# 2. Lexical Analysis

2025 Fall

Hunjun Lee

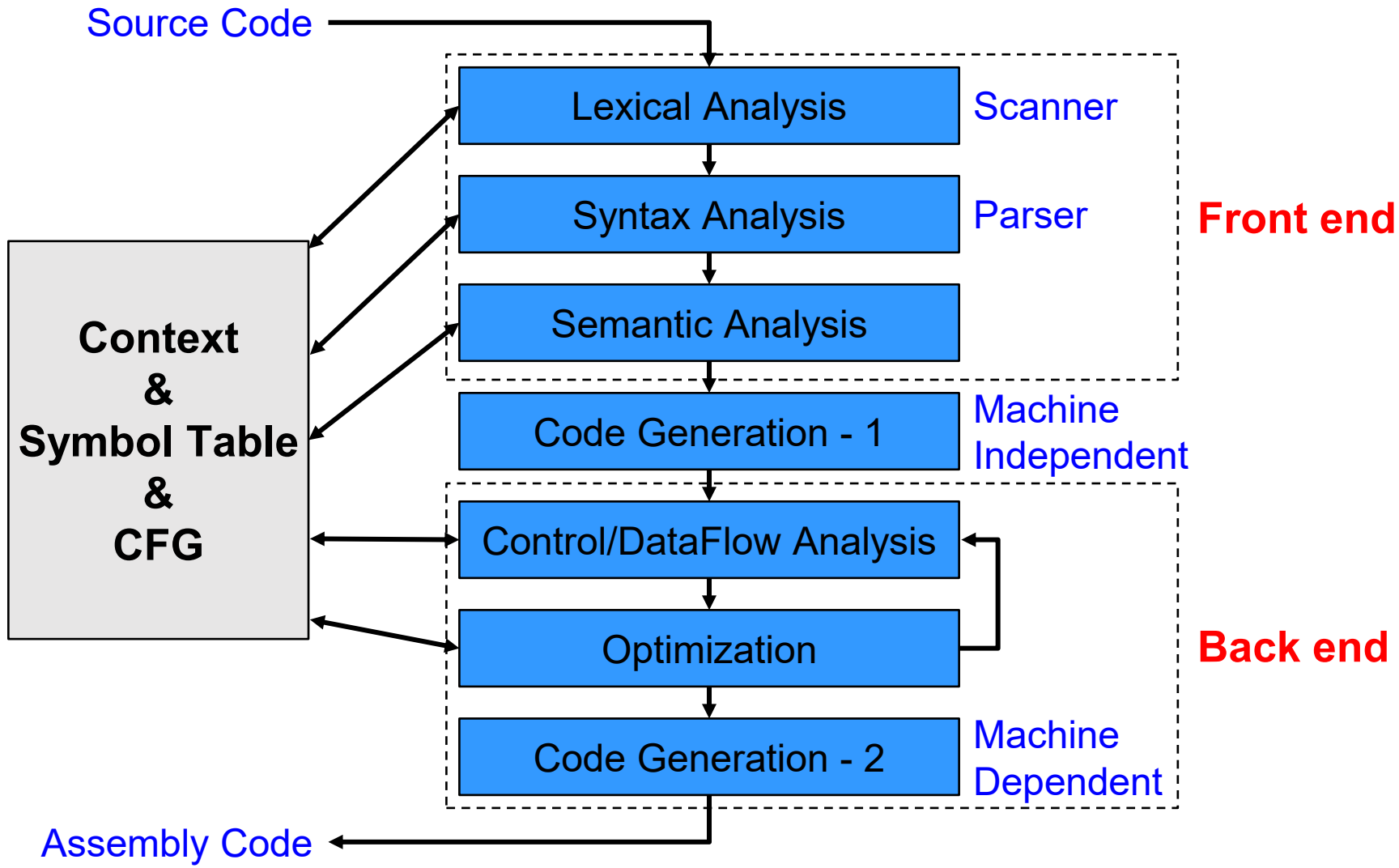
Hanyang University

# Reading

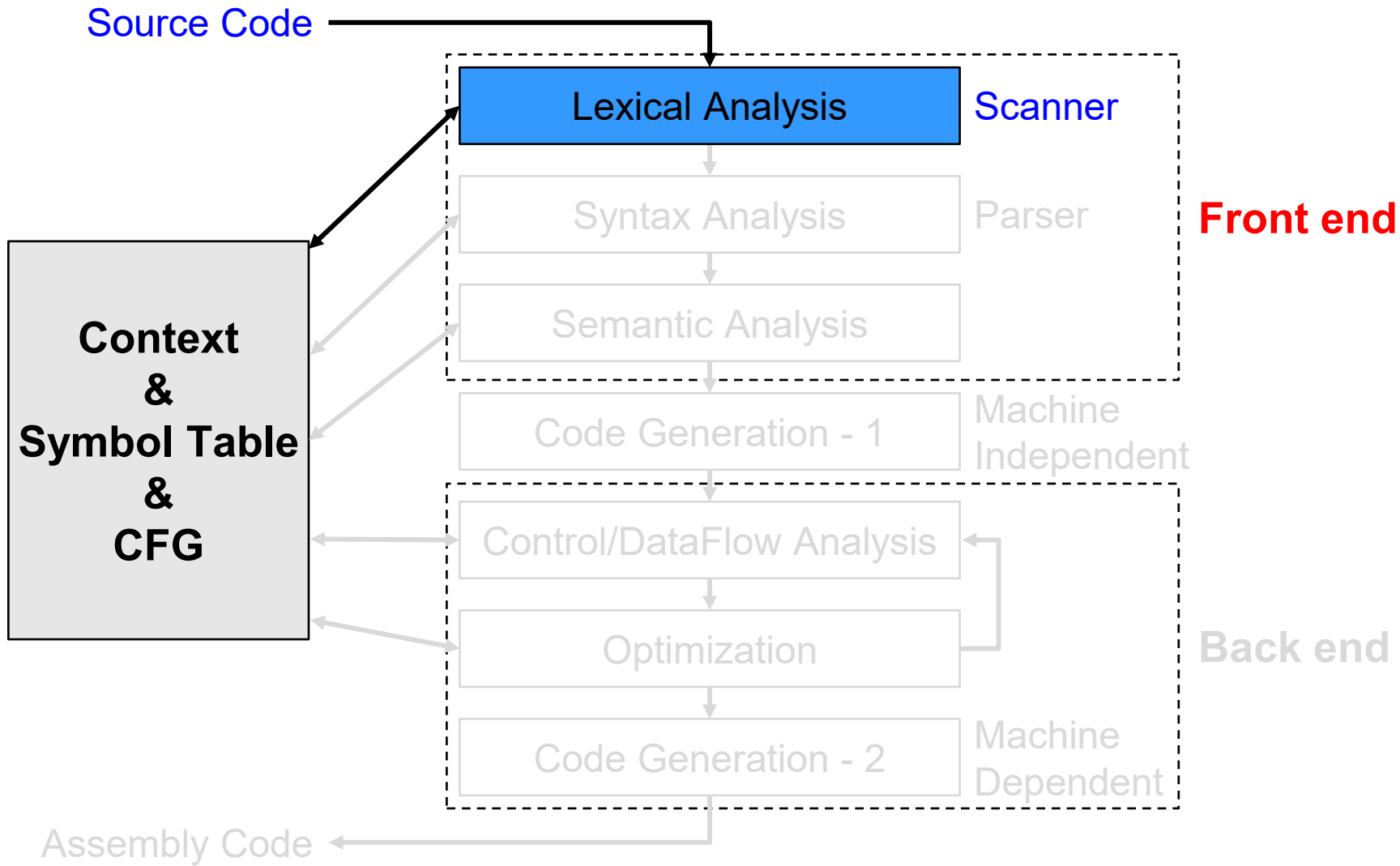
- **Ch2**
  - Understand high-level overview of compiler
  - Do not need to understand the details

- **Ch3**
  - Read carefully and go over examples

# Lexical Analysis

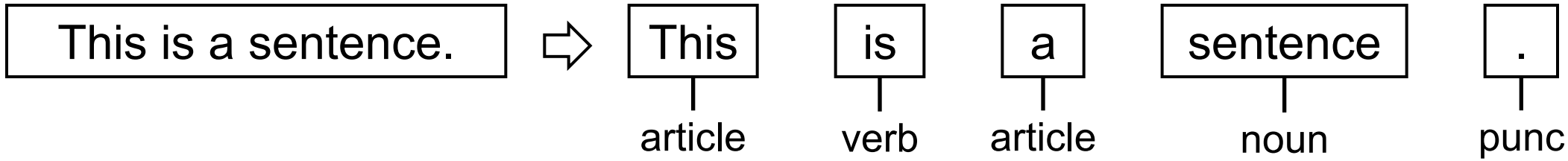


# Lexical Analysis

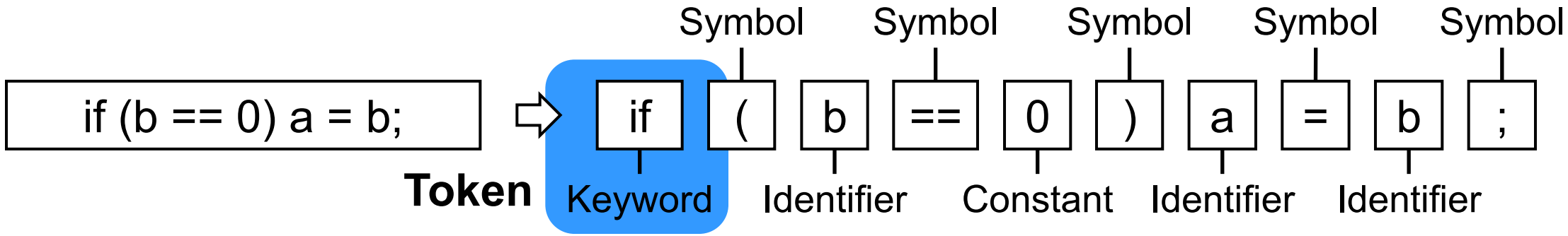


# Lexical Analysis (Scanner)

- **Language: Recognizing words from sentences**




- **Program: Dividing programs into "tokens"**



# Looks Easy, But …

- **Fortran removes all the whitespaces to generate complex examples**

**Identifier**

```
do 5 I = 1.25
```

**Keyword Label Identifier**

```
do 5 I = 1,25
  a = a + 1
end do
```

# Tokens

- Identifiers: x, y11, elsex
- Keywords: if, else, while, for, break
- Integers: 2, 1000, -20
- Floating-points: 2.0, -0.001, .02, 1e5
- Symbols: +, *, {,}, ++, <<, <, <=
- Strings: “x”, “TEST”, “Compiler”

# Specification, Recognition, and Automation

- **Specification: how to specify lexical patterns?**
  - Regular expression (RE)
- **Recognition: how to recognize the specified patterns?**
  - Deterministic finite automata (DFA)
- **Automation: how to generate DFA from RE?**
  - Automatic generation tool (Lex)
  - Thompson's construction (RE -> NFA)
  - Subset construction (NFA -> DFA)

# Specification, Recognition, and Automation

- **Specification: how to specify lexical patterns?**
  - Regular expression (RE)

- **Recognition: how to recognize the specified patterns?**
  - Deterministic finite automata (DFA)

- **Automation: how to generate DFA from RE?**
  - Automatic generation tool (Lex)
  - Thompson's construction (RE -> NFA)
  - Subset construction (NFA -> DFA)

# Regular Expression

- **RE is an inductively defined rules to describe tokens**
  - **a** ordinary character stands for itself
  - **ε** empty string
  - **R|S** either R or S (alteration), where R,S = RE
  - **RS** R followed by S (concatenation)
  - **R*** concatenation of R, 0 or more times (closure)

# Language

- **Indicates a set of target strings defined by RE**
  - Given a regular expression R, L(R) denotes the language
  - L(abc) = {abc}
  - L(hello|goodbye) = {hello, goodbye}
  - L(1(0|1)*) = all binary numbers that start with a 1

- **Each token is defined using a regular expression**
  - Keyword L(if|else|while|break…)
  - Integer L((-|+|ε)(1|2|3|4|5…|9)(0|1|2|…|9)*)

# RE Notational Shorthand

- **There are simpler ways to describe complex relations**
  - R+ one or more strings of R: R(R*)
  - R? optional R: (R|ε)
  - [abcd] one in the listed characters: (a|b|c|d)
  - [a-z] one in the range: (a|b|c|d|…|z)
  - [a-zA-Z]
  - [^ab] anything but one of the listed chars
  - [^a-z] one character not from this range

# RE Notational Shorthand Example

- **Describe an integer and real number**
  - digit [0-9] (or \d)
  - digits digit+
  - pos_int [1-9]digit*
  - int (-? pos_int) | 0
  - real int (.(pos_int | 0))?

# Class Exercise

- **Q1. What are the difference?**
  - [abc]
  - abc

- **Q2. How to describe scientific notations?**
  - -1.4E+17, -2.3E1, 8.3E-2
  - int (-? pos_int) | 0
  - opt_frac (.digits)
  - opt_exponent (E(+|-)?digits)
  - sci_note int opt_frac opt_exponent

# Multiple Matches

- **There are cases where there are multiple token matches**
  - elsex = 0
    - Option 1: else / x / = / 0 / ;
    - Option 2: elsex / = / 0 / ;

- **There are additional rules to choose one among multiple matches**
  - The longest matching token is selected
  - If there are ties, match the one with the highest priority (specification order)
    - Keyword {if | else | while …}
    - Identifier [a-zA-Z]+

# Specification, Recognition, and Automation

- Specification: how to specify lexical patterns?
  - Regular expression (RE)

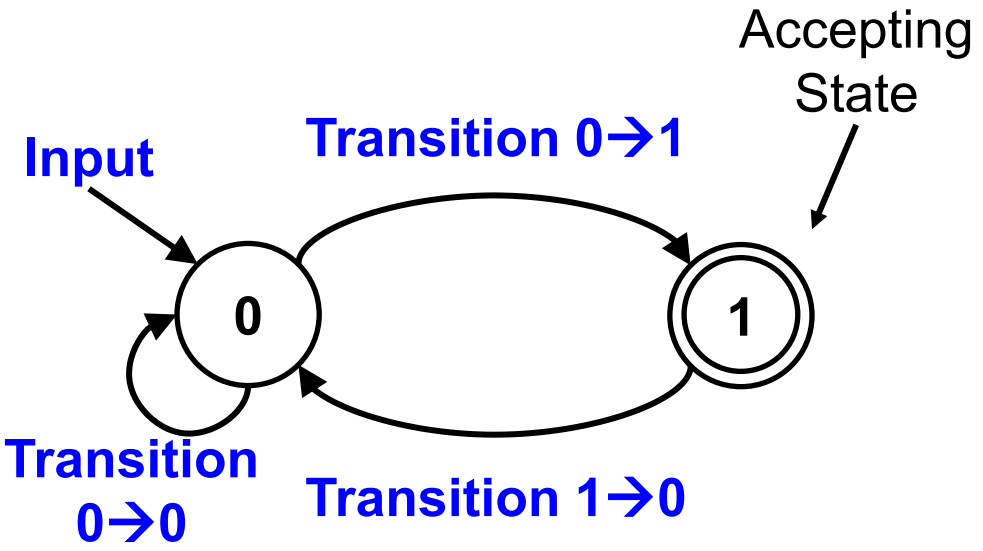
- **Recognition: how to recognize the specified patterns?**
  - Deterministic finite automata (DFA)

- Automation: how to generate DFA from RE?
  - Automatic generation tool (Lex)
  - Thompson's construction (RE -> NFA)
  - Subset construction (NFA -> DFA)

# Finite State Automata (FSA)

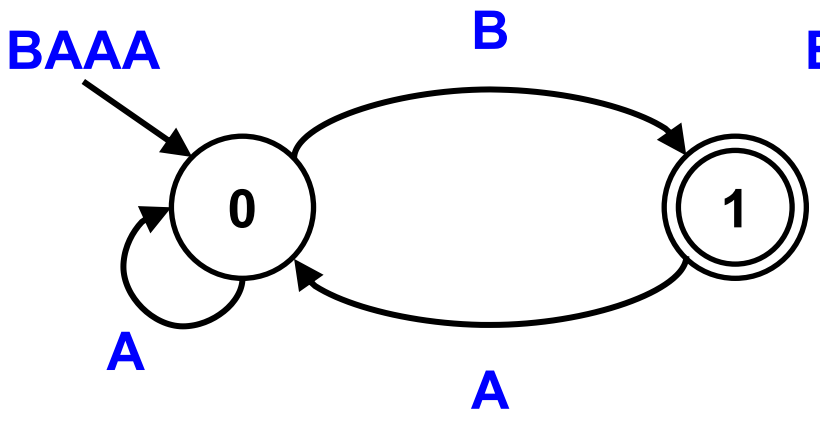
- **Lexical analysis relies on the FSA to recognize the specifications based on REs**
  - REs generate regular sets and FSAs recognize them

- **FSA consists of …**
  - An input
  - A finite set of states
  - A start state
  - A set of accepting states
  - A set of transitions from states to states

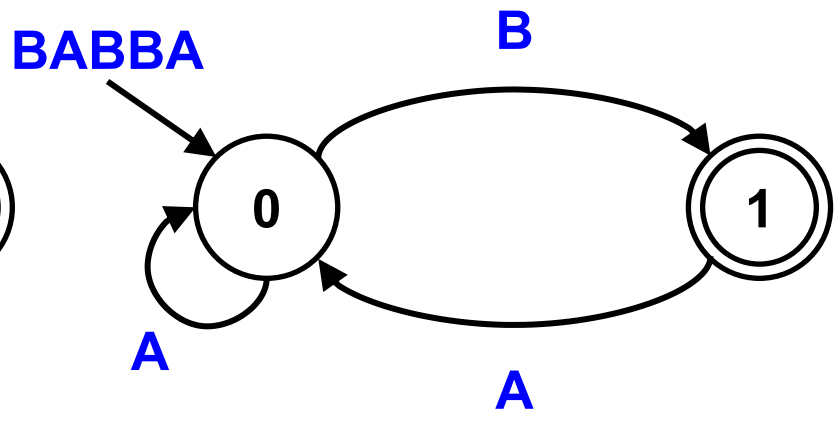


# FSA Representation

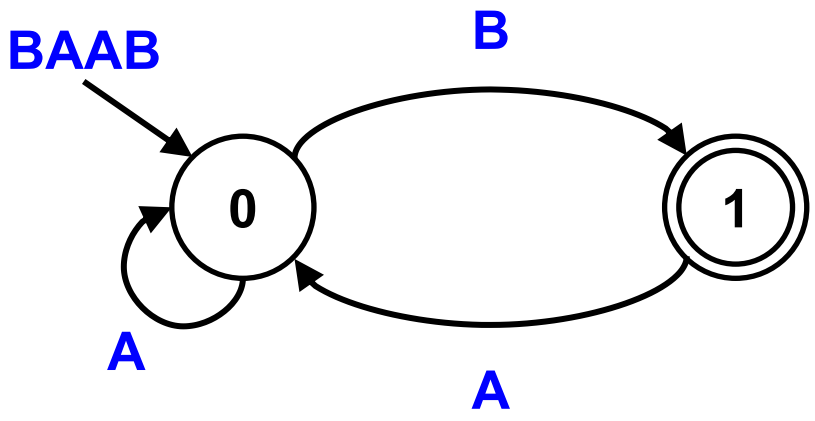
- **FSA is a transition from state to state for all the input sequence**
  - Case1: Rejects the input if it ends up in a non-accepting state
  - Case2: Rejects the input if the input sequence does not have a transition
  - Case3: Accepts the input if it ends up in an accepting state




Case 1




Case 2




Case 3

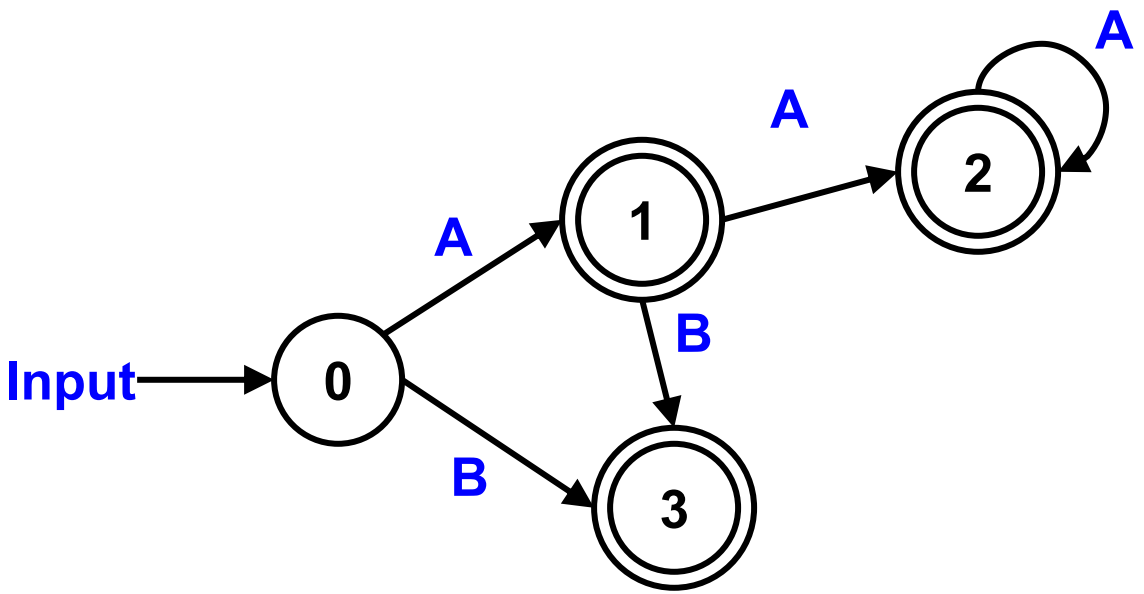
# Two Types of FSA

- **Deterministic finite automata (DFA)**
  - There is a single transition per input per state
  - There are no ε moves

- **Non-deterministic finite automata (NFA)**
  - There can be multiple transitions per input per state
  - There can be ε moves (transition without making a move to the input)
  - Accept, if at least one of the choices leads to an accepting state

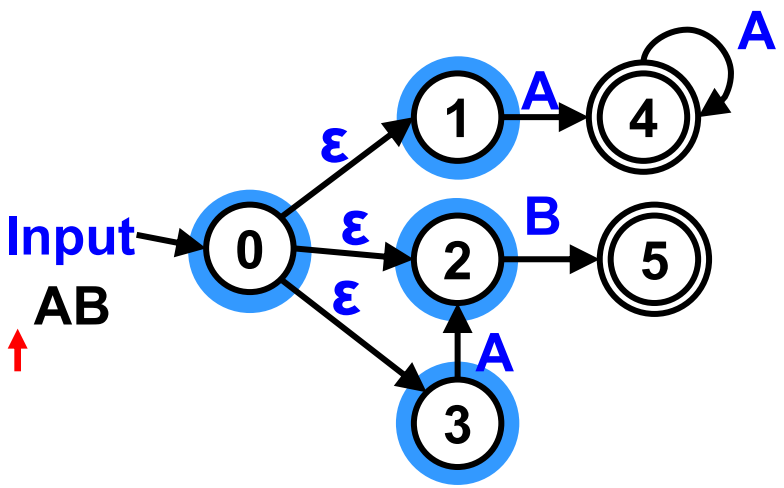
# DFA Example

- DFA recognizing: AA* | B | AB

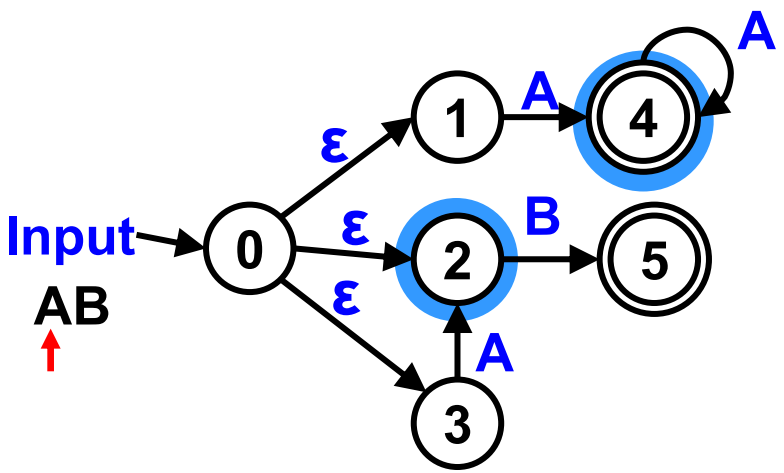


# NFA Example

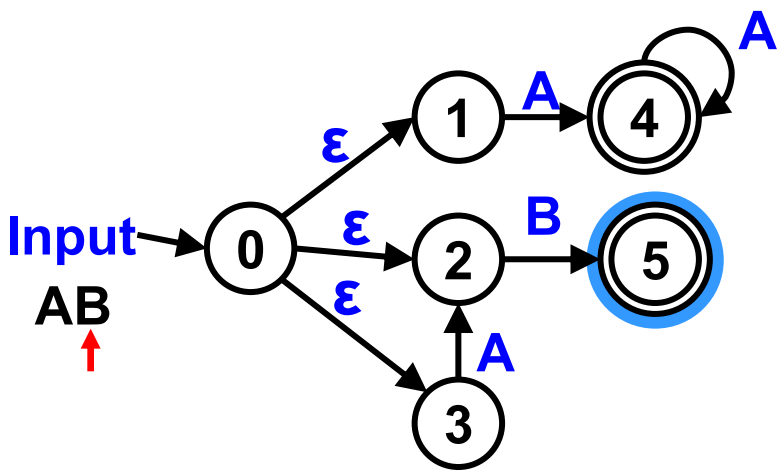
- NFA recognizing: AA* | B | AB




Step 1




Step 2



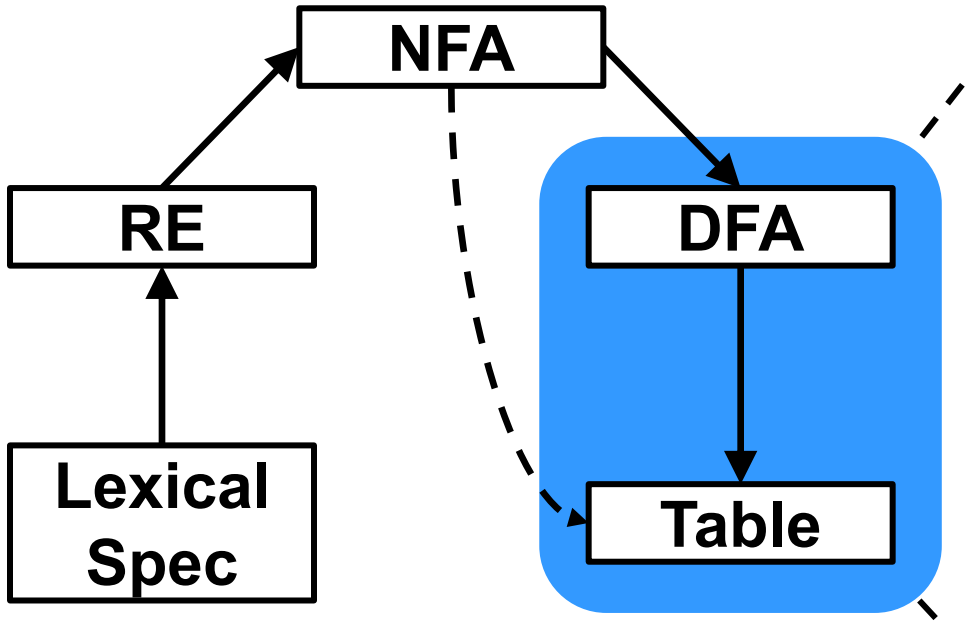

Step 3

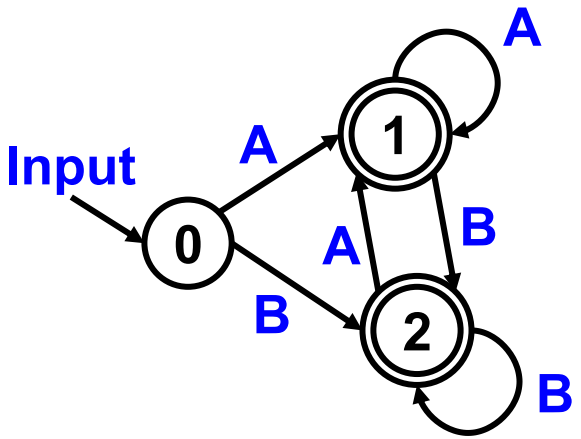
# DFA vs. NFA

- **DFA**
  - Requires more states compared to NFAs (generally)
  - **Easy to implement** and involves simple traversal

- **NFA**
  - Requires less states compared to DFAs
  - Hard to implement and involves complex traversal
  - **Easily converted from the REs**

# Overall Lexical Analysis Procedure




DFA

| | A | B |
|---|---|---|
| 0 | 1 | 2 |
| 1 | 1 | 2 |
| 2 | 1 | 2 |

Table

```
i = 0;
state = 0;
while (input[i]) {
  state = Table[state, input[i]];
  if (state == invalid) break;
  i++;
}
```

Implementation

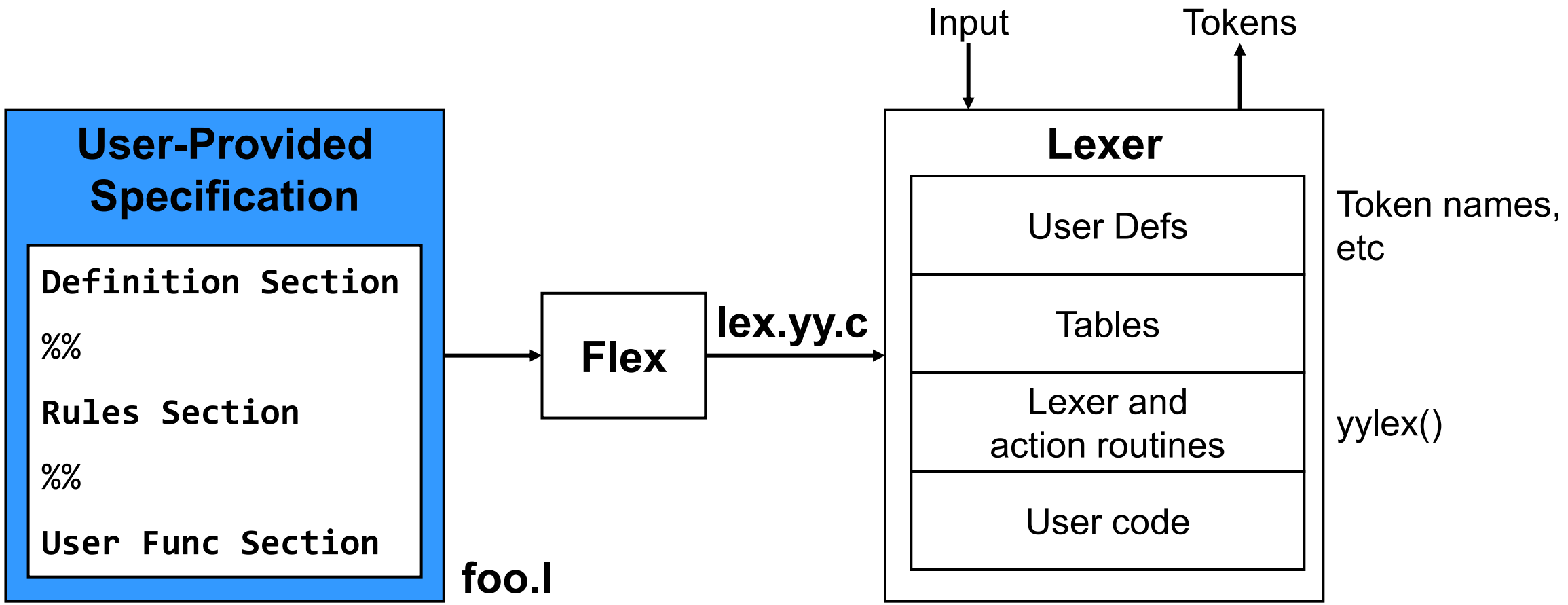
# Specification, Recognition, and Automation

- **Specification: how to specify lexical patterns?**
  - Regular expression (RE)
- **Recognition: how to recognize the specified patterns?**
  - Deterministic finite automata (DFA)
- **Automation: how to generate DFA from RE?**
  - Automatic generation tool (Lex)
  - Thompson's construction (RE -> NFA)
  - Subset construction (NFA -> DFA)

# Automatic Generation of Lexers

- **There are two representative programs at Bell Labs**
  - Lex: transforms an input program into the alphabet of the grammar processed by Yacc (Flex: faster implementation of Lex)
  - Yacc/Bison: Yet another compiler/compiler
- **Input**
  - List of regular expressions in priority order
  - Associated action with each RE
- **Output**
  - A program that reads input and breaks it up into tokens according to the REs

# Lex / Flex

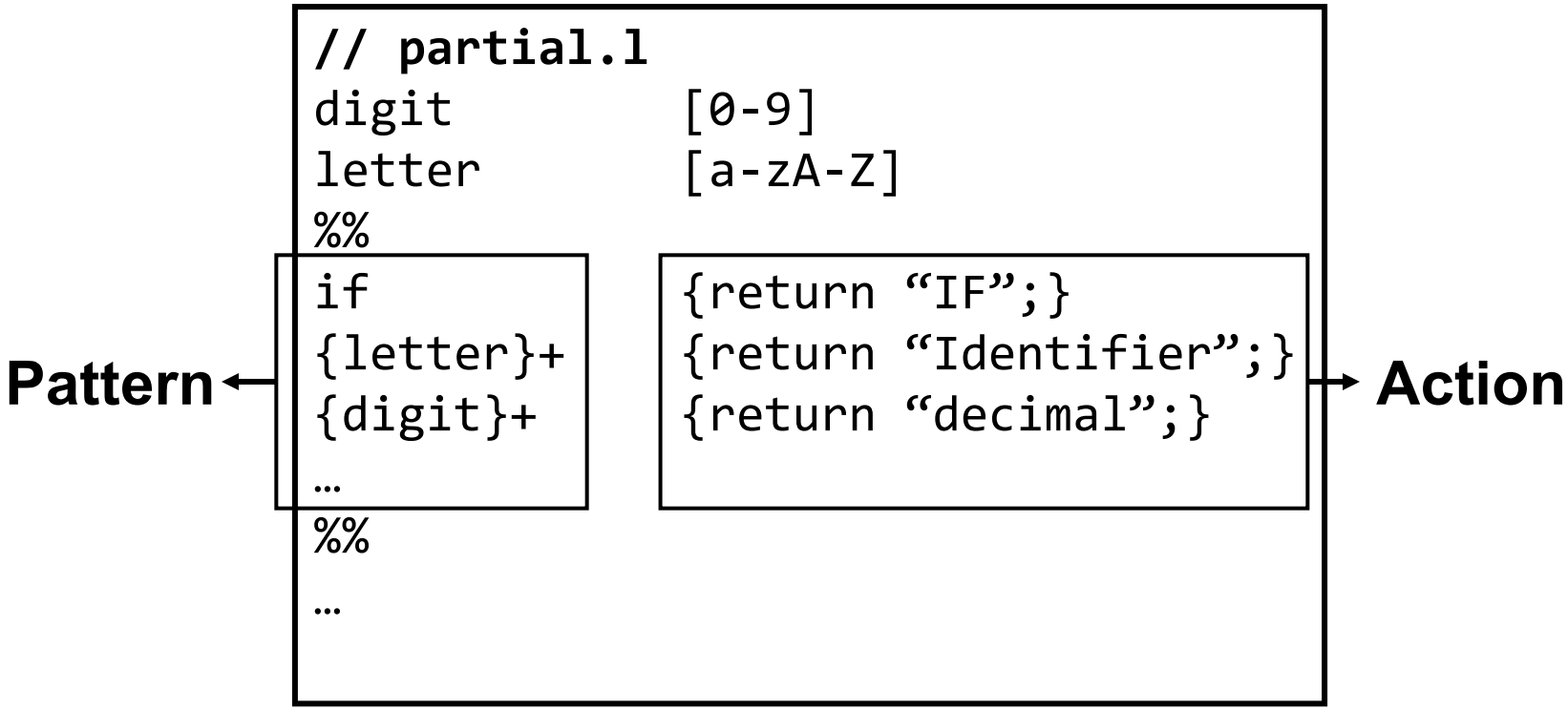

Input
Tokens
User-Provided Specification
Definition Section
%%
Rules Section
%%
User Func Section
foo.l
Flex
lex.yy.c
Lexer
User Defs
Tables
Lexer and action routines
User code
Token names, etc
yylex()

# Lex Specification

- **Definition section**
  - Users can declare or include variables, enumeration, etc using the code in between “%{“ and “%}”
  - Users provide names sub-rules for complex patterns used in rules
- **Rules section**
  - Contains lexical patterns for tokens and actions to be performed upon match
- **User function section**
  - Define functions that are copied to the resulting lexer program

- **Refer to the man page for details:**
  - https://man.freebsd.org/cgi/man.cgi?query=lex&sektion=1

# Partial Flex Example



# Flex Program Example

```
// count.l
%{
        #include <stdio.h>
        int num_lines = 0;
        int num_chars = 0;
%}
%%
\n      {++num_lines; ++num_chars}
.       {++num_chars;}

%%
main()
{
        // executes rules section
        yylex();
        printf("%d, %d", num_chars, num_lines)
}
```

```
flex count.l
     |
     v
gcc lex.yy.c -lfl
     |
     v
./a.out examples.txt
```

# More Complex Example

```
// complex.l
%{
        #include <stdio.h>
        int num_lines = 0;
%}
%%
[ \t]  {//skip whitespace}
a |
an |
the    {printf("%s: is an article\n", yytext);}
[a-z]+ {printf(“%s: ???\n”, yytext);}
%%
main()
{
        yylex();
}
```

- yytext: a pointer to the first character of the token
- yyleng: a length of the token

# Lex Regular Expression Meta Chars

| Meta Char | Meaning |
|---|---|
| . | match any single char (except \n) |
| * | Kleene closure (0 or more) |
| [] | Match any character within brackets<br>- in first position matches -<br>^ in first position inverts set |
| ^ | matches beginning of line |
| $ | matches end of line |
| {a,b} | match count of preceding pattern from a to b times, b optional |
| \ | escape for metacharacters |
| + | positive closure (1 or more) |
| ? | matches 0 or 1 REs |
| \| | alteration |
| / | provides lookahead |
| () | grouping of RE |
| <> | restricts pattern to matching only in that state |

# Practical Regular Expression Use Cases - 1

- **There are plenty of useful Linux commands that benefits from regular expressions**
  - `grep`

    If you want to find lines that contain (TEST)+ from file test.txt

    ```
    grep -n '\(TEST\)\+' test.txt
    ```
  - `find`

    If you want to find file name that contains test

    ```
    find -name "test*"
    ```
  - `sed`

    If you want to replace (TEST)+ to BEST for the file test.txt

    ```
    sed -i 's/\(TEST\)\+/BEST' test.txt
    ```

# Practical Regular Expression Use Cases - 2

- **There are plenty of useful Linux commands that benefits from regular expressions**
  - `awk`

    awk is a pattern & action command sequence for a given filename

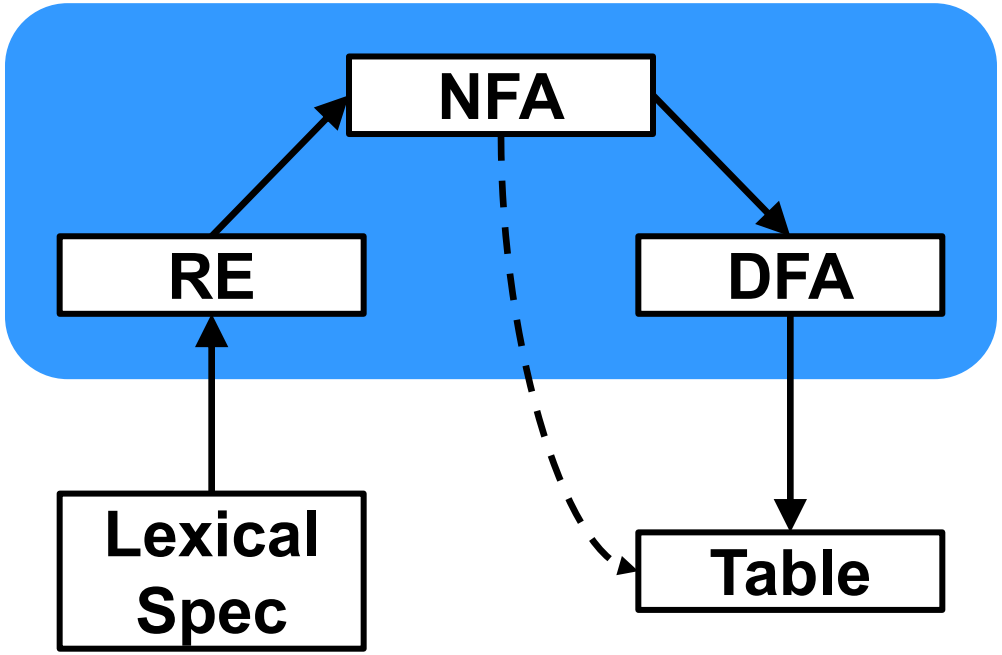
    ```
    awk 'pattern {action}' filename
    awk '$1 ~ /fail/ {print $2}' file.txt
    ```

```
// file.txt
fail1 100
succ1 200
succ1 101
succ1 110
fail2 120
succ1 300
```

# Specification, Recognition, and Automation

- **Specification: how to specify lexical patterns?**
  - Regular expression (RE)
- **Recognition: how to recognize the specified patterns?**
  - Deterministic finite automata (DFA)
- **Automation: how to generate DFA from RE?**
  - Automatic generation tool (Lex)
  - **Thompson's construction (RE -> NFA)**
  - **Subset construction (NFA -> DFA)**

# Review: Lexical Analysis Procedure

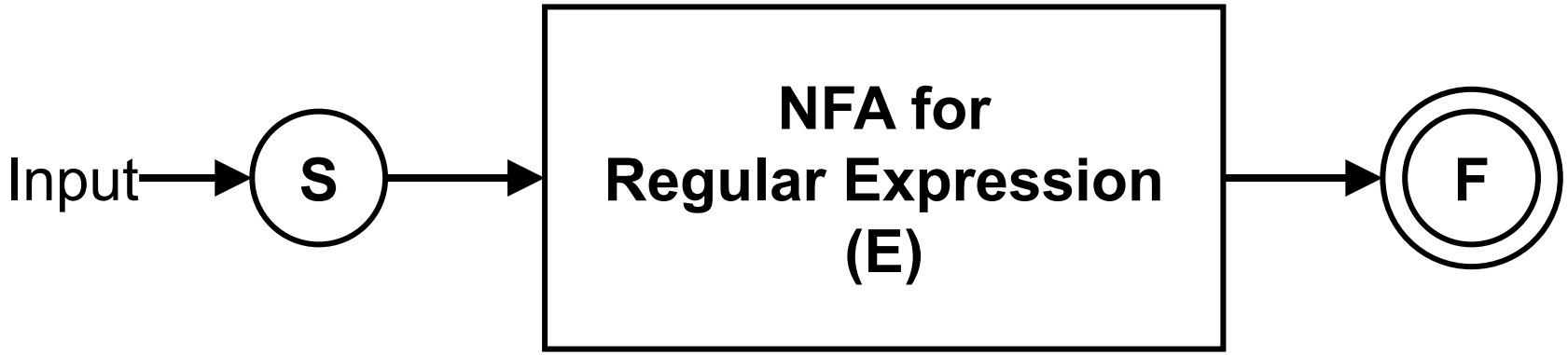


# Specification, Recognition, and Automation

- **Specification: how to specify lexical patterns?**
  - Regular expression (RE)
- **Recognition: how to recognize the specified patterns?**
  - Deterministic finite automata (DFA)
- **Automation: how to generate DFA from RE?**
  - Automatic generation tool (Lex)
  - **Thompson's construction (RE -> NFA)**
  - **Subset construction (NFA -> DFA)**

# Regular Expression to NFA

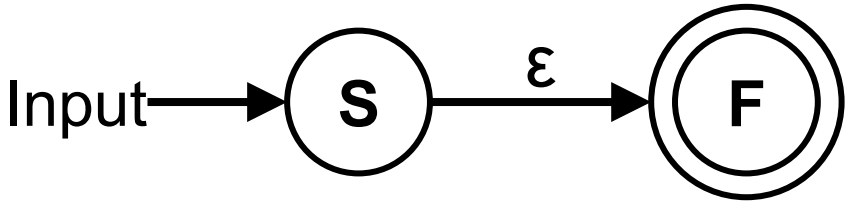
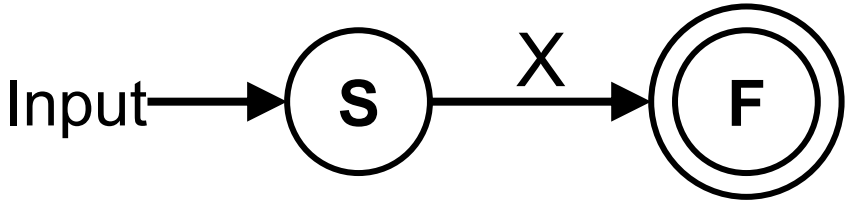
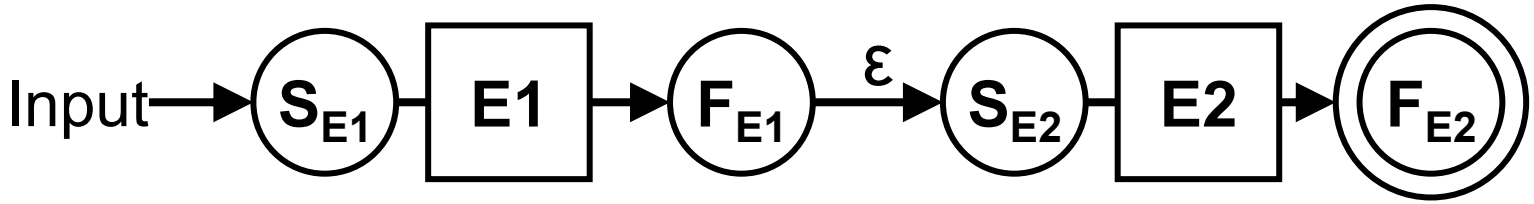
- **Thompsons's construction algorithm automatically generates NFAs from REs**
  - Exists a general rule to build the NFA
  - Define rules for each base RE
  - Combine for more complex REs



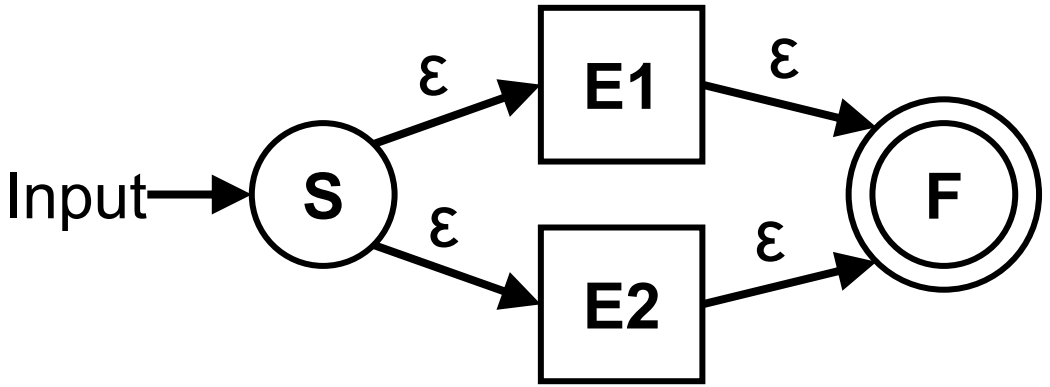
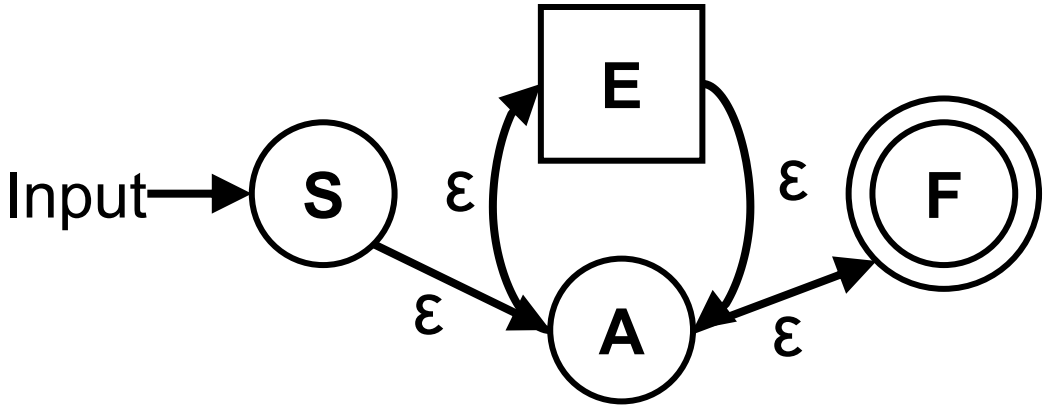
# Recall: Regular Expression

- **RE is an inductively defined rules to describe tokens**
  - **a** ordinary character stands for itself
  - **ε** empty string
  - **R|S** either R or S (alteration), where R,S = RE
  - **RS** R followed by S (concatenation)
  - **R*** concatenation of R, 0 or more times

# Thompson Construction Primitives

| Regular Expression | NFA |
| --- | --- |
| ε (empty string) |  |
| x (alphabet symbol) |  |
| E1 E2 (concat) |  |

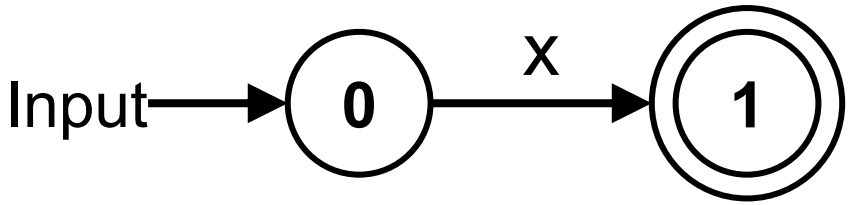
# Thompson Construction Primitives

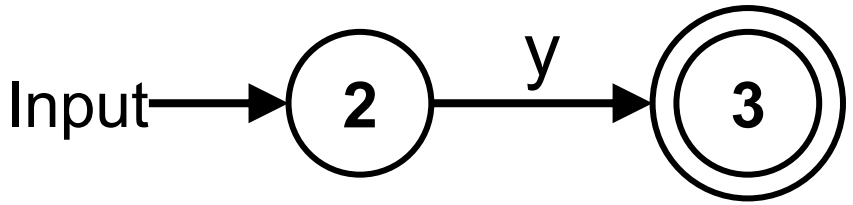
| Regular Expression | NFA |
| --- | --- |
| E1 \| E2 (Alteration) |  |
| E* (Closure) |  |

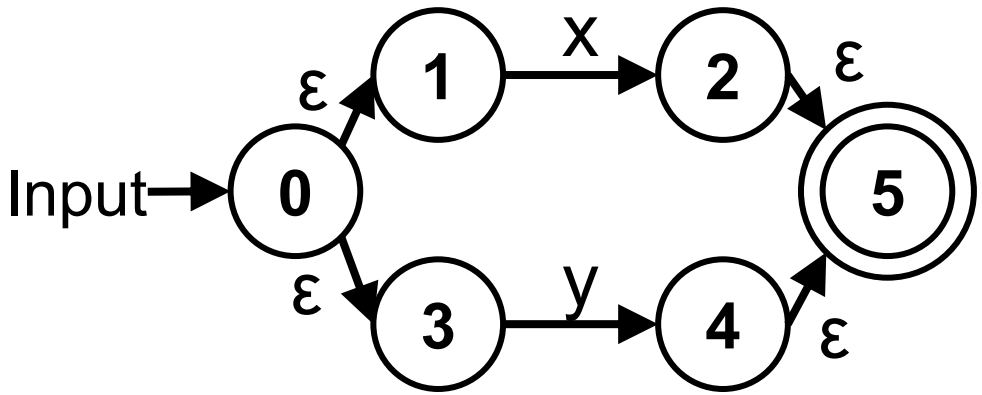
# Thompson Construction - Example

- **NFA for (x | y)***

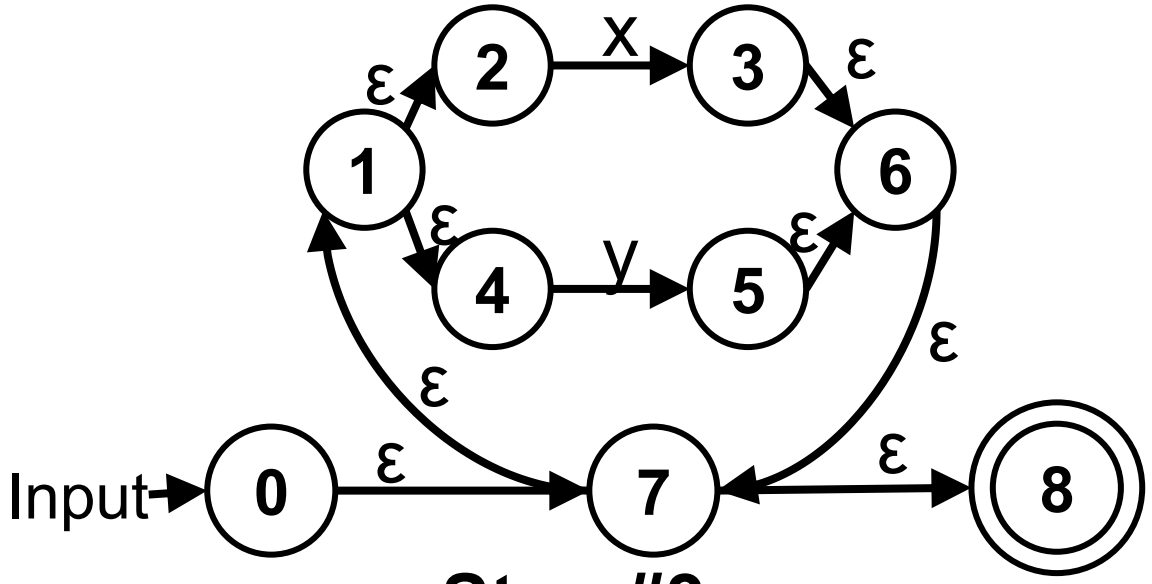
**Step #1**



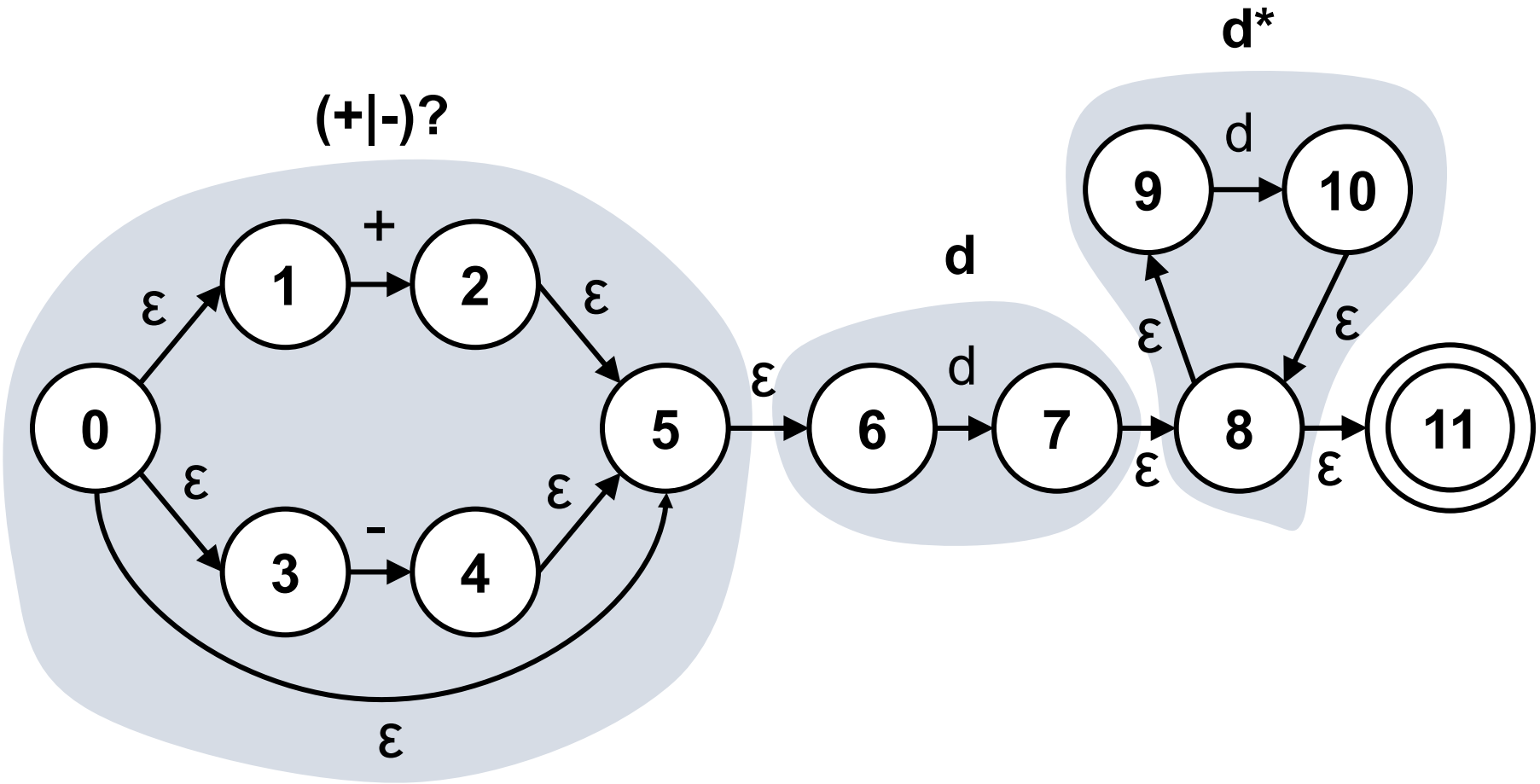

**Step #2**




**Step #3**

# Exercise

- Develop an NFA for the (+|-)?d+



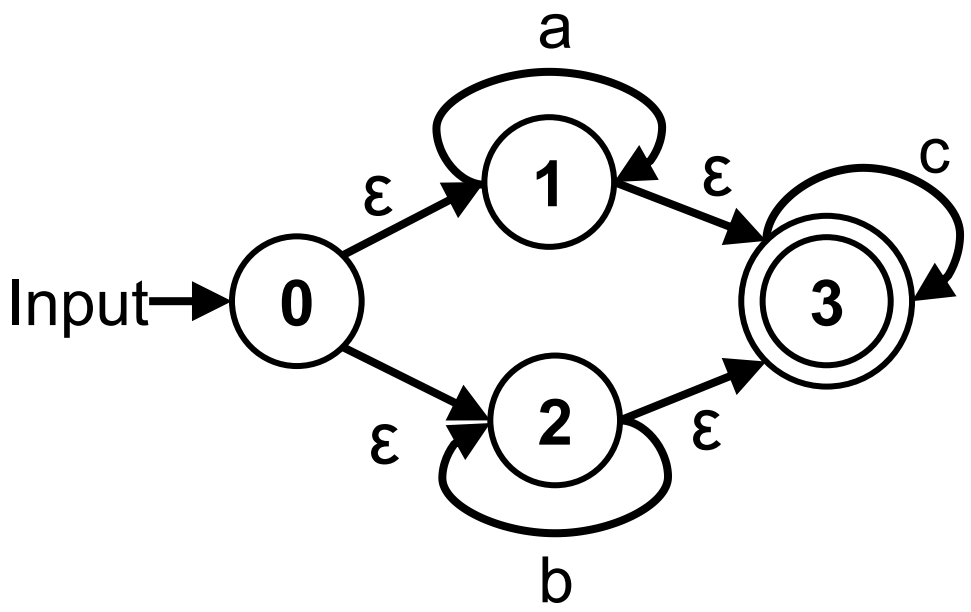
# Specification, Recognition, and Automation

- **Specification: how to specify lexical patterns?**
  - Regular expression (RE)
- **Recognition: how to recognize the specified patterns?**
  - Deterministic finite automata (DFA)
- **Automation: how to generate DFA from RE?**
  - Automatic generation tool (Lex)
  - **Thompson's construction (RE -> NFA)**
  - **Subset construction (NFA -> DFA)**

# Challenges in NFA to DFA Transition

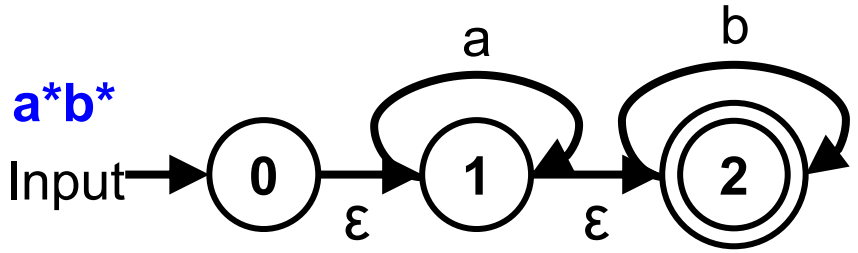
- **The key challenge is to remove the non-deterministic transitions**
  - Handle multiple transitions due to the ε transitions in NFAs

**(a* | b*) c***

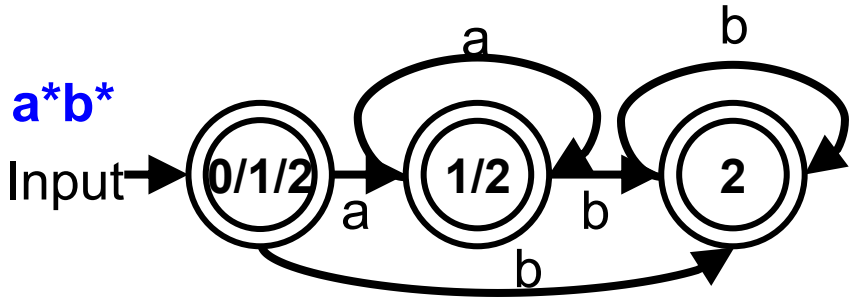


# NFA to DFA

- **Each state in DFA becomes a set of states in NFA**
- **Use ε-closure to translate the states and transitions**
  - Any state reachable from S by ε transitions is in the ε-closure, and it represents a single big state in DFA
  - If any states in the set is an accepting state, the big state is also an accepting state

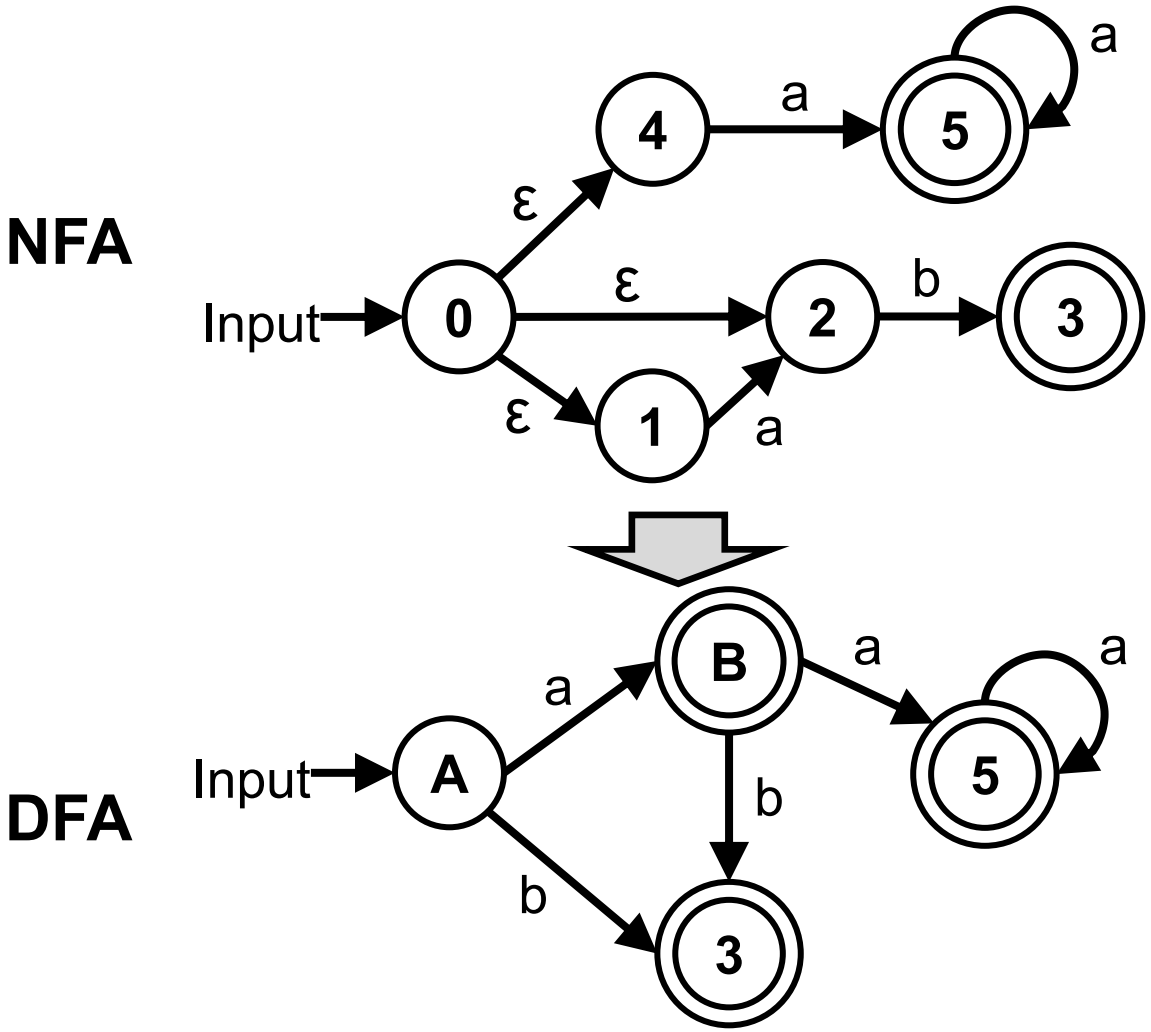



ε-closure(0) = {0,1,2}
ε-closure(1) = {1,2}
ε-closure(2) = {2}




Δ(ε-closure(0), a) ➔ ε-closure(1)
Δ(ε-closure(0), b) ➔ ε-closure(2)
Δ(ε-closure(1), a) ➔ ε-closure(1)
Δ(ε-closure(1), b) ➔ ε-closure(2)
Δ(ε-closure(2), b) ➔ ε-closure(2)

# NFA to DFA Example




**Step #1**

ε-closure(0) = {0,1,2,4}

Δ(ε-closure(0), a) ➔ ε-closure(2) | ε-closure(5)

Δ(ε-closure(0), b) ➔ ε-closure(3)

**Step #2**

ε-closure(2) | ε-closure(5) = {2,5} ➔ ε-closure(2|5)

ε-closure(3) = {3}

Δ(ε-closure(2|5), a) ➔ ε-closure(5)

Δ(ε-closure(2|5), b) ➔ ε-closure(3)

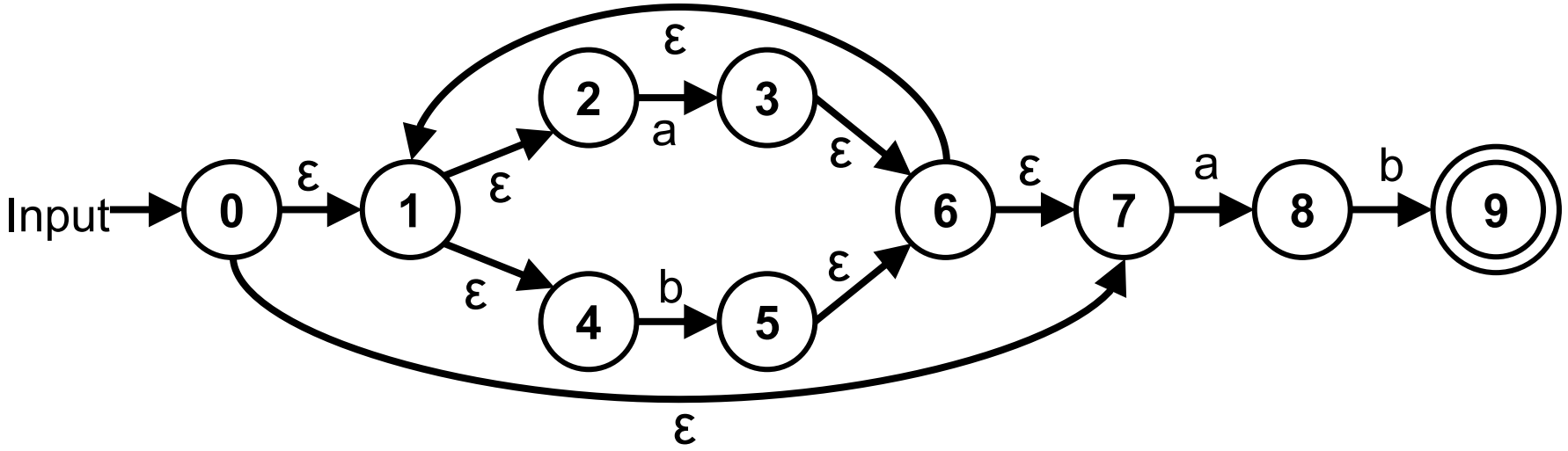
**Step #3**

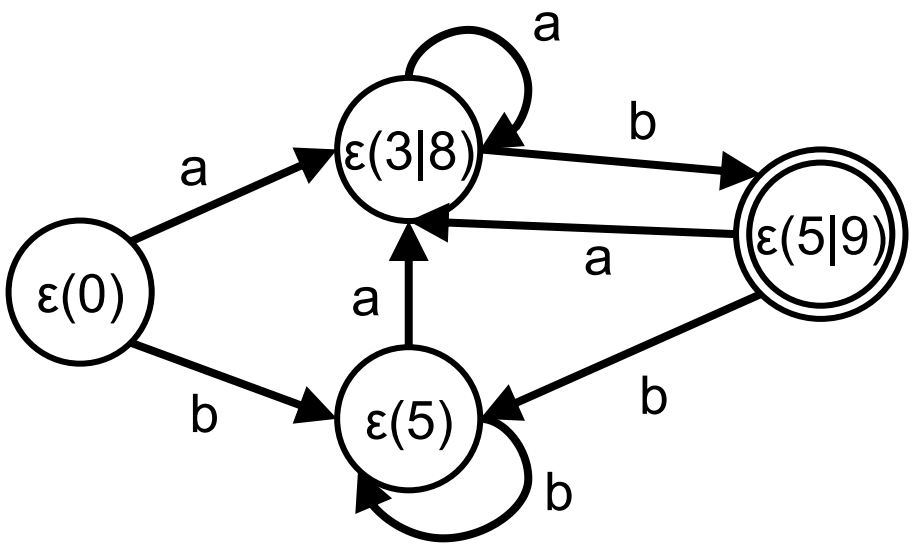
ε-closure(5) = {5}

Δ(ε-closure(5), a) ➔ ε-closure(5)

We have four states ➔ ε-closure(0) (A), ε-closure(2|5) (B), ε-closure(5) (5), and ε-closure(3) (3)

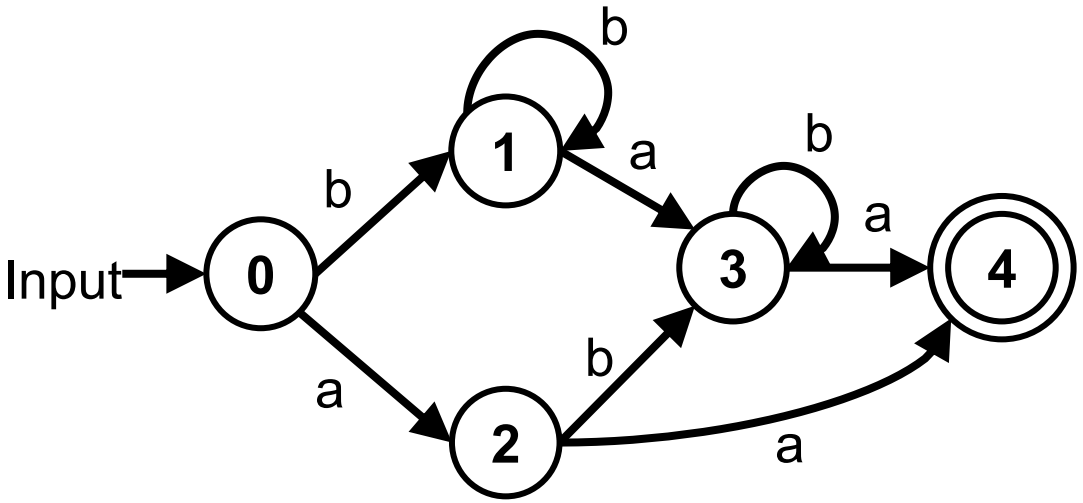
# Class Exercise

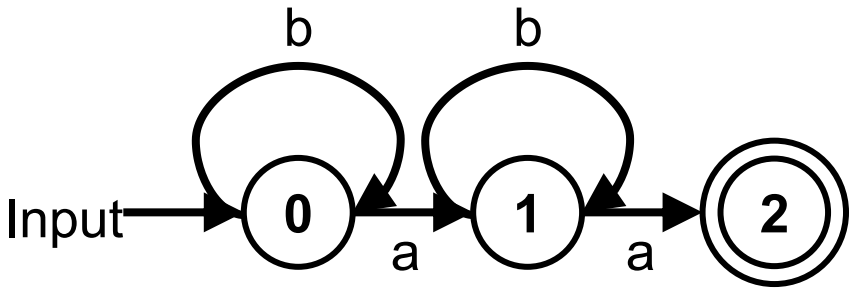


# DFA Optimization

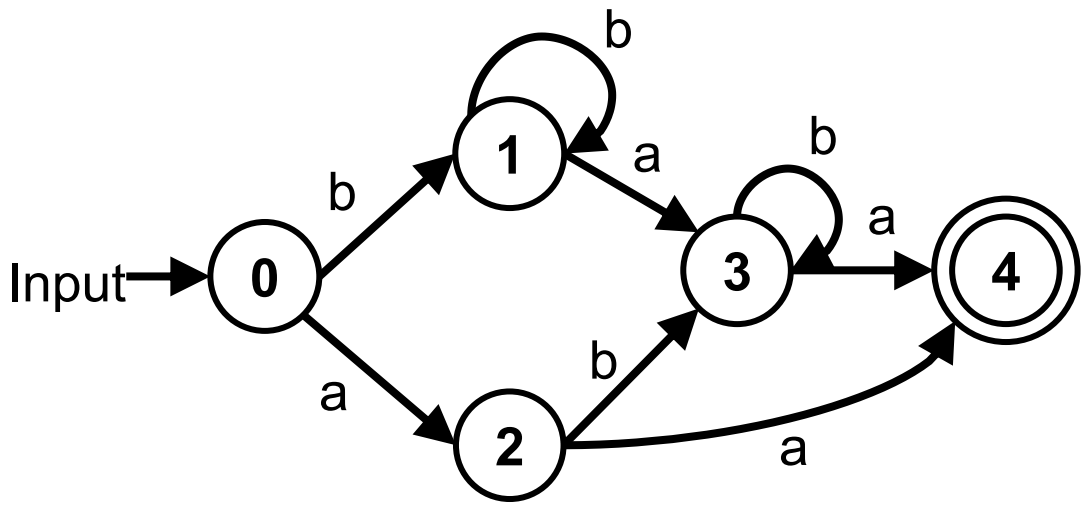
- **There are room to optimize your DFA**
  - Some DFAs contain redundant or equivalent states



# DFA Optimization

- **Find groups of equivalent states and merge them**
  - It is about finding distinguishable states
    - If the two states arrive at the same destination state for each input, they are not distinguishable

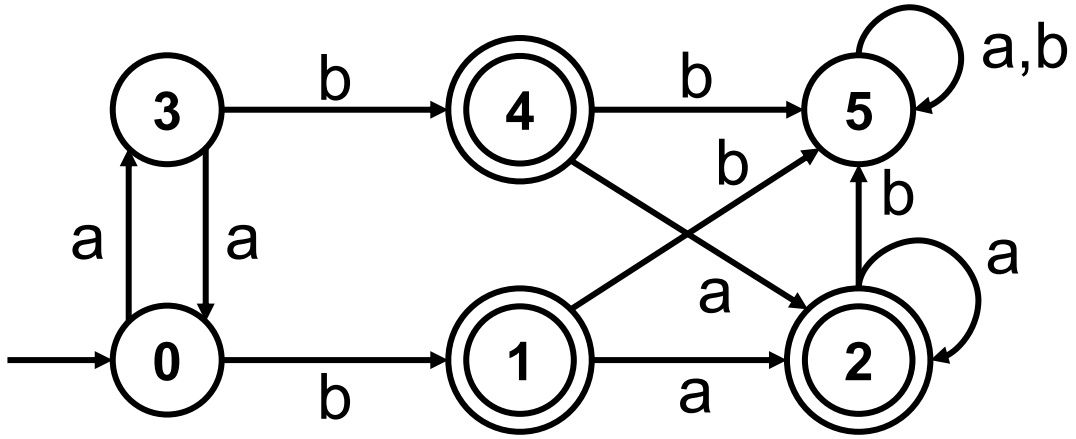



Δ(2, a) = 4 & Δ(2, b) = 3
Δ(3, a) = 4 & Δ(3, b) = 3
➔ **merge 2 and 3**

Δ(0, a) = 2 & Δ(0, b) = 1
Δ(1, a) = 3 (==2) & Δ(1, b) = 1
➔ **merge 0 and 1**

# DFA Optimization

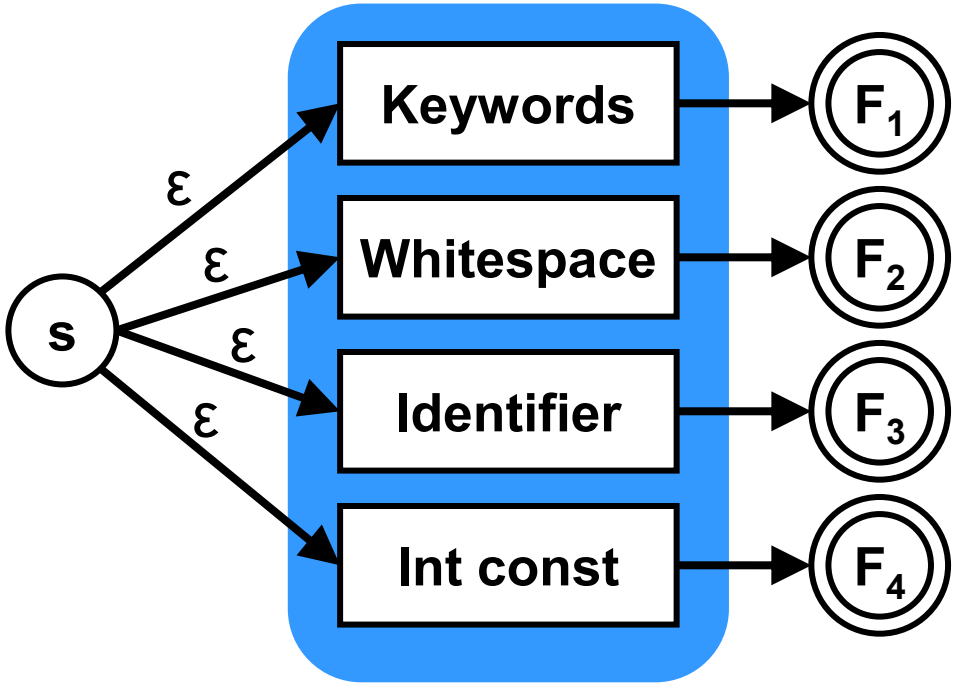
- **Iteratively minimize a DFA**
  - Step 1. Divide the states into two sets (non-accepting & accepting states)
  - Step 2. For each set, iterate over a  pair of states and split them into different sets if they are distinguishable
    - Two states (i, j) are distinguishable if for any input symbol a, Δ(i, a) and Δ(j, a) are in different sets
  - Step 3. Repeat Step 2 until the sets do not change



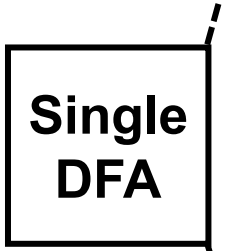

**Iter #1:** Sets = {0,3,5}, {1,2,4}
**Iter #2:** Sets = {0,3}, {5}, {1,2,4}
**Iter #3:** Sets = {0,3}, {5}, {1,2,4}
➔ Finalized

# Converting Multiple REs

- Combine the NFAs of all the REs into a single NFA
- Convert to a DFA and follow token matching rules



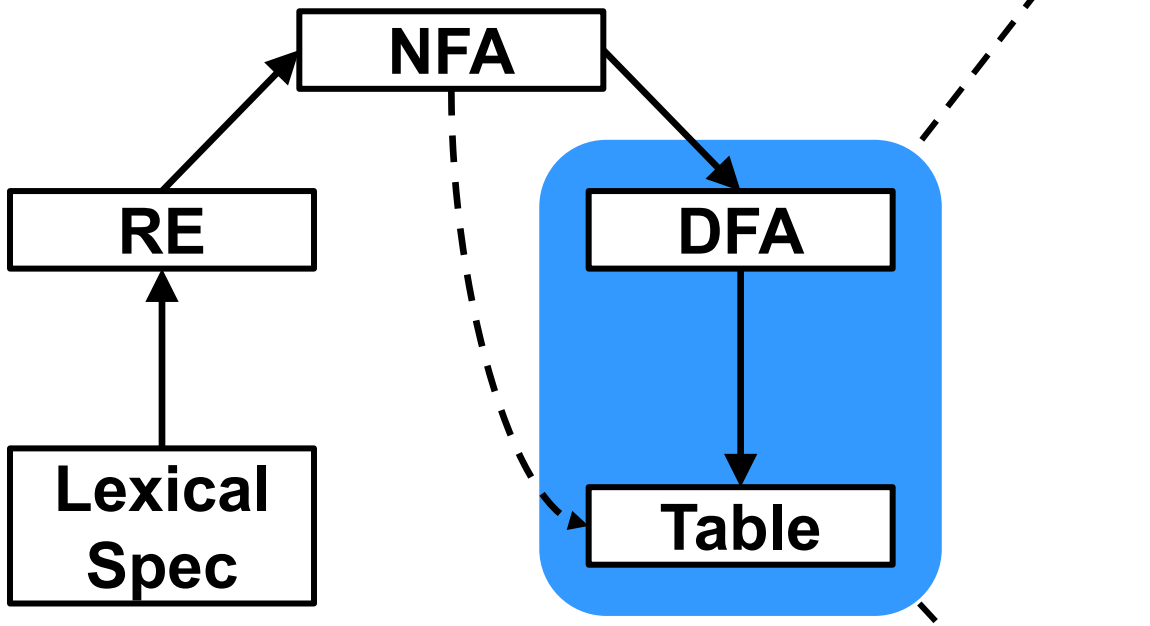

Combine per-token NFAs

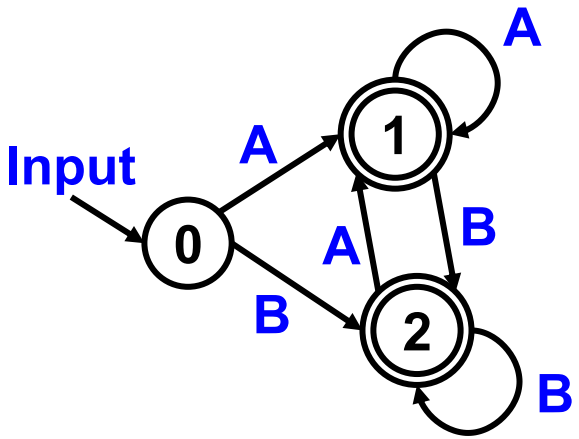



**Token Matching Rules**

1. Associate tokens with **final (accepting) states**
2. When in a final state, **look if there is a further transition**
3. Select **according to the priority** if a final state is for multiple tokens

# Recall: DFA to Implementation




DFA

| | A | B |
|---|---|---|
| 0 | 1 | 2 |
| 1 | 1 | 2 |
| 2 | 1 | 2 |

Table

```
i = 0;
state = 0;
while (input[i]) {
  state = A[state, input[i]];
  if (state == invalid) break;
  i++;
}
```

Implementation