Lexical Analysis

What is Lexical Analysis

- Comes from the word lexicon, or dictionary
- **Lexical Analysis**: Partitioning a string into words
  - These words are also called **tokens**

- We will use this code as a running example:

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

- Code is provided as input string to lexical analysis

  $$\text{“}if(i == j)\backslash n\backslash tz = 0;\backslash nelse\backslash n\backslash tz = 1;\backslash n\text{”}$$

- Goal is turning the string into tokens, or tokenization

What is a Token ?

- Smallest unit that has meaning in a string
  - In English, tokens are English words:
    nouns, verbs, adjectives, ...
  - In a programming language:
    identifiers, integers, keywords, whitespace, ...

- A token is a tuple `(type, lexeme)`
  - `type`: the token type that the token belongs to
    - Identifier: string of letters and digits, starting with a letter
    - Integer: string of digits
    - Keyword: “else”, “if”, “while”, ...
    - Whitespace: string of blanks, newlines, and tabs
  - `lexeme`: actual string value of this token

Lexical Analysis is the act of Tokenization

- Output of lexical analysis is a stream of tokens
- Tokens are the input to Syntax Analysis (a.k.a. Parsing)
  - Parsers rely on token type to figure out role of each token
    E.g. a keyword is treated differently from an identifier

Lexical Analysis Tokenization Example

- Given "$if(i == j)\backslash n\backslash tz = 0;\backslash nelse\backslash n\backslash tz = 1;\backslash n$"
- What would be an output of lexical analysis?
  Recall a token is a tuple `(type, lexeme)`
- Output:
  (keyword, "if")(left-parenthesis, "(")(identifier, "i")(equals-op, "==")(identifier, "j")(right-parenthesis, ")")(whitespace, "$\backslash n\backslash t$")(identifier, "z")(assign-op, "=")(integer, "0")(semicolon, ";")(whitespace, "$\backslash n$")(keyword, "else")(whitespace, "$\backslash n\backslash t$")(identifier, "z")(assign-op, "=")(integer, "1")(semicolon, ";")(whitespace, "$\backslash n$")
- The lexer usually discards "non-interesting" tokens that don't contribute to parsing, e.g., whitespace, comments

Some language features makes lexing difficult

- FORTRAN compilation rule: whitespace is insignificant
  - Reason: inaccuracy of card punching by operators
- Consider
  - DO 5I=1,25
  - DO 5I=1.25
- This is the interpretation of the two statements:
  - Former: Iterate from I=1 to I=25 with step size 5
  - Latter: Assign1.25 to variable DO5I
- Reading left-to-right, cannot tell if DO5I is a variable or DO statement; Have to continue until "," or "." is reached.
  - "lookahead" may be required to decide on tokens
  - Feedback necessary from parser to lexical analysis

C++ language has difficult features too

- C++ has the Right Angle Brackets >> issue: https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2005/n1757.html
- typedef std::vector<std::vector<int> > Table; // OK
  typedef std::vector<std::vector<int>> Table; // Error
- Why? >> is read as one token which can either be:
  - A stream operator (e.g. cin >> var)
  - Or a shift operator (e.g. 1 >> 2)
- Space is needed in between to create two > tokens
- Fixed in C+11 standard so this is no longer an error
  - That makes tokenization decision on >> context dependent
  - Again forcing lexical analysis to get feedback from parser

Lexical Analysis Implementation

- Step 1:
  - Define a set of token types
    - Refer to language specifications
    - Types you choose depends on design of parser
    - Recall "$if(i == j)\backslash n \backslash tz = 0; \backslash nelse \backslash n \backslash tz = 1; \backslash n$"
    - Should "==" be one token? or two tokens? Depends.
    - Should "if","then","else" be separate types or just one keyword type? Depends.
- Step 2:
  - For each token type, describe which string belongs to it

Describing strings belonging to a token type

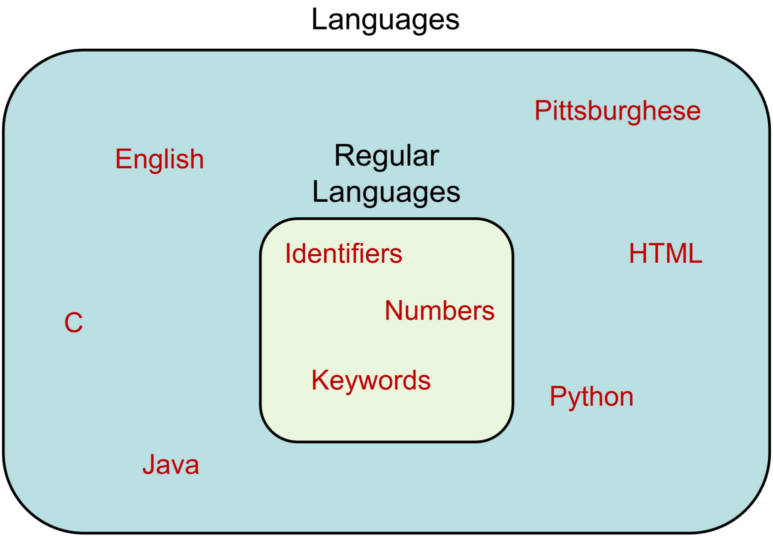
- A token type is something that looks like this:

  Identifier: string of letters and digits, starting with a letter
- Is there a more formal (mathematical) way to express this?
  - Yes! By using the formalism of **Languages**.

- Definition of **Language**:
  Let $\sum$ be a set of characters, a **language** over $\sum$ is a set of strings of the characters drawn from $\sum$
  - So by definition, any token type is a Language
  - And so are English, Java, Python, and HTML

- Some Languages can be very difficult to express formally
  - Imagine having to formally describe the English language!

Token types belong to a subset of Languages (called Regular Languages)



Regular Expressions express Regular Languages

- Definition of **Regular Expression**

  The **regular expressions (REs)** over $\sum$ are the total set of expressions that can be constructed using the following components:

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

- **Regular Languages** are defined as languages that can be expressed using **Regular Expressions.**

Atomic Regular Expressions

- Single character denotes a set of one string
  ‘c’ = { “c” }

- *Epsilon* or ϵ character denotes a zero length string ε = { “” }

- Empty set is { } = ϕ, not the same as ϵ
  size(ϕ) = 0
  size(ε) = 1
  length(ε) = 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)
  $A^* = \cup_{i \geq 0} A^i$ where $A^i = A...A$ (i times)

  in particular
  $A^* = \{\varepsilon\} + A + AA + AAA + ...$
  $A+ = A + AA + AAA + ... = A A^*$

The L(*Expression*) Notation

- L(*Expression*): means the language defined by *Expression*
- Some example languages defined using the L() notation:
  - $L(\varepsilon)$ = { “” }
  - L(‘c’) = { “c” }
  - $L(A+B) = L(A) \cup L(B)$
  - $L(AB) = \{ ab \mid a \in L(A) \text{ and } b \in L(B) \}$
  - $L(A^*) = \cup_{i \geq 0} L(A^i)$

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* = 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* = 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* = digit$^+$
    - **Q:** Is '000' an integer?
    - **Q:** Define an RE that excludes the above sequence.

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 string of blanks, newlines, 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 (letter* + digit*) the same?

- Whitespace: a non-empty string of blanks, newlines, tabs
  - whitespace = (‘ ’ + ‘\n’ + ‘\t’) $^+$

More Examples

- Phones number: consider (412) 624-0000
  - $\sum$ = digit $\cup$ { -, (,) }
  - area = digit ^{3}
  - exchange = digit ^{3}
  - phone = digit ^{4}

  - phoneNumber = '(' area ')' exchange '-' phone

- Email address: student @ pitt.edu
  - $\sum$ = letter $\cup$ { ., @ }
  - name = letter $^+$

  - emailAddress = name '@' name '.' name

More Regular Expression Notations

- Some "syntactic sugar" for regular expressions:

Union:	A + B	$\equiv$ A $\mid$ B
Option:	A + ε	$\equiv$ A ?
Range:	'a' + 'b' + ... + 'z'	$\equiv$ [a-z]
Excluded range:		
	complement of [a-z]	$\equiv$ [^a-z]

Implementation of Lexical Analysis

- We learnt how to precisely define token types
  — Regular expression (RE)

Implementation of Lexical Analysis

- We learnt how to precisely define token types
  — Regular expression (RE)

- How do we get from the RE to the actual tokenizer?

Implementation of Lexical Analysis

- We learnt how to precisely define token types
  — Regular expression (RE)

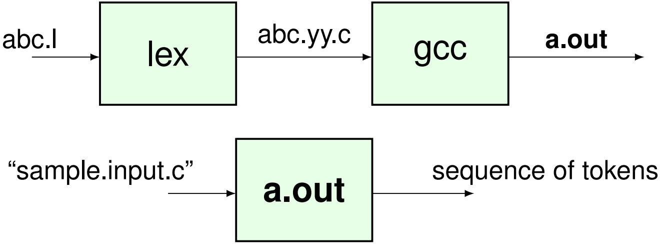
- How do we get from the RE to the actual tokenizer?
  - **Solution 1:** Write the code manually from REs

Implementation of Lexical Analysis

- We learnt how to precisely define token types
  — Regular expression (RE)

- How do we get from the RE to the actual tokenizer?
  - **Solution 1:** Write the code manually from REs
  - **Solution 2:** Convert RE to code using a tool
    - Tool generates the source code from the given REs
    - Lex (for C), Flex (for C++), JLex (for Java)

Lex: a Tool for Lexical Analysis



- Lex enables token definition using declarative coding
  - Declare tokens using regular expressions
  - Write action performed for each RE using imperative coding
- We will first describe structure of specification file
- The internals of the tool will be discussed later

Example Lex Specification File

```
/* 1. Regular expression definitions section */
%{
/* Code block inserted for includes and declarations */
#include <stdlib.h>
%}
string      [a-z]+
space       [ ]+
%%
 /* 2. Rules section: action for each regular expression */
{string}     { printf("lexeme: %s, len=%d\n", yytext, yyleng); }
{space}      { /* No action */ }
%%
/* 3. User code section */
int main() {
  while( yylex() != 0 ) {}
  return 0;
}
```

Example Lex Specification File: Explanation

- Overview of operation:
  1. Parser calls `yylex()` when ready to process the next token
  2. `yylex()` tokenizes longest string that matches an RE
  3. `yylex()` stores the token lexeme in `yytext`
  4. `yylex()` stores length of lexeme in `yyleng`
  5. `yylex()` executes the action { ... } in the rule for RE
  6. If action returns a value, `yylex()` returns that value.
     If action doesn't return, `yylex()` scans next token.
     `yylex()` returns 0 if no more tokens (EOF).

- To test the lexer without a parser, we need a lexer driver
  - The `main()` function serves as the lexer driver
  - On piping "hello world!" to input of lexer (a.out):

```
$ echo "hello world!"  | ./a.out
lexeme:  hello, len=5
lexeme:  world, len=5
!
```

How is the Specification File Converted to a Lexer

- The problem we face is

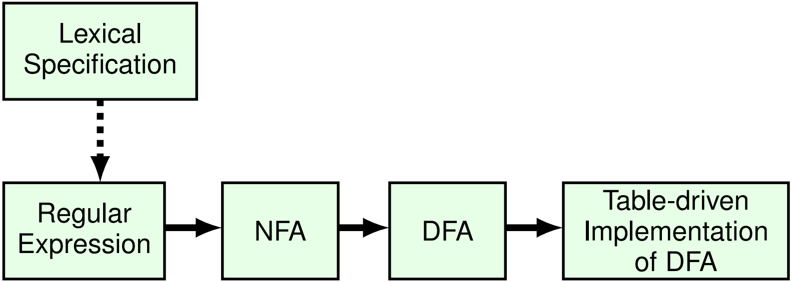
  Given a string **s** and a regular expression **RE**, is

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

Implementing Lexical Analysis with Finite Automata

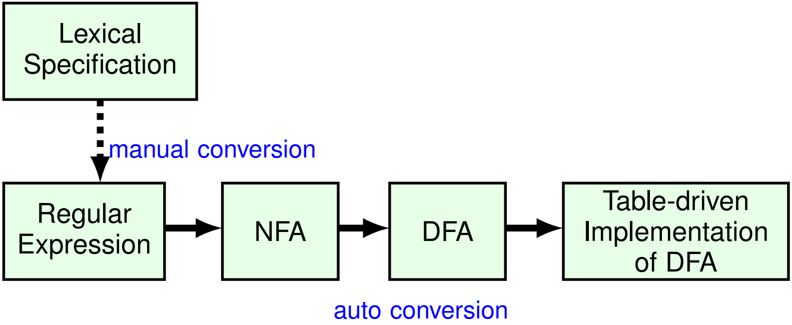
An Overview of RE to FA

- Our implementation sketch



An Overview of RE to FA

- Our implementation sketch



Implementation Outline

- RE ➟ NFA ➟ DFA ➟ Table-driven Implementation
  - Tokens are specified using Regular Expressions
  - Tokens are accepted using a table-driven DFA
    - Deterministic Finite Automata (DFAs)
    - Non-deterministic Finite Automata (NFAs)
  - Table implementations

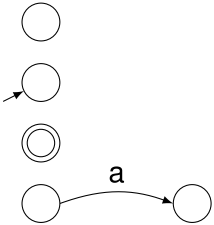
- I will soon show $RL \equiv L(RE) \equiv L(NFA) \equiv L(DFA)$
  - Will show an automated way to take RE all the way to DFA

Finite Automata

- A finite automaton 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 δ: $S_a \xrightarrow{input} S_b$

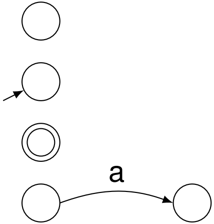
State Graph is a graphical representation of FA

- 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 transitions

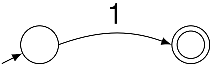


State Graph is a graphical representation of FA

- 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 transitions

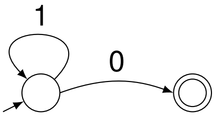


- Example: a finite state automaton that accepts only “1”



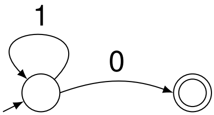
More State Graph Examples

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



More State Graph Examples

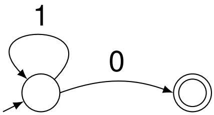
- A finite automaton accepting any number of **1**s followed by a single **0**. Here we have Alphabet = {0,1}



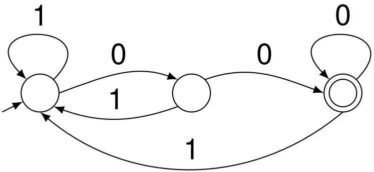
- Example: What language does the following state graph recognize? Here we have Alphabet = {0,1}

More State Graph Examples

- A finite automaton 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}



How tokens are accepted by Finite Automata

- Transition δ: $S_a \xrightarrow{c} S_b$ read as:
  When in state S_a, go to state S_b when scanned char is “c”

1. Begin at start state $n \in S$.
2. Transition until end of input or no transition possible
3. Read current state x
   - If $x \in$ accepting set F, then $\Rightarrow$ accept
   - Otherwise, $\Rightarrow$ reject

Table Implementation of a DFA

- Given the state graph of a DFA,

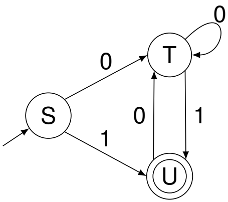
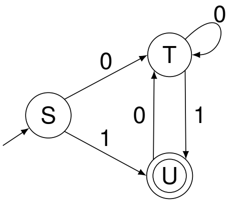


Table Implementation of a DFA

- Given the state graph of a DFA,



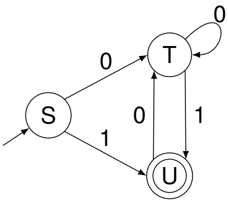
→ input characters

state ↓

	0	1
S		
T		
U		

Table Implementation of a DFA

- Given the state graph of a DFA,



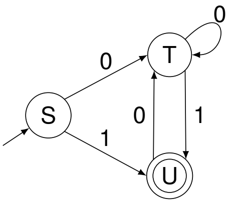
→ input characters

state ↓

	0	1
S	T	U
T	T	U
U	T	x

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

Table-driven Code:

```
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");
}
```

Table-driven Code is identical across DFAs

- Each RE has a different DFA, meaning different table
- But table-driven code for tokenization remains the same!

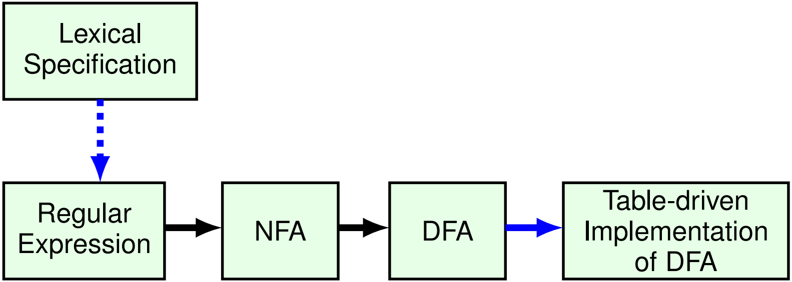
Table-driven Code is identical across DFAs

- Each RE has a different DFA, meaning different table
- But table-driven code for tokenization remains the same!
- Revisit our implementation outline

  RE ➟ NFA ➟ **DFA** ➟ **Table-driven Implementation**

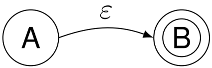
From RE to FA

- Our implementation sketch



Epsilon Moves

- Another kind of transition: ε-moves
  - Moves from state A to state B without reading input

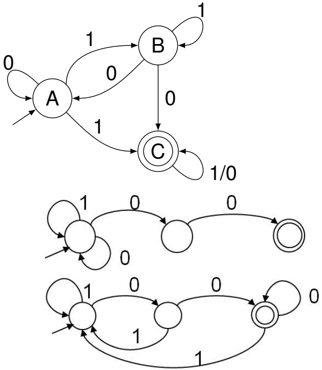


- ε-moves makes transitions nondeterministic
  - Instead of next character determining the next state, now machine has a choice of staying in A or moving to B

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
- DFAs are easier to implement
  - Real machines are deterministic by nature just like DFAs
  - Given a nondeterministic choice, hard to know what to do

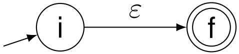
Examples



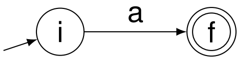
Converting RE to NFA

- 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 | t

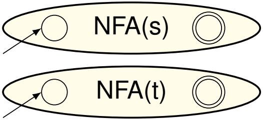
  - r = s t

  - r = s *

Converting RE to NFA (cont.)

- Step 2: processing compound REs

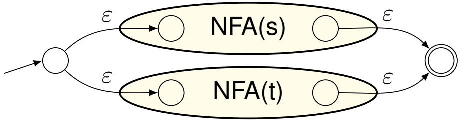
  - r = s | t



  - r = s t
  - r = s *

Converting RE to NFA (cont.)

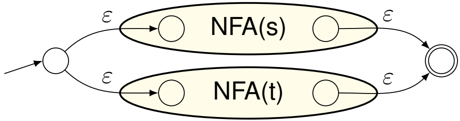
- Step 2: processing compound REs
  - r = s | t



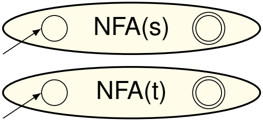
  - r = s t
  - r = s *

Converting RE to NFA (cont.)

- Step 2: processing compound REs
  - ➢ r = s | t



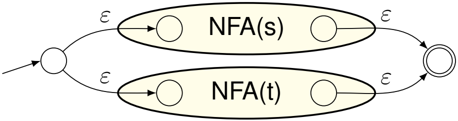
  - ➢ r = s t



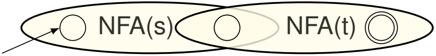
  - ➢ r = s *

Converting RE to NFA (cont.)

- Step 2: processing compound REs
  - r = s | t



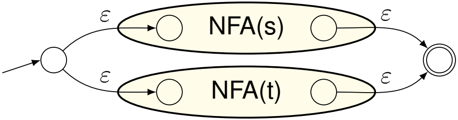
  - r = s t



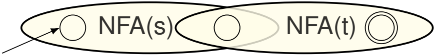
  - r = s *

Converting RE to NFA (cont.)

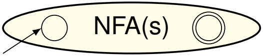
- Step 2: processing compound REs
  - r = s | t



  - r = s t



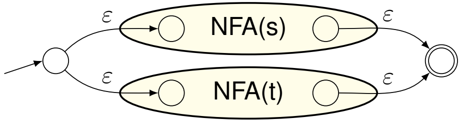
  - r = s *



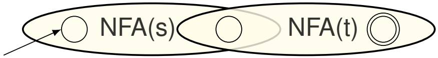
Converting RE to NFA (cont.)

- Step 2: processing compound REs

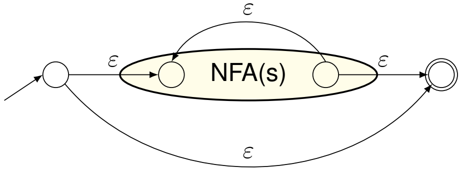
  - r = s | t



  - r = s t



  - r = s *

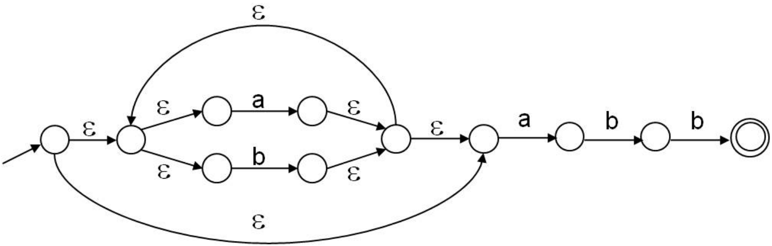


In-class Practice

- Convert “**(a|b)*****a b b**” to NFA

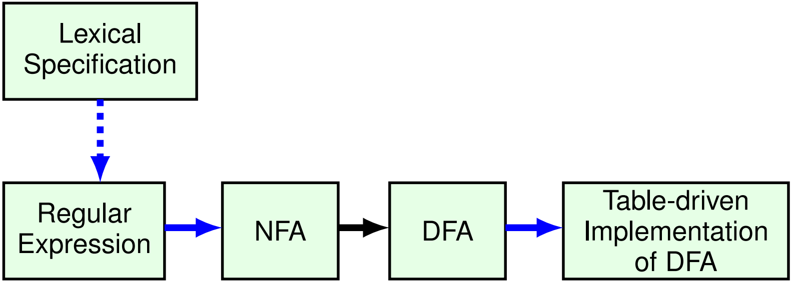
In-class Practice

- 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 has a choice of:
  - Whether to make ε-moves
  - Which of multiple transitions for a single input to take

Execution of Finite Automata

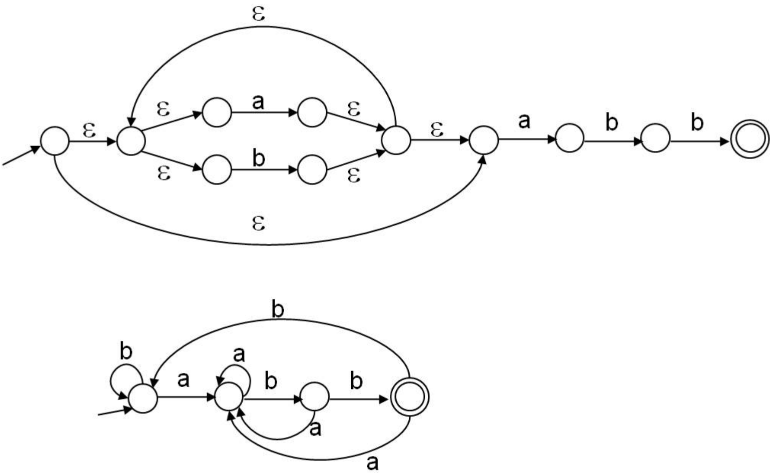
- A DFA can take only one path through the state graph
  - Completely determined by input
- A NFA has a choice of:
  - Whether to make ε-moves
  - Which of multiple transitions for a single input to take
- When to accept a token for DFAs and NFAs
  - DFA: When input string leads to a final state
  - NFA: When input string **can** lead to a final state

Execution of Finite Automata

- A DFA can take only one path through the state graph
  - Completely determined by input
- A NFA has a choice of:
  - Whether to make ε-moves
  - Which of multiple transitions for a single input to take
- When to accept a token for DFAs and NFAs
  - DFA: When input string leads to a final state
  - NFA: When input string **can** lead to a final state
- **Question:** Which one is more expressive?

$L(NFA) \equiv L(DFA) \equiv RL$

- Both accept "**(a|b)*****a b b**"



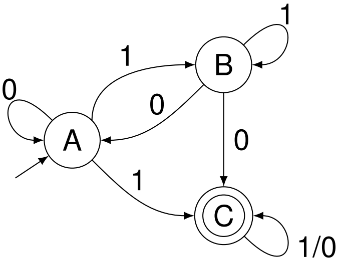
How to Convert NFA to DFA

- Basic idea: Given NFA, simulate its execution using 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

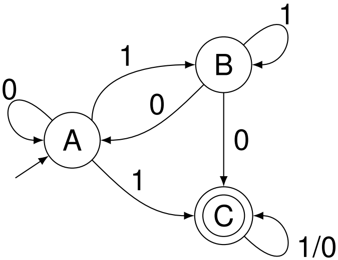
Example NFA to DFA

- What is the Equivalent DFA ?



Example NFA to DFA

- What is the Equivalent DFA ?

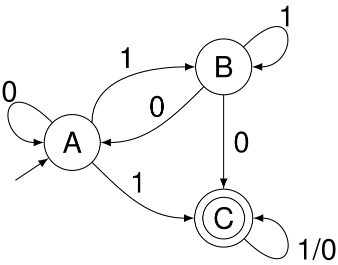


state ↓ → input characters

	0	1
A		
B		
C		

Example NFA to DFA

- What is the Equivalent DFA ?

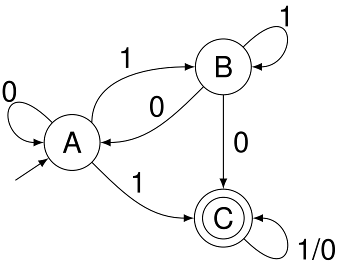


state ↓ → input characters

	0	1
A	A	BC
B	AC	B
C	C	C

Example NFA to DFA

- What is the Equivalent DFA ?

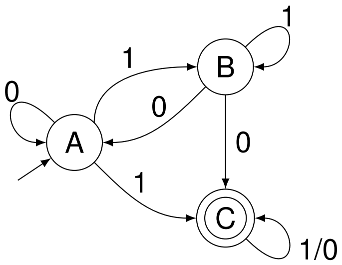


state ↓ → input characters

	0	1
A	A	BC
B	AC	B
C	C	C

Example NFA to DFA

- What is the Equivalent DFA ?

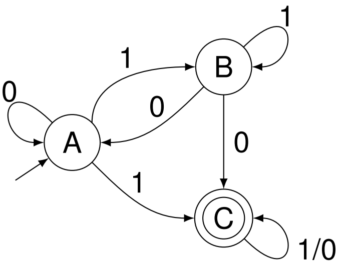


state ↓ → input characters

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

Example NFA to DFA

- What is the Equivalent DFA ?

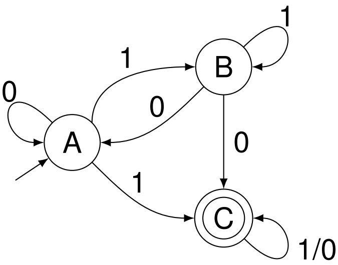


state ↓ → input characters

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

Example NFA to DFA

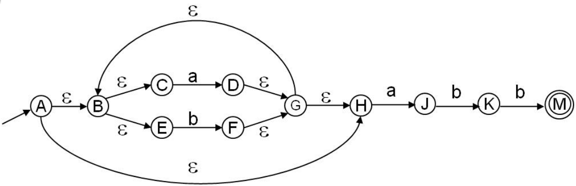
- What is the Equivalent DFA ?



state ↓ → input characters

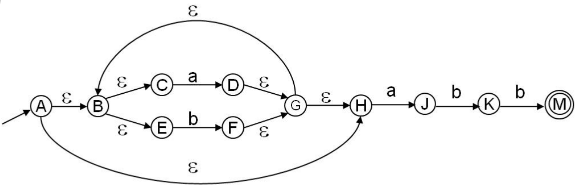
	0	1
A	A	BC
B	AC	B
C	C	C
AC	AC	BC
BC	AC	BC
AB	x	x
ABC	x	x

Algorithm Illustrated: Converting NFA to DFA



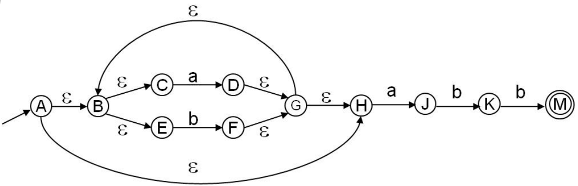
	ε	a	b
A			
B			
C			
D			
E			
F			
G			
H			
J			
K			
M			

Step 1: Construct the Table



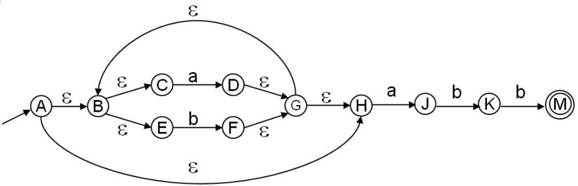
	ε	a	b
A	BH		
B	CE		
C		D	
D	G		
E			F
F	G		
G	BH		
H		J	
J			K
K			M
M			

Step 2: Construct ε-closure



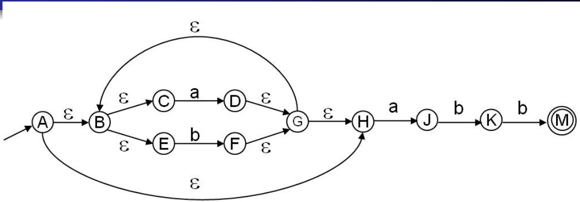
	ε	a	b
A	BHCE		
B	CE		
C		D	
D	GBHCE		
E			F
F	GBHCE		
G	BHCE		
H		J	
J			K
K			M
M			

Step 3: Update Other Columns



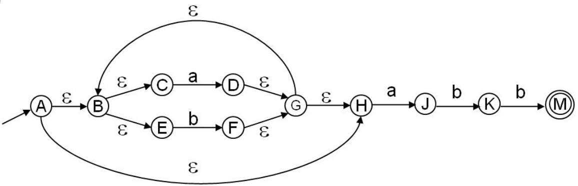
	ε	a	b
A	BHCE	DJ	F
B	CE	D	F
C		D	
D	GBHCE	DJ	F
E			F
F	GBHCE	DJ	F
G	BHCE	DJ	F
H		J	
J			K
K			M
M			

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			

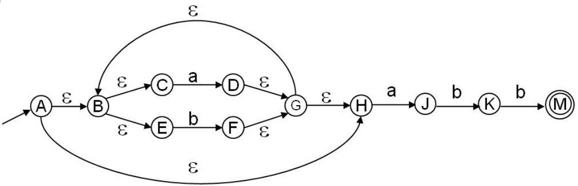
Step 4: Construct a New Table



	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

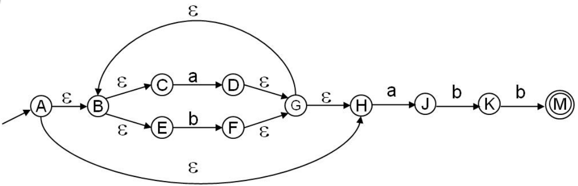
Step 4: Construct a New Table



	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

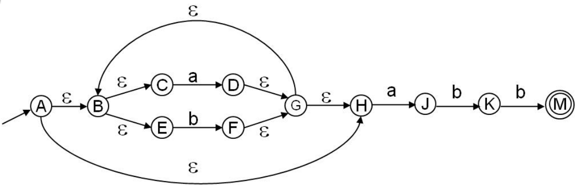
Step 4: Construct a New Table



	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

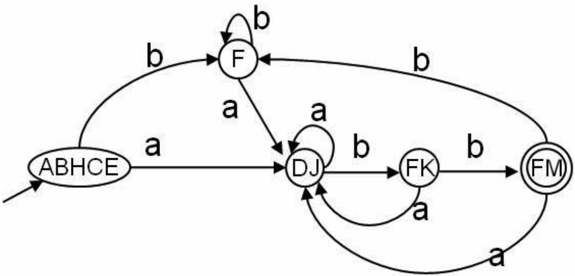
Step 5: Generate the DFA

	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

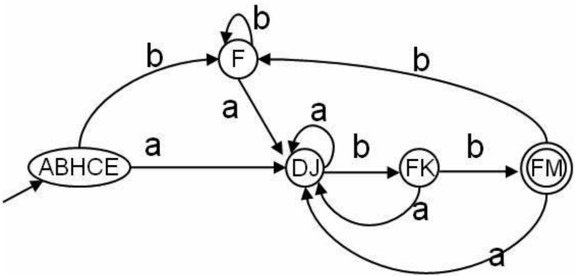
- Note: the number of states is not minimized



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



- States ABHCE and F can be merged! Why?

NFA to DFA. Space Complexity

- An NFA may be in many states at any time
- If NFA has N states, how many DFA states?
  - NFA must be in some subset of those N states
  - How many non-empty subsets are there ?
    - $2^N - 1$ many states
- The resulting DFA has $O(2^N)$ space complexity
  - Remember this is big-O. Typically much less.

NFA to DFA Time Complexity

- A DFA can be implemented by a 2D table T
  - One dimension is “states”, the other 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 T[S_a,c] = S_b
  - Requires $O(|X|)$ steps, where $|X|$ is the length of input
- NFA execution
  - At a given step, there is a set of possible states, up to N
  - On input c, must access table for each possible state to get next set of possible states
  - Requires $O(|X| * N)$ steps

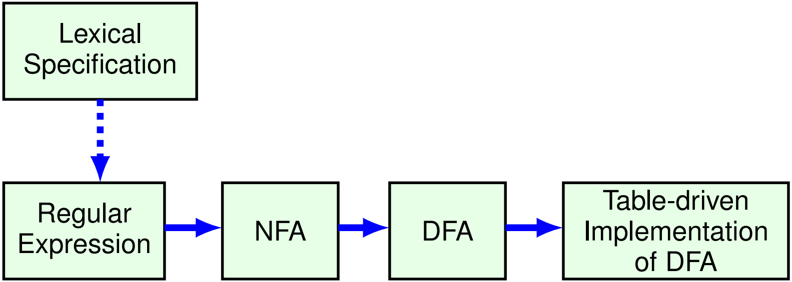
Implementation in Practice

- GNU **lex**
  - Convert regular expression to NFA
  - Convert NFA to DFA
  - Perform DFA state minimization to reduce space
  - Generate transition table from DFA
  - Perform table compression to further reduce space

- Most other automated lexers also trade off space for speed by choosing DFA over NFA

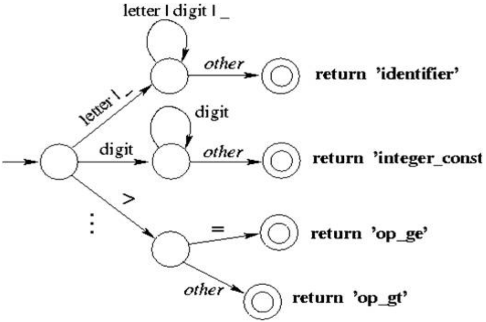
From RE to FA

- Our implementation sketch



Structure of a Scanner Automaton

- A scanner recognize multiple REs



How many characters should be matched?

- In general, find the longest match possible
- If same length, first rule in lex file takes precedence

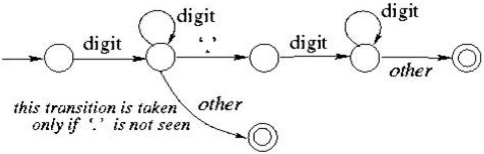
  Example:

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

  (numConst, 123.45)

  rather than

  (numConst, 123), (dot, “.”), (numConst, 45)



Beyond Regular Languages

- Regular languages are expressive enough for tokens
- Can express identifiers, strings, comments, etc.
- However, it is the least expressive formal language .
  - Many languages are not regular
    - C programming language is not
    - XML or JSON is not
    - "(((...)))" is also not
  - Finite automata cannot express nested structures
    - Can only remember a finite number of "("s encountered
- We need a more powerful language for parsing
  - In the next lecture, we will discuss **context-free languages**