CS4410 : Spring 2013

Lexing & Parsing

Notes

- PS0 due Wednesday, 11:59pm
- Reading:
  - Relevant chapters on Lexing & Parsing in Appel
  - OCamlLex & OCamlYacc documentation + tutorial

Parsing

- Two pieces conceptually:
  - Recognizing syntactically valid phrases.
  - Extracting semantic content from the syntax.
    - E.g., What is the subject of the sentence?
    - E.g., What is the verb phrase?
    - E.g., Is the syntax ambiguous? If so, which meaning do we take?
      - "Fruit flies like a banana"
      - "2 * 3 + 4"
      - "x ^ f y"
- In practice, solve both problems at the same time.

Specifying Syntax

We use grammars to specify the syntax of a language.

exp → int | var | exp '+' exp | exp '*' exp | 'let' var '=' exp 'in' exp 'end'

int → '-'?digit+

var → alpha(alpha|digit)*

digit → '0' | '1' | '2' | '3' | '4' | … | '9'

alpha → [a-zA-Z]

Naïve Matching

To see if a sentence is legal, start with the first non-terminal), and keep expanding non-terminals until you can match against the sentence.

N → ‘a’ | ‘(‘ N ‘)’ “((a))”

N → ‘(’ N ‘)’

→ ‘(’ ‘(’ N ‘)’ ‘)’

→ ‘(’ ‘(’ ‘a’ ‘)’ ‘)’ = “((a))”

Alternatively

Start with the sentence, and replace phrases with corresponding non-terminals, repeating until you derive the start non-terminal.

N → ‘a’ | ‘(’ N ‘)’ “((a))”

‘(’ ‘(’ ‘a’ ‘)’ ‘)’ →

‘(’ ‘(’ N ‘)’ ‘)’ →

‘(’ N ‘)’ → N

Highly Non-Deterministic

- For real grammars, automating this non-deterministic search is non-trivial.
  - As we'll see, naïve implementations must do a lot of back-tracking in the search.
- Ideally, given a grammar, we would like an efficient, deterministic algorithm to see if a string matches it.
  - There is a very general cubic time algorithm.
  - Only linguists use it ☺.
  - (In part, we don't because recognition is only half the problem.)
- Certain classes of grammars have much more efficient implementations.
  - Essentially linear time with constant state (DFAs).
  - Or linear time with stack-like state (Pushdown Automata).

Tools in your Toolbox

- Manual parsing (say, recursive descent).
  - Tedious, error prone, hard to maintain.
  - But fast & good error messages.
- Parsing combinators
  - Encode grammars as (higher-order) functions.
    - Basically, functions that generate recursive-descent parsers.
    - Makes it easy to write & maintain grammars.
  - But can do a lot of back-tracking, and requires a limited form of grammar (e.g., no left-recursion.)
- Lex and Yacc
  - Domain-Specific-Languages that generate very efficient, table-driven parsers for general classes of grammars.
  - Learn about the theory in CS3800
  - Need to know a bit here to understand how to effectively use these tools.

CS4410 : Spring 2013

Regular Expressions &Finite-State Automata

Regular Expressions

- Non-recursive grammars
  - ε (epsilon – matches empty string)
  - Literals (‘a’, ‘b’, ‘2’, ‘+’, etc.) drawn from alphabet
  - Concatenation (R_1 R_2)
  - Alternation (R_1 | R_2)
  - Kleene star (R*)
- Non-terminals are expanded away

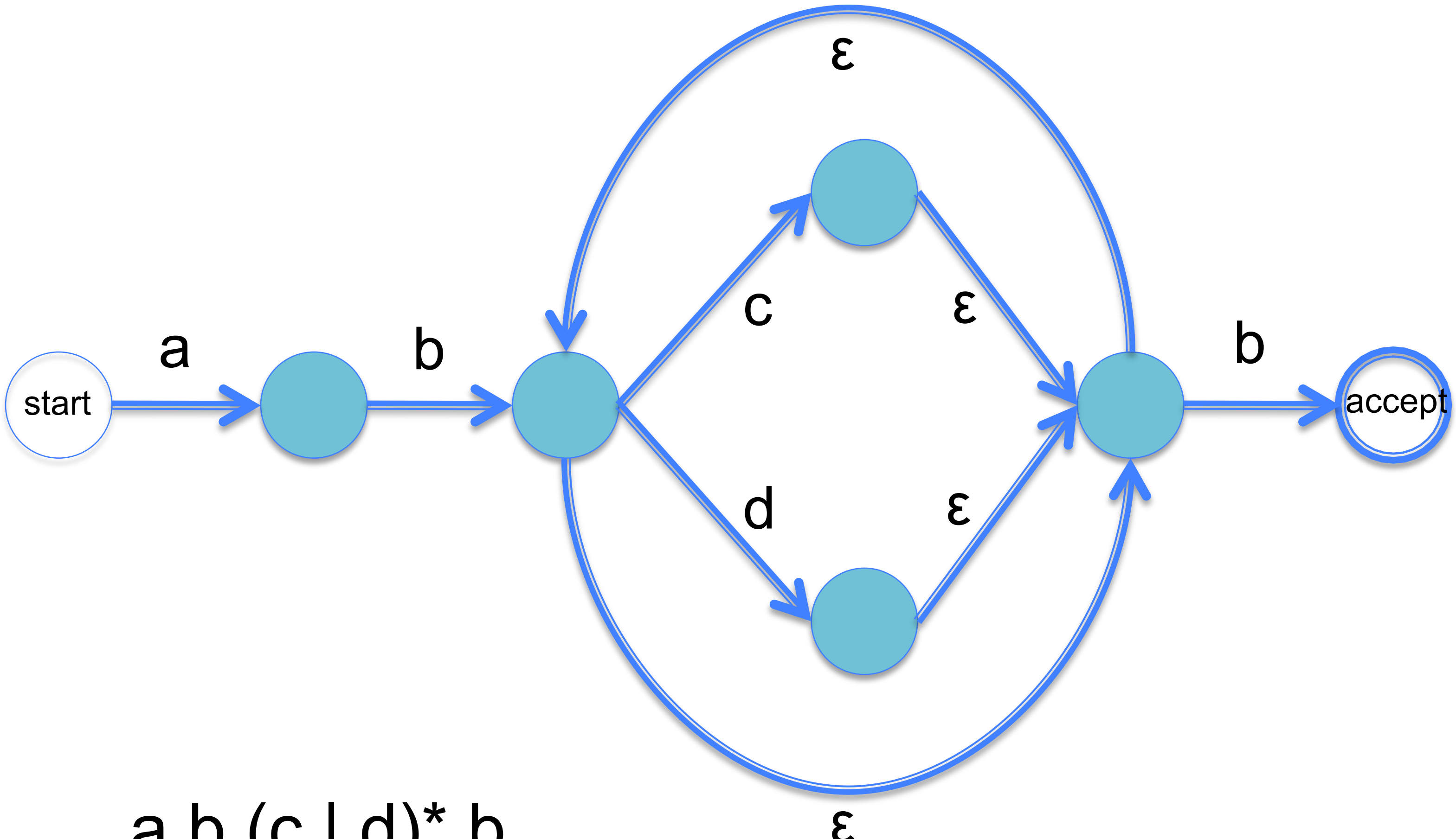
S → a b X* b

X → (c | d)

S → a b (c | d)* b

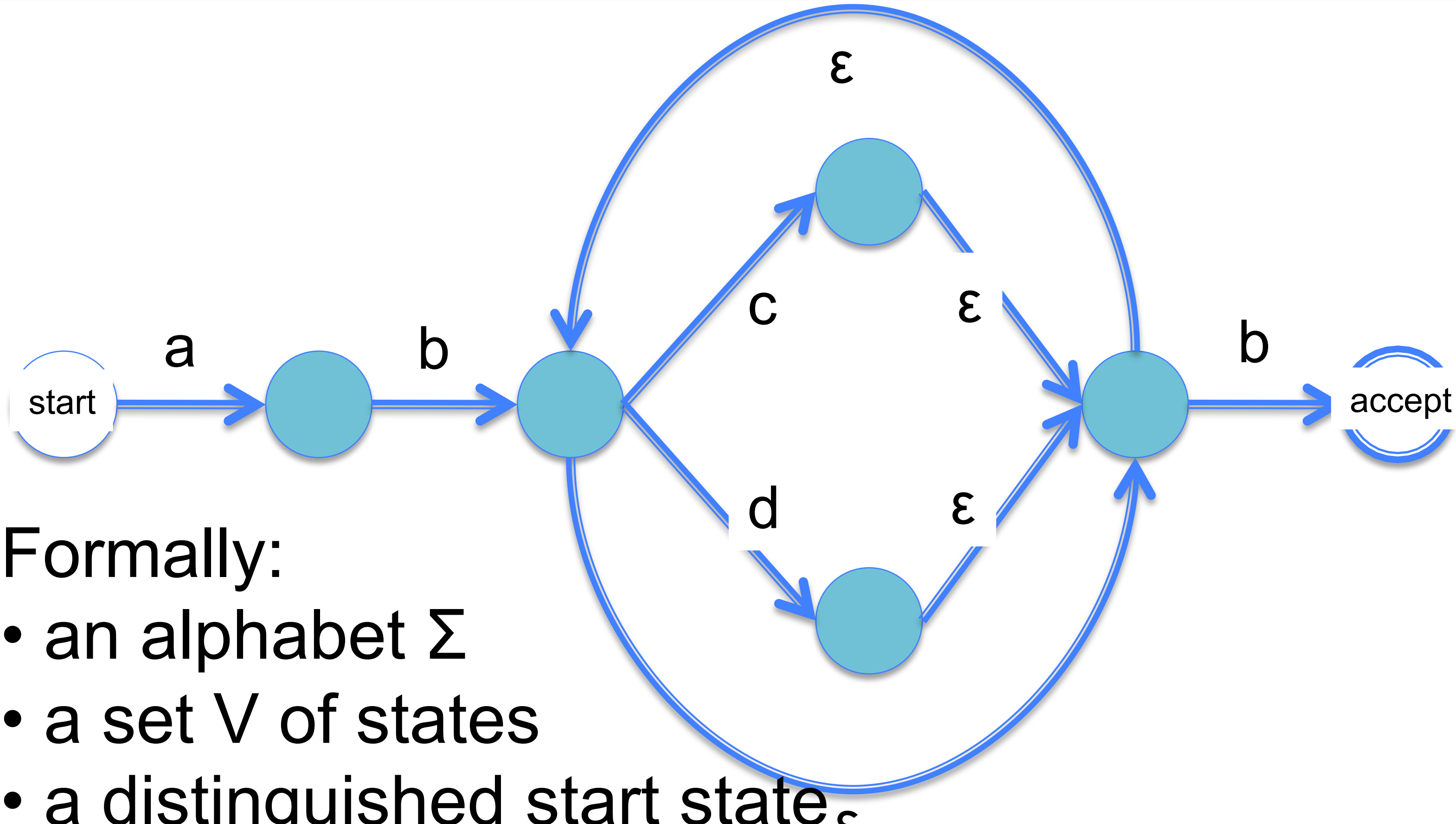
- As are other abbreviations (e.g., R+ = RR*)

Graphical Representation



a b (c | d)* b

Non-Deterministic, Finite-State Automaton

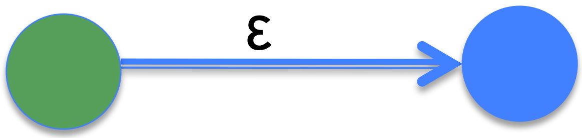


Formally:

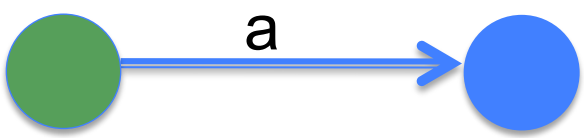
- an alphabet Σ
- a set V of states
- a distinguished start state
- one or more accepting states
- transition relation: $\delta : V * (\Sigma + \varepsilon) * V \rightarrow$ bool

Translating RegExps

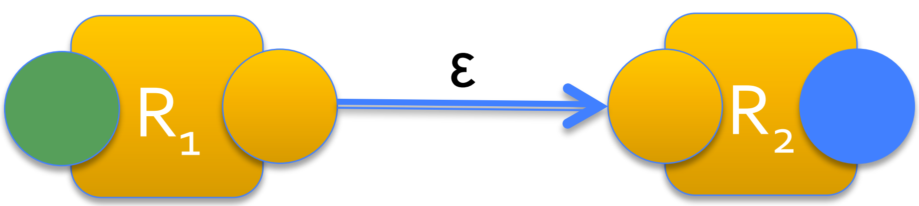
- Epsilon:



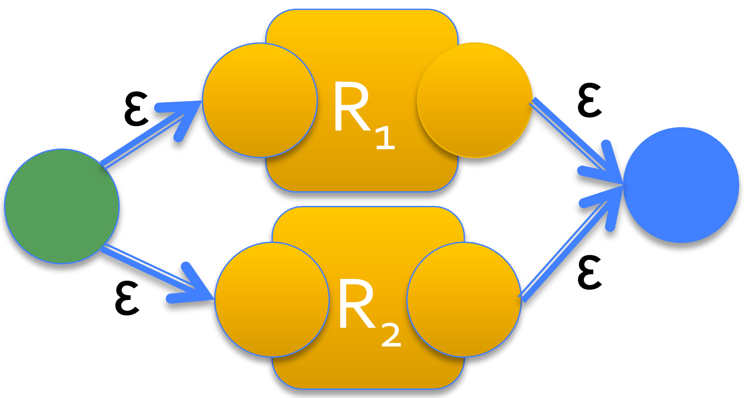
- Literal ‘a’:



- $R_1 R_2$

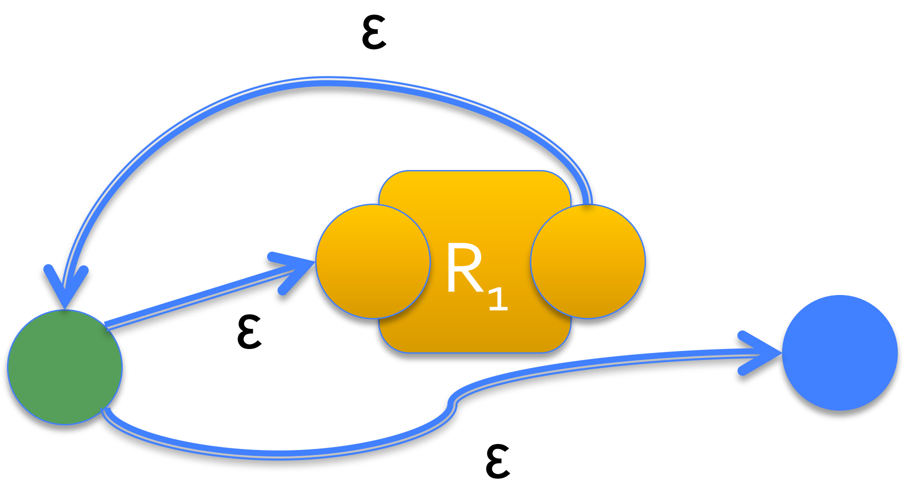


- $R_1 | R_2$



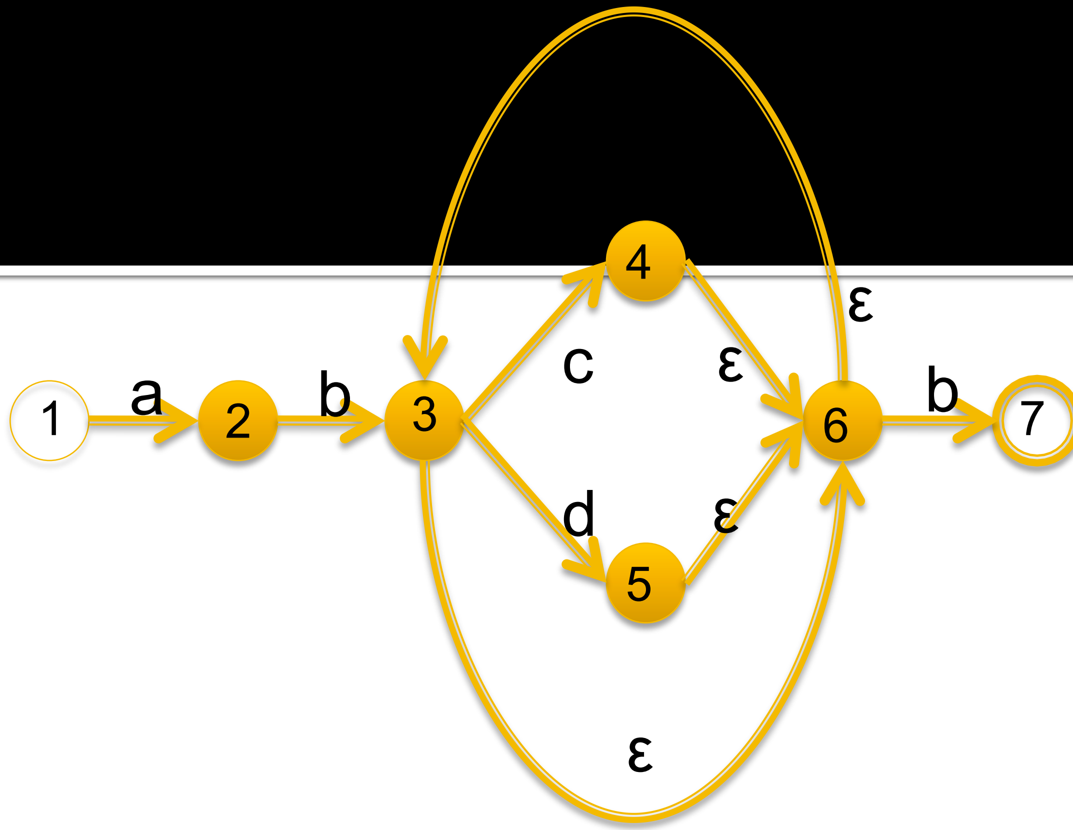
Translating RegExps

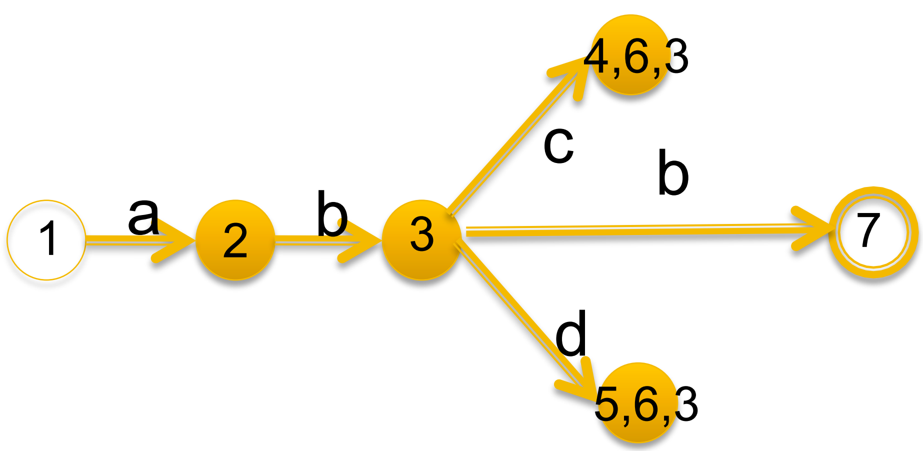
- R*

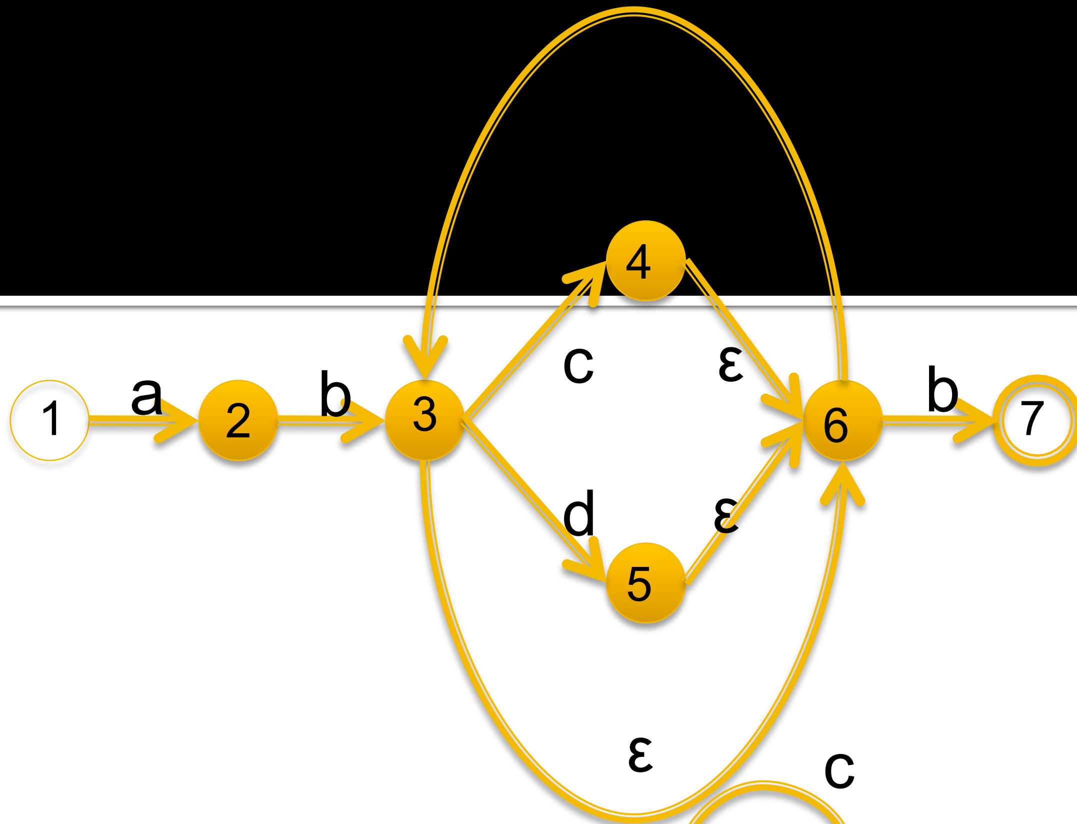


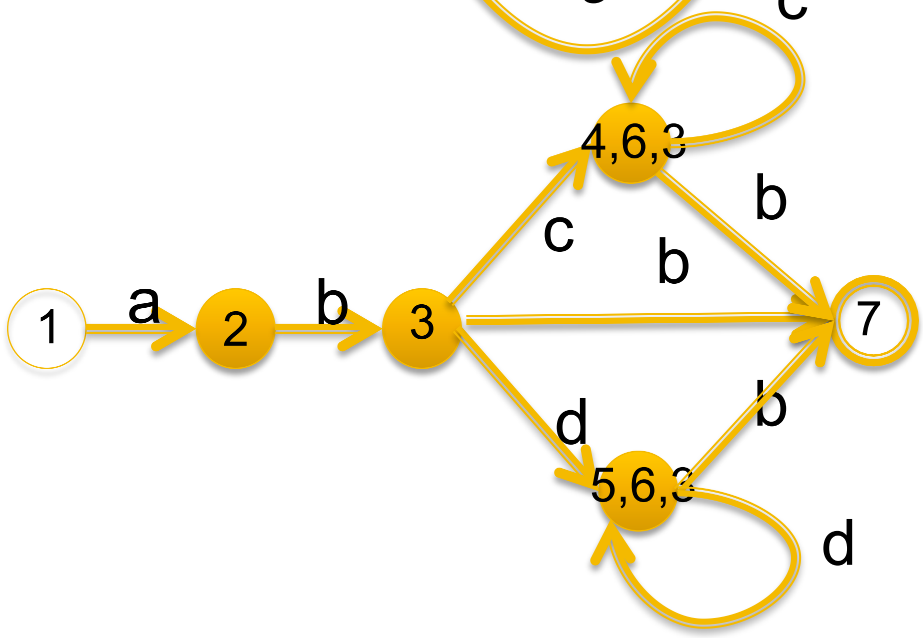
Converting to Deterministic

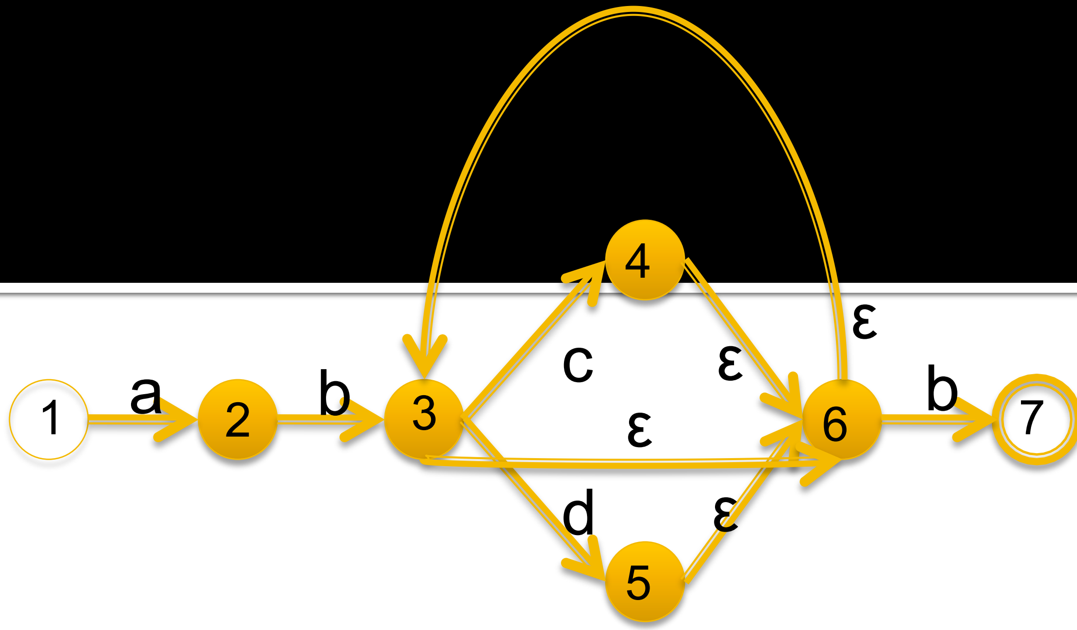
- Naively:
  - Give each state a unique ID (1,2,3,…)
  - Create super states
    - One super-state for each subset of all possible states in the original NDFA.
    - (e.g., {1},{2},{3},{1,2},{1,3},{2,3},{123})
  - For each super-state (say {23}):
    - For each original state s and character c:
      - Find the set of accessible states (say {1,2}) skipping over epsilons.
      - Add an edge labeled by c from the super state to the corresponding super-state.
- In practice, super-states are created lazily.


1
a
2
b
3
4
c
ε
ε
6
b
7
d
ε
5
ε

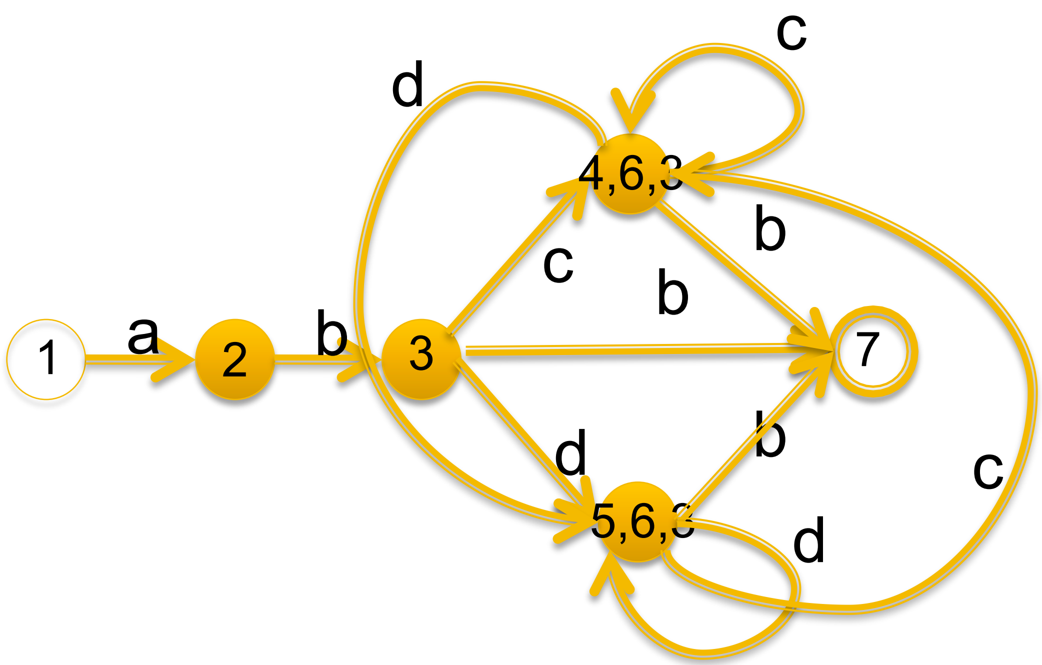

4,6,3
c
b
1
a
2
b
3
7
d
5,6,3


1
a
2
b
3
c
4
ε
6
b
7
d
5
ε
ε


c
4,6,3
b
c
b
1
a
2
b
3
7
d
b
5,6,3
d



1
a
2
b
3
c
4
ε
ε
ε
6
b
7
d
5
ε



1
a
2
b
3
d
c
4,6,3
c
b
b
7
d
b
5,6,3
d
c

Once We have a DFA

- Deterministic Finite State automata are easy to simulate:
  - For each state and character, there is at most one transition we can take.
- Usually record the transition function as an array, indexed by states and characters.
- Lexer starts off with a variable s initialized to the start state:
  - Reads a character, uses transition table to find next state.
  - Look at the output of Lex!