Corralling Complex Regular Expressions

Allen Baker

Complex regular expressions are difficult to construct manually and virtually unreadable. This paper describes a method for constructing correct and maintainable complex regular expressions using only basic tools like a text editor and a string substitution tool.

What is the problem?

Regular expressions longer than a few dozen characters resemble gibberish. Here is one that is not too bad:

```
(((?<Qualifier>(\w)+)[\s\n\r]*(\.)[\s\n\r]*)*(?<Name>(?!\bif\b|\bfor\b|\bwhile\b|\bswitch\b|\bt
ry\b|\bcatch\b)(\w)+)[\s\n\r]*(?<Arguments>(\(.*\))))
```

Simplistic regular expression to find method calls in a Java source code file

We can arrange this regular expression structurally to make it easier to create, modify, and understand like this:

```
(
   (
      (?<Qualifier>
         (\w)+
      )
      [\s\n\r]*(\.)[\s\n\r]*
   )*
   (?<Name>
      (?!
         \bif\b|\bfor\b|\bwhile\b|\bswitch\b|\btry\b|\bcatch\b
      )
      (\w)+
   )
   [\s\n\r]*
   (?<Arguments>
      (\(.*\))
   )
)
```

Simplistic regular expression to find method calls in a Java source code file (structured)

While this helps, what if our regular expression is more like this one for robustly matching Java method declarations:

```
(?:(?<ACCESS>((public|protected|private)\s+)?))(?:(?<MODIFIERS>((abstract|final|native|trans
ient|static|synchronized)\s+)*))(?:((?<TYPEPARAMETERS>(<\s*(([A-Za-z]\w*(\[])?|\?)(\s+extend
s\s+[A-Za-z]\w*(\[])?)?|(([A-Za-z]\w*(\\[])?|\?)(\s+extends\s+[A-Za-z]\w*(\[])?)?\s*<\s*([A-
Za-z]\w*(\[])?|\\?)(\s+extends\s+[A-Za-z]\w*(\[])?)?\s*>))(\s*,\s*(([A-Za-z]\w*(\[])?|\?)(\s
+extends\s+[A-Za-z]\w*(\[])?)?|(([A-Za-z]\w*(\[])?|\?)(\s+extends\s+[A-Za-z]\w*(\[])?)?\s*<\
s*([A-Za-z]\w*(\[])?|\?)(\s+extends\s+[A-Za-z]\w*(\[])?)?\s*>)))*\s*>))?\s+)?)(?:(?<RETURNTY
PE>(([A-Za-z]\w*(\\[])?|\?)(\s+extends\s+[A-Za-z]\w*(\[])?)?|(([A-Za-z]\w*(\[])?|\?)(\s+exte
nds\s+[A-Za-z]\w*(\[])?)?\s*<\s*([A-Za-z]\w*(\[])?|\?)(\s+extends\s+[A-Za-z]\w*(\[])?)?\s*>)
))\s+)(?:(?<METHODNAME>[A-Za-z]\w*))(?:\s*(?<PARAMETERLIST>\(\s*(((((([A-Za-z]\w*(\[])?|\?)(\
s+extends\s+[A-Za-z]\w*(\[])?)?|(([A-Za-z]\w*(\[])?|\?)(\s+extends\s+[A-Za-z]\w*(\[])?)?\s*<
\s*([A-Za-z]\w*(\[])?|\?)(\s+extends\s+[A-Za-z]\w*(\[])?)?\s*>))\s+[A-Za-z]\w*)|((([A-Za-z]\
w*(\[])?|\?)(\s+extends\s+[A-Za-z]\w*(\[])?)?|(([A-Za-z]\w*(\[])?|\?)(\s+extends\s+[A-Za-z]\
w*(\[])?)?\s*<\s*([A-Za-z]\w*(\[])?|\?)(\s+extends\s+[A-Za-z]\w*(\[])?)?\s*>))\.\.\.\s+[A-Za
-z]\w*))*|(((((A-Za-z]\w*(\[])?|\?)(\s+extends\s+[A-Za-z]\w*(\[])?)?|(([A-Za-z]\w*(\[])?|\?)
```

```
(\s+extends\s+[A-Za-z]\w*(\[])?)?\s*<\s*([A-Za-z]\w*(\[])?|\?)(\s+extends\s+[A-Za-z]\w*(\[])
?)?\s*>))\s+[A-Za-z]\w*)(\s*,\s*((([A-Za-z]\w*(\[])?|\?)(\s+extends\s+[A-Za-z]\w*(\[])?)?|((
[A-Za-z]\w*(\[])?|\?)(\s+extends\s+[A-Za-z]\w*(\[])?)?\s*<\s*([A-Za-z]\w*(\[])?|\?)(\s+exten
ds\s+[A-Za-z]\w*(\[])?)?\s*>))\s+[A-Za-z]\w*))*)*(\s*,\s*((([A-Za-z]\w*(\[])?|\?)(\s+extends
\s+[A-Za-z]\w*(\[])?)?|(([A-Za-z]\w*(\[])?|\?)(\s+extends\s+[A-Za-z]\w*(\[])?)?\s*<\s*([A-Za
-z]\w*(\[])?|\?)(\s+extends\s+[A-Za-z]\w*(\[])?)?\s*>))\.\.\.\s+[A-Za-z]\w*))?)\s*\)))(?:(\s
+(?<THROWER>throws\s+[A-Za-z]\w*(\s*,\s*[A-Za-z]\w*)*))?)
```

"Mr Ugly" - Regular expression to find method declarations in a Java source code file

Good luck sitting down and writing that out manually. For brevity, we will henceforth refer to this regular expression as "Mr Ugly". Although structuring this regular expression helps, it contains so many nested groups that it is nearly impossible to construct and modify it manually or to comprehend it.

What is the solution?

There are tools available that attempt to make it easier to construct something like Mr Ugly, but I have found them unsatisfactory at reducing the construction complexity. Some even make the task more difficult. I use a simple hierarchical substitution method for building complex as well as simpler regular expressions. This method, which works much like the C++ preprocessor #define expansion process, greatly simplifies construction, modification, and comprehension. Consequently, it enables the construction and maintenance of regular expressions that are more comprehensive and suffer from far fewer oversights and other errors than is possible using common methods.

Without resorting to any fancy tools, we can apply this method to any sized regular expression. The idea is very simple and is conceptually just like the programming process of creating a library of reusable functions and then assembling those functions into a unique sequence and configuration to create a unique application.

Begin with syntax. I use the following: a string enclosed in curly braces represents one token.

```
{iAmOneToken}
```

Syntax of a token

I run the words together and capitalize the first letter of all but the first (camel case) but this is not necessary. We could place spaces or any other punctuation between words and capitalize whatever we wanted.

The next step is to define what the token means, its value. Here is an example that defines the token `{identifier}` as a regular expression that matches any sequence of characters that starts with a letter – either upper or lower case – and is followed by zero or more "word" characters – letters or digits.

```
{identifier}=[A-Za-z]\w*
```

A token definition: *token* `=` *definition*

Now that we know how to define a token as a small regular expression, let us define several primitive tokens that we can reuse to construct a more complex regular expression. We will start with tokens for "required" whitespace `{rqdWs}` and "optional" whitespace `{optWs}` followed by the `{identifier}` from above and a couple of others.

```
{rqdWs}=\s+
{optWs}=\s*
{identifier}=[A-Za-z]\w*
{string}="([^"]|\\")*"
{elipsis}=\.\.\.
{indexer}=\[]
```

"Primitive" tokens

The next step is to begin combining tokens into compound tokens. To do this we simply define a token as an assembly of other tokens and maybe some regular expression code blended in. Here is an example where we create a token for a comma that separates the items in a list. We want the items in our lists separated by commas but we also want to allow a lot of, some, a little, or no whitespace on either side of the commas – in other words, we want optional whitespace around our commas. Therefore, we include the token for optional whitespace that we defined above on both sides of the comma character.

```
{commaSeperator}={optWs},{optWs}
```

A token for a comma that separates items in a list

Now if we replaced the optional whitespace tokens `{optWs}` with their definition we would get this:

```
\s*,\s*
```

Regular expression for a comma that separates items in a list

This regular expression will match all those commas separating our list items regardless of how much if any whitespace surrounds them. It will match all the separations between the items in this list for example:

```
This , is, a ,list,of   , comma/whitespace-separated ,    words
```

List of comma/whitespace separated strings

Note that if we are defining a regular expression to process small lists like this, it may be easy to remember to put the optional whitespace around the comma. However, in a very large regular expression construct with many different lists to match, it is very easy to overlook a small detail like this. It is also very hard to troubleshoot a large regular expression to find a simple error like that.

Now let us add an additional layer of token definition and then work through the process of generating a complete regular expression from it. Here we define a token `{list}` that is composed of some of the other tokens we have already defined. This token will match any arbitrary comma separated list of identifiers.

```
{list}={identifier}({commaSeperator}{identifier})*
```

Token defined to match a list of identifiers

The first step is to replace the tokens in the definition with their definitions. I wrote a very simple string substitution program to do this but the search and replace function of any text editor does the job just as well.

When we replace the {identifier} tokens with their definitions, our regular expression looks like this:

```
{list}=[A-Za-z]\w*({commaSeperator}[A-Za-z]\w*)*
```

{identifier} tokens replaced by their definitions

Now when we replace {commaSeperator} with its definition, we get the following:

```
{list}=[A-Za-z]\w*({optWs},{optWs}[A-Za-z]\w*)*
```

{commaSeparator} token replaced by its definition

We still do not have a valid regular expression because there are a couple of our tokens – the optional whitespace tokens {optWs} – in the middle of it. Therefore, we have to do another iteration of token replacement to replace the optional whitespace tokens with their definitions:

```
{list}=[A-Za-z]\w*(\s*,\s*[A-Za-z]\w*)*
```

{optWs} tokens replaced by their definitions

Now we have a valid regular expression that will match any arbitrary comma separated list of identifiers.

That is all there is to it. Now let us see how this method tackles a regular expression like Mr Ugly. We want a regular expression that can match any method declaration that is at least as complex as either of these two:

```
public static synchronized <TRet,T2,T3,T4> TRet methodName(T1 p1,T2 p2,T3 p3,T4 p4) throws
iOException,NumberFormatException

public static native final <TRet,T1 extends Comparable,T2,T3,T4> TRet<TPrm> methodName(List<?
extends T> list,T1 p1,T2 p2,T3 p3,T4 p4)
```

Complex Java method declarations

We also need it to match any method declaration that is as simple as this:

```
void methodName()
```

Simple Java method declaration

To our primitives from above, we add several more of increasing complexity. Each token in our list should be decomposable into a combination of tokens from below it in the list. Here is our complete list of tokens:

```
{accessGroup}=(?:{access})
{access}=(?<ACCESS>((public|protected|private){rqdWs})?)

{modifiersGroup}=(?:{modifiers})
{modifiers}=(?<MODIFIERS>((abstract|final|native|transient|static|synchronized){rqdWs})*)

{typeParametersGroup}=(?:{typeParametersOption})
{typeParametersOption}=({typeParametersSet}{rqdWs})?
{typeParametersSet}=(?<TYPEPARAMETERS>{typeParameters})?
{typeParameters}=(<{optWs}{typeSpec}({commaSeperator}{typeSpec})*{optWs}>)

{returnTypeGroup}=(?:{returnType}{rqdWs})
{returnType}=(?<RETURNTYPE>{typeSpec})

{methodNameGroup}=(?:{methodName})
{methodName}=(?<METHODNAME>{identifier})

{parameterGroup}=(?:{optWs}{parameterList})
{parameterList}=(?<PARAMETERLIST>\({optWs}({oneParameter}*|{multipleParameters}){optWs}\))
{multipleParameters}={normalParameter}({commaSeperator}{normalParameter})*)*({commaSeperator}{varArgsParameter})?
{oneParameter}={normalParameter}|{varArgsParameter})
{varArgsParameter}={varArgsType}{rqdWs}{identifier})
{normalParameter}={typeSpec}{rqdWs}{identifier})
{varArgsType}={typeSpec}{elipsis}

{throwGroup}=(?:{throwerOption})
{throwerOption}=({rqdWs}{thrower})?
{thrower}=(?<THROWER>throws{rqdWs}{thrownList})
{thrownList}={typeName}({commaSeperator}{typeName})*

{typeSpec}={regularType}|{parameterizedType})
{parameterizedType}={regularType}{optWs}<{optWs}{regularType}{optWs}>)
{regularType}={baseType}{typeExtension}?
{typeExtension}=({rqdWs}extends{rqdWs}{concreteType})
{baseType}=({concreteType}|{unknownType})
{unknownType}=\?
{concreteType}={typeName}({indexer})?
{typeName}={identifier}

{commaSeperator}={optWs},{optWs}

{rqdWs}=\s+
{optWs}=\s*
{identifier}=[A-Za-z]\w*
{string}="([^"]|\\")*"
{elipsis}=\.\.\.
{indexer}=\[]
```

All the tokens defined

We build each token in the list from a blend of raw regular expression code and tokens that are defined further down in the list.

Next, we place some tokens into a separate list and "expand" them. Here is the list of some of our tokens that when expanded and concatenated together, will form Mr Ugly.

```
{accessGroup}{modifiersGroup}{typeParametersGroup}{returnTypeGroup}{methodNameGroup}{parameterG
roup}{throwGroup}
```

The tokens for generating Mr Ugly

The final step to generate Mr Ugly is to run a recursive or iterating string substitution process on this list using our list of token definitions as the list of string substitutions to make. You could do this manually by pasting in the old/new (token/definition) string pairs in order into a text editor's global search and replace dialog. However, that would be unnecessarily painful because this is an easily automated task.

To automate, we need a tool that can read a list of pairs of strings where each pair is composed of the old string to replace and the new string to replace it with, and then perform the substitutions in order. I built a simple utility named "Substitute" that does just that, but there are numerous free, downloadable programs to do this available. Many text editors provide this capability as well.

I will use my "Substitute" program as the example. Substitute runs as a stand-alone program that reads an input file of string pairs and applies them to a target file. To use it, we copy the entire list of token definitions into a text file; let us call the file "subs.txt". Next, we copy our list of the tokens for generating Mr Ugly into a separate text file; let us call the file "MrUgly.txt". Then we run Substitute, telling it to use subs.txt as a file of string replacement pairs and MrUgly.txt as the file to apply the substitutions to.

```
Substitute –fsubs.txt MrUgly.txt
```

The command line instruction for running Substitute

Substitute reads each line of subs.txt in order. For each line, Substitute uses the equals character ("=") to separate the line into two strings. It performs a global search and replace on MrUgly.txt using the first string on the line (the part before the equal sign) as the old string and the second string on the line (the part after the equal sign) as the new string. When Substitute finishes, the file MrUgly.txt contains the regular expression we call Mr Ugly.

What did we gain?

This method does several things for us. Most importantly of course, it allows us to correctly generate arbitrarily large and complex regular expressions from simpler building blocks. Additionally though, the method provides us with an intermediate representation of our regular expression that is far more readable and maintainable than the raw regular expression. It is a simple matter to go into a specific token, tweak its definition, and rerun Substitute (or similar program) to regenerate the regular expression. It is substantially easier than trying to edit a large and complex regular expression directly.

This method brings with it all the same advantages as other modular or "building block" methods. These include:

- It makes it easier to get the regular expression right the first time, or at least much earlier. The replication of tedious detail by way of string substitution makes it possible to focus on those details one time in a small sub-expression, get it right there, and then substitute the well-done and sound code in multiple places in the final regular expression.
- It is much easier to debug. Errors localize to a single token's definition, which is much smaller and easier to analyze and grasp than the raw regular expression.
- It is easier to edit a small portion of the regular expression without causing collateral damage to the remainder of the expression. It is very easy within a raw expression to inadvertently include or remove a parentheses or some other group-affecting character in such a way as to wreck the correctness of a large remaining chunk or even the entire raw regular expression. Additionally, a small piece of the regular expression frequently is replicated several times in a raw expression, and trying to edit all those replicated instances directly, dramatically compounds the chances of causing collateral damage. It is much better to edit it one time in one place and let automated string substitution take care of "fixing" all the replicated instances.
- It is easier to understand a complex regular expression if it is broken down into small, independent pieces that can be comprehended in their entirety.
- It is easier to reuse the small independent pieces of regular expression that comprise a large and complex raw expression.
- It is easier to test small independent pieces of regular expression because they can be tested in isolation.
- There is less code to write. A piece of regular expression only has to be written one time but can be replicated many times in a complex raw regular expression and reused in numerous other regular expressions.

The method described in this paper is a software design technique for regular expressions that emphasizes separating the functionality of complex regular expressions into independent, interchangeable modules or tokens, such that each contains everything necessary to execute only one aspect of the desired functionality. Conceptually, tokens represent a separation of concerns, and improve maintainability by enforcing logical boundaries between components. As with other areas of Software Engineering, the modularization that this method promotes brings manifold benefits to the process of constructing, modifying, and comprehending regular expressions.