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

  “$if(i == j)\backslash n\backslash t\backslash tz = 0; \backslash telse\backslash n\backslash tz = 1; \backslash n$”

- Goal: partition input string into substrings
  - where these substrings are **tokens**

What is a Token ?

- Token: a "word" in language (smallest unit with meaning)
  - Categorized into classes according to its role in language
  - Token classes in English:

    noun, verb, adjective, ...
  - Token classes in a programming language:

    integer, keyword, whitespace, identifier, ...

- Token classes (should) comprise distinct sets of strings
  - Integer: a non-empty string of digits
  - Keyword: “else”, “if”, “while”, ...
  - Whitespace: a sequence of blanks, newlines, and tabs
  - Identifier: a string of letters and digits, starting with a letter, that is not a keyword

What is Tokenization?

- Lexical analysis is also called **tokenization**
- Tokenization:
  - Partitioning input string to a sequence of tokens
  - Classifying each token into a token class
- Tokens are input to the syntax analyzer (also called Parser)
  - Parser relies on token classes to identify roles
    (E.g., a keyword is treated differently than an identifier)

Designing a Lexical Analyzer

- Step 1:
  - Define a finite set of token classes
    - Describe all items of interest
    - Depends on language, design of parser

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

    - identifier, integer, keyword, whitespace
    - should "==" be one token? or two tokens?

- Step 2:
  - Describe which string belongs to which token class

Lexical Analyzer: Implementation

- An implementation must do two things
  1. Recognize the token class the substring belongs to
  2. Return the value or lexeme of the token
- A token is a tuple `(class, lexeme)`
  "$if(i == j)\backslash n \backslash t \backslash tz = 0; \backslash telse \backslash n \backslash tz = 1; \backslash n$"
  - ➢ Result of lexical analysis:
    (keyword, *if*) (leftparen,() (id,*i*) (equals,$==$) (id,*j*)
    (rightparen,)) (id,*z*) (assign,$=$) (int, 0) (semicolon, ;)
    (keyword, *else*) (id,*z*) (assign,$=$) (int, 1) (semicolon, ;)
- The lexer usually discards "non-interesting" tokens that don't contribute to parsing, e.g., whitespace, comments
- If token classes are non-ambiguous, tokens can be recognized in a single left-to-right scan of input string
- Problems can occur when classes are ambiguous

Ambiguous tokens in FORTRAN

- FORTRAN compilation rule: whitespace is insignificant
  - Motivated from inaccurate card punching by operators
- Consider
  - DO 5I=1,25
  - DO 5I=1.25
- Different interpretations of DO 5I
  - The first: a loop iteration from 1 to 25 with step size 5
  - The second: an assignment of 1.25 to variable DO5I
- Reading left-to-right, cannot tell if DO5I is a variable or DO statement; Have to continue until "," or "." is reached.

Ambiguous tokens in C++

- The problem is not only limited to Fortran
- C++ template syntax

  FOO<Bar>
- C++ stream syntax

  cin >> var

- Now, the problem

  FOO<Bar<Bazz>>

  Is “>>” a stream operator or two consecutive brackets?

Lesson Learned

- Observations from examples:
  - “lookahead” or “lookbehind” may be required to decide among different choices of tokens
  - Extracting some tokens requires looking at the larger context or structure
  - Structure emerges only at parsing stage with parse tree
  - Hence, sometimes feedback from parser needed for lexing
- However, by and large, tokens do not overlap
  - Tokenizing can be done in one pass w/o parser feedback
  - Clean division between lexical and syntactic analyses

Specifying Tokens in a Language

- Token specification is part of language specification
- All language specifications should be:
  - Formal, unambiguous, and easy to understand
  - Easy to implement efficiently in a compiler
- **Regular Expressions** is a good way to specify tokens
  - Simple yet powerful (able to express patterns)
  - Tokenizer implementation can be generated automatically from specification (using a translation tool)
  - Resulting implementation is provably efficient
- But first we need to take a detour and talk about languages

Languages

Definition

Let $\sum$ be a set of characters, a **language** over $\sum$ is a set of strings of the characters drawn from $\sum$

Examples of Languages

- Alphabet $\sum$ = (set of) English characters
  Language L = (set of) English sentences
- Alphabet $\sum$ = (set of) Digits, +, -
  Language L = (set of) Integer numbers
- Alphabet $\sum$ = (set of) English characters
  Language L = (set of) Tokens in C
- Languages are **subsets of all possible strings**
  - Not all strings of English characters are sentences
  - Not all sequences of digits and signs are integers
  - Not all strings of English characters are tokens
- The set of tokens is also a language, just like English (!)

Regular Expressions and Regular Languages

- Need a notation to specify strings in a particular language
  - ➢ More complex languages need more complex notations
- A simple notation is **regular expressions**
  - ➢ Can express simple patterns (e.g. repeating sequences)
  - ➢ Not powerful enough to express English (or even C)
  - ➢ But powerful enough to express tokens such as identifiers
- Languages that can be expressed using regular expressions are called **regular languages**
- We will learn more complex languages and how to express them later in the lecture

Atomic Regular Expressions

- Smallest RE that cannot be broken down further
- 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
  $A^* = \{\varepsilon\} + A + AA + AAA + ...$
  $A+ = A + AA + AAA + ... = A\,A^*$

Regular Expressions

Definition

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

Definition

The **regular languages (RLs)** over $\sum$ are the total set of languages that can be expressed using REs.

- L(*expression*): the language generated by *expression*
  - $L(\varepsilon)$ = { “” }
  - L(‘c’) = { “c” }
  - $L(A+B) = L(A) \cup L(B)$
  - $L(AB) = \{ ab \mid a \in L(A) \text{ and } b \in L(B) \}$
  - $L(A^*) = \cup_{i \geq 0} L(A^i)$

RE Examples

- Keyword: “else” or “if” or “while” or ...

RE Examples

- Keyword: “else” or “if” or “while” or ...
  - keyword = ‘else’ + ‘if’ + ‘while’ + ...
  - ‘else’ abbreviates
    ‘e’ (concatenate) ‘l’ (concatenate) ‘s’ (concatenate) ‘e’

RE Examples

- Keyword: “else” or “if” or “while” or ...
  - keyword = ‘else’ + ‘if’ + ‘while’ + ...
  - ‘else’ abbreviates

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

RE Examples

- Keyword: “else” or “if” or “while” or ...
  - keyword = ‘else’ + ‘if’ + ‘while’ + ...
  - ‘else’ abbreviates

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

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

RE Examples

- Keyword: “else” or “if” or “while” or ...
  - keyword = ‘else’ + ‘if’ + ‘while’ + ...
  - ‘else’ abbreviates
    ‘e’ (concatenate) ‘l’ (concatenate) ‘s’ (concatenate) ‘e’

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

RE Examples

- Keyword: “else” or “if” or “while” or ...
  - keyword = ‘else’ + ‘if’ + ‘while’ + ...
  - ‘else’ abbreviates
    ‘e’ (concatenate) ‘l’ (concatenate) ‘s’ (concatenate) ‘e’

- 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 RE Examples

- Identifier: strings of letters or digits, starting with a letter
  - letter = ‘A’ + ... + ‘Z’ + ’a’ + ... + ‘z’
  - Identifier = letter (letter + digit)*

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

More RE 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 sequence of blanks, newlines and tabs
  - whitespace = (‘ ’ + ‘\n’ + ‘\t’) $^+$

More RE 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

RE in Programming Languages

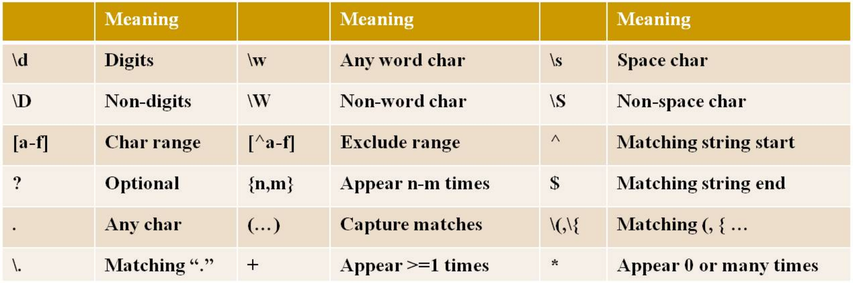
- RE used in languages
  - RE in PERL,

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

Implementation of Lexical Analysis

Implementation of Lexical Analysis

- We have learnt how to specify tokens for lexical analysis
  — Regular expression (RE)

Implementation of Lexical Analysis

- We have learnt how to specify tokens for lexical analysis
  — Regular expression (RE)

- How do we go from specification to implementation?

Implementation of Lexical Analysis

- We have learnt how to specify tokens for lexical analysis
  — Regular expression (RE)

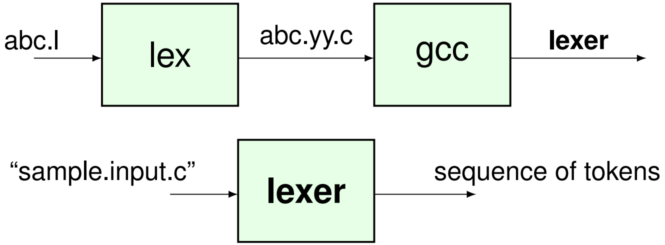
- How do we go from specification to implementation?
  - **Solution 1:** to implement using a tool — Lex (for C), Flex (for C++), Jlex (for java)
    - Programmer specifies tokens using RE formalism
    - The tool generates the source code from the given REs

Implementation of Lexical Analysis

- We have learnt how to specify tokens for lexical analysis
  — Regular expression (RE)

- How do we go from specification to implementation?
  - **Solution 1:** to implement using a tool — Lex (for C), Flex (for C++), Jlex (for java)
    - Programmer specifies tokens using RE formalism
    - The tool generates the source code from the given REs
  - **Solution 2:** to write the code yourself
    - More freedom; even tokens not expressible through REs
    - But difficult to verify; not self-documenting; not portable; usually not efficient
    - Generally not encouraged

Lex: a Tool for Lexical Analysis



- Big difference from your previous coding experience
  - Write REs instead of C code to implement them
  - Write actions in C associated with each RE
    (Usually generating the correct token)
- abc.yy.c is C code after REs in abc.l are translated to C
- Implementation of the Lex tool itself will be discussed later

Lex Specifications

```
%{ /* include, extern, etc. */
int yyline = 1, yycolumn = 1;
#include "token.h"
%}
 /* pattern definitions. format: name + definition */
number    [0-9]+
newline   \n
%%
 /* rules. format: pattern + action */
{number}    {
   printf("Token: int const %s (len=%d)", yytext, yyleng);
   return ICONSTNUM;
}
{newline}   yyline++;
%%
 /* auxiliary user code */
int myTableInsert() { ... }
```

Lex Generated Code

```
int yyline = 1, yycolumn = 1;
#include “token.h”
/* rules. format: pattern + action */
int yylex (void) {
   while ( 1 ) { /* loop until token returned or EOF */
     /* read input until pattern detected and assign yy_act */
     switch(yy_act) {
     case 1:
       printf(“Token: int const %s (len=%d)”, yytext, yyleng);
       return ICONSTNUM;
     case 2:
       yyline++;
     }
   }
}
 /* auxiliary user code */
int myTableInsert() { ... }
```

yylex() Operation

- Parser calls yylex() repeatedly to retrieve tokens from lexer
- yylex() reads in characters until a token is returned or EOF
  1. Read in characters until pattern detected or EOF
  2. For characters not part of any pattern, print to stdout
  3. For string of characters matching a pattern:
     - Assign pointer to beginning of string to *yytext*
     - Assign length of string to *yyleng*
     - Perform corresponding user action using above variables (May cause a token to be returned)
  4. Repeat from Step 1.
- yylex() always tries to consume longest string possible
  - ➢ Given string “if”, it chooses token (keyword, *if*) over two tokens (id, *i*) (id, *f*)
  - ➢ Given two patterns of same length, patterns are chosen in precedence they appear in specification file

Project 1 Implementation Notes

- Write regular expressions for all tokens in language
- Comments: keep track of nesting level if nesting allowed
- String table maintained by you to detect duplicate identifiers / strings
- *yytext*, *yyleng* maintained by lex library
- *yylval*, *yyline*, *yycolumn* maintained by you
- Special characters
  - '\n' — newline
  - '\t' — tab
  - '\'' — single quote
  - '\\' — backslash

Discussion of RE and Lexical Analysis

- Lexer uses RE to extract tokens from input string
- Regular Expressions is only a language specification
  - An implementation is still needed
  - How does Lex translate REs in a specification file to pattern matching C code?
- The code should be able to answer the question:

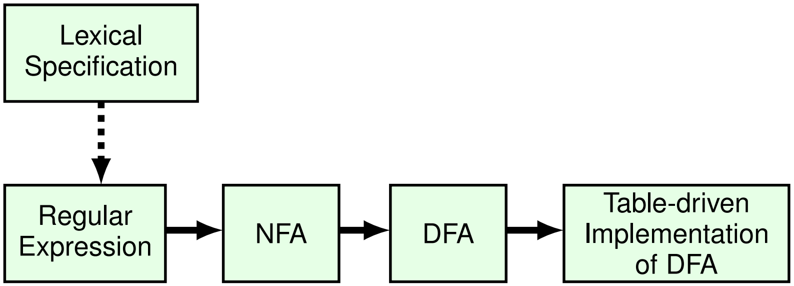
  Given a string **s** and a regular expression **RE**,

  is **s** $\in$ **L(RE) ?**

Implementing Lexical Analysis with Finite Automata

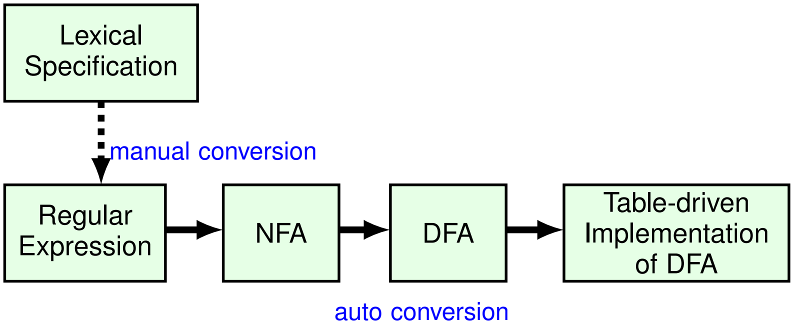
An Overview of RE to FA

- Our implementation sketch



An Overview of RE to FA

- Our implementation sketch



Implementation Outline

- RE ➟ NFA ➟ DFA ➟ Table-driven Implementation
- We will discuss in this order:
  1. Deterministic Finite Automata (DFAs)
  2. Converting DFAs to Table-driven implementations
  3. Non-deterministic Finite Automata (NFAs)
  4. Converting REs to NFAs
  5. Converting NFAs to DFAs
- Let's start by talking about what a DFA is

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

Notations

- Some alternative notations we will use:

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]

Finite Automata

- Automaton (pl. automata): A machine or program
- Finite automaton: A program with a finite number of states
- 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: $S_a \xrightarrow{input} S_b$

More About Transitions

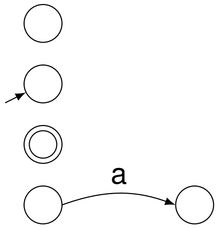
- Transition δ: $S_1 \xrightarrow{a} S_2$ means:

  When in state S_1, on input “a”, go to state S_2
- Begin from start state n, consume input chars one by one
- At the end of input, if current state X
  - $X \in$ accepting set F, then $\Rightarrow$ accept
  - otherwise, $\Rightarrow$ reject

- An FA is a program for classifying strings (accept, reject)
  - In other words, a program for recognizing a language
  - What languages are recognizable? L(FA) $\equiv$ L(RE)
  - The Lex tool essentially does the following translation: REs (Specification) $\Rightarrow$ FAs (Implementation)

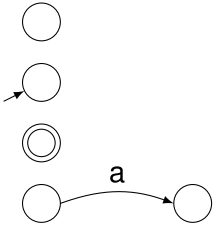
State Graph

- A **state graph** is a good way to visualize a FA
- A **state graph** includes
  - A set of states
  - A start state
  - A set of accepting states
  - A set of transitions

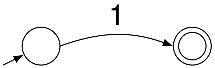


State Graph

- A **state graph** is a good way to visualize a FA
- A **state graph** includes
  - ➢ A set of states
  - ➢ A start state
  - ➢ A set of accepting states
  - ➢ A set of transitions

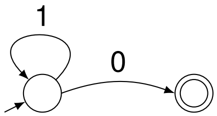


- Example: a finite state automata that accepts only “1”



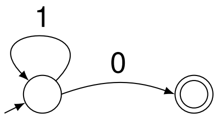
More Examples

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



More Examples

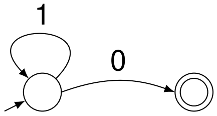
- A finite automata accepting any number of **1**s followed by a single **0**. Alphabet = {0,1}.



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

More Examples

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



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

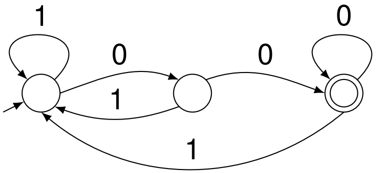


Table Implementation of a DFA

- Now let's convert the DFA to a table implementation

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

Table Implementation of a DFA

- Given the state graph of a DFA,

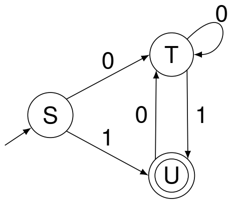
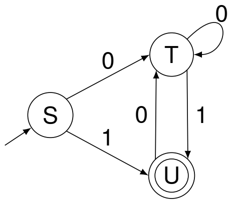


Table Implementation of a DFA

- Given the state graph of a DFA,



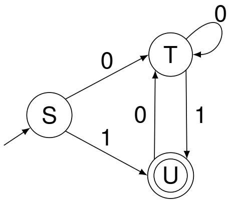
→ input characters

state ↓

	0	1
S		
T		
U		

Table Implementation of a DFA

- Given the state graph of a DFA,



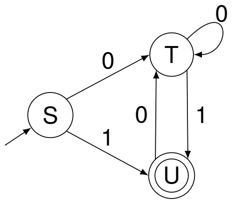
→ input characters

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:

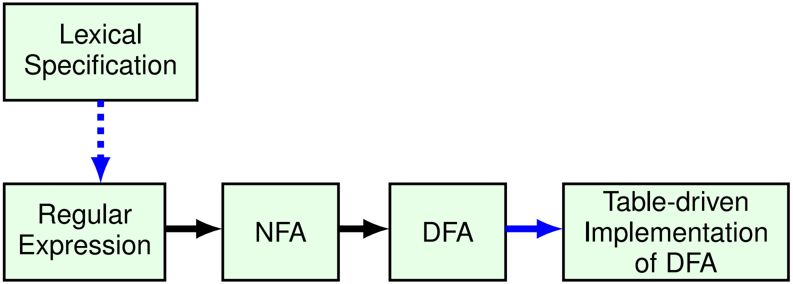
```
DFA() {
    state = “S”;
    while (!done) {
        ch = fetch_input();
        state = Table[state][ch];
        if (state == “x”)
            print(“reject”);
    }
    if (state ∈ F)
        printf(“accept”);
    else
        printf(“reject”);
}
```

Discussion

- Implementation can be automatically generated
  - Each RE has a different DFA, and different tables
  - However string recognition code is identical regardless
  - Hence, Lex tool need only generate new table for new DFA
- Implementation is provably efficient
  - Needs finite memory O(S X Σ)
    - Size of transition table
  - Needs finite time O(input length)
    - Number of state transitions

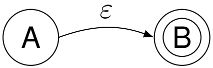
From RE to NFA

- Our implementation sketch



Epsilon Moves

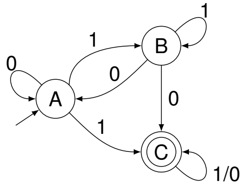
- Another kind of transition: ε-moves
  - Machine can move from state A to state B without reading any input

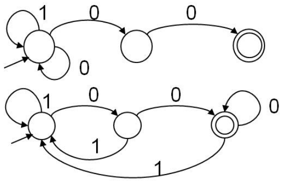

$$\begin{tikzcd} A \arrow[r, bend left, "\varepsilon"] & \boxed{B} \end{tikzcd}$$

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

NFA Examples



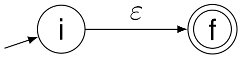


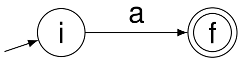
Execution of Finite Automata

- A DFA can take only one path through the state graph
  - Completely determined by input
- An NFA has multiple choices available to it
  - Whether to make ε-moves
  - Which of multiple transitions for a single input to take
  - Acceptance of NFAs
    - An NFA can end up in multiple states
    - **Rule**: NFA accepts an input if at least one of those final states is an accepting state
- $DFA \subseteq NFA$ and $L(DFA) \subseteq L(NFA)$
  - All DFAs are NFAs by definition
- Now is $L(NFA) \subseteq L(DFA)$?
  - Are all Ls expressible using NFA expressible using DFA?
  - Yes, as we will later learn, hence $L(NFA) \equiv L(DFA)$.

Converting RE to NFA

- McNaughton-Yamada-Thompson Algorithm
- Step 1: processing atomic REs
  - ε expression
  - single character RE a



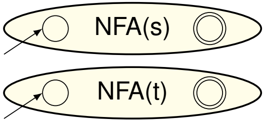


Converting RE to NFA (cont.)

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

Converting RE to NFA (cont.)

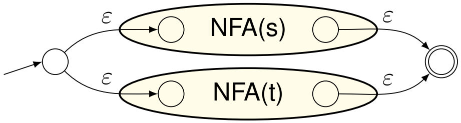
- Step 2: processing compound REs
  - r = s | t



  - r = s t
  - r = s *

Converting RE to NFA (cont.)

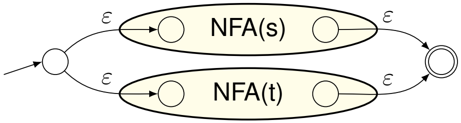
- Step 2: processing compound REs
  - r = s | t



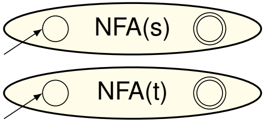
  - r = s t
  - r = s *

Converting RE to NFA (cont.)

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



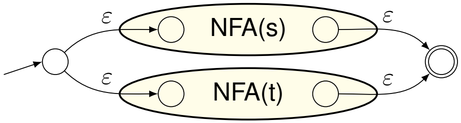
  - r = s t



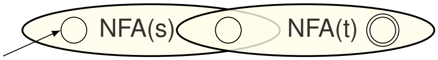
  - r = s *

Converting RE to NFA (cont.)

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



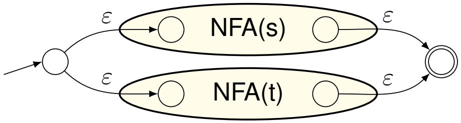
  - r = s t



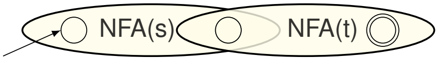
  - r = s *

Converting RE to NFA (cont.)

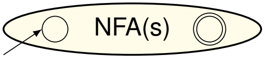
- Step 2: processing compound REs
  - r = s | t



  - r = s t

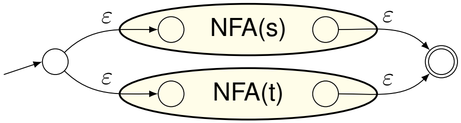


  - r = s *

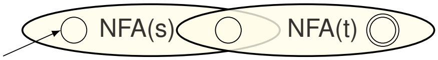


Converting RE to NFA (cont.)

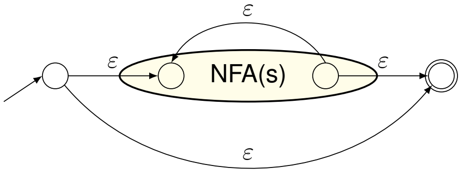
- Step 2: processing compound REs
  - r = s | t



  - r = s t



  - r = s *

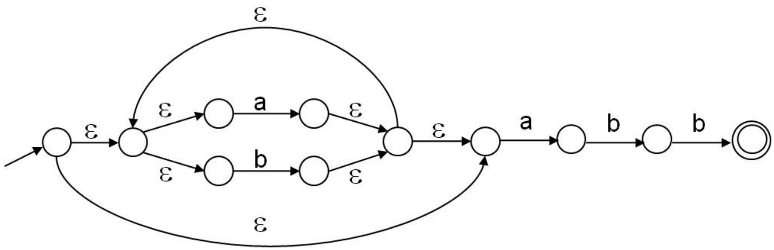


In-class Practice

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

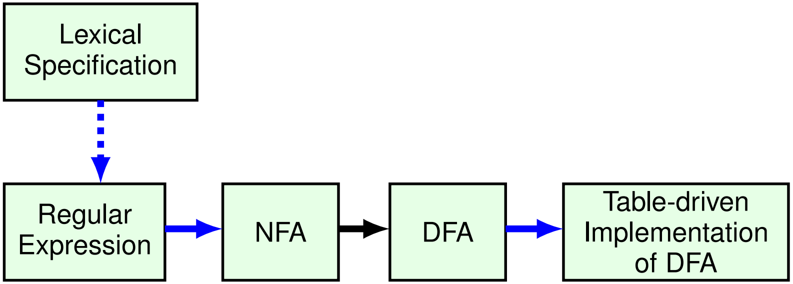
In-class Practice

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



From RE to FA

- Our implementation sketch

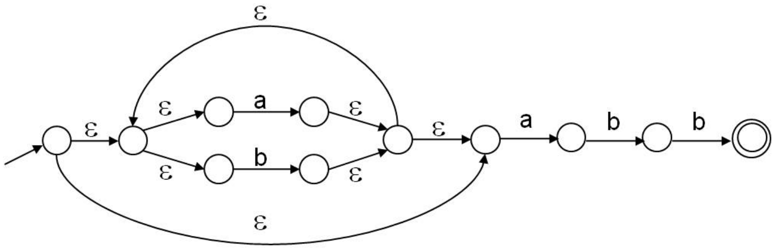


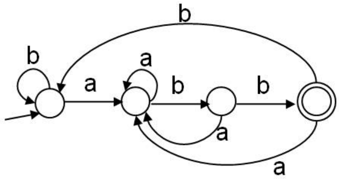
Converting NFA to DFA

- **Question:** is $L(NFA) \subseteq L(DFA)$?
  - Otherwise, conversion would be futile
- **Theorem:** $L(NFA) \equiv L(DFA)$
  - Both recognize regular languages L(RE)
  - Will show $L(NFA) \subseteq L(DFA)$ by construction (NFA $\rightarrow$ DFA)
  - Since $L(DFA) \subseteq L(NFA)$, $L(NFA) \equiv L(DFA)$
- Resulting DFA consumes more memory than NFA
  - Potentially larger transition table as shown later
- But DFAs are faster to execute
  - For DFAs, number of transitions == length of input
  - For NFAs, number of potential transitions can be larger
- NFA $\rightarrow$ DFA conversion is done because the speed of DFA far outweigh its extra memory consumption

NFA and DFA

- Both accept “**(a|b)*****a b b**”





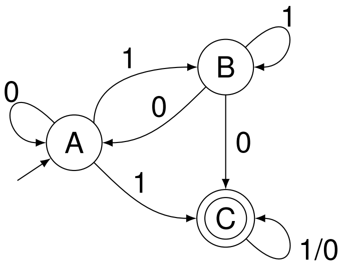
How to Convert NFA to DFA

- Idea: Given a NFA, simulate its execution using a DFA
  - At step *n*, the NFA may be in any of multiple possible states
  - Set of possible states is a state in the new DFA
  - If any possible state is an accepting state in the NFA, it is an accepting state in the DFA
- 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

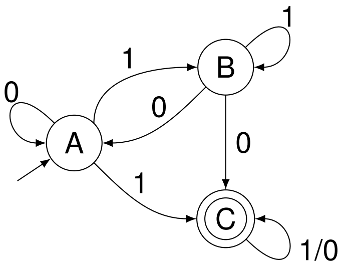
Example NFA to DFA

- What is the Equivalent DFA ?



Example NFA to DFA

- What is the Equivalent DFA ?

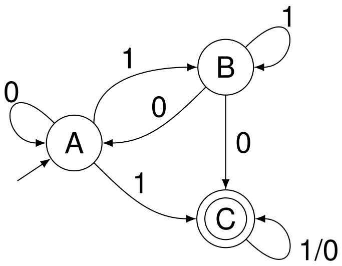


state ↓   → alphabet

	0	1
A		
B		
C		

Example NFA to DFA

- What is the Equivalent DFA ?

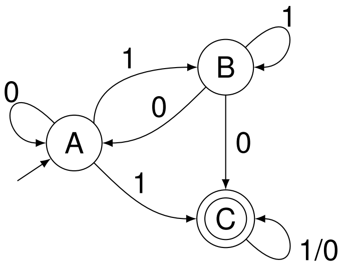


state ↓ → alphabet

	0	1
A	A	BC
B	AC	B
C	C	C

Example NFA to DFA

- What is the Equivalent DFA ?

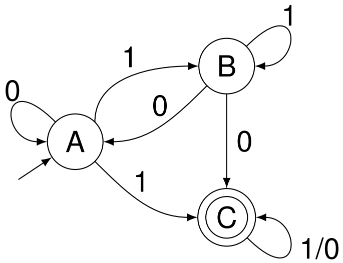


state ↓ → alphabet

	0	1
A	A	BC
B	AC	B
C	C	C

Example NFA to DFA

- What is the Equivalent DFA ?

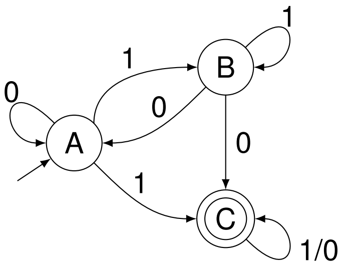


state ↓ → alphabet

	0	1
A	A	BC
B	AC	B
C	C	C

Example NFA to DFA

- What is the Equivalent DFA ?

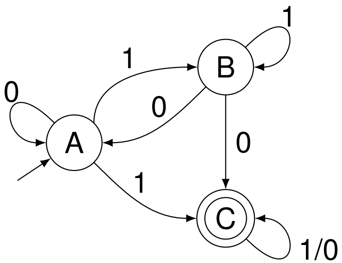


state ↓ → alphabet

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

Example NFA to DFA

- What is the Equivalent DFA ?

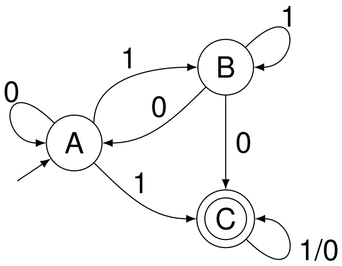


state ↓ → alphabet

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

Example NFA to DFA

- What is the Equivalent DFA ?

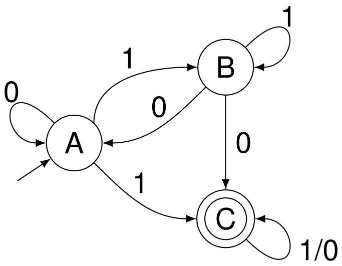


state ↓ → alphabet

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

Example NFA to DFA

- What is the Equivalent DFA ?

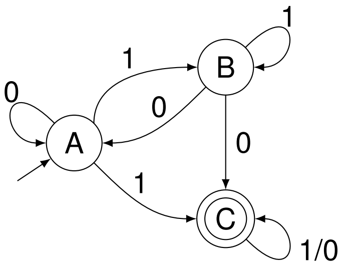


state ↓ → alphabet

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

Example NFA to DFA

- What is the Equivalent DFA ?

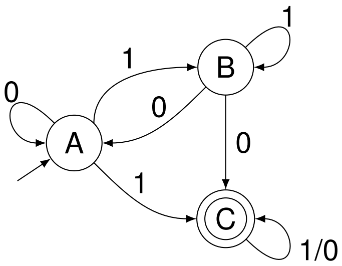


state ↓ → alphabet

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

Example NFA to DFA

- What is the Equivalent DFA ?

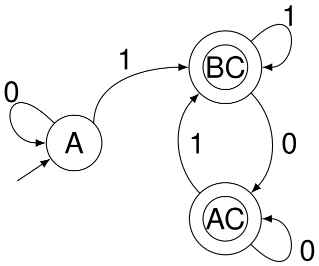


state ↓ → alphabet

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

Example NFA to DFA

- What is the Equivalent DFA ?

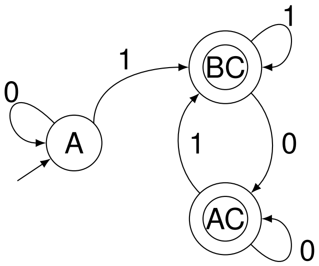


state ↓ → alphabet

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

Example NFA to DFA

- What is the Equivalent DFA ?



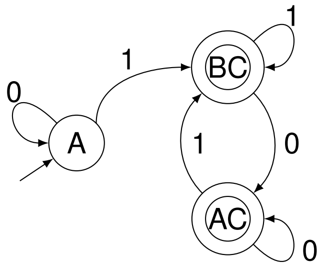
state ↓ → alphabet

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

- Is the DFA minimal? (See Textbook 3.9.6: Minimization)

Example NFA to DFA

- What is the Equivalent DFA ?



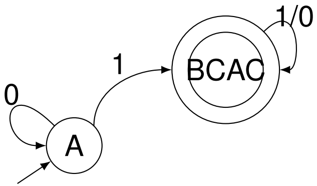
state ↓ → alphabet

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

- Is the DFA minimal? (See Textbook 3.9.6: Minimization)
  - ➢ States BC and AC do not need differentiation ⇒ merge

Example NFA to DFA

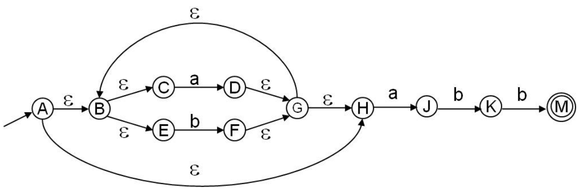
- What is the Equivalent DFA ?



state ↓ → alphabet

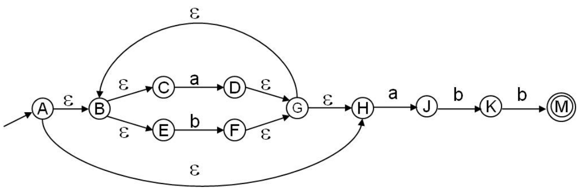
	0	1
A	A	BC
B	AC	B
C	C	C
BC	AC	BC
AC	AC	BC
AB	x	x
ABC	x	x

Algorithm Illustrated: Converting NFA to DFA



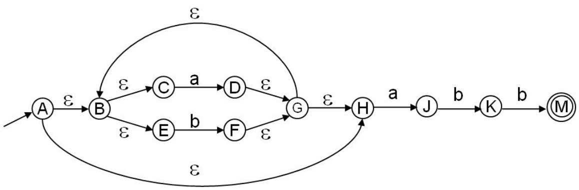
	ε	a	b
A			
B			
C			
D			
E			
F			
G			
H			
J			
K			
M			

Step 1: Construct the NFA Table



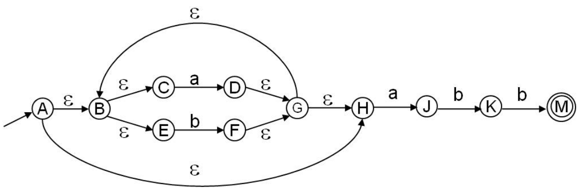
	ε	a	b
A	BH		
B	CE		
C		D	
D	G		
E			F
F	G		
G	BH		
H		J	
J			K
K			M
M			

Step 2: Update ε Column to ε-closure



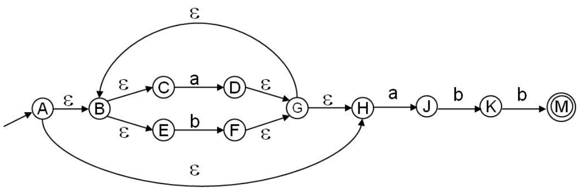
	ε	a	b
A	BHCE		
B	CE		
C		D	
D	GBHCE		
E			F
F	GBHCE		
G	BHCE		
H		J	
J			K
K			M
M			

Step 3: Update Other Columns Based on ε-closure



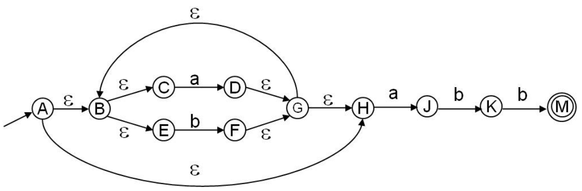
	ε	a	b
A	BHCE	DJ	F
B	CE	D	F
C		D	
D	GBHCE	DJ	F
E			F
F	GBHCE	DJ	F
G	BHCE	DJ	F
H		J	
J			K
K			M
M			

Step 4: Construct the DFA 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			

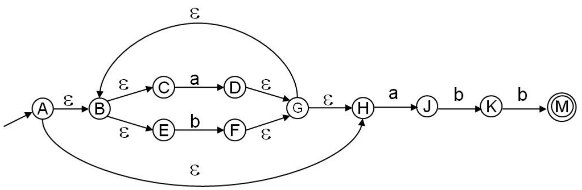
Step 4: Construct the DFA Table



	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
A	DJ	F

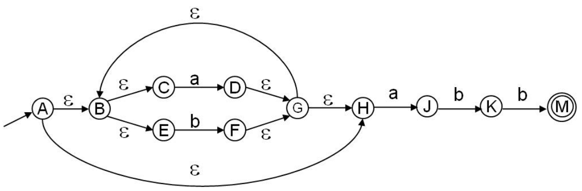
Step 4: Construct the DFA Table



	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
A	DJ	F
DJ	DJ	FK
F	DJ	F

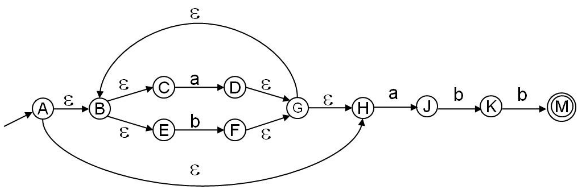
Step 4: Construct the DFA Table



	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
A	DJ	F
DJ	DJ	FK
F	DJ	F
FK	DJ	FM

Step 4: Construct the DFA 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
A	DJ	F
DJ	DJ	FK
F	DJ	F
FK	DJ	FM
FM	DJ	F

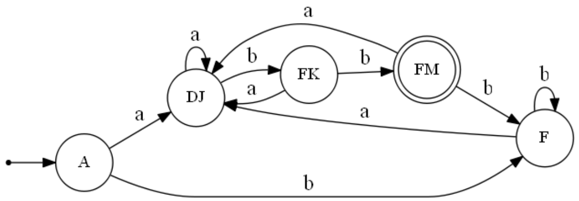
Step 4: Construct the DFA Table

	a	b
A	DJ	F
DJ	DJ	FK
F	DJ	F
FK	DJ	FM
FM	DJ	F

Step 4: Construct the DFA Table

	a	b
A	DJ	F
DJ	DJ	FK
F	DJ	F
FK	DJ	FM
FM	DJ	F

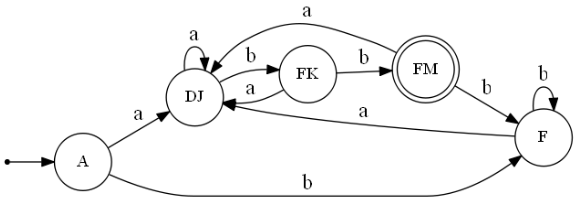
- Converted DFA state graph:



Step 4: Construct the DFA Table

	a	b
A	DJ	F
DJ	DJ	FK
F	DJ	F
FK	DJ	FM
FM	DJ	F

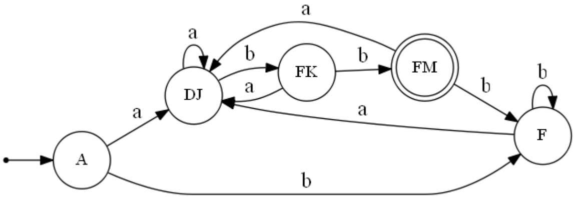
- Converted DFA state graph:



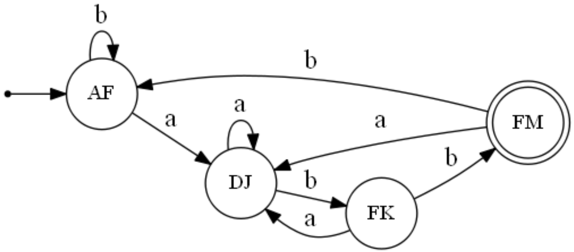
- Is the DFA minimal? (See Textbook 3.9.6: Minimization)
  - States A and F do not need differentiation ⇒ merge

Step 5: (Optional) Minimize DFA

- Original DF; before merging A and F:



- Minimized DFA; Do you see the original RE: (a|b)*abb?



NFA to DFA. Space Complexity

- An NFA may be in many states at any time
- How many different possible states in DFA?
  - If there are N states in NFA, the 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 where N is the number of original states (typically much fewer)

NFA to DFA Time Complexity

- DFA execution
  - Requires $O(|X|)$ steps, where $|X|$ is the length of input
  - Each step takes constant time:
    - If current state is S and input is c, then read T[S,c]
    - Update current state to state T[S,c]
  - Time complexity = $O(|X|)$
- NFA execution
  - Requires $O(|X|)$ steps, where $|X|$ is the length of input
  - Each step takes $O(N^2)$ time, where N is number of states:
    - Current state is a set of potential states, up to N
    - On input c, must union all T[$S_{potential}$,c], up to N times
    - Each union operation takes $O(N)$ time
  - Time complexity = $O(|X| * N^2)$

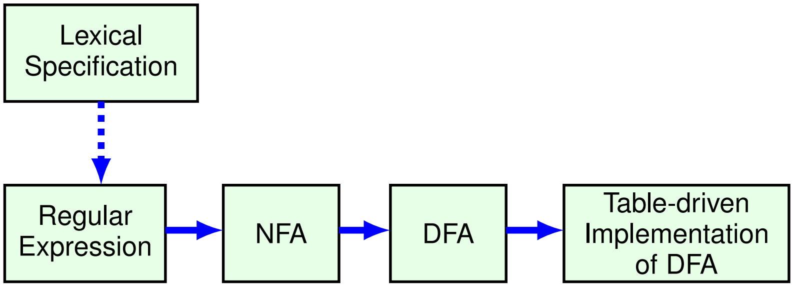
Implementation in Practice

- GNU **Lex**
  1. Converts regular expression to NFA
  2. Converts NFA to DFA
  3. Performs DFA state minimization to reduce space
  4. Generates transition table from DFA
  5. Performs table compression to further reduce space

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

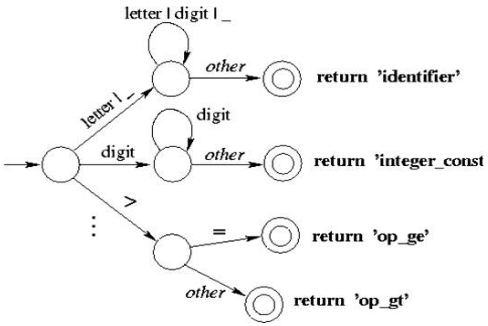
From RE to FA

- Our implementation sketch



Structure of a Scanner Automaton

- A scanner recognize multiple REs



How much should we match?

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

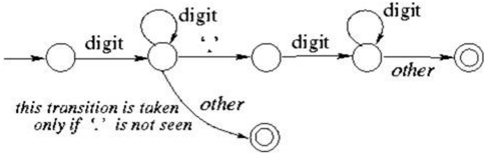
  Example:

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

      (numConst, 123.45)

    rather than

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



How to Match Keywords?

- Example: to recognize the following tokens

  Identifiers: letter(letter|digit)*
  Keywords: if, then, else
- Approach 1: Make REs for keywords and place them before REs for identifiers so that they will take precedence
  - Will result in more bloated finite state machine
- Approach 2: Recognize keywords and identifiers using same RE but differentiate using special keyword table
  - Will result in more streamlined finite state machine
  - But extra table lookup is required
- Usually approach 2 is more efficient than 1, but you can implement approach 1 in your projects for simplicity

Beyond Regular Languages

- Regular languages are expressive enough for tokens
  - Can express identifiers, strings, comments, etc.
- However, it is the weakest (least expressive) language
  - Many languages are not regular
    - C programming language is not
    - The language matching braces “{{{...}}}” is also not
  - Finite automata cannot count # of times char encountered
    - Crucial for analyzing languages with nested structures (e.g. nested for loop in C language)
- We need a more powerful language for parsing
  - In the next lecture, we will discuss **context-free languages**