Lecture #4: Regular Operations and Regular Expressions
Regular Expressions in Practice

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.

Regular expressions currently have a variety of applications — but they do not generally look like what has been described in the lecture notes. This supplemental document provides a bit more information about what regular expressions are like "in practice".

Dealing with Special Characters — "Escape" Sequences

The description of regular expressions, in the lecture notes, includes an inconvenient requirement about the alphabet, Σ — namely, that Σ does not include any of the special symbols

$$\Sigma, \lambda, \emptyset, \cup, \circ, ^\star, (, \text{ or }) \tag{1}$$

We certainly *will* wish to work with alphabets including some, or all, of these special symbols; in particular, it is generally unrealistic to require that the alphabet not include the left and right bracket.

This inconvenient requirement is generally eliminated by introducing one more special symbol, "\", and using it to include ***escape sequence*** in regular expressions. If you wish to include one of the special symbols in a regular expression as an element of the alphabet, then the special character "\" should be included immediately before it in the regular expression.

For example, if the symbol "Σ" belongs to the alphabet Σ then the language a regular expression

$$\backslash\Sigma$$

is a set of size one, including the symbol "Σ" — and not the entire input alphabet, which is the language of the regular expression

$$\Sigma$$

This introduces another problem — how do you recognize "$\backslash$" as a symbol in Σ? You do so using the escape sequence "$\backslash\backslash$" so that, if $\backslash \in \Sigma$ then the language of the regular expression

$$\backslash\backslash$$

is the set of size one, including the symbol "$\backslash$".

Now, we may include any symbol in the alphabet Σ that we want, and write regular expressions in $\Sigma^\star$.

Simplifying Regular Expressions — Which Makes Things More Complicated

Simplifications and Shortcuts

All but the shortest and simplest regular expressions can be long and hard to read — so several ***simplifications*** and ***shortcuts*** are often used to produce shorter strings, replacing the "regular expressions" defined so far — and that have the same languages as the regular expressions that they "simplify".

- ***Brackets can (sometimes) be left out*** — so that, for example, the regular expression (over the alphabet $\Sigma = \{\mathtt{a}, \mathtt{b}\}$)

  $$((\mathtt{a} \cup \mathtt{b}))^\star$$

  can be replaced by a "simplified" regular expression

  $$(\mathtt{a} \cup \mathtt{b})^\star$$

  — which is (strictly speaking) not a regular expression over Σ at all (because it cannot be produced using the construction, defined in the lecture notes, that is based on the definition of a "regular expression").

- ***The "$\circ$" symbol can be left out*** —so that, for example, the regular expression

  $$(\mathtt{a} \circ \mathtt{b})$$

can be replaced by a "simplified" regular expression

$$\texttt{(ab)}$$

—- or even by a further "simplified" regular expression

$$\texttt{ab}$$

- ***New notation is introduced for a common construction:*** If R is a regular expression over an alphabet Σ then

$$(R)^+$$

(which uses the new symbol, "$^+$") is a "simplified" language with the same language as the regular expression

$$(R \circ (R)^\star)$$

— that is, the language including of *one* or more strings in the language of R.

Once again, this "simplified regular expression" can (sometimes) be further simplified, and replaced by

$$R^+$$

More new notation can be introduced for another common construction: Let R be a regular expression over an alphabet Σ, and let k be a non-negative integer. Suppose that regular expressions $S_{R,0}, S_{R,1}, S_{R,2}, \ldots$ are defined as follows.

  - $S_{R,0} = \lambda$.
  - $S_{R,1} = R$.
  - For every integer i such that $i \geq 1$, $S_{R,i+1}$ is the regular expression

$$(S_{R,i} \circ R)$$

Then it can be proved, by induction on k, that $S_{R,k}$ is a regular expression over Σ for every non-negative integer k, and the language of $S_{R,k}$ is the set of strings that are the concatenations of k strings in the language of R.

As a shortcut, the simplified regular expression

$$(R)^k$$

is used instead of the regular expression $S_{R,k}$ (and understood to have the same language) for every non-negative integer k. This"simplified regular expression" can (sometimes) be further simplified, and replaced by

$$R^k$$

A Complication: Ambiguity

As noted above, brackets can "sometimes" be removed. Sometimes they should not because the language of the resulting "simplified regular expression" is not what you might expect!

Suppose, for example, that $\Sigma = \{\mathtt{a}, \mathtt{b}, \mathtt{c}\}$. Then

$$((\mathtt{a} \cup \mathtt{b}))^\star$$

is a regular expression, R_1, over Σ, whose language is the set of all strings that do not include any copies of "c" — so that the string ab is in the language of this regular expression. The string

$$(\mathtt{a} \cup (\mathtt{b})^\star)$$

is also a regular expression, R_2, over the alphabet Σ. The language of R_2 is the union of $\{\mathtt{a}\}$ and the set of strings in $\Sigma^\star$ that only include (zero or more) copies of "b" — so that the string ab is *not* in the language of this regular expression.

Now, if we remove the outer brackets of the above regular expressions then we obtain the "simplified" regular expressions

$$(\mathtt{a} \cup \mathtt{b})^\star$$

and

$$\mathtt{a} \cup (\mathtt{b})^\star$$

respectively — and the meaning of each is reasonably clear. However if we continue by removing the remaining brackets then we obtain the further "simplified" expression

$$\mathtt{a} \cup \mathtt{b}^\star \tag{2}$$

Now, since this has been obtained by "simplifying" both R_1 and R_2, its "language" could be *either* the language of R_1 or of R_2 — so it is not clear whether the string ab belongs to the language of this "simplified" regular expression, or not.

Let us say that a "simplified" regular expression is ***ambiguous*** if it can be obtained as the simplification of two or more regular expressions (over its alphabet) by using the simplification rules that have been given above — so that the simplified regular expression shown at line (2) is an "ambiguous" simplified regular expression.

Precedence Rules

You might not realize it, but you are already familiar with this kind of problem! Consider the "simplified" arithmetic expression

$$2 + 2 \times 2.$$

This might correspond to the arithmetic expression

$$(2 + (2 \times 2)),$$

which has value $2 + 4 = 6$ — or it might correspond to the arithmetic expression

$$((2 + 2) \times 2)$$

which has value $4 \times 2 = 8$. You will, ideally, remember that multiplication has higher ***precedence*** than addition — that is, multiplications should be applied *before* additions, so that the expression should be considered to be equivalent to

$$(2 + (2 \times 2))$$

and have value $2 + 4 = 6$.

Simplified regular expressions can be handled (that is, the effects of "ambiguity" overcome) in the same way:

- The star operation has higher ***precedence*** than the other operations in regular expressions — so that
  - $R_1 \cup R_2^\star$ has the same language as $(R_1 \cup (R_2)^\star)$ — and *not* generally the same language as $((R_1 \cup R_2))^\star$
  - $R_1 \circ R_2^\star$ has the same language as $(R_1 \circ (R_2)^\star)$ — and *not* generally the same language as $((R_1 \circ R_2))^\star$.
- Concatenation has higher ***precedence*** than union — so that $R_1 \circ R_2 \cup R_3$ has the same language as $((R_1 \circ R_2) \cup R_3)$ — and *not* generally the same language as $(R_1 \circ (R_2 \cup R_3))$.

When shortcuts "$(R)^+$" and "$(R)^k$" are used (for a regular expression R over Σ and $k \geq 0$ these should have the same precedence as the star operation.

Examining the "simplified" regular expression at line (2) once again, one can see that — since the star operation has a higher precedence than "$\cup$", the language of this expression should be the language of

$$\texttt{(a} \cup \texttt{(b)}^\star\texttt{)}$$

— that is, the union of $\{\texttt{a}\}$ and the set of strings that (only) include zero or more copies of the symbol "b".

Brackets must *not* be used when their removal would change the language of the regular expression, when precedence rules are applied. Keeping a few more brackets than are strictly needed can also make your "simplified" regular expression easier for someone else to understand.

What about "Parsing"?

If "simplified" regular expressions are also to be used in computer software than a version of the ***parsing problem*** for "simplified" regular expression (instead of "regular expressions") must be defined and solved. Furthermore, the ***parse tree*** produced from a "simplified" regular expression must correspond to the language of the "simplified" regular expression when the above precedence rules are applied.

These problems can be solved (so that a useful "parsing" algorithm for simplified regular expressions can be developed and used) — but the resulting parsing algorithm is more complicated than the parsing algorithm for regular expressions — and this is (well) beyond the scope of this course.

Industry Standards

As noted above, more practical use of regular expressions 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. As you will see below, quite a lot of changes were made to produce versions of "regular expressions" that could be included in command-line instructions or in code.

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 Python and 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 two (SRE — Simple Regular Expressions) and another (BSE — Basic Regular Expressions) are now 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, "L07_practical_regular_expressions.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}" L07_practical_regular_expressions.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

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

- `vi` and `emacs` on UNIX or Linux

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 provide 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

```
(.{5})(.{3})
```
(3)

— which 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 (3), 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

$$(.*)\backslash 1$$

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.