Lexical Analysis

Chapter 2

Lexical Analysis Tasks

- Eliminate white space and comments
- Group characters into tokens
- Speed is important

Terminology

- Lexeme:

  String of input characters matched for a token

- Token:

  Data structure containing token type and value

Types of Tokens

- Values

  `1`, `3.14`, `true`, `'c'`, `"abc"`, …

- Identifiers

  `x`, `yz`, `x42`,

- Keywords

  `if`, `while`, …

- Symbols

  `+`, `<`, `<=`, `;`, …

Token Examples

- Input:

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

- Output:

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

Construction of Lexical Analyzer

- Describe lexemes as regular expressions (REs)
- Translate REs into non-deterministic finite automaton (NFA)
- Translate NFA into deterministic finite automaton (DFA)
- Implement table-driven DFA
- Use JLex for translating REs into DFA

Regular Expressions

• Symbols	`a`
• Alternation	`a | b`
• Concatenation	`a b`
• Repetition	`a*`
• Parentheses	`(a|b)`
• Empty	ε

Examples of Regular Expressions

- Integer constants

```
0 | ((1|2|3|4|5|6|7|8|9)(0|1|2|3|4|5|6|7|8|9)*)
```

- Identifiers

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

Abbreviations

`ab|c`	`(ab)|c`
`[abcd]`	`(a|b|c|d)`
`[a-z]`	`(a|…|z)`
`[~X]`	any character other than `X`
`X?`	`X|`ε
`X+`	`X(X*)`
`"+"`	the string itself
`\+`	`"+"`
`.`	`[~\n]`, i.e., anything but newline

Lexical Analyzer for Tiger

- `ErrorMsg/ErrorMsg.java`
  Keeps track of line count, prints error message

- `Parse/Lexer.java`
  Lexical analyzer interface

- `Parse/Main.java`
  Test program that calls lexical analyzer and prints tokens

- `Parse/sym.java`
  Enumeration constants for tokens, generated by JavaCUP

- `Parse/Tiger.lex`
  JLex source code for lexical analyzer

JLex Source

```
package Parse;
import ErrorMsg.ErrorMsg;
%%
%{
...
%}
%function nextToken
...
digits=[0-9]+
%%
if     { return new Token(IF); }
...    ...
.      { error("Illegal character"); }
```

JLex Strategy

- Longest match
  - The rule that matches the most characters wins
  - Example

    ```
    "<"
    "<="
    ```

  - If input is <=, it will be matched as a single lexeme

- Rule priority
  - If multiple rules match the same input, the first rule wins
  - Example

    ```
    if
    {Ident}
    ```

  - The rule for identifiers has to be after all the keywords

Recognizing the Longest Match

- Example from textbook

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
  - Example: nested comments cannot be described with a single RE

- Allow additional computation for complicated input
  - Example: translate escape character sequences in strings

Start State Example

- Recognizing a single comment with multiple REs

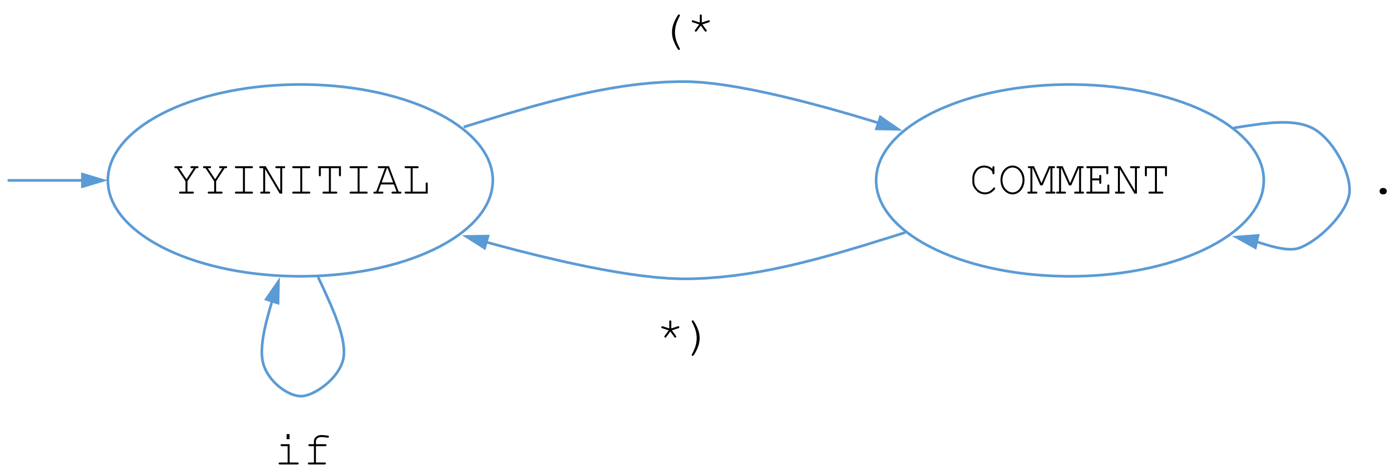
```
%state COMMENT

%%

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

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

Start States as a Higher-level Automaton



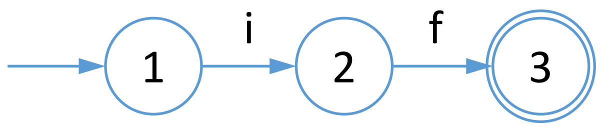
Escape Character Sequences

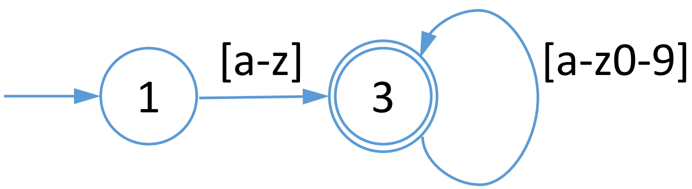
- Escape character sequences in strings must be translated
- E.g., the two character lexeme `\\n` should be translated into the newline or Line Feed character with ASCII code 10

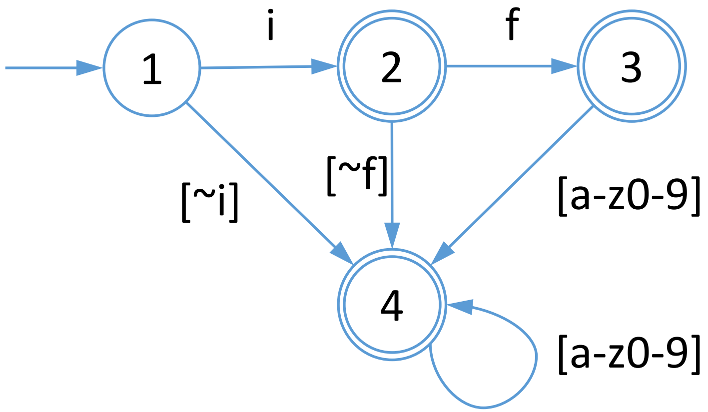
- Implementation for strings
  - Enter `STRING` state with initial double quote, initialize string buffer
  - Any escape character sequence is translated and added to buffer
  - Other characters are added without translation
  - Exit `STRING` state with final double quote and return token

- See the Wikipedia page on the ASCII character encoding
- Different languages have different escape character sequences

What’s a DFA?







DFA Implementation as Table

- 2-dimensional table

State	a	b	c	d	e	f	...
0	0	0	0	0	0	0	
1	2	2	2	...			
2	...						
3	...						

DFA Implementation with Loop and Switch

- Code with state variable

```
state = 0;
while (! end_of_file()) {
    switch (state) {
        case 0: ...;  break;
        case 1: ...;  state = 17;  break;
        ...
    }
}
```

DFA Implementation with Goto

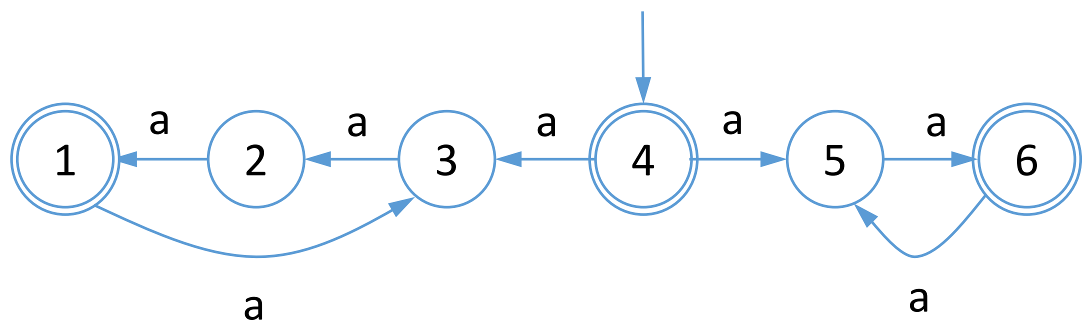
- Or simply

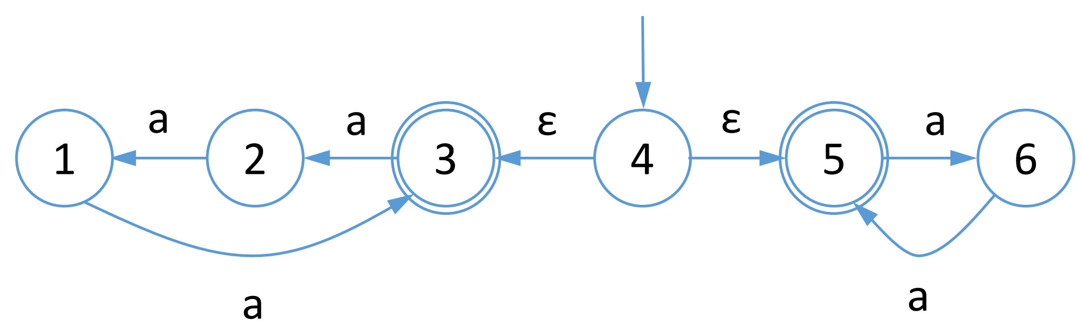
```
state0:
    ...;
    goto state17;
state1:
    ...;
    goto state1;
...

end:
```

What's an NFA?

- Multiple outgoing edges on the same input
- State transitions that don't consume any input (epsilon edges)



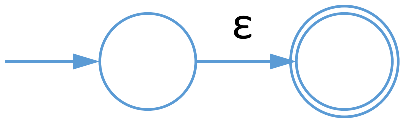


NFA Behavior and Implementation

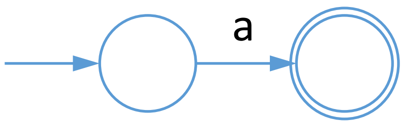
- Semantically: non-deterministic (random) choice of edge to follow
- Imagine an Oracle that knows which choice will lead to a final state
- Implementation: follow all states, keep track of set of possible states

RE->NFA (Thompson's Construction)

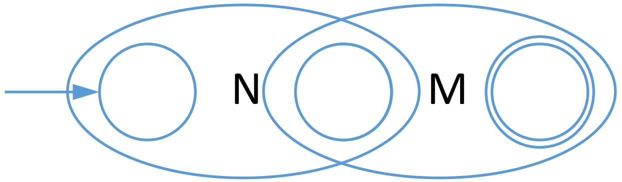
- ε



- a

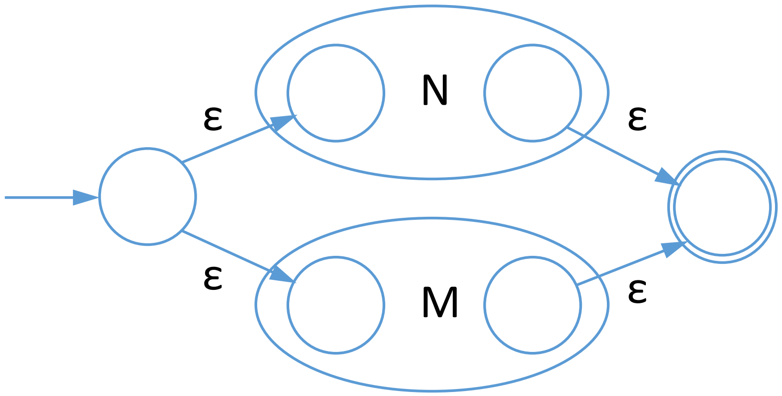


- N M

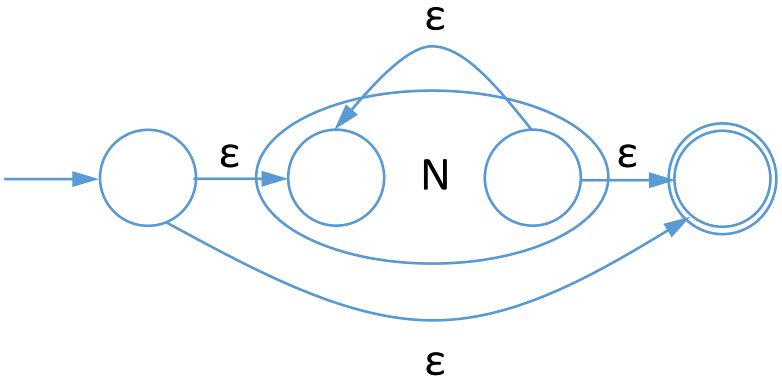


RE->NFA (Thompson’s Construction), cont’d

- N | M



- N *



Thompson’s vs. Appel’s Construction

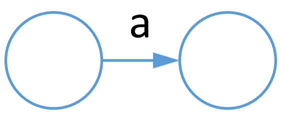
- For Appel’s RE->NFA algorithm, see Fig. 2.6 on p. 26 of textbook

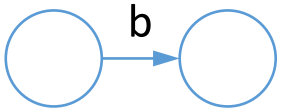
- Thompson translates each subexpression into NFA with a start state and one final state
- Appel has an incoming labeled edge instead of a start state, adds start state for finished NFA at the very end

- Tompson’s construction is easier and works even if sub-automata were constructed by hand
- Appel’s construction results in fewer states and fewer ε-transitions

Example for Thompson's Construction, 1/5

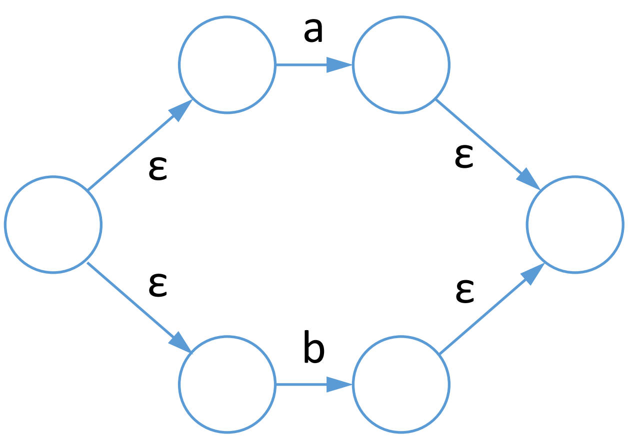
- (a|b)*abb





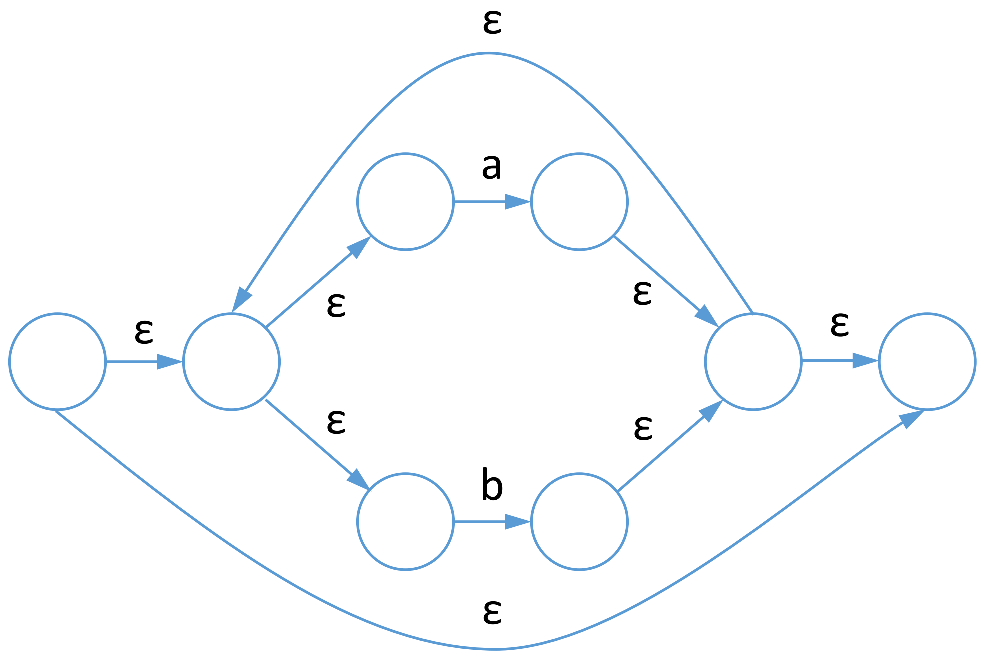
Example for Thompson's Construction, 2/5

- (a|b)*abb



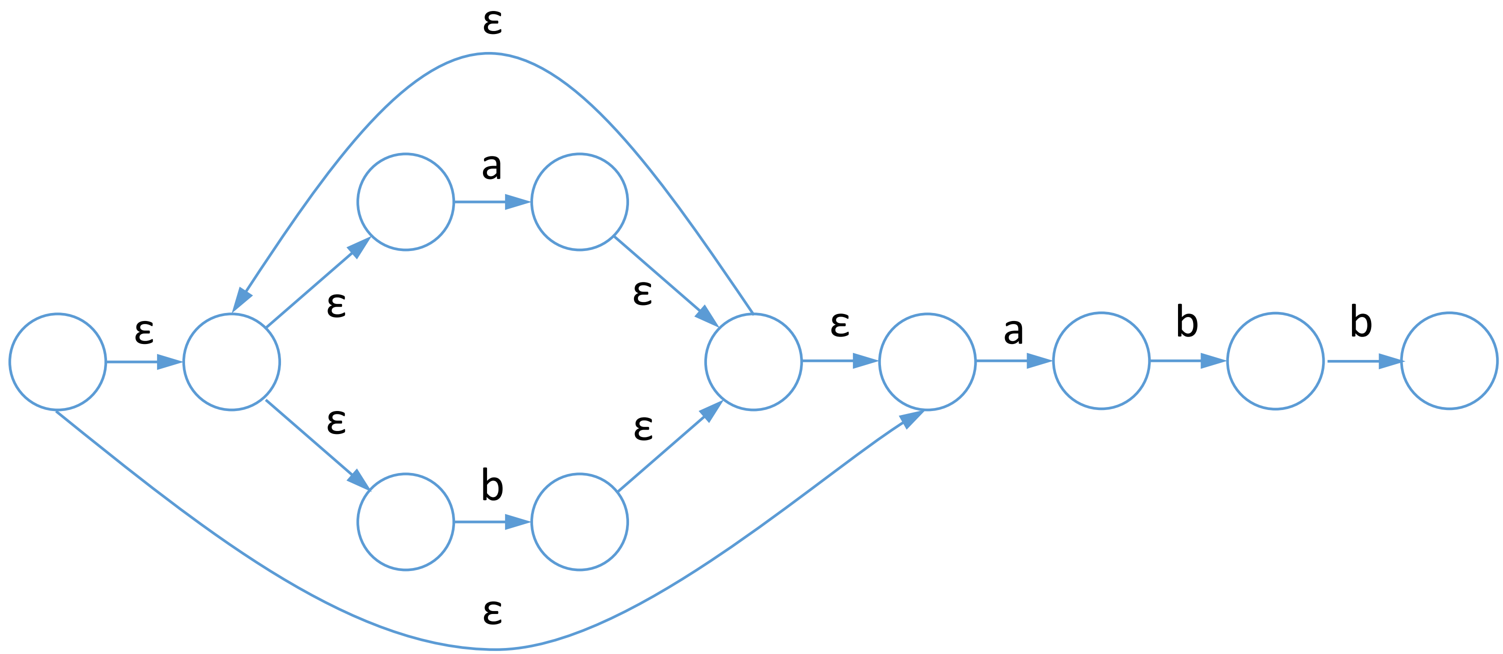
Example for Thompson’s Construction, 3/5

- (a|b)*abb



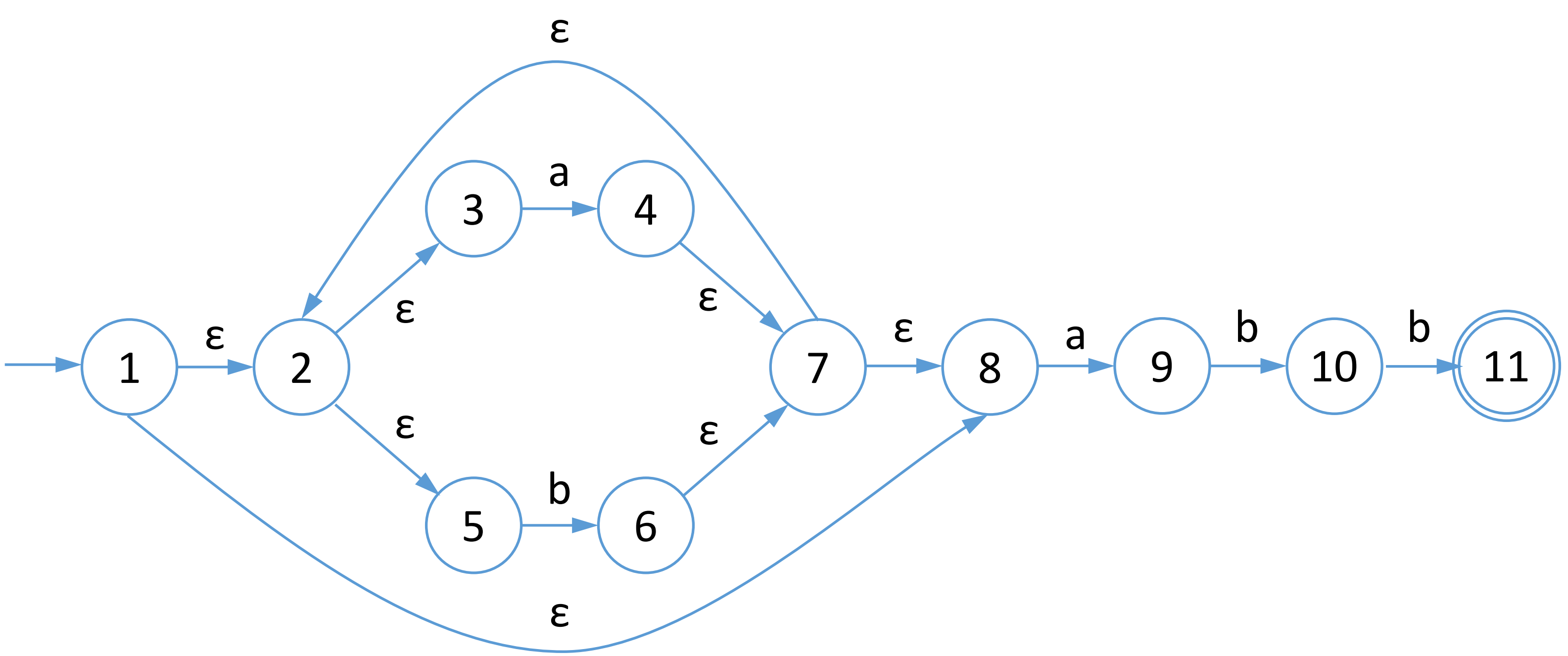
Example for Thompson's Construction, 4/5

- (a|b)*abb



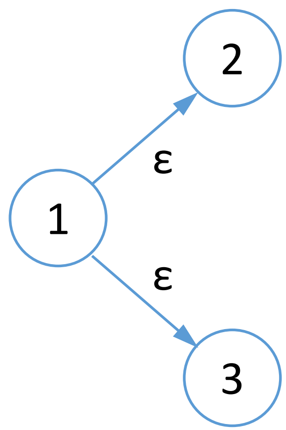
Example for Thompson's Construction, 5/5

- (a|b)*abb

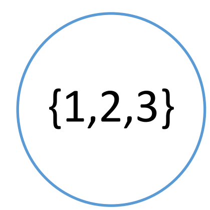


NFA->DFA (Subset Construction)

- Main idea: a set of NFA states corresponds to a single DFA state



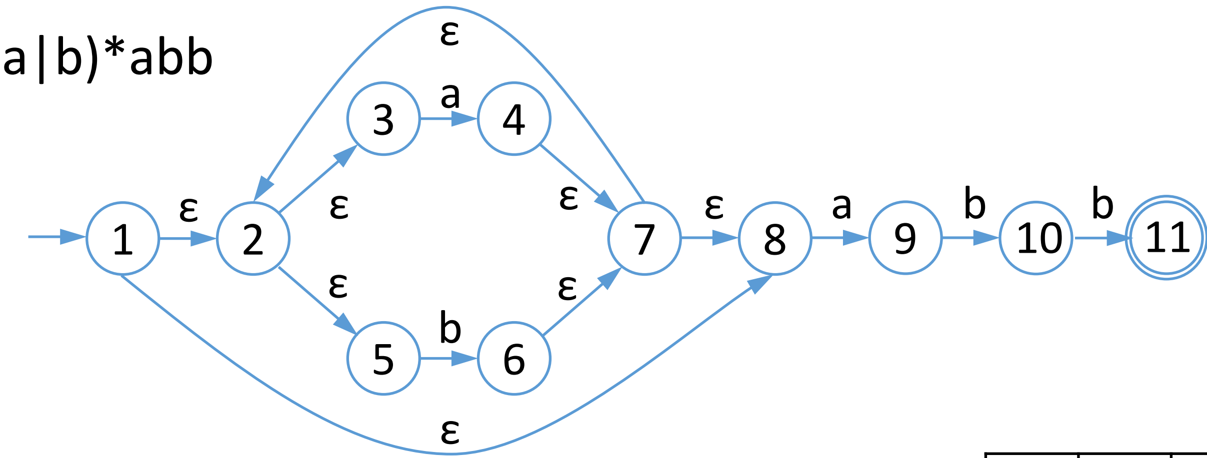
Set of NFA states



DFA state

Example for NFA->DFA Translation, 1/6

(a|b)*abb

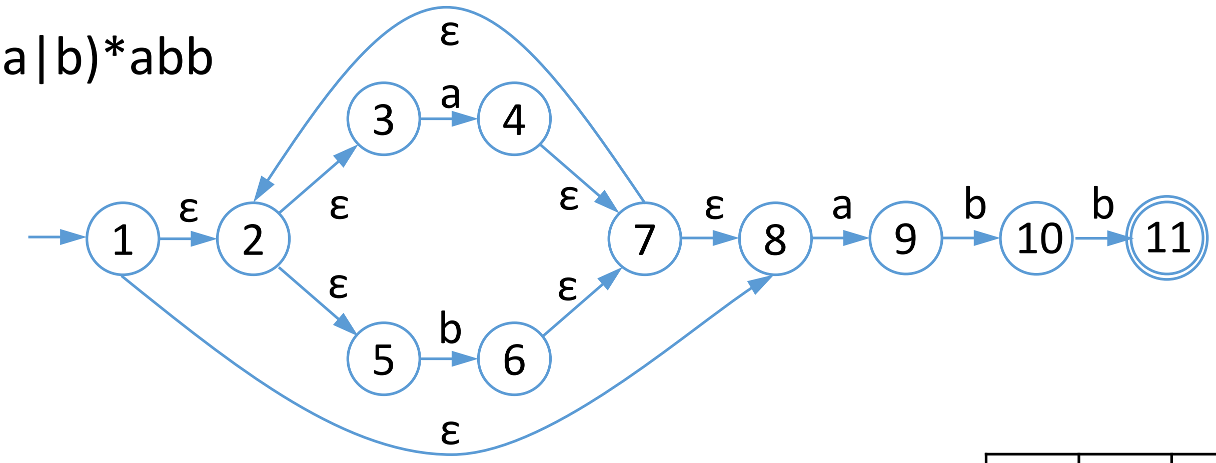


A = {<u>1</u>, 2, 3, 5, 8}

	a	**b**
A		

Example for NFA->DFA Translation, 2/6

(a|b)*abb



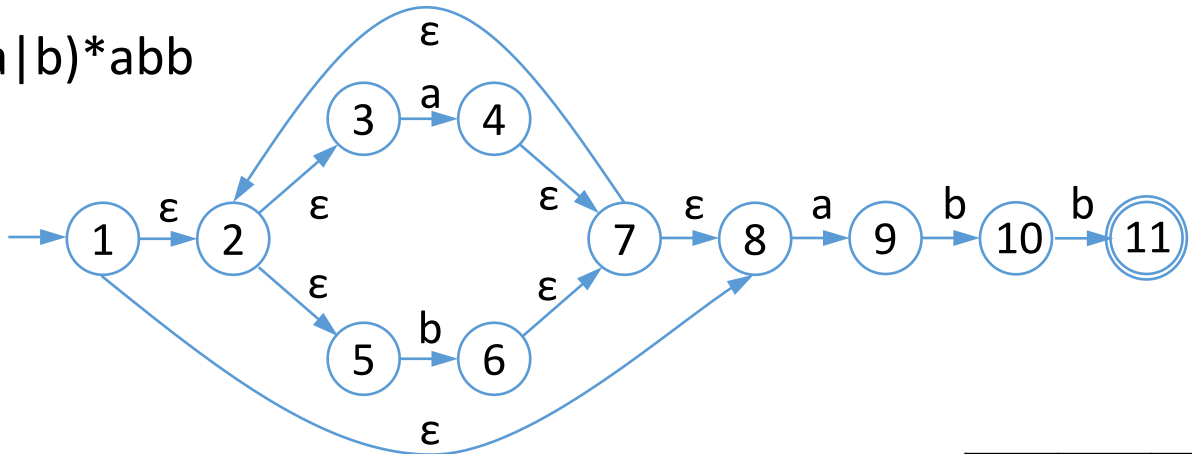
A = {<u>1</u>, 2, 3, 5, 8}

B = {<u>4</u>, <u>9</u>, 2, 3, 5, 7, 8}

	a	**b**
A	B	
B		

Example for NFA->DFA Translation, 3/6

(a|b)*abb



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

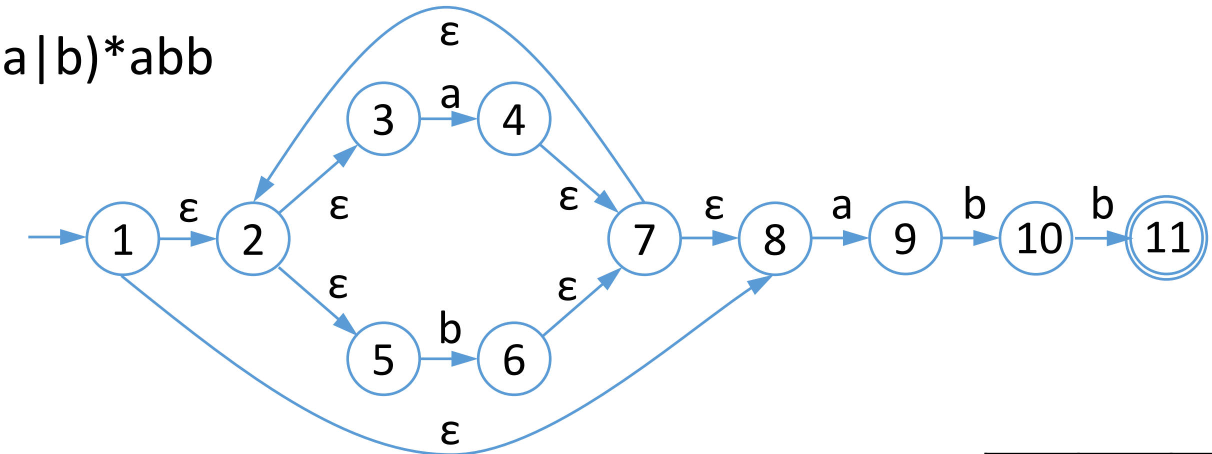
$B = \{\underline{4}, \underline{9}, 2, 3, 5, 7, 8\}$

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

	a	**b**
A	B	C
B		
C		

Example for NFA->DFA Translation, 4/6

(a|b)*abb



A = {1, 2, 3, 5, 8}

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

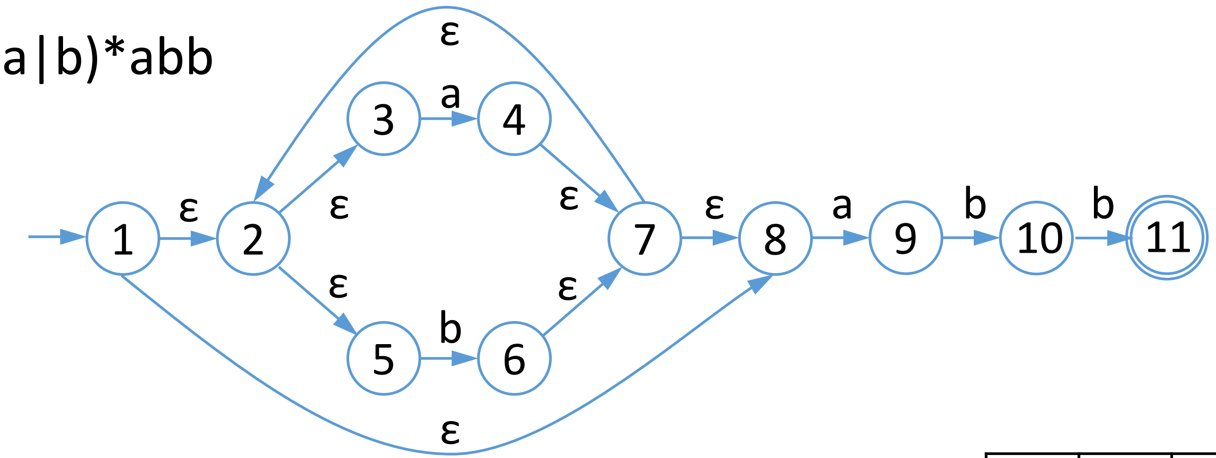
C = {6, 2, 3, 5, 7, 8}

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

	a	**b**
A	B	C
B	B	D
C		
D		

Example for NFA->DFA Translation, 5/6

(a|b)*abb



A = {<u>1</u>, 2, 3, 5, 8}

B = {<u>4</u>, <u>9</u>, 2, 3, 5, 7, 8}

C = {<u>6</u>, 2, 3, 5, 7, 8}

D = {<u>6</u>, <u>10</u>, 2, 3, 5, 7, 8}

E = {<u>6</u>, <u>11</u>, 2, 3, 5, 7, 8}

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

Example for NFA->DFA Translation, 6/6

A = {<u>1</u>, 2, 3, 5, 8}

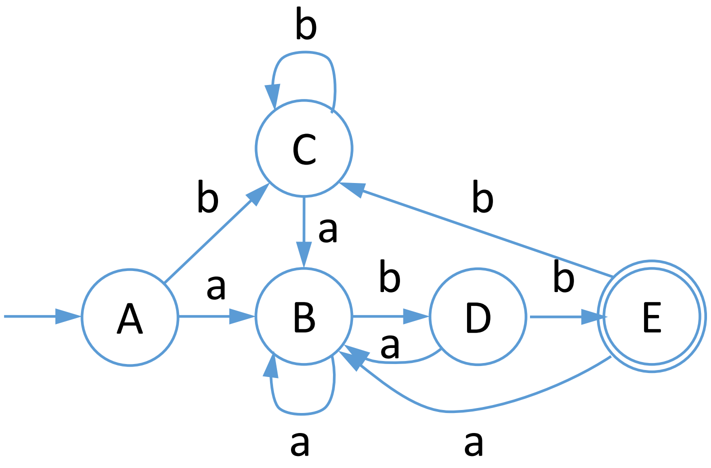
B = {<u>4</u>, <u>9</u>, 2, 3, 5, 7, 8}

C = {<u>6</u>, 2, 3, 5, 7, 8}

D = {<u>6</u>, <u>10</u>, 2, 3, 5, 7, 8}

E = {<u>6</u>, <u>11</u>, 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

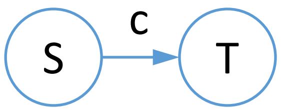
Out: set of states that can be reached with ε-edges from S

DFA-edge:

In: set of states S

input symbol c

Out: set of states T, such that



From each state in S:

follow all transitions on symbol c

then calculate ε-closure

NFA->DFA Algorithm, cont'd

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 labeled c from S to T;
until (no more edge can be added)
```

DFA Optimization

- Given: DFA transition table

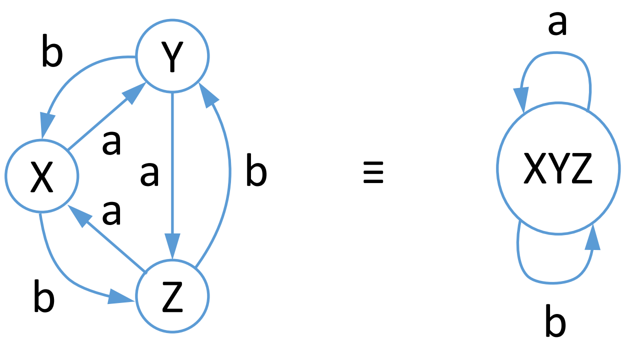
	a	**b**
A	B	C
B	B	D
C	B	C
D	B	E
E	B	C

- Find: optimized table
- E.g., A and C look the same

Idea: Combine Identical Rows

- Works for A and C above
- E is different from A and C because it's a final state
- Doesn't work for:

	a	**b**
X	Y	Z
Y	Z	X
Z	X	Y



DFA Optimization Algorithm

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

Example

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

Split D from ABC

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

Split B from AC

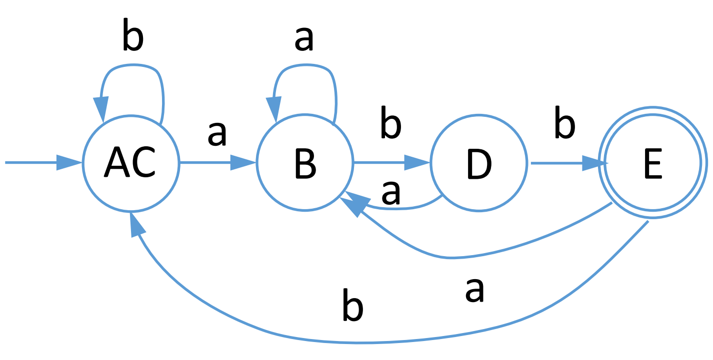
	a	b
AC	B	AC
B	B	D
D	B	E
E	B	AC

Done

Solution

- (a|b)*abb -> NFA
- NFA -> DFA
- DFA is optimized

- Result:



What is a Language?

- Definition should work for programming languages as well as natural languages
- Should be easy enough to explain to non-computer scientists

Language Definition

- Given: an alphabet (e.g., ASCII)
- A language is the set of all valid strings over the alphabet

Example Languages

- (a|b)*abb
- 0 | [1-9][0-9]*
- ?
- ?

- {abb, aabb, babb, aaabb, ababb, baabb, bbabb, … }
- Natural numbers
- Java
- English

Classification of Languages

- By Noam Chomsky, MIT

Language	**Tool**	**Use**
Regular	regular expression	lexical analysis
Context-free	BNF grammar	parsing
Context-sensitive	rewrite systems	semantic analysis
Unrestricted	Turing machine	

Limitations of Languages

- "Regular languages can't count"
  - $a^n b^n$ is not regular
  - E.g., matching parentheses or nested comments

- "Context-free languages can't remember counts"
  - $a^n b^n c^n$ is not context-free
  - E.g., matching number of parameter declarations with number of arguments

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

Scanning Problems in Tiger

- Nested comments ($a^n b^n$)
  - Solution: use start states
- Strings with escape character sequences
  - Is actually regular, but start states help with the translation

Scanning Problems in Other Languages

- PL/I: no distinction between identifiers and keywords

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

  - Solution: let the parser deal with it
- FORTRAN loops

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

  - Bad language design, warn programmers of common errors
- C++ declarations

```
C x(int);    // forward declaration of function x
C y(5);      // Constructor call for object y
C z(a);      // Depends
```

  - Solution: resolve in semantic analysis or feed type info back to lexical analysis

Summary

- Lexical analysis split from parsing to make parsing simpler
- Set of valid lexemes is a regular language
- Lexemes are described using regular expressions
- Translation RE->NFA->DFA->optimized DFA can be automated

- Lexical analyzer generators translate REs to table-driven DFAs
  - E.g., lex, flex, JLex, ML-Lex, etc.
- JLex uses longest-match and rule-priority strategies
- JLex offers start states for scanning non-regular language constructs