Applications of Regular Expressions

Brian A. Carter, Leah A. Hubert, and Arthur C. Walton

October 4, 2007

1 Introduction

Regular expressions aim to solve a simple problem: defining strings to match other strings. In the most simple of circumstances, a regular expression is the same as locating some substring within a document. For example, `document` is a valid regular expression that would match the word "document" in this sentence. However, the power of regular expressions extends far beyond matching a single word: regular expressions can be used to match complex patterns of symbols.

Consider, for example, the following regular expression:

$$\texttt{[0-9]}^*$$

This regular expression matches all numbers in base 10. Obviously, this could be a very useful task to perform. Consider that, for example, you need to convert all numbers in a document from base 10 to base 16. We could simply use a regular expression (this regular expression, in fact) to locate and extract all the base-10 numbers that occur in the document.

2 Regular Expressions in Web Search Engines

One use of regular expressions that used to be very common was in web search engines. Archie, one of the first search engines, used regular expressions exclusively to search through a database of filenames on public FTP servers[1]. Once the World Wide Web started to take form, the first search engines for it also used regular expressions to search through their indexes. Regular expressions were chosen for these early search engines because of both their power and easy implementation.

It is a fairly trivial task to convert search strings into regular expressions that accept only strings that have some relevance to the query. In the case of a search engine, the strings input to the regular expression would be either whole web pages or a pre-computed index of a web page that holds only the most important information from that web page. A query such as `regular expression` could be translated into the following regular expression.

$$(\Sigma^*\texttt{regular}\Sigma^*\texttt{expression}\Sigma^*)^* \cup (\Sigma^*\texttt{expression}\Sigma^*\texttt{regular}\Sigma^*)^*$$

Σ, then, of course, would be the set of all characters in the character encoding used with this search engine.

The results returned to the user would be the set of web pages that were accepted by this regular expression. Many other features commonly seen in search engines are also easy to convert into regular expressions. One example of this is adding quotes around a query to search for the whole string. The query `"regular expression"` could be converted into the following regular expression:

$$(\Sigma^*\texttt{regular expression}\Sigma^*)^*$$

Most of the other common features can also be easily converted into regular expressions.

Regular expressions are not used anymore in the large web search engines because with the growth of the web it became impossibly slow to use regular expressions. They are however still used in many smaller search engines such as a find/replace tool in a text editor or tools such as `grep`.

3 Regular Expressions in Software Engineering

Software applications are nowadays component-based for reusability. A problem with component based applications is how to precisely define the interplay on the component interfaces. Since a programmer does not have complete control over the integrating components, it can lead to unpredictable behavior. Solving this problem would make it easy to understand the specifications of the interfaces, as well as the correctness in implementation in terms of whether it adheres to the specifications. A precise and formal specification for component behavior is a necessity in order to automate black-box testing. From textual specifications, a finite state machine is produced, with its transitions labeled by messages, their data, and the constraints on their data. The usefulness of regular languages in this comes from the fact that regular languages deal in the finite which is true in reference to a computer.

Regular expressions are also used in test case characterizations. These test case characterizations are related to the programs directly, or to the corresponding models for it. A generic framework is developed where test cases are characterized, and coverage criteria are defined for test sets. Coverage analysis can then be done, and test cases and test sets can be generalized. Regular language theory is used to handle paths, and regular expressions are used over the terminals and nonterminals of the paths, which are called "regular path expressions." Operations done on the paths are restricted to the level of regular expressions, and the only paths of interest are the regular path expressions that are feasible. A regular expression is sufficient enough to abstractly describe a test case or a class of test cases, but sets of expressions require their own criteria. The use of regular language theory makes it easy for coverage analysis and test set generation.

The most expenditure for software stems from maintaining it rather than its actual development, and much of that expenditure goes to testing. Regression testing is an important part of the software development cycle. It is a process that is used to determine if a modified program still meets its specifications or if new errors have been found. There is research being done to improve regression testing to make it more efficient and specifically more economical. Regular language theory does not play a huge part in regression testing, but on an integration level, a relation can be established to finite automaton and regular languages and their properties.

4 Regular Expressions in Lexical Analysis

Lexical analysis is the process of tokenizing a sequence of symbols to eventually parse a string[2]. To perform lexical analysis, two components are required: a scanner and a tokenizer.

A token is simply a block of symbols, also known as a lexeme. The purpose of tokenization is to categorize the lexemes found in a string to sort them by meaning. For example, the C programming language could contain tokens such as numbers, string constants, characters, identifiers (variable names), keywords, or operators.

The best way to define a token is by a regular expression. We can simply define a set of regular expressions, each matching the valid set of lexemes that belong to this token type. This is the process of scanning. Often, this process can be quite complex and may require more than one pass to complete. Another option is to use a process known as backtracking—that is, rereading an earlier part of a string to see if it matches a regular expression based on some information that could only be obtained by analyzing a later part of the string. It is important to note, however, that the process of scanning does not produce the set of tokens in the document; it simply produces a set of lexemes. The tokenizer must assign these lexemes to tokens.

In tokenization, we generally use a finite state machine to define the lexical grammar of the language we are analyzing. To generate this finite state machine, we again turn to regular expressions to define which tokens may be composed of which lexemes. For example, to determine if a lexeme is a valid identifier in C, we could use the following regular expression:

```
[a-zA-Z_][a-zA-Z_0-9]*
```

This regular expression says that identifiers must begin with a Roman letter or an underscore and may be followed by any number of letters, underscores, or numbers.

However, there is one problem with the process of tokenization: we are unable to use regular expressions to match complex recursive patterns, such as matching opening and closing parentheses, for example. This is because these strings are not in a regular language and therefore cannot be matched by a regular expression. To deal with this problem, we must invoke the use of a parser—this is beyond the scope of this document.

After we have our text broken up into a set of tokens, we must pass the tokens on to the parser so that it can continue to analyze the text. Numerous programs exist to automate this process. For example, we could use `yacc` to convert BNF-like[1] grammar specifications to a parser which can be used to deal with the tokens produced through lexical analysis. Similarly, many lexical analyzers (often called simply *lexers*) exist to automate the process of scanning and tokenization—one of the best known is `lex`.

5 Conclusion

Regular expressions have numerous applications throughout computer science, ranging from the mundane (compilers) to the tasks we perform every day (using search engines). Over the years, the use of regular expressions in search engines has waned, giving way to more advanced techniques for indexing and matching search strings. However, regular expressions are still heavily used in the field of lexical analysis. In addition, the field of software engineering will likely always have a use for regular expressions.

[1] That is, using an expression similar to the Backus-Naur Form. This is a commonly used format to describe grammars.

References

[1] Wall, Aaron, *Search Engine History*, <http://www.searchenginehistory.com/>, October 3, 2007.

[2] "Lexical analysis," *Wikipedia*, <http://en.wikipedia.org/wiki/Lexical_analysis>, October 3, 2007.