



中山大學  
SUN YAT-SEN UNIVERSITY

计算机学院 (软件学院)

SCHOOL OF COMPUTER SCIENCE AND ENGINEERING

# Compilation Principle

## 编译原理

### 第4讲：词法分析(4)

张献伟

[xianweiz.github.io](http://xianweiz.github.io)

DCS290, 3/12/2024



中山大學  
SUN YAT-SEN UNIVERSITY



# Welcome[欢迎加入]

- 21级计科/1班（李文军）
  - 21级计科/2班（冯剑琳 + 林瀚）
  - 21级计科/3班（张献伟）
  - 21级计科/系统+AI等（赵帅）
- 21级计科/2班（冯剑琳 + 林瀚）
  - 21级计科/3班+（张献伟）
  - 21级计科/系统+AI等+（赵帅）



Welcome  
Glad you're here!

We'll never be ready. So I guess that means we're as ready as we'll ever be.

Neal Shusterman

# Time/Location[课时安排]

---

- 编译原理（3学分，54学时）

- 排课：1-18周

- 周二：1-9周

- 周四：1-18周

- 每次授课包括2个课时

- 第五节：14:20 - 15:05，第六节：15:15 - 16:00

- 地点：教学大楼 ~~C104~~B205

- 编译器构造实验（1学分，36学时）

- 排课：1-18周

- 周四：1-18周

- 每次实验包括2个课时

- 第七节：16:30 - 17:15，第八节：17:25 - 18:10

- 地点：实验中心 B202

# Slides/Office Hours[课件及答疑]

- 课件

- 英文为主，术语中文标注

- 课后或课前上传

- 主页: <https://arcsysu.github.io/teach/dcs290/s2024.html>

- 作业及实验提交

- 超算习堂: <https://easyhpc.net/course/164>

- **课程QQ群: 189 205 980**

- 通知提醒、答疑讨论

- 线下答疑

- 理论课前课间，或实验课期间

- 其他时间需预约

- Email: [zhangxw79@mail.sysu.edu.cn](mailto:zhangxw79@mail.sysu.edu.cn)

xianweiz.github.io

## > Teaching

### - Undergraduate

§ **DCS290 - Compilation Principle**, [2024s, 2023s, 2022s, 2021s].

§ DCS3013 - Computer Architecture, [2022f].

### - Graduate

§ DCS5637/6207 - Advanced Computer Architecture, [2023f, 2022f, 2021f].



# Grading[考核标准]

## • 编译原理

- 课堂参与（15%）- 点名、提问、测试
- 课程作业（25%）- 5次左右，理论
- 期末考试（60%）- 闭卷

## • 编译器构造实验

- 课堂参与（10%）- 签到、练习等
- Project 1（20%）- Lexical Analysis
- Project 2（20%）- Syntax/Semantic Analysis
- Project 3（20%）- IR Generation
- Project 4（30%）- Code Optimization

## • 理论

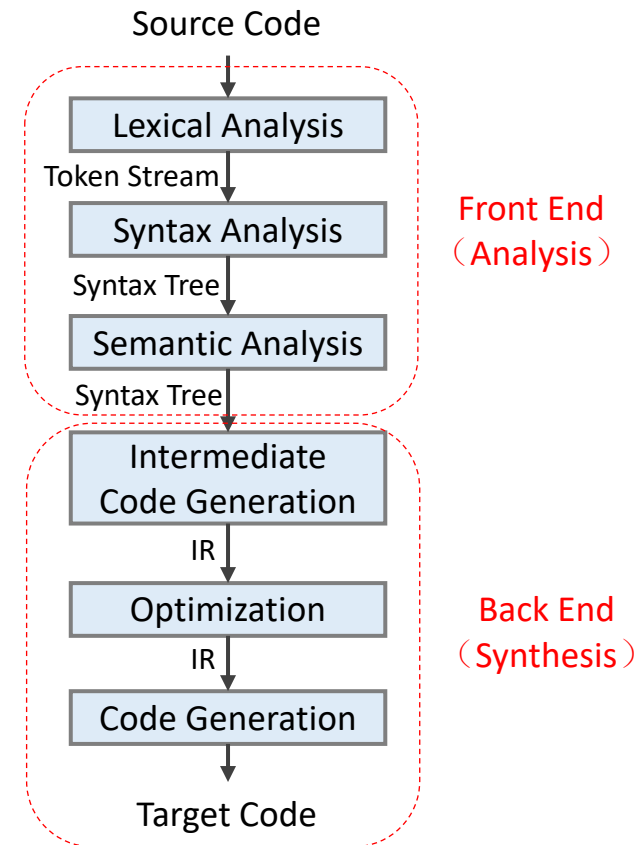
- 随机点名
  - 缺席优先
- 随机提问
  - 后排优先
- 随机测试
  - 不定时间

## • 实验

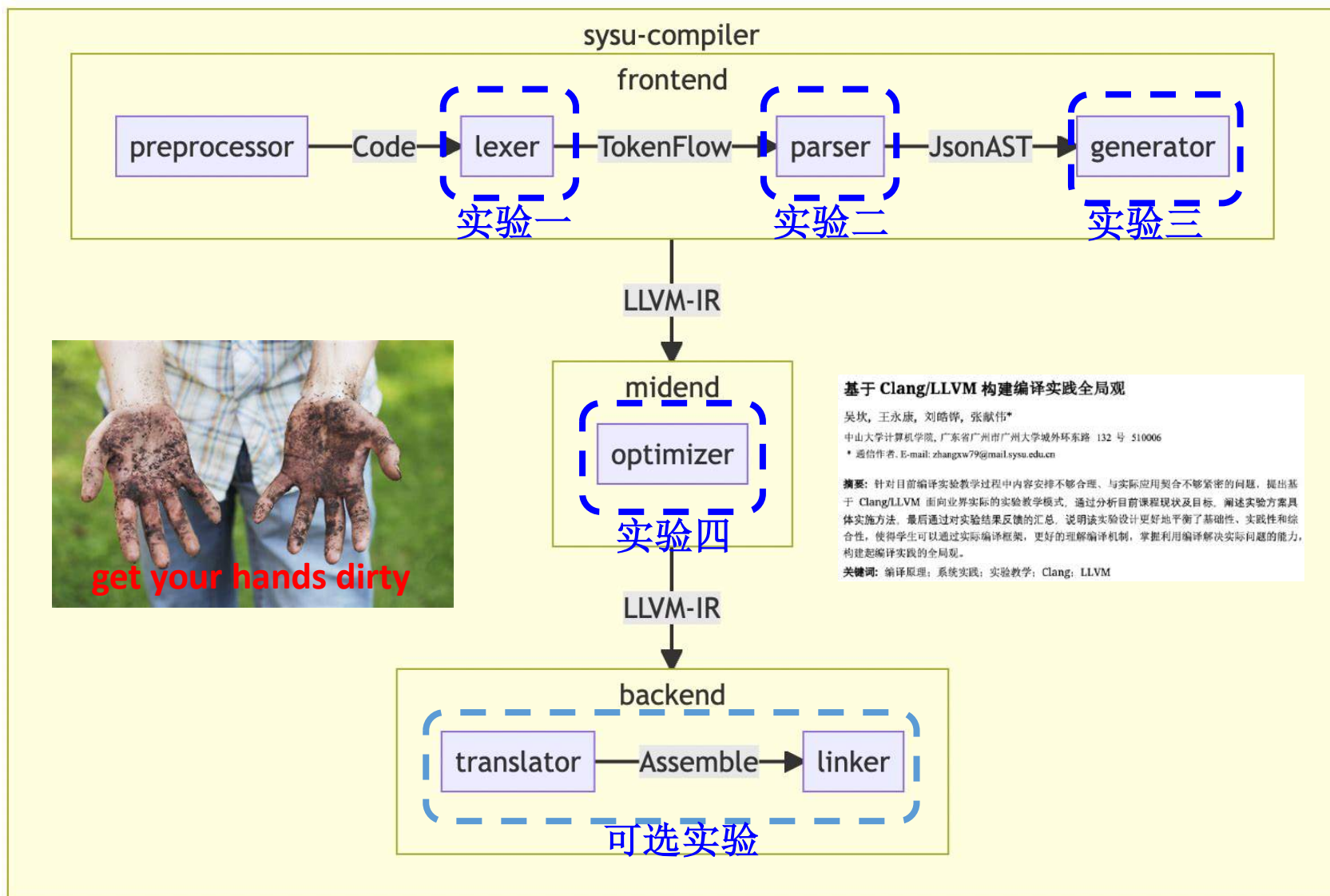
- 个人完成
  - 杜绝抄袭
- 按时提交
  - 硬性截止
- 侧重代码实现
  - 简略报告

# Schedule-Lec[理论安排]

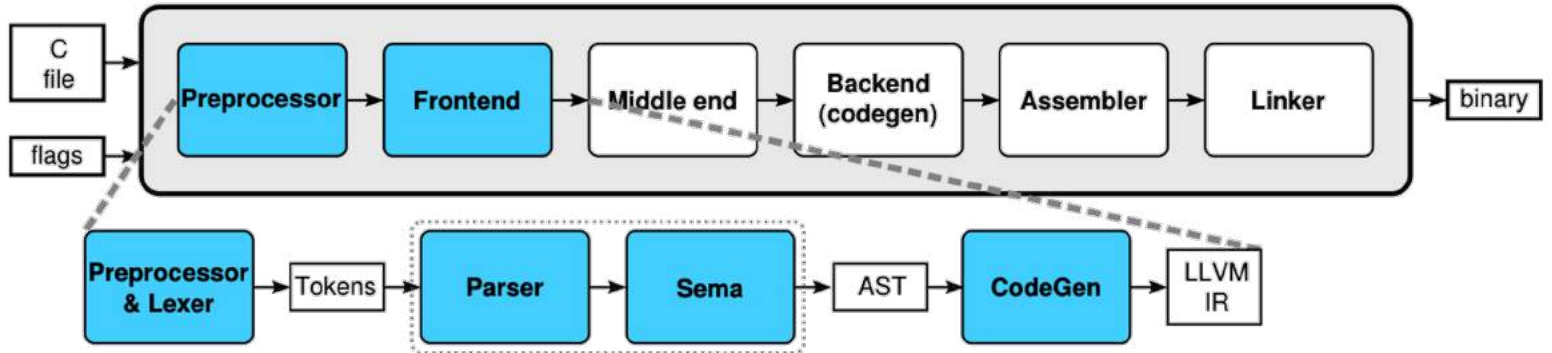
- **Lexical:** source code  $\rightarrow$  tokens
  - RE, NFA, DFA, ...
- **Syntax:** tokens  $\rightarrow$  AST or parse tree
  - CFG, LL(1), LALR(1), ...
- **Semantic:** AST  $\rightarrow$  AST + symbol table
  - SDD, SDT, typing, scoping, ...
- **Int. Code Generation:** AST  $\rightarrow$  IR
  - TAC, offset, CodeGen, ...
- **Optimization:** IR  $\rightarrow$  (optimized) IR
  - BB, CFG, DAG, ...
- **Code Generation:** IR  $\rightarrow$  Instructions
  - Instruction, register, stack, ...



# Schedule-Lab[实验安排]



# Lab[实验: 编译器构造]



## SYsU-lang

```
000_main.sysu.c
1 int main(){
2     return 3;
3 }
```

- sysu-compiler
- ① lexer
  - ② parser
  - ③ generator
  - ④ optimizer

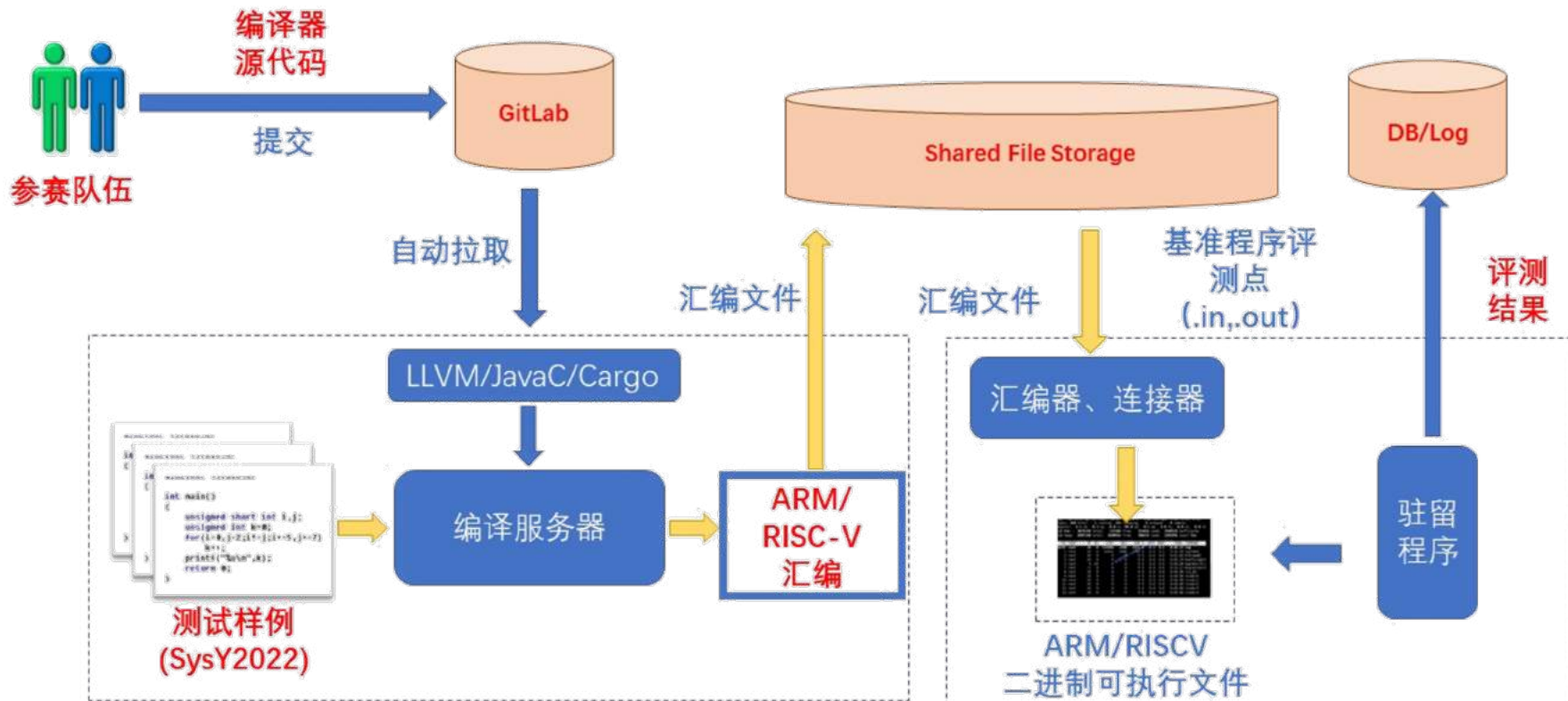
```
a.S
1     .text
2     .file    "_"
3     .globl  main                // -- Begin function main
4     .p2align 2
5     .type   main,@function
6 main:
7     .cfi_startproc                // @main
8 // %bb.0:                          // %entry
9     mov     w0, #3
10    ret
11 .Lfunc_end0:
12    .size   main, .Lfunc_end0-main
13    .cfi_endproc
14
15 _    .section    ".note.GNU-stack","",@progbits
// -- End function
```

clang → executable



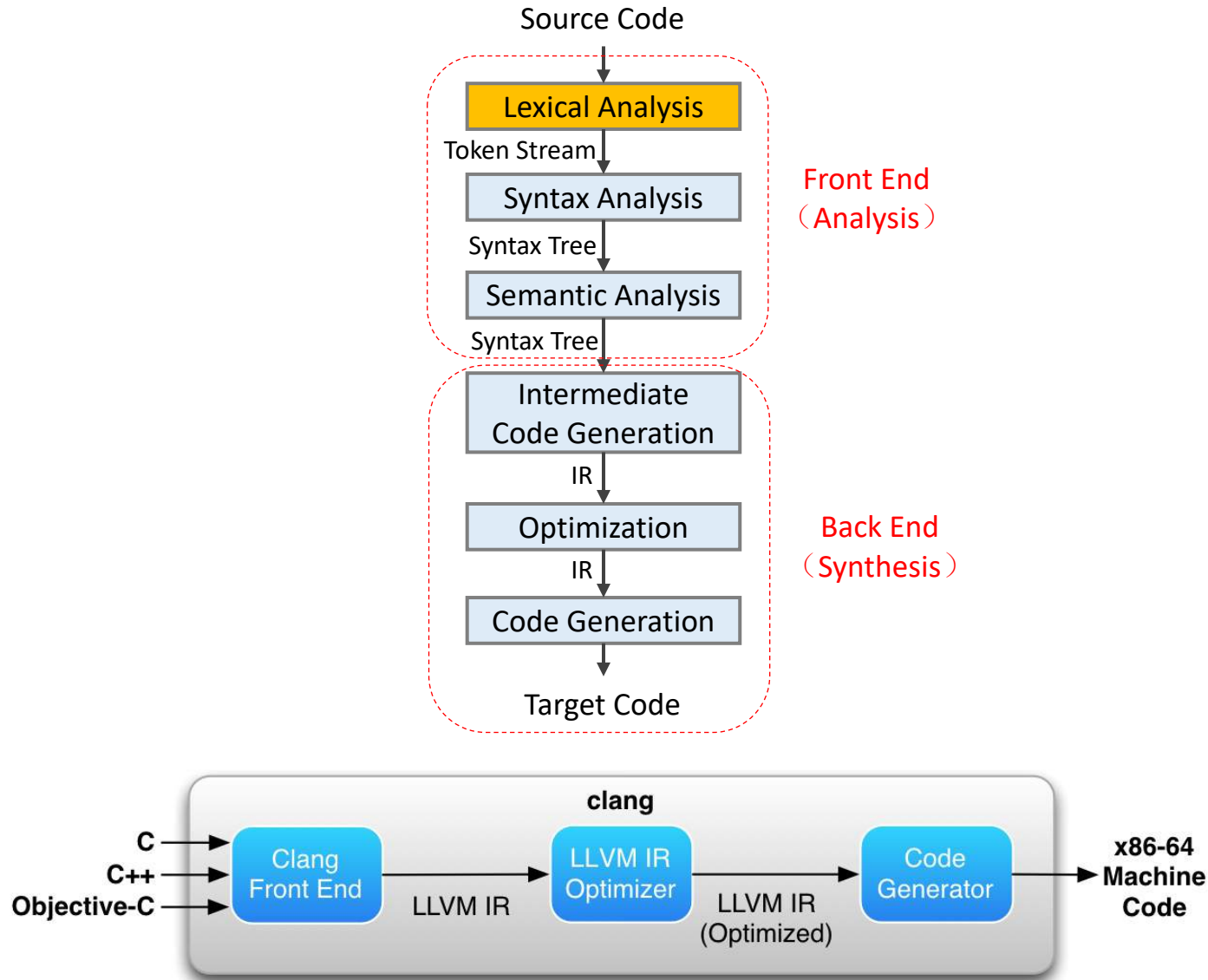
# 编译系统设计赛

- 综合运用各种知识，构思并实现一个综合性编译系统



|                |     |     |    |
|----------------|-----|-----|----|
| 全国大学生计算机系统能力大赛 | 国家级 | 特等奖 | 80 |
|                |     | 一等奖 | 80 |
|                |     | 二等奖 | 20 |

# Structure of a Typical Compiler[结构]



# Example

```
void main(){
  int a, b, c;
  if (b == c)
    return 1;
}
```

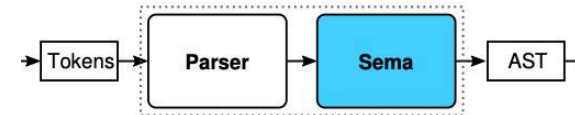
\$clang -cc1 -dump-tokens test.c

```
void 'void' [StartOfLine] Loc=<parse.c:1:1>
identifier 'main' [LeadingSpace] Loc=<parse.c:1:6>
l_paren '(' [LeadingSpace] Loc=<parse.c:1:10>
r_paren ')' [LeadingSpace] Loc=<parse.c:1:11>
l_brace '{' [LeadingSpace] Loc=<parse.c:1:12>
int 'int' [StartOfLine] [LeadingSpace] Loc=<parse.c:2:3>
identifier 'a' [LeadingSpace] Loc=<parse.c:2:7>
comma ',' [LeadingSpace] Loc=<parse.c:2:8>
identifier 'b' [LeadingSpace] Loc=<parse.c:2:10>
comma ',' [LeadingSpace] Loc=<parse.c:2:11>
identifier 'c' [LeadingSpace] Loc=<parse.c:2:13>
semi ';' [LeadingSpace] Loc=<parse.c:2:14>
if 'if' [StartOfLine] [LeadingSpace] Loc=<parse.c:3:3>
l_paren '(' [LeadingSpace] Loc=<parse.c:3:6>
identifier 'b' [LeadingSpace] Loc=<parse.c:3:7>
equalequal '==' [LeadingSpace] Loc=<parse.c:3:9>
identifier 'c' [LeadingSpace] Loc=<parse.c:3:12>
r_paren ')' [LeadingSpace] Loc=<parse.c:3:13>
return 'return' [StartOfLine] [LeadingSpace] Loc=<parse.c:4:5>
numeric_constant '1' [LeadingSpace] Loc=<parse.c:4:12>
semi ';' [LeadingSpace] Loc=<parse.c:4:13>
r_brace '}' [StartOfLine] Loc=<parse.c:5:1>
eof '' [LeadingSpace] Loc=<parse.c:5:2>
```

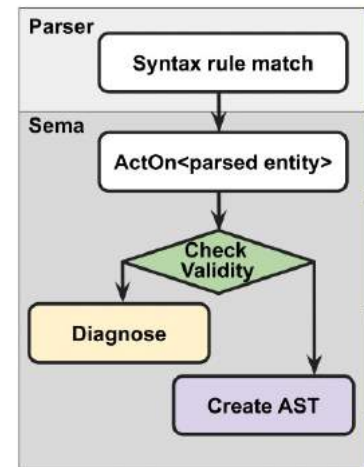


\$clang -Xclang -ast-dump -fsyntax-only test.c

```
-FunctionDecl 0x27999470 <parse.c:1:1, line:5:1> line:1:6 main 'void ()'
-CompoundStmt 0x27999800 <col:12, line:5:1>
  -DeclStmt 0x279996f8 <line:2:3, col:14>
    -VarDecl 0x27999570 <col:3, col:7> col:7 a 'int'
    -VarDecl 0x279995f0 <col:3, col:10> col:10 used b 'int'
    -VarDecl 0x27999670 <col:3, col:13> col:13 used c 'int'
  -IfStmt 0x279997e8 <line:3:3, line:4:12>
    -BinaryOperator 0x27999780 <line:3:7, col:12> 'int' '=='
      -ImplicitCastExpr 0x27999750 <col:7> 'int' <LValueToRValue>
        -DeclRefExpr 0x27999710 <col:7> 'int' lvalue Var 0x279995f0 'b' 'int'
      -ImplicitCastExpr 0x27999768 <col:12> 'int' <LValueToRValue>
        -DeclRefExpr 0x27999730 <col:12> 'int' lvalue Var 0x27999670 'c' 'int'
    -ReturnStmt 0x279997d8 <line:4:5, col:12>
      -ImplicitCastExpr 0x279997c0 <col:12> 'void' <ToVoid>
      -IntegerLiteral 0x279997a0 <col:12> 'int' 1
```

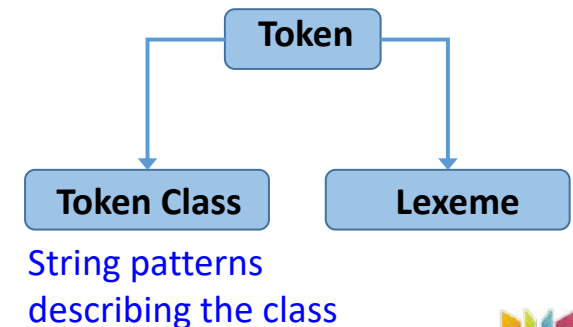
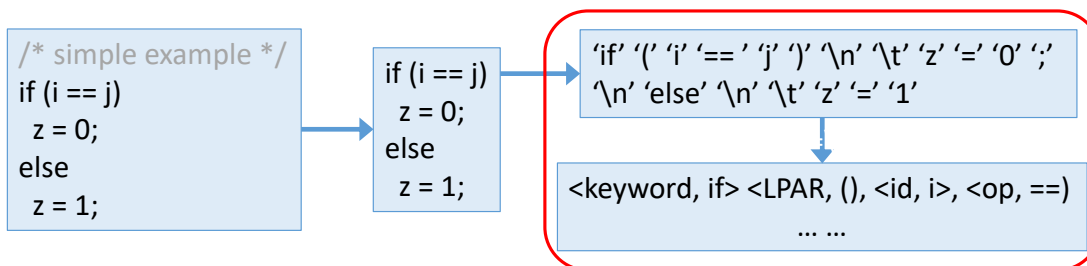


Sema is tight coupling with parser



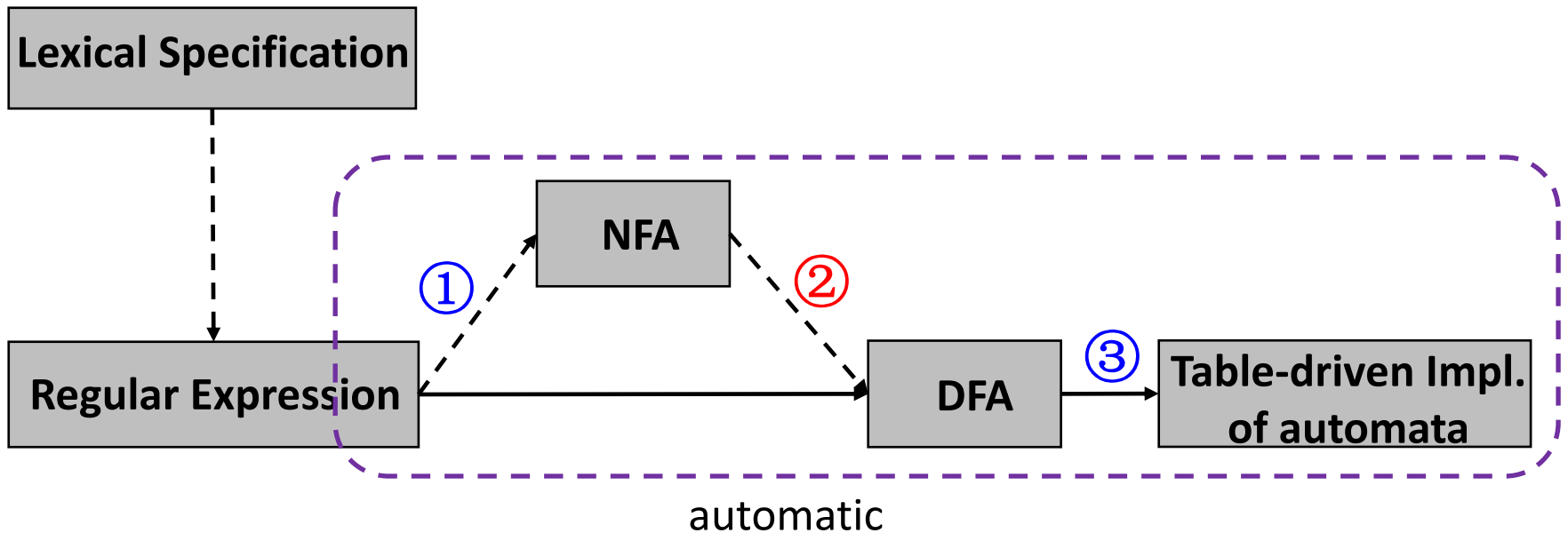
# Lexical Analysis

- **Workflow**
  - Partition the input character stream to lexemes
  - Identify the token class of each lexeme
- **Regular Expression** is a good way to specify tokens
  - Simple yet powerful (able to express patterns)
- **Finite Automata** is to construct a token recognizer for languages given by regular expressions
  - A program for classifying tokens (accept, reject)



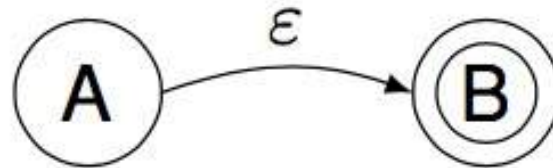
# The Conversion Flow[转换流程]

- Outline: RE  $\rightarrow$  NFA  $\rightarrow$  DFA  $\rightarrow$  Table-drive Implementation
  - ③ Converting DFAs to table-driven implementations
  - ① Converting REs to NFAs
  - ② Converting NFAs to DFAs



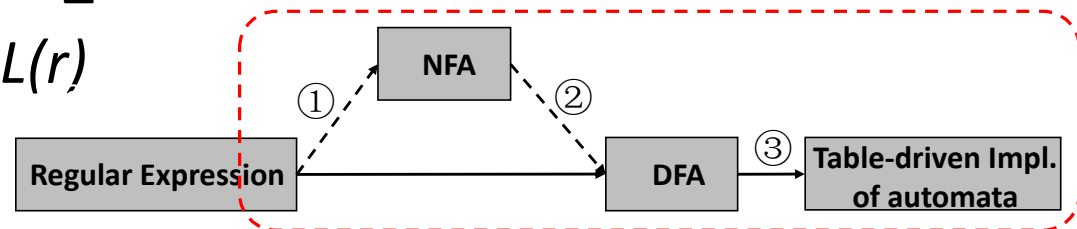
# RE $\rightarrow$ NFA

- NFA can have  $\epsilon$ -moves
  - Edges labelled with  $\epsilon$
  - Move from state  $A$  to state  $B$  without reading any input



- **M-Y-T algorithm** (Thompson's construction) to convert any RE to an NFA that defines the same language [正则表达式转换到自动机]
  - Input: RE  $r$  over alphabet  $\Sigma$
  - Output: NFA accepting  $L(r)$

McNaughton-Yamada-Thompson



# RE $\rightarrow$ NFA (cont.)

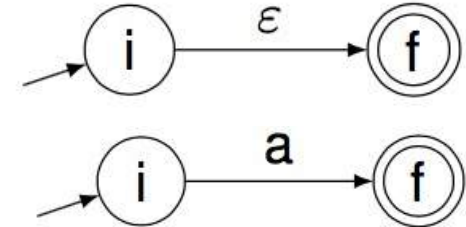
- Step 1: processing atomic REs

- $\epsilon$  expression[空]

- $i$  is a new state, the start state of NFA

- $f$  is another new state, the accepting state of NFA

- Single character RE  $a$ [单字符]

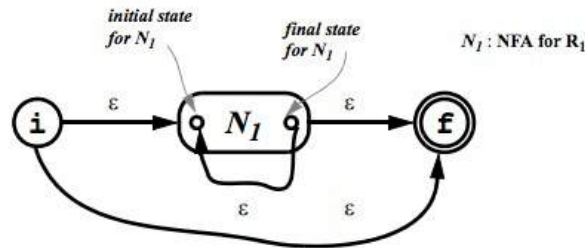
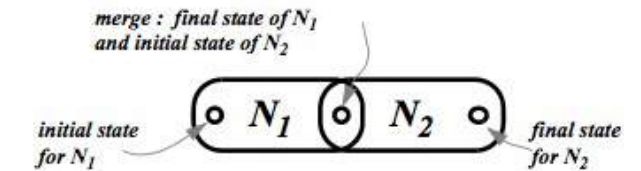
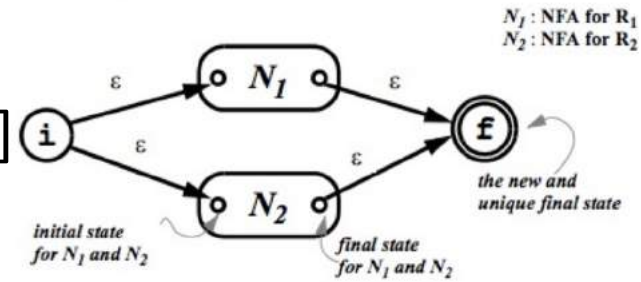


- Step 2: processing compound REs[组合]

- $R = R_1 \mid R_2$

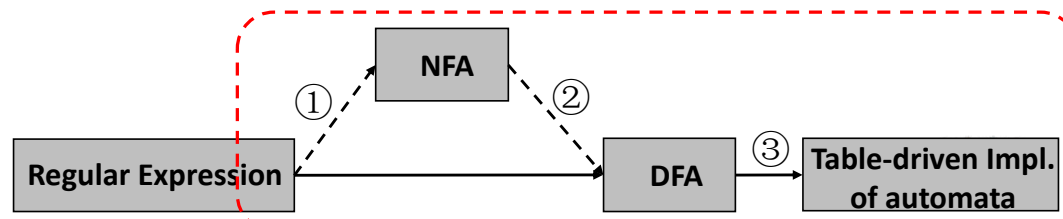
- $R = R_1 R_2$

- $R = R_1^*$



# NFA $\rightarrow$ DFA: Idea

- Algorithm to convert[转换算法]
  - Input: an NFA  $N$
  - Output: a DFA  $D$  accepting the same language as  $N$
- **Subset construction**[子集构建]
  - Each state of the constructed DFA corresponds to a set of NFA states[一个DFA状态对应多个NFA状态]
    - Hence, the name ‘subset construction’
  - After reading input  $a_1a_2\dots a_n$ , the DFA is in that state which corresponds to the set of states that the NFA can reach, from its start state, following paths labeled  $a_1a_2\dots a_n$





# NFA $\rightarrow$ DFA: Steps

---

- The **initial state** of the DFA is the set of all states the NFA can be in without reading any input[初始状态]
- For any state  $\{q_i, q_j, \dots, q_k\}$  of the DFA and any input  $a$ , the **next state** of the DFA is the set of all states of the NFA that can result as next states if the NFA is in any of the states  $q_i, q_j, \dots, q_k$  when it reads  $a$ [下一状态]
  - This includes states that can be reached by reading  $a$  followed by any number of  $\epsilon$ -transitions
  - Use this rule to keep adding new states and transitions until it is no longer possible to do so
- The **accepting states** of the DFA are those states that contain an accepting state of the NFA[接收状态]

# NFA $\rightarrow$ DFA: Algorithm

Initially,  $\varepsilon$ -closure( $s_0$ ) is the only state in  $Dstates$  and it is unmarked

**while** there is an unmarked state  $T$  in  $Dstates$  **do**

    mark  $T$

**for** each input symbol  $a \in \Sigma$  **do**

$U := \varepsilon$ -closure(move( $T, a$ ))

**if**  $U$  is not in  $Dstates$  **then**

            add  $U$  as an unmarked state to  $Dstates$

**end if**

$Dtran[T, a] := U$

**end do**

**end do**

- Operations on NFA states:

- $\varepsilon$ -closure( $s$ ): set of NFA states reachable from NFA state  $s$  on  $\varepsilon$ -transitions **alone**

- $\varepsilon$ -closure( $T$ ): set of NFA states reachable from some NFA state  $s$  in set  $T$  on  $\varepsilon$ -transitions **alone**;  $= \bigcup_{s \in T} \varepsilon$ -closure( $s$ )

- move( $T, a$ ): set of NFA states to which there is a transition on input symbol  $a$  from some state  $s$  in  $T$

# Minimization Algorithm

- The algorithm

- Partitioning the states of a DFA into groups of states that **cannot be distinguished (i.e., equivalent)**
- Each groups of states is then merged into a single state of the min-state DFA

- For a DFA  $(\Sigma, S, n, F, \delta)$

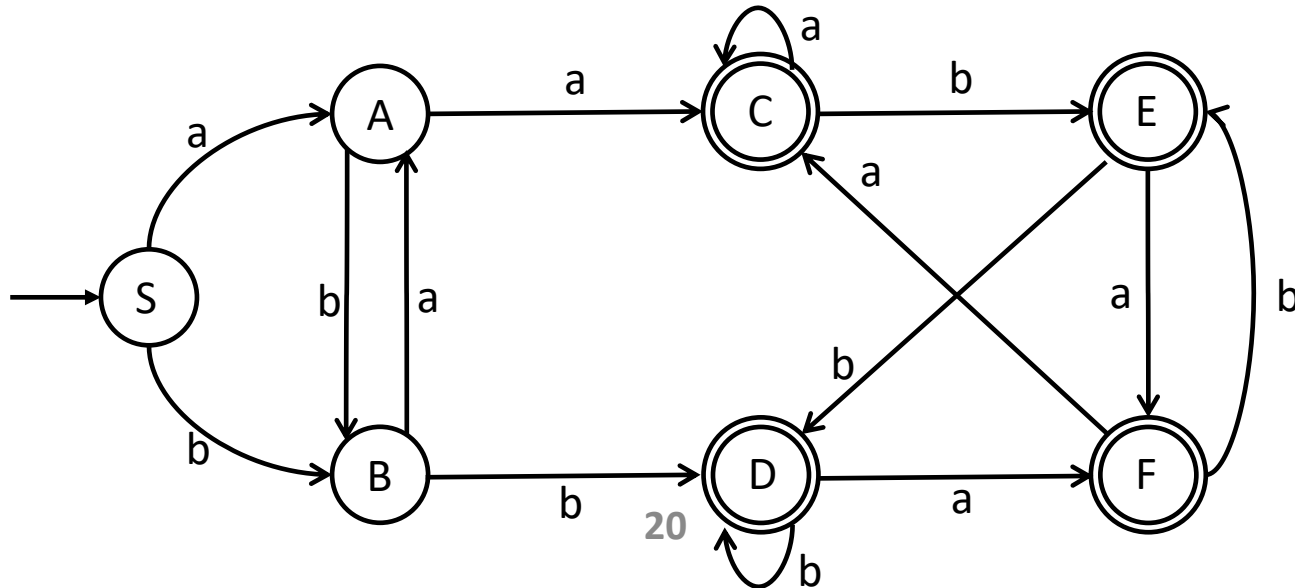
- The initial partition  $P_0$ , has two sets and  $\{S - F\}$
- Splitting a set (i.e., partitioning a set by input symbol  $\alpha$ )

```
P ← {F}, {S - F}
while (P is still changing)
  T ← {}
  for each state s ∈ P
    for each α ∈ Σ
      partition s by α into s1 & s2
      T ← T ∪ s1 ∪ s2
  if T ≠ P then
    P ← T
```

- Assume  $q_a$  and  $q_b \in S$ , and  $\delta(q_a, \alpha) = q_x$  and  $\delta(q_b, \alpha) = q_y$
- If  $q_x$  and  $q_y$  are not in the same set, then  $S$  must be split (i.e.,  $\alpha$  splits  $S$ )
- One state in the final DFA cannot have two transitions on  $\alpha$

# Example

- P0:  $s_1 = \{S, A, B\}$ ,  $s_2 = \{C, D, E, F\}$
- For  $s_1$ , further splits into  $\{S\}$ ,  $\{A\}$ ,  $\{B\}$ 
  - a:  $S \rightarrow A \in s_1$ ,  $A \rightarrow C \in s_2$ ,  $B \rightarrow A \in s_1 \Rightarrow$  a distincts  $s_1 \Rightarrow \{S, B\}$ ,  $\{A\}$
  - b:  $S \rightarrow B \in s_1$ ,  $A \rightarrow B \in s_1$ ,  $B \rightarrow D \in s_2 \Rightarrow$  b distincts  $s_1 \Rightarrow \{S\}$ ,  $\{B\}$ ,  $\{A\}$
- For  $s_2$ , all states are equivalent
  - a:  $C \rightarrow C \in s_2$ ,  $D \rightarrow F \in s_2$ ,  $E \rightarrow F \in s_2$ ,  $F \rightarrow C \in s_2 \Rightarrow$  a doesn't
  - b:  $C \rightarrow E \in s_2$ ,  $D \rightarrow D \in s_2$ ,  $E \rightarrow D \in s_2$ ,  $F \rightarrow E \in s_2 \Rightarrow$  b doesn't

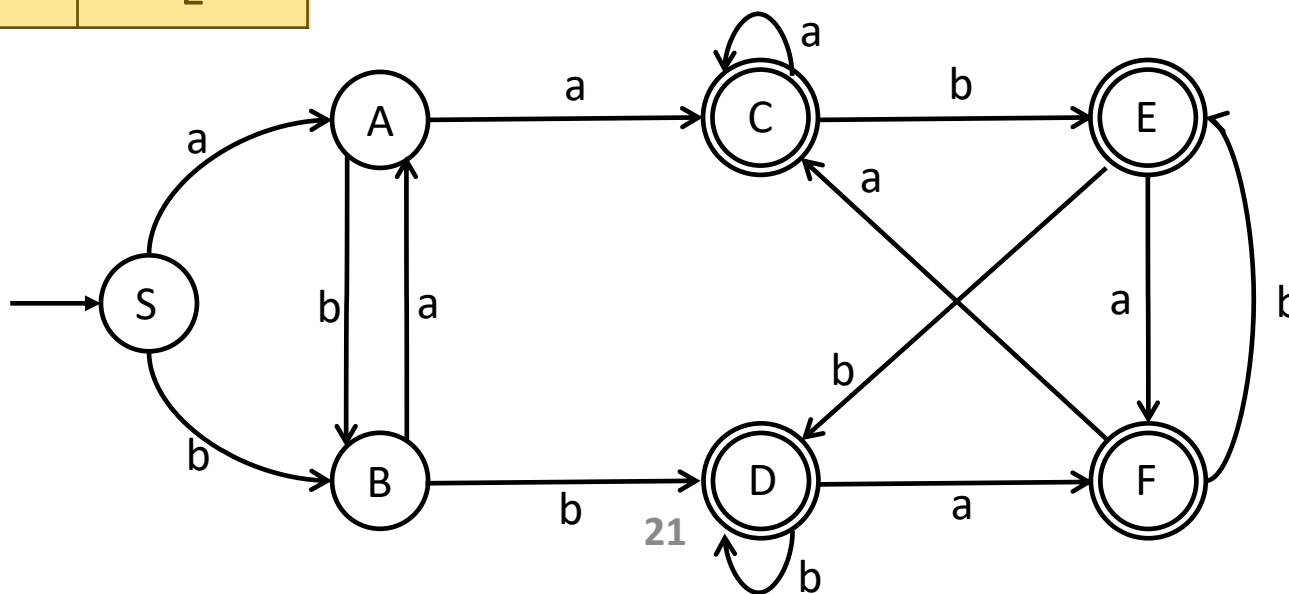


# Example (cont.)

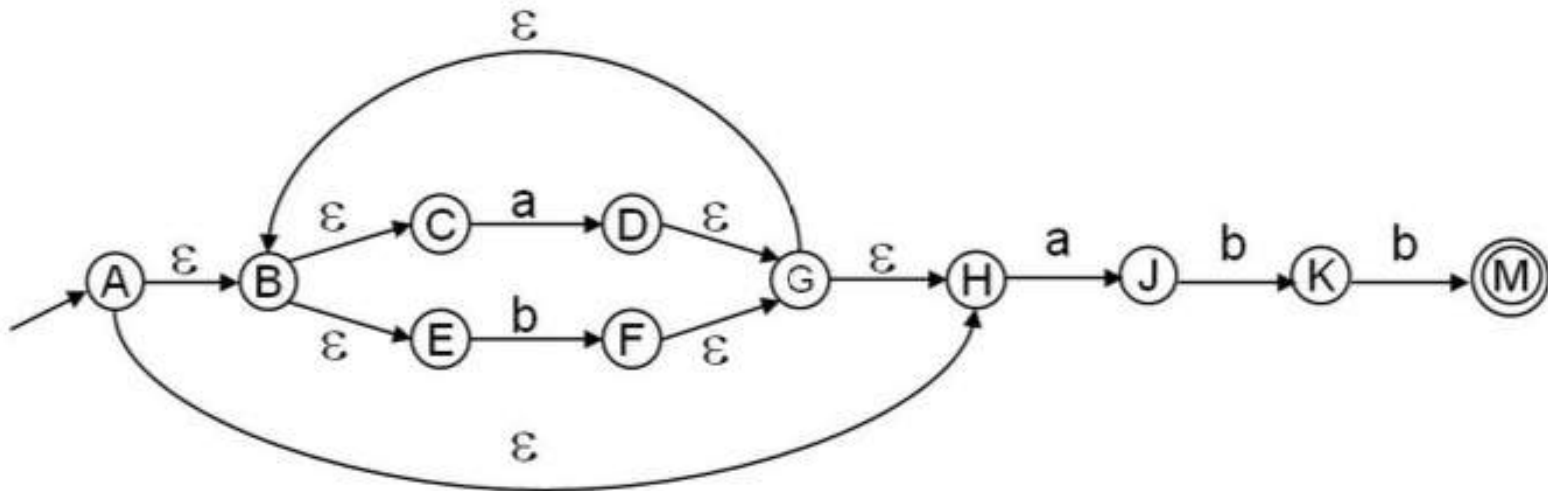
|   | a | b |
|---|---|---|
| S | A | B |
| A | C | B |
| B | A | D |
| C | C | E |
| D | F | D |
| E | F | D |
| F | C | E |

|    | a | b |
|----|---|---|
| S  | A | B |
| A  | C | B |
| B  | A | D |
| CF | C | E |
| DE | F | D |

|      | a  | b  |
|------|----|----|
| S    | A  | B  |
| A    | C  | B  |
| B    | A  | D  |
| CFDE | CF | DE |



# NFA $\rightarrow$ DFA: More Example

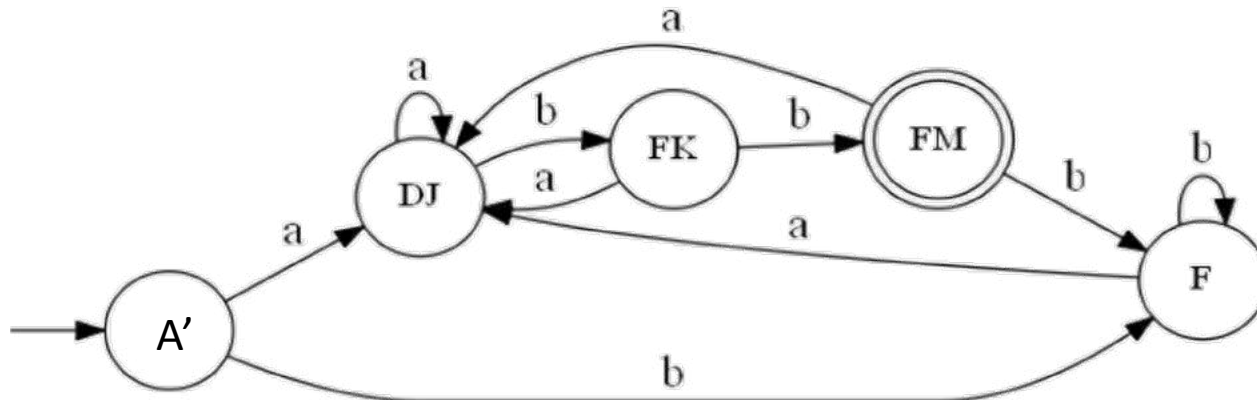


- Start state of the equivalent DFA
  - $\epsilon$ -closure(A) = {A, B, C, E, H} = A'
- $\epsilon$ -closure(move(A', a)) =  $\epsilon$ -closure({D, J}) = {B, C, D, E, H, G, J} = B'
- $\epsilon$ -closure(move(A', b)) =  $\epsilon$ -closure({F}) = {B, C, E, F, G, H} = C'
- ... ..

# NFA $\rightarrow$ DFA: More Example (cont.)

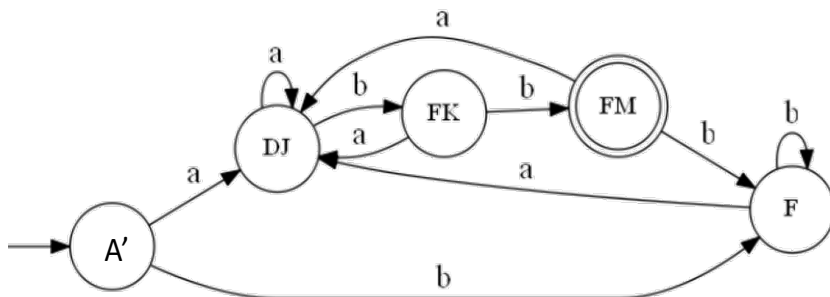
|    | a  | b  |
|----|----|----|
| A' | DJ | F  |
| DJ | DJ | FK |
| F  | DJ | F  |
| FK | DJ | FM |
| FM | DJ | F  |

- Is the DFA minimal?
  - States A' and F should be merged
- Should we merge states A' and FM?
  - NO. A' and FM are in different sets from the very beginning (FM is accepting, A' is not).



# NFA $\rightarrow$ DFA: More Example (cont.)

- P0:  $s_1 = \{A', DJ, FK, F\}$ ,  $s_2 = \{FM\}$
- For  $s_1$ , further splits into  $\{A', DJ, F\}$ ,  $\{FK\}$ 
  - a:  $A' \rightarrow DJ \in s_1$ ,  $DJ \rightarrow DJ \in s_1$ ,  $FK \rightarrow DJ \in s_1$ ,  $F \rightarrow DJ \in s_1 \Rightarrow$  a doesn't distinct
  - b:  $A' \rightarrow F \in s_1$ ,  $DJ \rightarrow FK \in s_1$ ,  $FK \rightarrow FM \in s_2$ ,  $F \rightarrow F \in s_1 \Rightarrow$  b distincts  $s_1 \Rightarrow s_{11} = \{A', DJ, F\}$ ,  $s_{12} = \{FK\}$
- For  $s_{11}$ , further splits into  $\{A', F\}$ ,  $\{DJ\}$ 
  - a:  $A' \rightarrow DJ \in s_{11}$ ,  $DJ \rightarrow DJ \in s_{11}$ ,  $F \rightarrow DJ \in s_{11} \Rightarrow$  a doesn't distinct
  - b:  $A' \rightarrow F \in s_{11}$ ,  $DJ \rightarrow FK \in s_{12}$ ,  $F \rightarrow DJ \in s_{11} \Rightarrow$  b distincts  $s_{11} \Rightarrow s_{111} = \{A', F\}$ ,  $s_{112} = \{DJ\}$
- For  $s_{111}$ , impossible to further split
- Final states:  $S_{111} = \{A', F\}$ ,  $S_{112} = \{DJ\}$ ,  $S_{12} = \{FK\}$ ,  $S_2 = \{FM\}$

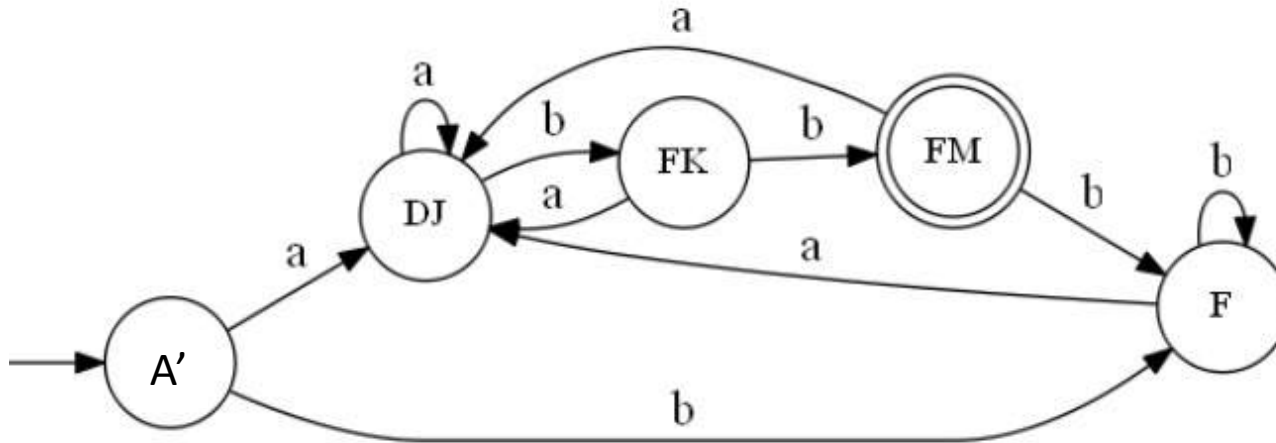


|    | a  | b  |
|----|----|----|
| A' | DJ | F  |
| DJ | DJ | FK |
| F  | DJ | F  |
| FK | DJ | FM |
| FM | DJ | F  |

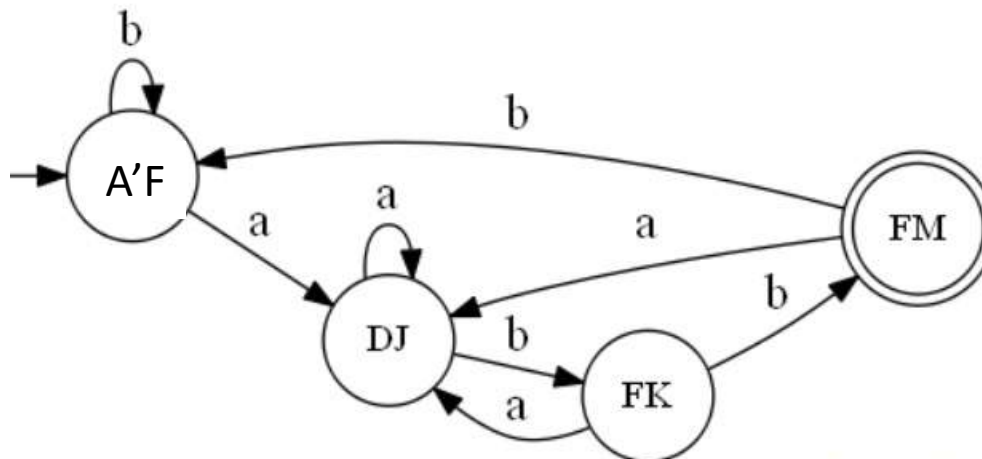


# NFA $\rightarrow$ DFA: More Example (cont.)

- Original DFA: before merging  $A'$  and  $F$

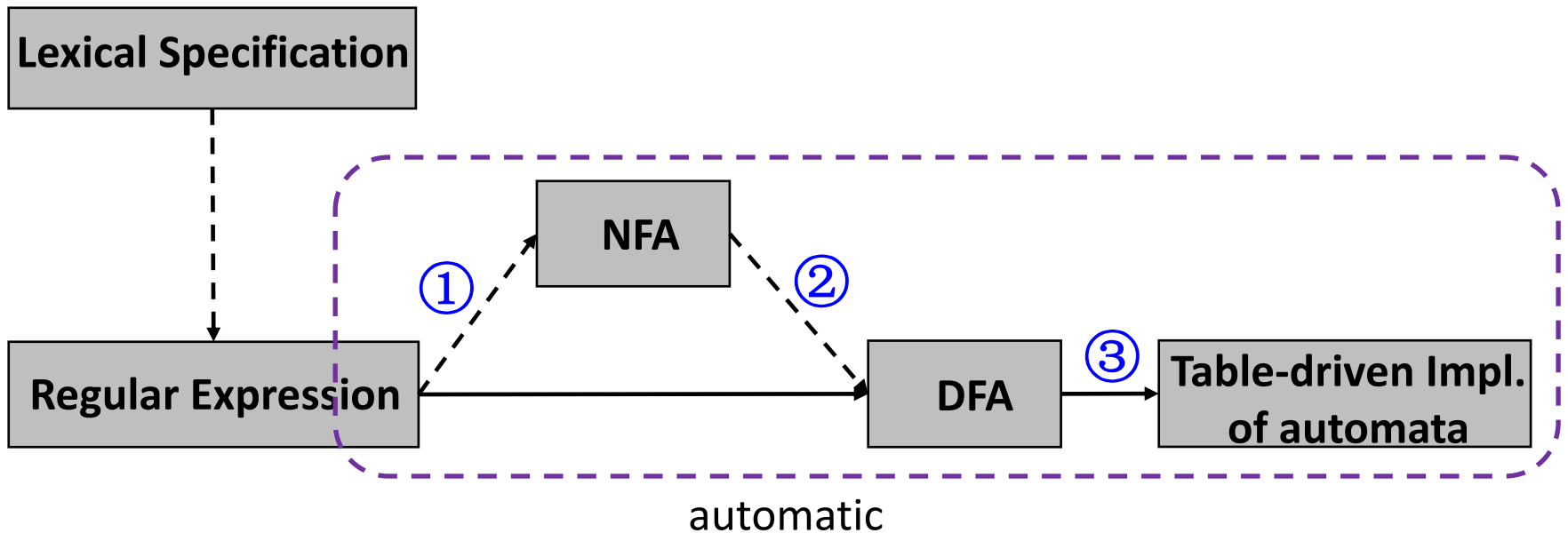


- Minimized DFA: Do you see the original RE  $(a|b)^*abb$



# The Conversion Flow[转换流程]

- Outline: RE  $\rightarrow$  NFA  $\rightarrow$  DFA  $\rightarrow$  Table-drive Implementation
  - ③ Converting DFAs to table-driven implementations
  - ① Converting REs to NFAs (M-Y-T algorithm)
  - ② Converting NFAs to DFAs (subset construction)
  - ③' DFA minimization (partition algorithm)



# NFA $\rightarrow$ DFA: Space Complexity[空间复杂度]

---

- NFA may be in many states at any time
- How many different possible states in DFA?
  - If there are  $N$  states in NFA, the DFA must be in some subset of those  $N$  states
  - How many non-empty subsets are there?
    - $2^N - 1$
- The resulting DFA has  $O(2^N)$  space complexity, where  $N$  is number of original states in NFA
  - For real languages, the NFA and DFA have about same number of states

# NFA $\rightarrow$ DFA: Time Complexity[时间复杂度]

---

- DFA execution

- Requires  $O(|X|)$  steps, where  $|X|$  is the input length
- Each step takes constant time
  - If current state is  $S$  and input is  $c$ , then read  $T[S, c]$
  - Update current state to state  $T[S, c]$
- Time complexity =  $O(|X|)$

**Deterministic:**  
unique transition

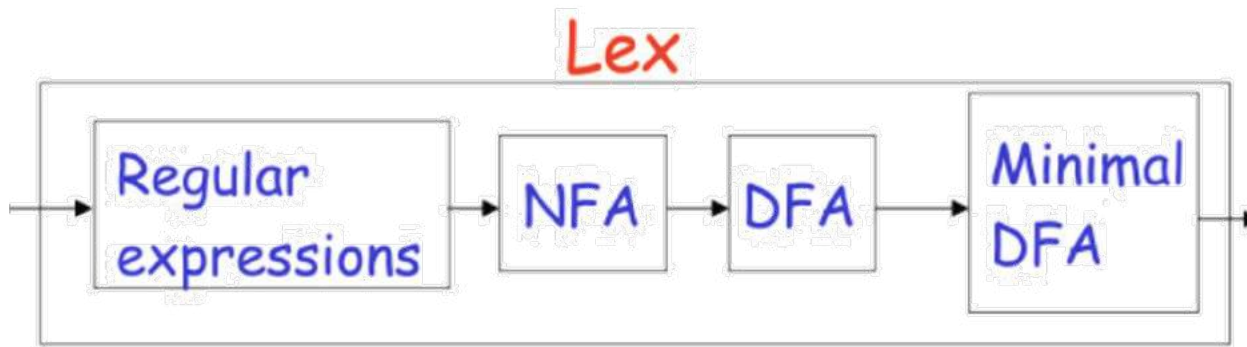
- NFA execution

- Requires  $O(|X|)$  steps, where  $|X|$  is the input length
  - Anyway, the input symbols should be completely processed
- Each step takes  $O(N^2)$  time, where  $N$  is the number of states
  - Current state is a set of potential states, up to  $N$
  - On input  $c$ , must union all  $T[S_{\text{potential}}, c]$ , up to  $N$  times
    - Each union operation takes  $O(N)$  time
- Time complexity =  $O(|X| * N^2)$

**Non-deterministic:**  
from current state,  
you can transit to any  
(including itself)

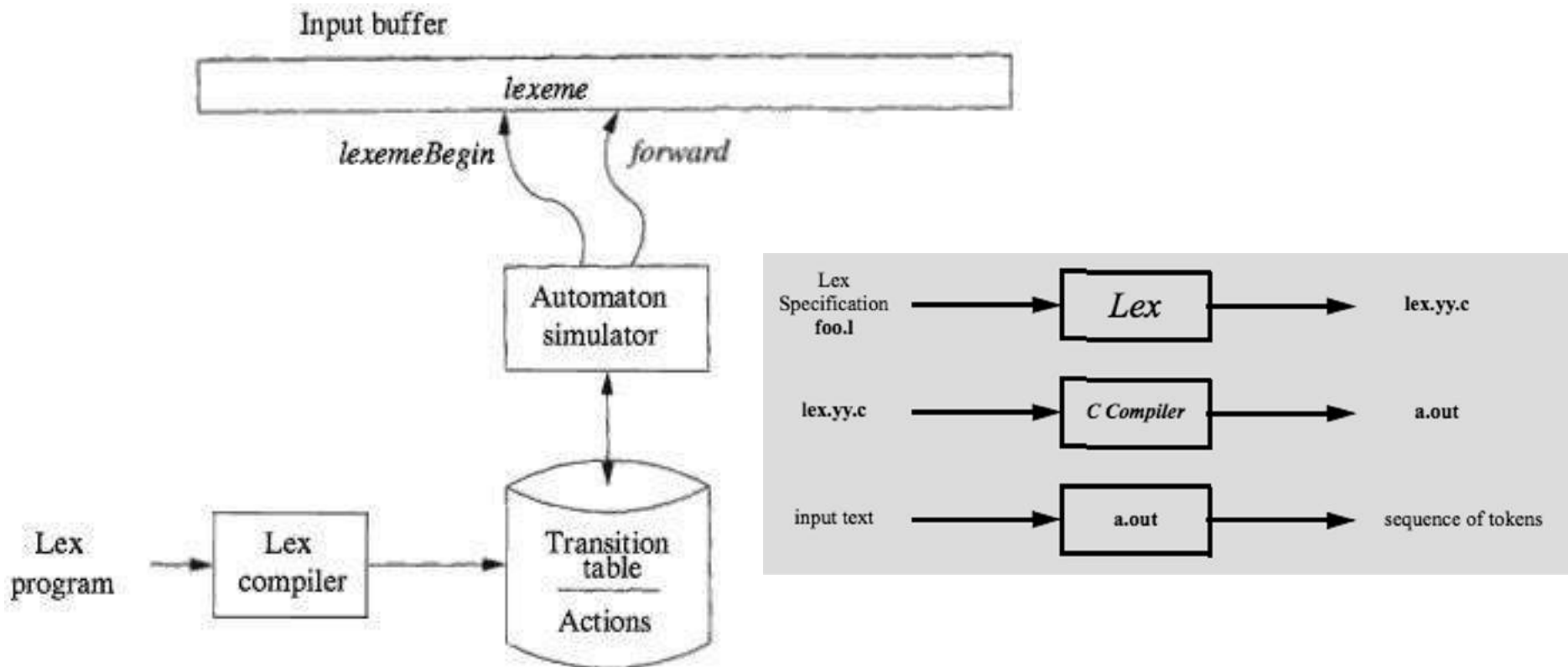
# Implementation in Practice[实际实现]

- Lex: RE  $\rightarrow$  NFA  $\rightarrow$  DFA  $\rightarrow$  Table
  - Converts regular expressions to NFA
  - Converts NFA to DFA
  - Performs DFA state minimization to reduce space
  - Generate the transition table from DFA
  - Performs table compression to further reduce space
- Most other automated lexers also choose DFA over NFA
  - Trade off space for speed



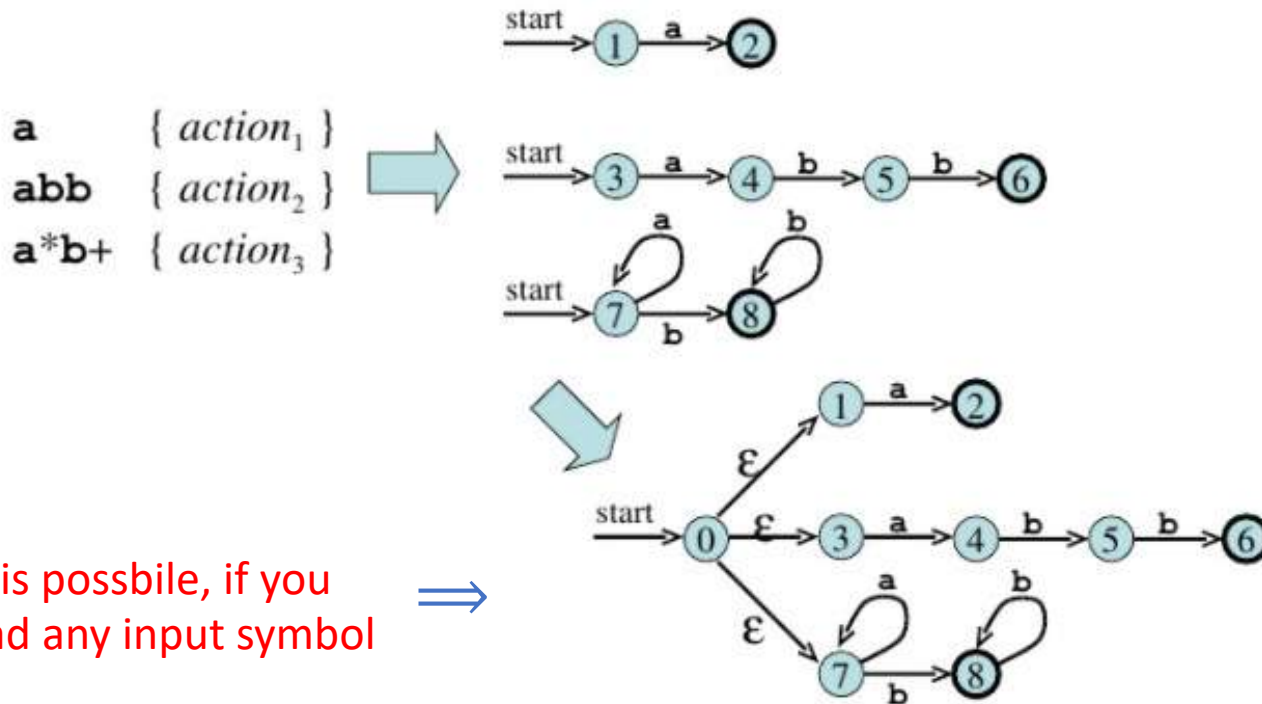
# Lexical Analyzer Generated by Lex

- A Lex program is turned into a transition table and actions, which are used by a FA simulator
- Automaton recognizes matching any of the patterns



# Lex: Example

- Three patterns, three NFAs
- Combine three NFAs into a single NFA
  - Add start state 0 and  $\epsilon$ -transitions



# Lex: Example (cont.)

```
ptn1    a
ptn2    abb
ptn3    a*b+
```

```
%%
```

```
{ptn1} { printf("\n<%s, %s>", "ptn1", yytext); }
{ptn2} { printf("\n<%s, %s>", "ptn2", yytext); }
{ptn3} { printf("\n<%s, %s>", "ptn3", yytext); }
```

```
%%
```

```
int main(){
    yylex();
    return 0;
}
```



\$flex lex.l

\$clang lex.yy.c -o mylex -ll

```
[root@aa51dde06c76:~/test# echo "aaba" | ./mylex
```

```
<ptn3, aab>
<ptn1, a>
```

```
[root@aa51dde06c76:~/test# echo "abba" | ./mylex
```

```
<ptn2, abb>
<ptn1, a>
```



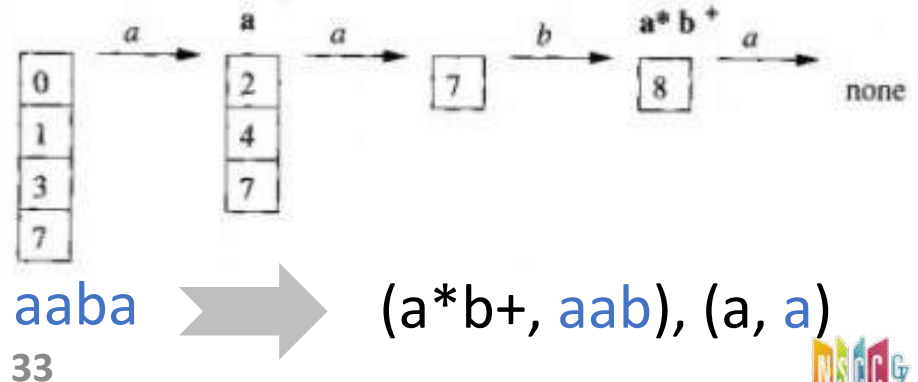
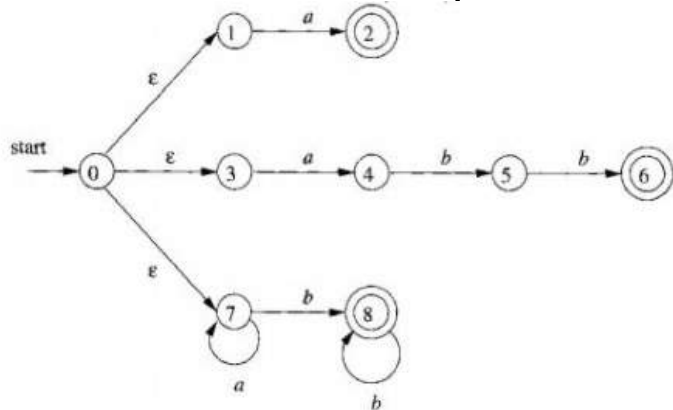
# Lex: Example (cont.)

- NFA's for lexical analyzer

- Input: **aaba**

- $\epsilon$ -closure(0) = {0, 1, 3, 7}
- Empty states after reading the fourth input symbol
  - There are no transitions out of state 8
  - Back up, looking for a set of states that include an accepting state
- State 8:  $a^*b^+$  has been matched
  - Select **aab** as the lexeme, execute action<sub>3</sub>
  - Return to parser indicating that token w/ pattern  $p_3=a^*b^+$  has been found

a  
abb  
 $a^*b^+$



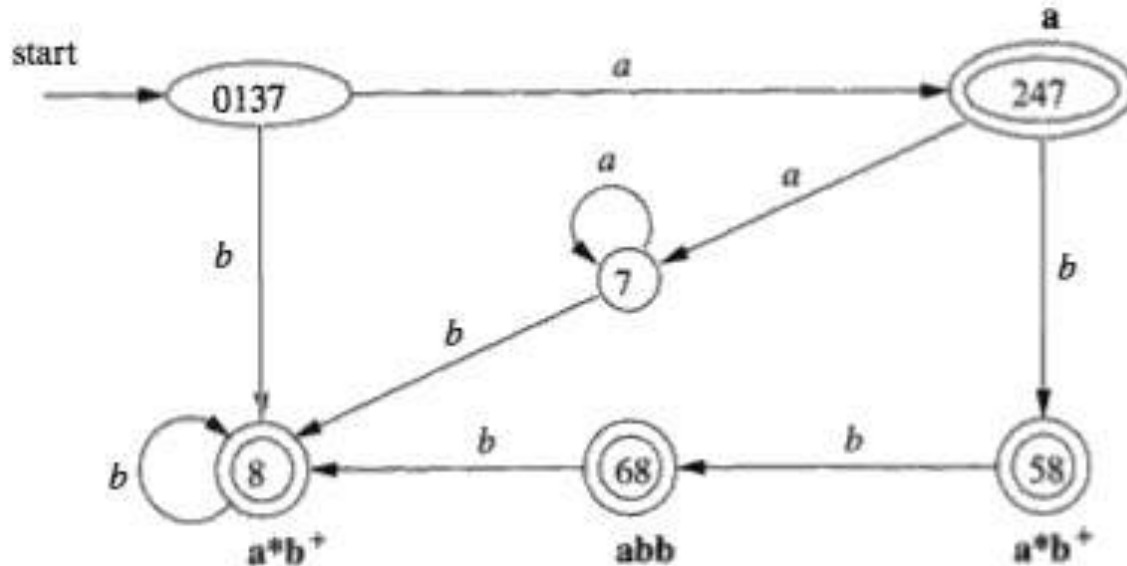
# Lex: Example (cont.)

- DFA's for lexical analyzer

- Input: **abba**

- Sequence of states entered:  $0137 \rightarrow 247 \rightarrow 58 \rightarrow 68$
- At the final  $a$ , there is no transition out of state 68
  - 68 itself is an accepting state that reports pattern  $p_2 = \mathbf{abb}$

a  
abb  
 $a^*b^+$



# How Much Should We Match? [匹配多少]

- In general, find the **longest match** possible

- We have seen examples

- One more example: input string **aabbb ...**

- Have many prefixes that match the third pattern
- Continue reading *b*'s until another *a* is met
- Report the lexeme to be the initial *a*'s followed by as many *b*'s as there are

```
a      { action1 }  
abb    { action2 }  
a*b+  { action3 }
```

- If same length, rule appearing first takes precedence

- String **abb** matches both the second and third

- We consider it as a lexeme for  $p_2$ , since that pattern listed first

```
ptn1  a  
ptn2  abb  
ptn3  a*b+
```

<ptn2, abb>

%%

```
ptn1  a  
ptn2  abb  
ptn3  a*b+
```

<ptn3, abb>

%%

```
{ptn1} { printf("\n<ptn1, %s>", "ptn1", yytext); } {ptn1} { printf("\n<ptn1, %s>", "ptn1", yytext); }  
{ptn2} { printf("\n<ptn2, %s>", "ptn2", yytext); } {ptn3} { printf("\n<ptn3, %s>", "ptn3", yytext); }  
{ptn3} { printf("\n<ptn3, %s>", "ptn3", yytext); } {ptn2} { printf("\n<ptn2, %s>", "ptn2", yytext); }
```

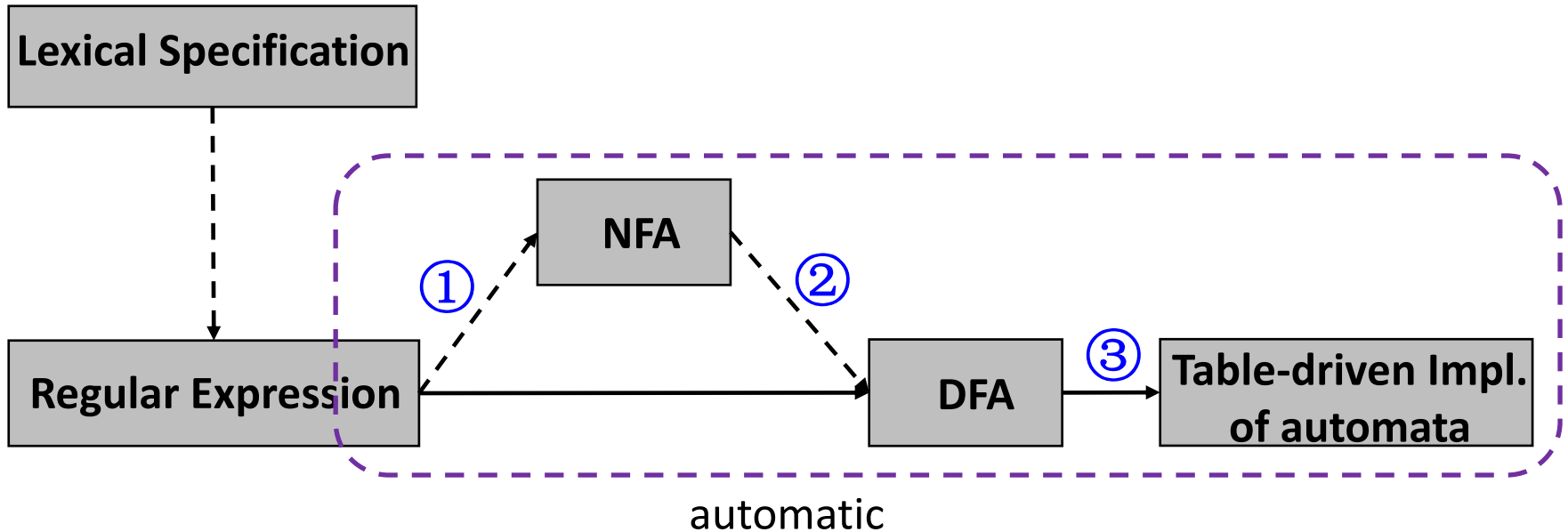
# How to Match Keywords?[匹配关键字]

---

- Example: to recognize the following tokens
  - Identifiers: `letter(letter|digit)*`
  - Keywords: `if, then, else`
- **Approach 1:** make REs for keywords and place them before REs for identifiers so that they will take precedence
  - Will result in more bloated finite state machine
- **Approach 2:** recognize keywords and identifiers using same RE but differentiate using special keyword table
  - Will result in more streamlined finite state machine
  - But extra table lookup is required
- Usually approach 2 is more efficient than 1, but you can implement approach 1 in your projects for simplicity

# The Conversion Flow[转换流程]

- Outline: RE  $\rightarrow$  NFA  $\rightarrow$  DFA  $\rightarrow$  Table-drive Implementation
  - ③ Converting DFAs to table-driven implementations
  - ① Converting REs to NFAs (M-Y-T algorithm)
  - ② Converting NFAs to DFAs (subset construction)
  - ③' DFA minimization (partition algorithm)



# Beyond Regular Languages

---

- Regular languages are **expressive enough for tokens**
  - Can express identifiers, strings, comments, etc.
- However, it is the **weakest** (least expressive) language
  - Many languages are not regular
  - C programming language is not
    - The language matching braces “{{{...}}}” is also not
  - FA cannot count # of times char encountered
    - $L = \{a^n b^n \mid n \geq 1\}$
    - Crucial for analyzing languages with nested structures (e.g. nested for loop in C language)
- We need a more powerful language for parsing
  - Later, we will discuss context-free languages (CFGs)