This is the published version of the power point presentation:


Available from Deakin Research Online:

http://hdl.handle.net/10536/DRO/DU:30063702

Reproduced with the kind permission of the copyright owner

Copyright : 2008, IEEE
Object Oriented Regular Expressions

Henry Larkin

School of Information Technology and Electrical Engineering
University of New South Wales
ADFA, Canberra, Australia
research@logic.nu

Abstract

Regular expressions are used to parse textual data to match patterns and extract variables. They have been implemented in a vast number of programming languages with a significant quantity of research devoted to improving their operational efficiency. However, regular expressions are limited to finding linear matches. Little research has been done in the field of object-oriented results which would allow textual or binary data to be converted to multi-layered objects. This is significantly relevant as many of today’s data formats are object-based. This paper extends our previous work by detailing an algorithmic approach to perform object-oriented parsing, and provides an initial study of benchmarks of the algorithms of our contribution.

Keywords: regular expressions, object-oriented programming, XML.

1. Introduction

There exists a vast quantity of textual presentation formats used within the field of computers today. These include network protocols such as HTTP and SMTP through to formats such as HTML, XML and BibTeX through to languages such as Java, C++ and Perl. All have specific formats as to how information is represented in human-readable form. Regular expressions describe textual structures so that they can be matched and extracted for use in programming languages. They provide significant conveniences over using generic programming language operations in order to describe data formats.

As an example, in Figure 1 a regular expression syntax (using Perl's syntax [3]) is used to match a sender email address in SMTP. It first matches the text “From:” followed by at least one (“+”) alphanumeric character (“\w”). It then matches the exact text “@”, followed again by at least one alphanumeric character. It then matches a “.” and finally ends with at least one alphabet character (“^[a-z]+”). Such a regular expression could be used to match text such as “From: research@logic.nu”.

Figure 1: Example Regular Expression

One of the key problems of regular expressions in modern-day programming is that they lack object-oriented extraction. Object-oriented programming itself is based on the principle that many data structures encapsulate other data structures with each data structure serving a specific purpose. Little research has looked into developing a regular expression algorithm to represent and convert textual and binary formats of object-oriented data.

For example, the description in Figure 1 could be improved by simply specifying a separate description for an email address as well as providing a variable in which a resulting match should be placed. In Figure 2 for example, a regular expression would match the text “From:” followed by the regular expression described by an EmailAddress description. The resulting data is then placed in a variable senderEmail which may be part of an object in which multiple fields are stored. The regular expression then matches the text “To:” after a newline, followed again by another instance of the EmailAddress description which is stored in a variable receiverEmail. This regular expression in turn might be named Email and could be used, for example, within a repetition description to extract multiple Email objects to form a list. This greatly increases the power over traditional regular expressions which match single expressions at a time. It provides clear-cut definitions of how elements of an object are represented in presentation formats.
Figure 2: Example Expression with Variables

This paper continues on from the initial concept the authors proposed in [9] with the design and implementation of the algorithms involved in creating an object-oriented regular expression parser.

This paper is organised as follows. Section 2 will present recent and related work in the field of regular expressions. Section 3 will then detail the novel design for implementing object-orientation in regular expressions. Section 4 will then discusses performance benchmarks and issues from the prototype implementation. Section 5 then outlines future work and concludes the paper.

2. Related Work

Regular expressions have been a widely researched topic. General regular expression research [11][7][6][2] covers a variety of implementations of regular expressions in programming languages as well as optimisations on improving the speed of regular expression processing. This existing research focuses on parsing and extracting simple text, with no ability to create objects or represent objects by producing the text as output. Regular expression tools have been created for more specific purposes also. Lex [10] and Yacc [8], for example, are tools for another specific type of text parsing. They parse the text representation of programming languages into syntax trees in order for compilers to produce machine code. While these applications of regular expressions are useful, they still contain the problems previously mentioned.

Another specific area of regular expression research has been in XML. XML has been gaining much attention in recent years. Parsers specifically designed for XML-structured data have been designed to convert XML into objects in various programming languages [4][1][5]. For example, the Xtatic language [4] is an extension to the C# programming language. It combines the tree-structured data model of XML with the classes and objects model used in conventional object-oriented programming languages. Figure 3 shows an example block of XML and a corresponding object created using Xtatic.

XML parsers generate objects and usable variables, but do so only for XML. These parsers are not adaptable to regular expressions which allow data to be represented by any textual presentation format. In contrast to XML, regular expressions themselves are often used to describe XML structure, and with the introduction of variables could create parsers for XML offering identical functionality to these XML-specific parsers.

Figure 3: An XML fragment and the corresponding Xtatic value

3. Design

To create regular expression descriptions for use in object-oriented languages, this paper presents several key concepts: Simple and Complex expressions, an Assignment data structure, a Match Tree and finally a tiny parsing algorithm.

An expression is a representation and associated logic of the criteria required for a particular match. A match itself is the specific identification of data as matching a given expression which can be later used to extract that data and interpret it. Expressions do not store any actual matched data, only a Match object does that. An Expression simply defines how a match will occur and how to later interpret the data. Expressions come in the form of inbuilt simple expressions (such as matching a number or a specific string of text), inbuilt complex expressions (such as sequence, choice, iteration) which can contain other expressions, and finally user-defined expressions (using any mixture of simple and complex expressions).

Every expression has two specific algorithms: getMatch and getData. The getMatch algorithm will contain logic to check whether a match exists at a given position in the Match Tree. The algorithm will return the resulting Match object if a match is possible. In the case of Complex Expressions (which may contain multiple smaller matches), the resulting Match object may contain several child matches which form part of the greater match.
The *getdata* algorithm provides a means of retrieving the actual data of a match. Data is not retrieved and constructed at the time of a match because there may be many possible matches at various sections of the document which are incorrect when the document being converted is analysed as a whole. The *getdata* algorithm for any expression is given a Match object which was generated by the expression with the *getMatch* algorithm, containing information about what elements in the Match Tree in particular were matched. This is then used to generate the resulting data, which is returned by the algorithm. For example, the *getdata* algorithm of a Number expression would take the Match object (indicating that objects “4”, “5”, and “6” form the match), and then convert them into an integer object and return it to the caller.

### 3.1.1. Simple expressions

Simple Expressions are inbuilt expressions to aid text parsing. They exist both as the building blocks of Complex expressions and to optimise common expressions. They also allow certain programming language-specific primitive types to be constructed. Three minimal expressions have been considered: Exact String, String and Number. An Exact String will match exactly the string which is defined by a user. A String expression will match any combination of characters based on given character sets (e.g. alphabet, numeric, alphanumeric, punctuation). A Number expression will match any legitimate sequence of digits, with the option of allowing a decimal place.

### 3.1.2. Complex expressions

Complex expressions are expressions which can contain other expressions. They are used differently from Simple expressions in that only Complex expressions may be contained within the ruleset and can be matched by the engine. Simple expressions, on the other hand, are matched by Complex expressions. This is an optimisation choice, as in testing it became evident that it was computationally cheaper to rematch raw characters than to iterate over a Match Tree for all combinations of those characters which generated potential matches. This is discussed later in this paper.

There are three algorithms of importance in Complex expressions: *getmatch*, *getdata* and *loadExpression*.

As a Complex expression contains other expressions through Assignment objects (discussed next), the *getmatch* algorithm simply calls the Assignment’s *getmatch* method for each expression and uses that accordingly. For example, an Iteration expression will continually try to match a given expression and separator until no more can be found.

The *getdata* algorithm for a Complex expression is much simpler than its Simple expression counterpart. This is because a Complex expression is simply a collection of other expressions. Therefore the algorithm simply iterates over the Match's list of child Match objects, and calls the *getdata* algorithm of each of them in turn. The exception to this is the Named expression, which will be discussed later.

Complex expressions have a further algorithm, *loadExpression*. The *loadExpression* algorithm is provided to load an expression into the given ruleset. Many expressions are made up of smaller expressions, and the *loadExpression* algorithm, depending on the type of expression, will break these down and add them individually to the ruleset. This way each individual Complex expression is matched independently, and the *getMatch* algorithm of a Complex expression (through the Assignment object), simply checks if that individual expression has already been matched and is in the Match Tree. The reason for this is to prevent recursive loops. Consider the case where, in creating a calculator expression parser, an Addition object would contain two values, where either value could be a number, or another Addition object, or a Subtraction object, etc. If an Addition object can contain another Addition object as the very first item it matches, then it would continue looking for an Addition within an Addition indefinitely. The only approach to avoid this situation is to insist on there being *substance* before creating a match, and for there to be substance there must already exist the smaller object in the Match Tree already. Thus the reason for this approach.

There are five basic Complex Expressions which are the building blocks for any other expression: Sequence, Selection, Iteration, Named and Reference expressions.

The Sequence expression is merely an expression consisting of one or more other Simple or Complex expressions. It matches each of them sequentially without any characters in between.

The Selection expression is similar to the Sequence expression in that it contains one or more other Simple or Complex expressions. It, however, only matches the first expression to successfully match. The Selection expression is also useful in creating precedence selections (such as that in math operations). For example, an Addition expression could specify higher precedence expressions before
lower ones (e.g. a Multiply would be before another
Addition in the Selection expression).

The Iteration expression allows a single expression
to be matched repeatedly, with an optional separator
expression. A minimum and maximum count may also
be specified.

The Reference expression is a placeholder for a
link to a Named expression. This is fundamental to
allowing objects to become part of larger objects. An
eexample of a reference to a Named expression
“EmailAddress” is given in Figure 4.

“From” + (senderEmail=EmailAddress) + “in”

Figure 4: Reference Expression Example

A user themselves (one creating a ruleset for a
specific data representation) defines expressions in
terms of Named Expressions. These are expressions
which are aliased with a name and can instantiate a
given data type. For example, a Named Expression for
an email address, given an alias of “address” and
generating an Email Address object is listed in Figure
5. It includes two Assignments which map a result to a
given field within the generated Email Address object.

Named expressions have a slightly modified
getData
algorithm in order to construct objects. The
dataGet
algorithm will first create the data type
indicated in the Named expression (if one is provided)
and add it to the Variables Stack with the name
provided with the Named expression (the Variables
Stack may have more than one variable with the same
name, with the most recently added the one used).
Following this, the getData algorithm of the inner
expression is called. Finally the variable is removed
from the Variables Stack and returned.

3.1.2.1 Assignment

The Assignment object is fundamental to Complex
expressions. Technically speaking, every Complex
expression contains Assignment objects which in turn
contain other linked expressions. It defines how a
smaller match within a Complex expression will be
applied to a field within the resulting object. For
example, consider the Named expression in Figure 5.
Two Assignments (in brackets) are used to specify
where to assign the resulting data of smaller
expressions. The first maps a string of text to the
“user” field, and the second maps a string of text and
decimal points to a “domain” field. This assignment
step is vital to the construction of objects, yet has not
been suggested in previous research on regular
expressions.

EmailAddress address :=
(user=\w+) + “@” + (domain=\w+.[a-z]+)

Figure 5: Email Address Named Expression

The structure of an Assignment object (Figure 6)
consists of a location, an expression, and whether the
field is optional or not. The location is a field or
algorithm name specified within the resulting object.
It can include variable references as precursors to the
field or method name. For example, Figure 7 shows
two different precursors, the variable aliased with
“address”, and the keyword “this” which represents
the very last variable created. Whenever the parser is
going through the getData phase, a Variables Stack is
passed as a parameter into each getData algorithm.
This stack contains a list of variable names and
associated objects which have been generated so far.
Every time a new object is created which is named, it
is a variable name for smaller expressions to reference
as required. This allows smaller expressions to assign
data or call algorithms of the larger created objects if
necessary. The special keyword “this” can be used to
reference the very last variable created (the most
immediate object being constructed).

Figure 6: Assignment Object Structure

EmailAddress address :=
(address.user=\w+) + “@” +
(this.domain=\w+.[a-z]+)

Figure 7: Use of Variable Precursors

The Assignment object specifies the expression
which must match, and a flag to indicate whether the
field is optional or not. If the field is optional, a larger
expression containing the Assignment can still match
even if the data does not exist.

The Assignment object has its own getMatch
algorithm, and is responsible for determining if a
match exists at a given position and where the match
takes the search position up to. In other words, the
getMatch algorithm will return the new Match Tree
position (the same number in the case of an optional
expression which was unmatched), and -1 if there was
no match. The algorithm performs as such:
for a Simple expression, get a match instantly on that position by calling the Simple expression's `getMatch` algorithm.

for a link to a Named expression, see if that exact Named expression object is in the Match Tree in a Match object.

for a link to a Complex expression, see if that exact Complex expression object is in the Match Tree in a Match object.

Finally, the Assignment object also defines a `getData` algorithm. This algorithm takes data from a given expression's `getData` method and assigns it to the appropriate place inside an object. If no assignment location is specified, then the data is simply returned to be consumed as part of a larger block of data. The algorithm works as such:

First, the algorithm will check if there is any actual assignment to take place. If there is no assignment location then the code will simply return the data.

Secondly, the code will search for the base object inside the Variable Stack. If the base object is the word "this", then the last variable in the list is used.

From there, the code will iterate over every variable (separated by "." symbols) and retrieve them from inside each parent object.

The last item is then either a field name or an algorithm accepting a single Object data type, which is where the data which was found to represent the expression of the assignment will be placed.

### 3.2. The Match Tree

The Match Tree is a fundamental data structure to the parsing algorithm (detailed next). The Match Tree contains a list of every unique match ever found during a document parse. When first created, it takes a given body of text and creates Match objects for every character in the text as the smallest unit measurable. It creates branches for each match's starting position and then further creates leaves for each match at that position, sorted largest to smallest. This allows the largest (most successful) match for any position to be returned first.

A Match object itself simply contains the expression which was matched (so as to call the correct `getData` algorithm later as needed) and the start and end index positions of the original document where a match occurred (for Simple expressions) or a list of child Match objects (for Complex expressions). The Match Tree is simply a huge tree of document index positions and all matches at each position.

The Match Tree has two algorithms which then perform all work: `addElement` and `getElements`. The `addElement` algorithm takes a new Match object and adds it into the Match Tree. It does this in two steps:

1. The code will scan the tree for a duplicate match (as only unique matches are stored).
2. The new Match object will be added to the tree in the correct position.

The code will return true if the new Match was successfully added (if the exact same expression wasn't already matched), so that the Parsing Algorithm stops after no new matches are found overall.

The `getElements` algorithm returns a list of all elements at a given index position (in order from largest to smallest because of the natural order of the list). As there will always be characters in the stack, there will always be at least one result. Optionally a specific type of expression to search for can be passed to the algorithm, with a list of only Match objects for that expression returned.

### 3.3. The Parsing Algorithm

The parsing algorithm is used to continually search for matches given a ruleset of expressions and a text document until no other matches can be found. The algorithm is listed in pseudo-code in Figure 8. Quite simply, it first generates an initial Match Tree given the text document. It will then continue searching for matches by going over the ruleset of Complex expressions and checking if it can be matched for any index position within the Match Tree. If the document is syntactically correct, then one match should cover the entire document, and its `getData` algorithm can then be called to generate the resulting data type.

```
1. Generate a new Match Tree with the given text.
2. Repeat while there are new matches:
   a. Iterate over each Complex expression in the ruleset
      i. Iterate over every raw character position in the Match Tree and call the Complex expression's match algorithm, adding the resulting match to the Match Tree.
3. Call the `getData` algorithm on the first stack element and return the result.
```

Figure 8: Parsing Algorithm
4. Performance

The parser, as described in Section 3, was implemented in the Java programming language as a proof-of-concept and to benchmark initial performance. It is suspected that because of the high storage costs of every match that only small documents could be parsed within reasonable time and memory constraints. A simple ruleset was created with two Named expressions. The first is to match a number, followed by “..” followed by another number (e.g. “1..5”). The second Named expression consists of an Iteration expression over the first Named expression, separated by colons. Sample text was then created consisting of the same text (“1..5”) and repeated for several sets. Table 1 shows the results in both time and the number of Match objects generated for each of these experiments. As can quickly be seen, as the size of text increases, both the time and number of Match objects increase dramatically.

Table 1: Benchmarks

<table>
<thead>
<tr>
<th>Sets</th>
<th>4</th>
<th>8</th>
<th>16</th>
<th>32</th>
<th>64</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time</td>
<td>2ms</td>
<td>4ms</td>
<td>12ms</td>
<td>28ms</td>
<td>126ms</td>
</tr>
<tr>
<td>Match</td>
<td>206</td>
<td>514</td>
<td>1418</td>
<td>4378</td>
<td>14906</td>
</tr>
</tbody>
</table>

5. Conclusions and Future Work

This paper presents a novel approach to parsing text using an object-oriented regular expression approach. This is increasingly relevant in both object-oriented programming languages and in object-oriented data representation.

There are two key areas which have been identified for future work. Firstly, the algorithm, while it does perform the task, is considerably slow to the point where large documents would take minutes to parse. To overcome this it is necessary to dismiss multiple overlapping Simple expressions based on all potential options before and after said expression. For example, consider Figure 9 consisting of an example Unix password file. Each line consists of fields of a single user, and each field is separated by a colon “:”, and each user is separated by an end-of-line marker. If it is known that only a colon or an end-of-line marker can break a chain of characters, then instead of a simple String match beginning at each possible character, only a match beginning after a colon or end-of-line character should be created. This could drastically reduce false positives from Simple expressions.

Furthermore, the current design does not consider any form of error feedback or correction. If a document is parsed which is missing any information, an appropriate response and calculated guesses as to what missing objects would most easily generate a complete match would be extremely useful. Further research into these issues will be pursued.

Bibliography