HARVARD
John A. Paulson
School of Engineering
and Applied Sciences

CS153: Compilers
Lecture 3: Lexical Analysis

Stephen Chong

https://www.seas.harvard.edu/courses/cs153

Announcements

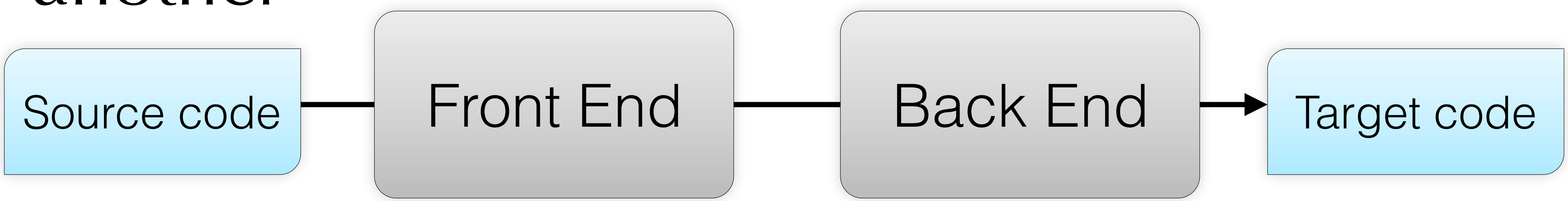
- Project 1 out
  - Due Thursday Sept 20
- Project 2 will be released Thursday Sept 13
  - Due in three weeks
- Anyone looking for a partner?
  - Post on Piazza, or let Prof Chong know
- Office hours start this week!
  - See webpage for details

Today

- Lexical analysis!
- Regular expressions
- (Nondeterministic) finite state automata (NFA)
- Converting NFAs to deterministic finite state automata (DFAs)
- Hand written lexer
- MLLex

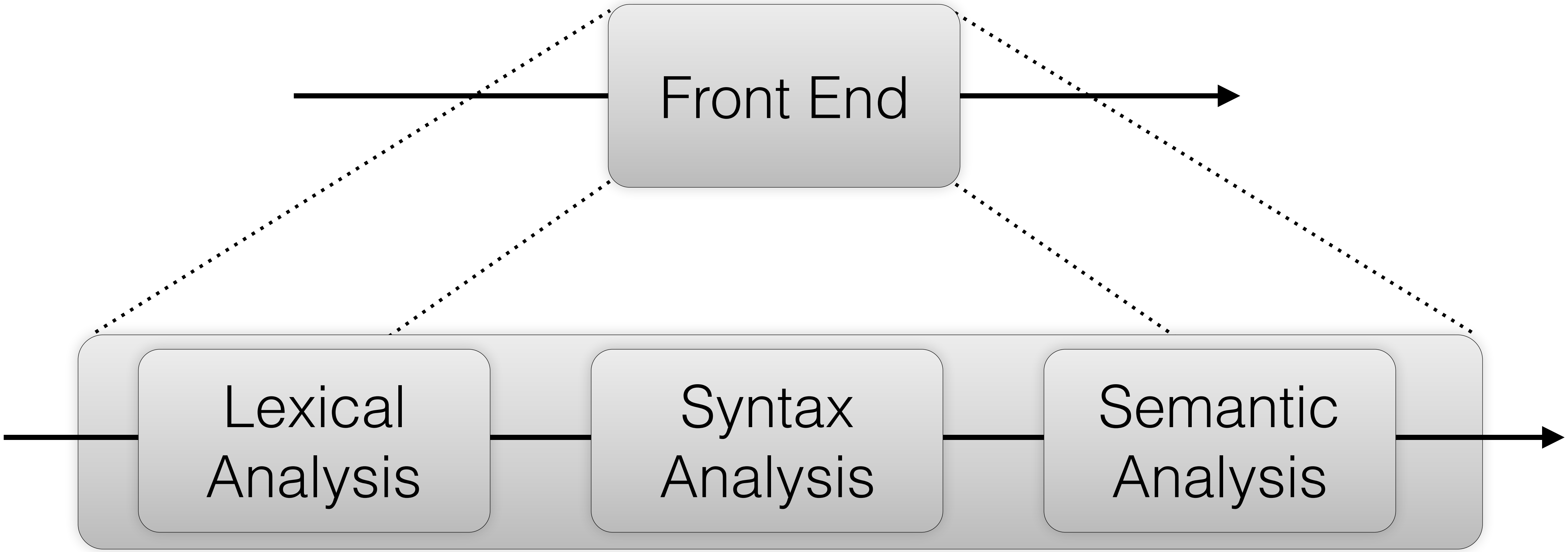
Lexing and Parsing

- Compiler translates from one language to another



- Front end: Analysis
  - pulls apart program, understand structure and meaning
- Back end: Synthesis
  - puts it back together in a different way

Lexing and Parsing



- **Lexical analysis**: breaks input into individual words, aka "tokens"
- **Syntax analysis**: parses the phrase structure of program
- **Semantic analysis**: calculates meaning of program

Lexical Tokens

- A **lexical token** is a sequence of characters that can be treated as a unit for parsing
- A language classifies lexical tokens into **token types**

Type	Examples
ID	`foo n14 last`
NUM	`73 0 00 515 082`
REAL	`66.1 .5 10. 1e67`
IF	`if`
COMMA	`,`
NOTEQ	`!=`
LPAREN	`(`

- Tokens constructed from alphabetic chars are called **reserved words**, typically can't be used as identifiers
  - E.g., IF, VOID, RETURN

Lexical Tokens

- Examples of nontokens

comment	`/* here's a comment */`
preprocessor directive	`#include <stdio.h>`
preprocessor directive	`#define NUMS 5 , 6`
macro	`NUMS`
blanks, tabs, newlines	

Example

- Given a program

```
float match0(char *s) /* find a zero */
{if (!strncmp(s,"0.0", 3))
  return 0.;
}
```

the lexer returns the sequence of tokens

FLOAT ID(`match0`) LPAREN CHAR STAR ID(`s`) RPAREN
LBRACE IF LPAREN BANG ID(`strncmp`) LPAREN ID(`s`)
COMMA STRING(`0.0`) COMMA NUM(`3`) RPAREN
RPAREN RETURN REAL(`0.0`) SEMI RBRACE EOF

How to Describe and Implement Lexing?

- Could describe in natural language, and implement in an ad hoc way
- But we will specify lexical tokens using **regular expressions**, and implement lexers using **deterministic finite automata**
  - Elegant mathematics connects the two

Regular Expressions

- Each regular expression stands for/matches a set of strings
- Grammar
  - $\varnothing$ (matches no string)
  - **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*)

Set of Strings

- $[\![\varnothing]\!] = \{\ \}$
- $[\![\varepsilon]\!] = \{\ \text{“”}\ \}$
- $[\![\text{‘a’}]\!] = \{\ \text{“a”}\ \}$
- $[\![R_1\ R_2]\!] = \{\ s \mid s = \alpha\text{^}\beta \text{ and } \alpha \in [\![R_1]\!] \text{ and } \beta \in [\![R_2]\!]\ \}$
- $[\![R_1 \mid R_2]\!] = \{\ s \mid s \in [\![R_1]\!] \text{ or } s \in [\![R_2]\!]\ \}$
  $= [\![R_1]\!] \cup [\![R_2]\!]$
- $[\![R^*]\!] = [\![\ \varepsilon \mid RR^*\ ]\!]$
  $= \{\ s \mid s = \text{“”} \text{ or } s = \alpha\text{^}\beta \text{ and } \alpha \in [\![R]\!] \text{ and } \beta \in [\![R^*]\!]\ \}$

Examples

- `( 0 | 1 )* 0`
  - Binary numbers that are multiples of 2
- `b*(abb*)*(a|ε)`
  - Strings of `a`'s and `b`'s without consecutive `a`'s
- `(a|b)*aa(a|b)*`
  - Strings of `a`'s and `b`'s with consecutive `a`'s

Extensions

- We might recognize numbers as:
  - digit ::= [0-9]
  - number ::= '–'? digit+
- Here, [0-9] shorthand for 0 | 1 | … | 9
- '–'? shorthand for ('–' | ε) (i.e., the minus – is optional)
- digit+ shorthand for (digit digit*) (i.e., at least one digit)
- So number ::=

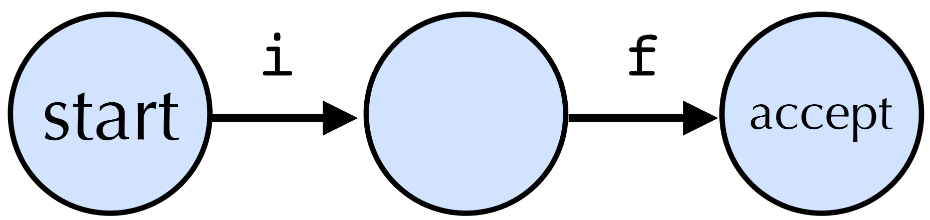
  ('–' | ε) ((0 | 1 | … | 9)(0 | 1 | … | 9)*)

Example

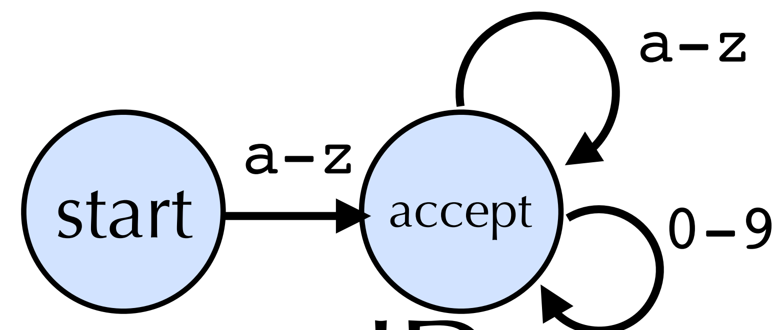
Reg Exp	Token
`if`	IF
`[a-z][a-z0-9]*`	ID
`[0-9]+`	NUM
`([0-9]+ "." [0-9]*) | ([0-9]* "." [0-9]+)`	REAL

- In general, we want the longest match:
  - longest initial substring of the input that can match a regular expression is taken as next token
- E.g., given input **`iffy`**, we want the token ID(**`iffy`**) rather than IF

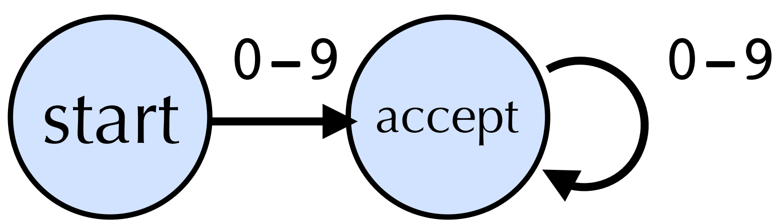
Graphical representation



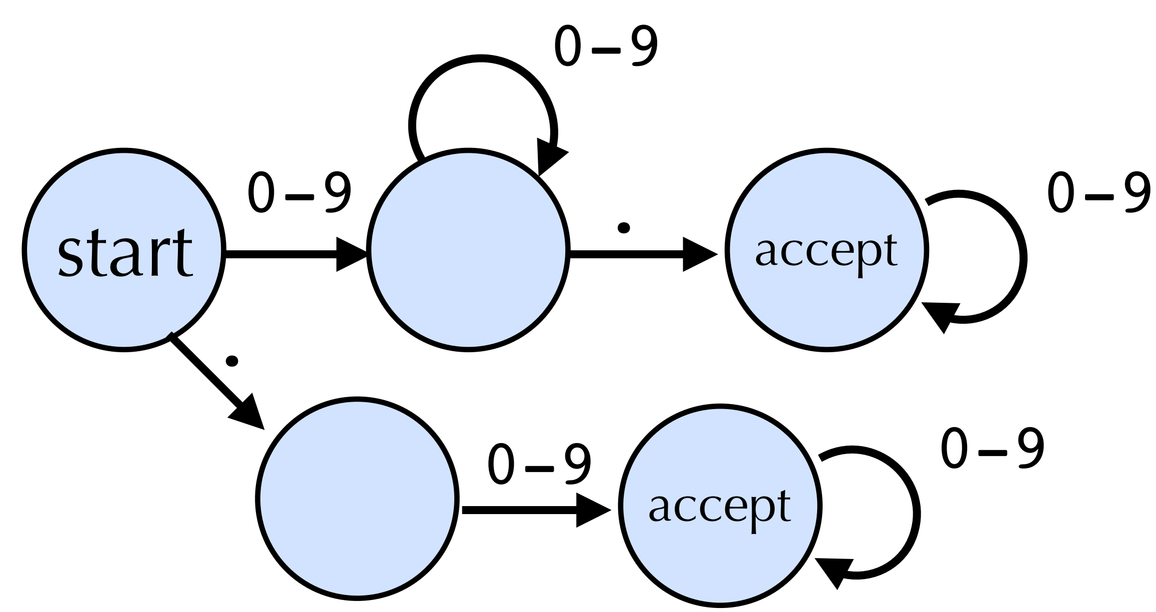
IF



ID

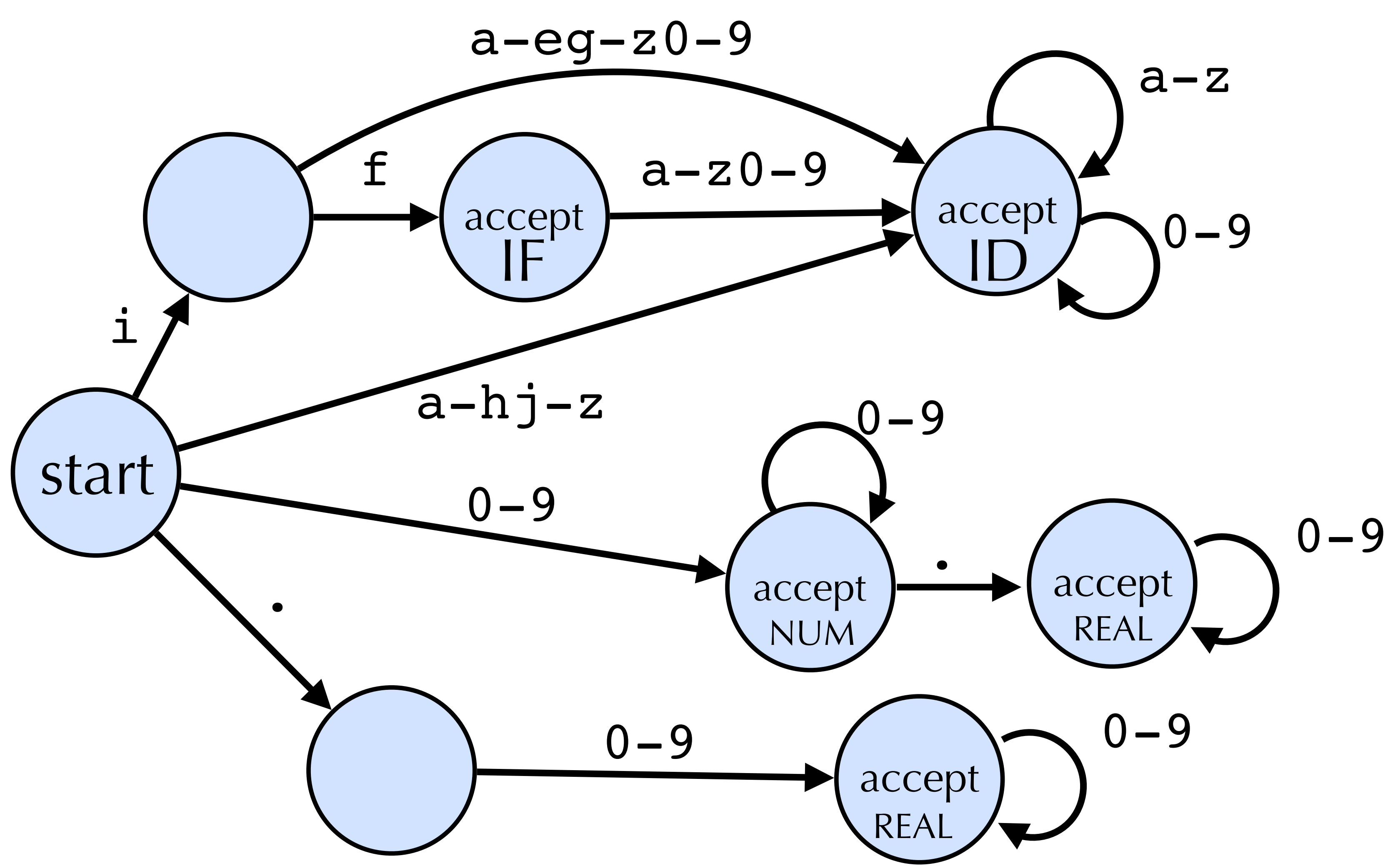


NUM



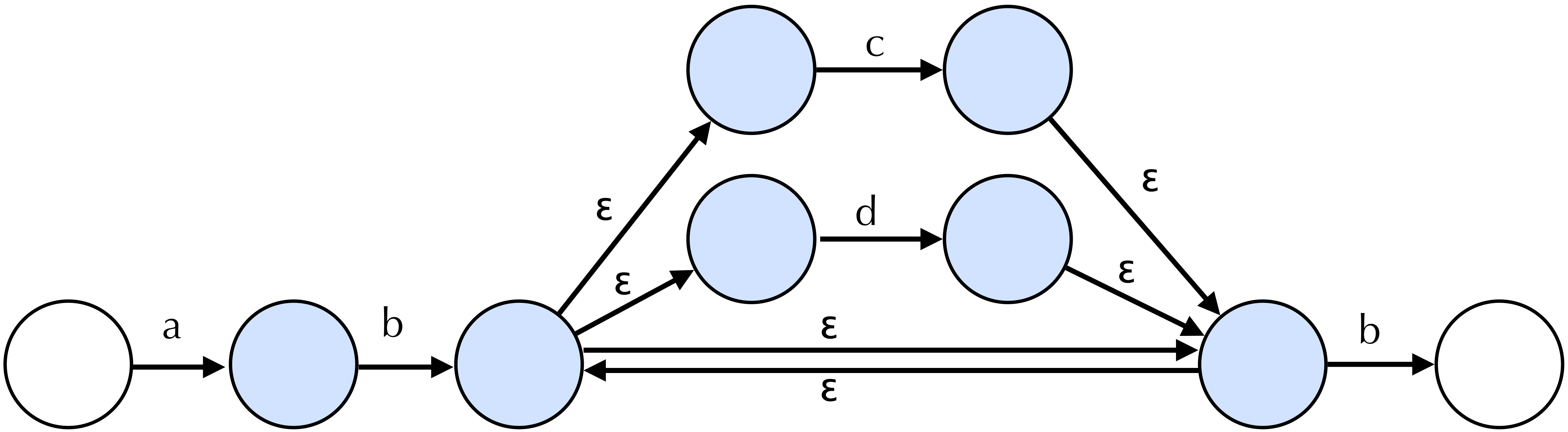
REAL

Combined finite automaton



Non-Deterministic Finite State Automaton

a b (c | d)* b

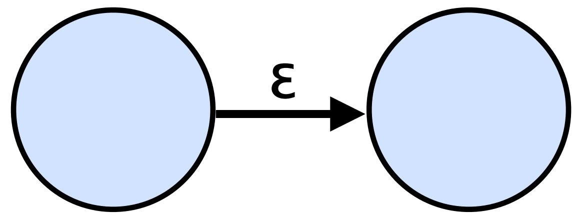


- Formally a non-deterministic finite state automaton (NFA) has
  - an alphabet Σ
  - a (finite) set V of states
  - distinguished start state
  - one or more accepting states
  - transition relation $\delta \subseteq V \times (\Sigma + \varepsilon) \times V$

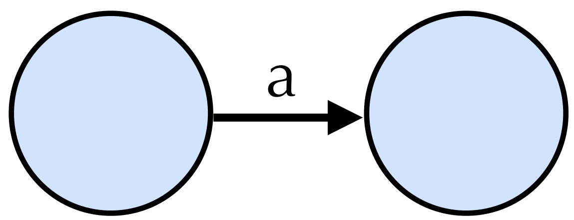
For this example, what's the alphabet, set of states, transition relation, etc.?

Translating Regular Expressions

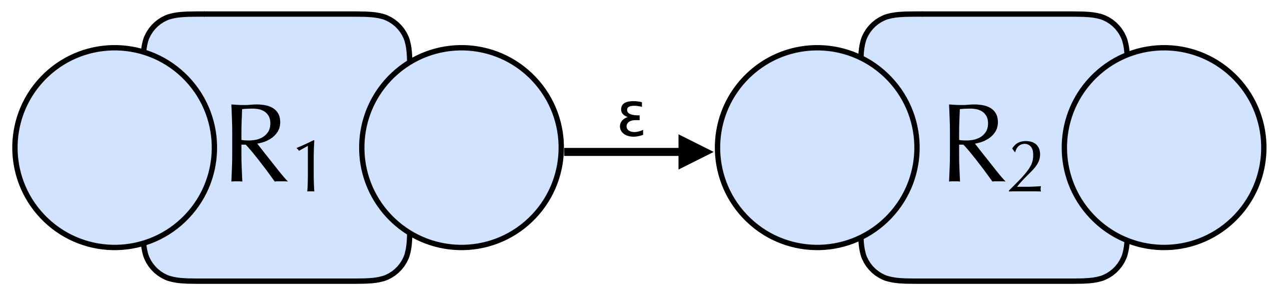
- Epsilon ε



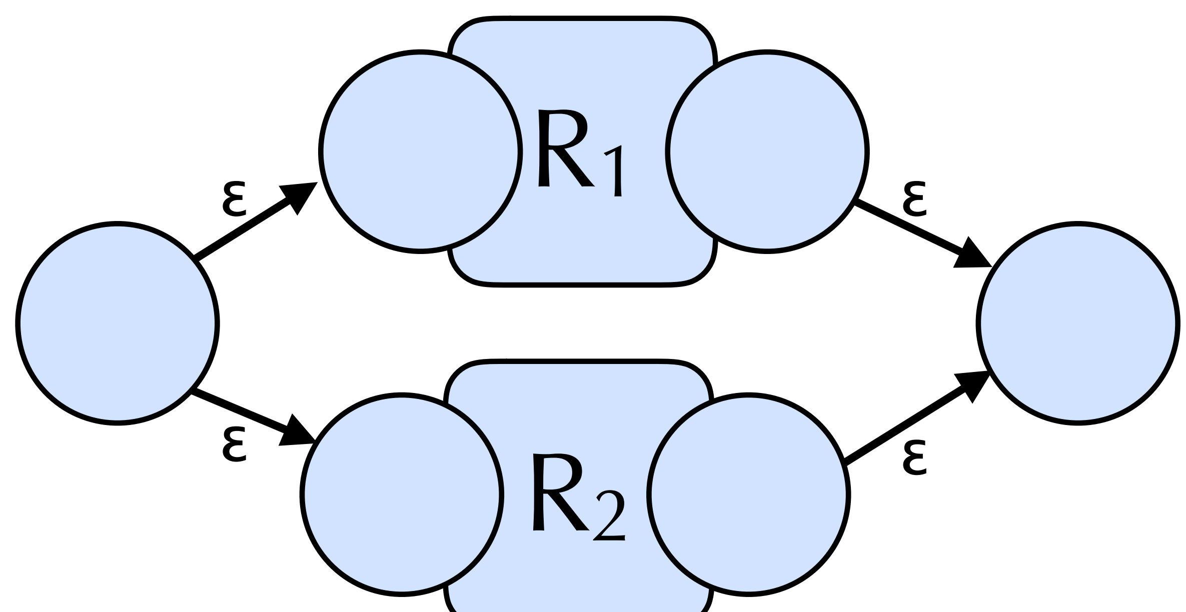
- Literal ‘a’



- Concatenation R_1R_2

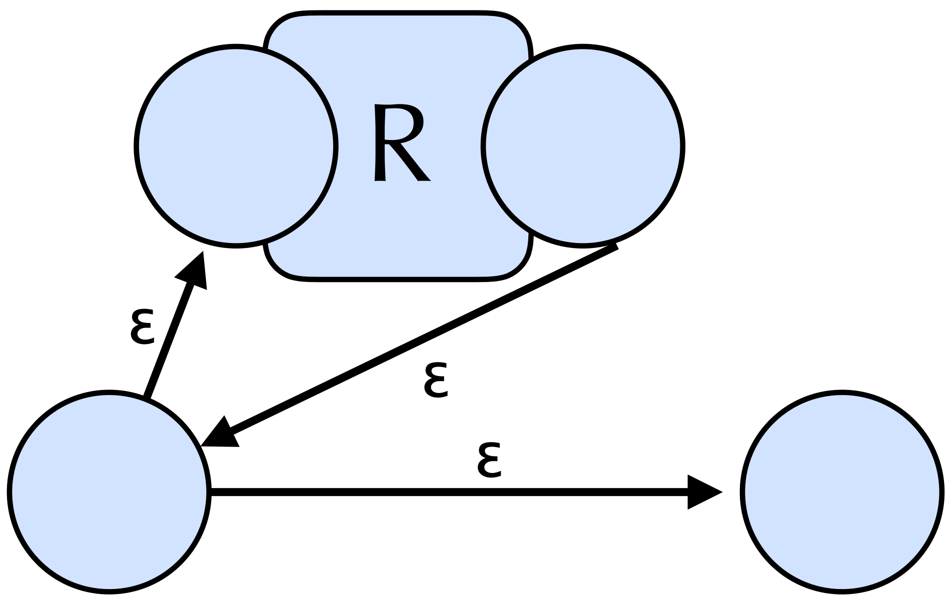


- Alternation $R_1 \mid R_2$



Translating Regular Expressions

- Kleene star R*



Converting to Deterministic

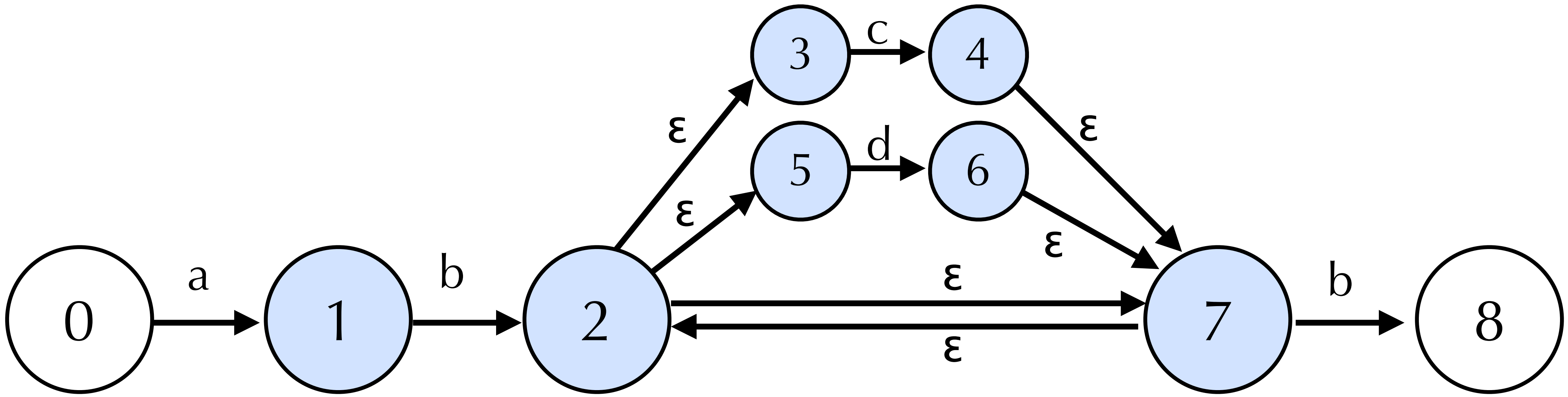
- NFAs are useful: easy to compose regular expressions
- But implementing an NFA is harder: it requires guessing which transition edge to take
- We can convert NFAs to Deterministic Finite Automata (DFAs)

Converting to Deterministic

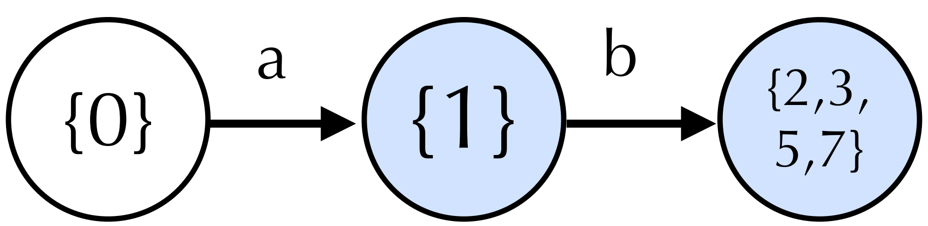
- Basic idea: each state in DFA will represent a **set of states** of the NFA
- Given set of NFA states S:
  - edge(S, 'a') = { NFA states reachable from S using 'a' edge }
  - closure(S) = S∪{ NFA states reachable from S using one or more **ε** edges }
- Algorithm sketch:
  - Start state of DFA is closure(s_0), where s_0 is DFA start state
  - Given DFA state S, and literal a, construct DFA state
    T = closure(edge(S, 'a')), and add edge from S to T labeled 'a'
    - Only if T is non-empty
  - Repeat until no more new DFA states
  - DFA state S is an accepting state if ∃ NFA accepting state s such that s∈S

Example: NFA to DFA

NFA:



DFA:

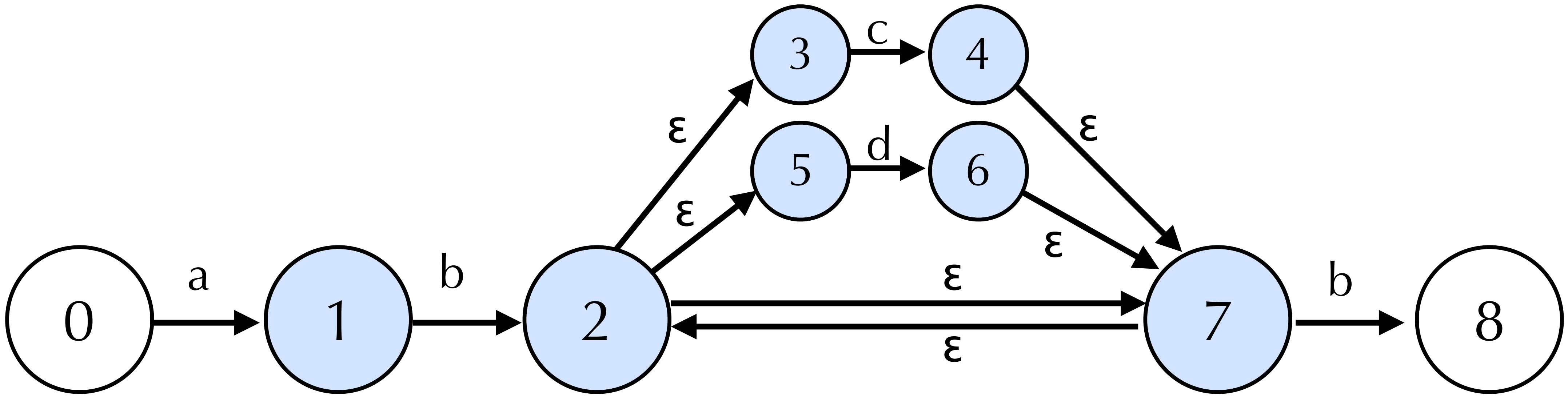


edge({1}, ‘b’) = {2}

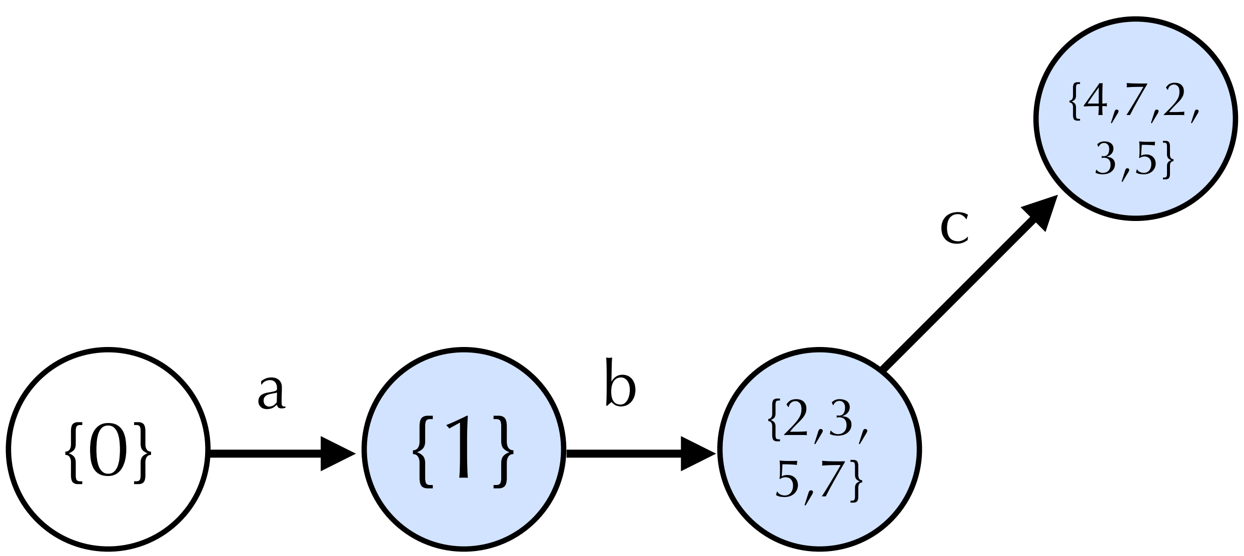
closure({2}) = {2,3,5,7}

Example: NFA to DFA

NFA:



DFA:

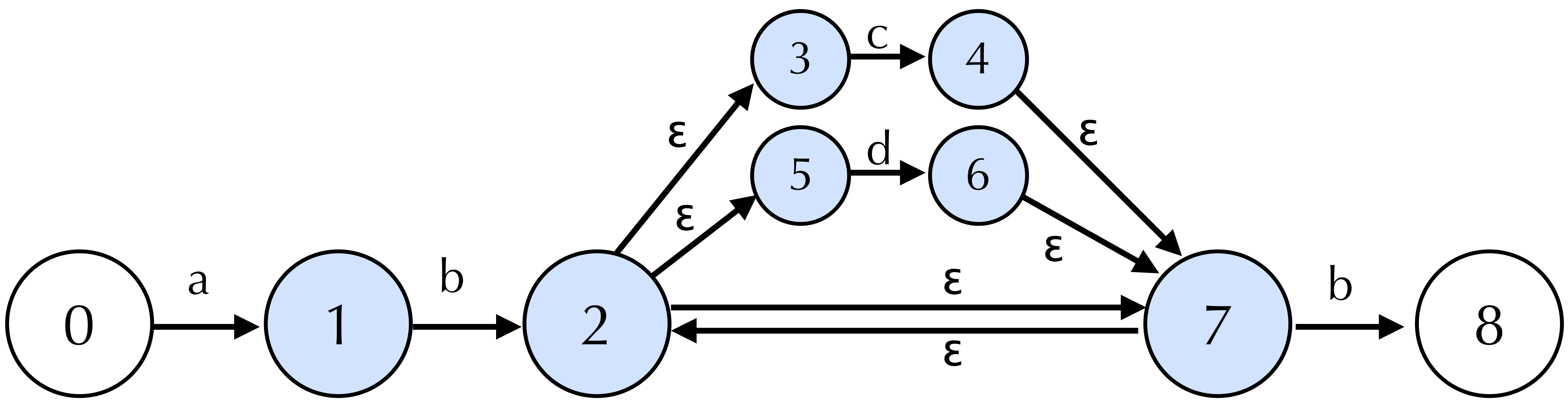


edge({2,3,5,7}, ‘c’) = {4}

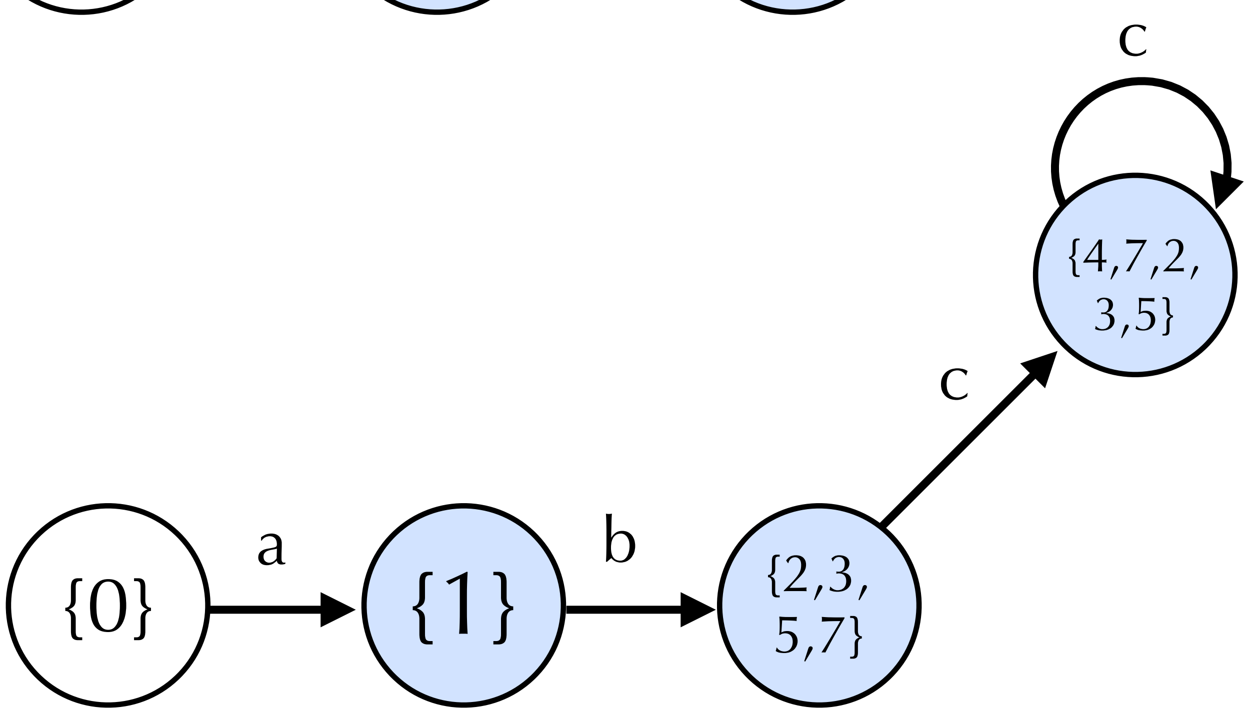
closure({4}) = {4,7,2,3,5}

Example: NFA to DFA

NFA:



DFA:

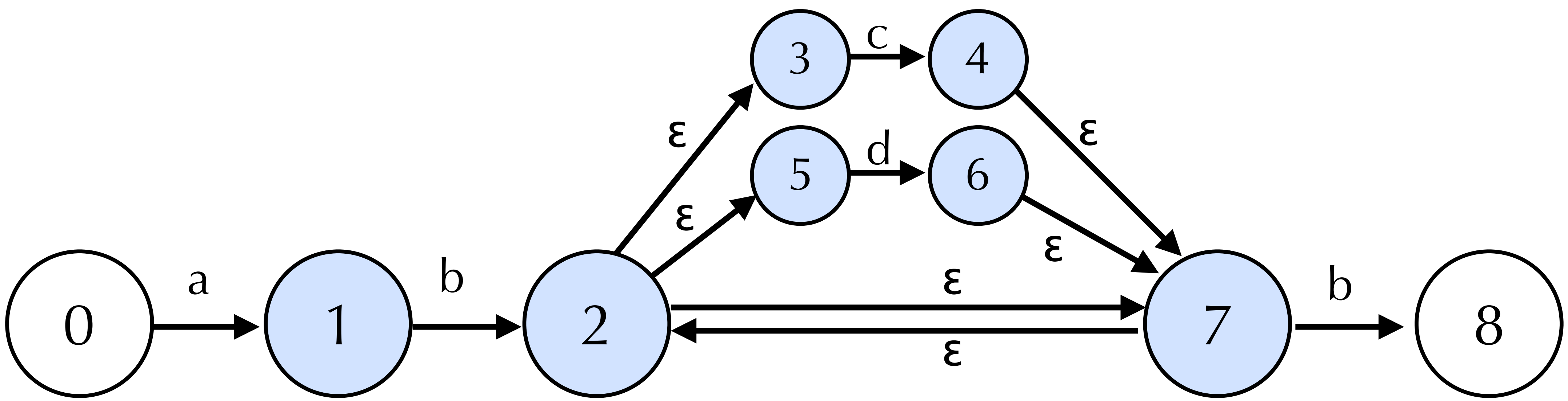


edge({4,7,2,3,5}, ‘c’) = {4}

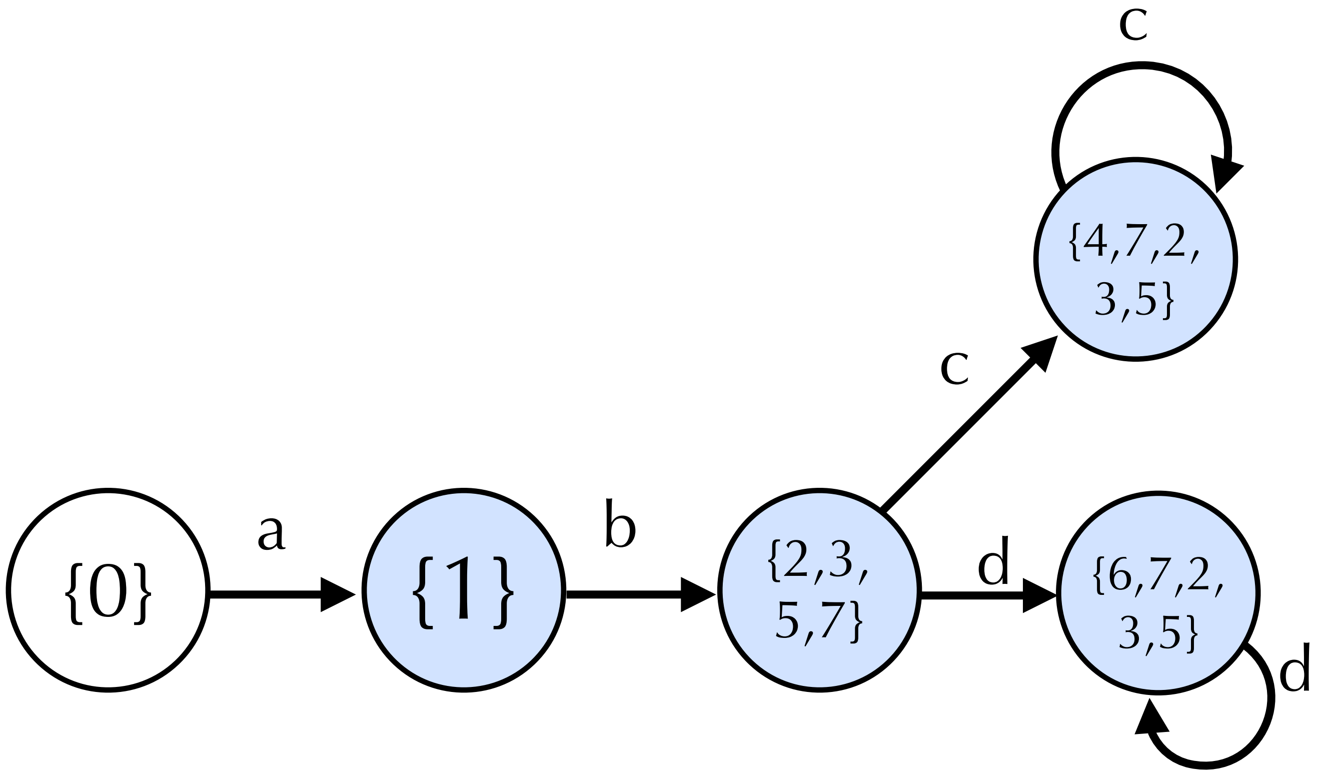
closure({4}) = {4,7,2,3,5}

Example: NFA to DFA

NFA:

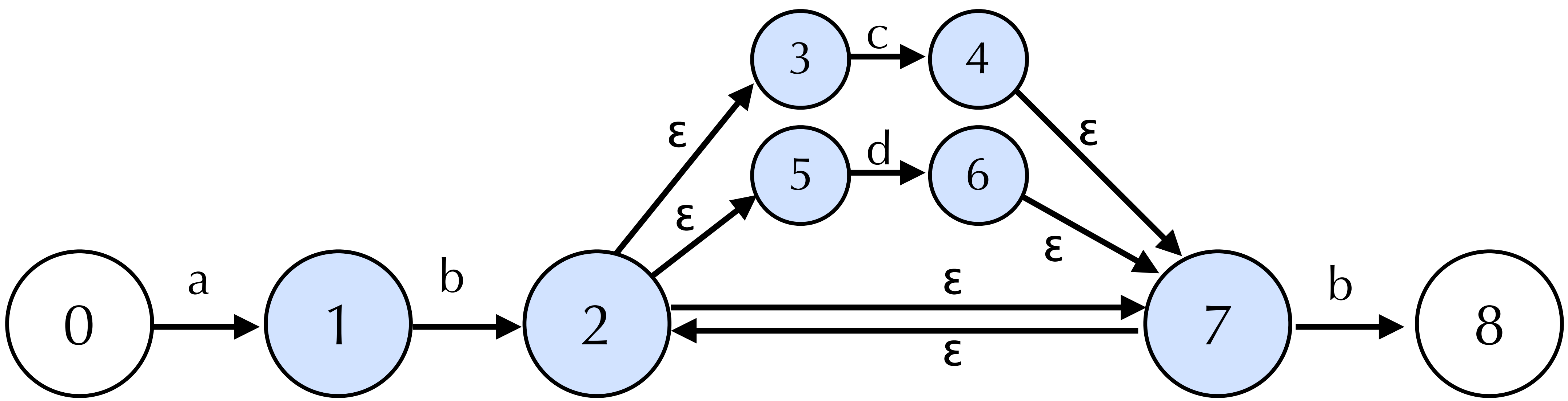


DFA:

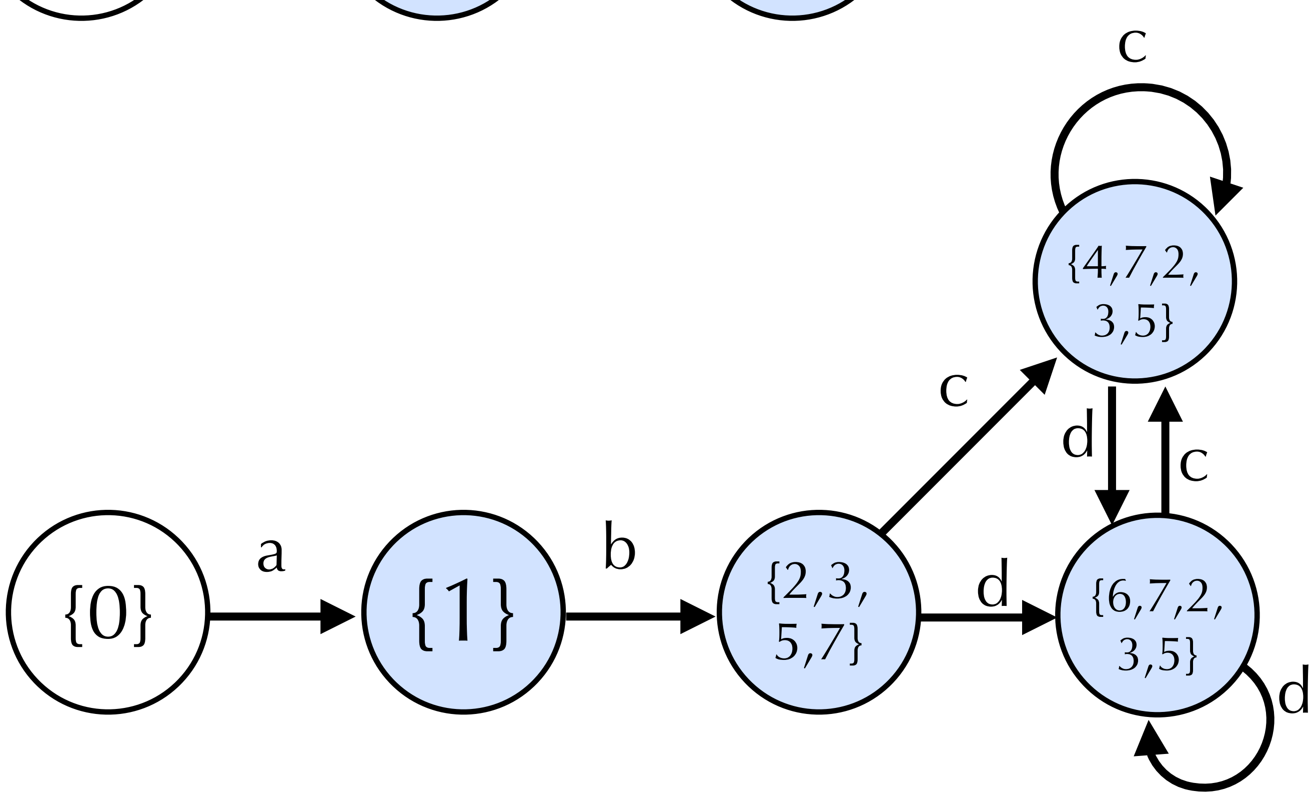


Example: NFA to DFA

NFA:



DFA:



edge({4,7,2,3,5}, ‘d’) = {6}

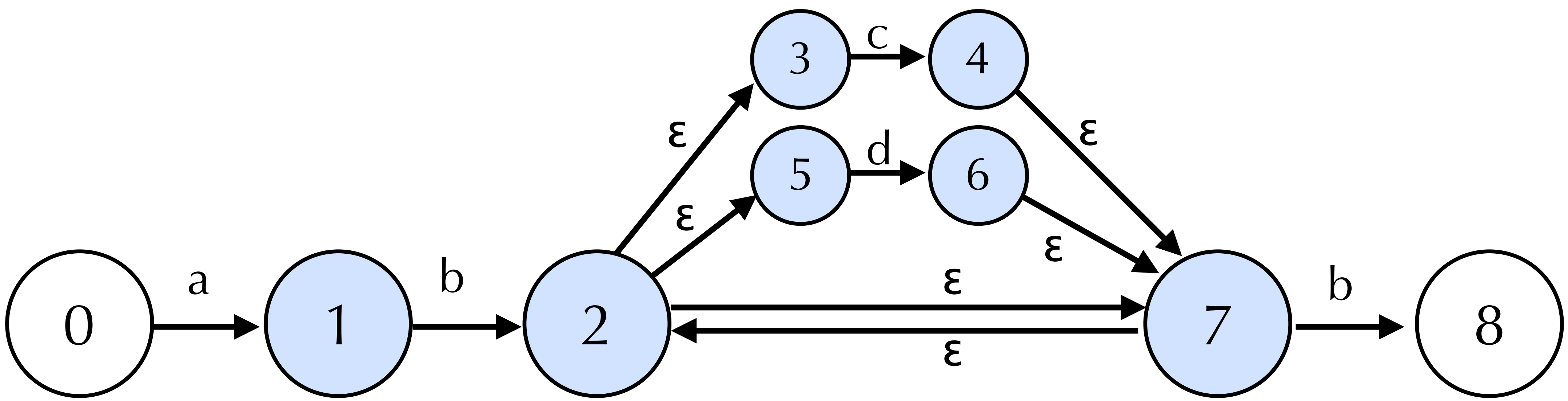
closure({6}) = {6,7,2,3,5}

edge({6,7,2,3,5}, ‘c’) = {4}

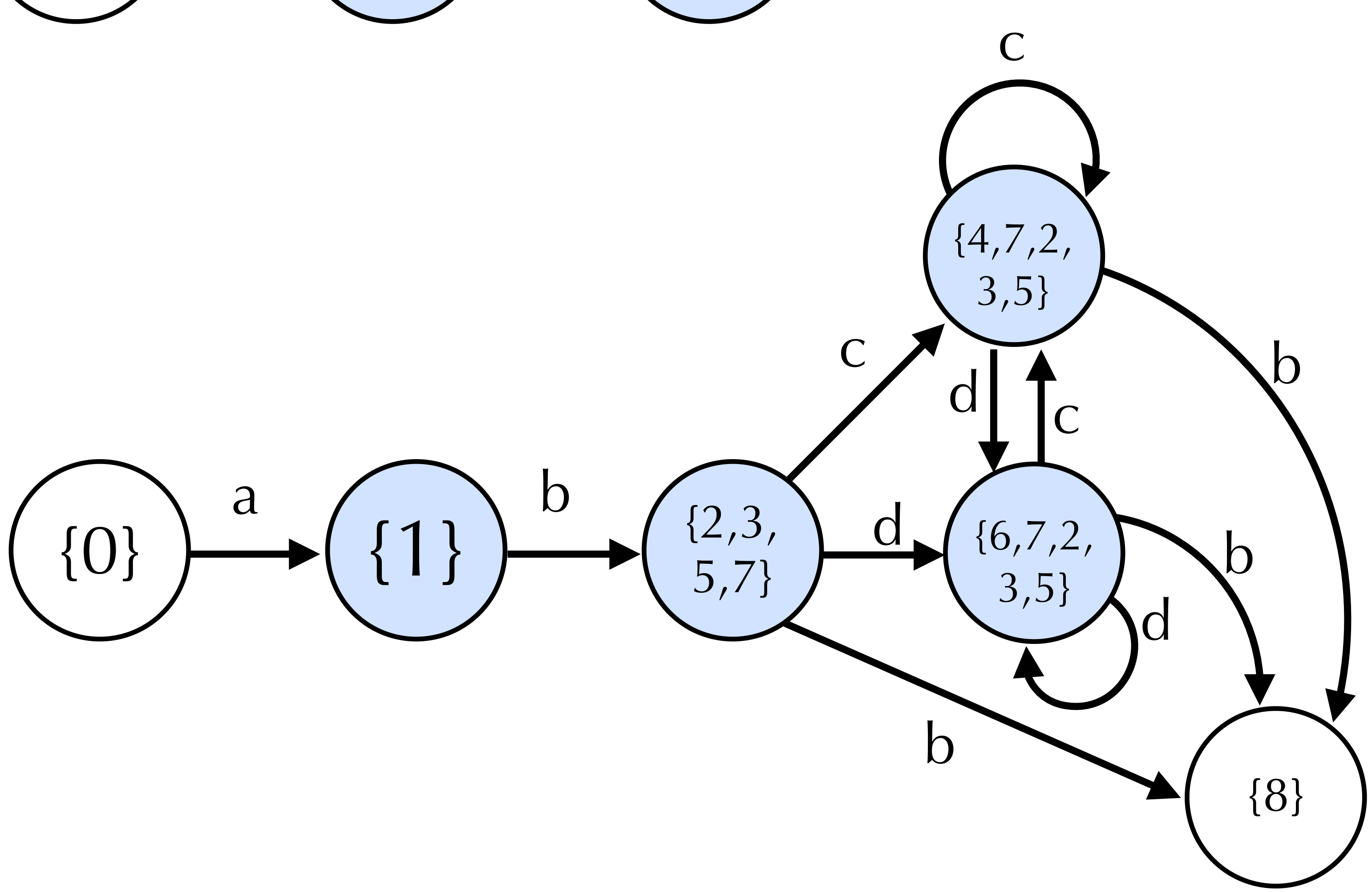
closure({4}) = {4,7,2,3,5}

Example: NFA to DFA

NFA:



DFA:



edge({2,3,5,7}, ‘b’) = {8}

edge({4,7,2,3,5}, ‘b’) = {8}

edge({6,7,2,3,5}, ‘b’) = {8}

closure({8}) = {8}

Check that this DFA is, in fact, deterministic!

Using DFAs

- DFAs are easy to simulate
  - For each state and character, there is at most one edge to take
- Usually record transition function as array indexed by state and characters
  - See Appel Chap 2.3 for an example, or the output of MLLex

Lexing

- We can now construct DFAs from regular expressions!
  - Enables matching of regular expressions
- But we need to produce sequence of tokens
- Let's look at some ad-hoc ML code for lexing
- Then MLLex example

Lexer example

- See file `Lec03-lexer.ml`

Interface for Basic Lexing

```
module type LEX = sig
  (* an ['a regexp] matches a string and returns an ['a] value *)
  type 'a regexp

  (* [ch c] matches ["c"] and returns ['c'] *)
  val ch : char -> char regexp

  (* [eps] matches [""] and returns [()] *)
  val eps : unit regexp

  (* [void] never matches (so never returns anything) *)
  val void : 'a regexp

  (* [r1 ++ r2] matches [s] and returns [v] if [r1] matches [s] and
     returns [v], or else [r2] matches [s] and returns [v]. *)
  val (++) : 'a regexp -> 'a regexp -> 'a regexp

  (* [r1 $ r2] matches [s] and returns [(v1,v2)] if [s = s1 ^ s2]
     and [r1] matches [s1] and returns [v1], and [r2] matches [s2]
     and returns [v2]. *)
  val ($) : 'a regexp -> 'b regexp -> ('a * 'b) regexp

  (* [star r] matches [s] and returns the list [vs] if either
     [s = ""] and [vs = []], or else [s = s1 ^ s2] and [vs = v1::v2]
     and [r] matchs s1 and returns v1, and [star r] matches [s2] and
     returns [v2]. *)
  val star : 'a regexp -> ('a list) regexp

  (* [r % f] matches [s] and returns [f(w)]
     if [r] matches [s] and returns [w] *)
  val (%) : 'a regexp -> ('a -> 'b) -> 'b regexp

  (* [lex r s] tries to match [s] against [r] and returns the list
     of all values that we can get out of the match. *)
  val lex : 'a regexp -> string -> 'a list
end
```

Extended RegExp

```
module ExtendLex(L : LEX) = struct
  include L

  (* matches one or more *)
  let plus(r: 'a regexp) : ('a list) regexp = (r $ (star r)) % cons

  (* when we want to just return a value and
     ignore the values we get out of r. *)
  let (%%) (r:'a regexp) (v:'b) : 'b regexp =  r % (fun _ -> v)

  (* optional match *)
  let opt(r:'a regexp) : 'a option regexp = (r % (fun x -> Some x)) ++ (eps %% None);;

  let alts (rs: ('a regexp) list) : 'a regexp = List.fold_right (++) rs void

  let cats (rs: ('a regexp) list) : ('a list) regexp =
    List.fold_right (fun r1 r2 -> (r1 $ r2) % cons) rs
      (eps % (fun _ -> []))

  (* Matches any digit *)
  let digit : char regexp =
    alts (List.map (fun i -> ch (char_of_int (i + (int_of_char '0'))))
            [0;1;2;3;4;5;6;7;8;9])

  (* Matches 1 or more digits *)
  let natural : int regexp =
    (plus digit) %
    (List.fold_left (fun a c -> a*10 + (int_of_char c) - (int_of_char '0')) 0)

  (* Matches a natural or a natural with a negative sign in front of it *)
  let integer : int regexp =
    natural ++ (((ch '-') $ natural) % (fun (_,n) -> -n))

  (* Generate a list of numbers [i,i+1,...,stop] -- assumes i <= stop *)
  let rec gen(i:int)(stop:int) : int list =
    if i > stop then [] else i::(gen (i+1) stop)

  (* Matches any lower case letter *)
  let lc_alpha : char regexp =
    let chars = List.map char_of_int (gen (int_of_char 'a') (int_of_char 'z')) in
      alts (List.map ch chars)

  (* Matches any upper case letter *)
  let uc_alpha : char regexp =
    let chars = List.map char_of_int (gen (int_of_char 'A') (int_of_char 'Z')) in
      alts (List.map ch chars)

  (* Matches an identifier a la Ocaml:  must start with a lower case letter,
     followed by 1 or more letters (upper or lower case), an underscore, or a digit. *)
  let identifier : string regexp =
    (lc_alpha $ (star (alts [lc_alpha; uc_alpha; ch '_'; digit]))) %
    (fun (c,s) -> implode (c::s))
```

A Lexer for a Little ML Language

```
type token =
      INT of int | ID of string | LET | IN | PLUS | TIMES | MINUS | DIV | LPAREN
    | RPAREN | EQ ;;

 let keywords = [ ("let",LET) ; ("in",IN) ]

 (* here are the regexps for a little ML language *)
 let token_regexps = [
   integer % (fun i -> INT i) ;
   identifier % (fun s ->
                  try List.assoc s keywords
                  with Not_found -> ID s) ;
   (ch '+') %% PLUS ;
   (ch '*') %% TIMES ;
   (ch '-') %% MINUS ;
   (ch '/') %% DIV ;
   (ch '(') %% LPAREN ;
   (ch ')') %% RPAREN ;
   (ch '=') %% EQ ;
 ];;

 (* so we can define a regexp to match any legal token *)
 let token = alts token_regexps ;;

 (* white space *)
 let ws = (plus (alts [ch ' ' ; ch '\n' ; ch '\r' ; ch '\t'])) %% () ;;

 (* document -- zero or more tokens separated by one or more white spaces *)
 let doc : token list regexp =
   ((opt ws) $ ((star ((token $ ws) % fst)) $ (opt token))) %
   (fun p -> let (_,(ts,topt)) = p in
       match topt with
         | None -> ts
         | Some t -> ts @ [t])
end
```

Implementing Basic Lexing Interface

```
module Lex =
struct
  (* Given a char list, this returns a list of pairs of an ['a]
     and the unconsumed characters. (It's a list of pairs to
     handle nondeterminism.)

     The only problem with this is that it will loop forever
     on certain regular expressions (e.g., (star eps)).
  *)
  type 'a regexp = char list -> ('a * char list) list

  let ch(c:char) : char regexp =
    function
      | c'::rest -> if c = c' then [(c,rest)] else []
      | _ -> []

  let eps : unit regexp = fun s -> [((), s)]

  let void : 'a regexp = fun s -> []

  let (++)(r1 : 'a regexp) (r2: 'a regexp) : 'a regexp =
    fun s -> (r1 s) @ (r2 s)

  let ($)(r1: 'a regexp) (r2:'b regexp) : ('a * 'b) regexp =
    fun s ->
      List.fold_right
        (function (v1,s1) -> fun res ->
          (List.fold_right
            (function (v2,s2) ->
              fun res -> ((v1,v2),s2)::res) (r2 s1) res)) (r1 s) []

  let (%) (r:'a regexp) (f:'a -> 'b) : 'b regexp =
    fun s ->
      List.map (function (v,s') -> (f v,s')) (r s)

  let rec star(r:'a regexp) : ('a list) regexp =
    fun s -> (((r $ (star r)) % cons) ++ (eps % (fun _ -> []))) s

  let lex (r: 'a regexp) (s:string) : 'a list =
    let results = r (explode s) in
    let uses_all = List.filter (fun p -> snd p = []) results in
      List.map fst uses_all
end

module ExtendedLex = ExtendLex(Lex)
```

ocamllex example

- Lexer generator
- `ocamllex` *`lexer`*`.mll`
- Produces an output file *`lexer`*`.ml`

Structure of ocamllex File

```
{ header }
let ident = regexp ...
rule entrypoint1 [arg1 ... argn] =
  parse regexp { action }
      | ...
      | regexp { action }
and entrypoint2 [arg1 ... argn] =
  parse ...
and ...
{ trailer }
```

- Header and trailer are arbitrary OCaml code, copied to the output file
- Can define abbreviations for common regular expressions
- Rules are turned into (mutually recursive) functions with `args1 ... argn lexbuf`
  - `lexbuf` is of type `Lexing.lexbuf`
  - Result of function is the result of ml code `action`

MLLex example

- See `Lec03-mllexeg.mll`