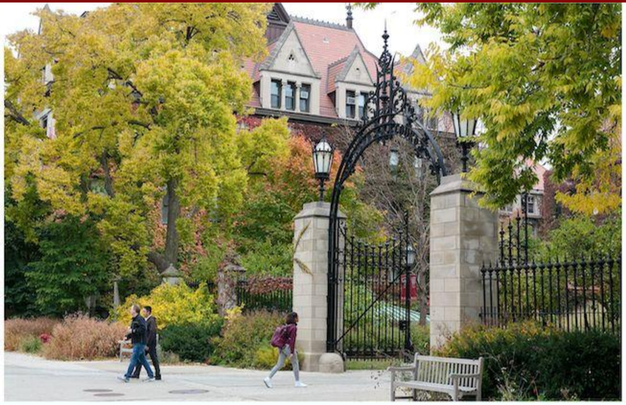
MPCS 51300 - Compilers
M2: Lexical Analysis (Scanner)

Remote Students please mute your microphones, thank you.


Lamont Samuels

THE UNIVERSITY OF CHICAGO | MASTERS PROGRAM IN COMPUTER SCIENCE

Agenda

- Lexical analysis overview
- Regular expressions
- (Nondeterministic) finite state automata (NFA)
- Converting NFAs to deterministic finite state automata (DFAs)
- Coding a Scanner

Lexical Analysis

- The main object of lexical analysis is to break the input source code into individual words, known as tokens (or lexemes)
- A lexical token is a series of character that can be treated as distinct objects that can carry associated data with them (e.g., numeric value, variable name, line numbers, etc.)
  - We use these tokens for next step of parsing
- A language classified lexical tokens into token types

Token Type	Examples
ID	bar num myList
INT	2 100 0 089
REAL	33.2 0.6 1e78
IF	if
RPAREN	)

- Lexical analysis ***may*** ignore whitespace and comments, or items not required to understand the meaning of the program.

Lexical Analysis Goal (Review)

- Input source code:

```
if (x==y) x=45;
```

- Character Stream:

I	f		(	x	=	=	y	)		x	=	4	5	;	

- Token Stream:

IF	LPAREN	ID(x)	EQ	ID(y)	RPAREN	ID(x)	ASSIGN	INT(45)	SCOLON

Lexical Analysis: Specifying Tokens

- The first step in lexical analysis is determining how we can specify our tokens.
- Most compilers use regular expressions to describe programming language tokens
  - A regular expression ***R*** defines a regular language ***L***, which is a set of strings over some alphabet Σ, such as ASCII characters or unicode.
    - Each member of the set is known as a word or sentence.
    - ***L(R)*** is the "language" defined by ***R***
      - L(xyz) = { "xyz" }
      - L(hello | world) = {"hello", "world"}
      - L([1-9][0-9]*) = all positive integer constants without a leading zero
- **Goal**: Define a regular expression for each kind of token

Regular Expression Fundamental Notation

- Given an alphabet Σ, the regular expressions over Σ and their corresponding regular languages are
  - $\varnothing$ denotes the empty set (empty language)
  - ε denotes the empty string
  - for each ***a*** in Σ, ***a*** denotes { a } the singleton set or the literal set
  - (Alternation) If ***R*** denotes ***L(R)*** and ***S*** denotes ***L(S)*** then ***R | S*** denotes any string from either ***L(R)*** or ***L(S)***
    - ***L(R|S) = L(R)*** ∪ ***L(S)***
  - (Concatenation) If ***R*** denotes ***L(R)*** and ***S*** denotes ***L(S)*** then ***RS*** denotes a string from ***L(R)*** followed by a string from ***L(S):***
    - ***L(RS) = {rs | r∈L(R) ^ s∈L(S)}***
  - (Kleene star) If ***R*** denotes ***L(R)*** then ***R**** denotes zero or more strings from ***L(R)*** concatenated together***.***
    - ***(ε | R | RR | RRR | RRRR | …)***
- Parentheses can be used to group REs if necessary
- Precedence (highest to lowest): parentheses, kleene star, concatenation, alternation

Regular Expression Examples

Regular Expression	Strings in L(R)
`a`	{"a"}
`ab`	{"ab"}
`a	b`	{"a", "b"}
`(ab)*`	{"", "ab", "abab", "ababa", …}

Convenient Regular Expression Shorthand

- The basic regular expression operations can produce all possible regular expressions; however, abbreviations exist for convenience

Abbreviation	Meaning	Explanation
r+	(rr*)	1 or more occurrences
r?	(r | ε)	0 or 1 occurrence
[a-z]	(A | b | c | … | z)	1 character in given range
[abcde]	(A | b | c | d | e)	1 of the given characters

- Note: There are many more abbreviations than these.

Lexical Specification

- We can define a lexical specification, which defines regular expressions to specify tokens

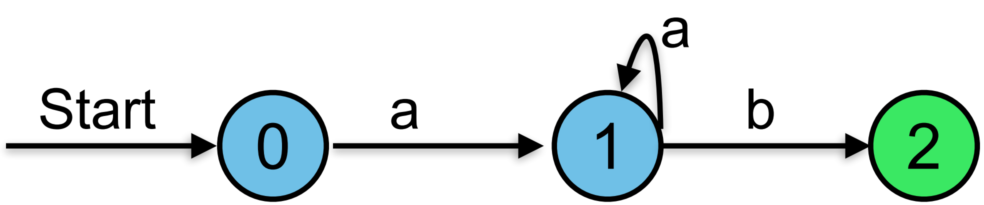
Regular Expression	Token
else	ELSE
[a-z][a-z0-9]*	ID
[0-9]+	INT

Regular Expression Implementation

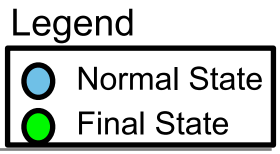
- How do we actually implement from a machine perspective the regular expressions in the specification?
  - The beginnings of implementing a scanner for languages is done by first converting the regular expressions into finite automata
    - A machine that recognize patterns.
    - Given a string ***s***, the scanner says "yes" if x is a word of the specified language and says "no" if cannot determine if it is part of the language.
- In order to understand finite automata you must understand transition diagrams.

Transition Diagram

- A flowchart that contains states and edges
  - Each edge is labeled with a character
  - A subset of states are designated as final (i.e. accepting) states.
- Transitions from state to state proceed along edges based on the next character from the character stream.



- Every string that ends in a final state is accepted
- If transitioning gets “stuck”; there is no transition for a given character then it’s an error.



Finite Automata

- Similar to transition diagrams
  - Have states and labelled edges
  - One unique start state and potentially one or more final states

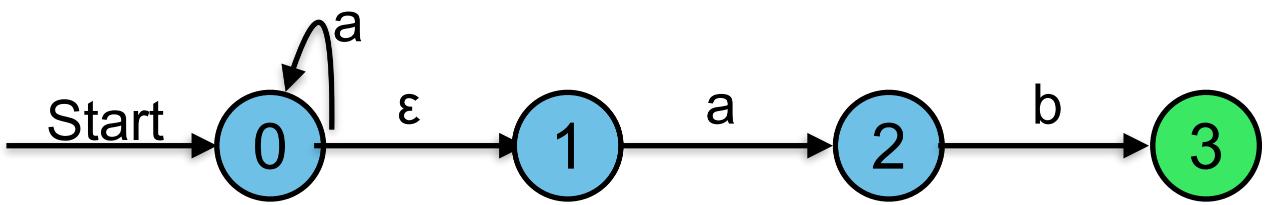
- Types of finite automata
  - Nondeterministic Finite Automata (NFA):
    - Can label edges with ε
    - A character can label 2 or more edges out of the same state

  - Deterministic Finite Automata (DFA):
    - No edges can be labeled with ε
    - A character can label at most one edge out of the same state

- Both NFAs and DFAs accepts a string x if there exists a path from start state to a final state labeled with characters in x
    - NFA can have multiple paths that could accept x
    - DFAs has only one unique path that could accept x

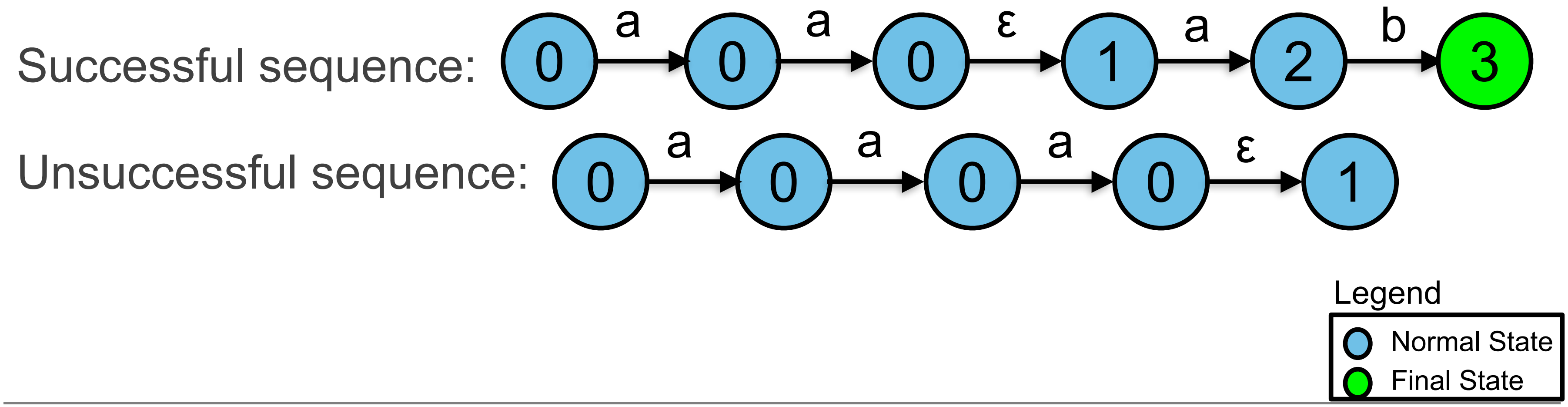
NFA Example

- The following NFA is for the regular expression: a*ab



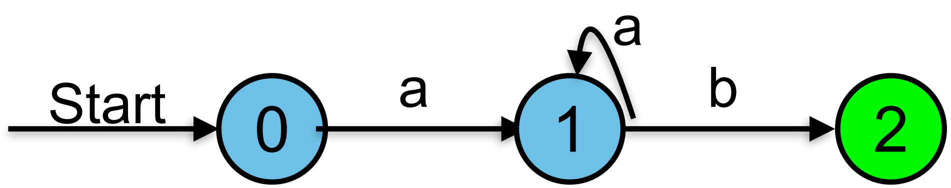
- There are many possible moves to accept a string for the regular expression. We only just need one sequence of moves

Input string: aaab



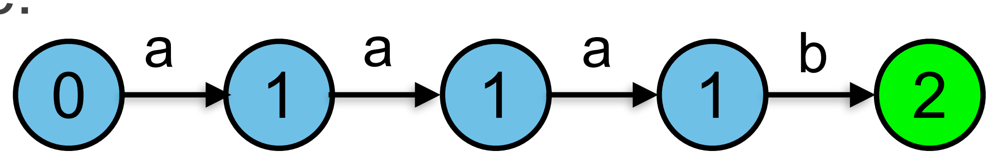
DFA Example

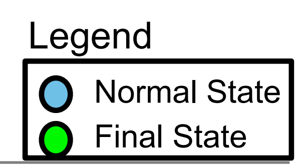
- The following DFA is for the regular expression: a*ab



Input string: aaab

Successful sequence:





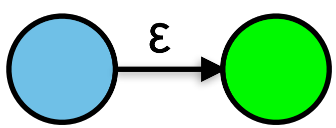
Automating Scanner Construction

- Goal: We need to covert our regular expressions that represent our tokens into finite automata so we can easily execute the scanner to generate the tokens.
- Steps to convert a lexical specification into code:
  1. Write down the RE for the input language
  2. Build a big NFA
  3. Build the DFA that simulates the NFA
  4. Systematically shrink the DFA (not this course :-( )
  5. Turn it into actual code

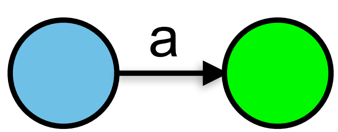
Regular Expression -> NFA

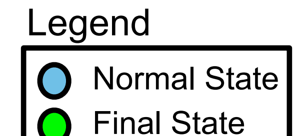
- Use Thompson construction rules to convert regular expressions into NFA form.
  - Always use unique names for all states
  - Always have at most one final state.
  - Combine your regular expressions with ε-moves

- Epsilon (ε)



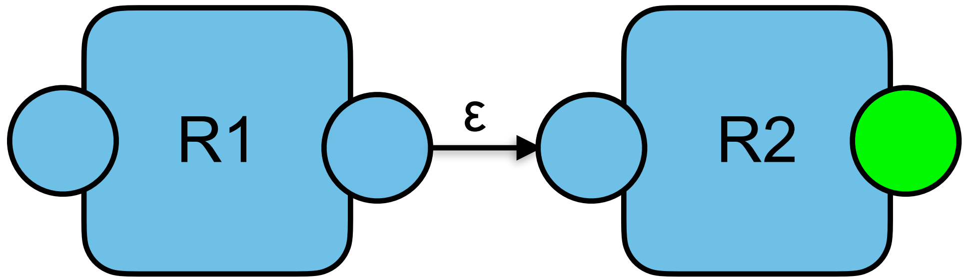
- Literal ‘a’ ($a \in \Sigma$)



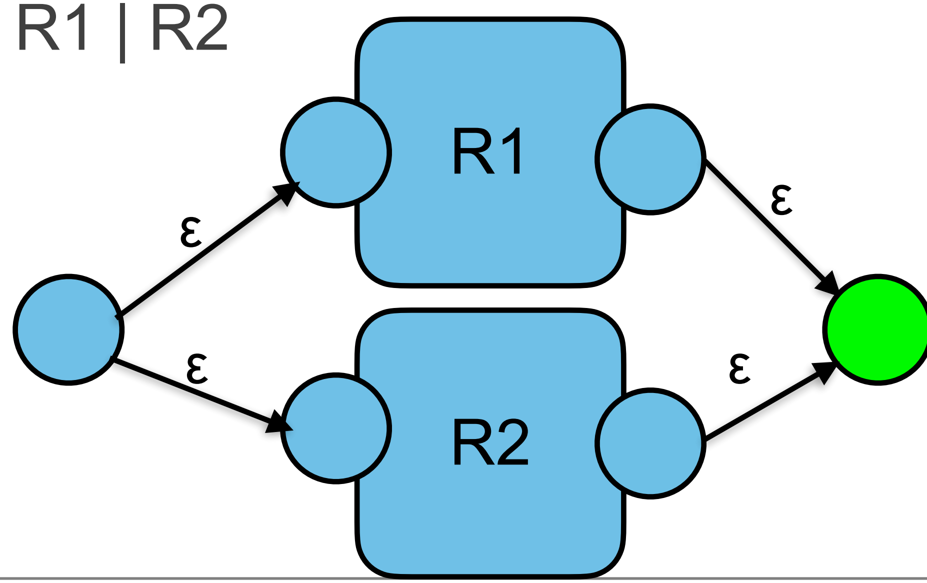


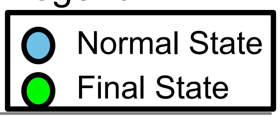
Regular Expression -> NFA

- Concatenation R1R2



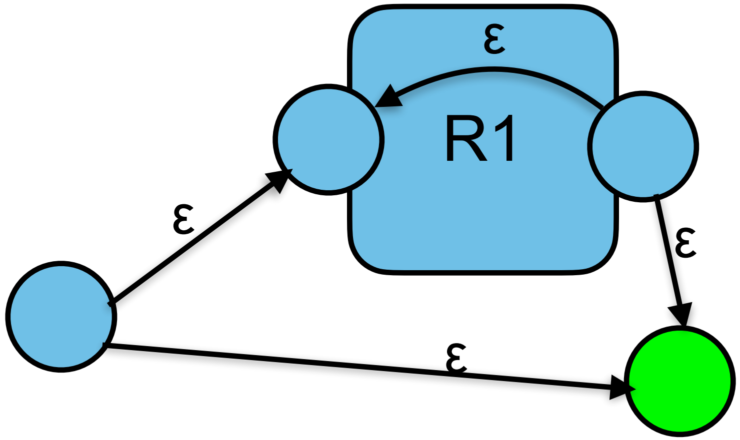
- Alternation R1 | R2

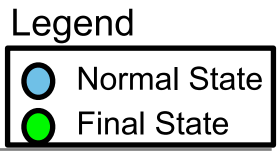




Regular Expression -> NFA

- Kleene star R*

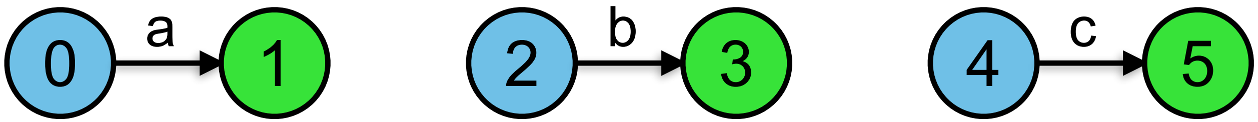




Regular Expression -> NFA

- Lets covert the regular expression: “a(b | c)*” to an NFA using Thompson construction rules

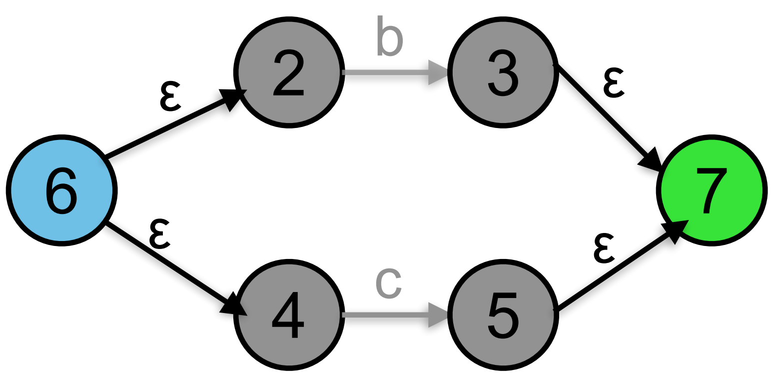
1. We will do the basic construction for the alphabet of the regular expression (i.e., literals “a”, “b”, and “c”)

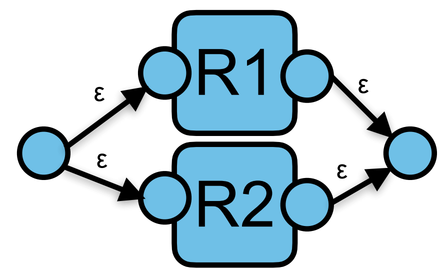


Regular Expression -> NFA

- Lets covert the regular expression: “a(b | c)*” to an NFA using Thompson construction rules

2. We will use the alteration rule to construct “b | c”, where R1 = “b” and R2 = “c”

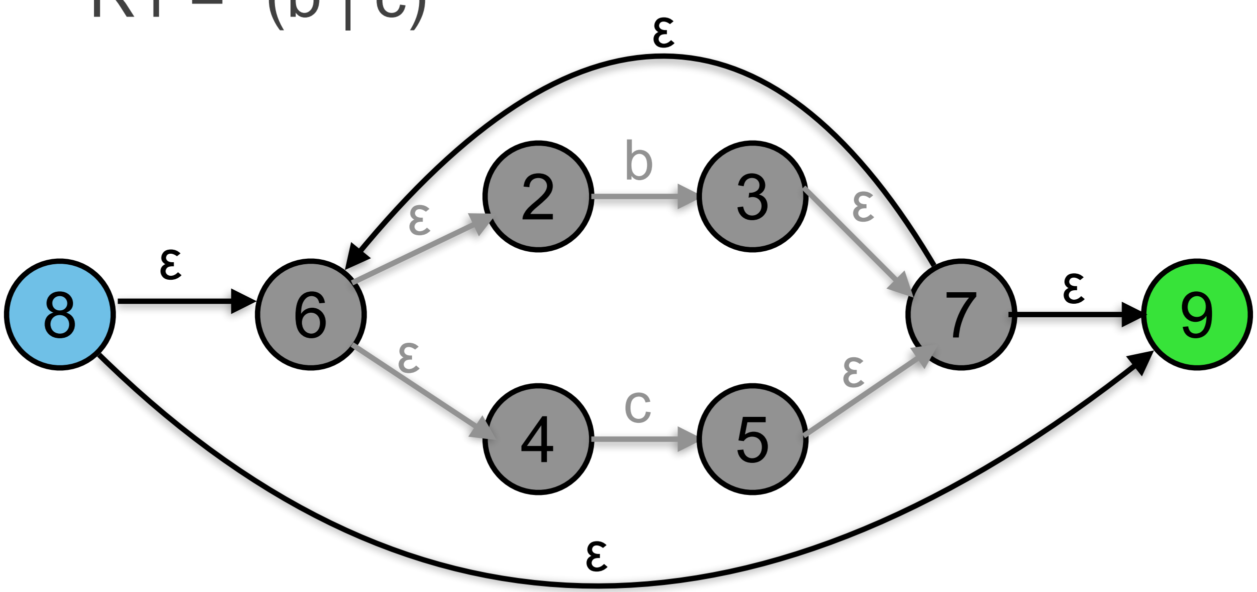


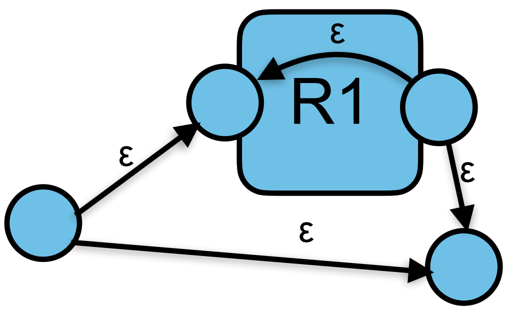


Regular Expression -> NFA

- Lets covert the regular expression: “a(b | c)*” to an NFA using Thompson construction rules

3. We will use the Kleene rule to construct “(b | c)*”, where R1 = “(b | c)*”

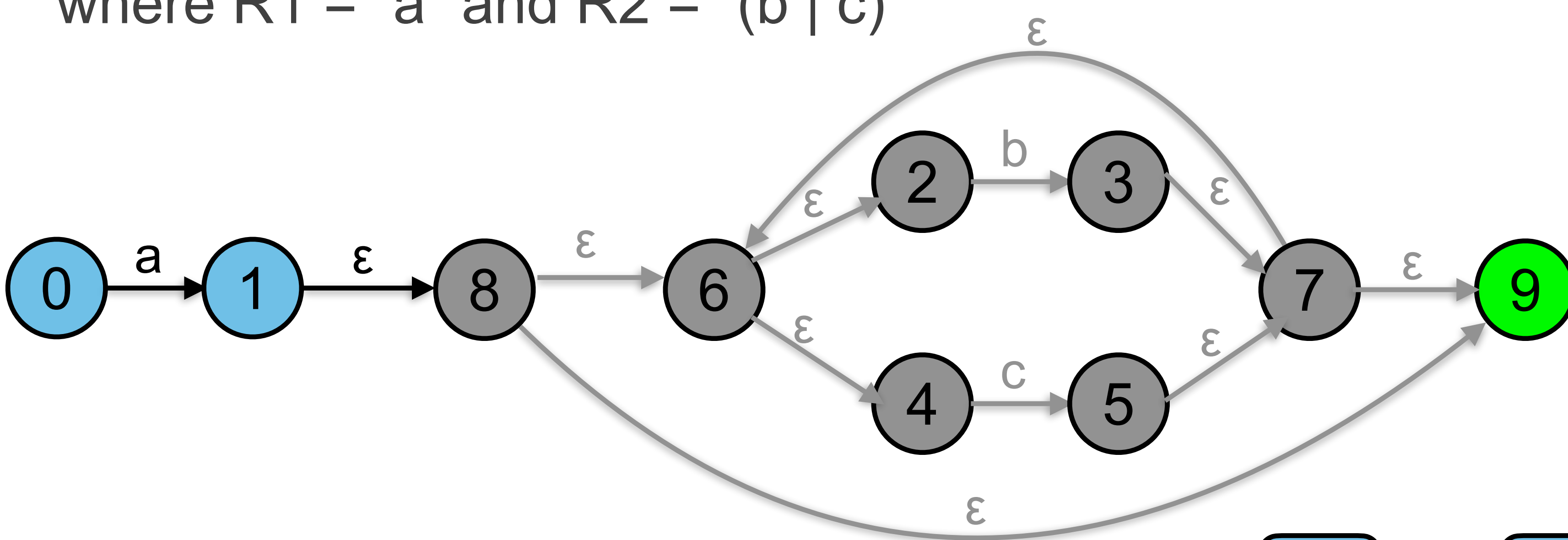


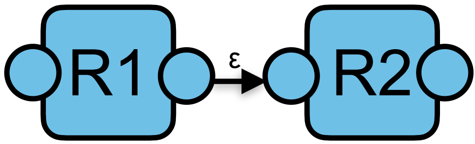


Regular Expression -> NFA

- Lets covert the regular expression: “a(b | c)*” to an NFA using Thompson construction rules

4. We will use concatenation rule to construct “a(b | c)*”, where R1 = “a” and R2 = “(b | c)*”





NFA →DFA with Subset Construction

The algorithm:

$s_0 \leftarrow \varepsilon$-closure($\{n_0\}$)
$S \leftarrow \{ s_0 \}$
$W \leftarrow \{ s_0 \}$
while ($W \neq \emptyset$)
select and remove s from W
for each $\alpha \in \Sigma$
$t \leftarrow \varepsilon$-closure(Move(s,α))
$T[s,\alpha] \leftarrow t$
if ($t \notin S$) then
add t to S
add t to W

Let's think about why this works

s_0 is a set of states

S & W are sets of sets of states

The algorithm halts:

1. S contains no duplicates (test before adding)
2. There is finite number of NFA states
3. while loop adds to S, but does not remove from S (monotone)

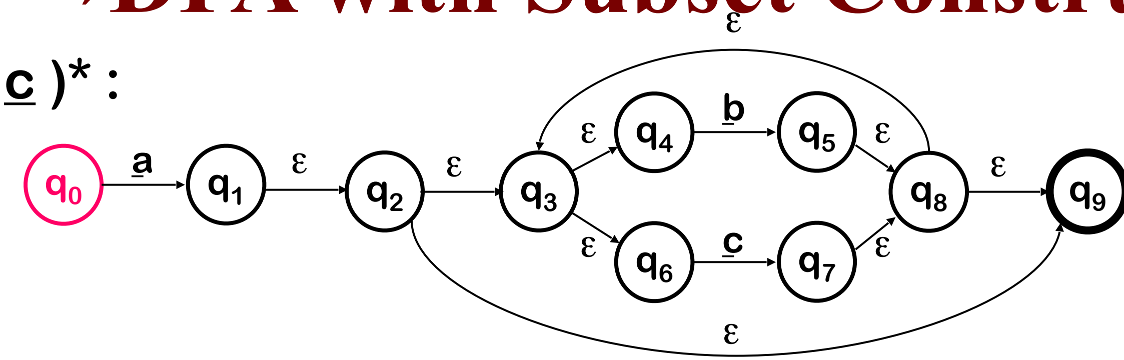
⇒ the loop halts

⇒ S and T form the DFA

- Two key functions
  - Move(si , a) is the set of states reachable from si by a
  - ε-closure(si) is the set of states reachable from si by ε

NFA →DFA with Subset Construction

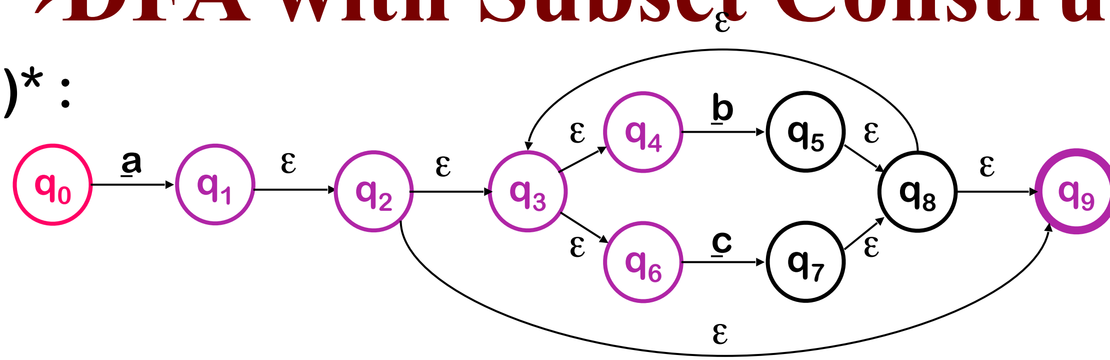
a (b | c)* :



States		ε-closure(Move(s,*))		
DFA	NFA	a	b	c
s_0	q_0			

NFA →DFA with Subset Construction

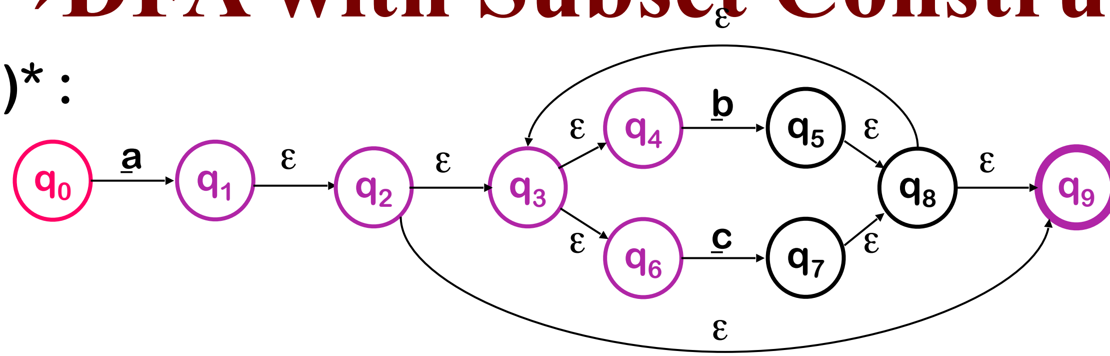
a (b | c)* :



States		ε-closure(Move(s,*))		
DFA	NFA	a	b	c
s_0	q_0	$q_1, q_2, q_3, q_4, q_6, q_9$		

NFA →DFA with Subset Construction

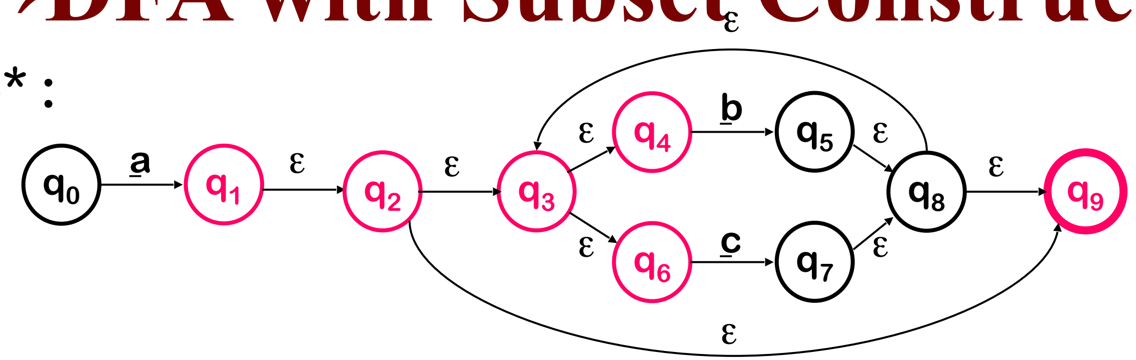
a (b | c)* :



States		ε-closure(Move(s,*))		
DFA	NFA	a	b	c
s_0	q_0	$q_1, q_2, q_3, q_4, q_6, q_9$	none	none

NFA →DFA with Subset Construction

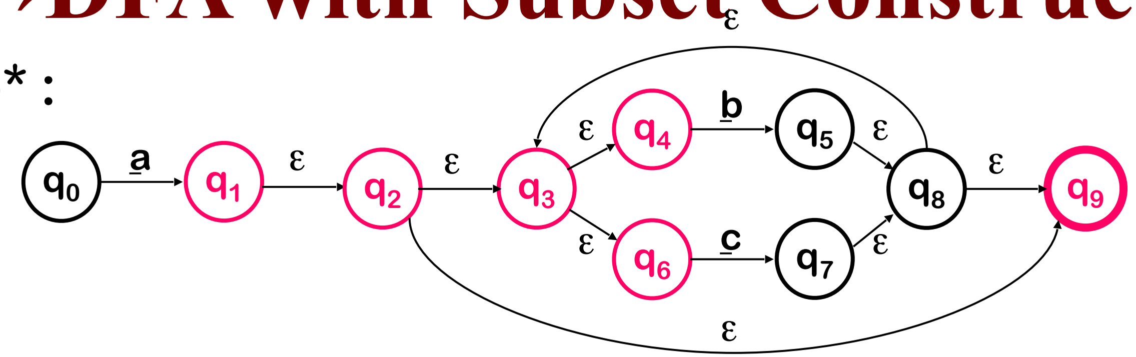
a (b | c)* :



States		ε-closure(Move(s,*))		
DFA	NFA	a	b	c
s_0	q_0	$q_1, q_2, q_3, q_4, q_6, q_9$	none	none
s_1	$q_1, q_2, q_3, q_4, q_6, q_9$			

NFA →DFA with Subset Construction

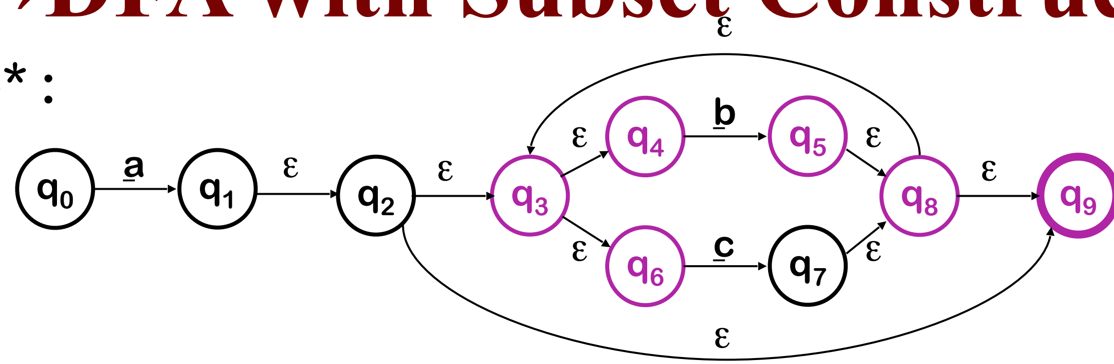
a (b | c)* :



States		ε-closure(Move(s,*))		
DFA	NFA	a	b	c
s_0	q_0	$q_1, q_2, q_3, q_4, q_6, q_9$	none	none
s_1	$q_1, q_2, q_3, q_4, q_6, q_9$	none		

NFA →DFA with Subset Construction

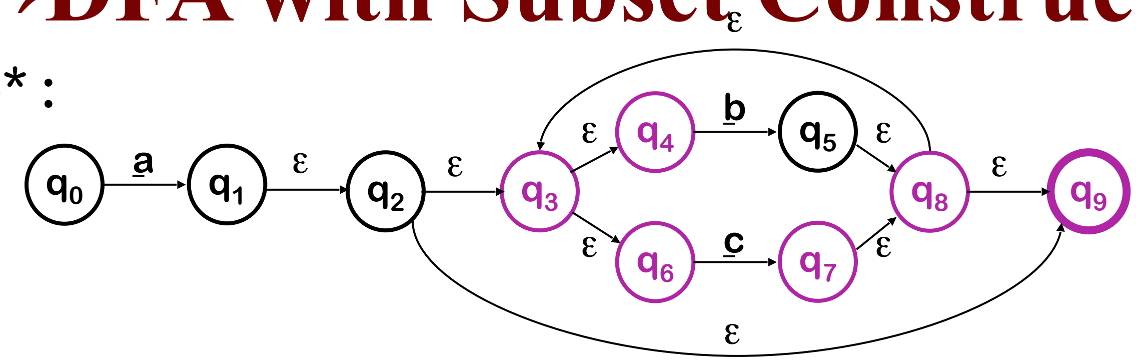
a (b | c)* :



States		ε-closure(Move(s,*))		
DFA	NFA	a	b	c
s_0	q_0	$q_1, q_2, q_3,$ q_4, q_6, q_9	none	none
s_1	$q_1, q_2, q_3,$ q_4, q_6, q_9	none	$q_5, q_8, q_9,$ q_3, q_4, q_6	

NFA →DFA with Subset Construction

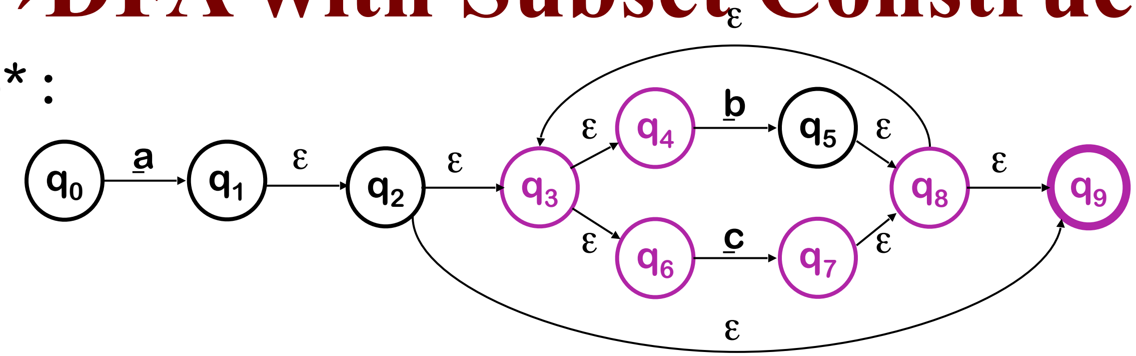
a (b | c)* :



States		ε-closure(Move(s,*))		
DFA	NFA	a	b	c
s_0	q_0	$q_1, q_2, q_3, q_4, q_6, q_9$	none	none
s_1	$q_1, q_2, q_3, q_4, q_6, q_9$	none	$q_5, q_8, q_9, q_3, q_4, q_6$	$q_7, q_8, q_9, q_3, q_4, q_6$

NFA →DFA with Subset Construction

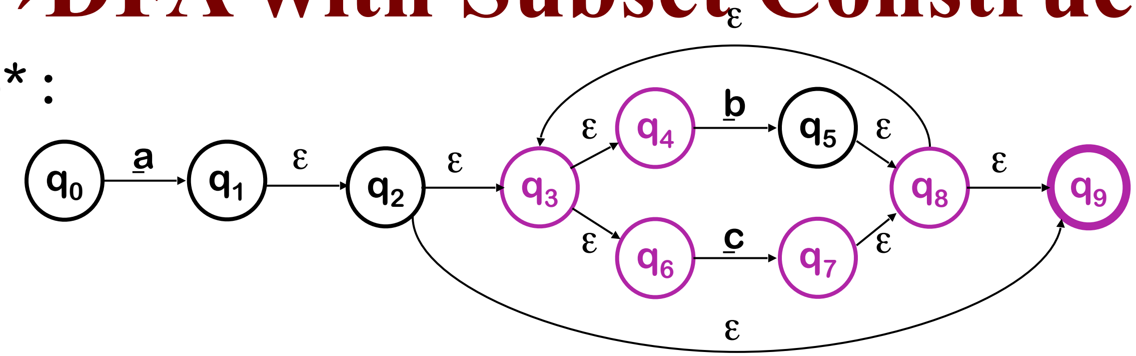
a (b | c)* :



States		ε-closure(Move(s,*))		
DFA	NFA	a	b	c
s_0	q_0	$q_1, q_2, q_3, q_4, q_6, q_9$	none	none
s_1	$q_1, q_2, q_3, q_4, q_6, q_9$	none	$q_5, q_8, q_9, q_3, q_4, q_6$	$q_7, q_8, q_9, q_3, q_4, q_6$
s_2	$q_5, q_8, q_9, q_3, q_4, q_6$			

NFA →DFA with Subset Construction

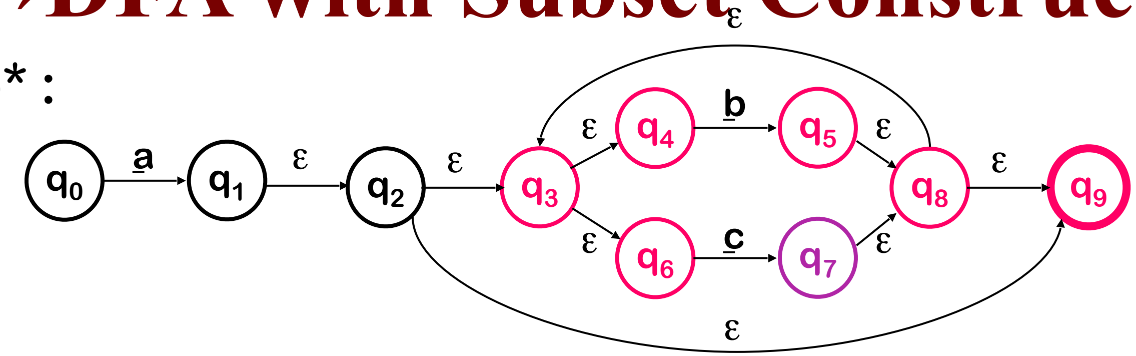
a (b | c)* :



States		ε-closure(Move(s,*))		
DFA	NFA	a	b	c
s_0	q_0	$q_1, q_2, q_3, q_4, q_6, q_9$	none	none
s_1	$q_1, q_2, q_3, q_4, q_6, q_9$	none	$q_5, q_8, q_9, q_3, q_4, q_6$	$q_7, q_8, q_9, q_3, q_4, q_6$
s_2	$q_5, q_8, q_9, q_3, q_4, q_6$			
s_3	$q_7, q_8, q_9, q_3, q_4, q_6$			

NFA →DFA with Subset Construction

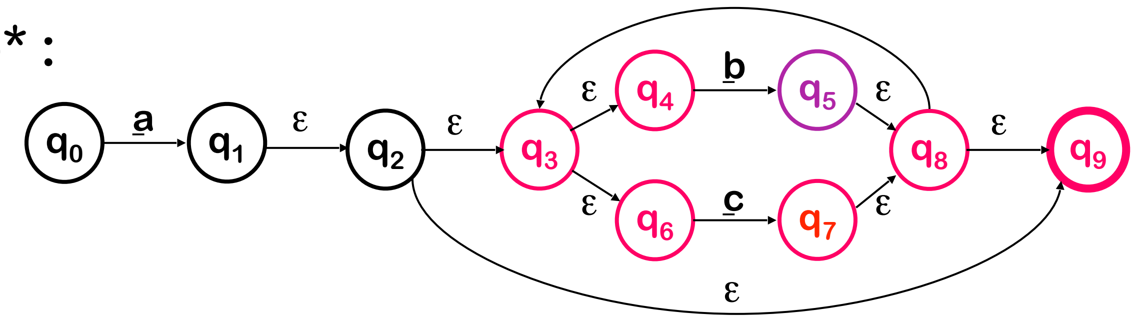
a (b | c)* :



States		ε-closure(Move(s,*))		
DFA	NFA	a	b	c
s_0	q_0	$q_1, q_2, q_3, q_4, q_6, q_9$	none	none
s_1	$q_1, q_2, q_3, q_4, q_6, q_9$	none	$q_5, q_8, q_9, q_3, q_4, q_6$	$q_7, q_8, q_9, q_3, q_4, q_6$
s_2	$q_5, q_8, q_9, q_3, q_4, q_6$	none	s_2	s_3
s_3	$q_7, q_8, q_9, q_3, q_4, q_6$			

NFA →DFA with Subset Construction

a (b | c)* :

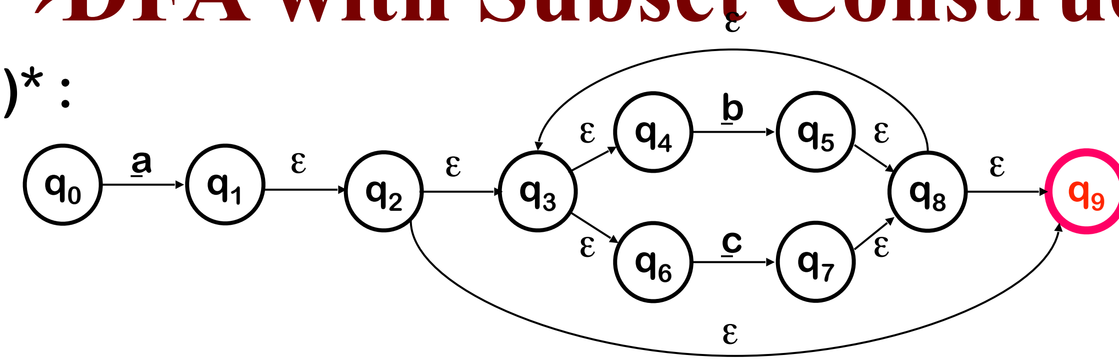


States		ε-closure(Move(s,*))		
DFA	NFA	a	b	c
s_0	q_0	$q_1, q_2, q_3, q_4, q_6, q_9$	none	none
s_1	$q_1, q_2, q_3, q_4, q_6, q_9$	none	$q_5, q_8, q_9, q_3, q_4, q_6$	$q_7, q_8, q_9, q_3, q_4, q_6$
s_2	$q_5, q_8, q_9, q_3, q_4, q_6$	none	s_2	s_3
s_3	$q_7, q_8, q_9, q_3, q_4, q_6$	none	s_2	s_3

THE UNIVERSITY OF CHICAGO | MASTERS PROGRAM IN COMPUTER SCIENCE

NFA →DFA with Subset Construction

a (b | c)* :

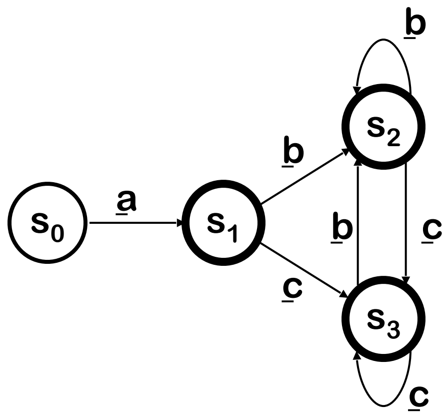


States		ε-closure(Move(s,*))		
DFA	NFA	a	b	c
s_0	q_0	$q_1, q_2, q_3, q_4, q_6, q_9$	none	none
s_1	$q_1, q_2, q_3, q_4, q_6, q_9$	none	$q_5, q_8, q_9, q_3, q_4, q_6$	$q_7, q_8, q_9, q_3, q_4, q_6$
s_2	$q_5, q_8, q_9, q_3, q_4, q_6$	none	s_2	s_3
s_3	$q_7, q_8, q_9, q_3, q_4, q_6$	none	s_2	s_3

Final states because of q_9

NFA →DFA with Subset Construction

- The DFA for a (b | c)*



	a	b	c
s_0	s_1	none	none
s_1	none	s_2	s_3
s_2	none	s_2	s_3
s_3	none	s_2	s_3

- Much smaller than the NFA (no ε-transitions)
- All transitions are deterministic
- Use same code skeleton as before

THE UNIVERSITY OF CHICAGO | MASTERS PROGRAM IN COMPUTER SCIENCE

Coding a Scanner: Table- Driven

- The common strategy is to simulate a DFA execution.
- One strategy is to implement a Table-Driven Scanner for DFA execution.
  - Make heavy use of indexing
  - index • Read the next character
  - index • Classify it
  - Find the next state
  - Branch back to the top

Note: There is more to this code. I'm just not showing it.

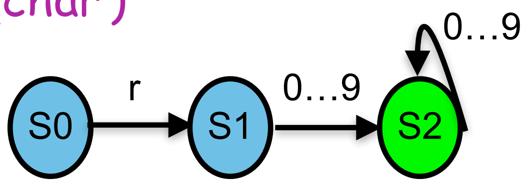
```
state ← s0 ;
while (state ≠ exit) do
   char ← NextChar( )
   cat ← CharCat(char )
   state ← δ(state,cat);
```

r	0, 1, 2, …, 9	EOF	Other
Register	Digit	Other	Other

CharCat(char)

State	Register	Digit	Other
S0	S1	Se	Se
S1	Se	S2	Se
S2	Se	Se	Se
Se	Se	Se	Se

δ(state,cat);



Coding a Scanner: Direct Coding

- Table-Drive strategy is not thee best the lookups into the various tables can be expensive.
- Alternative strategy: direct coding
  - Encode state in the program counter
    - Each state is a separate piece of code
  - Do transition tests locally and directly branch
  - Generate ugly, spaghetti-like code

Lexical Specification (cont.)

- Ambiguity: How do you break up text? Is the token stream 1 or 2?

```
elsex =45;
```

else	x	=	45	;

1

elsex	=	45	;

2

- Regular expressions are not enough to handle ambiguity.
- Most languages will choose the longest matching token
  - longest initial substring of the input that can match a regular expression is taken as next token
  - Ties in length are resolved by prioritizing the specification.
- Lexical specification = regular expressions + priorities + longest-matching token rule.

Coding a Scanner: Direct Coding

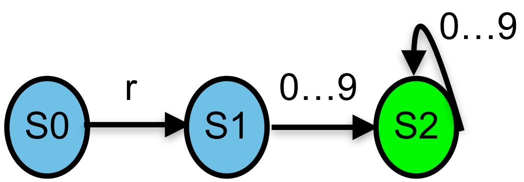
```
start: accept ← se
       lexeme ← ""
       count ← 0
       goto s0

s0:    char ← NextChar
       lexeme ← lexeme +  char
       count++
       if (char = 'r')
          then goto s1
          else goto sout

s1:    char ← NextChar
       lexeme ← lexeme +  char
       count++
       if ('0' ≤ char ≤ '9')
          then goto s2
          else goto sout

s2:    char ← NextChar
       lexeme ← lexeme +  char
       count ← 1
       accept ← s2
       if ('0' ≤ char ≤ '9')
          then goto s2
          else goto sout

sout: if (accept ≠ se)
       then begin
              for i ← 1 to count
                 RollBack()
              report success
           end
       else report failure
```



What About Hand-Coded Scanners?

Many (most?) modern compilers use hand-coded scanners

- Starting from a DFA simplifies design & understanding
- Avoiding straight-jacket of a tool allows flexibility
  - Computing the value of an integer
    - In LEX or FLEX, many folks use sscanf() & touch chars many times
    - Can use old assembly trick and compute value as it appears
  - Combine similar states
- Scanners are fun to write
  - Compact, comprehensible, easy to debug, …

Building Scanners Review

The point

- All this technology lets us automate scanner construction
- Implementer writes down the regular expressions
- Scanner generator builds NFA, DFA, minimal DFA, and then writes out the (table-driven or direct-coded) code
- This reliably produces fast, robust scanners

For most modern language features, this works

- You should think twice before introducing a feature that defeats a DFA-based scanner
- The ones we've seen (e.g., insignificant blanks, non-reserved keywords) have not proven particularly useful or long lasting