



Informatik II

Languages, Compilers, and Theory

Lecture 2: Lexical Analysis



Lexical analysis

- Lexical analysis groups sequence of characters into **tokens**
- Tokens are the smallest “units of meaning” in the compilation process and are the foundation for parsing (syntax analysis)
- The compiler component that performs lexical analysis is called the **scanner** and is often automatically generated from a high-level specification



Lexical analysis example

A GCD program in C

```
int gcd (int i, int j) {
    while (i != j) {
        if (i > j)
            i = i - j;
        else
            j = j - i;
    }
    printf("%d\n", i);
}
```

Tokens

```
int      gcd      (
int      i        ,
int      j        )
{        while    (
i        !=       j
)        {        if
(        i        >
j        )        i
=        i        -
j        ;        else
j        =       j
-        i        ;
}        printf   (
"%d\n"   ,        I
)        ;        }
```



Two questions?

- How do we describe the lexical structure of a programming language?
 - In other words, what are the tokens?
- How do we implement the scanner once we know what the tokens are?

How do we describe lexical structure?



- Attempt 1:
 - List all of the tokens
 - if else long int short char ; , : () { } ...
 - But what about constants (integers, floating point numbers, strings)?
 - Can't list them all, there are ~8 billion 32-bit integer and floating point constants combined and an **infinite** number of string constants!
 - Same problem with identifiers (variable, function, and user-defined type names)
 - Solution: need some way to concisely describe classes of tokens which can assume a large number of distinct values

How do we describe lexical structure?

- Attempt 2:
 - Regular expressions
 - Patterns that can be used to match text
 - Defined recursively as one of:
 - A character
 - The empty string ϵ
 - The concatenation of two regular expressions
 - $r_1.r_2$ is the value of r_1 **followed by** the value of r_2
 - The alternation of two regular expressions
 - $r_1|r_2$ is the value of r_1 **or** the value of r_2
 - The Kleene $*$ closure of a regular expression
 - r^* is **zero or more occurrences** of the value of r
 - Parentheses may be used to group regular expressions when they are combined to avoid ambiguity
 - E.g., does $r_1.r_2|r_3$ mean $(r_1.r_2)|r_3$ or $r_1.(r_2|r_3)$?

A few notes on regular expressions

- Shortcuts
 - Regular expressions may be defined as strings
 - $r = 'c_1c_2\dots c_n'$ is equivalent to
 - $r = c_1.c_2\dots.c_n$
 - Regular expressions may be defined as a range of characters
 - $r = [c_1-c_n]$ is equivalent to
 - $r = c_1|c_2|\dots|c_n$ for the contiguous sequence of n characters beginning with c_1 and ending with c_n
 - Example: $r = [a-d]$ is equivalent to $r = a|b|c|d$
 - The Kleene + closure of a regular expression r is **one or more occurrences** of the value of r and is formally defined as
 - $r^+ = r.r^*$
 - The symbol $.$ stands for any character other than newline

Lexical analysis: Regular expressions at work

A GCD program in C

```
int gcd (int i, int j) {
    while (i != j) {
        if (i > j)
            i = i - j;
        else
            j = j - i;
    }
    printf ("%d\n", i);
}
```

Regular expressions

- digit=[0-9]
- letter=[a-z]|[A-Z]
- punct=\|%
- INT='int'
- WHILE='while'
- IF='if'
- ID=letter.(letter|digit)*
- LPAREN=(
- RPAREN=)
- COMMA=,
- SEMI=;
- LBRACE={
- RBRACE=}
- EQ==
- MINUS=-
- GT=>
- SC=".(letter|digit|punct)*."

Tokens

INT	ID: gcd	LPAREN
INT	ID: i	COMMA
INT	ID: j	RPAREN
LBRACE	WHILE	LPAREN
ID: i	NEQ	ID: j
RPAREN	LBRACE	IF
LPAREN	ID: i	GT
ID: j	RPAREN	ID: i
EQ	ID: i	MINUS
ID: j	SEMI	ELSE
ID: j	EQ	ID: j
MINUS	ID: i	SEMI
RBRACE	ID: printf	LPAREN
SC: "%d\n"	COMMA	ID: i
RPAREN	SEMI	RBRACE

A closer look at lexical analysis

- How does the lexical analyzer handle white space?
 - `int gcd (int i, int j) {`
- How does the lexical analyzer handle programming language comments?
 - `/* gcd */ int gcd (int i, int j) {`
- How does the lexical analyzer handle conflicting regular expressions?
 - Given:
 - `WHILE='while'`
 - `ID=letter.(letter|digit)*`
 - Both regular expressions match the string "while", so which do we choose?



Handling white space

- White space can be recognized as a token with the following rule
 - $WS = (\backslash n | \backslash r | \backslash t | \backslash s)^*$
 - $\backslash n$ is an escape sequence for newline
 - $\backslash r$ is an escape sequence for carriage return
 - $\backslash t$ is an escape sequence for tab
 - $\backslash s$ is an escape sequence for space
- The white space token WS is usually unimportant to the syntax of programming languages, so it can just be deleted from the token stream



Handling comments

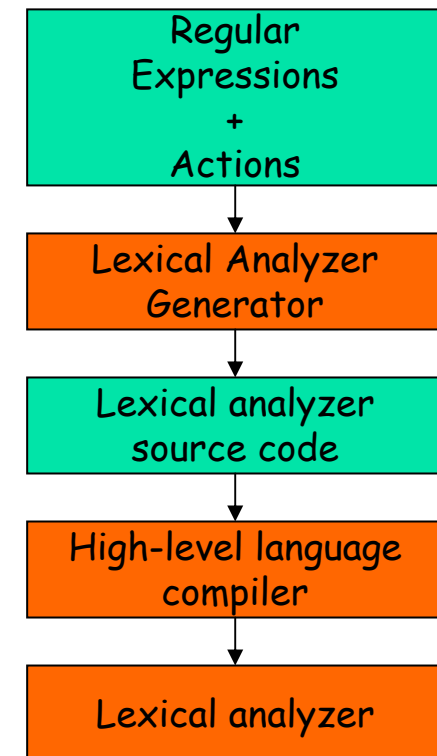
- Alternative 1: preprocessors
 - Special program that reads a file, deletes comments, performs other operations (such as macro expansion), and writes an output file for the lexical analyzer to read
- Alternative 2: comment tokens
 - Depending on the complexity of comments, we may be able to describe them using regular expressions
 - Single line comments can be matched with regular expressions
 - `SLC=//.*$`
 - `$` is a special symbol denoting the end of a line
 - Matches text like:
 - `// This is a comment`
 - Some comments are too complex to be matched with regular expressions
 - Arbitrarily nested comments
 - `/* level 1 /* level 2 */ back to level 1 */`
 - Usually handled by preprocessor

Handling conflicting regular expressions

- Given two regular expressions r_1 and r_2 which match a partial input $p='c1..ck'$, which expression do we select?
- Alternative 1: Longest match
 - Continue adding characters from the input stream to p until neither r_1 nor r_2 match. Remove one character and either r_1 or r_2 must match. The partial input p is the longest match, and if only one of r_1 or r_2 matches, select it
 - Example
 - $r_1='while'$
 - $r_2=letter.(letter|digit)^*$
 - Input
 - `int while48; ...`
 - When $p='while'$, both r_1 and r_2 match
 - When $p='while48;'$ neither r_1 nor r_2 match
 - When $p='while48'$ only r_2 matches, so select r_2
- Alternative 2: Rule priority
 - If longest match still results in a conflict, select the first regular expression listed in lexical definition of the language

Implementation of lexical analyzers

- How do we turn regular expressions into a lexical analyzer?
 - Convert regular expressions to a deterministic finite automata (DFA)
 - Why? DFAs are easier to simulate than regular expressions.
 - Write a program to simulate the DFA
 - The DFA **recognizes** tokens in the input text and becomes the lexical analyzer
 - When a token is recognized, a user defined action may take place.
 - For example, checking to make sure that the value of an integer constant will fit into 32-bits.
 - Conversion of regular expressions to a DFA, and writing the program to simulate the DFA can be done either **by hand**, or by another program called a **lexical analyzer generator**.



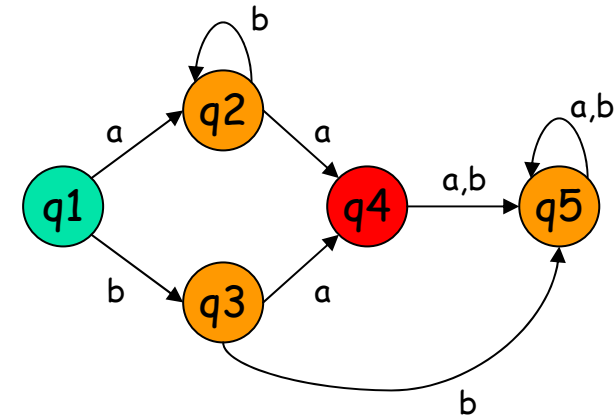


Finite Automata

- Formally, a finite automaton M is a 5-tuple, $M=(Q,\Sigma,q,F,\delta)$, where:
 - Q is a set of symbols called **states**
 - Σ is a set of symbols called **the alphabet**
 - q is the **start state**
 - F is a possibly empty set of **final states**
 - δ is a **transition function**
- $L(M)$, or the **language of M** , is the set of finite strings of symbols from the alphabet Σ which are accepted by M

An example finite automaton

- $Q = \{q_1, q_2, q_3, q_4, q_5\}$
- $\Sigma = \{a, b\}$
- $q = q_1$
- $F = \{q_4\}$
- $\delta =$
 - $\{(q_1, a), q_2\}, \{(q_1, b), q_3\},$
 - $\{(q_2, a), q_4\}, \{(q_2, b), q_2\},$
 - $\{(q_3, a), q_4\}, \{(q_3, b), q_5\},$
 - $\{(q_4, a), q_5\}, \{(q_4, b), q_5\},$
 - $\{(q_5, a), q_5\}, \{(q_5, b), q_5\}\}$



Input:

abba a **Not accepted!**



What language does M accept?

$(ab^*a) \mid ba$

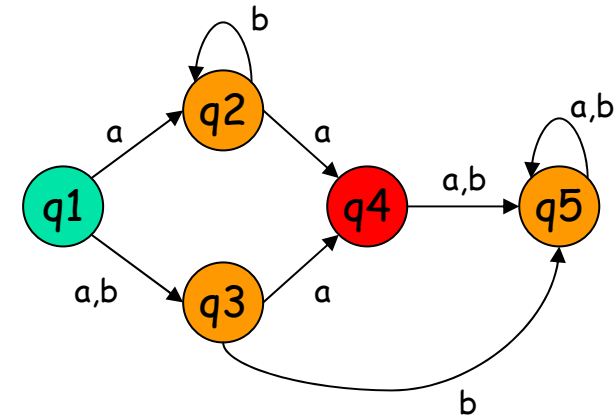


Two types of finite automata: Deterministic and nondeterministic

- Deterministic finite automata (DFA)
 - Transition function defined formally as
 - $\delta: Q \times \Sigma \rightarrow Q$
 - Given an input symbol and a state, produces a **single next state**
- Nondeterministic finite automata (NFA)
 - Transition function defined formally as
 - $\delta: Q \times \Sigma \rightarrow 2^Q$
 - Given an input symbol and a state, produces a possibly empty **set of possible next states**
- Other than transition functions DFAs and NFAs are the same

An example NFA

- $Q = \{q_1, q_2, q_3, q_4, q_5\}$
- $\Sigma = \{a, b\}$
- $q = q_1$
- $F = \{q_4\}$
- $\delta =$
 - $\{(q_1, a), \{q_2, q_3\}\}, \{(q_1, b), \{q_3\}\},$
 - $\{(q_2, a), \{q_4\}\}, \{(q_2, b), \{q_2\}\},$
 - $\{(q_3, a), \{q_4\}\}, \{(q_3, b), \{q_5\}\},$
 - $\{(q_4, a), \{q_5\}\}, \{(q_4, b), \{q_5\}\},$
 - $\{(q_5, a), \{q_5\}\}, \{(q_5, b), \{q_5\}\}$



Input:

abba a **Not accepted!**



What language does M accept?

$(ab^*a) \mid ba$

Same as before with DFA...



An interesting thing about finite automata

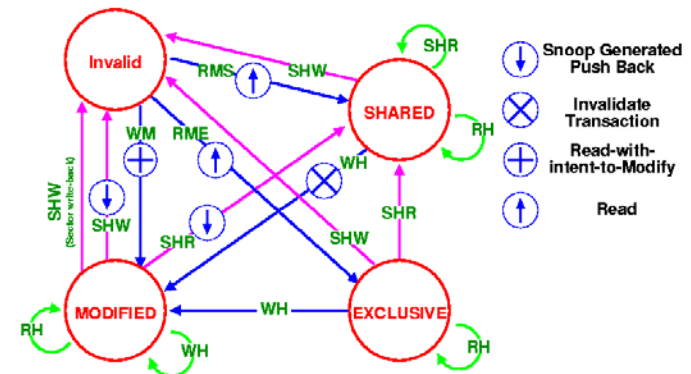
- Even though it may seem as if nondeterminism might give a finite automaton more power, NFAs and DFAs are formally equivalent
 - Any NFA can be converted into a DFA and vice versa
- Why make the distinction?
 - Easier to convert regular expressions to NFAs
 - Easier to simulate DFAs

An aside on finite automata

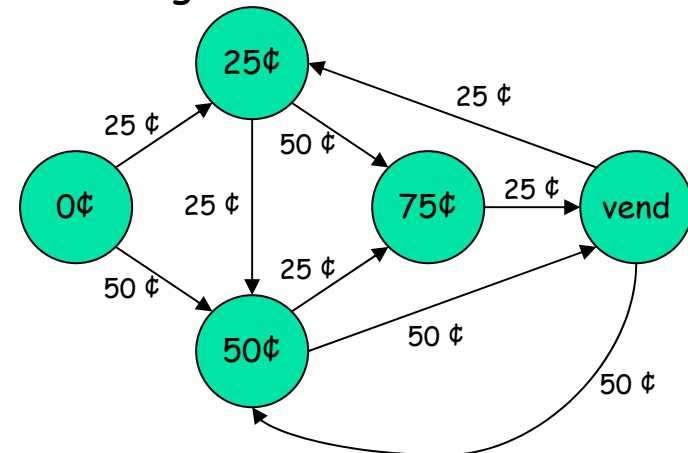
- Finite automata are useful for things other than lexical analysis
 - Most systems which makes **transitions** between a finite number of **states** can be modeled with finite automata
 - Example uses:
 - Describing, simulating, verifying and implementing protocols
 - Building fast, stateful circuits

MESI cache coherence protocol

(Courtesy: John Morris, University of Western Australia)



Vending machine automata

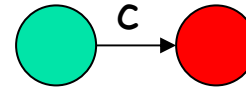


Building the lexical analyzer: Regular expressions to NFAs

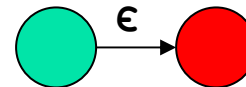
Regular Expression

NFA

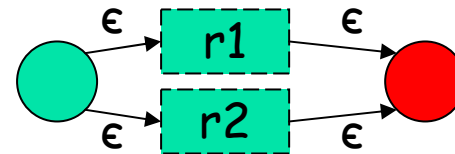
Character: c



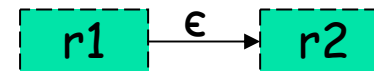
Empty string: ϵ



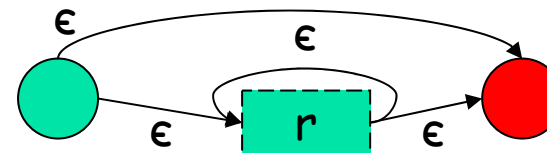
Alternation: $r1|r2$



Concatenation: $r1.r2$



Kleene Closure: r^*



Regular expressions to NFAs: An Example

- $r = ('ab'|'cd')^*$

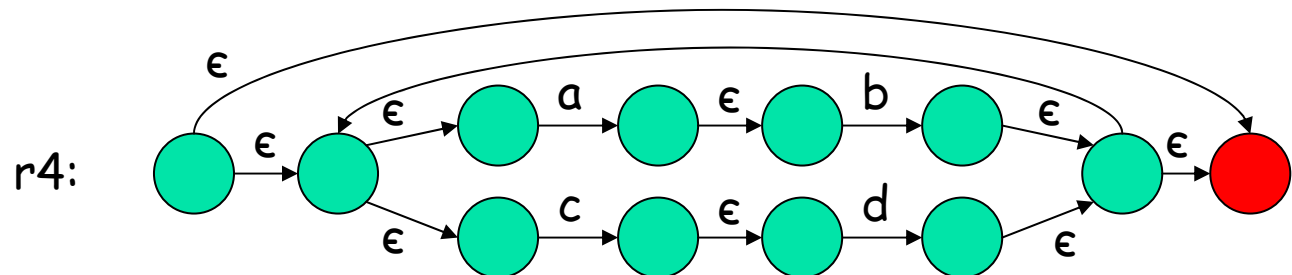
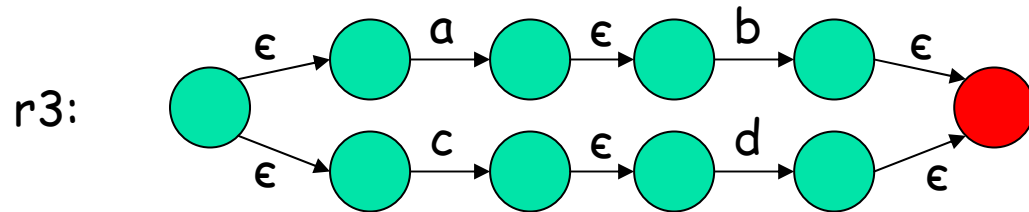
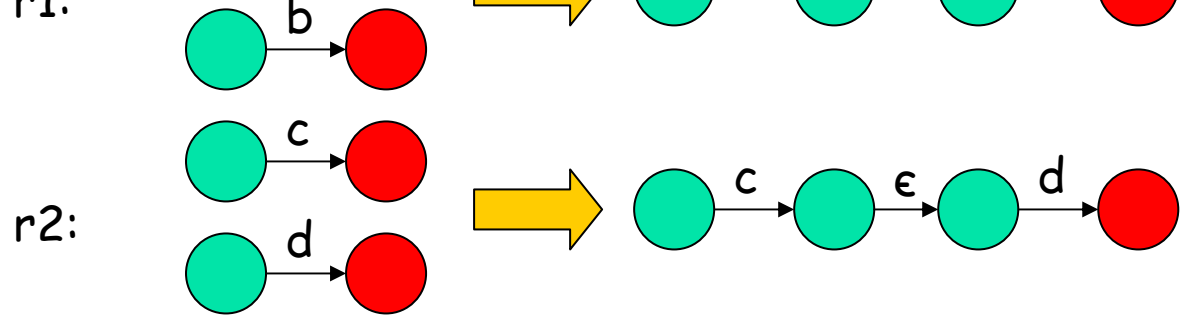
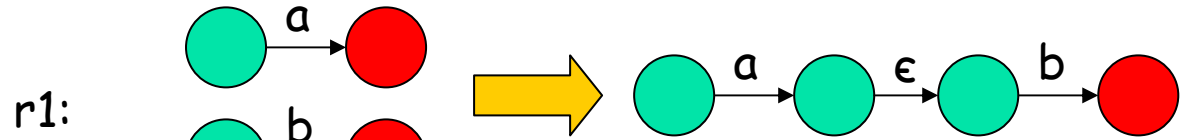
- Factor:

- $r1 = a.b$

- $r2 = c.d$

- $r3 = r1|r2$

- $r = r3^*$



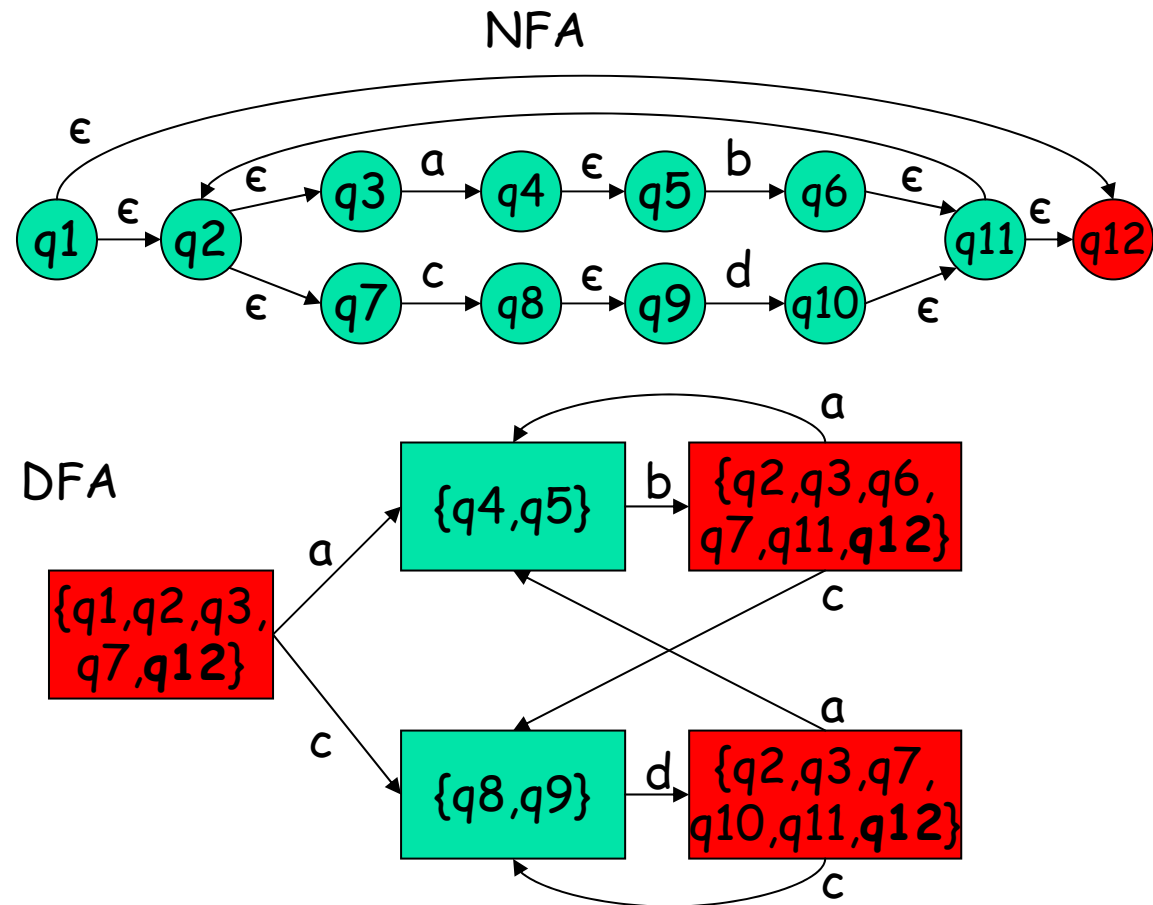


Theoretical result #1

- Definitions:
 - Finite automata **accept** or **recognize** languages.
 - Regular expressions **generate** languages.
 - $L(M)$ is the language accepted by the finite automaton M .
 - $L(R)$ is the language generated by regular expression R .
 - $L(R)$ and $L(M)$ are sets of finite strings of symbols from alphabets Σ_R and Σ_M
 - L_R is the set $\{L(r): \text{all regular expressions } r\}$
 - L_N is the set $\{L(n): \text{all nondeterministic finite automata } n\}$
- L_R is a subset of L_N
 - Nondeterministic finite automata accept all languages generated by regular expressions
 - Why? We've shown how to convert any regular expression to an NFA
- Are L_R and L_N the same sets?
 - Turns out the answer is **yes**
 - Prove by showing that L_N is a subset of L_R or that any nondeterministic finite automata can be converted to a regular expression
 - See any good theory of computation book for the details:
 - *Introduction to Automata Theory, Languages, and Computation* by Hopcroft and Ullman
 - *Introduction to the Theory of Computation* by Michael Sipser

Building the lexical analyzer: NFAs to DFAs

- Regular expression
 - $r=(ab'|cd')^*$
- Convert NFA to DFA using subset construction
- Label each DFA state as the set of states reachable from the previous state in one step
- If any NFA state in the set of reachable states is a final state, then the entire DFA state is final





Theoretical result #2

- Definitions:
 - Recall, L_N is the set $\{L(n): \text{all nondeterministic finite automata } n\}$
 - L_D is the set $\{L(d): \text{all deterministic finite automata } d\}$
- L_N is a subset of L_D and L_R is a subset of L_D
 - Deterministic finite automata accept all languages accepted by nondeterministic finite automata
 - By transitivity, deterministic finite automata also accept all languages generated by regular expressions
 - Why? We've shown how to convert any NFA to a DFA and any regular expression to an NFA
- Are L_D and L_N the same sets?
 - Turns out the answer is **yes**
 - Prove by showing that L_D is a subset of L_N or that any deterministic finite automata can be converted to a nondeterministic finite automata
 - Again, see any good theory of computation book for the details



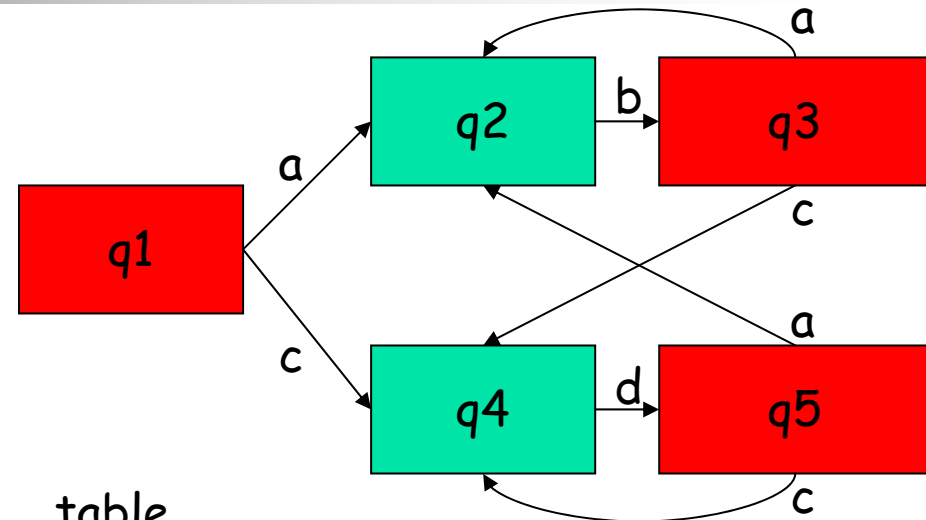
A Note on DFAs

- A DFA M built using the subset construction may not be **minimal**
 - In other words there may be an automaton M' where $L(M)=L(M')$ where M' has fewer states than M
- Minimal DFAs are better for implementation purposes
 - Fewer states requires less memory and generally lead to faster simulation
- Most automatic tools which convert NFAs to DFAs perform an **optimization** pass to reduce the number of DFA states
 - Finding a minimal DFA is a very hard problem (called NP-complete), so optimizers usually can't guarantee a minimal DFA
 - That's o.k. though—the fewer states the better!

Building the lexical analyzer: DFAs to code

DFAs can be efficiently simulated using a table based algorithm

```
void dfa (char *in) {
  s = in;
  state = start_state;
  while(1) {
    c = *s++;
    state = table[state][c];
    if (final[state]) {
      printf("Accepted %s\n", in);
      break;
    }
  }
}
```



table

	a	b	c	d
q1	q2	q6	q6	q6
q2	q6	q3	q6	q6
q3	q2	q6	q4	q6
q4	q6	q6	q6	q5
q5	q2	q6	q4	q6
q6	q6	q6	q6	q6

Building the lexical analyzer: Final steps



- DFA simulation code becomes the core of the lexical analyzer
- When DFA is in a final state
 - Execute user action code associated with the last matching regular expression rule according to **longest match** and/or **rule priority**
 - Remember current location in input stream, and return token to the token consumer (parser)

A real JLex lexical specification for a calculator language

```
1
2 import java.io.IOException;
3
4 %%
5
6 %public
7 %class Scanner
8
9 %type void
10 %eofval{
11     return;
12 %eofval}
13 %{
14     public static void main (String args []) {
15         Scanner scanner = new Scanner(System.in);
16         try {
17             scanner.yylex();
18         } catch (IOException e) { System.err.println(e); }
19     }
20
21     comment = ("#".*)
22     space = [\\ \\t\\b\\015]+
23     digit = [0-9]
24     integer = {digit}+
25     real = ({digit}+"."{|{digit}*|{digit}*"}{|{digit}+})
26     IF = ("if")
27     THEN = "then"
28     ELSE = else
29
30 %%
31
32 {space}          { System.out.println("space");
33     break;
34 }
35 {comment}        { System.out.println("comment");
36     break;
37 }
38 {integer}         { System.out.println("Integer CONSTANT\t" + yytext());
39     break;
40
41 {real}           { System.out.println("REAL    CONSTANT\t" + yytext());
42     break;
43 }
44 {IF}             { System.out.println("IF      Token\t" + yytext());
45     break;
46 }
47 {THEN}           { System.out.println("THEN   Token\t" + yytext());
48     break;
49 }
50 {ELSE}           { System.out.println("ELSE   Token\t" + yytext());
51     break;
52 }
53 \n               { System.out.println("NL");
54     break;
55 }
56 "+"             { System.out.println("ADD");
57     break;
58 }
59 "-"             { System.out.println("SUB");
60     break;
61 }
62 "*"             { System.out.println("MUL");
63     break;
64 }
65 "/"             { System.out.println("DIV");
66     break;
67 }
68 "%"             { System.out.println("MOD");
69     break;
70 }
71 "("             { System.out.println("LPAR");
72     break;
73 }
74 ")"             { System.out.println("RPAR");
75     break;
76 }
77 .               { System.out.println("error" + "+" + yytext() + "+");
78     break;
79 }
```

Roadmap of remaining lectures



- Next lecture (Chapter 2 from text)
 - Syntactic analysis
 - More automata theory and automatic generation of parsers
- Final five lectures
 - Names, scopes and binding (Chapter 3)
 - Control flow (Chapter 6)
 - Subroutines and control abstraction (Chapter 8)
 - Building a runnable program (Chapter 9)
 - Object oriented programming (Chapter 10)