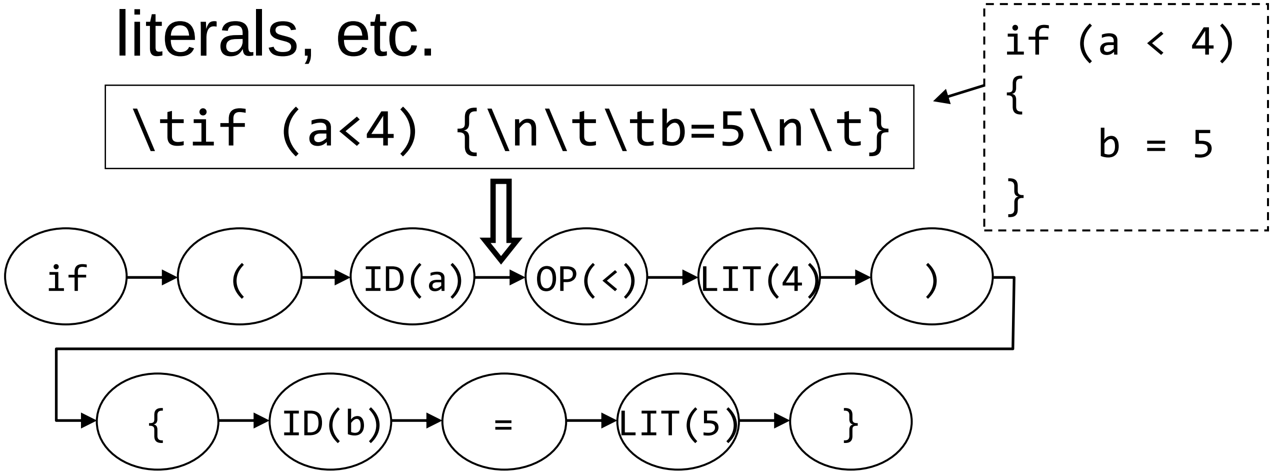
CS323: Compilers

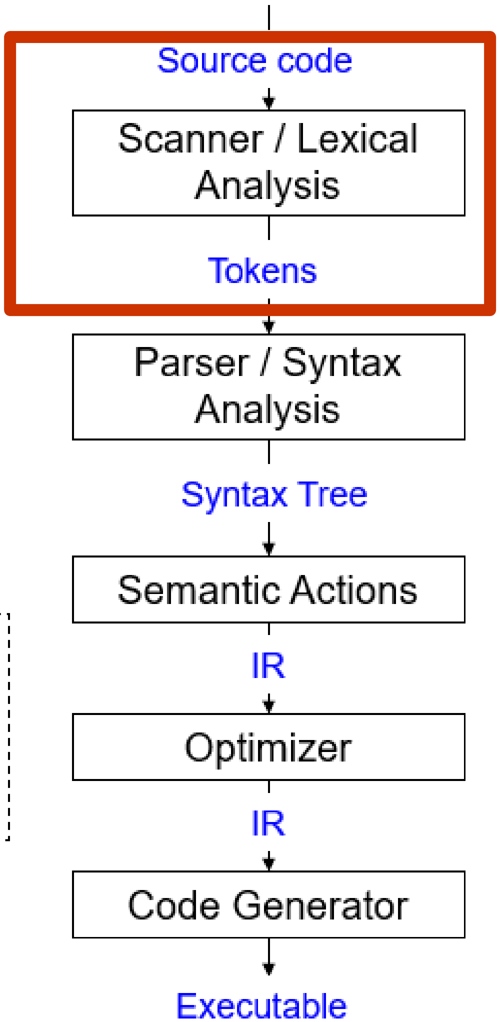
Spring 2023

Week 2: Scanners

Scanner - Overview

- Also called lexers / lexical analyzers
- Recall: scanners
  - See program text as a stream of letters
  - break input stream up into a set of tokens: Identifiers, reserved words, literals, etc.





Scanner - Motivation

- Why have a separate scanner when you can combine this with syntax analyzer (parser)?
  - Simplicity of design
    - E.g. rid parser of handling whitespaces
  - Improve compiler efficiency
    - E.g. sophisticated buffering algorithms for reading input
  - Improve compiler portability
    - E.g. handling ^M character in Linux (CR+LF in Windows)

Scanner - Tasks

1. Divide the program text into *substrings* *or* *lexemes*
   – place dividers

2. Identify the *class* of the substring identified
   – Examples of predefined categories: Identifiers, keywords, operators, etc.
      - Identifier – *strings of letters or digits starting with a letter*
      - Integer – *non-empty string of digits*
      - Keyword – *"if", "else", "for"* etc.
      - Blankspace - *t, \n, ' '*
      - Operator – *(,), <, =,* etc.

   – *Observation:* substrings can be categoriezed i.e. follow some pattern

Categorizing a Substring (English Text)

- What is the English language analogy for *class*?
  - Noun, Verb, Adjective, Article, etc.
  - In an English essay, each of these classes can have a set of strings.
  - Similarly, in a program, each class can have a set of substrings.

Exercise

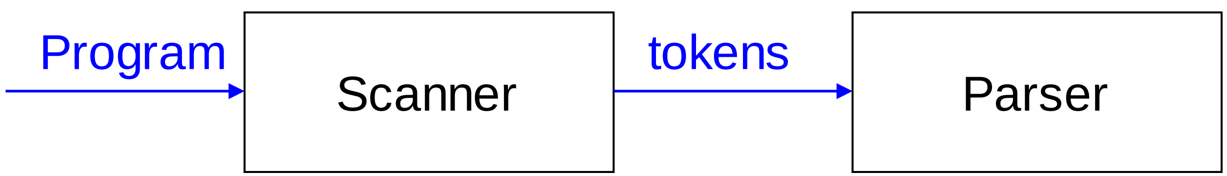
- How many tokens of class *identifier* exist in the code below?

```
for(int i=0;i<10;i++) {
      printf("hello");
}
```

Scanner Output

- A token corresponding to each lexeme
  - Token is a pair: <class, value>

A string / lexeme / substring of program text




```
E.g. int x = 0;
```

```
(Keyword, “int”),
(Identifier, “x”),
(“=”),
(Integer, “0”),
(“;”)
```

Scanners – interesting examples

- Fortran (white spaces are ignored)

  `DO 5 I = 1,25` ← DO Loop

  `DO 5 I = 1.25` ← Assignment statement

- PL/1 (keywords are not reserved)

  `DECLARE (ARG1, ARG2, . . ., ARGN);`

- C++

  Nested template: `Quad<Square<Box>> b;`

  Stream input: `std::cin >> bx;`

Scanners – interesting examples (discussion)

- How did we go about recognizing tokens in previous examples?
  - Scan left-to-right till a token is identified
  - One token at a time: continue scanning the remaining text till the next token is identified...
  - So on…

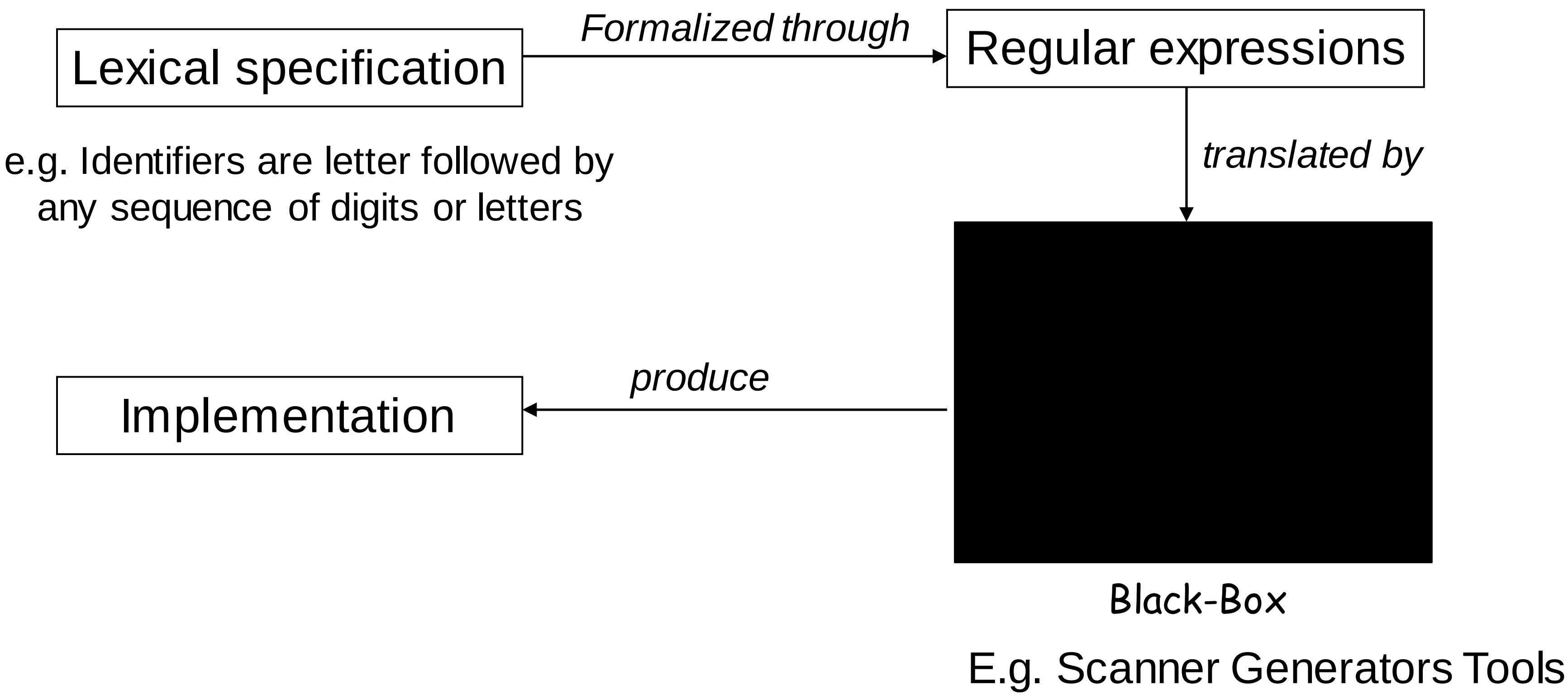
We always need to *look-ahead* to identify tokens

….but we want to minimize the amount of look-ahead done to simplify scanner implementation

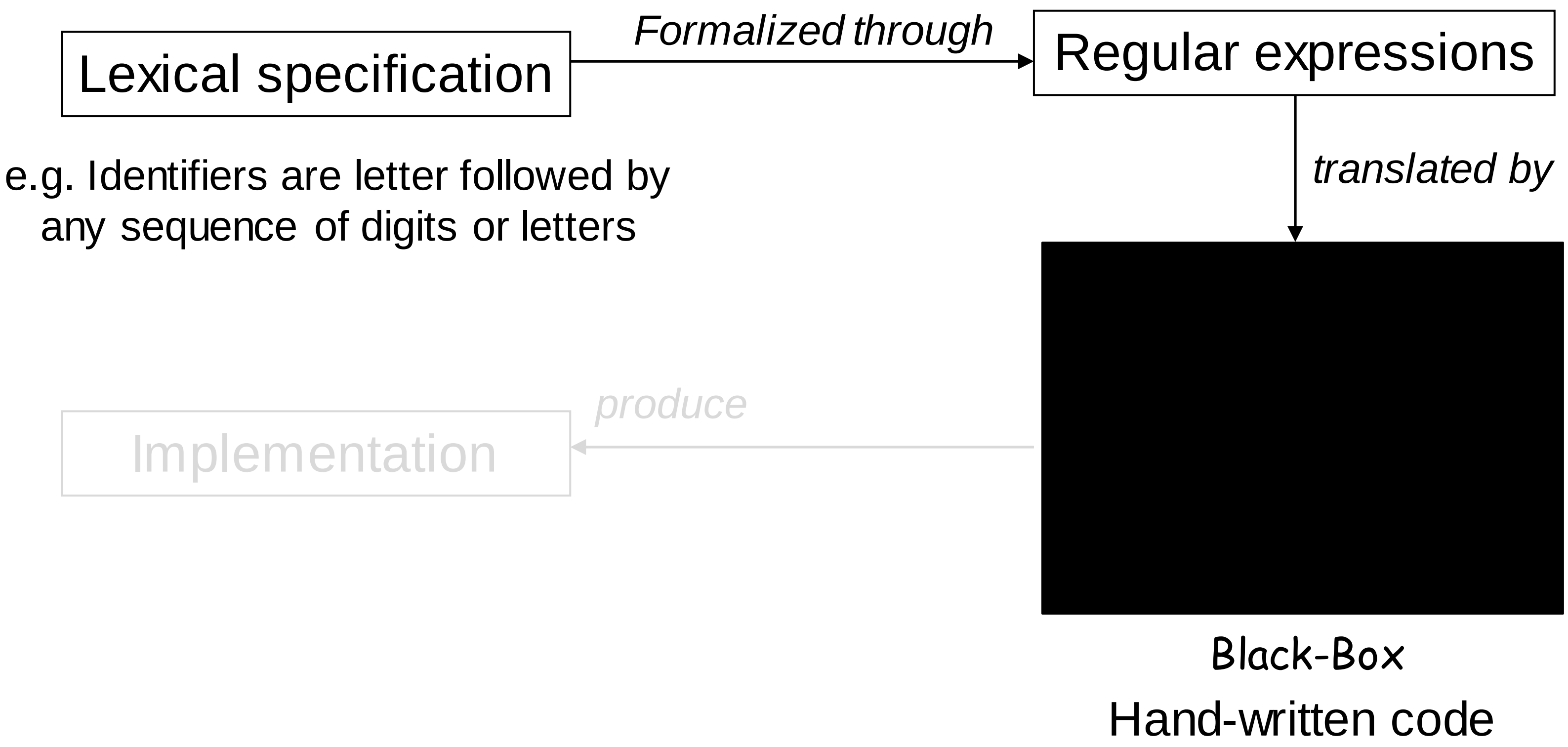
Scanners – what do we need to know?

1. How do we define tokens?
   - Regular expressions
2. How do we recognize tokens?
   - build code to find a lexeme that is a prefix and that belongs to one of the classes.
3. How do we write lexers?
   - E.g. use a lexer generator tool such as Flex

Scanner / Lexical Analyzer - flowchart



Scanner / Lexical Analyzer - flowchart



Scanner Generators

- Essentially, tools for converting regular expressions into scanners
  - `Lex (Flex)` generates C/C++ scanner program
  - `ANTLR` (ANother Tool for Language Recognition) generates Java program for translating program text (`JFlex` is a less popular option)
  - `Pylexer` is a Python-based lexical analyzer (not a scanner generator). *It just scans input, matches regexps, and tokenizes. Doesn't produce any program.*

Regular Expressions

- Used to define the structure of tokens
- Regular sets:

Informal: a set of strings defined by regular expressions

Formal: a language that can be defined by regular expressions

Start with a finite *character set* or *Vocabulary* (V). Strings are formed using this character set with the following rules:

Regular Expressions

- Strings are regular sets (with one element): pi 3.14159
  - So is the empty string: λ (ε instead)
- Concatenations of regular sets are regular: pi3.14159
  - To avoid ambiguity, can use ( ) to group regexps together
- A choice between two regular sets is regular, using |: (pi|3.14159)
- 0 or more of a regular set is regular, using *: (pi)*
- other notation used for convenience:
  - Use `Not` to accept all strings except those in a regular set
  - Use `?` to make a string optional: `x?` equivalent to `(x|λ)`
  - Use `+` to mean 1 or more strings from a set: `x+` equivalent to `xx*`
  - Use [ ] to present a range of choices: `[1-3]` equivalent to `(1|2|3)`

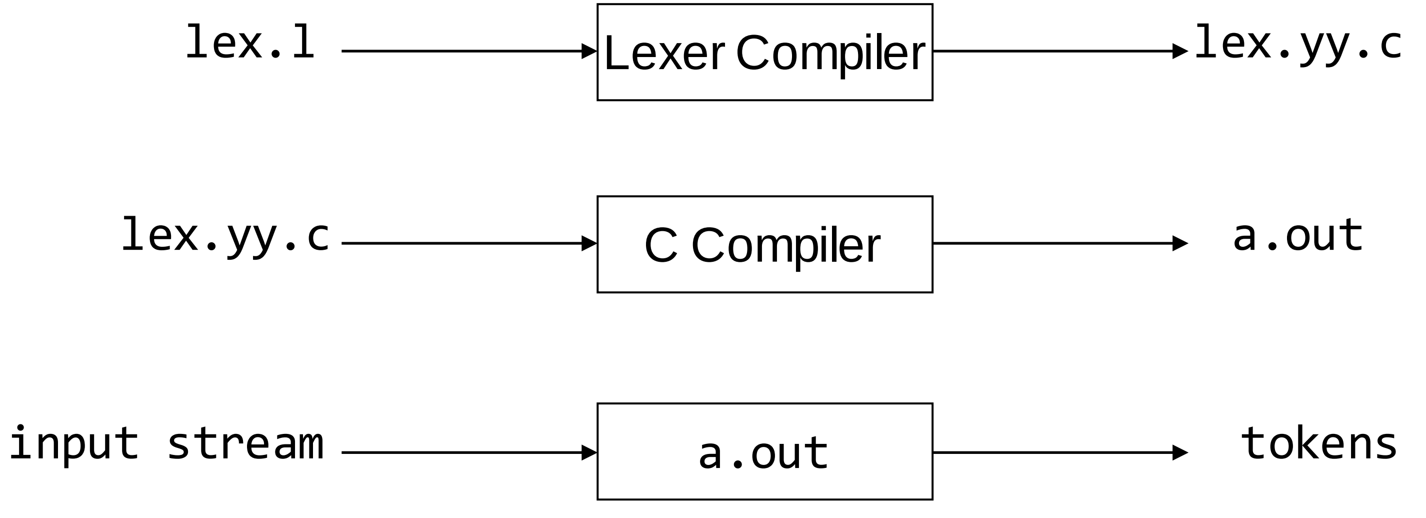
Regular Expressions for Lexical Specifications

- Digit: D = (0|1|2|3|4|5|6|7|8|9)
  alternative definition: [0-9]
- Letter: L = [A-Za-z]
- Literals (integers or floats): -?D+(.D*)?
- Identifiers: (_|L)(_|L|D)*
- Comments (as in Micro): //Not(\n)*\n
- More complex comments (delimited by ##, can use # inside comment):
  ## ((#|λ) Not(#))* ##

Lex (Flex)

- Commonly used Unix scanner generator (superseded by Flex)
- Flex is a domain specific language for writing scanners
- Features:
  - Character classes : define sets of characters (e.g., digits)
  - Token definitions : `regex {action to take}`

Lex (Flex)


lex.l
Lexer Compiler
lex.yy.c
lex.yy.c
C Compiler
a.out
input stream
a.out
tokens

Lex (Flex)

- Format of lex.l (3 parts separated by %%)

```
                                format: name definition
        Declarations                  e.g. DIGIT [0-9]

                              Refer to DIGIT here
                              using {} braces {DIGIT}
        %%                    expands to ([0-9])

        Translation rules

          format: pattern action
              e.g. {DIGIT}+ {printf("INTLITERAL");
        %%
User code mentioned here copied as is to lex.yy.c
        Auxiliary functions
```

Example: Lex (Flex)

```
DIGIT     [0-9]
ID        [a-z][a-z0-9]*

%%

{DIGIT}+  {
          printf( "An integer: %s (%d)\n", yytext,
          atoi( yytext ) );
       }

{DIGIT}+"."{DIGIT}* {
              printf( "A float: %s (%g)\n", yytext,
              atof( yytext ) );
         }

if|then|begin|end|procedure|function {
          printf( "A keyword: %s\n", yytext );
       }

{ID}      printf( "An identifier: %s\n", yytext );
```


slide courtesy: Milind Kulkarni

Lex (Flex)

- The order in which tokens are defined matters!
- Lex will match the longest possible token
  - "ifa" becomes ID(ifa), not IF ID(a)
- If two regexes both match, Lex uses the one defined first
  - "if" becomes IF, not ID(if)
- Use action blocks to process tokens as necessary
  - Convert integer/float literals to numbers
  - Remove quotes from string literals

Demo

Documentation

- Flex (manual web-version):
- Lexical Analysis With Flex, for Flex 2.6.2: Top (westes.github.io)
- Lex - A Lexical Analyzer Generator (compilertools.net)
- ANTLR

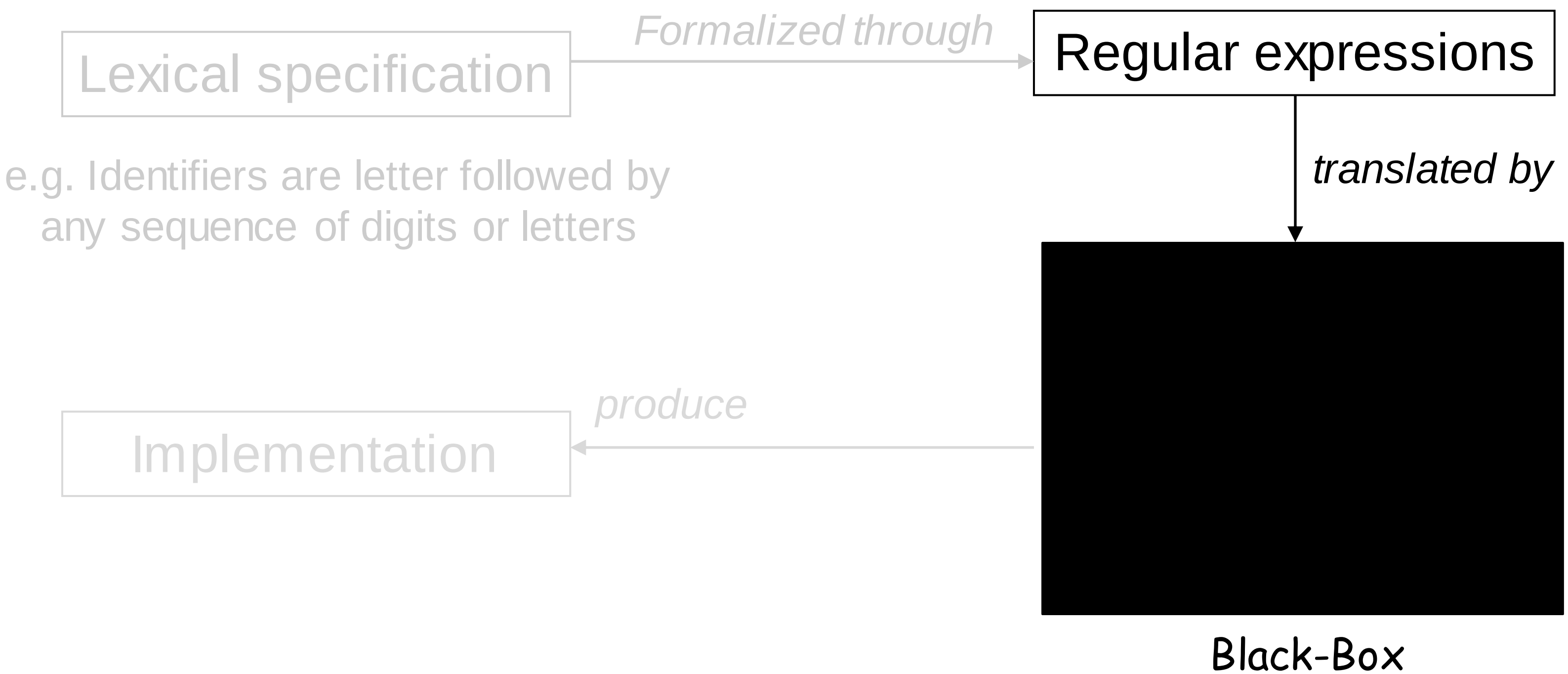
Summary

- We saw what it takes to write a scanner:
  - Specify how to identify token classes (using regexps)
  - Convert the regexps to code that identifies a *prefix* of the input program text as a *lexeme* matching one of the token classes
    - Use tools for automatic code generation (e.g. `Flex / ANTLR`)
      - *How do the tools convert regexps to code? Finite Automata*

OR

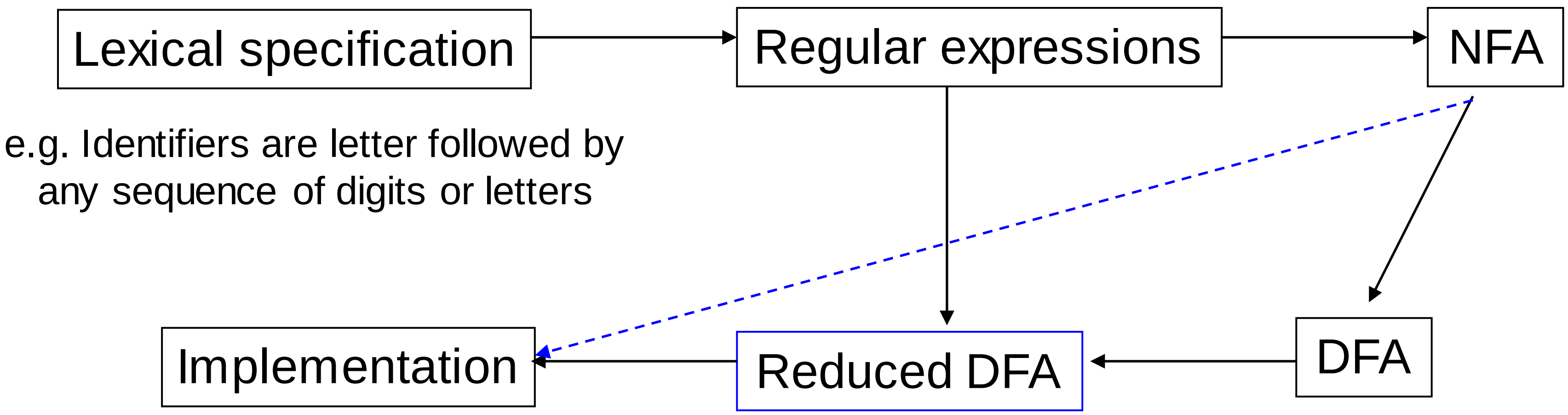
- Scanner code manually

Scanner- Implementation



How does a tool such as Flex generate code?

Scanner - flowchart

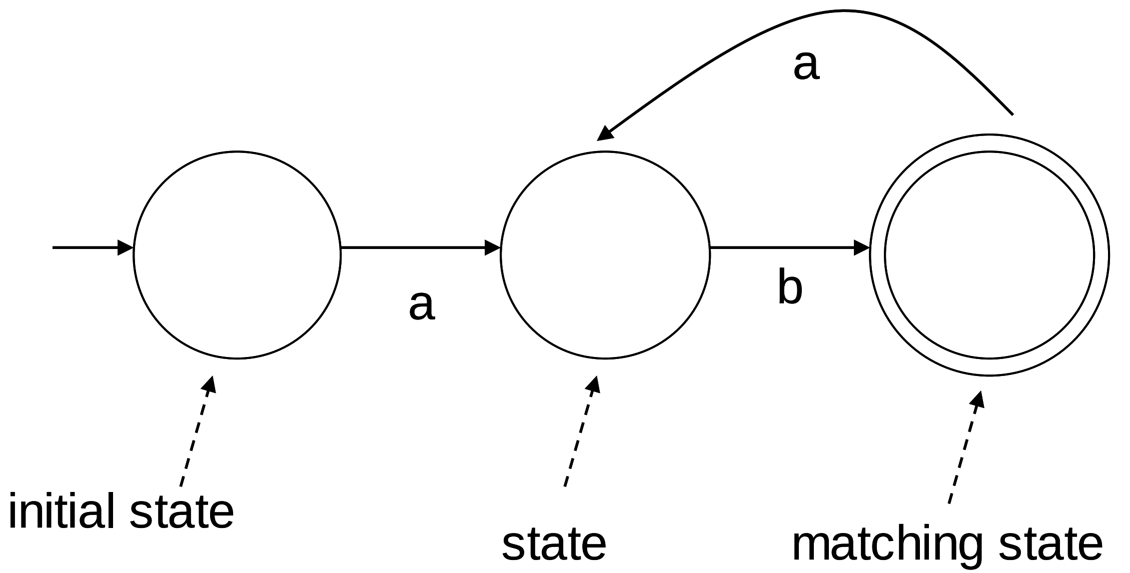


Finite Automata

- Another formal way to describe sets of strings (just like regular expressions)
- Also known as finite state machines / automata
- Reads a string, either recognizes it or not
- Two Features:
  - State: initial, matching / final / accepting, non-matching
  - Transition: a move from one state to another

Finite Automata

- Regular expressions and FA are equivalent*

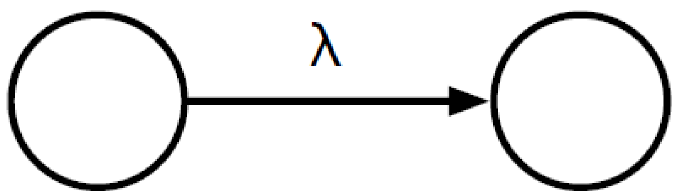


Exercise: what is the equivalent regular expression for this FA?

* Ignoring the *`empty`* regular language

λ transitions

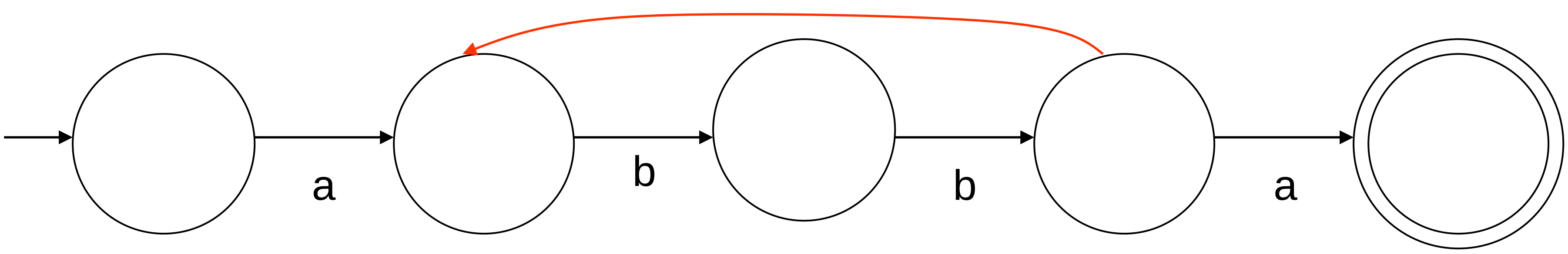
- Transitions between states that aren't triggered by seeing another character
  - Can *optionally* take the transition, but do not have to
  - Can be used to link states together



Think of this as an arrow to a state without a label

Non-deterministic Finite Automata

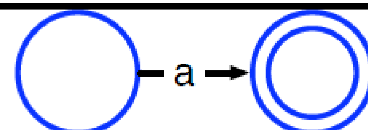
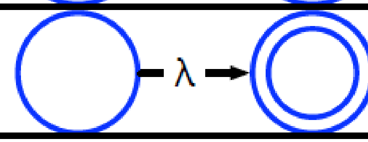
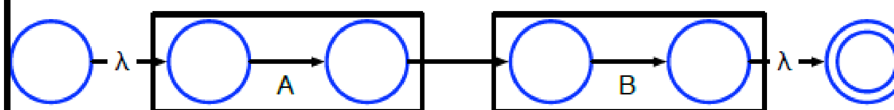
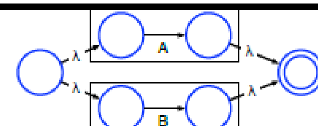
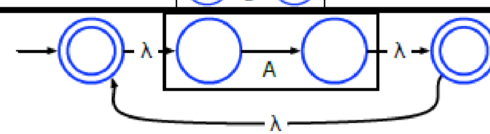
- A FA is non-deterministic if, from one state reading a single character could result in transition to multiple states (or has λ transitions)
- Sometimes regular expressions and NFAs have a close correspondence



≡

a(bb)+a

Building a FA from a regexp

Expression	FA
a	
λ	
AB	
A|B	
A*	

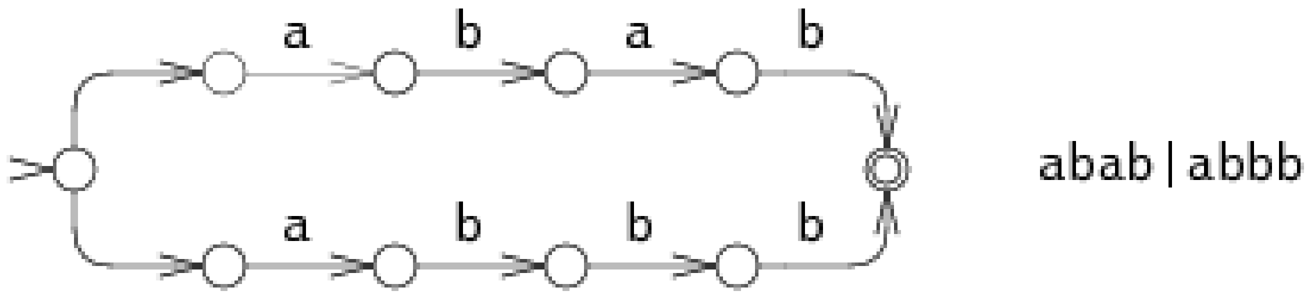
Mini-exercise: how do we build an FA that accepts Not(A)?

What about A? (? as in optional)

"Running" an NFA

- Intuition: take every possible path through an NFA
  - Think: parallel execution of NFA
  - Maintain a "pointer" that tracks the current state
  - Every time there is a choice, "split" the pointer, and have one pointer follow each choice
  - Track each pointer simultaneously
    - If a pointer gets stuck, stop tracking it
    - If any pointer reaches an accept state at the end of input, accept

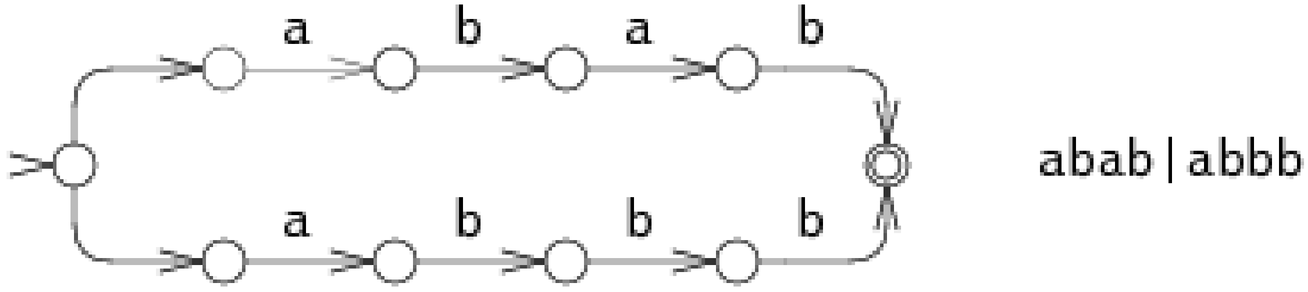
Running an NFA - Example



- NFAs are concise but slow
- Example:
  - Running the NFA for input string abbb requires exploring all execution paths

*Example taken from https://swtch.com/~rsc/regexp/regexp1.html

Running an NFA - Example



- NFAs are concise but slow
- Example:
  - Running the NFA for input string abbb requires exploring all execution paths
  - **Optimization: run through the execution paths in parallel**
    - ***Complicated. Can we do better?***

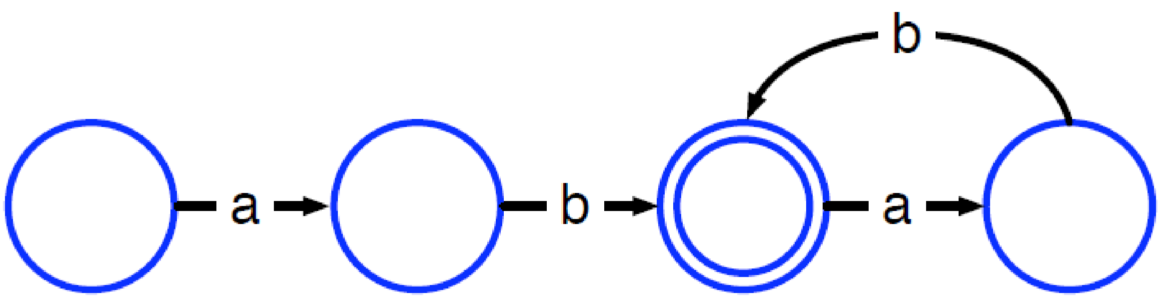
Deterministic Finite Automata

- Each possible input character read leads to at most one new state
  - Can convert NFAs to *deterministic* finite automata (DFAs)
    - No choices — never a need to “split” pointers
  - Initial idea: simulate NFA for all possible inputs, any time there is a new configuration of pointers, create a state to capture it
    - Pointers at states 1, 3 and 4 → new state {1, 3, 4}
  - Trying all possible inputs is impractical; instead, for any new state, explore all possible *next* states (that can be reached with a single character)
  - Process ends when there are no new states found
  - This can result in very large DFAs!

DFA reduction

- DFAs built from NFAs are not necessarily optimal
  - May contain many more states than is necessary

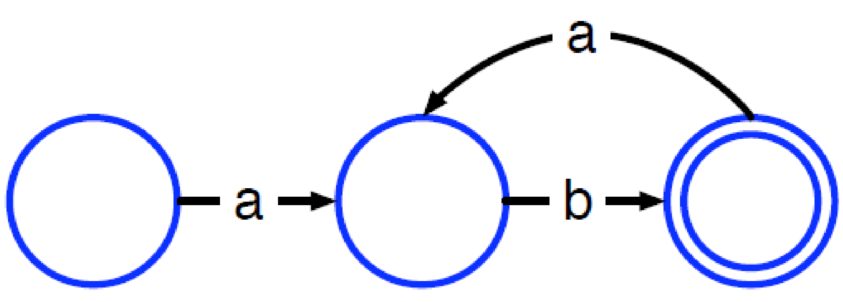
(ab)+ ≡ (ab)(ab)*



DFA reduction

- DFAs built from NFAs are not necessarily optimal
  - May contain many more states than is necessary

$$(ab)+ \equiv (ab)(ab)^*$$



DFA reduction

- Intuition: merge equivalent states
  - Two states are equivalent if they have the same transitions to the same states
- Basic idea of optimization algorithm
  - Start with two big nodes, one representing all the final states, the other representing all other states
  - Successively split those nodes whose transitions lead to nodes in the original DFA that are in different nodes in the optimized DFA

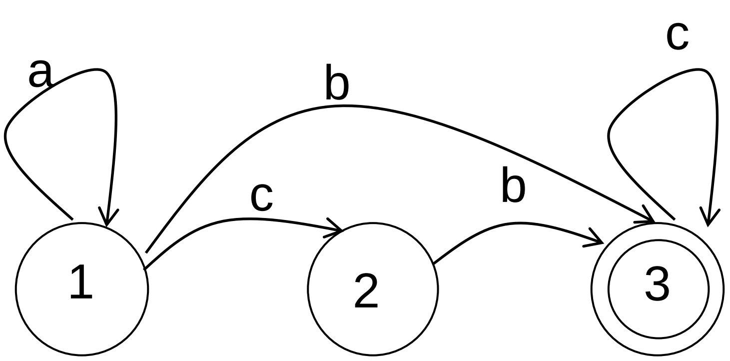
Implementation

- While doing lexical analysis, we need extensions to regular expressions
  - Match as long a substring as possible
  - Handle errors
- Good algorithms for substring matching
  - Require only a single pass over the input
    - Using `Tries`
  - Few operations per character
    - Table look-up method

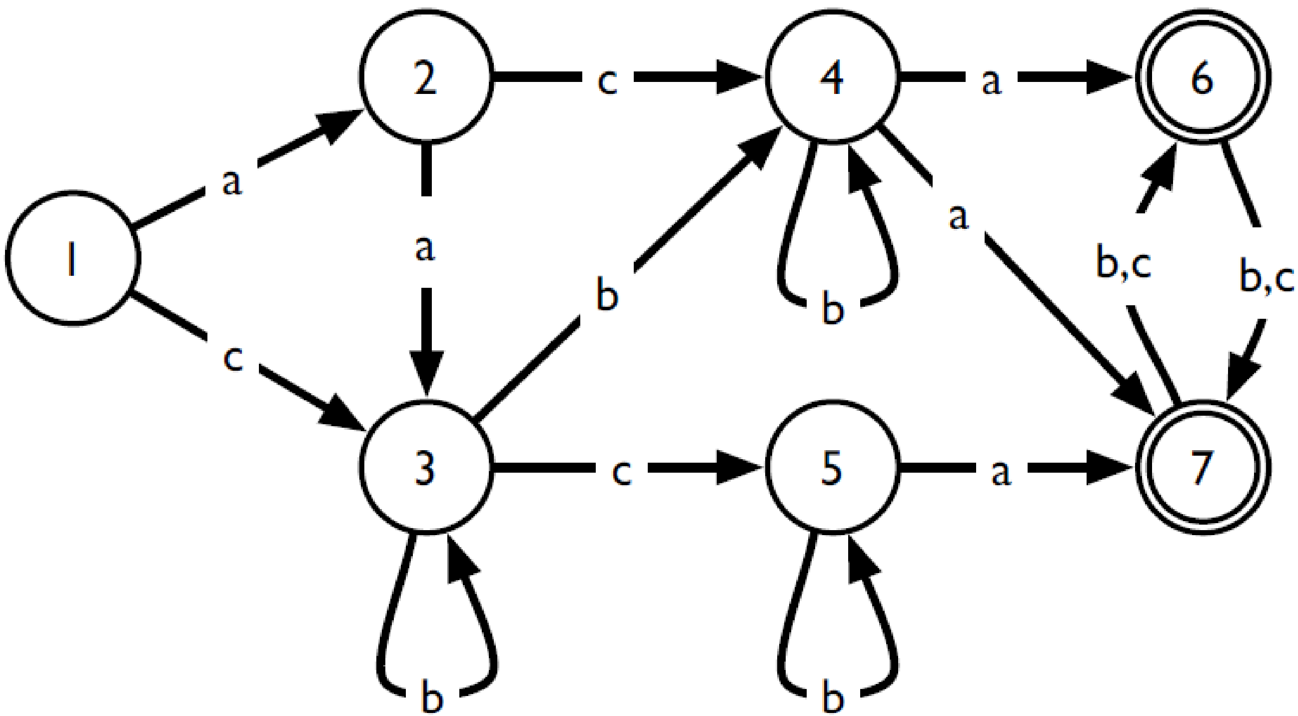
Implementation: Transition Tables

- A table encodes states and transitions of FA
  - 1 row per state
  - 1 column per character in the alphabet
  - Table entry: state (label)

State / Character	a	b	c
1	1	3	2
2	-	3	-
3	-	-	3

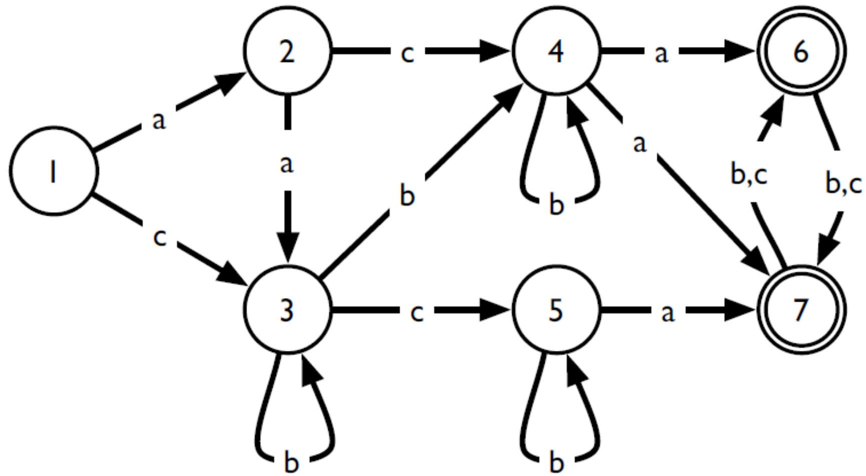


Example 1



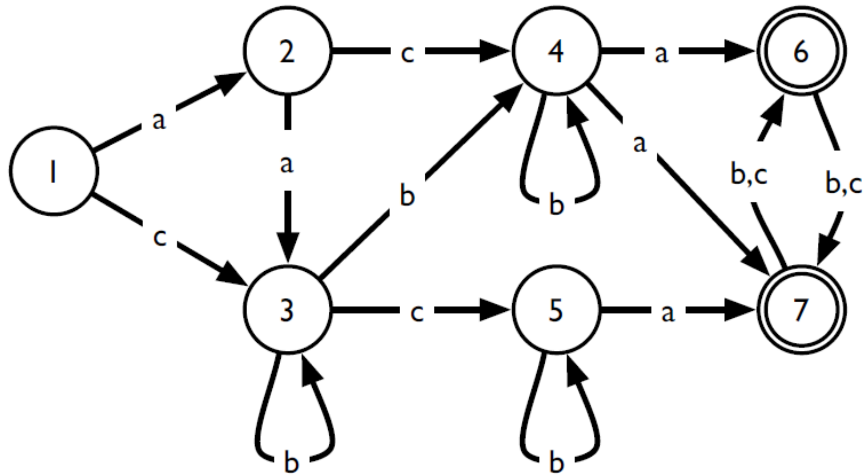
NFA OR DFA?

Example 1: NFA -> DFA



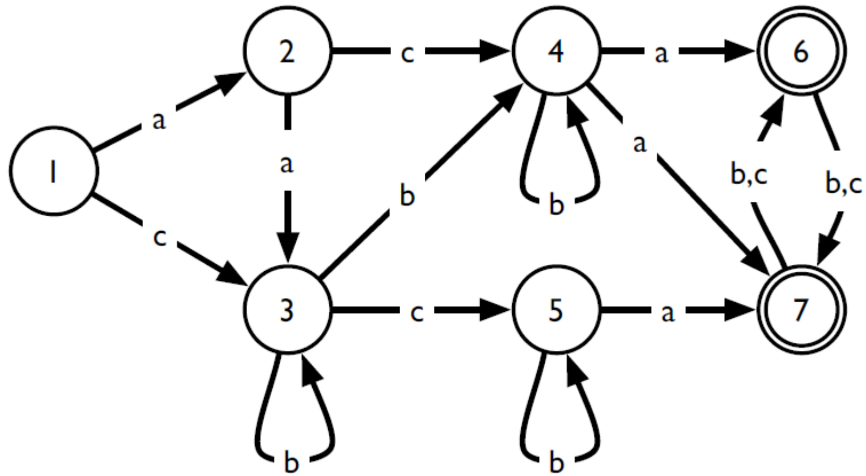
State / Char	a	b	c
1	2	-	3

Example 1: NFA -> DFA



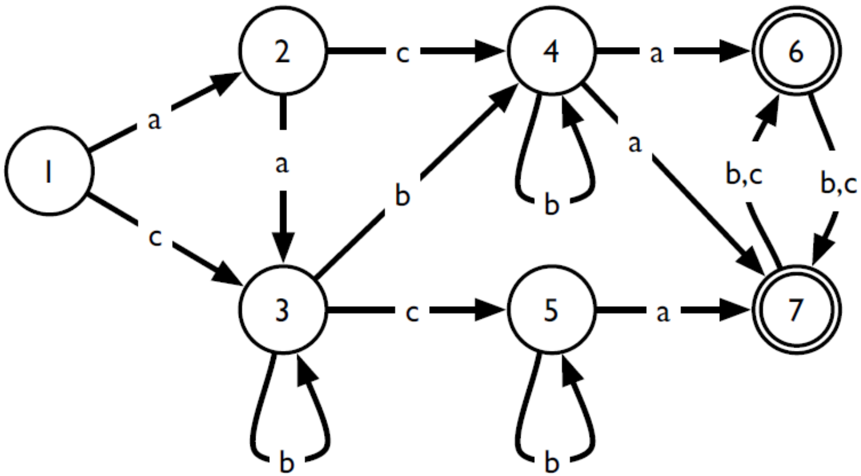
State / Char	a	b	c
1	2	-	3
2	3	-	4

Example 1: NFA -> DFA



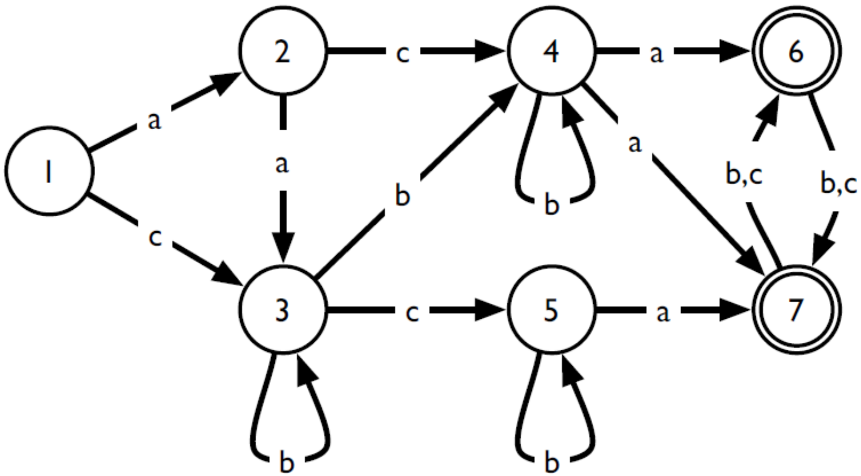
State / Char	a	b	c
1	2	-	3
2	3	-	4
3	-	3,4	5

Example 1: NFA -> DFA



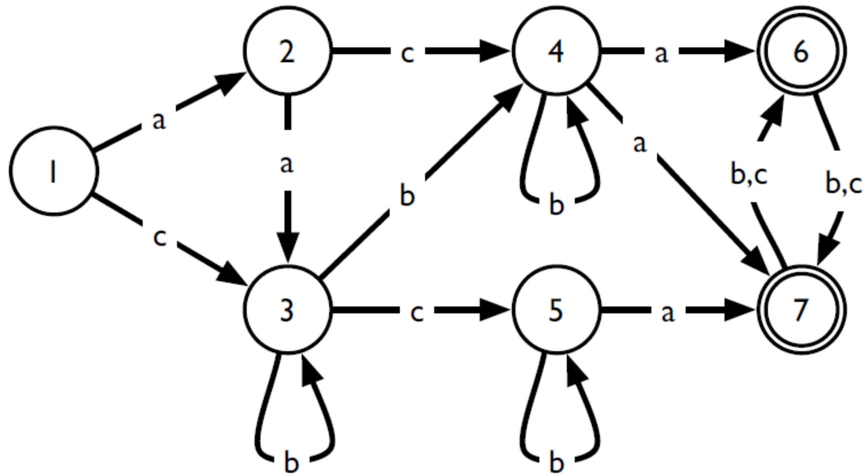
State / Char	a	b	c
1	2	-	3
2	3	-	4
3	-	3,4	5
4	6,7	4	-

Example 1: NFA -> DFA



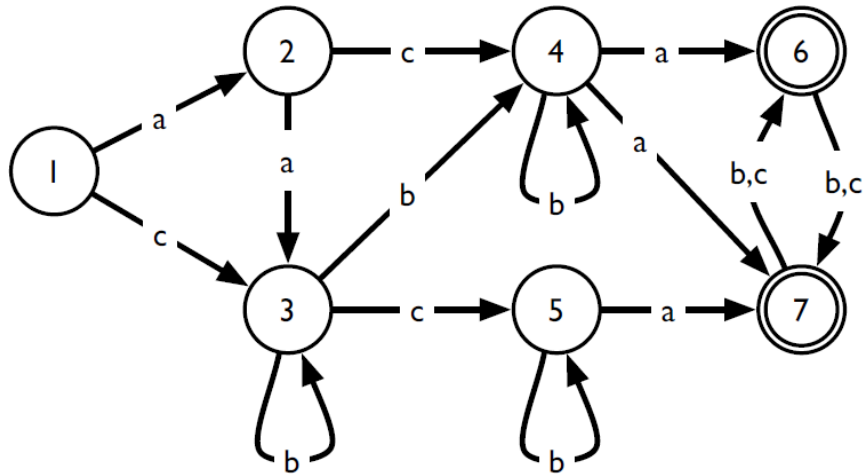
State / Char	a	b	c
1	2	-	3
2	3	-	4
3	-	3,4	5
4	6,7	4	-
3,4	6,7	3,4	5

Example 1: NFA -> DFA



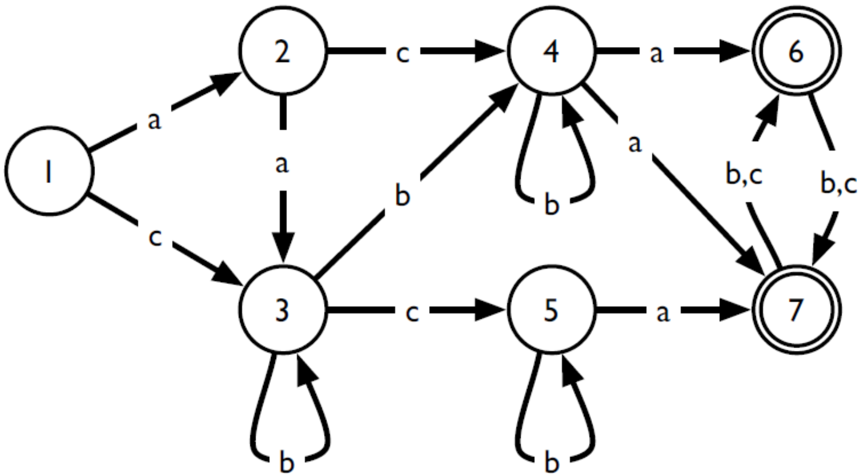
State / Char	a	b	c
1	2	-	3
2	3	-	4
3	-	3,4	5
4	6,7	4	-
3,4	6,7	3,4	5
5	7	5	-

Example 1: NFA -> DFA



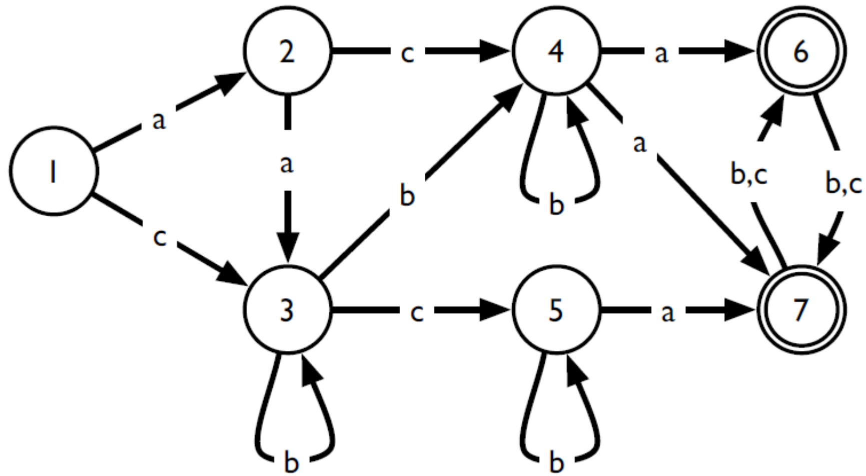
State / Char	a	b	c
1	2	-	3
2	3	-	4
3	-	3,4	5
4	6,7	4	-
3,4	6,7	3,4	5
5	7	5	-
6,7	-	6,7	6,7

Example 1: NFA -> DFA



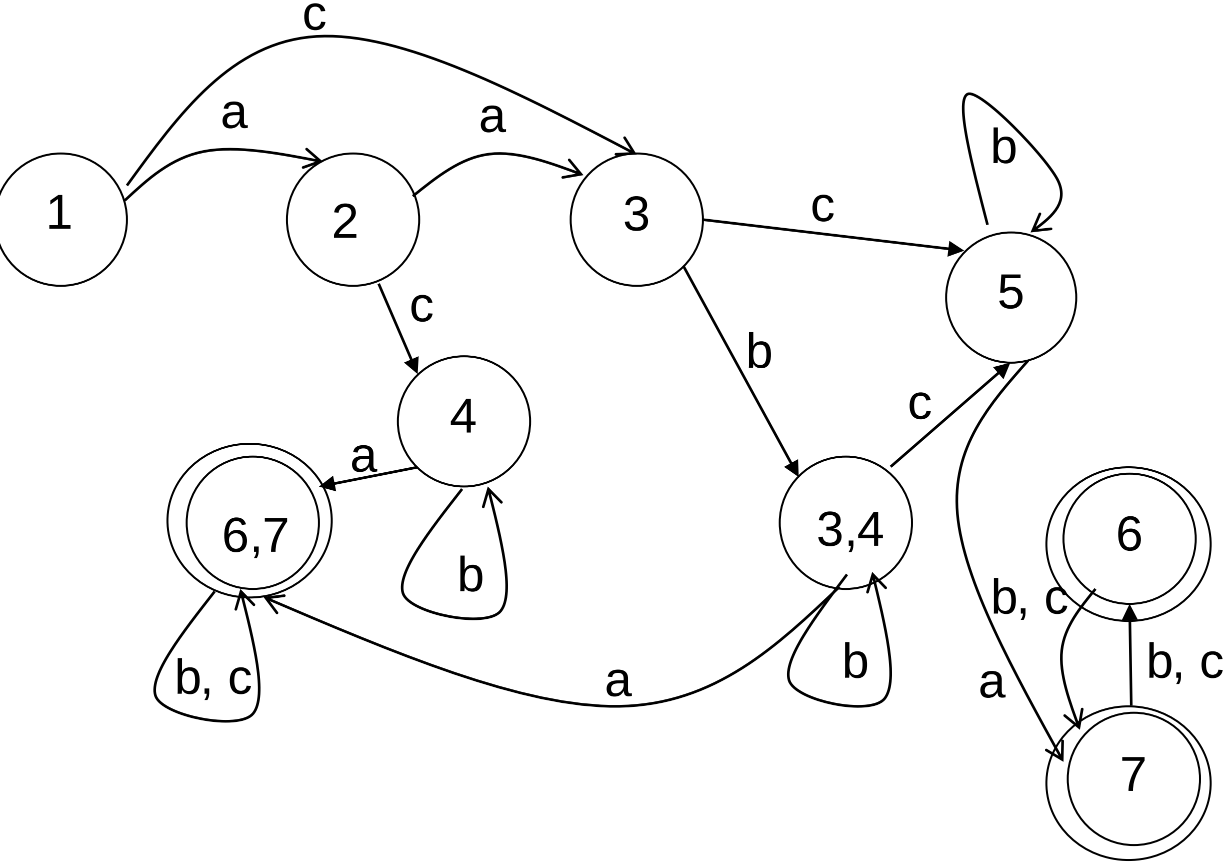
State / Char	a	b	c
1	2	-	3
2	3	-	4
3	-	3,4	5
4	6,7	4	-
3,4	6,7	3,4	5
5	7	5	-
6,7	-	6,7	6,7
7	-	6	6

Example 1: NFA -> DFA



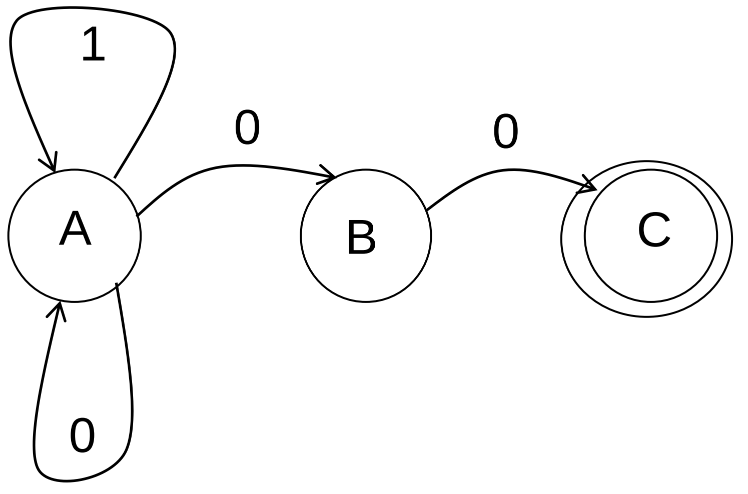
State / Char	a	b	c
1	2	-	3
2	3	-	4
3	-	3,4	5
4	6,7	4	-
3,4	6,7	3,4	5
5	7	5	-
6,7	-	6,7	6,7
7	-	6	6
6	-	7	7

Example 1: DFA



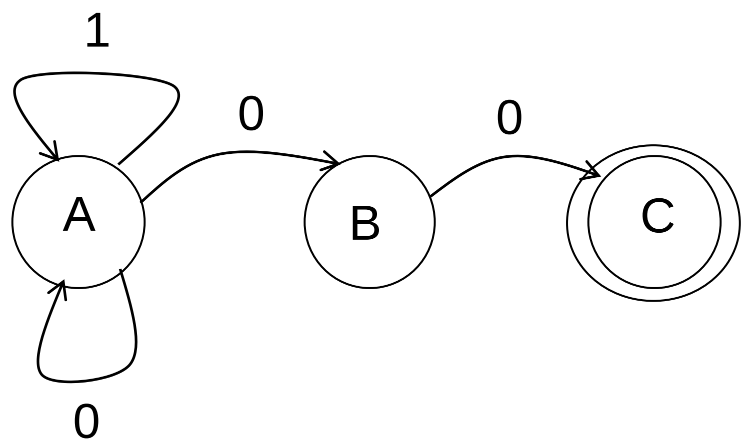
State	a	b	c
1	2	-	3
2	3	-	4
3	-	3,4	5
4	6,7	4	-
3,4	6,7	3,4	5
5	7	5	-
6,7	-	6,7	6,7
7	-	6	6
6	-	7	7

Example 2: NFA -> DFA



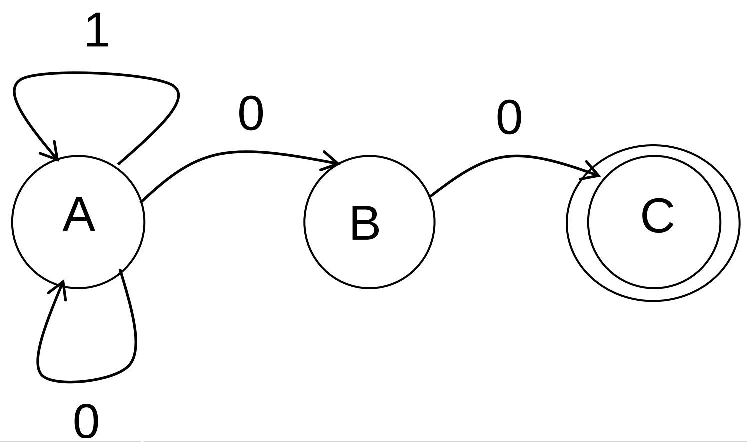
NFA OR DFA?

Example 2: NFA -> DFA



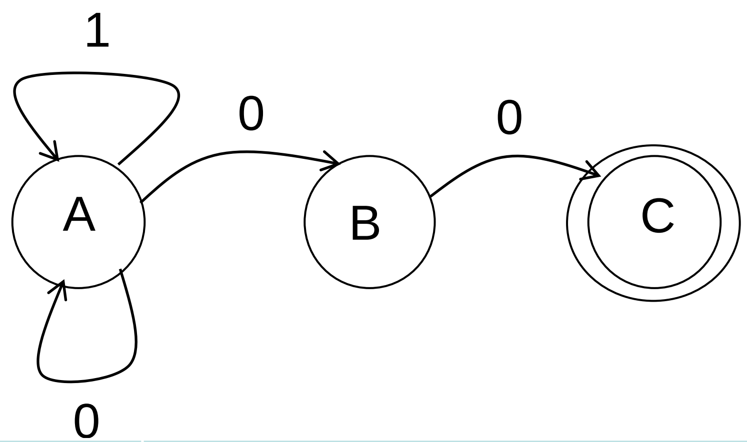
State/ char	0	1	Final ?	Comments
A	{A, B}	A	No	In state A, on seeing input 0, we have a choice to go to either state A or B

Example 2: NFA -> DFA



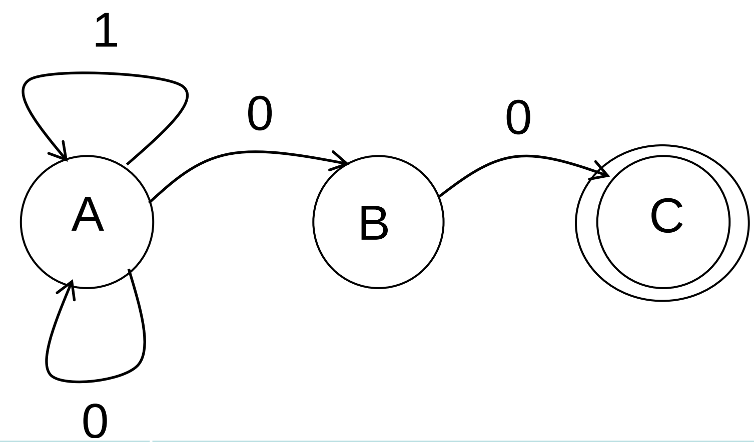
State/ char	0	1	Final ?	Comments
A	{A, B}	A	No	In state A, on seeing input 0, FA gives us a choice to go to either state A or state B
A,B	{A,B,C}	A	No	In state A,B we have two component states A and B. From A on input 0, FA takes us to states A and B. From B on 0 FA takes us to C. So, the set of states reachable from A,B on input 0 is A,B,C. Similarly, for input 1, from A FA takes us to A. From B on input 1, FA gets stuck in an error state.

Example 2: NFA -> DFA



State/ char	0	1	Final ?	Comments
A	{A, B}	A	No	In state A, on seeing input 0, FA gives us a choice to go to either state A or state B
A,B	{A,B,C}	A	No	In state A,B we have two component states A and B. From A on input 0, FA takes us to states A and B. From B on 0 FA takes us to C. So, the set of states reachable from A,B on input 0 is A,B,C. Similarly, for input 1, from A FA takes us to A. From B on input 1, FA gets stuck in an error state.
A,B,C	{A,B,C}	A	Yes	One of the component states C is final in the FA. Hence, A,B,C is a final state.

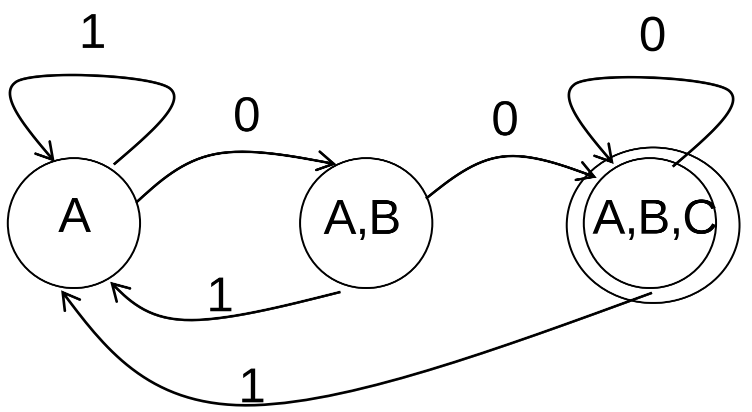
Example 2: NFA -> DFA



State/ char	0	1	Final ?	Comments
A	{A, B}	A	No	In state A, on seeing input 0, FA gives us a choice to go to either state A or state B
A,B	{A,B,C}	A	No	In state A,B we have two component states A and [illegible] tes A and B. From B on 0 FA takes us to C. So, the set of states reachable from A,B on input 0 is A,B,C. Similarly, for input 1, from A FA takes us to A. From B on input 1, FA gets stuck in an error state.
A,B,C	{A,B,C}	A	Yes	One of the component states C is final in the FA. Hence, A,B,C is a final state.

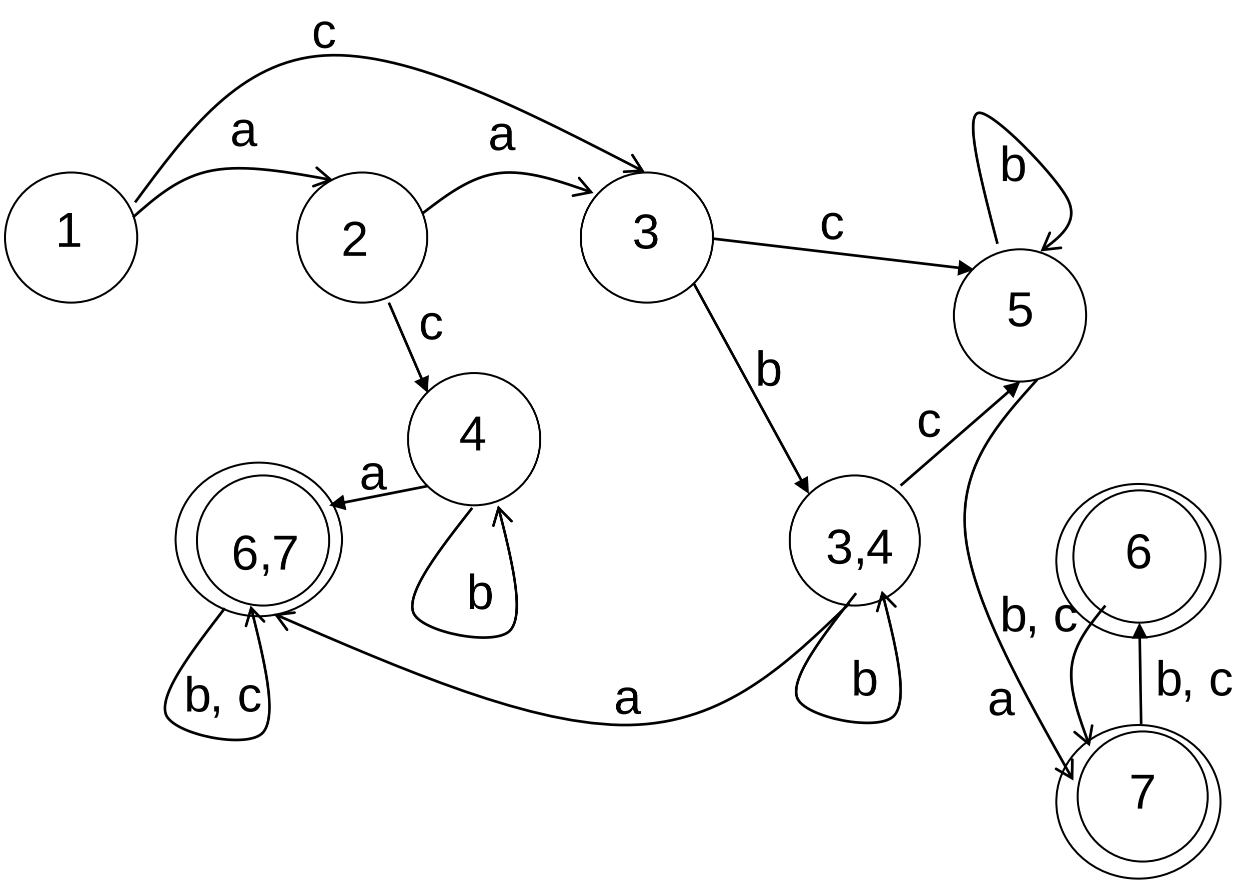
Should we consider states B and C in the table?

Example 2: DFA



State/ char	0	1	Final ?
A	{A, B}	A	No
A,B	{A,B,C}	A	No
A,B,C	{A,B,C}	A	Yes

Example 1: DFA



State	a	b	c
1	2	-	3
2	3	-	4
3	-	3,4	5
4	6,7	4	-
3,4	6,7	3,4	5
5	7	5	-
6,7	-	6,7	6,7
7	-	6	6
6	-	7	7

What states can be merged?

Example 1: Reduced DFA

What states can be merged?

State / Char	**a**	**b**	**c**
1	2	-	3
2	3	-	4
3	-	3,4	5
4	6,7	4	-
3,4	6,7	3,4	5
5	7	5	-
6,7	-	6,7	6,7
7	-	6	6
6	-	7	7

Example: Reduced DFA

What states can be merged?

Definition 8 (Equivalence of states) *Let $M = (Q, \Sigma, \delta, q_0, F)$ be a DFA. We say that two states $p, q \in Q$ are* **equivalent**, *and we write it $p \equiv q$, if for every string $x \in \Sigma^*$ the state that M reaches from p given x is accepting if and only if the state that M reaches from q given x is accepting.*

Definition 8 pic source: https://people.eecs.berkeley.edu/~luca/cs172/notemindfa.pdf

State / Char	**a**	**b**	**c**
1	2	-	3
2	3	-	4
3	-	3,4	5
4	6,7	4	-
3,4	6,7	3,4	5
5	7	5	-
6,7	-	6,7	6,7
7	-	6	6
6	-	7	7

Example: Reduced DFA

What states can be merged?

6 and 7

State / Char	a	b	c
1	2	-	3
2	3	-	4
3	-	3,4	5
4	6,7	4	-
3,4	6,7	3,4	5
5	**6_7_M**	5	-
6,7	-	6,7	6,7
6_7_M	-	6_7_M	6_7_M

Example: Reduced DFA

What states can be merged?

6,7 and 6_7_M

State / Char	**a**	**b**	**c**
1	2	-	3
2	3	-	4
3	-	3,4	5
4	**6,7_6_7_M**	4	-
3,4	**6,7_6_7_M**	3,4	5
5	**6,7_6_7_M**	5	-
6,7_6_7_M	-	6,7_6_7_M	6,7_6_7_M

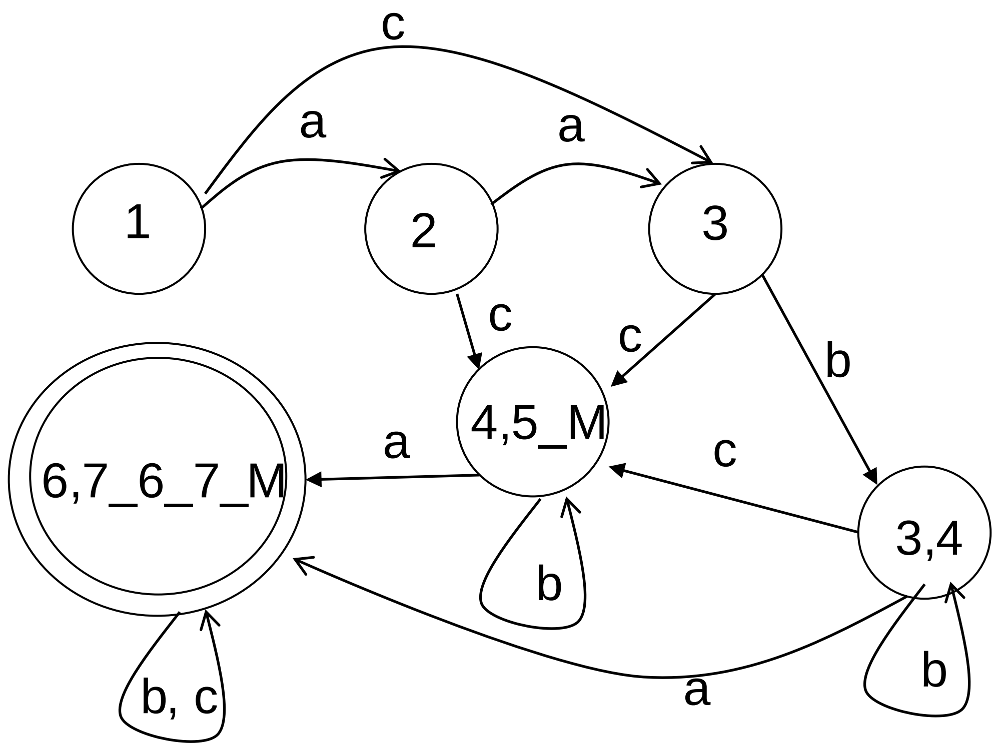
Example: Reduced DFA

What states can be merged?

4 and 5

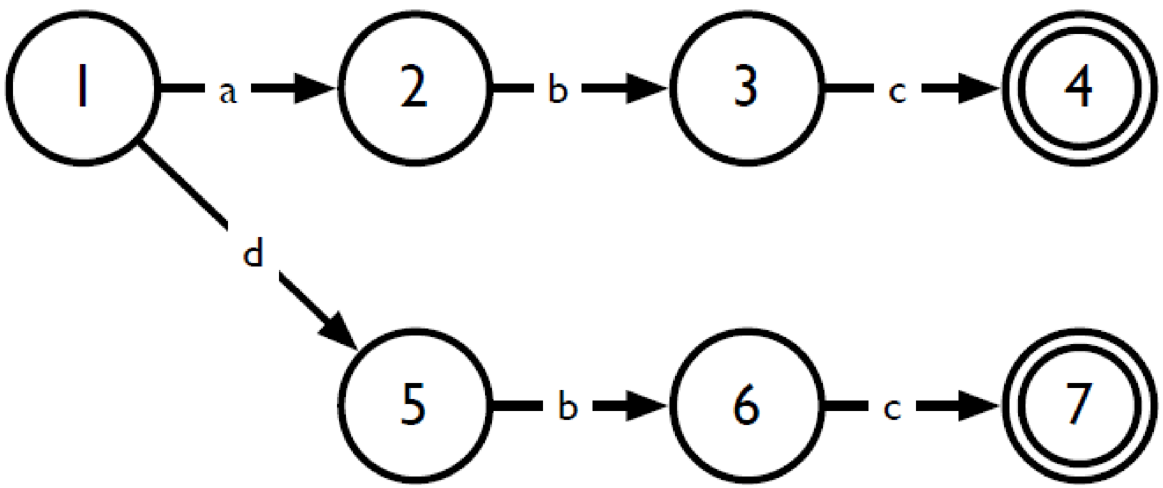
State / Char	**a**	**b**	**c**
1	2	-	3
2	3	-	**4_5_M**
3	-	3,4	**4_5_M**
4_5_M	6,7_6_7_M	**4_5_M**	-
3,4	6,7_6_7_M	3,4	**4_5_M**
6,7_6_7_M	-	6,7_6_7_M	6,7_6_7_M

Example: Reduced DFA



Exercise

- *Reduce the DFA*

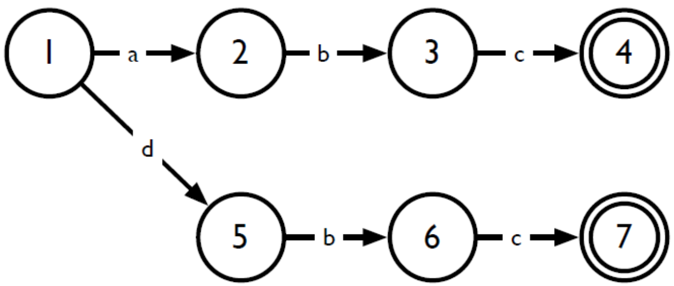


DFA Reduction (split-node)

- Algorithm
  - Start with all final states in one node and all non-final in another node. Call `Split()`

```
void Split(set_of_states* ss) {
  do {
      • Let S be any merged state corresponding to {s1, …, sn} and
        Let ‘c’ be any alphabet
      • Let t1, …, tn be the successor states to {s1, …, sn} under
        ‘c’
      • If (t1, …, tn do not all belong to the same merged state) {
        Split S into new states such that si and sj remain in the
        same merged state if and only if ti and tj are in the same
        merged state
    } while(more splits are possible)
}
```

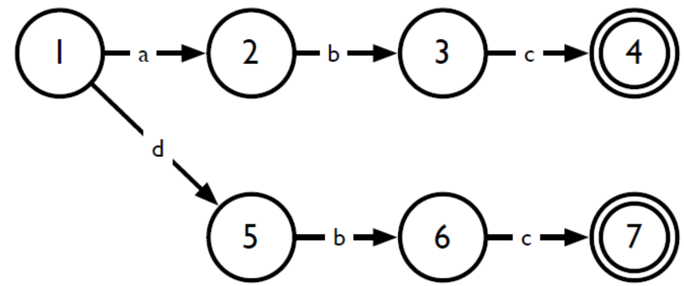
DFA Reduction (split-node)

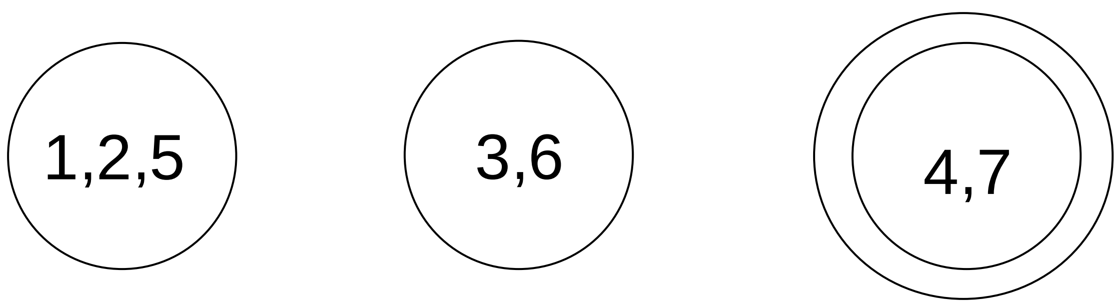


- Start with two big nodes
  - All final states in one and all non-final in another

DFA Reduction (split-node)

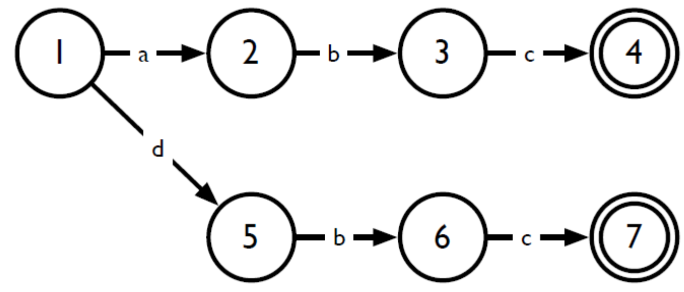
- Split 3,6 from 1,2, 3, 5, 6
  - 3,6 have common successor under ‘c’. 1,2,5 have no successor under ‘c’

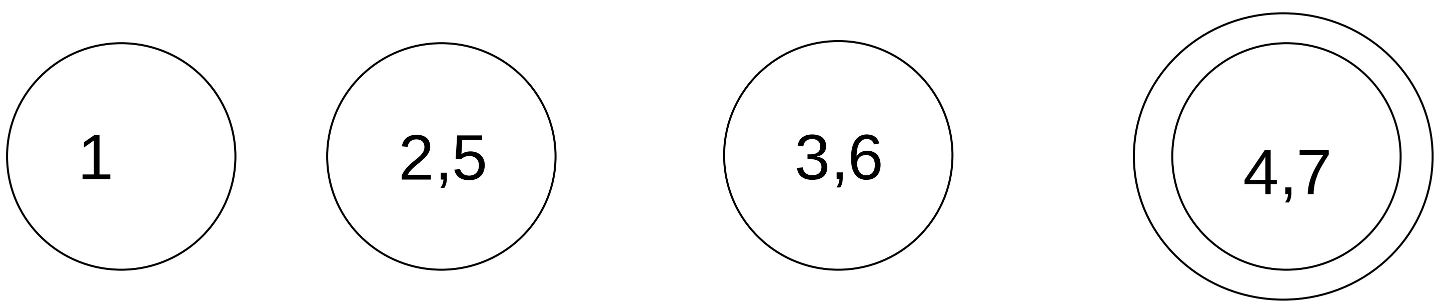




DFA Reduction (split-node)

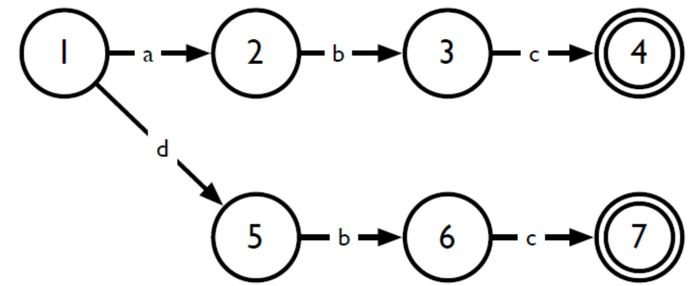
- Split 1 from 1,2, 5
  - 2 and 5 go to merged state 3,6 under 'b'. 1 does not.

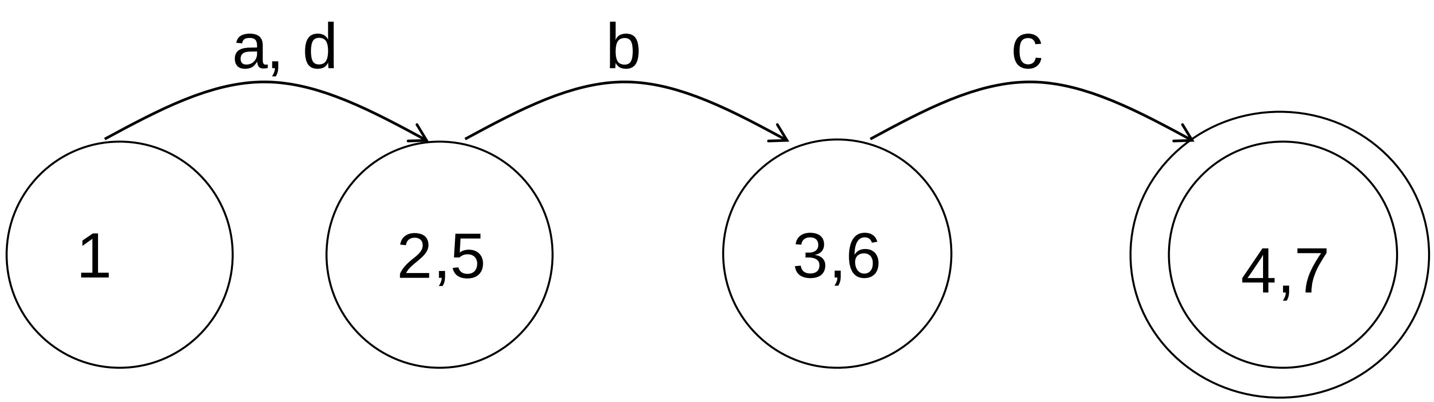




DFA Reduction (split-node)

- No more splits possible





DFA Program

- Using a transition table, it is straightforward to write a program to recognize strings in a regular language

```
state = initial_state; //start state of FA
while (true) {
   next_char = getc();
   if (next_char == EOF) break;
   next_state = T[state][next_char];
   if (next_state == ERROR) break;
   state = next_state;
}
if (is_final_state(state))
   //recognized a valid string
else
   handle_error(next_char);
```

Alternate implementation

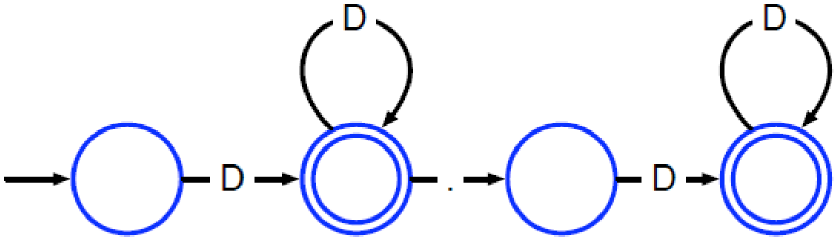
- Here's how we would implement the same program "conventionally"

```
next_char = getc();
while (next_char == 'a') {
    next_char = getc();
    if (next_char != 'b') handle_error(next_char);
    next_char = getc();
    if (next_char != 'c') handle_error(next_char);
    while (next_char == 'c') {
        next_char = getc();
        if (next_char == EOF) return; //matched token
        if (next_char == 'a') break;
        if (next_char != 'c') handle_error(next_char);
    }
}
handle_error(next_char);
```


Slide courtesy: Milind Kulkarni

Handling Lookahead

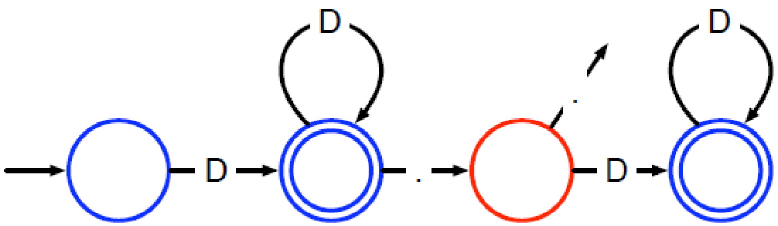
- E.g. distinguish between `int a` and `inta`
  - If the next char belongs to current token, continue
  - Else next char becomes part of next token
- Multi-character lookahead?
  - E.g. `DO I = 1, 100` (loop) vs. `DO I = 1.100` (variable assignment)
  - Solutions: Backup or insert special “action” state



Handling Lookahead

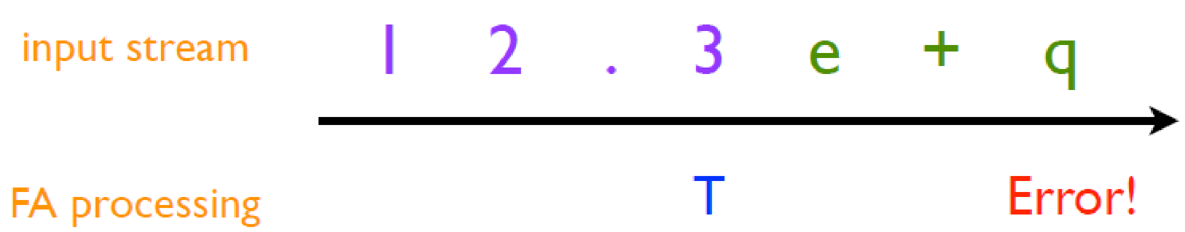
- E.g. distinguish between `int a` and `inta`
  - If the next char belongs to current token, continue
  - Else next char becomes part of next token
- Multi-character lookahead?
  - E.g. `DO I = 1, 100` (loop) vs. `DO I = 1.100` (variable assignment)
  - Solutions: Backup or insert special "action" state

123..44



General approach

- Remember states (T) that can be final states
- Buffer the characters from then on
- If stuck in a non-final state, back up to T, restore buffered characters to stream
- Example: 12.3e+q



Slide courtesy: Milind Kulkarni

Error Recovery

- What do we do if we encounter a lexical error (a character which causes us to take an undefined transition)?
- Two options
  - Delete all currently read characters, start scanning from current location
  - Delete *first* character read, start scanning from second character
    - This presents problems with ill-formatted strings (why?)
    - One solution: create a new regexp to accept runaway strings

Discussion

- Why separate class (token type) for each keyword?
  - Efficiency
    - Parsers take decisions based on token types. When decision making not possible, switch to token values, which are strings. String comparison is more expensive
  - Compatibility with parser generators
    - Some parser generators don’t support semantic predicates
  - Autocomplete / Intellisense

Discussion - Efficiency

```
switch(curToken.type) {
        case IF: parse_if_stmt();
                    break;
        ..
}
```

```
switch(curToken.type) {
        case KEYWORD: if(curToken.value==“if”);
                    parse_if_stmt();
        ..
}
```

Discussion - Compatibility

```
statement : IF condition body (ELSE body)? FI

statement : if condition body (else body)? fi
if: {current_token.value == "if"} KEYWORD ;
else: {current_token.value == "else"} KEYWORD ;
fi: …
KEYWORD: IF | ELSE | FI
```

Suggested Reading

- Alfred V. Aho, Monica S. Lam, Ravi Sethi and Jeffrey D.Ullman: Compilers: Principles, Techniques, and Tools, 2/E, AddisonWesley 2007
  - Chapter 3 (Sections: 3.1, 3,3, 3.6 to 3.9)
- Fisher and LeBlanc: Crafting a Compiler with C
  - Chapter 3 (Sections 3.1 to 3.4, 3.6, 3.7)