Editing Source Code Programmatically and Tokenizing Blaise Syntax via Regular Expressions

Jason Ostergren, The University of Michigan

Introduction

The Health and Retirement Study (HRS) is a national longitudinal survey that utilizes a Blaise CAI instrument for biennial HRS interviews of one to two hours in length given to around twenty thousand participants. The source code for the HRS is lengthy (about nine megabytes of text). Every two years, HRS makes extensive changes to the instrument. Sometimes there are repetitive changes that follow a pattern which makes any individual change simple, but greatly multiplies the chance of mistakes due to the quantity of code involved. These can also consume large quantities of programmer time. HRS has made a number of attempts over the years to streamline or automate the process of making certain kinds of source code changes, including one attempt, called Source Editor, which was the subject of a previous IBUC paper in 2007.

The main drawback of our previous method was that it became complicated to run and maintain. A lot of the complication was due to the tokenizing process, so HRS developed a regular expression to tokenize the entire Blaise source code. This also improved performance, so that the entire operation could be run in less than a minute, which aided the iterative development of a solution. HRS constructed a program around this simple method of tokenization to perform operations such as formatting the code, removing comments, updating descriptors, and adding new languages programmatically.

The quest for programmatic means of effecting source code updates

In addition to being very long, the HRS source code can be untidy. A number of different programmers and translators have worked on it over the course of two decades in two different survey programming languages, leaving many artifacts of past work and different programming styles. The formatting of the Blaise include files varies, as does the incidence of comments and sometimes metadata components of variables. Automatic formatting, comment adjustment or removal, and standardization of variable definitions (e.g., no missing descriptors allowed, all field texts show language identifier) can make it easier to maintain a certain degree of orderliness even with many hands in the code.

HRS has had a longstanding interest in revamping the variable metadata in the instrument during the reprogramming that occurs between biennial waves. In particular, HRS has gone to great effort to update descriptors from wave to wave, including constructing the aforementioned Source Editor. This sometimes proceeds on a field by field basis, but more often the survey designers want to see and alter large groups of descriptors at once, a task which requires some systematization of effort. In addition, HRS has performed wholesale changes to the field tags on some occasions and has regularly discussed (but not yet implemented) a refactoring of every variable name in the instrument to correct the sequence of the numeric part of the variable names (which becomes unordered as fields are moved around during redesigns). Because there is a desire to perform such operations at more frequent intervals, it is necessary to have automated means to do so.

Previous attempts and the need for yet another new tool

The idea of editing or authoring Blaise instruments through some exterior program is obviously nothing new, as previous IBUC papers show. HRS has tried multiple variations on this, including the conversion tools written to convert SurveyCraft code when Michigan adopted the Blaise system, the Source Editor, and an authoring product that generates Blaise code from a database. Oftentimes, these efforts to build tools are driven by the fact that specs rarely arrive in a timely manner or entirely complete, and it is expected in advance that the window for turning them into reliable Blaise code will benefit from any time-savings that can be gained through automation.

However, the chief reason for the new effort, was that the previous tool became difficult to maintain. As is often the case, there were a lot of moving parts and the proper function of the lengthy code, databases, external software dependencies, and supporting files for the tool depended heavily on the particular frame of reference of its programmer, who has moved on to another job. HRS already found it was difficult to keep it operational by the next development cycle.

Switching to regular expressions for tokenizing source code

Serendipitously, the work HRS did to unscramble the metadata of text substitutions (presented at IBUC 2010), involved developing a condition parser and evaluator, using regular expressions for tokenizing statements. At one point, HRS realized that it wouldn't be a great leap from that regex to one that could parse all of the Blaise source code.

Figure 1: A regular expression (for .NET) that matches tokens in a Blaise source file (split for readability)

```
(\s+
|\((?>(?!\{|\}).|\{(?<Depth>)|\}(?<-Depth>))*(?(Depth)(?!))\}
|"(?:[^"]+|"")*"(?!")
|'(?:[^']+|'')*'(?!')
|\((?>(?!\[|\]).|\[(?<Depth>)|\](?<-Depth>))*(?(Depth)(?!))\]
|\(\s*[0-9]+\s*,\s*[0-9]+\s*,\s*[0-9]+\s*\)
|\b(?:\((?>(?!\[|\]).|\[(?<Depth>)|\](?<-Depth>))*(?(Depth)(?!))\]|[.\w])+\b?
|:=|<>|>=|<=|[^\w])
```

It did not take long to produce such a regex and it turned out to be surprisingly fast given how slowly previous efforts at tokenizing Blaise would run (a few seconds versus several minutes to an hour). The new regex matches whitespace, comments, string literals, free-standing brackets, date literals, keywords, variable names, operators and other characters. It handles the HRS and a number of other surveys fielded by Michigan.

Figure 2: C# code for the regular expression

```
string regularExpression =
//Match whitespace
@"(\s+"
//Match comments, including nested ones
+ @"|\{(?>(?!\{|\}).|\{(?<Depth>)|\}(?<-Depth>))*(?(Depth)(?!))\}"
//Match string literals with double quotes
+ @"|""(?:[^""]+|"""")*""(?!"")"
//Match string literals with single quotes
+ @"|'(?:[^']+|'')*'(?!')"
//Match free-standing brackets like those after the IN operator
+ @"|\((?>(?!\{|\}).|\{(?<Depth>)|\}(?<-Depth>))*(?(Depth)(?!))\]"
//Match date literals
+ @"|\(\s*[0-9]+\s*,\s*[0-9]+\s*,\s*[0-9]+\s*\)"
//match keywords and variable names
+ @"|\b(?:\((?>(?!\[|\]).|\[(?<Depth>)|\](?<-
Depth>))*(?(Depth)(?!))\]|[.\w])+\b?"
//match operators and other characters
+ @"|:=|<>|>=|<=|[^\w])";
```

So far, I have not come across syntax it does not handle appropriately, but I would be interested in updating or revising it as needed. Figure 1 shows the regex by itself and figure 2 shows it defined as a string in C#; both figures are organized to show different parts of the regex on different lines. Note that the Depth keywords are a feature of .NET's regex implementation, which allow it to handle nesting.

Using the regular expression

Figure 3 shows the C# code for using this regex to generate a List (or any other kind of collection desired) from a Blaise source file. With a bit of extra code for handling INCLUDE statements, this can be made recursive in order to process all source files in an instrument in one run.

Figure 3: Sample C# code for generating a List of tokens from a Blaise source file

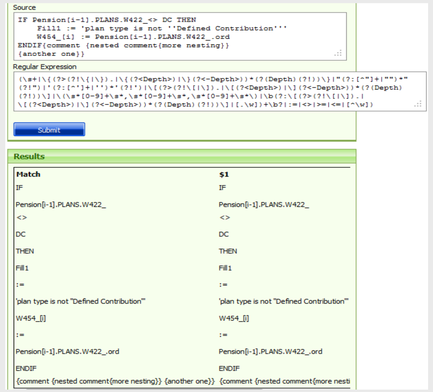
```
string fileContents = File.ReadAllText(filename, Encoding.Default);
MatchCollection tokenCollection = Regex.Matches(fileContents,
    regularExpression, RegexOptions.Singleline);
_tokens = new List<string>();
for (int i = 0; i < tokenCollection.Count; i++)
{
    string token = tokenCollection[i].Groups[1].Value;
    _tokens.Add(token);
}
```

Note that RegexOptions.Singleline is required for the Depth feature to work. Once the tokens are in the List, it is a simple matter to loop through them and change, add or delete them as needed.

Figure 4 illustrates the tokenization process with a breakdown of tokens from a short snippet of Blaise logic. The screenshot is taken from the site regexlib.com, which is one of many websites or applications available that aid in building, refining, or testing regular expressions. The first box shows a snippet of Blaise code, the second shows the regular expression, and the last box shows all the matches made by the regex, each on its own line. This example shows a few of the tricky things that this regex handles: matching fully qualified fieldnames with counters as a single token, correctly matching string literals with escaped quotes, and matching a set of nested comments as a single token.

To aid in the process of updating the token collections, HRS then does a first pass through each collection and generates an additional collection of objects that include the token and additional information about its role in the code. This makes it much simpler to write a loop which runs through and detects the tokens which are relevant to the operation to be performed.

Figure 4: Example tokenization (displayed with http://regexlib.com/RETester.aspx)



Making use of token collections to do valuable things

HRS has used this process to add two new languages to all fields and types in the instrument, where the text for the two new languages in positions 5 and 6 defaults to the text already entered for languages 3 and 4 respectively. This is obviously a one-off operation, but took relatively little effort (just hours to program this feature). There are other operations that HRS performs once every couple of years at least. For example, HRS has also reformatted the code with better spacing and capitalization, and removed comments from some files. Additionally, HRS has used this process to update descriptors for many fields. To do this, a first run through the tokens grabs all the descriptors and the defined fields (by capturing the blocks and include files in which they are located within the source) and writes them to a file. This file is manipulated and updated by non-technical people and then handed back, after which a second routine is run that loops through the tokens and replaces descriptors with the new values as specified in the file. Figure 5 shows a snippet from the end of the loop that accomplishes this latter task. The code that performs these operations was written quickly and is not optimized. Despite this, the code is fairly short and it runs in a matter of seconds, producing a brand new set of source code files that are identical to the originals except for the changes requested. The point of all this is really just that these sorts of operations can now be programmed reliably in a much shorter span of time than was typical of previous efforts.

Figure 5: A snippet from a loop for updating descriptors

```
    if (!descriptorField.Equals(string.Empty)
        && descriptorUpdates.ContainsKey(descriptorField)
        && !string.IsNullOrEmpty(descriptorUpdates[descriptorField]))
    {
        descriptor = descriptorUpdates[descriptorField];
        descriptor = "\"" + descriptor + "\"";
        if (tokenInfo.Component == FieldComponent.Type) //missing descriptor
        {
            descriptor = "/ " + descriptor + " " + token;
        }
        tokens.Add(descriptor);
    }
    else
    {
        tokens.Add(token);
    }
}
return tokens;
```

As well as being fast, this process has proved quite robust. A as a precaution, HRS does a directory compare of the original and modified code sets using WinDiff, which validates that the updates are happening and that nothing is being changed unintentionally.

Related uses of regular expressions in Blaise

The idea to make the regex discussed here actually came out of the HRS text substitution program. That program has a condition parser and evaluator, used to filter out statements that are not on the route when certain conditions obtain. Thus, it made use of a regex that matches tokens in a conditional statement obtained from the API's RulesNavigator. Additionally, it had a regex for tokenizing assignments, which are the content of the text substitution. These are slightly different from the one presented here, both because they handle a single line that contains less variation, and because the code looks slightly different in the RulesNavigator. Thus, depending on the application, it may be necessary or desirable to handle the regex differently, but this topic can be useful for a variety of activities associated with Blaise code.

Future possibilities

HRS has had a longstanding desire to be able to refactor every variable name in the instrument. Because some fields change position within flow during redesigns, the numeric part of HRS fieldnames tend to become unordered, and this effect is compounded by the fact that redesigns may happen every two years. It should be possible to refactor variable names much in the same way we update descriptors, but there are some additional layers of complication. In order to sequence the new names properly, the process needs to work with the Blaise API to determine order of appearance. Additionally, it is necessary to use the API to match the defined field names to the instance names and to correctly match fully-qualified references in the rules. The program HRS wrote does have a feature which produces a concordance between the defined and instance names (as a one-to-many relationship), but this has yet to be put to real-world use and is less straightforward than the other functions described here. Nonetheless, we think it is possible to set this up as an automated process which refactors all the fields in the instrument at once.

For Further Information

Please email jostergr@isr.umich.edu or visit http://www-personal.umich.edu/~jostergr/ for more information.