Lexical Analysis
What is Lexical Analysis

What do we want to do?

Example:

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

The input is just a string of characters

```
“if(i == j)\ntz = 0; \ntelse\ntz = 1; \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
  - 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 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 input to the syntax analyzer (also called 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
    - Depends on language, design of parser
    - Recall: 
      ```
      if(i == j)
      tz = 0;
telse
      tz = 1;
      ```
    - 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 \((\text{class}, \text{lexeme})\)

  \[
  \text{if}(i == j)\ n\ t\ tz = 0; \ \text{else}\ n\ tz = 1; \ n
  \]

  - 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 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.
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 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 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 without parser feedback
- Allows 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

- A token specification should define:
  - What token classes there are
  - What strings belong to which class

- Answer: use **Regular Expressions** 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 $\Sigma$ be a set of characters, a language over $\Sigma$ is a set of strings of the characters drawn from $\Sigma$.
Examples of Languages

- Alphabet $\sum = \text{(set of) English characters}$
  Language $L = \text{(set of) English sentences}$

- Alphabet $\sum = \text{(set of) Digits, +, -}$
  Language $L = \text{(set of) Integer numbers}$

- Alphabet $\sum = \text{(set of) English characters}$
  Language $L = \text{(set of) Tokens in C}$

The set of tokens is also a language, just like English (!)

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

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

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

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

- Iteration (Kleene closure)
  \[ A^* = \bigcup_{i \geq 0} A^i \quad \text{where } A^i = A \ldots A \text{ (} i \text{ times)} \]
  In particular
  \[ A^* = \{ \varepsilon \} + A + AA + AAA + \ldots \]
  \[ A^+ = A + AA + AAA + \ldots = A \cdot A^* \]
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 $\sum$
L(RE) is defined as the regular language generated by regular expression RE

- L(ε) = { "" }
- L('c') = { "c" }
- L(A+B) = L( A ) ∪ L( B )
- L(AB) = { ab | a ∈ L(A) and b ∈ L(B) }
- L(A*) = ∪ i≥0 L(A^i)
Keyword: “else” or “if” or “while” or ...

Q: Is '000' an integer?

Q: How to define another integer RE that excludes sequences with leading 0s?
Keyword: “else” or “if” or “while” or ...

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

- keyword = ‘else’ + ‘if’ + ‘while’ + ...
- ‘else’ abbreviates
  ‘e’ (concatenate) ‘l’ (concatenate) ‘s’ (concatenate) ‘e’
RE Examples

Keywords:
- `else`  `if`  `while`  ...
  - `keyword = 'else' + 'if' + 'while' + ...`
  - `'else' abbreviates
    - 'e' (concatenate) 'l' (concatenate) 's' (concatenate) 'e'

Integers:
- `digit = '0' + '1' + '2' + '3' + '4' + '5' + '6' + '7' + '8' + '9'
- `integer = digit digit*`
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

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

- phoneNumber = ‘(’ area ‘)’ exchange ‘-’ phone

Email address: student @ pitt.edu

- $\sum = \text{letter} \cup \{., @\}$
- name = letter $^+$

- emailAddress = name ‘@’ name ‘.’ name
RE in Programming Languages

RE used in languages

- 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("abrabcabab", "(a|b|r)+");
  ```
# Some Common REs in Programming Languages

<table>
<thead>
<tr>
<th></th>
<th>Meaning</th>
<th></th>
<th>Meaning</th>
<th></th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>\d</td>
<td>Digits</td>
<td>\w</td>
<td>Any word char</td>
<td>\s</td>
<td>Space char</td>
</tr>
<tr>
<td>\D</td>
<td>Non-digits</td>
<td>\W</td>
<td>Non-word char</td>
<td>\S</td>
<td>Non-space char</td>
</tr>
<tr>
<td>[a-f]</td>
<td>Char range</td>
<td>[^a-f]</td>
<td>Exclude range</td>
<td>^</td>
<td>Matching string start</td>
</tr>
<tr>
<td>?</td>
<td>Optional</td>
<td>{n,m}</td>
<td>Appear n-m times</td>
<td>$</td>
<td>Matching string end</td>
</tr>
<tr>
<td>.</td>
<td>Any char</td>
<td>(...)</td>
<td>Capture matches</td>
<td>\G,{</td>
<td>Matching (, { ...</td>
</tr>
<tr>
<td>.</td>
<td>Matching “.”</td>
<td>+</td>
<td>Appear &gt;=1 times</td>
<td>*</td>
<td>Appear 0 or many times</td>
</tr>
</tbody>
</table>
Implementation of Lexical Analysis
Implementation of Lexical Analysis

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

How do we go from specification to implementation?
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
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

```c
 %{ /* 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() { ... }```
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
- `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

Lexical Specification

Regular Expression → NFA → DFA → Table-driven Implementation of DFA
An Overview of RE to FA

Our implementation sketch

- Lexical Specification
  - Manual conversion
- Regular Expression
  - NFA
  - DFA
  - Table-driven Implementation of DFA
  - 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: $A \cup B \equiv A + B$

Option: $A \cup \varepsilon \equiv A ?$

Range: ‘a’ + ‘b’ + ... + ‘z’ $\equiv [a-z]$

Excluded range:

complement of [a-z]$\equiv [^{\text{a-z}}]$
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 \( \delta: S_a \xrightarrow{\text{input}} S_b \) read as
  - in state S1 on input “a” go to state S2

- At the end of input (or no transition possible), if current state \( X \)
  - \( X \in \) accepting set \( F \), then \( \Rightarrow \) accept
  - otherwise, \( \Rightarrow \) 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

Example: a finite state automata that accepts only "1"
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\}

![Finite Automata Diagram]

- Initial state
- States labeled 1 and 0
- Transitions from 1 back to itself and from 0 to itself
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\}
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\}
Given the state graph of a DFA,
Table Implementation of a DFA

Given the state graph of a DFA,

\[
\begin{array}{ccc}
\text{state} & \rightarrow \text{input characters} \\
S & 0 & 1 \\
T & & \\
U & & \\
\end{array}
\]
Given the state graph of a DFA,

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

State transition table:

<table>
<thead>
<tr>
<th>State</th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>S</td>
<td>T</td>
<td>U</td>
</tr>
<tr>
<td>T</td>
<td>T</td>
<td>U</td>
</tr>
<tr>
<td>U</td>
<td>T</td>
<td>X</td>
</tr>
</tbody>
</table>

→ Input characters
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");
}
```
Discussion

- 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
Discussion

- 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
From RE to FA

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 ε-moves

- Non-deterministic Finite Automata (NFA)
  - Can have multiple transitions for one input in a given state
  - Can have ε-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

- $\textit{DFA} \subseteq \textit{NFA}$ and $\textit{L(DFA)} \subseteq \textit{L(NFA)}$
  - All DFAs are NFAs by definition

- Now is $\textit{L(NFA)} \subseteq \textit{L(DFA)}$?
  - Are all Ls expressible using NFA expressible using DFA?
  - Yes, as we will later learn, hence $\textit{L(NFA)} \equiv \textit{L(DFA)}$. 
Converting RE to NFA

- McNaughton-Yamada-Thompson Algorithm

- Step 1: processing atomic REs
  - ε expression
  - single character RE a

```
  ε expression
  a
```

```
  i ε f
```

```
  i a f
```
Converting RE to NFA (cont.)

- Step 2: processing compound REs

- $r = s | t$

- $r = s \cdot t$

- $r = s^*$
Converting RE to NFA (cont.)

- Step 2: processing compound REs

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

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

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

- $r = s | t$

- $r = s \cdot t$

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

- $r = s \mid t$

- $r = s \cdot t$

- $r = s^*$
Converting RE to NFA (cont.)

Step 2: processing compound REs

- $r = s \mid t$

- $r = s \cdot t$

- $r = s^*$
In-class Practice

Convert \((a|b)^*a\ b\ b\) to NFA
In-class Practice

Convert “(a|b)* a b b” to NFA
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 ($NFA \to DFA$)
- 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 $\equiv$ length of input
- For NFAs, number of potential transitions can be larger

NFA $\to$ 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 \( \text{iff} \)
  
  \( S_b \) 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 ?

```
state ↓   → input characters
  0   1
  A
  B
  C
```

```
0   1
A
B
C
```
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:

- States: A, B, C
- Transitions:
  - A: 0 → A, 1 → B
  - B: 0 → B, 1 → C
  - C: 1/0 → C

Input characters: 0, 1
Example NFA to DFA

What is the Equivalent DFA?

```
state ↓ → input characters

<table>
<thead>
<tr>
<th></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>
```
Example NFA to DFA

What is the Equivalent DFA?

![Diagram of NFA to DFA conversion]

<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>
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>
<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?

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>
<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
Step 1: Construct the Table

<table>
<thead>
<tr>
<th></th>
<th>ε</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>D</td>
<td></td>
<td></td>
</tr>
<tr>
<td>D</td>
<td>G</td>
<td></td>
<td></td>
</tr>
<tr>
<td>E</td>
<td>F</td>
<td></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>J</td>
<td></td>
<td></td>
</tr>
<tr>
<td>J</td>
<td>K</td>
<td></td>
<td></td>
</tr>
<tr>
<td>K</td>
<td>M</td>
<td></td>
<td></td>
</tr>
<tr>
<td>M</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Step 2: Construct $\varepsilon$-closure

\[
\begin{array}{|c|c|c|c|}
\hline
 & \varepsilon & a & b \\
A & BHCE & & \\
B & CE & & \\
C & D & & \\
D & GBHCE & & \\
E & & F & \\
F & GBHCE & & \\
G & BHCE & & \\
H & & J & \\
J & & K & \\
K & & M & \\
M & & & \\
\hline
\end{array}
\]
Step 3: Update Other Columns

<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>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></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>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 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></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></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>

<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></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>

<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>
</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>DJ</td>
<td>F</td>
</tr>
<tr>
<td>D</td>
<td>GBHCE</td>
<td>DJ</td>
<td>F</td>
</tr>
<tr>
<td>E</td>
<td>GBHCE</td>
<td>DJ</td>
<td>F</td>
</tr>
<tr>
<td>F</td>
<td>BHCE</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>K</td>
<td></td>
<td></td>
</tr>
<tr>
<td>K</td>
<td>M</td>
<td></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>
</tbody>
</table>
Step 4: Construct a New Table

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

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>ABHCE</td>
<td>DJ</td>
</tr>
<tr>
<td>B</td>
<td>DJ</td>
<td>DJ</td>
</tr>
<tr>
<td>C</td>
<td>F</td>
<td>DJ</td>
</tr>
<tr>
<td>D</td>
<td>FK</td>
<td>DJ</td>
</tr>
<tr>
<td>E</td>
<td>FM</td>
<td>DJ</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
NFA to DFA. Space Complexity

- 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 $O(|X| \times N)$ 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 recognize multiple REs
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

\((\text{numConst}, 123.45)\)

rather than

\((\text{numConst}, 123), (\text{dot}, \text{"."}), (\text{numConst}, 45)\)
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.