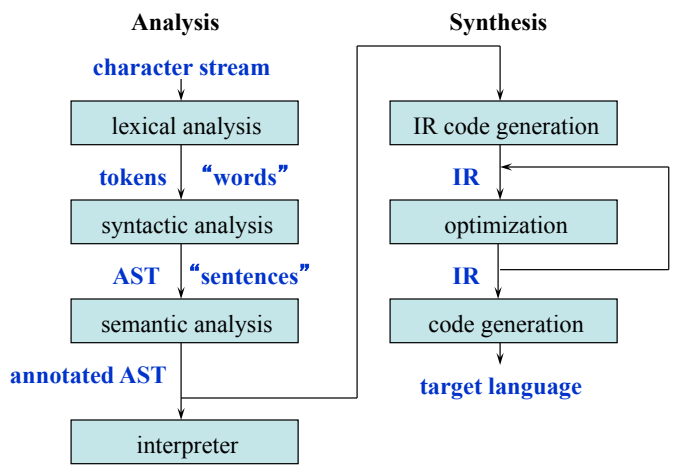
Plan for Today and Beginning Next week (Lexical Analysis)

Regular Expressions

Finite State Machines
- **DFAs: Deterministic Finite Automata**
- **Complications**
- **NFAs: Non Deterministic Finite State Automata**

From Regular Expressions to NFAs
From NFAs to DFAs

Structure of a Typical Compiler



Tokens for Example MeggyJava program

```
import meggy.Meggy;

class PA3Flower {
public static void main(String[] whatever){
   {
    // Upper left petal, clockwise
    Meggy.setPixel( (byte)2, (byte)4, Meggy.Color.VIOLET );
    Meggy.setPixel( (byte)2, (byte)1, Meggy.Color.VIOLET);
    …
   }
}
```

```
Tokens: Symbol(IMPORT,null), Symbol(MEGGY,null),
   Symbol(SEMI,null), Symbol(CLASS,null),
   Symbol(ID,"PA3Flower"), Symbol(LBRACE,null), …
```


About The Slides on Languages and Finite Automata

Slides Originally Developed by Prof. Costas Busch (2004)
- Many thanks to Prof. Busch for developing the original slide set.

Adapted with permission by Prof. Dan Massey (Spring 2007)
- Subsequent modifications, many thanks to Prof. Massey for CS 301 slides

Adapted with permission by Prof. Michelle Strout (Spring 2011)
- Adapted for use in CS 453
- **Adapted by Wim Bohm(added regular expr → NFA → DFA, Spr2012)**

Languages

A language is a set of strings
(sometimes called sentences)

String: A finite sequence of letters

Examples: "cat", "dog", "house", …

Defined over a fixed alphabet:

$$\Sigma = \{a, b, c, \ldots, z\}$$

Empty String

A string with no letters: ε (sometimes λ is used)

Observations: $|\varepsilon| = 0$

$$\varepsilon w = w\varepsilon = w$$

$$\varepsilon abba = abba\varepsilon = abba$$

Regular Expressions

Regular expressions describe regular languages
You have probably seen them in OSs / editors

Example: $(a \mid (b)(c))^*$

describes the language

$$L((a \mid (b)(c))^*) = \{\varepsilon, a, bc, aa, abc, bca, \ldots\}$$

Recursive Definition for Specifying Regular Expressions

Primitive regular expressions: $\varnothing, \ \varepsilon, \ \alpha$

where $\alpha \in \Sigma$, *some alphabet*

Given regular expressions r_1 *and* r_2

$$\left.\begin{array}{l} r_1 \mid r_2 \\ r_1 \ r_2 \\ r_1 \ * \\ (r_1) \end{array}\right\} \text{Are regular expressions}$$

Regular operators

choice:	A | B	a string from L(A) or from L(B)
concatenation:	A B	a string from L(A) followed by a string from L(B)
repetition:	A*	0 or more concatenations of strings from L(A)
	A^+	1 or more
grouping:	(A)	

Concatenation has precedence over choice: A|B C vs. (A|B)C

More syntactic sugar, used in scanner generators:

[abc] means a or b or c

[\t\n] means tab, newline, or space

[a-z] means a,b,c, …, or z

Example Regular Expressions and Regular Definitions

Regular definition:

name : regular expression

name can then be used in other regular expressions

Keywords "print", "while"

Operations: "+", "-", "*"

Identifiers:

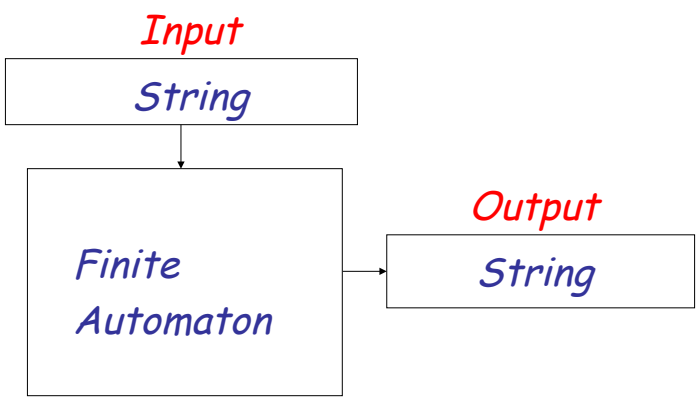
let : [a-zA-Z] // chose from a to z or A to Z

dig : [0-9]

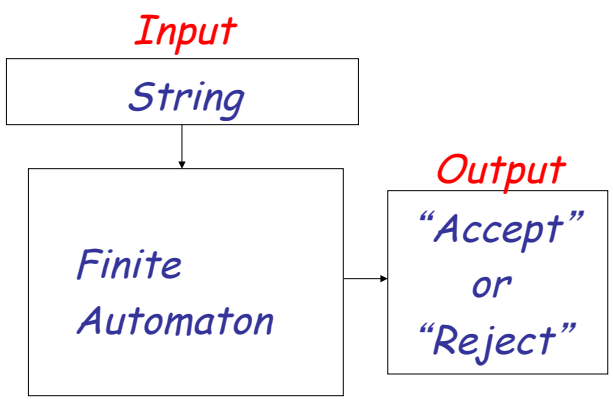
id : let (let | dig)*

Numbers: dig^+ = dig dig*

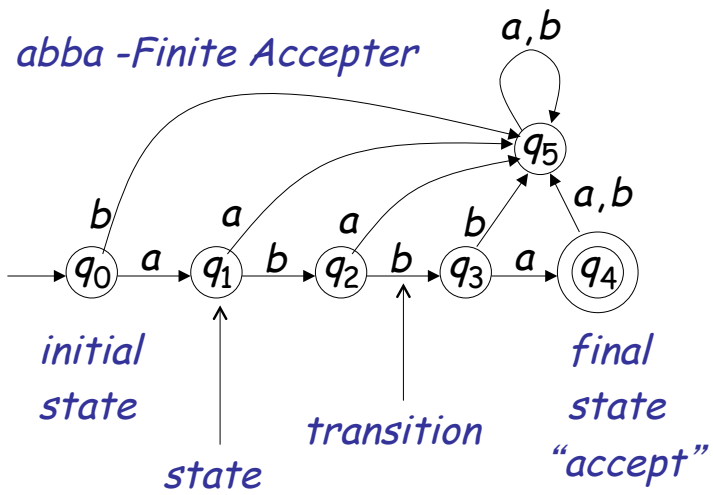
Finite Automaton



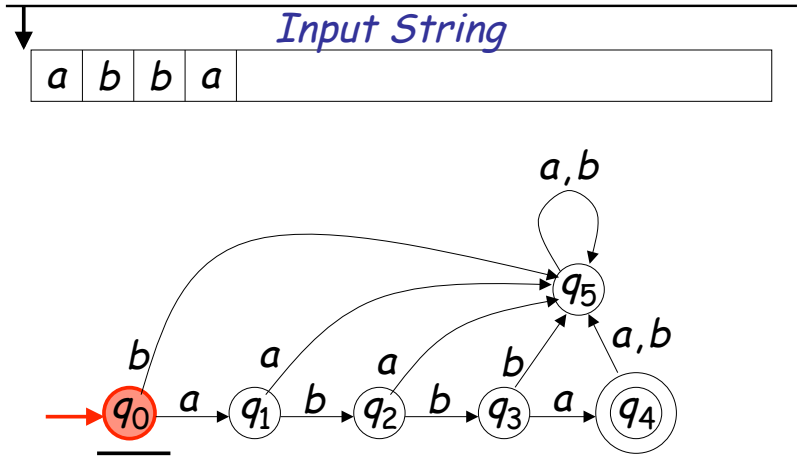
Finite Accepter



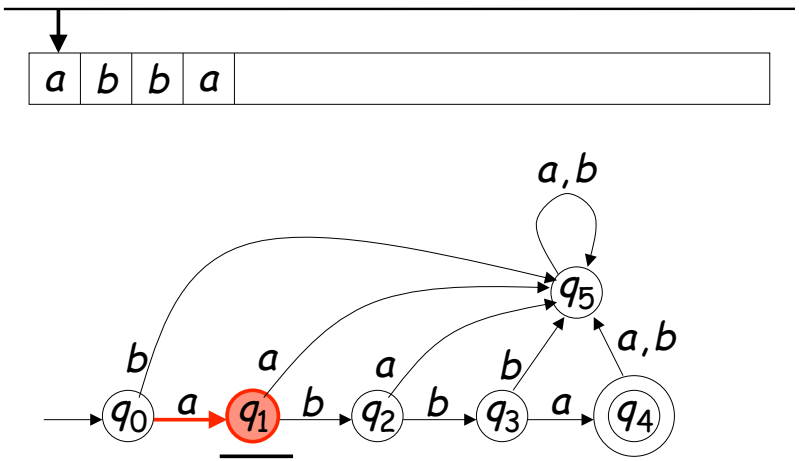
State Transition Graph

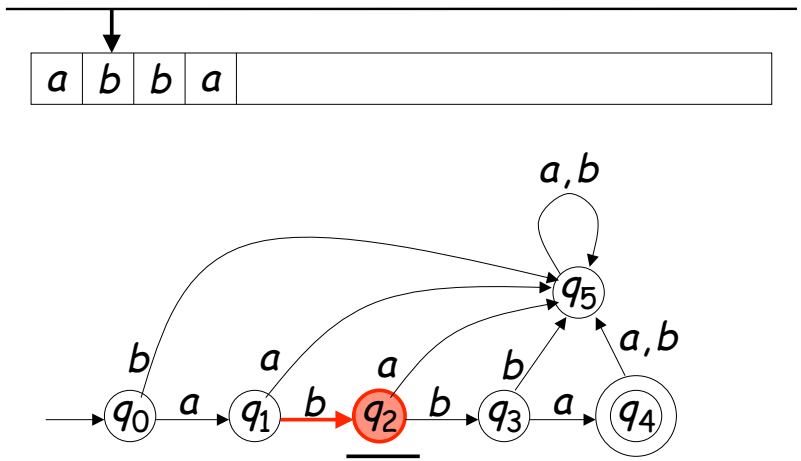


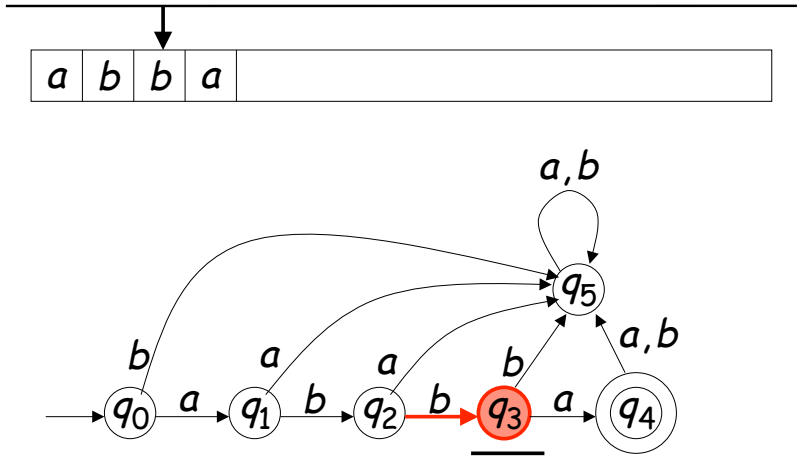
Initial Configuration

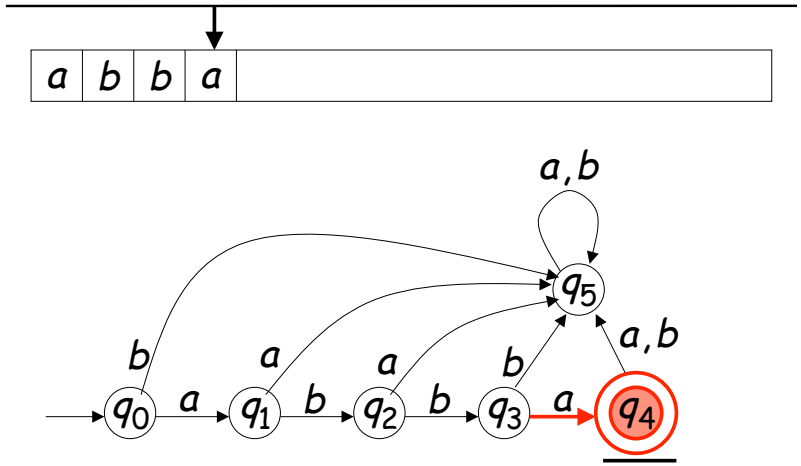


Reading the Input

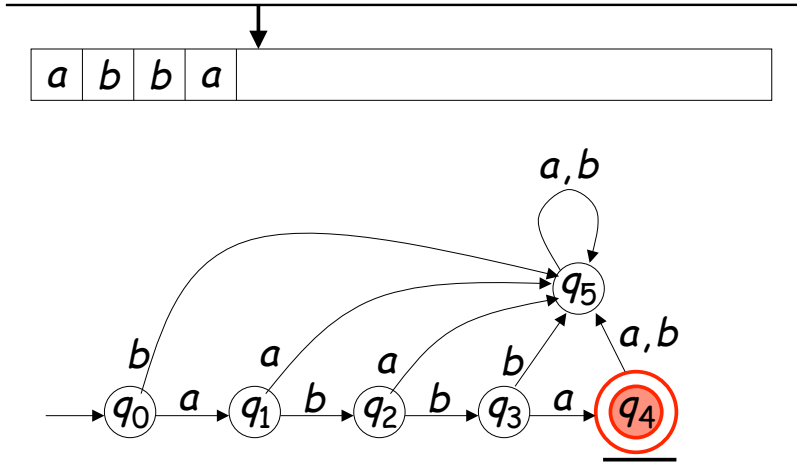






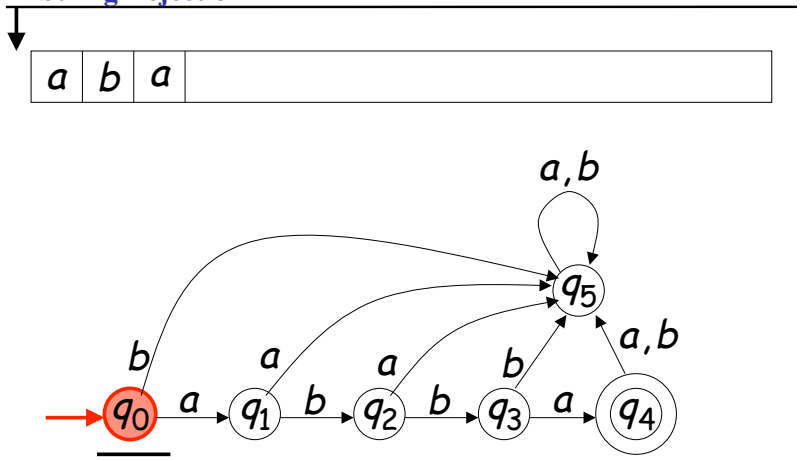


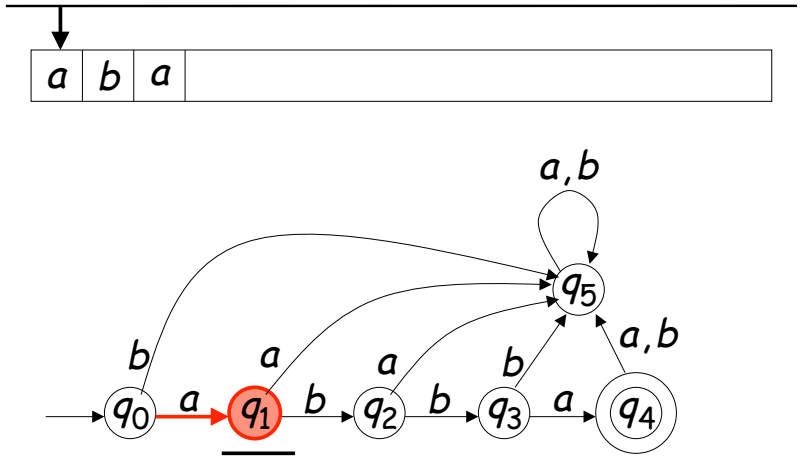
Input finished

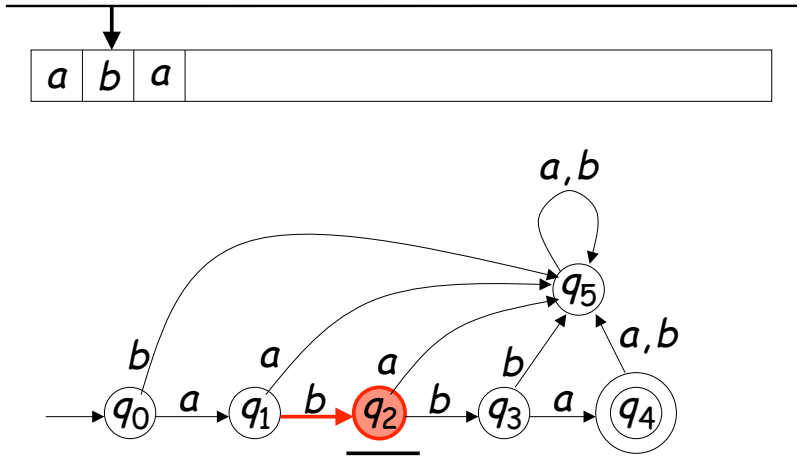


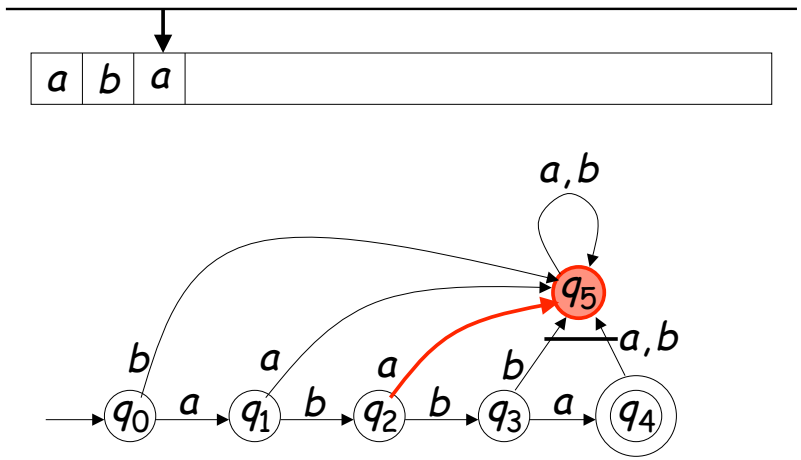
Output: "accept"

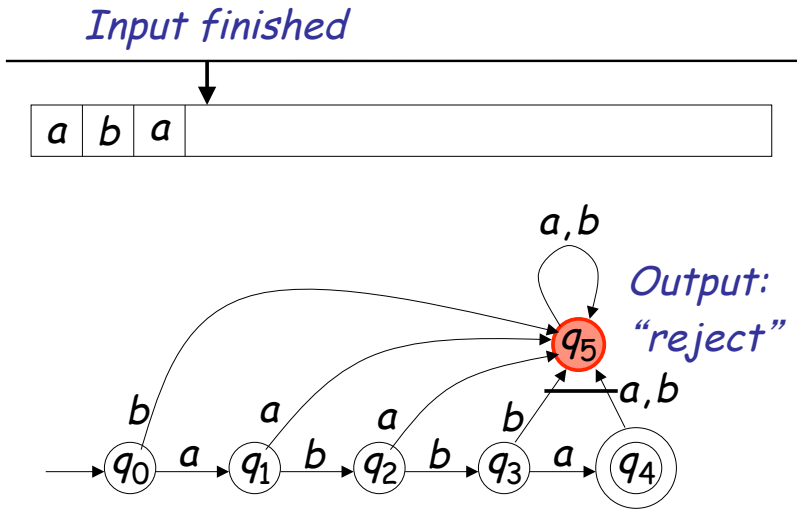
String Rejection



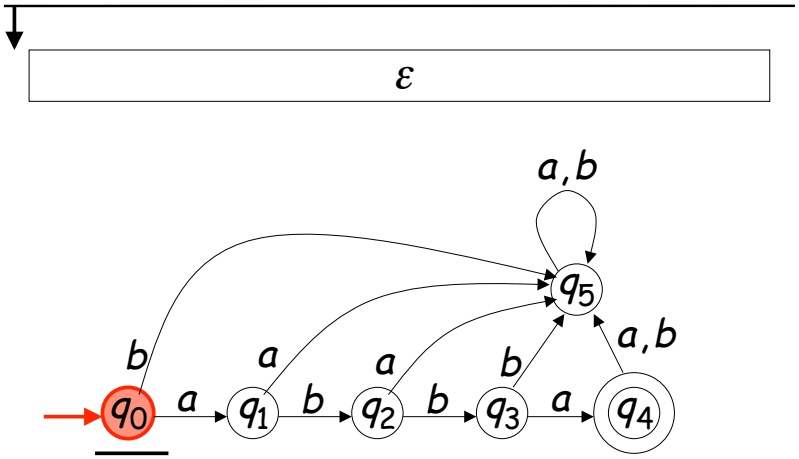

a b a
a,b
b
a
a
b
a,b
q0 a q1 b q2 b q3 a q4
q5

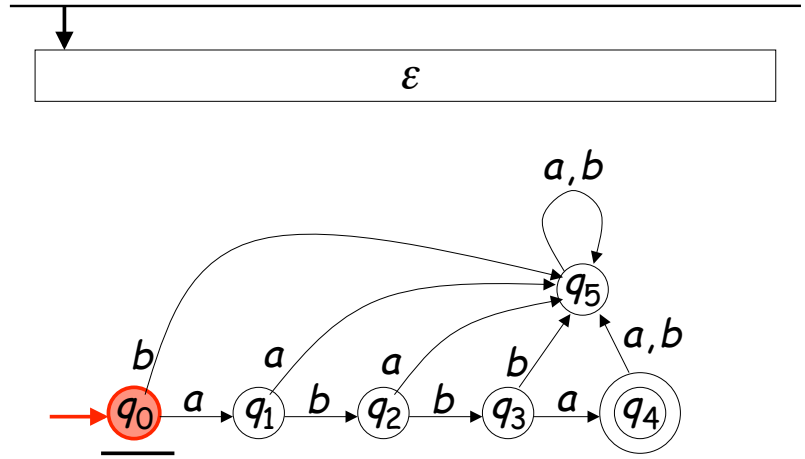

a b a
a,b
b
a
a
b
a,b
q0 a q1 b q2 b q3 a q4
q5


a b a
a,b
b
a
a
b
a,b
q0 a q1 b q2 b q3 a q4
q5


Input finished
a b a
a,b
Output:
"reject"
b
a
a
b
a,b
q0 a q1 b q2 b q3 a q4
q5

The Empty String

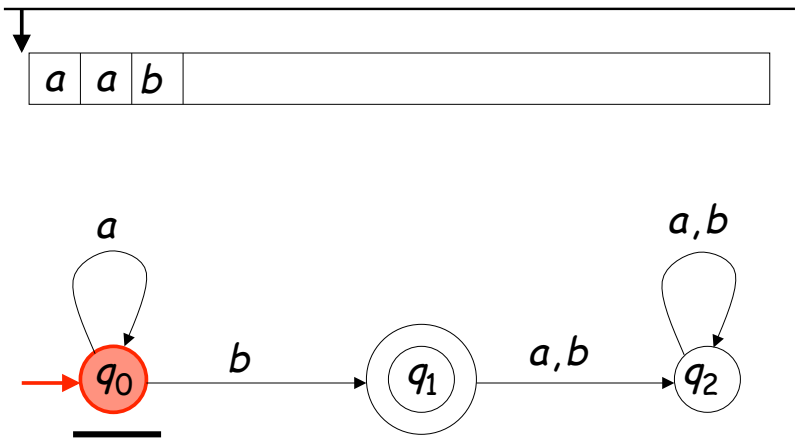


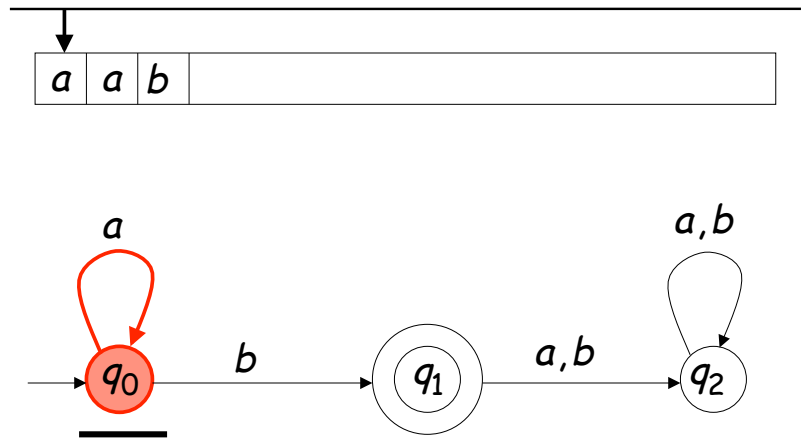


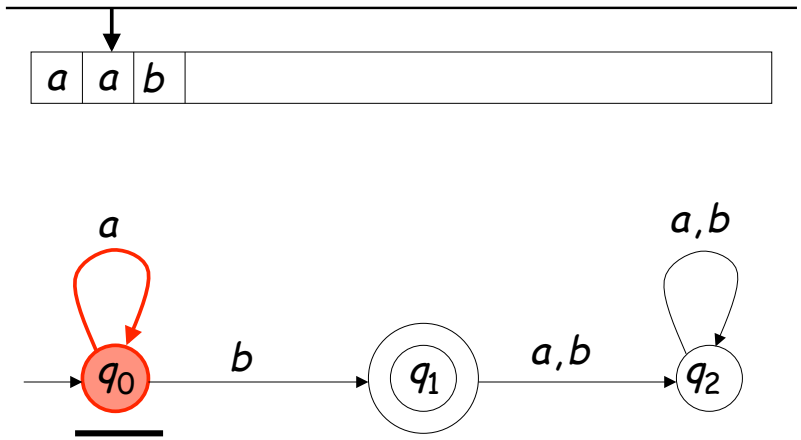
Output: "reject"

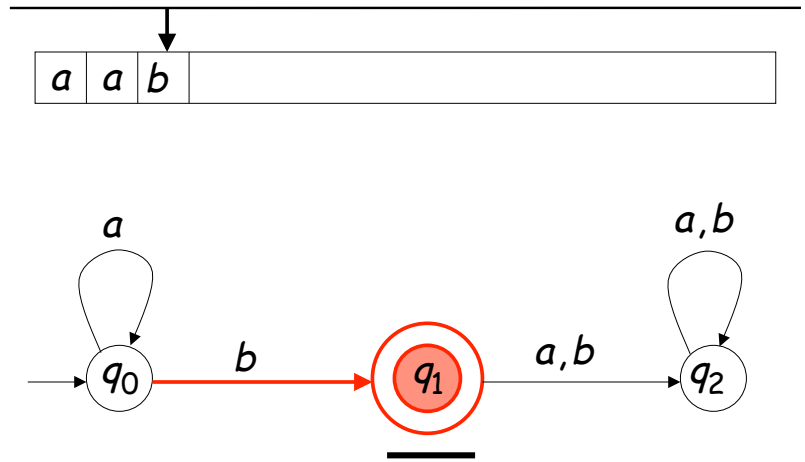
Would it be possible to accept the empty string?

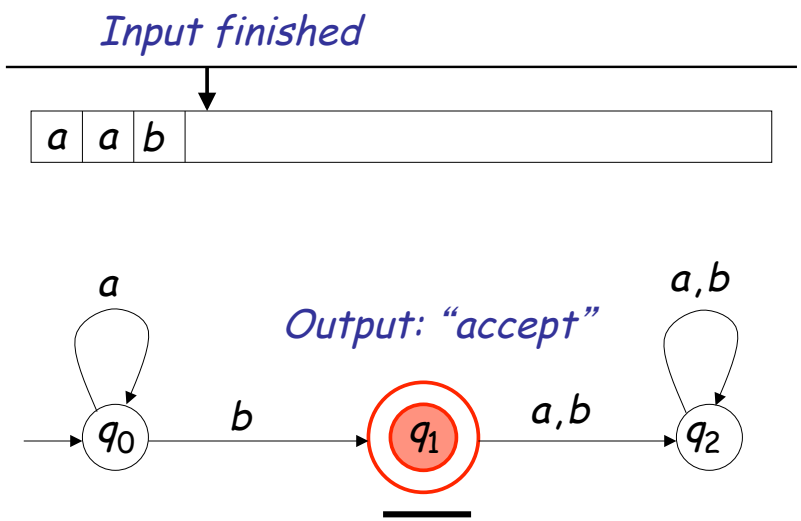
Another Example

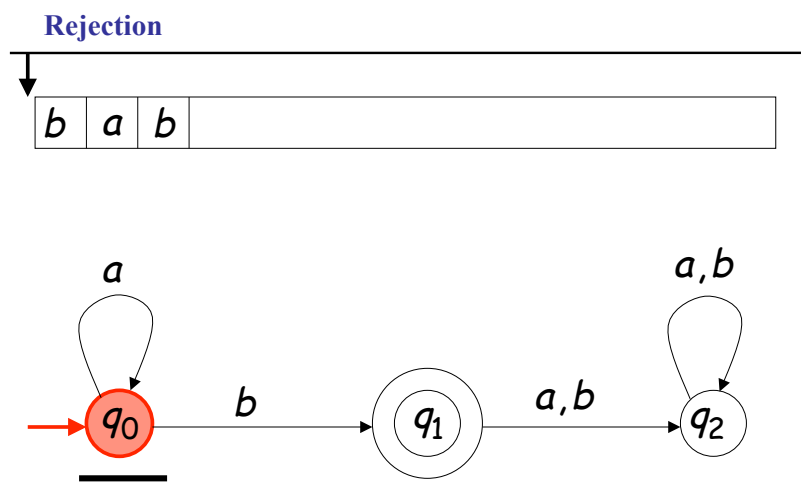


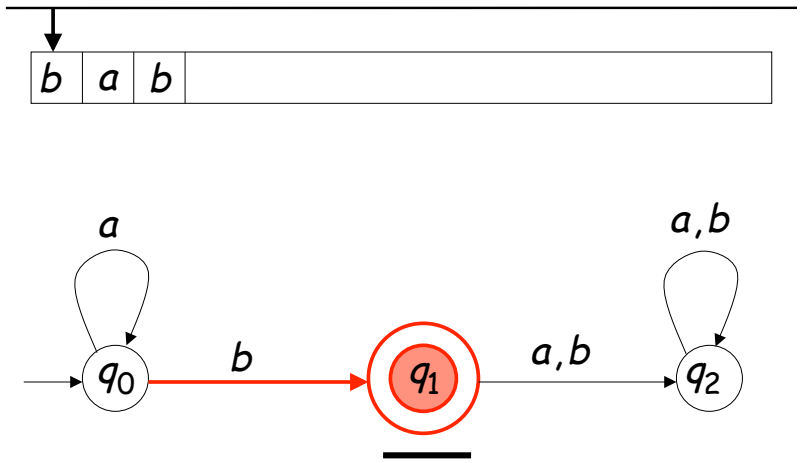


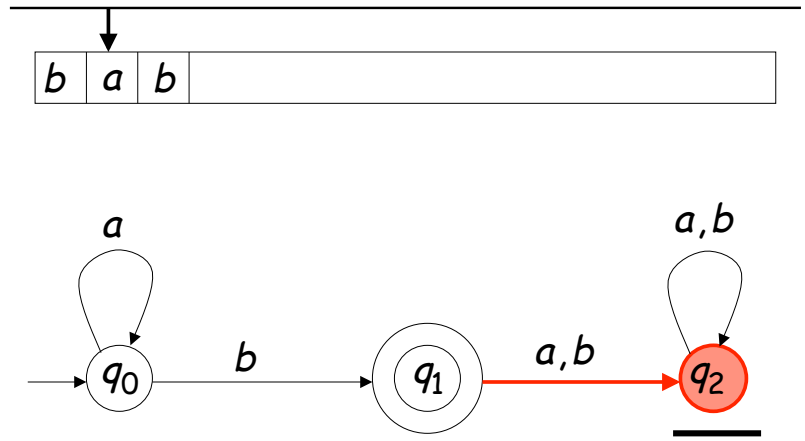

a a b
a
a,b
b
a,b
q0
q1
q2

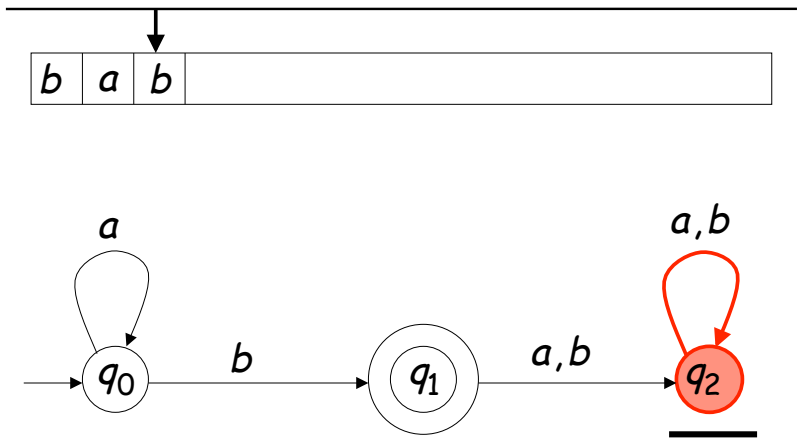

a a b
a
a,b
b
a,b
q0
q1
q2


Input finished
a a b
a
a,b
Output: "accept"
b
a,b
q0
q1
q2

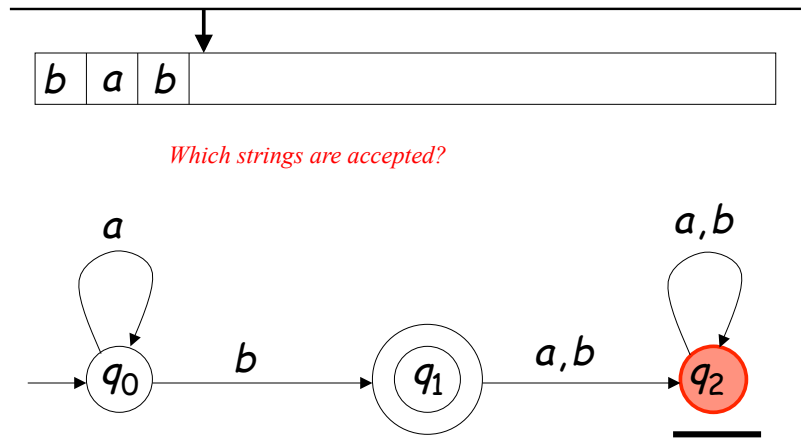

Rejection
b a b
a
a,b
b
a,b
q0
q1
q2







Input finished



Which strings are accepted?

Output: "reject"

Formalities

Deterministic Finite Automaton (DFA)

$$M = (Q, \Sigma, \delta, q_0, F)$$

Q : *set of states*

Σ : *input alphabet*

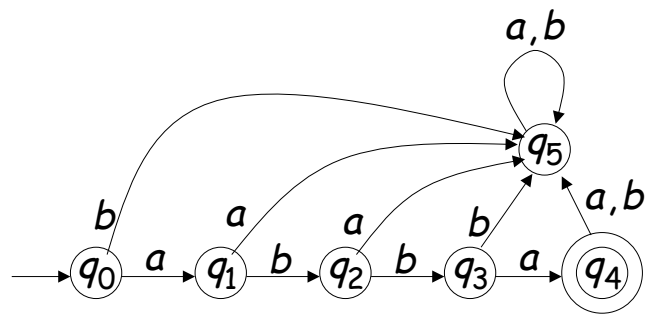
δ : *transition function*

q_0 : *initial state*

F : *set of final (accepting) states*

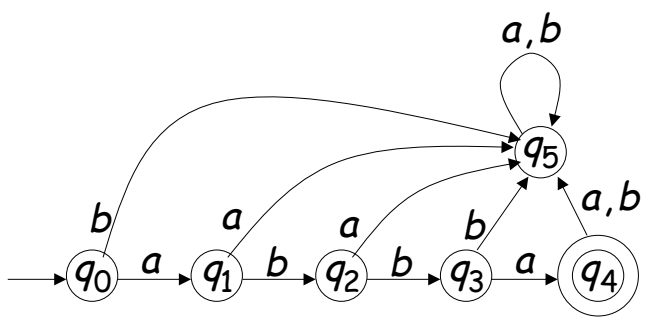
Input Alphabet Σ

$$\Sigma = \{a, b\}$$

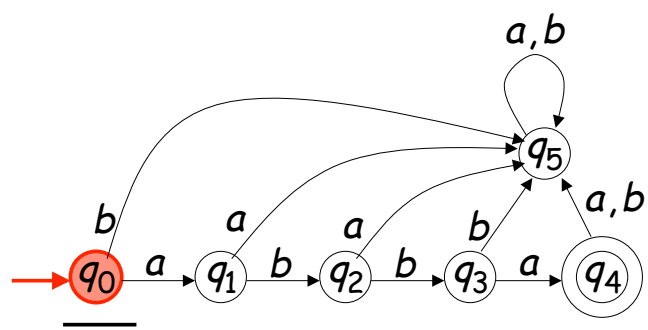


Set of States Q

$$Q = \{q_0, q_1, q_2, q_3, q_4, q_5\}$$

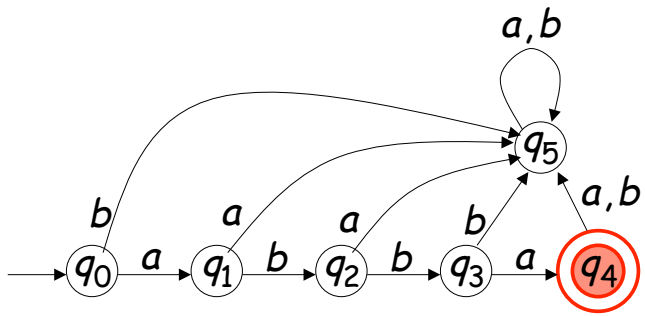


Initial State q_0



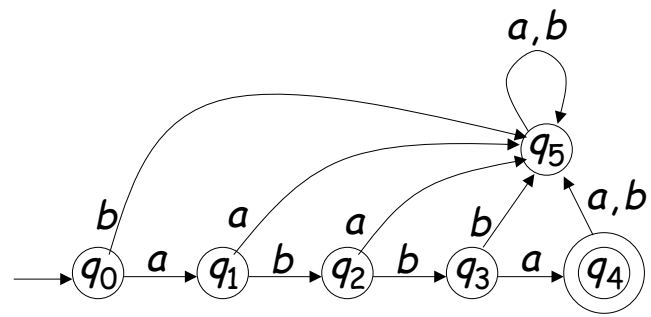
Set of Final States F

$$F = \{q_4\}$$

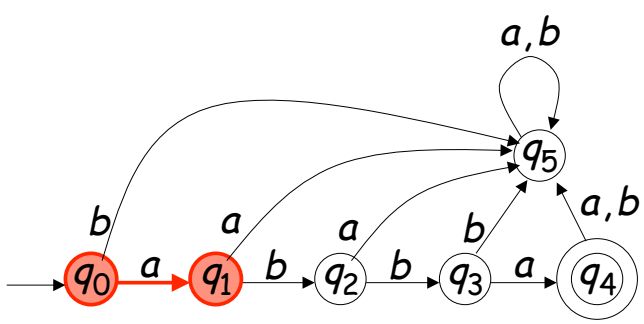


Transition Function δ

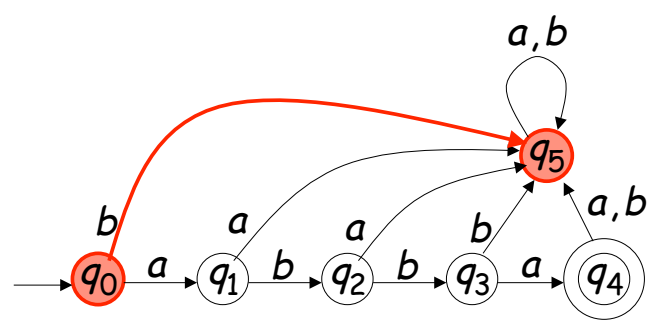
$$\delta : Q \times \Sigma \rightarrow Q$$



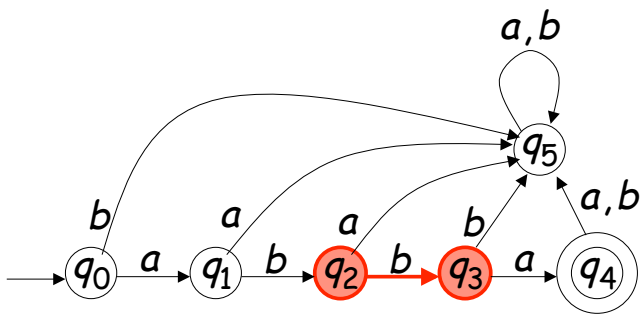
$$\delta(q_0, a) = q_1$$



$$\delta(q_0, b) = q_5$$

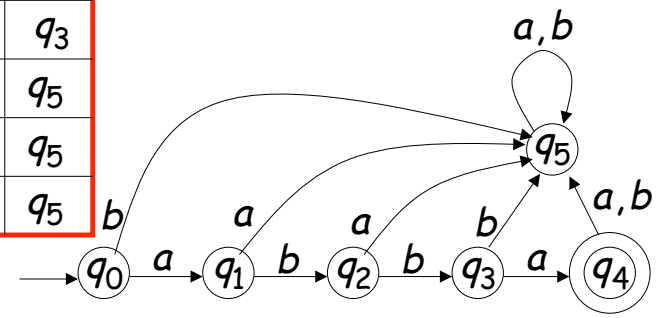


$$\delta(q_2, b) = q_3$$



Transition Function / table δ

δ	a	b
q_0	q_1	q_5
q_1	q_5	q_2
q_2	q_5	q_3
q_3	q_4	q_5
q_4	q_5	q_5
q_5	q_5	q_5



Complications

1. "1234" is an NUMBER but what about the "123" in "1234" or the "23", etc. Also, the scanner must recognize many tokens, not one, only stopping at end of file.

3. "if" is a keyword or reserved word IF, but "if" is also defined by the reg. exp. for identifier ID. We want to recognize IF.

4. We want to discard white space and comments.

5. "123" is a NUMBER but so is "235" and so is "0", just as "a" is an ID and so is "bcd", we want to recognize a token, but add attributes to it.

Complications 1

1. "1234" is an NUMBER but what about the "123" in "1234" or the "23", etc. Also, the scanner must recognize many tokens, not one, only stopping at end of file. So:

recognize the largest string defined by some regular expression, only stop getting more input if there is no more match. This introduces the need to reconsider a character, as it is the first of the next token

e.g. *fname(a,bcd);*

would be scanned as

ID OPEN ID COMMA ID CLOSE SEMI EOF

scanning *fname* would consume (, which would be put back and then recognized as OPEN

Complication 2

2. "if" is a keyword or reserved word IF, but "if" is also defined by the reg. exp. for identifier ID, we want to recognize IF, so

Have some way of determining which token (IF or ID) is recognized.

This can be done using priority, e.g. in scanner generators an earlier definition has a higher priority than a later one.

By putting the definition for IF before the definition for ID in the input for the scanner generator, we get the desired result.

What about the string "ifyouleavemenow"?

Complication 3

3. we want to discard white space and comments and not bother the parser with these. So:

in scanner generators, we can

specify, using a regular expression, white space e.g. [\t\n]

and return no token, i.e. move to the next

specify comments using a (NASTY) regular expression and again return no token, move to the next

Complication 4

4. "123" is a NUMBER but so is "235" and so is "0", just as "a" is an ID and so is "bcd", we want to recognize a token, but add attributes to it. So,

Scanners return Symbols, not tokens.
A Symbol is a (token, tokenValue) pair,
e.g. (NUMBER,123) or (ID,"a").

Often more information is added to a symbol, e.g. line number and position (as we will do in MeggyJava)

(Non) Deterministic Finite State Automata

A Deterministic Finite State Automaton (DFA) has disjoint character sets on its edges, i.e. the choice "which state is next" is deterministic.

A Non-deterministic Finite State Automaton (NFA) does NOT, i.e. it can have character sets on its edges that overlap (non empty intersection), and empty sets on the some edges (labeled ε).

NFAs are used in the translation from regular expressions to FSAs. E.g. when we combine the reg. exp for IF with the reg.exp for ID by just merging the two Transition graphs, we would get an NFA.

NFAs are a first step in creating a DFA for a scanner.
The NFA is then transformed into a DFA.

From regular expressions to NFAs

regexp

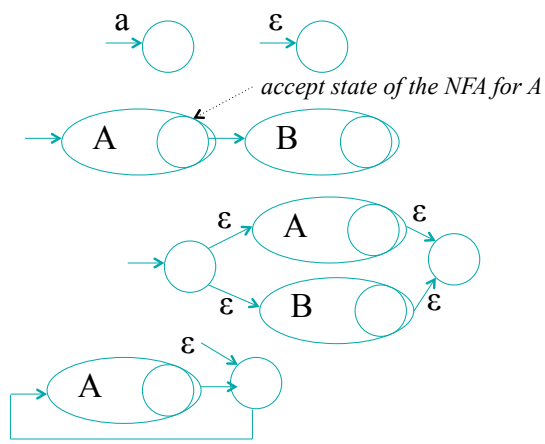
simple letter "a"

empty string

AB concat the NFAs

A|B split merge them

A* build a loop



The Problem

DFAs are easy to execute (table driven interpretation)

NFAs are easy to build from reg. exps, but hard to execute
we would need some form of guessing, implemented by back tracking

To build a DFA from an NFA we avoid the back track by taking all choices in the NFA at once, a move with a character or ε gets us to a set of states in the NFA, which will become one state in the DFA.

We keep doing this until we have exhausted all possibilities.
This mechanism is called transitive closure
(This ends because there is only a finite set of subsets of NFA states. How many are there?)

Example IF and ID

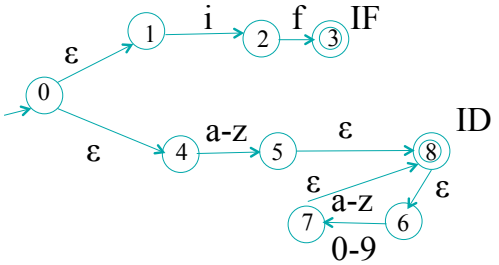
let : [a-z]
dig : [0-9]

tok : if | id

if : "i" "f"

id : let (let | dig)*

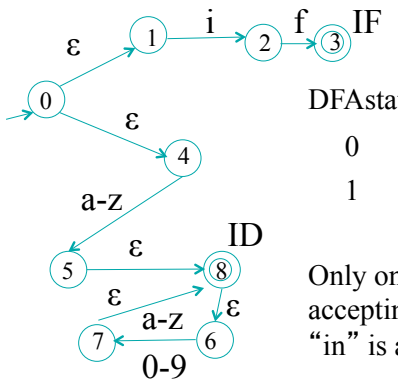
Example: NFA for IF and ID



let : [a-z]
dig : [0-9]
tok : if | id
if : "i" "f"
id : let (let | dig)*

IF has priority over ID.
From 0, with ε we can get to states 1 and 4
this is called an **ε-closure**
We can now simulate the behavior of the NFA and build a table for the DFA making character moves plus ε-closures

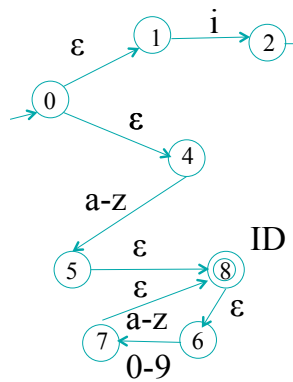
NFA simulation scanning "in"



DFAstate	NFAstates	Move	Next
0	0,1,4	i	2,5,8,6
1	2,5,6,8	n	6,7,8

Only one of the states in 6,7,8 is an accepting state, an ID accepting state, so "in" is an ID

NFA simulation scanning "if"



DFAstate	NFAstates	Move	Next
0	0,1,4	i	2,5,6,8
1	2,5,6,8	f	3,6,7,8

Two of the states in 3,6,7,8 are accepting, an IF accepting state (3) and an ID accepting state (8), IF has priority over ID, so "if" is an IF

Definitions: edge(s,c) and closure

edge(s,c): the set of all NFA states reachable from state s following an edge with character c

closure(S): the set of all states reachable from S with no chars or ε

$$closure(S) = T = S \cup (\bigcup_{s \in T} edge(s,\varepsilon))$$

T=S
repeat T' =T;
forall s in T' { T' =T; $T = T' \cup (\bigcup_{s \in T'} edge(s,\varepsilon))$ **}**
until T' ==T

This transitive closure algorithm terminates because there is a finite number of states in the NFA

DFAedge and NFA Simulation

Suppose we are in state DFA d = {s_i, s_k,s_l}
By moving with character c from d we reach a set of new NFA states, call these DFAedge(d,c), a new or already existing DFA state

$$DFAedge(d,c) = closure(\bigcup_{s \in d} edge(s,c))$$

NFA simulation:
let the input string be $c_1 \ldots c_k$
d=closure({s1}) // s_1 the start state of the NFA
for i from 1 to k
d = DFAedge(d,c_i)

Constructing a DFA with closure and DFAEdge

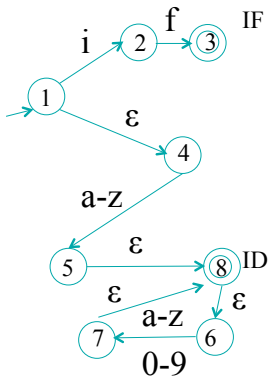
state d_1 = closure(s_1) the closure of the start state of the NFA

make new states by moving from existing states with a character c, using DFAEdge(d,c); record these in the transition table

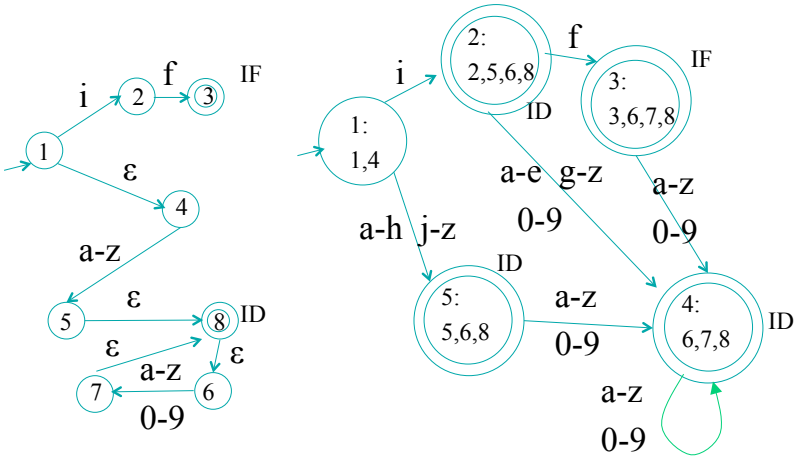
make accepts in the transition table, if there is an accepting state in d, decide priority if more than one accept state.

Instead of characters we use non-overlapping (DFA) character classes to keep the table manageable.

NFA to DFA (let's build it)



NFA to DFA



The transition table for IF ID

p	NFAstates(p)	i	f	a-h j-z	a-e,g-z 0-9	a-z,0-9	ACPT
1	{1,4}	{2,5,6,8}		{5,6,8}			
2	{2,5,6,8}		{3,6,7,8}		{6,7,8}		ID
3	{3,6,7,8}					{6,7,8}	IF
4	{6,7,8}					{6,7,8}	ID
5	{5,6,8}					{6,7,8}	ID

Suggested Exercise

Build an NFA and a DFA for integer and float literals

dot: “.”

dig: [0-9]

int-lit: dig$^+$

float-lit: dig* dot dig+