CPSC 313 — Supplemental Material for Lecture #9
Applications of Regular Expressions

A Brief History

Regular expressions were invented in 1951 by Stephen Cole Kleene, as part of his work in "recursion theory" (and early name for computability theory). They therefore began as part of "theoretical computer science". More practical use began approximately in 1968, when they were used for pattern matching in a text editor, and lexical analysis in a compiler.

Several variations of regular expressions were used in various utilities, included in the UNIX operating systems, developed at Bell Laboratories in the 1970's, such as the following.

- Text editors, to find and replace test: `ed`, `vi` and (later) `emacs`, as well as the "stream-oriented" text processor `sed`
- Programming languages for text processing, including `awk`
- Programs to support program compilation (including lexical analysis), initially including `lex`

The format of regular expressions included in these early utilities was eventually standardized — in the "`POSIX.2` standard", which is discussed below.

In the 1980's significantly more complicated regular expressions were included in the (report processing) programming language Perl. The "Perl syntax" for regular expressions is now the other syntax (along with that given by the `POSIX` standard) that is widely used. Support for regular expressions is now included in a variety of programming languages, including Java.

Many applications use their own (slightly different) syntax and features for regular expressions. Documentation for any application that you are interested in should be consulted for further details.

POSIX **Standard**

As noted above, a standard for regular expressions (strings of symbols over the ASCII character set) was developed in the 1970's. While three "sets of compliance" were identified one (SRE — Simple Regular Expressions) and another (BSE — Basic Regular Expressions) is primarily used to establish backward compatibility. The third (ERE — Extended Regular Expressions) is more significant and is the basis for what follows.

- All characters match themselves except for the following special characters

  ```
  . [] {} () \ * + ? | ^ $
  ```

- The backlash character, \, is an ***escape*** character that effectively removes the "special meaning" of the special symbols they follow, so that these symbols can also be included in regular expressions. For example, the regular expression

  ```
  \?
  ```

  matches the character "?", while

  ```
  ?
  ```

  would probably not be recognized as a valid regular expression, at all. As another example, the regular expression

  ```
  \\
  ```

  matches the backslash character, "\"

- The "dot" character . matches any single character.[1] This is sometimes called a ***wild-card***.

- When it appears at the beginning of a regular expression, or subexpression, the "caret' character ^ indicates that the regular expression should only match text at the *beginning* of a line. For example, the regular expression

  ```
  ^A
  ```

  (only) matches an A at the beginning of a line in the text being processed. Thus the "caret" is one example of an ***anchor character***.

[1] In some cases — with various "command flags" set — this can be prevented from matching either a NULL character or a newline character.

- When it appears at the end of a regular expression or subexpression, $ indicates that the regular expression should only match text at the *end* of a line. For example, the regular expression

  A$

  (only) matches an A at the end of a line in the text being processed. Thus $ is another example of an ***anchor character***.

- A ***bracket expression*** is a list of characters enclosed by [and]. This matches any single character in the list — except that if the first character is a caret, ^, then any character that is *not* listed can be matched.

  – Thus the regular expression

    [0123456789]

    matches any one of the digits $0, 1, \ldots, 9$, while the regular expression

    [^0123456789]

    matches any of character *except* one of these digits.

  – In bracket expressions, a ***range expression*** consists of two characters separated by a hyphen, which is matched by an character in the identified range. For example, the regular expression

    [0-3]

    matches any of the digits 0, 1, 2 or 3.

    Unfortunately, range expressions might not have the meaning you intend because characters might be ordered in the underlying character set in a way that is different than you imagine. For example, the regular expression

    [a-d]

    might match any of a, b, c or d (as you probably expect) — but in some situations it match any of a, A, b, B, c, C or d instead. Thus range expressions should probably be used with care.

  – Certain classes of characters, called ***character classes***, are predefined within bracket expressions. These seem to depend on the application being used but generally include the following.

    * [:lower:] — Lower-case alphabetic characters, that is, $a, b, c, \ldots, z$.
    * [:upper:] — Upper-case alphabetic characters, that is, $A, B, C, \ldots, Z$.
    * [:alpha:] — Alphabetic characters, including $a, b, c, \ldots, Z$ and $A, B, C, \ldots, Z$.
    * [:digit:] — Digits $0, 1, 2, \ldots, 9$.

    * `[:alnum:]` — Alphanumeric characters, that is, the characters matched by `[:alpha:]` or by `[:digit:]`

- One can simply write one regular expression after another to provide a regular expression that is the ***concatenation*** of simpler regular expressions. For example, the regular expression

  `A..`

  matches any string with length three beginning with `A`.

- The "|" symbol is used to form regular expressions whose languages are the ***union*** of the languages of simpler regular expressions. For example, the regular expression

  `calgary|edmonton`

  matches either one of the strings "`calgary`" or "`edmonton`".

- Brackets can be used to change the usual precedence of operations. Thus, while

  `calgary|edmonton`

  matches either the string "`calgary`" or "`edmonton`", the regular expression

  `calg(ary|edm)onton`

  matches either the string "`calgaryonton`" or the string "calgedmonton" instead.

- Several operators can be used for ***repetition***.

  – An asterisk, $*$, is to denote the "Kleene star" operation. For example, the regular expression

    `a*`

    matches a sequence of zero or more `a`'s.

  – A + symbol can be used to indicate that *one* or more patterns matching a given regular expression. For example, the regular expression

    `(00)+`

    matches an even number of `0`'s, where the number is greater than or equal to two (since the regular expression `00` must be matched *one* or more times).

  – A question mark, `?`, indicates that either *zero* or *one* string matching the preceding character (or subexpression) should be used. Thus

    `three( or four)?`

    matches either "`three`" or "`three or four`".

– The curly braces can be used to specify a number, or range of numbers, of copies of a preceding character or subexpression that must be matched. For example, if m and n are integers such that $m < n$ then "`{n}`" indicates that the preceding character (or subexpression) is to be matched exactly n times, "`{m,}`" indicates that the preceding character (or subexpression) is to be matched m or more times, and "`{m,n}`" indicates that the preceding character (or subexpression) is be matched between m and n times. Thus

```
(Ab){2,4}
```

matches any of the strings `AbAb`, `AbAbAb`, or `AbAbAbAb`.

Applications

Search

The `egrep` command in UNIX accepts a regular expression (following "`-e`" and the name of a text file, and lists the lines of the text containing strings that match the pattern.

For example, "`L10.tex`" is the name of the text file that was typeset (using software called LaTeX) to produce these typeset notes. An execution of the command

```
egrep -e "[A-Z]{4}" L10.tex
```

lists all the lines of the file including four capital letters in a row — including all the lines containing the word "`UNIX`."

When combined with other commands (by "piping") `egrep` can be used for other kinds of searches too.

For example, the command

```
ls | egrep -e "[p-t]*"
```

lists the names of all files in a directory that start with one of the letters "p," "q," "r", "s" or "t"[2].

There are text editors on all the major platforms, including

- `Notepad++` on Windows
- `BBEdit` and `TextMate` on a Macintosh, or
- `vi` and `emacs` on UNIX or Linux

[2] and, possibly, also "P," "Q," "R," "S" and "T," depending on system settings

that allow you to supply a regular expression to search for a pattern in a text file being edited.

The details are different for each text editor (so you will need to read the documentation for this if you are interested in this feature).

Many modern programming languages have libraries that support the use of regular expressions in computer programs, so that you write programs that use regular expressions to perform sophisticated searches in text files.

Early"web browsers" for the internet also allowed users to supply regular expressions in search bars in order to search for files.

This is, generally, not the case today. However, ***web scraping*** (also called ***web data extraction***) is now recognized as a software technique for extracting information from web sites, and web scraping software supports the use of regular expressions to do this.

All you need is access to web scraping software, and a background in computer programming (with access the the software libraries mentioned on the previous slide) to make sophisticated searches for information over the internet!

Data Base Support

Data Base Development is a significant area in computer science and virtually all of us rely on data bases all the time during our studies and work — even though might might not always realize it!

Students interested in this topic can learn more by taking CPSC 471.

Commonly used ***Data Base Management Systems*** — including MySQL and Oracle's implementation of SQL — support the use of regular expressions to search for information in databases.

Lexical Analysis

"Lexical analysis" is part of the process of compiling a computer program — that is, generating machine language from it that can be directly executed.

In this part of the process, characters in the computer program are grouped together and replaced with "lexical items" or "tokens" (things with the words `variable` or `expression`).

The details are beyond the scope of this course — you can learn more about this by taking CPSC 411 — but the "modern" way to identify the sets of characters that should be recognized is to give regular expressions for them.

Going Beyond the Regular Languages

Many applications support for regular expressions that provide "extended" regular expressions whose languages are not regular languages at all! The most notable example of such an extension concerns ***marked expressions*** which are "subexpressions"' enclosed by parentheses. In order to see what a "marked expression" looks like, consider the regular expression

$$\texttt{(.\{5\})(.\{3\})} \tag{1}$$

includes two marked subexpressions. The entire regular expression matches a string with length eight. The first marked subexpression matches the prefix of the matched string with length five and the second marked subexpression matches the rest, that is, the suffix with length three.

If n is a digit from one to nine, and a given regular expression includes n or more matched subexpressions, then $\backslash n$ refers to the substring, of the string matched by the entire regular expression, that is matched by the n^{th} matched subexpression. For example, consider the regular expression at line (1), above. If this is used to match the eight-letter string `elephant` then $\backslash 1$ refers to the substring `eleph` and $\backslash 2$ refers to `ant`.

Extended "regular expressions" whose languages are provably not regular languages, include examples like

$$\texttt{(.*)}\backslash 1$$

which matches strings of the form $\omega\omega$, where ω is any string of symbols. It will be proved that the language of this "extended regular expression" is not a regular language in lectures, later in this course.

More Information

There is a tremendous amount of online information about the ways that various text editors use regular expressions to search for (and, sometimes, replace) text.

The UNIX command `man` can be used to display information about a particular UNIX command. For example, you can execute the command

```
man egrep
```

to discover more about how to use the `egrep` command.