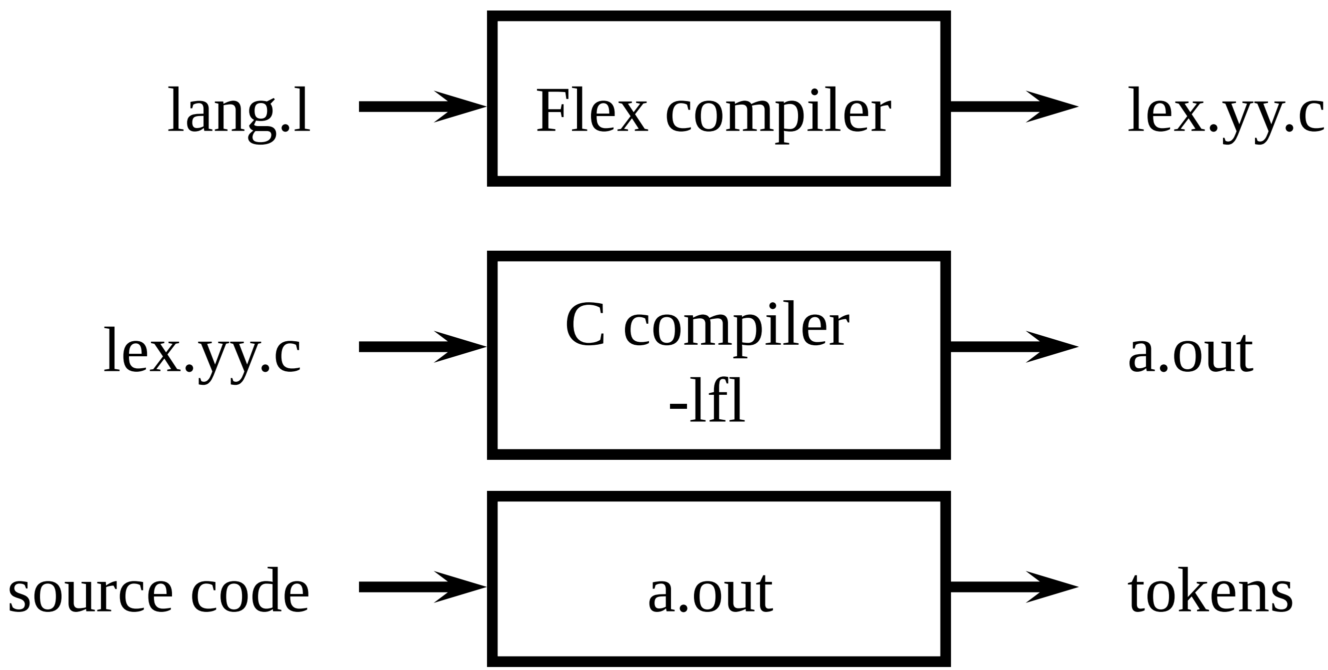


Time-Space Tradeoffs

- RE to NFA, simulate NFA
  - time: $O(|r| * |x|)$, space: $O(|r|)$
- RE to NFA, NFA to DFA, simulate DFA
  - time: $O(|x|)$, space: $O(2^{|r|})$
- Lazy transition evaluation
  - transitions are computed as needed at run time; computed transitions are stored in cache for later use

Flex – Lexical Analyzer Generator

A *language* for specifying lexical analyzers



Flex Programs

```
%{
auxiliary declarations
%}
regular definitions
%%
translation rules
%%
auxiliary procedures
```

Translation Rules

P_1 action$_1$

P_2 action$_2$

…

P_n action$_n$

where P_i are regular expressions and action$_i$ are C program segments

An Example

```
%%
username    printf( "%s", getlogin() );
```

By default, any text not matched by a flex lexical analyzer is copied to the output. This lexical analyzer copies its input file to its output with each occurrence of "username" being replaced with the user's login name.

An Example

```
%{
        int num_lines = 0, num_chars = 0;
%}
%%
\n      ++num_lines; ++num_chars;
.       ++num_chars;        /* all characters except \n */
%%
main() {
        yylex();
        printf("lines = %d, chars = %d\n",
                num_lines, num_chars);
}
```

An Example

```
%{
#define EOF        0
#define LE         25
#define EQ         26
 ...
%}
delim       [ \t\n]
ws          {delim}+
letter      [A-Za-z]
digit       [0-9]
id          {letter}({letter}|{digit})*
number      {digit}+(\.{digit}+)?(E[+\-]?{digit}+)?
%%
```

An Example

```
{ws}        { /* no action and no return */ }
if          {return (IF);}
else        {return (ELSE);}
{id}        {yylval=install_id(); return (ID);}
{number}    {yylval=install_num(); return (NUMBER);}
“<=”        {yylval=LE; return (RELOP);}
“==”        {yylval=EQ; return (RELOP);}
 ...
<<EOF>>     {return(EOF);}
%%
install_id() { ... }
install_num() { ... }
```

Functions and Variables

yylex()

a function implementing the lexical analyzer and returning the token matched

yytext

a global pointer variable pointing to the lexeme matched

yyleng

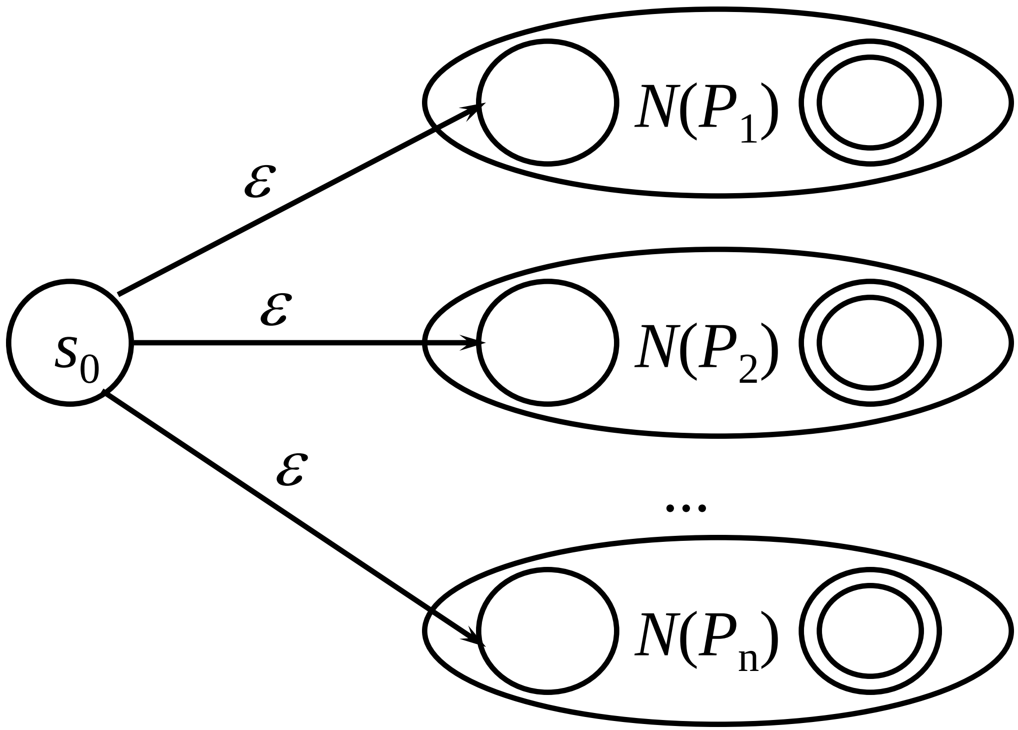
a global variable giving the length of the lexeme matched

yylval

an external global variable storing the attribute of the token

NFA from Flex Programs

$$P_1 \mid P_2 \mid \ldots \mid P_{\mathrm{n}}$$



Rules

- Look for the longest lexeme
  - number
- Look for the first-listed pattern that matches the longest lexeme
  - keywords and identifiers
- List frequently occurring patterns first
  - white space

Rules

- View keywords as exceptions to the rule of identifiers
  - construct a keyword table
- Lookahead operator: r_1/r_2 - match a string in r_1 only if followed by a string in r_2
  - DO 5 I = 1. 25
    DO 5 I = 1, 25
    DO/({letter}|{digit})* = ({letter}|{digit})*,

Rules

- Start condition: <*s*>*r* – match *r* only in start condition *s*

  <str>[^”]*   {/* eat up string body */}

- Start conditions are declared in the first section using either %s or %x

  %s str

- A start condition is activated using the BEGIN action

  \”   BEGIN(str);

- The default start condition is INITIAL

Lexical Error Recovery

- Error: none of patterns matches a prefix of the remaining input
- Panic mode error recovery
  - delete successive characters from the remaining input until the pattern-matching can continue
- Error repair:
  - delete an extraneous character
  - insert a missing character
  - replace an incorrect character
  - transpose two adjacent characters

Maintaining Line Number

- Flex allows to maintain the number of the current line in the global variable yylineno using the following option mechanism

  %option yylineno

  in the first section

共勉

子曰：學而不思則罔，思而不學則殆。

子曰：溫故而知新，可以為師矣。

-- 論語