Chapter Eight:
Regular Expression Applications

We have seen some of the implementation techniques related to DFAs and NFAs. These important techniques are like tricks of the programmer's trade, normally hidden from the end user. Not so with regular expressions: they are often visible to the end user, and part of the user interface of a variety of useful software tools.

Outline

Text File Search

- Unix tool: egrep
- Searches a text file for lines that contain a substring matching a specified pattern
- Echoes all such lines to standard output

Example: A Constant Substring

File `names`:

```
fred
barney
wilma
betty
```

egrep command and results:

```
% egrep 'a' names
barney
wilma
%
```

More Than Simple Substrings

- egrep understands a language of patterns
- Various dialects of its pattern-language are also used by many other tools
- Confusingly, these patterns are often called *regular expressions*, but they differ from ours
- To keep the two ideas separate, we'll call the text patterns used by egrep and other tools by their common nickname: *regexps*

A Regexp Dialect

* like our Kleene star: for any regexp *x*, *x** matches strings that are concatenations of zero or more strings from the language specified by *x*

| like our +: for any regexps *x* and *y*, *x*|*y* matches strings that match either *x* or *y* (or both)

() used for grouping

^ this special symbol at the start of the regexp allows it to match only at the start of the line

$ this special symbol at the end of the regexp allows it to match only at the end of the line

. matches any symbol (except the end-of-line marker)

Example

File `names`:

```
fred
barney
wilma
betty
```

egrep for `a`, followed by any string, followed by `y`:

```
% egrep 'a.*y' names
barney
%
```

Example

File `names`:

```
fred
barney
wilma
betty
```

egrep for odd-length string; what went wrong?

```
% egrep '.(..)*' names
fred
barney
wilma
betty
%
```

Example

File `names`:

```
fred
barney
wilma
betty
```

egrep for odd-length *line*:

```
% egrep '^.(..)*$' names
fred
barney
wilma
betty
%
```

Example

File `numbers`:

```
0
1
10
11
100
101
110
111
1000
1001
1010
```

egrep for numbers divisible by 3:

```
% egrep '^(0|1(01*0)*1)*$' numbers
0
11
110
1001
%
```

Outline

Capturing Parentheses

- Many regexp dialects can define more than just the regular languages
- Capturing parentheses:
  - `\(` *r* `\)` captures the text that was matched by the regexp *r*
  - `\n` matches the same text captured by the *n*th previous capturing left parenthesis
- Found in grep (but not most versions of egrep)

Example

File `test`:

```
abaaba
ababa
abbbabbb
abbaabb
```

grep for lines that consist of doubled strings:

```
% grep '^\(.*\)\1$' test
abaaba
abbbabbb
%
```

More Than Regular

- The formal language corresponding to that example is $\{xx \mid x \in \Sigma^*\}$
- It turns out that this language is not regular
  - Like DFAs, regular expressions can do only what you could implement in a computer using a fixed, finite amount of memory
  - Capturing parentheses must remember a string whose size is unbounded
- We'll see this more formally later

Outline

Many Regexp Tools

- Many programs make use of regexp dialects:
  - Text tools like emacs, vi, and sed
  - Compiler construction tools like lex
  - Programming languages like Perl, Ruby, and Python
  - Program language libraries like those for Java and the .NET languages
- How do all these systems implement regexp matching?

Implementing Regexps

- We've already seen how, roughly:
  - Convert regexp to an NFA
  - Simulate that
  - Or, convert to DFA and simulate that
- Many implementation tricks are possible; we haven't worried much about efficiency
- And some important details are different because regexps are used to match *substrings*

Using a DFA

- Our basic DFA decides after it reads the whole string
- For regexps, we need to find whether any *substring* is accepted
- That means running the DFA repeatedly, on each successive starting position
- Run the DFA until:
  - it enters an accepting state: that's a match
  - enters a non-accepting trap state: restart the DFA from the next possible starting position
  - hits the end of the string: restart the DFA from the next possible starting position

Which Match?

- Some tools needs to know which substring matched
- Capturing parentheses, for example
- If there is more than one match in a given string, which should the tool find?
  - The string `abcab` contains two substrings that match the regexp `ab`
- It isn't enough to specify the *leftmost* match: what if several matches start at the same place?
  - The string `abb` contains three substrings that match the regexp `ab*`, and they all start at the first symbol

Longest Leftmost

- Some tools are required to find the *longest leftmost* match in a string
  - The string `abbcabb` contains six matches for `ab*`
  - The first `abb` is the longest leftmost match
- That means running the DFA past accepting states
- Run the DFA starting from each successive position, until it enters a non-accepting trap or hits the end
  - As you go, keep track of the last accepting state entered, and the string position at the time
  - At the end of this iteration, if any accepting state was recorded, that is the longest leftmost match

Using an NFA

- Similar accommodations are required
- Run from each successive starting position
- When an implementation using backtracking finds a match, it cannot necessarily stop there
- If the longest match is required, it must remember the match and continue
- Explore all paths through the NFA to make sure the longest match is found

Outline

java.util.regex

- The Java package java.util.regex contains classes for working with regexps in Java
- Two particularly important ones:
  - The Pattern class
    - A compiled version of a regexp, ready to be given an input string to test
    - A bit like a Java representation of an NFA
  - The Matcher class
    - Has a Pattern, an input string to run it on, and the current state of the search for a match
    - Can find matches within a string and report their locations

Example

- A mini-grep written in Java
- We'll take a regexp from the command line, and make it into a Pattern
- Then, for each line of the standard input:
  - we'll make a Matcher for that line and use it to test for a match with our Pattern
  - If it matches, we'll echo the line to the standard output

```
import java.io.*;
import java.util.regex.*;

/**
 * A Java application to demonstrate the Java package
 * java.util.regex.  We take one command-line argument,
 * which is treated as a regexp and compiled into a
 * Pattern.  We then use that pattern to filter the
 * standard input, echoing to standard output only
 * those lines that match the Pattern.
 */
```

```
class RegexFilter {
  public static void main(String[] args)
        throws IOException {

    Pattern p = Pattern.compile(args[0]); // the regexp
    BufferedReader in =  // standard input
      new BufferedReader(new InputStreamReader(System.in));

    // Read and echo lines until EOF.

    String s = in.readLine();
    while (s!=null) {
      Matcher m = p.matcher(s);
      if (m.matches()) System.out.println(s);
      s = in.readLine();
    }
  }
}
```

Example, Continued

- Now this Java application can be used to do our divisible-by-three filtering:

```
% java RegexFilter '^(0|1(01*0)*1)*$' < numbers
0
11
110
1001
%
```

Outline

Preconstructing Automata

- Applications like grep take a regexp as user input:
  - Convert regexp to automaton
  - Simulate the automaton on a text file
  - Discard automaton when finished
- Other applications, like compilers, use the same regexp and automaton each time
- It would be a waste of time to do the regexp-to-automaton conversion each time the compiler is run

The lex Tool

- lex preconstructs automata:
  - Regexps as input
  - C source code for a DFA as output
- Similar tools exist for many other output languages
- Useful for applications like compilers that use the same set of regexps over and over

The lex Input File

```
definition section
%%
rules section
%%
user subroutines
```

- Definition section: a variety of preliminary definitions
- User subroutines: C code for extra C functions used from inside the rules section
- In our examples, both these sections will be empty:

```
%%
rules section
%%
```

Rules Section

```
%%
abc    {fprintf(yyout, "Found one.\n");}
.|\n   {}
%%
```

- The rules section is a list of regexps
- Each regexp is followed by some C code, to be executed whenever a match is found
- This example has two regexps with code

Rules Section

```
%%
abc   {fprintf(yyout, "Found one.\n");}
.|\n  {}
%%
```

- First regexp: `abc`
- Action when `abc` is matched: print a message to the current output file (`yyout`), which is the standard output in this case

Rules Section

```
%%
abc   {fprintf(yyout, "Found one.\n");}
.|\n  {}
%%
```

- Second regexp: `.|\n`
  - `.` matches any symbol except end-of-line
  - `\n` matches end-of-line
- Action when `.|\n` is matched: do nothing
- The string `abc` matches both regexps, but the lex-generated code goes with the longest match

Running lex

```
% flex abc.l
% gcc lex.yy.c -o abc -ll
%
```

- Assuming our example is stored as `abc.l`
- flex is the gnu implementation of lex
- C code output by flex is stored in `lex.yy.c`
- The gcc command compiles the C code
  - It puts the executable in `abc` (because of `-o abc`)
  - It links with the special lex library (because of `-ll`)

Running The lex Program

- Suppose `abctest` contains these lines:

```
abc
aabbcc
abcabc
```

- Then our lex program does this:

```
% abc < abctest
Found one.
Found one.
Found one.
%
```

Example

```
%%
^(0|1(01*0)*1)*$   {fprintf(yyout, "%s\n", yytext);}
.|\n               {}
%%
```

- This lex program echoes numbers divisible by 3
- The same thing we've already done with Java and with egrep
- The lex variable `yytext` gives us a way to access the substring that matched the regexp

Larger Applications

- For simple applications, the code produced by lex can be compiled as a stand-alone program, as in the previous examples
- For most large applications, the code produced by lex is one of many source files from which the full program is compiled
- Compilers are sometimes written this way:
  - lex generates some source
  - Other tools (like yacc) generate some source
  - Some is written by hand