Virus Pattern Matching

Wei Wang
Anti-virus Scanning

• Scanning is the basic building block of anti-virus software
  – Each virus is characterized by a “signature” or pattern
  – Sometimes called “scan strings”

• For example, the tricky jump always has two instructions:
  - To detect a tricky jump, we can then scan over the bytes of a file, and look for a 0x68 and a 0xC3, with four bytes in between.
Pattern Matching Basics

- Pattern matching is a well studied area
- Finite Automata constructed and used for all pattern matching tasks, e.g.,
  - String matching / processing
  - Syntax analysis
- Regular expressions (RE) are used to simplify pattern expression
- Lex or Flex are used to automatically convert patterns (RE) to finite automata to executable programs
What We Will Learn

• Regular expression

• Finite Automata
  – Deterministic/Non-deterministic finite automata
  – Conversion from Non-deterministic to deterministic
  – Minimizing deterministic finite automata

• Flex
  – Write flex grammar to convert regular expression to a program that performs pattern matching
Regular Expression

- Regular expression is an algebraic way to describe patterns/strings, more formally, languages.
- Regular expression contains:
  - Regular characters: means the character itself; most letters are regular characters
  - Special characters: special operations on regular characters
- $\varepsilon$ is a special character represents empty string – a string without any character
Regular Expression: Regular Characters

- Most letters, numbers and some punctuations are normal characters
- E.g., regular expression \( abc \) matches string “abc”, and only that string
- E.g., regular expression \( x87z \) matches string “x87z”, and only that string
Regular Expression: Sub-expression and Concatenation

- Parentheses '()' mark a subexpression
  - e.g., regular expression \((abc)\) matches string “abc”, and only that string
  - e.g., regular expression \((x87z)\) matches string “x87z”, and only that string

- Subexpressions and regular characters can be concatenated into one regular expression
  - e.g., regular expression \((x87z)mu(abc)\) matches string “x87zmuabc”, and only that string
Regular Expression: Special Characters

- *: matches zero or more of a sub-expression
  - e.g., \texttt{ab*} matching any string starts with an \texttt{a}, following by zero or more \texttt{b}'s, such as “a”, “ab”, “abb”, “abbb”, “abbbb” …
  - e.g., \texttt{(ab)*} matching any string that repeats “ab”, including the empty string, such as \texttt{ε}, “ab”, “abab”, “ababab”, “abababab” …
Regular Expression: Special Characters cont'd

• +: matches one or more of a sub-expression
  - e.g., \(a^+\) matching any string starts with an \(a\), following by one or more \(b\)'s, such as “ab”, “abb”, “abbb”, “abbb” ...
  - e.g., \((ab)^+\) matching any string that repeats “ab”, excluding the empty string, such as “ab”, “abab”, “ababab”, “abababab” ...
Regular Expression: Special Characters cont'd

• | : matches one or another
  - e.g., \texttt{ab} | \texttt{bc} matches “ab” or “bc”
  - e.g., \texttt{x(10|01)x} matches “x10x” or “x01x”

• . : matches one character
  - e.g., \texttt{a.b} matches any 3-character string starts with \texttt{a} and ends with \texttt{b}, such as “acb”, “axb”, “a0b” …
  - e.g. \texttt{a.*b} matches any strings starts with \texttt{a} and ends with \texttt{b}, such as “axxxb”, “ab”, “a098xb” …
Regular Expression: Special Characters cont'd

- [ and ]: matches a single character that is contained within the brackets.
  - e.g., a[bc]d, matches “abd” or “acd”
  - e.g., x[0–9]y, matches any string starts with x, ends with y, and has one digit in middle, i.e., “x0y”, “x1y”, “x2y”, … , “x9y”
  - e.g., 0[a–zA–Z]1, matches any string starts with 0, ends with 1, and has letter in middle, suc as, “0x1”, “0q1”, “0L1”, …
Regular Expression: Special Characters cont'd

- [^ and ]: matches any character that is **not** contained within the brackets
  - e.g., `xyz[^abc]`, matches any 4-character string starts with “xyz” and **does not** end with an `a`, `b` or `c`.

- { and } : specifies the number of occurrence of subexpression
  - e.g., `a{3,5}`, matches any string with 3 to 5 `a`'s
  - e.g., `[0-9]{2,9}`, matches any string with 2 to 9 digits
Regular Expression: Special Characters cont'd

• There are more special characters, defined by various standards. You can find them online.

• Sometimes, you need to put "\" before a special character for it to be recognized a special character
  – e.g., basic regular syntax of POSIX

• Sometimes, you need to put "\" before a special character for it to recognized a regular character
  – e.g., extended regular syntax of POSIX
Some Regular Expression Examples

- A phone number;
  - `[0-9]{3,3}\-[0-9]{3,3}\-[0-9]{4,4}`

- An email address with only lower case characters, numbers, dot and @
  - `[a-z][a-z0-9]*@[a-z0-9.]*[a-z]`
Finite-state Automata

- Finite-state Automata is a simple idealized machine used to recognize patterns within input strings
- Non-deterministic Finite Automata (NFA):
  - Used to convert regular expressions into finite-state automata
- Deterministic Finite Automata (DFA):
  - Converted from NFA for better implementation of pattern matching
  - NFA and DFA are equivalent in pattern matching
- Constructing DFA is the standard approach for arbitrary pattern matching or substring matching
A Finite Automaton Example

- This automaton matches any string with a substring “ab”
  - “S” is the start state
  - “ab” is the acceptance state (a match found)
  - “rej” is the rejection state (no match found)
  - An arrow represents a state change based on input character
Non-deterministic Finite Automata

- A non-deterministic finite automaton is 5-tuple: \( M = \{Q, \Sigma, \delta, q_0, F\} \)
- \( Q \) is a finite set of states
- \( \Sigma \) is a finite set of permissible input characters
- \( \delta \) is a mapping from \( Q \times \Sigma \) to \( Q \)
- \( q_0 \in Q \), the start state
- \( F \subseteq Q \) is the set of final states
Converting RE to NFA

- Thompson's constructions
- Only one start state, one final state

![Diagram of NFA]

No arcs from outside except those from “S”, no arcs leaving except those to “F”
Converting RE to NFA cont'd

- A NFA matches one character input $c$

- A NFA matches an empty string: this is why NFA is non-deterministic. Because of the empty input, a NFA can in either “S” or “F” state
Converting RE to NFA cont'd

- Union of two NFAs (for [ ] and '|' )
  - i.e., \( \text{RE1} | \text{RE2} \). Let \( \text{N1} \) be \( \text{RE1}'s \) NFA, \( \text{N2} \) be \( \text{RE2}'s \) NFA
Converting RE to NFA cont'd

- Concatenation of two NFAs
  - i.e., $RE_1RE_2$. Let $N_1$ be $RE_1$'s NFA, $N_2$ be $RE_2$'s NFA
Converting RE to NFA cont'd

- A NFA matches \( RE_1^* \) (zero or more occurrence of pattern \( RE_1 \))

- How about \( RE_1^+ \)?
A NFA Example

- Regular expression: \([ab]9^*[cd]\)
Deterministic Finite Automata

• NFA is every hard to implement, because,
  - The ε transition
  - For certain state and input there is no move

• Deterministic Finite Automata (DFA)
  - Removes the ε transition,
  - For each state and an input character, there is one and only one transition to a next state

• Each NFA can be converted into a DFA
Deterministic Finite Automata cont'd

A deterministic finite automaton is 5-tuple: 
\[ M = \{Q, \Sigma, \delta, q_0, F\} \]

The elements have similar meanings as those in NFA.
A DFA Example

- Regular expression: \([ab]9^*[cd]\)

Any input not marked goes to rejection
Minimizing DFA

- There is a DFA with minimal states for any pattern
- Minimal DFA can be found by reducing a non-minimal DFA with DFA minimization algorithms
- Minimal DFA requires fewer memory to implement
- Example: \([ab]^9[cd]\)
Flex – A Lexical Analyzer Generator

• Given a pattern, flex automatically generate a C-program that can scan over an input string, and find the substrings that match the pattern.

• Flex specification is composed of rules (patterns) and actions:
  – Rules define what patterns to match
  – Actions define what to do with matched substrings

![Diagram showing the process of using flex and gcc to generate a C source file and an executable.]

rules and actions in plain text, *.l file

flex

C source file that implement the pattern matching and actions

gcc

Executable that performs pattern matching and actions
Flex File Syntax

• Flex syntax:

```c
{%
  /* Extra includes and variable declarations
   in C syntax */
%
%
/* definitions for short cuts*/
%
%
/* rules and actions*/
Patterns/rules    { /*actions in C */ } 
%
%
/* user code in C */
```
Flex File Syntax

- Flex syntax with examples:

```c
{%
    /* Extra includes and variable declarations in C syntax */
    #include <stdio.h>
    int global_counter = 0;
%

/* name definitions */
DIGIT [0-9] /* declaration DIGIT to be a single number */
%
/* rules and actions */
/* in Flex, declared names are put in {} to use */
/* yytext is a predefined flex variable with the value of matched substring */
{DIGIT}+    { printf("found %s\n", yytext);}
%
/* user code in C */
int main() { yylex(); return 0; } /* yylex() starts scanning*/
```
Flex Compilation

• Compile a flex with the following command
  – flex flex_source.l
  – A C file named lex.yy.c will be generated
• Then compile the lex.yy.c with gcc
• Example demonstration: phone number matching