Lexical Analysis

- elim. white space, comments
- group chars into tokens
- speed is important

Lexeme:

string of chars matched for a token

Token:

datastructure containing token type and value

Types of tokens

Values

1, 3.14, true, 'c', "abc", ...

Identifiers

x, y2, x42, ...

Keywords

if, while, ...

Symbols

+, <, <=, ;, ...

Example

Input:

```
x = y * 5;
```

Output:

```
ID(x), ASSIGN, ID(y),
MUL, INTCONST(5), SEM
```

Idea

- describe lexemes as regular expressions (REs)
- translate REs into NFA
- translate NFA into DFA
- implement table-driven DFA
- use JLex to translate REs into DFA

Regular Expressions

symbols	a
alternation	a|b
concatenation	ab
repetition	a*
parentheses	(a|b)
nothing	ε

Example:

Identifier:

(a|...|z|A|..|Z)
(a|...|z|A|...|Z|0|..|9|_|$)*

Abbreviations

<table>
<tr><td>ab|c</td><td>(ab)|c</td></tr>
<tr><td>[abcd]</td><td>(a|b|c|d)</td></tr>
<tr><td>[a-z]</td><td>(a|...|z)</td></tr>
<tr><td>[~X]</td><td>anything but X</td></tr>
<tr><td>X?</td><td>X|ε</td></tr>
<tr><td>X+</td><td>X(X*)</td></tr>
<tr><td>"+"</td><td>the string itself</td></tr>
<tr><td>+</td><td>"+"</td></tr>
<tr><td>.</td><td>anything but \n</td></tr>
</table>

Compiler

ErrorMsg/ErrorMsg.java

Parse/Lexer.java

Parse/Main.java

Parse/sym.java

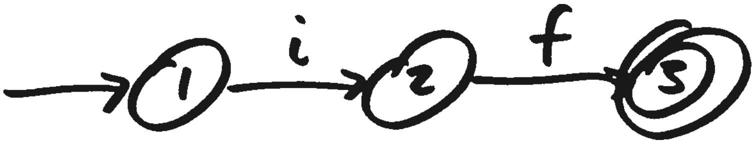
Parse/Tiger.lex

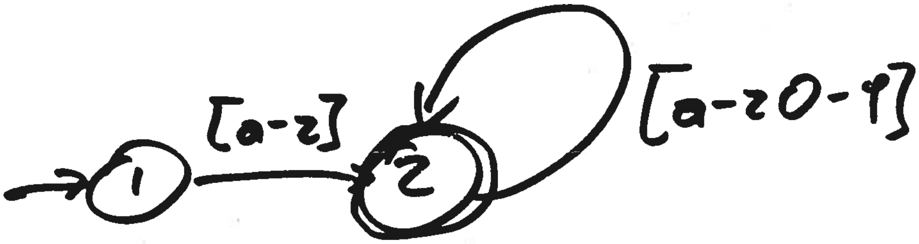
JLex Source

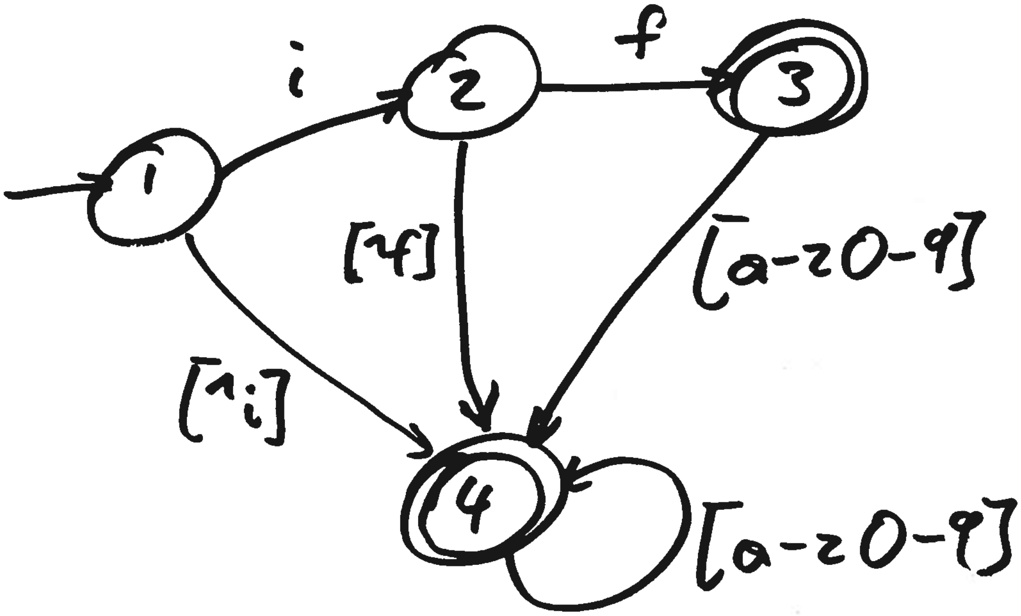
```
package Parse;
import ErrorMsg.ErrorMsg;

%%
%{
   ⋮
%}
%function nextToken
      ⋮
digits = [0-9]+
%%
if                {...}
 ⋮
.                 {...}
```

What's a DFA?







DFA implementation

2-dim table

in state	a	b	c	d	e	. . .
0	0	0	0	0	0	
1	2	2	2	.	—	-
2	⋮	⋱				
3	⋮		⋱			
⋮	⋮			⋱		

DFA implementation

```
state =0;
while (¬end-of-file()) {
    switch (state) {
        case 0: - - - - - - - break;
        case 1: - - ; state=17; break;
            ⋮
    }
}
```

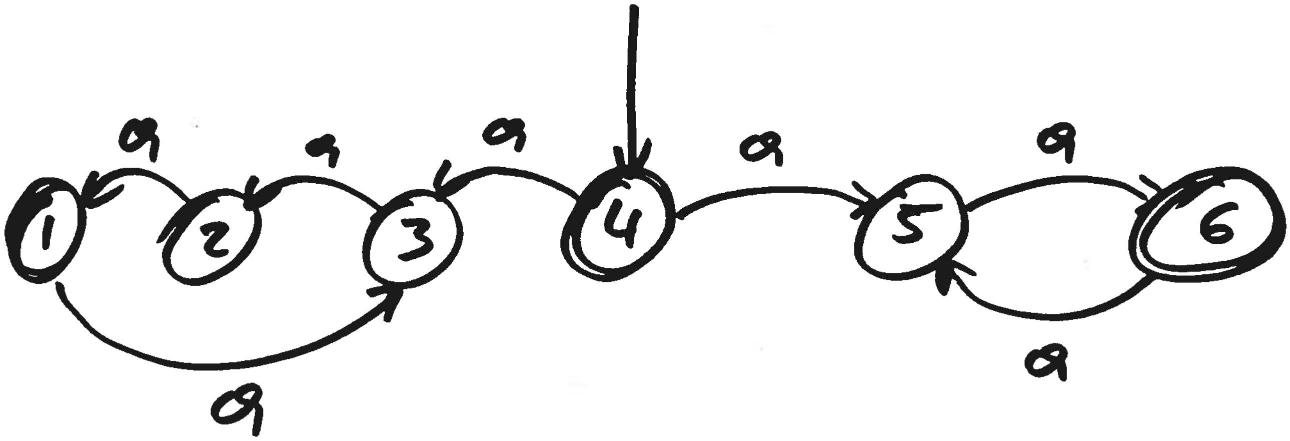
DFA implementation

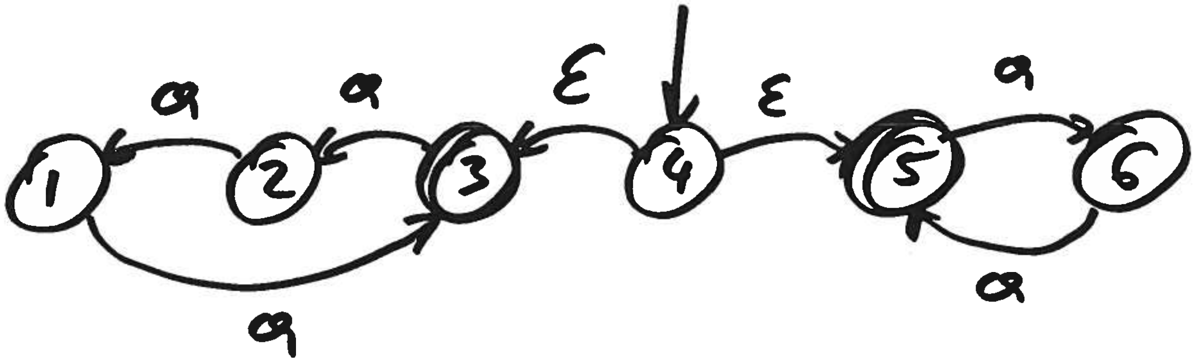
```
state0:
         . _ _ _ _
         goto state17;
state1:
         . _ _ _
         goto state1;

   .
   .
   .

   .
 end:
```

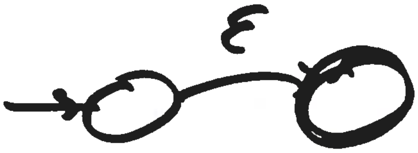
What's an NFA?



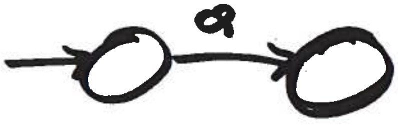


RE → NFA (Thompson's construction)

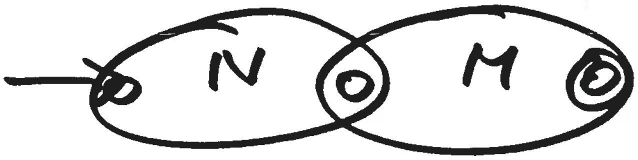
ε



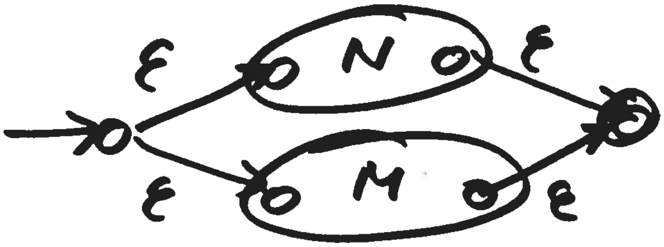
a



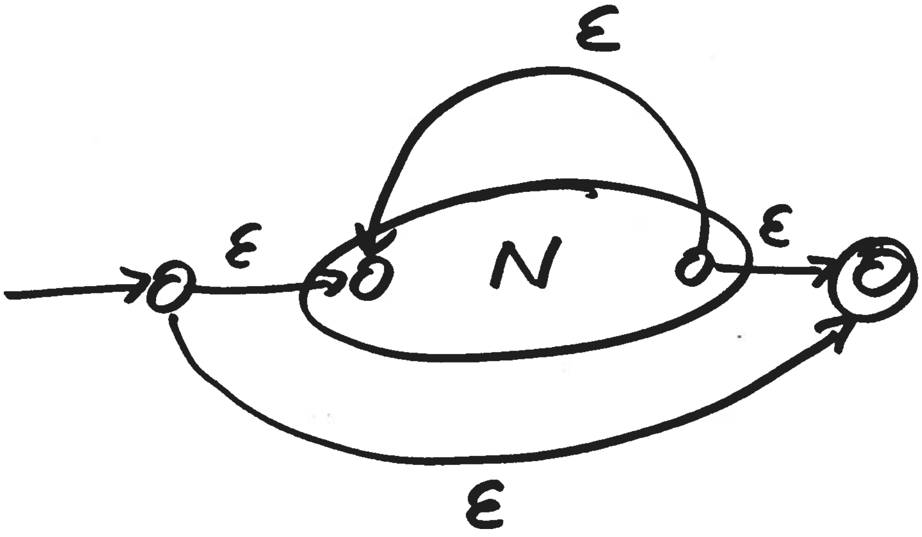
$N \cdot M$



$N|M$



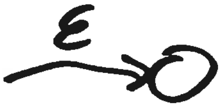
$N*$



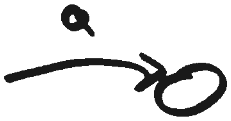
RE → NFA (Appel's construction)

p.26

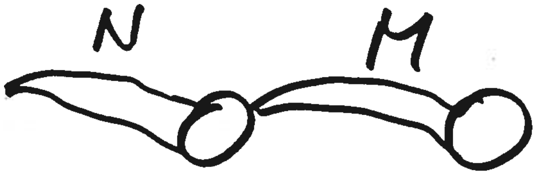
ε



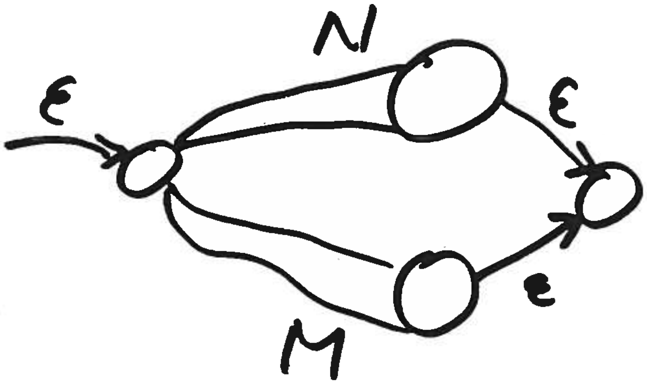
a



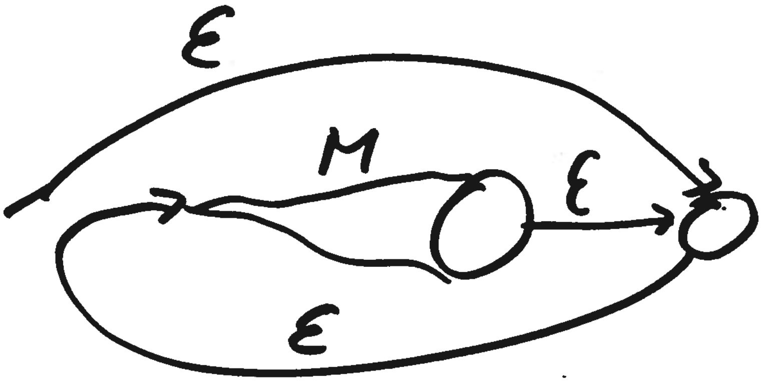
NM



N|M

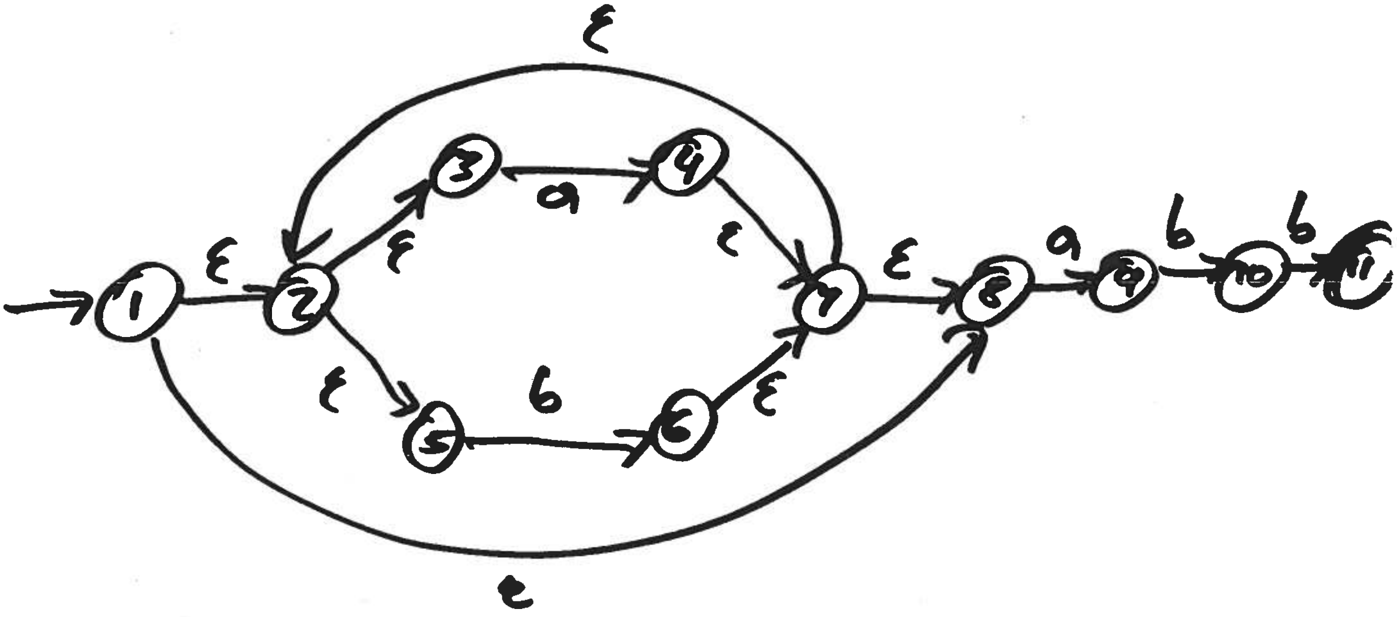


M*

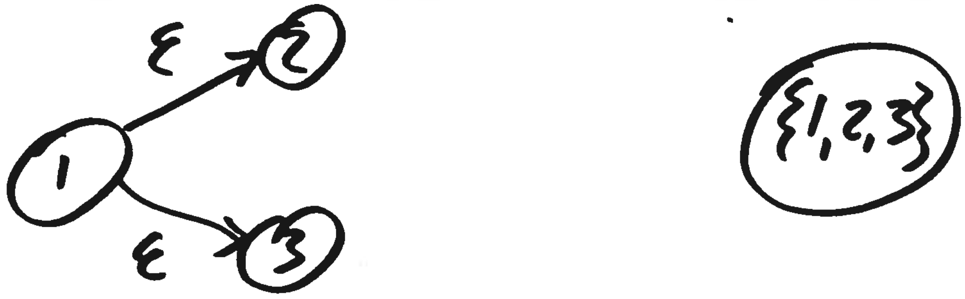


Example

(a|b)* a b b



NFA → DFA

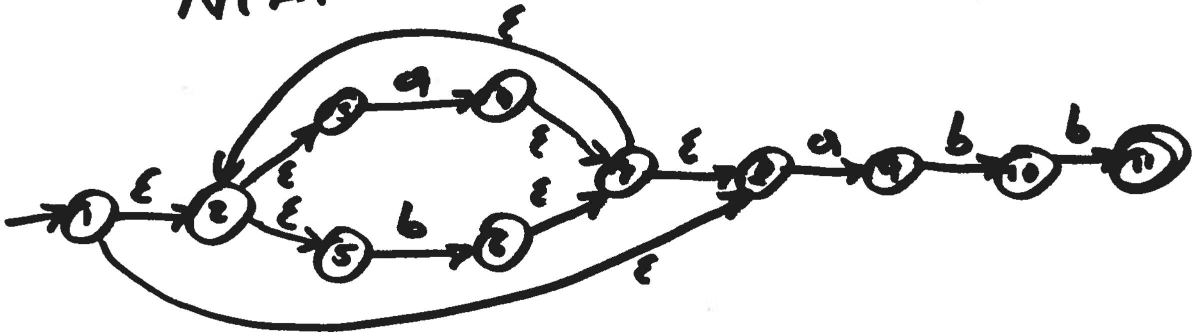


set of NFA states = DFA state

NFA → DFA Translation

RE: (a|b)*abb

NFA:



DFA:

$A = \{\underline{1}, 2, 3, 5, 8\}$

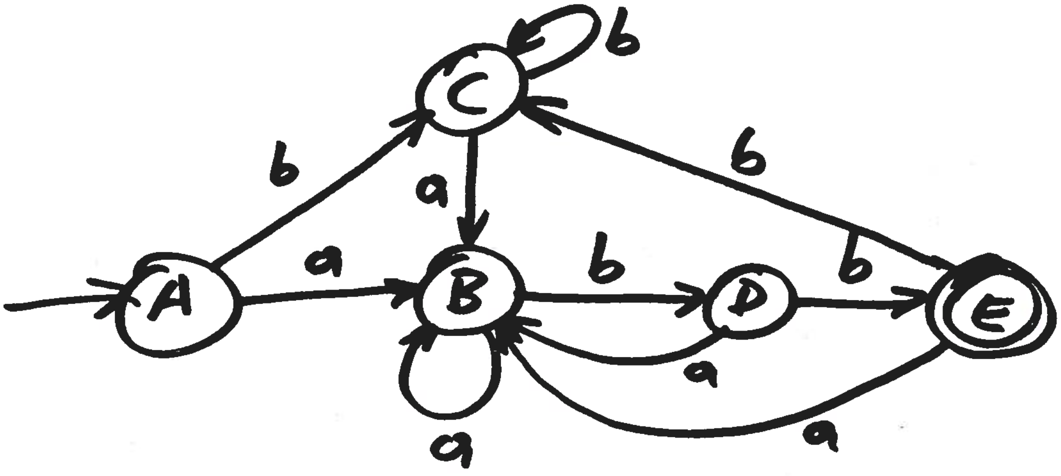
$B = \{\underline{4}, \underline{9}, 2, 3, 5, 7, 8\}$

$C = \{\underline{6}, 2, 3, 5, 7, 8\}$

$D = \{\underline{6}, \underline{10}, 2, 3, 5, 7, 8\}$

$E = \{\underline{6}, \underline{11}, 2, 3, 5, 7, 8\}$

	a	b
A	B	C
B	B	D
C	B	C
D	B	E
E	B	C



NFA → DFA Algorithm

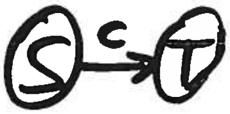
ε-closure:

In: set of states S

Out: set of states that can be reached with ε-edges from S

DFA-Edge

In: set of states S
input symbol c

Out: set of states T, s.t. 

From states in S:

- follow all transitions on c
- then calculate ε-closure

NFA → DFA Algorithm (cont.)

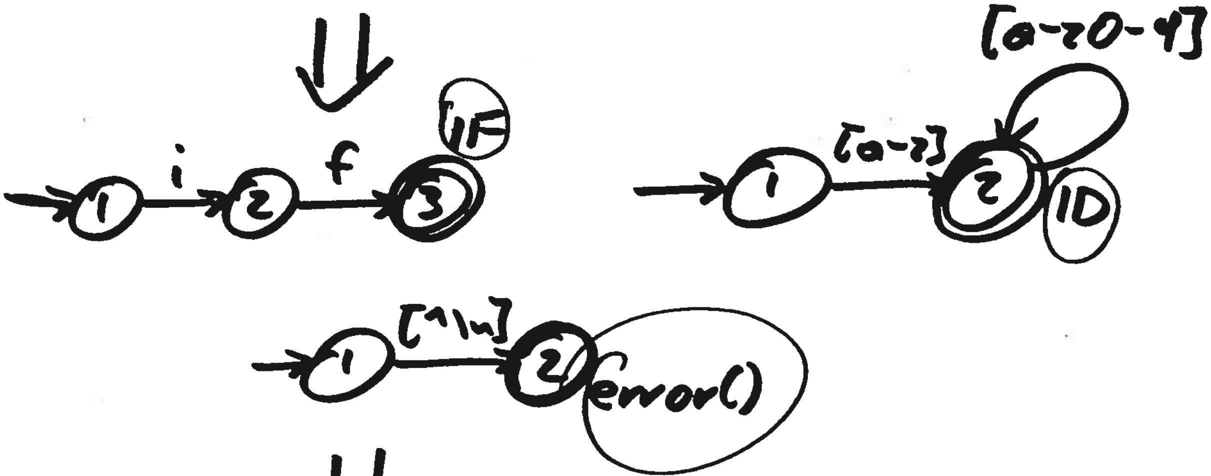
```
start-state of DFA =
    ε-closure(start state of NFA);

loop
    pick DFA state S and input c;
    T = DFA-Edge(S, c);
    if (T didn't exist yet)
        add state T to DFA;
    add edge $\xrightarrow{c}$ from S to T;
until(no more edge can be added)
```

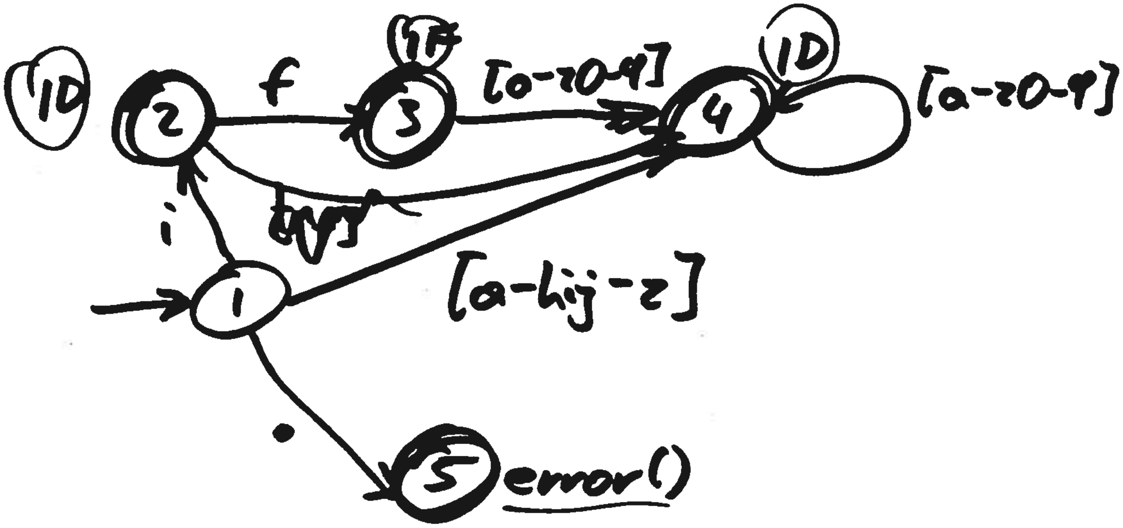
JLex Translation

```
if                 {return IF;}
[a-z][a-z0-9]*     {return ID;}
.                  {error();}
```

⇓



⇓



⇓

DFA

⇓

Tables

JLex Strategy

- rule priority

  rule for ID has to be after keywords

  if
  ⋮
  {ident}

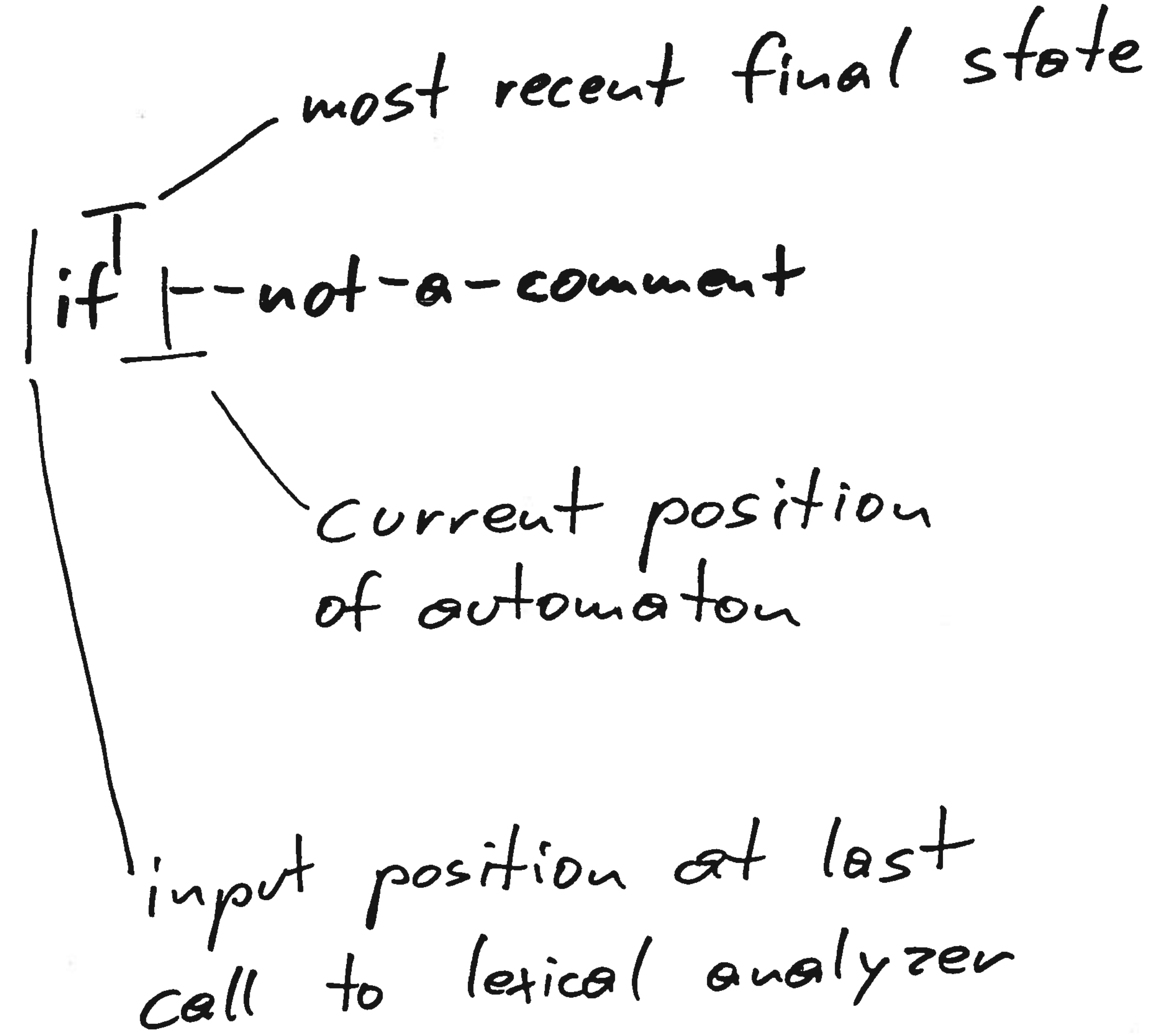
- longest match

  the rule that matches more characters wins

  "<="
  "<"

Recognizing the Longest Match

Example from p. 24


most recent final state
if --not-a-comment
current position of automaton
input position at last call to lexical analyzer

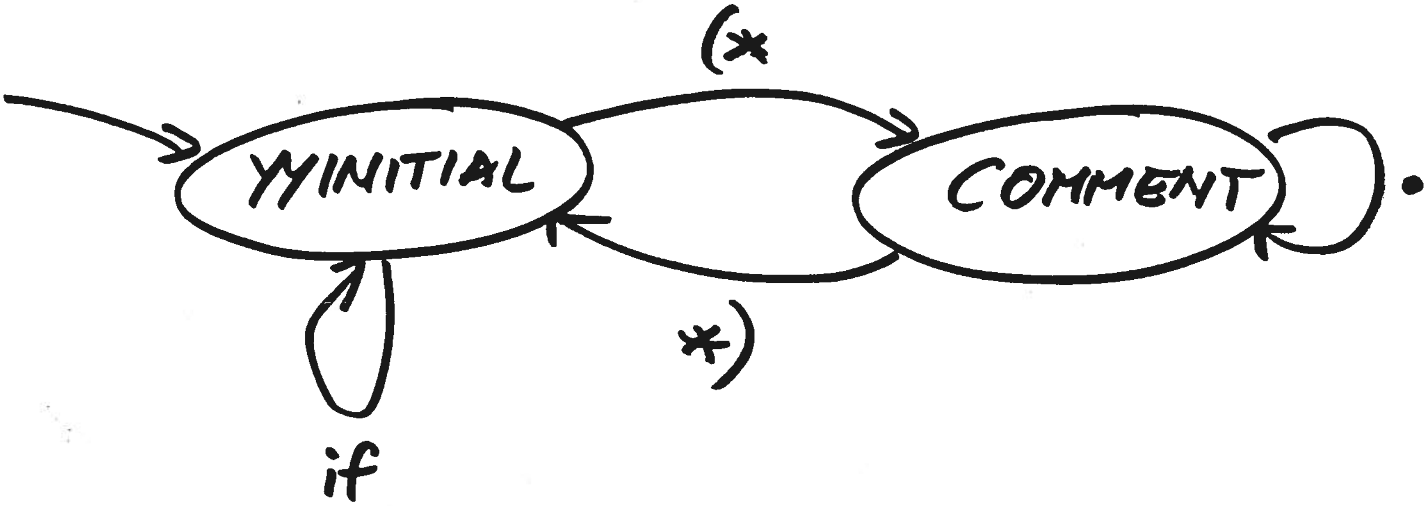
Start States

- allow breaking up the recognition of a token into multiple REs
- allow additional computation for complicated input

Example (p. 33):

```
%state COMMENT
%%
<YYINITIAL> if       {...}
<YYINITIAL> "(*"     {yybegin(COMMENT);
<COMMENT> "*)"    {yybegin(YYINITIAL);}
<COMMENT> .       { }
```

Start States (cont.)



A RE not prefixed by a <STATE> operates in all states.

DFA Optimization

Given:

	a	b
A	B	C
B	B	D
C	B	C
D	B	E
E	B	C

Find: optimized table

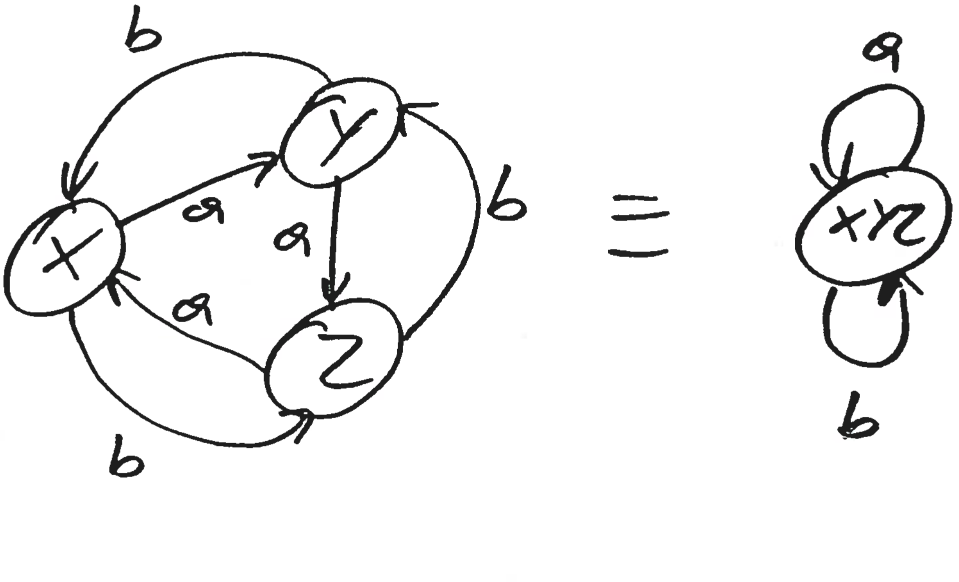
E.g.: A and C look the same

Idea: combine identical rows

works for A and C above

Doesn't work for

	a	b
X	Y	Z
Y	Z	X
Z	X	Y



DFA Optimization Algorithm

- Combine all final states into one.
- Combine all non-final states into one.
- Split a group of states that violates the grouping.
- Repeat the previous state until no more splits necessary.

Example

	a	b
A	ABCD	ABCD
B	ABCD	ABCD
C	ABCD	ABCD
D	ABCD	E
E	ABCD	ABCD

split

	a	b
A	ABC	ABC
B	ABC	D
C	ABC	ABC
D	ABC	E
E	ABC	ABC

split off

	a	b
AC	B	AC
B	B	D
D	B	E
E	B	AC

done

Solution:

$(a|b)^* abb$

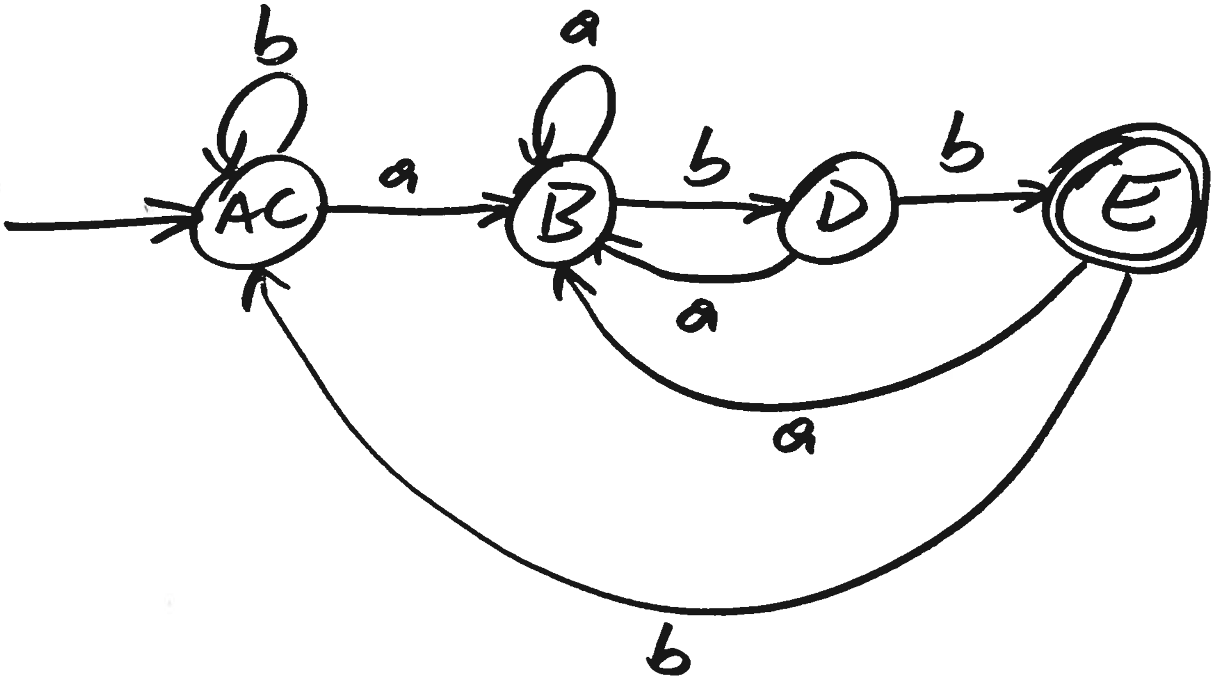
$\Downarrow$

NFA

$\Downarrow$

DFA

$\Downarrow$



Def.: Language

Given: an alphabet
(e.g., ASCII)

A language is the set of all valid strings over the alphabet.

Example Languages

$(a \mid b)^* abb$	$\{abb, aabb, babb, aaabb, ababb, baabb, bbabb, \ldots\}$
$0 \mid [1-9][0-9]^*$	$\mathbb{N}$
?	Java
?	English

Classification of Languages

(by Noam Chomsky, MIT)

language	tool	use
regular	RE	scanning
context-free	BNF	parsing
context-sensitive	rewrite systems	sem. anal.
unrestricted	Turing Machine	—

Limitations of Languages

"Regular lang. can't count."

$a^n b^n$ not regular

/*../*...*/..*/

"Context-free languages can't remember counts."

$a^n b^n c^n$ not context-free

```
int foo(int, int);
i=foo(1, 2);
j=foo(3, 4);
```

Difficult Scanning Problems

- nested comments ($a^n b^n$)
- strings with escape chars.
- PL/I :

```
IF IF = THEN THEN THEN = ELSE
```

- FORTRAN:

vs. 1,10 for range

```
DO 20 I = 1. 10
   ⋮
20 CONTINUE
```

- C++:

```
C x (int);    forward decl. of meth
C y (5);      decl of obj y constructor
C z (a);      is a a type or a variable
```

Summary

- lex. anal. split from parsing to make parser simpler
- set of valid lexemes is a regular language
- lexemes described by REs
- REs translated to NFA
- NFA translated to DFA
- DFA optimized
- DFA implemented with tables
- RE → NFA → DFA → opt. DFA → tables automated with lex/flex/JLex
- JLex uses rule priority/longest match
- JLex offers start states for scanning non-regular constructs