

Lexical Analysis

What is Lexical Analysis

□ What do we want to do?

Example:

```
if (i==j)
  z = 0;
else
  z = 1;
```

□ The input is just a string of characters

"if(i = j)\n\t\tz = 0; \telse\n\t\tz = 1; \n"

□ Goal: partition input string into substrings

➤ where these substrings are **tokens**

What is a Token ?

A syntactic category of entities over alphabet

➤ In English:

noun, verb, adjective, ...

➤ In a programming language:

identifier, integer, keyword, whitespace, ...

Tokens correspond to sets of strings

- Identifier: strings of letters and digits, starting with a letter
- Integer: a non-empty string of digits
- Keyword: “else”, “if”, “while”, ...
- Whitespace: a non-empty sequence of blanks, newlines, and tabs

What are Tokens For ?

- ❑ Classify program substrings according to role
- ❑ Output of lexical analysis is a stream of tokens
- ❑ Tokens are the input to the Parser
 - Parser relies on token distinctions
a keyword is treated differently than an identifier

Lexical Analyzer in FORTRAN

- ❑ FORTRAN compilation rule: **whitespace is insignificant**
 - rule was motivated from the inaccuracy of card punching by operators
- ❑ Consider
 - DO 5I=1,25
 - DO 5I=1.25
- ❑ We have
 - The first: a loop iterates from 1 to 25 with step 5
 - The second: an assignment
- ❑ Reading left-to-right, cannot tell if DO5I is a variable or DO statement; Have to continue until “,” or “.” is reached.

Lesson Learned

Two important observations:

- The goal is to partition the string. This is implemented by reading left-to-right, recognizing one token at a time.
- “lookahead” may be required to decide where one token ends and the next one begins.

Lexical Analysis in C++

- Unfortunately, the problems continue today

- C++ template syntax

`FOO<Bar>`

- C++ stream syntax

`cin> >var`

- Now, the problem

`FOO<Bar<Bazz> >`

Regular Languages

- ❑ Unfortunately, the problems continue today
- ❑ To describe tokens, we adopt **Regular Languages** formalism
 - Simple and useful theory
 - Easy to understand
 - Efficient implementations

Languages

Definition

Let Σ be a set of characters, a **language** over Σ is a set of strings of the characters drawn from Σ

Examples of Languages



Alphabet = English characters
Language = English sentences



Alphabet = ASCII
Language = C programs

- Not every string on English characters is an English sentence
- Note all ASCII strings are valid C programs

Notation

- ❑ Languages are **sets of strings**
- ❑ Need some notation for specifying which set we want to designate a language
 - Regular languages are those with some special properties
 - The standard notation for regular language is **regular expression**

Atomic Regular Expressions

- Single character denotes a set of one string
 $'c' = \{ "c" \}$
- Epsilon* or ϵ character denotes a set of non-zero length string
 $\epsilon = \{ "" \}$
- Empty set is $\{ \} = \phi$, not the same as ϵ
 $\text{size}(\phi) = 0$
 $\text{size}(\epsilon) = 1$
 $\text{length}(\epsilon) = 0$

Compound Regular Expressions

□ Union: if A and B are REs, then
 $A + B = \{ s \mid s \in A \text{ or } s \in B \}$

□ Concatenation of sets/strings
 $AB = \{ ab \mid a \in A \text{ and } b \in B \}$

□ Iteration (Kleene closure)
 $A^* = \bigcup_{i \geq 0} A^i \quad \text{where } A^i = A \dots A \text{ (} i \text{ times)}$

in particular

$$A^* = \{ \varepsilon \} + A + AA + AAA + \dots$$

$$A^+ = A + AA + AAA + \dots = A A^*$$

Regular Expressions

Definition

The **regular expressions (REs)** over Σ are the smallest set of expressions including


- ε
- 'c' where $c \in \Sigma$
- $A + B$ where A, B are **RE** over Σ
- AB where A, B are **RE** over Σ
- A^* where A is a **RE** over Σ

Notation

 This notation means

- $L(\varepsilon) = \{ \text{""} \}$
- $L('c') = \{ \text{"c"} \}$
- $L(A+B) = L(A) \cup L(B)$
- $L(AB) = \{ ab \mid a \in L(A) \text{ and } b \in L(B) \}$
- $L(A^*) = \bigcup_{i \geq 0} L(A^i)$

Examples

 Keywords: “else” or “if” or “while” or ...

Examples

□ Keywords: “else” or “if” or “while” or ...

➤ ‘else’ + ‘if’ + ‘while’ + ...

➤ ‘else’ abbreviates

‘e’ (concatenate) ‘l’ (concatenate) ‘s’ (concatenate) ‘e’

Examples

Keywords: “else” or “if” or “while” or ...

➤ ‘else’ + ‘if’ + ‘while’ + ...

➤ ‘else’ abbreviates

‘e’ (concatenate) ‘l’ (concatenate) ‘s’ (concatenate) ‘e’

➤ keywords = { ‘else’, ‘if’, ‘then’, ‘while’, ... }

Examples

Keywords: “else” or “if” or “while” or ...

- ‘else’ + ‘if’ + ‘while’ + ...
- ‘else’ abbreviates
‘e’ (concatenate) ‘l’ (concatenate) ‘s’ (concatenate) ‘e’
- keywords = { ‘else’, ‘if’, ‘then’, ‘while’, ... }

Integer

- digit = ‘0’ + ‘1’ + ‘2’ + ‘3’ + ‘4’ + ‘5’ + ‘6’ + ‘7’ + ‘8’ + ‘9’
- integer = digit digit*

Examples

Keywords: “else” or “if” or “while” or ...

- ‘else’ + ‘if’ + ‘while’ + ...
- ‘else’ abbreviates
‘e’ (concatenate) ‘l’ (concatenate) ‘s’ (concatenate) ‘e’
- keywords = { ‘else’, ‘if’, ‘then’, ‘while’, ... }

Integer

- digit = ‘0’ + ‘1’ + ‘2’ + ‘3’ + ‘4’ + ‘5’ + ‘6’ + ‘7’ + ‘8’ + ‘9’
- integer = digit digit*
 - **Q:** is ‘000’ an integer?

Examples

Keywords: “else” or “if” or “while” or ...

- ‘else’ + ‘if’ + ‘while’ + ...
- ‘else’ abbreviates
‘e’ (concatenate) ‘l’ (concatenate) ‘s’ (concatenate) ‘e’
- keywords = { ‘else’, ‘if’, ‘then’, ‘while’, ... }

Integer

- digit = ‘0’ + ‘1’ + ‘2’ + ‘3’ + ‘4’ + ‘5’ + ‘6’ + ‘7’ + ‘8’ + ‘9’
- integer = digit digit*
 - **Q:** is ‘000’ an integer?
 - **Q:** how to define another integer RE that excludes sequences with leading 0s?

More Examples

❏ Identifier: strings of letters or digits, starting with a letter

- letter = 'A' + ... + 'Z' + 'a' + ... + 'z'
- Identifier = letter (letter + digit)*

❏ Whitespace: a non-empty sequence of blanks, newlines and tabs

- whitespace = (' ' + '\n' + '\t') +

More Examples

❏ Identifier: strings of letters or digits, starting with a letter

- letter = 'A' + ... + 'Z' + 'a' + ... + 'z'
- Identifier = letter (letter + digit)*
 - **Q:** is (letter* + digit*) the same?

❏ Whitespace: a non-empty sequence of blanks, newlines and tabs

- whitespace = (' ' + '\n' + '\t') +

More Examples

☐ Phones number: consider (412) 624-0000

- $\Sigma = \text{digit} \cup \{ -, (,) \}$
- $\text{area} = \text{digit}^3$
- $\text{exchange} = \text{digit}^3$
- $\text{phone} = \text{digit}^4$
- $\text{phoneNumber} = '(' \text{ area } ')' \text{ exchange } '-' \text{ phone}$

☐ Email address: student @ pitt.edu

- $\Sigma = \text{letter} \cup \{ ., @ \}$
- $\text{name} = \text{letter}^+$
- $\text{emailAddress} = \text{name } '@' \text{ name } '.' \text{ name}$

More Examples in Practice

RE used in languages

- By itself, it is a string, but semantically gets interpreted as a RE

- RE in PERL,
if (\$str =~ /(\\d+)/) ...

here,

- \$str denotes a variable
- =~ denotes RE matching
- (\\d+) defines a RE pattern

- RE in C#,
Match m = Regex.Match("abrabceaab", "(a|b|r)+");

Some Common REs in Programming Languages

	Meaning		Meaning		Meaning
<code>\d</code>	Digits	<code>\w</code>	Any word char	<code>\s</code>	Space char
<code>\D</code>	Non-digits	<code>\W</code>	Non-word char	<code>\S</code>	Non-space char
<code>[a-f]</code>	Char range	<code>[^a-f]</code>	Exclude range	<code>^</code>	Matching string start
<code>?</code>	Optional	<code>{n,m}</code>	Appear n-m times	<code>\$</code>	Matching string end
<code>.</code>	Any char	<code>(...)</code>	Capture matches	<code>\(,\{</code>	Matching (, { ...
<code>\.</code>	Matching “.”	<code>+</code>	Appear ≥ 1 times	<code>*</code>	Appear 0 or many times

Implementation of Lexical Analysis

Implementation of Lexical Analysis

- We have learnt the formalism for lexical analysis
 - Regular expression (RE)

Implementation of Lexical Analysis

- We have learnt the formalism for lexical analysis
 - Regular expression (RE)
- How to actually get the lexical analyzer?

Implementation of Lexical Analysis

- ❑ We have learnt the formalism for lexical analysis
 - Regular expression (RE)

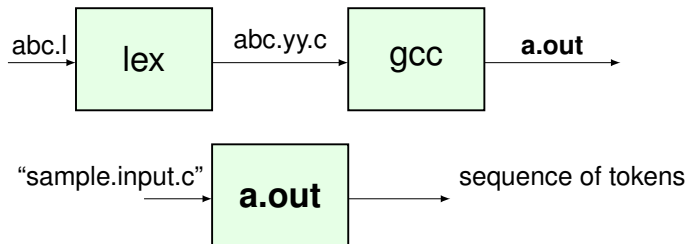
- ❑ How to actually get the lexical analyzer?
 - **Solution 1:** to implement using a tool — Lex (for C), Flex (for C++), Jlex (for java)
 - Programmer specifies the interesting tokens using REs
 - The tool generates the source code from the given REs

Implementation of Lexical Analysis

- ❑ We have learnt the formalism for lexical analysis
 - Regular expression (RE)

- ❑ How to actually get the lexical analyzer?
 - **Solution 1:** to implement using a tool — Lex (for C), Flex (for C++), Jlex (for java)
 - Programmer specifies the interesting tokens using REs
 - The tool generates the source code from the given REs
 - **Solution 2:** to write the code starting from scratch
 - This is also the code that the tool generates
 - The code is table-driven and is based on finite state automaton

Lex: a Tool for Lexical Analysis



- ❑ Big difference from your previous coding experience
 - Writing REs instead of the code itself
 - Writing actions associated with each RE
- ❑ A specification file has well-defined structures
- ❑ The detailed implementation will be discussed later

Lex Specifications

```
%{ /* include, extern, etc. */  
extern int yytext, yylineno;  
#include "token.h"  
%}  
/* declarations : declare variables, constants & regular definitions, */  
digit      [0-9]  
number     [0-9]+  
%%  
/* transition rules: regular expressions and actions */  
/* R1 action where actions are program fragments written in C. */  
number     { printf("Token: int const %s and %d", yytext, yyline); }  
%%  
/* auxiliary procedures */  
myTableInsert()  
{ ... }
```

Implementation Notes

- ❑ Write regular expressions for all/some of tokens
- ❑ Comments: keep track of nesting level
- ❑ String table written for you
- ❑ *yyline*, *yycolumn* maintained by yourself
- ❑ *yytext*, *yylen* maintained by lex
- ❑ Special characters
 - `'\n'` — newline
 - `'\t'` — tab
 - `'\''` — single quote
 - `'\\'` — backslash

Discussion of RE and Lexical Analysis

- We use RE to assist lexical analysis
- Regular Expressions describe many useful languages
- Regular Expressions is a language specification
 - An implementation is still needed
- The problem we face is

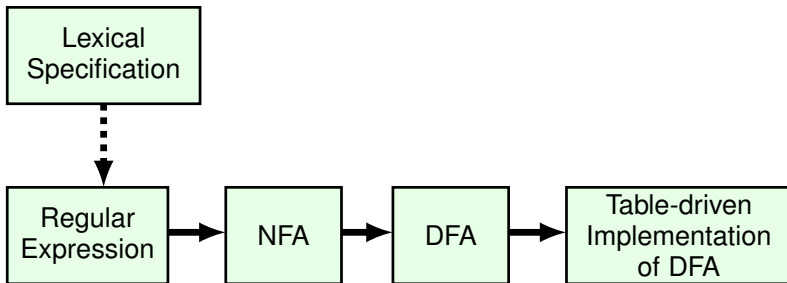
Given a string **s** and a regular expression **RE**, is

$$\mathbf{s} \in \mathbf{L(RE)} ?$$

Implementing Lexical Analysis with Finite Automata

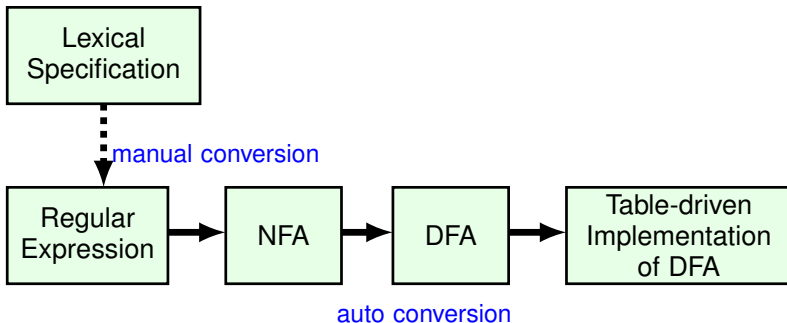
An Overview of RE to FA

Our implementation sketch



An Overview of RE to FA

Our implementation sketch



Implementation Outline

- ❏ RE \Rightarrow NFA \Rightarrow DFA \Rightarrow Table-driven Implementation
 - Specifying lexical structure using regular expressions
 - Finite automata
 - Deterministic Finite Automata (DFAs)
 - Non-deterministic Finite Automata (NFAs)
 - Table implementations

Notations

- In the following discussion, we use some alternative notations

Union: $A \mid B \equiv A + B$

Option: $A \varepsilon \equiv A$

Range: $\text{'a' + 'b' + ... + 'z'} \equiv [a-z]$

Excluded range:
complement of $[a-z] \equiv [\hat{a-z}]$

Finite Automata

□ A finite automata consists of 5 components
 $(\Sigma, S, n, F, \delta)$

- (1). An input alphabet Σ
- (2). A set of states S
- (3). A start state $n \in S$
- (4). A set of accepting states $F \subseteq S$
- (5). A set of transitions $\delta: S_a \xrightarrow{\text{input}} S_b$

□ For lexical analysis

- Specification — Regular expression
- Implementation — Finite automata

More About Transition

Transition $\delta: S_a \xrightarrow{\text{input}} S_b$

read as

in state S1 on input “a” go to state S2

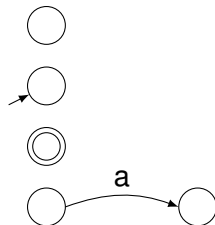
At the end of input (or no transition possible), if current state X

- $X \in \text{accepting set } F$, then \Rightarrow **accept**
- otherwise, \Rightarrow **reject**

State Graph

- Sometimes we use **state graph** to represent a FA
- A **state graph** includes

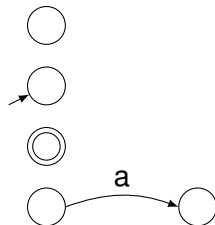
- A set of states
- A start state
- A set of accepting states
- A set of transition



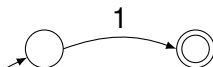
State Graph

- Sometimes we use **state graph** to represent a FA
- A **state graph** includes

- A set of states
- A start state
- A set of accepting states
- A set of transition

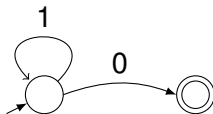


- Example: a finite state automata that accepts only "1"



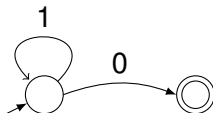
More Examples

- A finite automata accepting any number of **1**s followed by a single **0**. Here we have Alphabet = $\{0,1\}$



More Examples

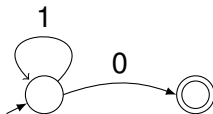
- A finite automata accepting any number of **1**s followed by a single **0**. Here we have Alphabet = $\{0,1\}$



- Example: What language does the following state graph recognize? Here we have Alphabet = $\{0,1\}$

More Examples

- A finite automata accepting any number of **1**s followed by a single **0**. Here we have Alphabet = {0,1}



- Example: What language does the following state graph recognize? Here we have Alphabet = {0,1}

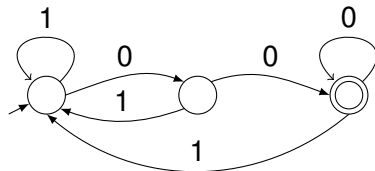


Table Implementation of a DFA

Given the state graph of a DFA,

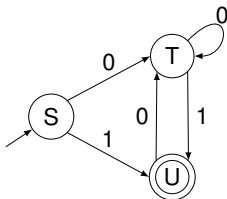
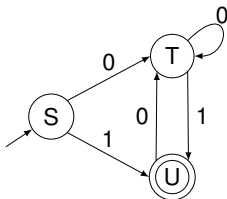


Table Implementation of a DFA

Given the state graph of a DFA,

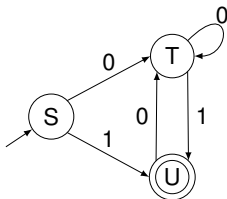


→ input characters

state ↓		0	1
	S		
	T		
	U		

Table Implementation of a DFA

Given the state graph of a DFA,

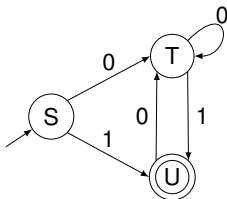


→ input characters

state ↓

	0	1
S	T	U
T	T	U
U	T	x

Given the state graph of a DFA,



→ input characters

state ↓

	0	1
S	T	U
T	T	U
U	T	x

```
DFA() {
    state = "S";
    while (!done) {
        ch = fetch_input();
        state = Table[state][ch];
        if (state == "X")
            perror("error");
    }
    if (state ∈ F)
        printf("accept");
    else
        printf("reject");
}
```

Discussion

- ❑ Each RE has a different DFA / state graph
- ❑ For different REs,
 - their tables are different
 - their DFA recognition code is the same

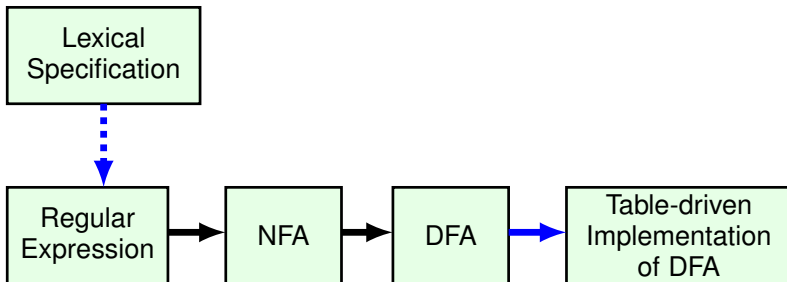
Discussion

- ❑ Each RE has a different DFA / state graph
- ❑ For different REs,
 - their tables are different
 - their DFA recognition code is the same
- ❑ Revisit our implementation outline

RE \Rightarrow NFA \Rightarrow **DFA** \Rightarrow **Table-driven Implementation**

From RE to FA

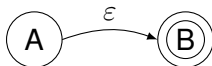
Our implementation sketch



Epsilon Moves

□ Another kind of transition: ε -moves

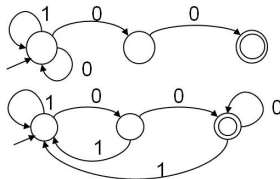
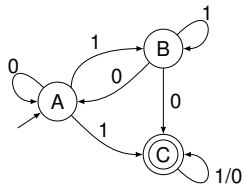
- Machine can move from state A to state B without reading any input



Deterministic and Nondeterministic Automata

- ❑ Deterministic Finite Automata (DFA)
 - One transition per input per state
 - No ϵ -moves
- ❑ Non-deterministic Finite Automata (NFA)
 - Can have multiple transitions for one input in a given state
 - Can have ϵ -moves
- ❑ Finite automata have finite memory
 - Need only to encode the current state

Examples

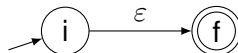


Converting RE to NFA

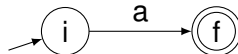
McNaughton-Yamada-Thompson Algorithm

Step 1: processing atomic REs

➤ ϵ expression



➤ single character RE a



Converting RE to NFA (cont.)

Step 2: processing compound REs

➤ $r = s \mid t$

➤ $r = s t$

➤ $r = s^*$

Converting RE to NFA (cont.)

Step 2: processing compound REs

$$\triangleright r = s \mid t$$

$$\triangleright r = s t$$

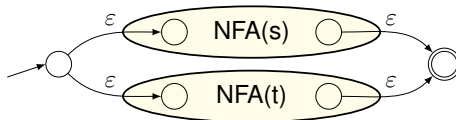
$$\triangleright r = s^*$$



Converting RE to NFA (cont.)

Step 2: processing compound REs

➤ $r = s \mid t$



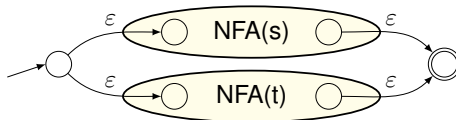
➤ $r = s t$

➤ $r = s^*$

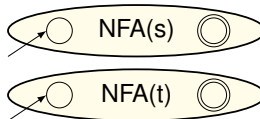
Converting RE to NFA (cont.)

Step 2: processing compound REs

➤ $r = s \mid t$



➤ $r = s t$

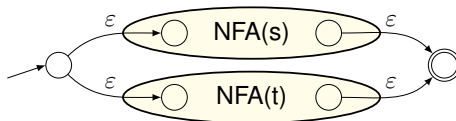


➤ $r = s^*$

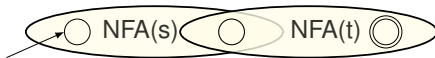
Converting RE to NFA (cont.)

Step 2: processing compound REs

➤ $r = s \mid t$



➤ $r = s t$

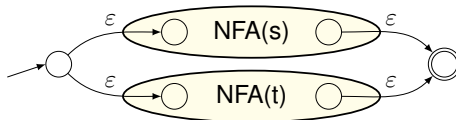


➤ $r = s^*$

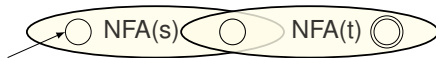
Converting RE to NFA (cont.)

Step 2: processing compound REs

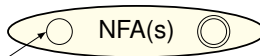
➤ $r = s \mid t$



➤ $r = s t$



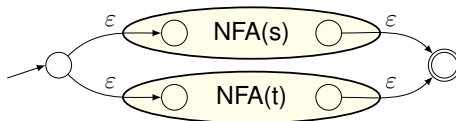
➤ $r = s^*$



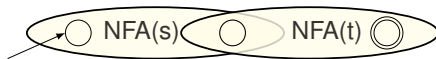
Converting RE to NFA (cont.)

Step 2: processing compound REs

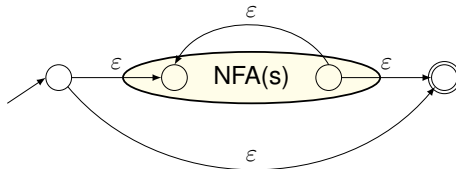
➤ $r = s \mid t$




➤ $r = s t$



➤ $r = s^*$

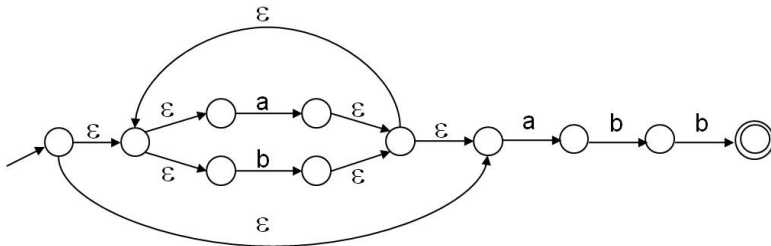


In-class Practice

 Convert “**(a|b)* a b b**” to NFA

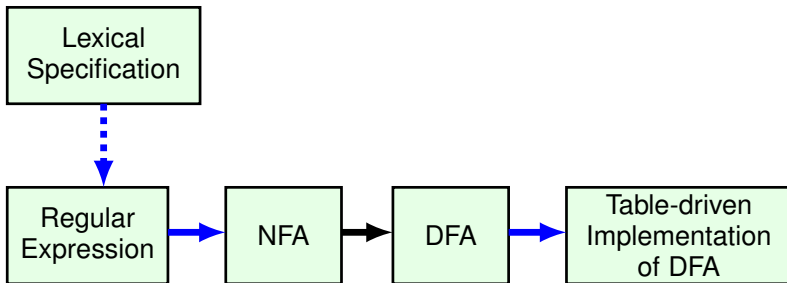
In-class Practice

Convert “**(a|b)* a b b**” to NFA



From RE to FA

Our implementation sketch



Execution of Finite Automata

- ❏ A DFA can take only one path through the state graph
 - Completely determined by input
- ❏ A NFA can take
 - Whether to make ϵ -moves
 - Which of multiple transitions for a single input to take
 - Acceptance of NFAs
 - An NFA can get into multiple states
 - **Rule:** the NFA accepts it if can get in a final state

Execution of Finite Automata

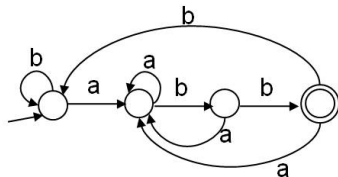
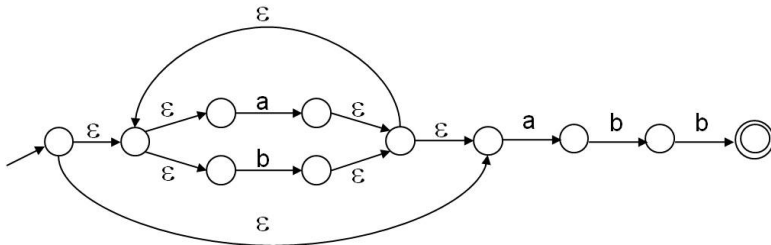
- ❏ A DFA can take only one path through the state graph
 - Completely determined by input
- ❏ A NFA can take
 - Whether to make ϵ -moves
 - Which of multiple transitions for a single input to take
 - Acceptance of NFAs
 - An NFA can get into multiple states
 - **Rule:** the NFA accepts it if can get in a final state
- ❏ **Question:** which one is more powerful?

Comparing NFA and DFA

- ❑ **Theorem:** NFAs and DFAs recognize the same set of languages
- ❑ Both recognize regular languages
- ❑ DFAs are faster to execute
 - There are no choices to consider
- ❑ For a given language, NFA can be simpler than DFA
- ❑ DFA can be exponentially larger than NFA
 - Example: DFA and NFA that accept “**(a|b)* a b b**”

NFA and DFA

Both accept “**(a|b)* a b b**”



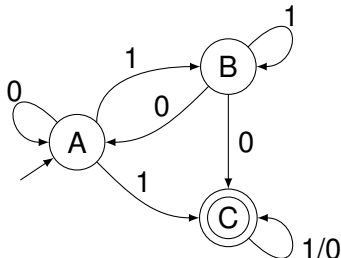
How to Convert NFA to DFA

- Basic idea: Given a NFA, simulate its execution using a DFA
 - At step n , the NFA may be in any of multiple possible states
- The new DFA is constructed as follows,
 - A state of DFA \equiv a non-empty subset of states of the NFA
 - Start state \equiv the set of NFA states reachable through ε -moves from NFA start state
 - A transition $S_a \xrightarrow{c} S_b$ is added **iff**

 S_b is the set of NFA states reachable from any state in S_a after seeing the input c , considering ε -moves as well

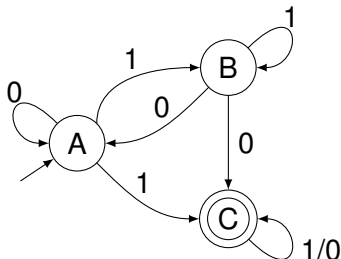
Example NFA to DFA

What is the Equivalent DFA ?



Example NFA to DFA

What is the Equivalent DFA ?

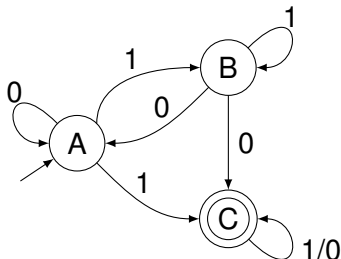


state ↓ → input characters

	0	1
A		
B		
C		

Example NFA to DFA

What is the Equivalent DFA ?

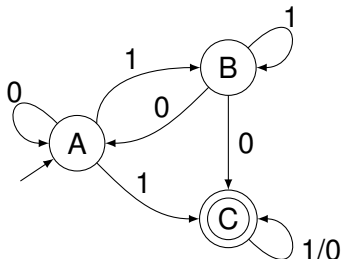


state ↓ → input characters

	0	1
A	A	BC
B	AC	B
C	C	C

Example NFA to DFA

What is the Equivalent DFA ?

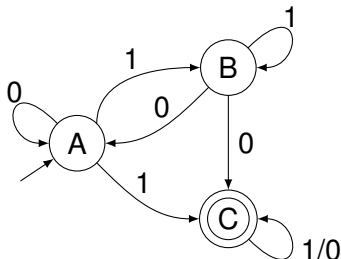


state ↓ → input characters

	0	1
A	A	BC
B	AC	B
C	C	C

Example NFA to DFA

What is the Equivalent DFA ?

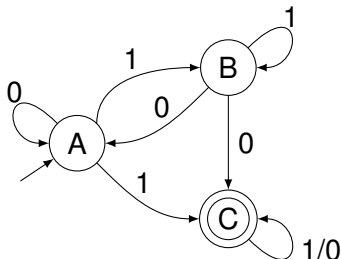


state ↓ → input characters

	0	1
A	A	BC
B	AC	B
C	C	C
AC		
BC		

Example NFA to DFA

What is the Equivalent DFA ?

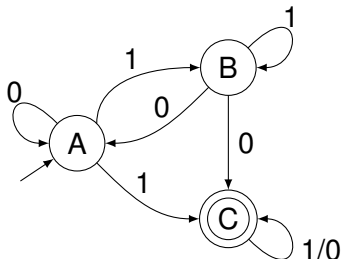


state ↓ → input characters

	0	1
A	A	BC
B	AC	B
C	C	C
AC	AC	BC
BC	AC	BC

Example NFA to DFA

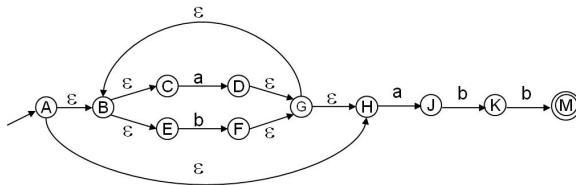
What is the Equivalent DFA ?



state ↓ → input characters

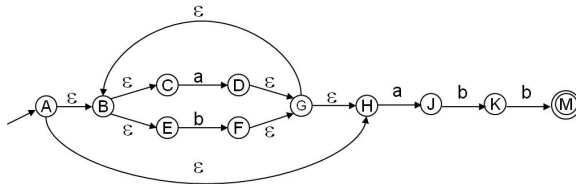
	0	1
A	A	BC
B	AC	B
C	C	C
AC	AC	BC
BC	AC	BC
AB	x	x
ABC	x	x

Algorithm Illustrated: Converting NFA to DFA



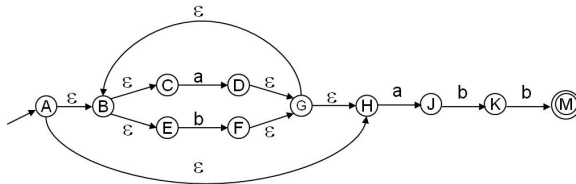
	ϵ	a	b
A			
B			
C			
D			
E			
F			
G			
H			
J			
K			
M			

Step 1: Construct the Table



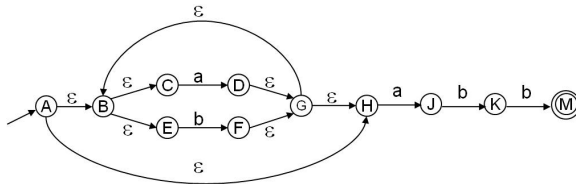
	ϵ	a	b
A	BH		
B	CE		
C		D	
D	G		
E			F
F	G		
G	BH		
H		J	
J			K
K			M
M			

Step 2: Construct ϵ -closure



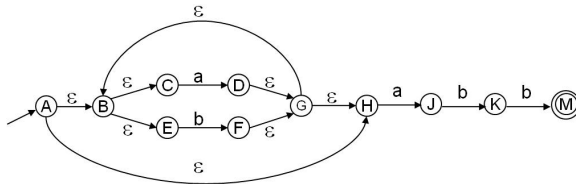
	ϵ	a	b
A	BHCE		
B	CE		
C		D	
D	GBHCE		
E			F
F	GBHCE		
G	BHCE		
H		J	
J			K
K			M
M			

Step 3: Update Other Columns



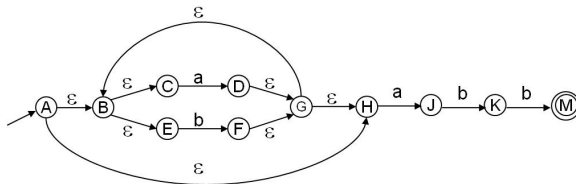
	ϵ	a	b
A	BHCE	DJ	F
B	CE	D	F
C		D	
D	GBHCE	DJ	F
E			F
F	GBHCE	DJ	F
G	BHCE	DJ	F
H		J	
J			K
K			M
M			

Step 4: Construct a New Table



	ϵ	a	b
A	BHCE	DJ	F
B	CE	D	F
C		D	
D	GBHCE	DJ	F
E			F
F	GBHCE	DJ	F
G	BHCE	DJ	F
H		J	
J			K
K			M
M			

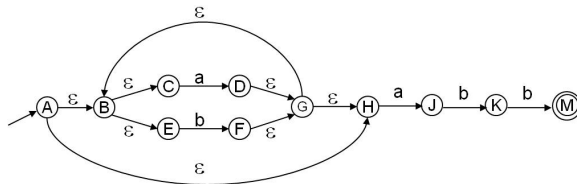
Step 4: Construct a New Table



	ϵ	a	b
A	BHCE	DJ	F
B	CE	D	F
C		D	
D	GBHCE	DJ	F
E			F
F	GBHCE	DJ	F
G	BHCE	DJ	F
H		J	
J			K
K			M
M			

	a	b
ABHCE	DJ	F

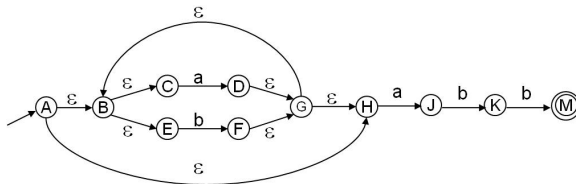
Step 4: Construct a New Table



	ϵ	a	b
A	BHCE	DJ	F
B	CE	D	F
C		D	
D	GBHCE	DJ	F
E			F
F	GBHCE	DJ	F
G	BHCE	DJ	F
H		J	
J			K
K			M
M			

	a	b
ABHCE	DJ	F
DJ	DJ	FK
F	DJ	F

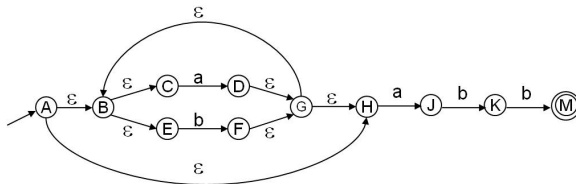
Step 4: Construct a New Table



	ϵ	a	b
A	BHCE	DJ	F
B	CE	D	F
C		D	
D	GBHCE	DJ	F
E			F
F	GBHCE	DJ	F
G	BHCE	DJ	F
H		J	
J			K
K			M
M			

	a	b
ABHCE	DJ	F
DJ	DJ	FK
F	DJ	F
FK	DJ	FM

Step 4: Construct a New Table



	ϵ	a	b
A	BHCE	DJ	F
B	CE	D	F
C		D	
D	GBHCE	DJ	F
E			F
F	GBHCE	DJ	F
G	BHCE	DJ	F
H		J	
J			K
K			M
M			


	a	b
ABHCE	DJ	F
DJ	DJ	FK
F	DJ	F
FK	DJ	FM
FM	DJ	F

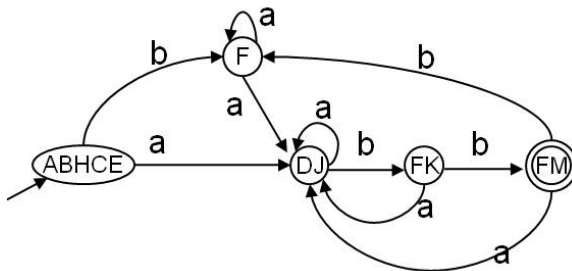
Step 5: Generate the DFA

	a	b
ABHCE	DJ	F
DJ	DJ	FK
F	DJ	F
FK	DJ	FM
FM	DJ	F

Step 5: Generate the DFA

	a	b
ABHCE	DJ	F
DJ	DJ	FK
F	DJ	F
FK	DJ	FM
FM	DJ	F

 Note: the number of states is not minimized



NFA to DFA. Remarks

- ❏ An NFA may be in many states at any time
- ❏ How many different possible states?
 - If there are N states, the NFA must be in some subset of those N states
 - How many non-empty subsets are there ?
 - $2^N - 1$ many states

Implementation of DFA Revisited

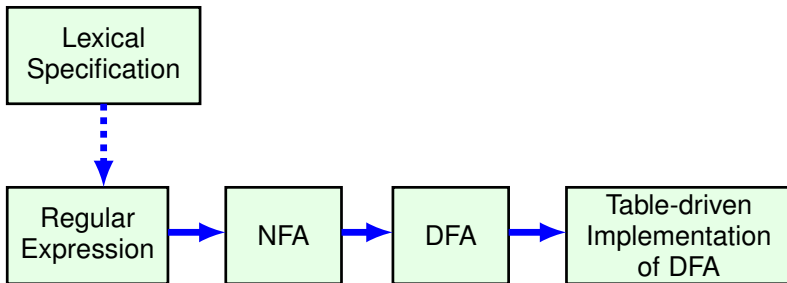
- A DFA can be implemented by a 2D table T
 - One dimension is “states”, the other dimension is “input characters”
 - For $S_a \xrightarrow{c} S_b$, we have $T[S_a, c] = S_b$
- DFA execution
 - If the current state is S_a and input is c , then read $T[S_a, c]$
 - Update the current state to S_b , assuming $S_b = T[S_a, c]$
 - It is efficient

Implementation Discussion

- ❑ NFA to DFA conversion is the heart of automated tools such as **lex**
- ❑ DFA could be very big
- ❑ In practice, lex-like tools trade off speed for space in the choice of NFA and DFA representations

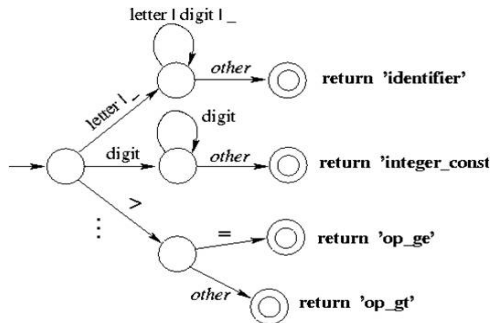
From RE to FA

Our implementation sketch



Structure of a Scanner Automaton

■ A scanner recognize multiple REs



How much should we match?

- In general, find the longest match possible

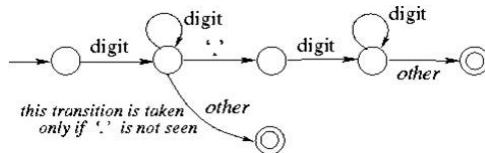
Example:

on input **123.45**, we match it as

(numConst, 123.45)

rather than

(numConst, 123), (dot, "."), (numConst, 45)



How to Match Keywords?

- ❑ Approach 1: Hardcode the keywords
- ❑ Approach 2: When the token is identified, check a special table

Example: to recognize the following tokens

Identifiers: `letter(letter|digit)*`

Keywords: `if, then, else`

Beyond Regular Languages

- ❑ Regular language is powerful, accomplish our lexical analysis task
- ❑ Regular language can describe email addresses, phone numbers, ... , etc.
- ❑ However, it is the weakest formal language
 - Many languages are not regular
 - C programming language is not
 - “(((...)))” is also not
 - Finite automata cannot remember # of times
- ❑ We need more powerful languages for describing these structures
 - In the next lecture, we will introduce **context-free language**