Lexical Analysis
What is Lexical Analysis

- What do we want to do?
  
  Example:
  
  ```java
  if (i==j)
      z = 0;
  else
      z = 1;
  ```

- The input is just a string of characters
  
  "if(i = j)\n  tz = 0; \ntelse\n  tz = 1; \n"

- Goal: partition input string into substrings
  
  ➢ where these substrings are **tokens**
What is a Token?

- Token: a "word" in the given language
  - Categorized into classes according to its role in language
  - In English:
    - noun, verb, adjective, ...
  - In a programming language:
    - identifier, integer, keyword, whitespace, ...

- Token classes comprise distinct sets of strings
  - Identifier: strings of letters and digits, starting with a letter
  - Integer: a non-empty string of digits
  - Keyword: “else”, “if”, “while”, ...
  - Whitespace: a non-empty sequence of blanks, newlines, and tabs
What are Tokens For?

- Classify program substrings according to role
- Output of lexical analysis is a stream of tokens
- Tokens are the input to the Parser
  - Parser relies on token classes to identify roles
    - a keyword is treated differently than an identifier
- Hence, lexical analysis is also called tokenization
Designing a Lexical Analyzer

- **Step 1:**
  - Define a finite set of token classes
    - Describe all items of interest
    - Depend on language, design of parser
    - Recall: "if \((i = j)\) \(nt = 0\); else \(nt = 1\); \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 \((\text{class, lexeme})\)
  
  \[
  \text{``if}(i = j)\backslash n\text{t}\text{z} = 0; \backslash \text{else}\text{n}\text{t}\text{z} = 1; \backslash n``
  \]

  - Result of lexical analysis:
    - (keyword, if) (leftparen, () (id, i) (equals, =) (id, j) (rightparen, ))
    - (id, z) (equals, =) (int, 0) (semicolon, ;)
    - (keyword, else) (id, z) (equals, =) (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
  - rule was motivated from the inaccuracy of card punching by operators

- Consider
  - DO 5I=1,25
  - DO 5I=1.25

- We have
  - The first: a loop iterates from 1 to 25 with step 5
  - The second: an assignment

- Reading left-to-right, cannot tell if DO5I is a variable or DO statement; Have to continue until “,” or “.” is reached.
The problem is not only limited to Fortran.

C++ template syntax

```cpp
FOO<Bar>
```

C++ stream syntax

```cpp
cin >> var
```

Now, the problem

```cpp
FOO<Bar<Bazz>>
```

Is the “>>” a stream operator or just two consecutive brackets?
Lesson Learned

- Observations from examples:
  - “lookahead” or “lookbehind” may be required to decide among different choices of tokens
  - Extracting some tokens require 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 without parser feedback
  - Allows clean division between lexical and syntactic analyses
Question: how can tokens in a language be described formally

- Define what token classes there are
- Define what strings belong to which class

Answer: adopt Regular Language formalism

- Simple yet powerful (able to express patterns)
- An implementation can be generated automatically from formalism
- Resulting implementation is provably efficient
Definition

Let \( \Sigma \) be a set of characters, a **language** over \( \Sigma \) is a set of strings of the characters drawn from \( \Sigma \).
Examples of Languages

- Alphabet $\sum = $ English characters
  Language $L = $ English sentences

- Alphabet $\sum = $ Digits, +, -
  Language $L = $ Integer numbers

- Are typically **subsets of all possible strings**
  - Not all strings of English characters are sentences
  - Not all sequences of digits and signs are integers

- Need some notation to specify that language (subset)
  - A notation for languages is **regular expressions**
    - Can express simple patterns (e.g. repeating sequences)
    - Natural fit to express tokens such as identifiers
  - Languages that can be expressed using regular expressions are called **regular languages**
Atomic Regular Expressions

- Single character denotes a set of one string
  \['c' = \{ "c" \}\]

- *Epsilon* or \(\epsilon\) character denotes a zero length string \(\epsilon = \{ \"\" \}\)

- Empty set is \(\{ \} = \phi\), not the same as \(\epsilon\)
  - \(\text{size}(\phi) = 0\)
  - \(\text{size}(\epsilon) = 1\)
  - \(\text{length}(\epsilon) = 0\)
Drilldowns:

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

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

- **Iteration (Kleene closure)**
  $$A^* = \bigcup_{i \geq 0} A^i \quad \text{where } A^i = A \cdots A \ (i \text{ times})$$
  In particular:
  $$A^* = \{ \varepsilon \} + A + AA + AAA + \ldots$$
  $$A^+ = A + AA + AAA + \ldots = A \ A^*$$
Regular Expressions

Definition

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

- $\varepsilon$
- ‘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 $\sum$
This notation means

- \( L(\varepsilon) = \{ "" \} \)
- \( L(\text{`c'}) = \{ "c" \} \)
- \( L(A+B) = L(A) \cup L(B) \)
- \( L(AB) = \{ ab \mid a \in L(A) \text{ and } b \in L(B) \} \)
- \( L(A^*) = \bigcup_{i \geq 0} L(A^i) \)
Keywords: “else” or “if” or “while” or ...

Q: is '000' an integer?

Q: how to define another integer RE that excludes sequences with leading 0s?
Examples

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

- ‘else’ + ‘if’ + ‘while’ + ...
- ‘else’ abbreviates
  - ‘e’ (concatenate) ‘l’ (concatenate) ‘s’ (concatenate) ‘e’

Q: is ‘000’ an integer?
Q: how to define another integer RE that excludes sequences with leading 0s?
Keywords: “else” or “if” or “while” or ...

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

Keywords: “else” or “if” or “while” or ...
  ➢ ‘else’ + ‘if’ + ‘while’ + ...
  ➢ ‘else’ abbreviates
    ➢ ‘e’ (concatenate) ‘l’ (concatenate) ‘s’ (concatenate) ‘e’
  ➢ keywords = { ‘else’, ‘if’, ‘then’, ‘while’, ... }

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

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

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

Integer

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

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

- ‘else’ + ‘if’ + ‘while’ + ...
- ‘else’ abbreviates
  - ‘e’ (concatenate) ‘l’ (concatenate) ‘s’ (concatenate) ‘e’

- Keywords = { ‘else’, ‘if’, ‘then’, ‘while’, ... }

_integer

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

- Q: is ‘000’ an integer?
- Q: how to define another integer RE that excludes sequences with leading 0s?
More Examples

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

- Whitespace: a non-empty sequence of blanks, newlines and tabs
  - whitespace = ( ‘ ’ + ‘\n’ + ‘\t’ ) +
More Examples

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

- **Whitespace:** a non-empty sequence of blanks, newlines and tabs
  - whitespace = ( ‘ ’ + ‘\n’ + ‘\t’) +
More Examples

Phones number: consider (412) 624-0000

- $\sum = \text{digit} \cup \{ -, (, ) \}$
- area = digit
- exchange = digit
- phone = digit
- phoneNumber = '(' area ')' exchange '-' phone

Email address: student @ pitt.edu

- $\sum = \text{letter} \cup \{ ., @ \}$
- name = letter
- emailAddress = name '@' name '.' name
RE used in languages

- By itself, it is a string, but semantically gets interpreted as a RE
- RE in PERL,
  
  ```perl
  if ($str =~ /\(\d+/) ) ...
  ```

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

- RE in C#,
  ```csharp
  Match m = Regex.Match("abrabceeaab", "(a|b|r)+");
  ```
# Some Common REs in Programming Languages

<table>
<thead>
<tr>
<th>Meaning</th>
<th>Meaning</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>\d</td>
<td>Digits</td>
<td>\w</td>
</tr>
<tr>
<td>\D</td>
<td>Non-digits</td>
<td>\W</td>
</tr>
<tr>
<td>[a-f]</td>
<td>Char range</td>
<td>[^a-f]</td>
</tr>
<tr>
<td>?</td>
<td>Optional</td>
<td>{n,m}</td>
</tr>
<tr>
<td>.</td>
<td>Any char</td>
<td>(...)</td>
</tr>
<tr>
<td>.</td>
<td>Matching “.”</td>
<td>+</td>
</tr>
<tr>
<td></td>
<td>Matching string start</td>
<td>^</td>
</tr>
<tr>
<td></td>
<td>Matching string end</td>
<td>$</td>
</tr>
<tr>
<td></td>
<td>Matching (, { ...</td>
<td>(),{</td>
</tr>
<tr>
<td></td>
<td>Appear 0 or many times</td>
<td>*</td>
</tr>
</tbody>
</table>
Implementation of Lexical Analysis
We have learnt the formalism for lexical analysis
— Regular expression (RE)

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

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

How to actually get the lexical analyzer?

Solution 1: to implement using a tool — Lex (for C), Flex (for C++), Jlex (for java)

- Programmer specifies tokens using RE formalism
- The tool generates the source code from the given REs
We have learnt the formalism for lexical analysis — Regular expression (RE)

How to actually get the lexical analyzer?

Solution 1: to implement using a tool — Lex (for C), Flex (for C++), Jflex (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
  - Writing REs instead of the code itself
  - Writing actions associated with each RE

- The specification file (e.g. abc.l), has well-defined format

- Detailed implementation of the tool itself will be discussed later
Lex Specifications

```c
{% /* include, extern, etc. */
extern int yytext, yylineno;
#include "token.h"
%
/* pattern definitions. format: name + definition */
digit [0-9]
number [0-9]+
%%
/* rules. format: pattern + action */
number {
    printf("Token: int const \%s and \%d", yytext, yylineno);
    return ICONSTNUM;
}
%%
/* auxiliary user code */
int myTableInsert() { ... }
```
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
- `yyline`, `yycolumn` maintained by you
- `yytext`, `yyleng` maintained by lex library
- Special characters
  - `'\n'` — newline
  - `'\t'` — tab
  - `'\''` — single quote
  - `'\\'` — backslash
Discussion of RE and Lexical Analysis

- Lexer uses RE to extract tokens from input string
- Lexer always tries to consume as many characters as possible in a single token when given a choice
  - Given string “if”, it chooses token (keyword, if) over two tokens (id, i) (id, f)
  - Given two tokens of same length, patterns are chosen in precedence they appear in specification file

- Regular Expressions is only a language specification
  - An implementation is still needed
  - How does Lex translate a specification file to C code?

- The problem we face is 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

- Lexical Specification

- Regular Expression

- NFA

- DFA

- Table-driven Implementation of DFA
An Overview of RE to FA

Our implementation sketch

- Lexical Specification
- Regular Expression
- NFA
- DFA
- Table-driven Implementation of DFA

Conversion:
- manual conversion
- auto conversion
Implementation Outline

- RE $\rightarrow$ NFA $\rightarrow$ DFA $\rightarrow$ Table-driven Implementation
  - Specifying lexical structure using regular expressions
  - Finite automata
    - Deterministic Finite Automata (DFAs)
    - Non-deterministic Finite Automata (NFAs)
  - Table implementations
In the following discussion, we use some alternative notations

Union: \( A | B \equiv A + B \)

Option: \( A | \epsilon \equiv A ? \)

Range: \( 'a' + 'b' + ... + 'z' \equiv [a-z] \)

Excluded range: complement of [a-z]\( \equiv [^a-z] \)
Finite Automata

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

- For lexical analysis
  - Specification — Regular expression
  - Implementation — Finite automata
More About Transition

- Transition $\delta: S_a \overset{\text{input}}{\rightarrow} S_b$
  - read as
  in state S1 on input “a” go to state S2

- At the end of input (or no transition possible), if current state $X$
  - $X \in \text{accepting set } F$, then $\Rightarrow \text{accept}$
  - otherwise, $\Rightarrow \text{reject}$
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
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 automata that accepts only “1”
A finite automata accepting any number of 1s followed by a single 0. Here we have Alphabet = \{0,1\}

```
1

0

0
```
More Examples

A finite automata accepting any number of 1s 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\}
A finite automata accepting any number of 1s 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\}
Given the state graph of a DFA,
Given the state graph of a DFA,
Table Implementation of a DFA

Given the state graph of a DFA,

\[
\begin{array}{ccc}
\text{state} & \text{0} & \text{1} \\
S & T & U \\
T & T & U \\
U & T & X \\
\end{array}
\]

input characters

\[
\begin{array}{ccc}
& 0 & 1 \\
S & T & U \\
T & T & U \\
U & T & X \\
\end{array}
\]
Table Implementation of a DFA

Given the state graph of a DFA,

Table-driven Code:

```c
DFA() {
    state = "S";
    while (!done) {
        ch = fetch_input();
        state = Table[state][ch];
        if (state == "x")
            perror("reject");
    }
    if (state ∈ F)
        printf("accept");
    else
        printf("reject");
}
```
Each RE has a different DFA / state graph

For different REs,
- their tables are different
- their DFA recognition code is the same
- Lex tool need only generate new table for new DFA

Finite automata need only finite memory
- Need only to encode the set of states in a table

Finite automata need only a finite number of transitions
- Number of transitions == number of input characters
Each RE has a different DFA / state graph

For different REs,
- their tables are different
- their DFA recognition code is the same
- Lex tool need only generate new table for new DFA

Finite automata need only finite memory
- Need only to encode the set of states in a table

Finite automata need only a finite number of transitions
- Number of transitions == number of input characters

Revisit our implementation outline

RE $\rightarrow$ NFA $\rightarrow$ DFA $\rightarrow$ Table-driven Implementation
Our implementation sketch

Lexical Specification

Regular Expression → NFA → DFA → Table-driven Implementation of DFA
Another kind of transition: \( \varepsilon \)-moves

- Machine can move from state A to state B without reading any input
Deterministic and Nondeterministic Automata

- Deterministic Finite Automata (DFA)
  - One transition per input per state
  - No $\varepsilon$-moves

- Non-deterministic Finite Automata (NFA)
  - Can have multiple transitions for one input in a given state
  - Can have $\varepsilon$-moves
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 $\varepsilon$-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
  - $\varepsilon$ expression
    - $\varepsilon$ expression
  - single character RE $a$
    - $a$ in the NFA
Step 2: processing compound REs

- $r = s \mid t$
- $r = s \cdot t$
- $r = s^*$
Step 2: processing compound REs

- $r = s \mid t$
- $r = s \cdot t$
- $r = s^*$
Step 2: processing compound REs

- \( r = s \mid t \)

- \( r = s \cdot t \)

- \( r = s^* \)
Step 2: processing compound REs

- $r = s \mid t$
  - $NFA(s)$
  - $NFA(t)$

- $r = s \cdot t$
  - $NFA(s)$
  - $NFA(t)$

- $r = s^*$
Step 2: processing compound REs

- $r = s \mid t$
- $r = s \cdot t$
- $r = s^*$
Converting RE to NFA (cont.)

Step 2: processing compound REs

- \( r = s | t \)

- \( r = s \cdot t \)

- \( r = s^* \)
Step 2: processing compound REs

- \( r = s \mid t \)
- \( r = s \cdot t \)
- \( r = s^* \)
Convert \((a|b)^*a\ b\ b\) to NFA
Convert “(a|b)*a b b” to NFA
From RE to FA

Our implementation sketch

Lexical Specification

Regular Expression → NFA → DFA → Table-driven Implementation of DFA
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 $== \text{length of input}$
  - 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
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 $\varepsilon$-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 $\varepsilon$-moves as well
Example NFA to DFA

What is the Equivalent DFA?
Example NFA to DFA

What is the Equivalent DFA?

What is the Equivalent DFA?

state ↓ → input characters

<table>
<thead>
<tr>
<th>state</th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td></td>
<td></td>
</tr>
<tr>
<td>B</td>
<td></td>
<td></td>
</tr>
<tr>
<td>C</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Example NFA to DFA

What is the Equivalent DFA?

State ↓ → Input Characters

<table>
<thead>
<tr>
<th>state</th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>A</td>
<td>BC</td>
</tr>
<tr>
<td>B</td>
<td>AC</td>
<td>B</td>
</tr>
<tr>
<td>C</td>
<td>C</td>
<td>C</td>
</tr>
</tbody>
</table>

Diagram:

```
0 → A (0, 1) → B → C (1, 0) → A
```

Input characters: 0, 1
Example NFA to DFA

What is the Equivalent DFA?

```

<table>
<thead>
<tr>
<th>State</th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>A</td>
<td>BC</td>
</tr>
<tr>
<td>B</td>
<td>AC</td>
<td>B</td>
</tr>
<tr>
<td>C</td>
<td>C</td>
<td>C</td>
</tr>
</tbody>
</table>
```

Diagram of the NFA:

- **A**
  - 0: to **A**
  - 1: to **B**
- **B**
  - 0: to **A**
  - 1: to **C**
- **C**
  - 1/0: to **B**

```
Example NFA to DFA

What is the Equivalent DFA?

State ↓ → input characters

<table>
<thead>
<tr>
<th>state</th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>A</td>
<td>BC</td>
</tr>
<tr>
<td>B</td>
<td>AC</td>
<td>B</td>
</tr>
<tr>
<td>C</td>
<td>C</td>
<td>C</td>
</tr>
<tr>
<td>AC</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BC</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
What is the Equivalent DFA?

```
state ↓   → input characters

<table>
<thead>
<tr>
<th>state</th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>A</td>
<td>BC</td>
</tr>
<tr>
<td>B</td>
<td>AC</td>
<td>B</td>
</tr>
<tr>
<td>C</td>
<td>C</td>
<td>C</td>
</tr>
<tr>
<td>AC</td>
<td>AC</td>
<td>BC</td>
</tr>
<tr>
<td>BC</td>
<td>AC</td>
<td>BC</td>
</tr>
</tbody>
</table>
```
Example NFA to DFA

What is the Equivalent DFA?

Diagram:

State transition table:

<table>
<thead>
<tr>
<th>State</th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>A</td>
<td>BC</td>
</tr>
<tr>
<td>B</td>
<td>AC</td>
<td>B</td>
</tr>
<tr>
<td>C</td>
<td>C</td>
<td>C</td>
</tr>
<tr>
<td>AC</td>
<td>AC</td>
<td>BC</td>
</tr>
<tr>
<td>BC</td>
<td>AC</td>
<td>BC</td>
</tr>
<tr>
<td>AB</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>ABC</td>
<td>x</td>
<td>x</td>
</tr>
</tbody>
</table>
Algorithm Illustrated: Converting NFA to DFA

<table>
<thead>
<tr>
<th></th>
<th>ε</th>
<th>a</th>
<th>b</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>C</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>D</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>E</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>F</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>G</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>H</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>J</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>K</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>M</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Step 1: Construct the Table

<table>
<thead>
<tr>
<th></th>
<th></th>
<th>a</th>
<th>b</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>BH</td>
<td></td>
<td></td>
</tr>
<tr>
<td>B</td>
<td>CE</td>
<td></td>
<td></td>
</tr>
<tr>
<td>C</td>
<td></td>
<td>D</td>
<td></td>
</tr>
<tr>
<td>D</td>
<td>G</td>
<td></td>
<td></td>
</tr>
<tr>
<td>E</td>
<td></td>
<td>F</td>
<td></td>
</tr>
<tr>
<td>F</td>
<td>G</td>
<td></td>
<td></td>
</tr>
<tr>
<td>G</td>
<td>BH</td>
<td></td>
<td></td>
</tr>
<tr>
<td>H</td>
<td></td>
<td>J</td>
<td></td>
</tr>
<tr>
<td>J</td>
<td></td>
<td>K</td>
<td></td>
</tr>
<tr>
<td>K</td>
<td></td>
<td>M</td>
<td></td>
</tr>
<tr>
<td>M</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Step 2: Construct $\varepsilon$-closure

<table>
<thead>
<tr>
<th></th>
<th>$\varepsilon$</th>
<th>a</th>
<th>b</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>BHCE</td>
<td></td>
<td></td>
</tr>
<tr>
<td>B</td>
<td>CE</td>
<td></td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>D</td>
<td></td>
<td></td>
</tr>
<tr>
<td>D</td>
<td>GBHCE</td>
<td></td>
<td></td>
</tr>
<tr>
<td>E</td>
<td>F</td>
<td></td>
<td></td>
</tr>
<tr>
<td>F</td>
<td>GBHCE</td>
<td></td>
<td></td>
</tr>
<tr>
<td>G</td>
<td>BHCE</td>
<td></td>
<td></td>
</tr>
<tr>
<td>H</td>
<td>J</td>
<td></td>
<td></td>
</tr>
<tr>
<td>J</td>
<td></td>
<td>K</td>
<td></td>
</tr>
<tr>
<td>K</td>
<td></td>
<td></td>
<td>M</td>
</tr>
<tr>
<td>M</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Step 3: Update Other Columns

<table>
<thead>
<tr>
<th></th>
<th>(\varepsilon)</th>
<th>a</th>
<th>b</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>BHCE</td>
<td>DJ</td>
<td>F</td>
</tr>
<tr>
<td>B</td>
<td>CE</td>
<td>D</td>
<td>F</td>
</tr>
<tr>
<td>C</td>
<td></td>
<td>D</td>
<td></td>
</tr>
<tr>
<td>D</td>
<td>GBHCE</td>
<td>DJ</td>
<td>F</td>
</tr>
<tr>
<td>E</td>
<td></td>
<td></td>
<td>F</td>
</tr>
<tr>
<td>F</td>
<td>GBHCE</td>
<td>DJ</td>
<td>F</td>
</tr>
<tr>
<td>G</td>
<td>BHCE</td>
<td>DJ</td>
<td>F</td>
</tr>
<tr>
<td>H</td>
<td></td>
<td>J</td>
<td></td>
</tr>
<tr>
<td>J</td>
<td></td>
<td></td>
<td>K</td>
</tr>
<tr>
<td>K</td>
<td></td>
<td></td>
<td>M</td>
</tr>
<tr>
<td>M</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Step 4: Construct a New Table

<table>
<thead>
<tr>
<th></th>
<th>ε</th>
<th>a</th>
<th>b</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>BHCE</td>
<td>DJ</td>
<td>F</td>
</tr>
<tr>
<td>B</td>
<td>CE</td>
<td>D</td>
<td>F</td>
</tr>
<tr>
<td>C</td>
<td>D</td>
<td></td>
<td></td>
</tr>
<tr>
<td>D</td>
<td>GBHCE</td>
<td>DJ</td>
<td>F</td>
</tr>
<tr>
<td>E</td>
<td></td>
<td></td>
<td>F</td>
</tr>
<tr>
<td>F</td>
<td>GBHCE</td>
<td>DJ</td>
<td>F</td>
</tr>
<tr>
<td>G</td>
<td>BHCE</td>
<td>DJ</td>
<td>F</td>
</tr>
<tr>
<td>H</td>
<td></td>
<td>J</td>
<td></td>
</tr>
<tr>
<td>J</td>
<td></td>
<td></td>
<td>K</td>
</tr>
<tr>
<td>K</td>
<td></td>
<td></td>
<td>M</td>
</tr>
<tr>
<td>M</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Step 4: Construct a New Table

<table>
<thead>
<tr>
<th></th>
<th></th>
<th>a</th>
<th>b</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>BHCE</td>
<td>DJ</td>
<td>F</td>
</tr>
<tr>
<td>B</td>
<td>CE</td>
<td>D</td>
<td>F</td>
</tr>
<tr>
<td>C</td>
<td>D</td>
<td></td>
<td></td>
</tr>
<tr>
<td>D</td>
<td>GBHCE</td>
<td>DJ</td>
<td>F</td>
</tr>
<tr>
<td>E</td>
<td></td>
<td></td>
<td>F</td>
</tr>
<tr>
<td>F</td>
<td>GBHCE</td>
<td>DJ</td>
<td>F</td>
</tr>
<tr>
<td>G</td>
<td>BHCE</td>
<td>DJ</td>
<td>F</td>
</tr>
<tr>
<td>H</td>
<td></td>
<td>J</td>
<td></td>
</tr>
<tr>
<td>J</td>
<td></td>
<td>K</td>
<td></td>
</tr>
<tr>
<td>K</td>
<td></td>
<td>M</td>
<td></td>
</tr>
<tr>
<td>M</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
</tr>
</thead>
<tbody>
<tr>
<td>ABHCE</td>
<td>DJ</td>
<td>F</td>
</tr>
</tbody>
</table>
Step 4: Construct a New Table

<table>
<thead>
<tr>
<th>A</th>
<th>BHCE</th>
<th>DJ</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>B</td>
<td>CE</td>
<td>D</td>
<td>F</td>
</tr>
<tr>
<td>C</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>D</td>
<td>GBHCE</td>
<td>DJ</td>
<td>F</td>
</tr>
<tr>
<td>E</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>F</td>
<td>GBHCE</td>
<td>DJ</td>
<td>F</td>
</tr>
<tr>
<td>G</td>
<td>BHCE</td>
<td>DJ</td>
<td>F</td>
</tr>
<tr>
<td>H</td>
<td></td>
<td>J</td>
<td></td>
</tr>
<tr>
<td>J</td>
<td></td>
<td></td>
<td>K</td>
</tr>
<tr>
<td>K</td>
<td></td>
<td></td>
<td>M</td>
</tr>
<tr>
<td>M</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Step 4: Construct a New Table

<table>
<thead>
<tr>
<th></th>
<th>ε</th>
<th>a</th>
<th>b</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>BHCE</td>
<td>DJ</td>
<td>F</td>
</tr>
<tr>
<td>B</td>
<td>CE</td>
<td>D</td>
<td>F</td>
</tr>
<tr>
<td>C</td>
<td>D</td>
<td></td>
<td></td>
</tr>
<tr>
<td>D</td>
<td>GBHCE</td>
<td>DJ</td>
<td>F</td>
</tr>
<tr>
<td>E</td>
<td></td>
<td>F</td>
<td></td>
</tr>
<tr>
<td>F</td>
<td>GBHCE</td>
<td>DJ</td>
<td>F</td>
</tr>
<tr>
<td>G</td>
<td>BHCE</td>
<td>DJ</td>
<td>F</td>
</tr>
<tr>
<td>H</td>
<td></td>
<td>J</td>
<td></td>
</tr>
<tr>
<td>J</td>
<td>BHCE</td>
<td></td>
<td>K</td>
</tr>
<tr>
<td>K</td>
<td></td>
<td></td>
<td>M</td>
</tr>
<tr>
<td>M</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
</tr>
</thead>
<tbody>
<tr>
<td>ABHCE</td>
<td>DJ</td>
<td>F</td>
</tr>
<tr>
<td>DJ</td>
<td>DJ</td>
<td>FK</td>
</tr>
<tr>
<td>F</td>
<td>DJ</td>
<td>F</td>
</tr>
<tr>
<td>FK</td>
<td>DJ</td>
<td>FM</td>
</tr>
</tbody>
</table>
Step 4: Construct a New Table

<table>
<thead>
<tr>
<th></th>
<th>ε</th>
<th>a</th>
<th>b</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>BHCE</td>
<td>DJ</td>
<td>F</td>
</tr>
<tr>
<td>B</td>
<td>CE</td>
<td>D</td>
<td>F</td>
</tr>
<tr>
<td>C</td>
<td>D</td>
<td></td>
<td></td>
</tr>
<tr>
<td>D</td>
<td>GBHCE</td>
<td>DJ</td>
<td>F</td>
</tr>
<tr>
<td>E</td>
<td></td>
<td>F</td>
<td></td>
</tr>
<tr>
<td>F</td>
<td>GBHCE</td>
<td>DJ</td>
<td>F</td>
</tr>
<tr>
<td>G</td>
<td>BHCE</td>
<td>DJ</td>
<td>F</td>
</tr>
<tr>
<td>H</td>
<td>J</td>
<td></td>
<td></td>
</tr>
<tr>
<td>J</td>
<td></td>
<td>K</td>
<td></td>
</tr>
<tr>
<td>K</td>
<td></td>
<td>M</td>
<td></td>
</tr>
<tr>
<td>M</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
</tr>
</thead>
<tbody>
<tr>
<td>ABHCE</td>
<td>DJ</td>
<td>F</td>
</tr>
<tr>
<td>DJ</td>
<td>DJ</td>
<td>FK</td>
</tr>
<tr>
<td>F</td>
<td>DJ</td>
<td>F</td>
</tr>
<tr>
<td>FK</td>
<td>DJ</td>
<td>FM</td>
</tr>
<tr>
<td>FM</td>
<td>DJ</td>
<td>F</td>
</tr>
</tbody>
</table>
Step 5: Generate the DFA

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
</tr>
</thead>
<tbody>
<tr>
<td>ABHCE</td>
<td>DJ</td>
<td>F</td>
</tr>
<tr>
<td>DJ</td>
<td>DJ</td>
<td>FK</td>
</tr>
<tr>
<td>F</td>
<td>DJ</td>
<td>F</td>
</tr>
<tr>
<td>FK</td>
<td>DJ</td>
<td>FM</td>
</tr>
<tr>
<td>FM</td>
<td>DJ</td>
<td>F</td>
</tr>
</tbody>
</table>
Step 5: Generate the DFA

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
</tr>
</thead>
<tbody>
<tr>
<td>ABHCE</td>
<td>DJ</td>
<td>F</td>
</tr>
<tr>
<td>DJ</td>
<td>DJ</td>
<td>FK</td>
</tr>
<tr>
<td>F</td>
<td>DJ</td>
<td>F</td>
</tr>
<tr>
<td>FK</td>
<td>DJ</td>
<td>FM</td>
</tr>
<tr>
<td>FM</td>
<td>DJ</td>
<td>F</td>
</tr>
</tbody>
</table>

Note: the number of states is not minimized
An NFA may be in many states at any time

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

The resulting DFA has $O(2^N)$ space complexity where $N$ is the number of original states
NFA to DFA Time Complexity

- A DFA can be implemented by a 2D table $T$
  - One dimension is “states”, the other dimension is “input characters”
  - For $S_a \xrightarrow{c} S_b$, we have $T[S_a,c] = S_b$

- DFA execution
  - If the current state is $S_a$ and input is $c$, then read $T[S_a,c]$
  - Update the current state to $S_b$, assuming $S_b = T[S_a,c]$
  - 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 set of next possible states
  - Requires $O(|X| \ast N)$ 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

- Lexical Specification
- Regular Expression
- NFA
- DFA
- Table-driven Implementation of DFA
A scanner recognizes multiple REs

- `letter l digit l_` -> node
- `other` -> node
- `return 'identifier'`
- `letter l` -> node
- `digit` -> node
- `other` -> node
- `return 'integer_const'`
- `digit` -> node
- `other` -> node
- `return 'op_ge'`
- `>` -> node
- `=` -> node
- `other` -> node
- `return 'op_gt'`
How much should we match?

- In general, find the longest match possible
- If same length, rule appearing first in lex file 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 will implement 1 in your projects for simplicity
Beyond Regular Languages

- Regular languages are expressive enough to describe tokens during lexical analysis.
- Regular languages can express identifiers, strings, comments, etc.

However, it is the weakest (least expressive) formal language:

- Many languages are not regular
  - C programming language is not
  - "((((...)))" is also not
- Finite automata cannot remember # of times

- We need a more powerful language for parsing
  - In the next lecture, we will discuss context-free languages