Final Report **Pavel Sozonov, Yuliya Palamarchuk**
Innopolis University Fourth Year Bachelor Program
Dr. V. Ivanov, Prof. G. Succi Extracting metrics for Ruby using JavaCC
Due Date: 25/04/17

Introduction

The objective of our work was to develop an application that will be able to extract metrics from programs on Ruby language. With this purpose we developed own BNF grammar for Ruby, and transformed it in the format JavaCC. Finally, using the parser that was generated by JavaCC, we developed the metric extractor application with Command Line Interface, that extracts 16 types of metrics (described in Section 1.2), and saves it in the CSV file.

1 Background

1.1 JavaCC

Brereton [3] describes basic principles of JavaCC and how to use it. JavaCC is used for easy defining parser logic for languages. JavaCC receives as input a language's definition and generate Java code, which can parse programs on that language and perform some functions on it. Generated by JavaCC code do not have dependencies from any external files or libraries to be compiled.

Often web-based application use some query system in order to help end users search for information. This purpose need some sort of language for compose queries. Simple queries like most Google queries can be processed by simple instruments like Java's StringTokenizer. But, if this language supports e.g. AND/OR logic, or uses parentheses for defining precedence, then you will need more powerful instrument for processing it. JavaCC is perfectly fit for solving this problem.

Java CC uses .jj file to save language definition, and it consists of:

1. Definition of the parsing context

   The section which begins from PARSER_BEGIN and ends with PARSER_END, contain Java code which is copied into final code of parser as is. In brackets after PARSER_BEGIN and PARSER_BEGIN tags indicates name of parser class, which you want to use. This Java code should contain method main that will be used as entry point to the parser, instantiate object of parser class, initialize it with parsed input string and invoke method parse in order to begin parsing. Also, this section may contain code needed for pre- or post-parsing activities.

2. Definition of "white space"

   In the SKIP section should be defined characters which are used as delimiters and otherwise are ignored.

3. Definition of "tokens"

Token are the minimal unit of parsed string which has meaning to the language. E.g. in expression 'Author = "John Smith"' there is three tokens "author", "=" and "John Smith". Each token consists of token name and regular expression. The process that finds tokens in an input string is called tokenizer.

The TOKEN section of .jj file should contain definitions of all possible tokens of the language. At the beginning should be added token's definitions for reserved words, afterwards, string's and quoted string's definitions, etc. The order is important, because tokenizer uses rules in the same order as they appear in the .jj file.

Example:

```
TOKEN: { <STRING: (["A"-"Z", "0"-"9"])+ > }
```

4. Definition of the syntax of the language itself in terms of the tokens

   Now, in terms of defined tokens, can be determined parsing rules, also called productions.

   Example:

```
void expression() : {}
{ queryTerm() (( <AND> | <OR> ) queryTerm())* }

void queryTerm() : {}
{ ( <AUTHOR> ) ( <EQUALS> | <NOTEQUAL>) ( <STRING> )
| <LPAREN> expression() <RPAREN> }
```

   Where expression and queryTerm are productions, and all words in angle-brackets are tokens.

5. Definition of the behavior that will happen at each stage of parsing

   When previous steps are completed, the .jj file is valid and parser can be compiled with JavaCC. As result, will be generated bunch of .java files, which in turn can be compiled with Java into .class files. Finally, java class file which contains method main can be executed, and perform parsing of input string.

   Example of output for input string "author = John Smith"

```
Call:   parse
  Call:   expression
    Call:   queryTerm
      Consumed token: <"author">
      Consumed token: <"=">
      Consumed token: <<QUOTED_STRING>:
          ""John Smith"">
    Return: queryTerm
  Return: expression
  Consumed token: <<EOF>>
Return: parse
```

In order to make parser useful, each token in the production definitions may be supplemented by additional directives on the Java language. Java code should be placed after the corresponding token and must be enclosed in curly braces.

1.2 Metrics

The article [11] explains what is measurement and metrics. It says that measurement plays a significant role in peoples everyday life, and of course in Software Development sphere. The most easiest way of measurement is comparison, but it is not suitable for everything. Mathematical models help to handle this problem: we have a real world entities and we assign symbols (numbers) to some characteristics, that we want to compare. Than we compare the symbols (numbers) and transfer the deductions referring to real world entities. It is very important to assign the right symbols (numbers), otherwise we can get wrong results.Results of measurement can be wrong if the representational conditions are not satisfied.

As a matter of fact, not every measurement suits for the particular purpose.

Measures can be:

- objective (using devices) and subjective (given by people)
- direct (in the real world) and indirect (in the mathematical world)

The most useful metrics in Software Development are size and complexity.

Size can be measured by lines of code (LOC) and functional points.

Lines of code become known only when software has been already developed, otherwise function points method can subjectively predict the size.

Complexity of program structure can be measured by the cyclomatic complexity (CC). Program code should be represented in a flow graph form. Than we should count the number of connected regions in which the graph divides the surface. The obtained number describes the cyclomatic complexity or CC = number of branches + 1 or CC = number of arcs - number of nodes + 2.

Halstead approach to measure complexity:

1. Count number of distinct operators (n1) and operands (n2)
2. Count total occurrences of distinct operators (N1) and operands (N2)
3. n = n1 + n2; N = N1 + N2
4. Volume of the program: V = N * log2n
5. Difficulty: D = n1/2 * N2/n2
6. Effort: E = V * D
7. Time: T = E/18

Fenton and Pfleeger [12] taxonomy of metrics in software development:

- Product. Internal - structure (ex.LOC). External- performance (ex. time between failure).

- Process. Internal - organization and management (work hours in a week). External - relationships with the customer, stakeholders and users (milestones).
- Resources. Internal - skills of each element (average degree of workers). External - skills as a whole element (number of teams).

Dj Wu et al. [14] provided a list of the *23 commonly-used object-oriented metrics*. We will consider them below.

Chidamber and Kemerer [5] metrics (the CK suite):

1. Weighted Methods per Class (WMC). It is counted as sum of complexity of each method in the class. Complexity of each method calculated as cyclomatic complexity. If method is not written yet, the complexity of each method equals 1.
2. Depth of Inheritance Tree (DIT). It counted as a maximum path in the inheritance hierarchy of a class.
3. Number Of Children (NOC).
4. Coupling Between Objects (CBO). Sum of instances of another classes or methods of another classes, that have been used in class.
5. Response For a Class (RFC). Number of methods plus number of accessible methods of another classes.
6. Lack of Cohesion in Methods (LCOM). It is calculate the cohesiveness of methods in pair-wise for class. NCM - two methods do not have shared attributes of the class. CM - two methods have at list one shared attribute of the class. LCOM(C) = max((NCM – CM),0).

Lionel C Briand et al. [4] metrics:

7. TCC (Tight class cohesion) TCC = (Number of pairs of directly connected public methods using common attributes) / (Number of pairs of public methods)
8. LCC (Loose class cohesion) LCC = (Number of pairs of directly and indirectly connected public methods using common attributes) / (Number of pairs of public methods)

Rachel Harrison et al. [7] metrics:

9. CF (Coupling factor)

$$CF = \frac{\sum_{i=1}^{TC}\sum_{j=1}^{TC} isClient(C_i, C_j)}{TC^2 - TC}$$

where TC is total number of classes and

$$isClient(C_i, C_j) = \begin{cases} 1 & iff c_i \Rightarrow C_j and C_i \neq C_j \\ 0 & otherwise \end{cases}$$

Couplings due to the use of the inheritance are not included in CF. [2]

10. MIF (Method inheritance factor). MIF = (Number of methods inherited in all classes) / (Number of methods defined and inherited in all classes)

11. AIF (Attribute inheritance factor). AIF = (Number of attributes inherited in all classes) / (Number of attributes defined and inherited in all classes)

12. MHF (Method hiding factor). Let V (M) = number of classes where the method M is visible, then

$$MHF = 1 - \frac{\sum V(M)/(totalNumberOfClasses - 1)}{numberOfAttributesOfAllClasses}$$

13. AHF (Attribute hiding factor). Let V (A) = number of classes where the attribute A is visible, then

$$AHF = 1 - 1 - \frac{\sum V(A)/(totalNumberOfClasses - 1)}{numberOfAttributesOfAllClasses}$$

14. PF (Polymorphism factor).

$$PF = \frac{\sum_{i=1}^{TC} M_o(C_i)}{\sum_{i=1}^{TC}[M_n(C_i) \times DC(C_i)]}$$

where TC is the total number of classes and Mn(Ci) = Number of new methods of the class Ci, Mo(Ci) = Number of overriding methods of the class Ci, DC(Ci) = Number of descendants of the class Ci

Brian Henderson-Sellers [8] metrics:

15. NOM (Number of methods). The number of methods defined in a class

16. NOA (Number of attributes) The number of attributes defined in a class

17. DAC (Data abstract coupling). The number of ADT(Abstract Data Type) instances defined in a class

18. MPC (Message passing coupling) The number of send statements defined in a class

19. NMO (Number of methods overridden by a subclass). The number of methods in a subclass overridden from its base class

20. RR (Reuse ratio) RR = (Total number of super classes) / (Total number of classes)

21. SR (Specialization ratio) SR = (Total number of sub-classes) / (Total number of super classes)

Yen-Sung Lee et al. [9] [10] metrics:

22. CC (Class complexity) The sum of complexity for all methods in a class based on the information flow

23. ICH (Information based cohesion) The number of invocations to other member functions/methods

All above metrics cover checking of:

- Size and complexity
- Coupling
- Cohesion
- Inheritance
- Encapsulation
- Polymorphism
- Re-usability

2 Implementation

2.1 Grammar

Original ruby complier uses YACC to creating language parser. YACC is a program with same functionality as JavaCC, it receives a language grammar as input and generates parser on the C programming language. YACC is predecessor of JavaCC. Thus, if download a sources of ruby, we can find a file "parse.y" (configuration file for YACC), which contains all grammar of ruby. But file parse.y has quite difficult syntax, so in a subdirectory "sample" there is a file "exyacc.rb", which can be used to extract a grammar from the "parse.y" in a more readable form. In our work we use last stable version of ruby 2.4.0, and below are shown commands, that were used to extract the grammar.

```
git clone https://github.com/ruby/ruby; cd ruby; git checkout v2_4_0
ruby sample/exyacc.rb < parse.y > ruby_2.4.0_grammar.txt
```

File "ruby_2.4.0_grammar.txt" along with other results of our work can be found in the github repository [13].

At this stage, we had the grammar but in that form it did not fit to JavaCC, so we rewrote it in the appropriate format. We created new JavaCC configuration file "RubyParser.jj", save the original grammar as a comments, and added new code in JavaCC notation. In terms of JavaCC, were defined 129 tokens, and 222 productions. Afterward were added other important parts of the configuration file, such as the "white spaces" block, and the block with the initial source code. Size of the configuration file was 3.5 KLOC at that moment. The configuration file was compiled with JavaCC, were fixed some minor bugs, and were identified the one major problem, the grammar contains 29 left recursions, which cannot be resolved by JavaCC automatically. There are methods to eliminate left recursion, which we successfully applied and take as a result generated by JavaCC ruby parser.

However, during the second stage we modified our grammar in order to achieve the best possible compatibility with Ruby language. The reason was that our research shows

that structure of this language is very flexible and it is almost impossible to achieve 100% support it using such tool as JavaCC [6]. Official web site of Ruby has rich and detailed documentation, but even it do not provide any formal grammar of this language [1]. Eventually we prepared grammar which well supports Ruby syntax, with one small limitation. In Ruby all methods can be invoked by two way with braces (e.g. puts("Hello, world!")) or without (e.g. puts "Hello, world!"). This feature of language is very difficult treatable by JavaCC and can cause ambiguity. So we prepare grammar with assumption that all programs which will be tested always use first method of method invocation - with braces.

2.2 Structure of application

Our project has three base classes:

1. Class Main. It is entry point of our application, comprise function main. It instantiate Command Line Interface and run it. Function main is invoked without arguments.
2. Class CLI. Command Line Interface 2.3
3. Class RubyParser15. This class contains Ruby parser generated with JavaCC. It is used by class CLI when operation test file is performed.
4. Class MetricsCounter. Class used by classes CLI and Parser as a storage for metrics.

Source code of the project is presented in our GitHub repository [13] (branch Master)

2.3 Command Line Interface

Command Line Interface comprise several commands:

1. h - show help
2. f - show names of all sample Ruby files from "tests" subfolder
3. t - show names of all files and suggest chose file which will test (name of file will be read from keyboard). If file name is correct, it extracts the metrics from this file and append this metrics to end of file Report.csv in "tests" subfolder.
4. a - tests all files with extantion *.rb and writes results in Report.csv file.
5. e - exit

2.4 Structure of report file

Report file has name "Report.csv" and is located in the project's subfolder "tests". It has format csv, so all metrics are separated by "\t" symbol. Data with new metrics always appends to the end of existing file, if file do not exits it creates automatically.

Columns description: The first column has date and time when list of metrics was appended to report file.

The following columns shows implemented metric results.

1. Number of methods
2. Number of statements
3. Number of classes
4. Number of attributes
5. Number of loops
6. Number of conditions
7. Reuse ratio
8. Number of children
9. Specialization Ratio
10. Method Inheritance Factor
11. Attributes Inheritance Factor
12. Number of Overridden methods
13. Lack of Cohesion Methods
14. Tight Class Cohesion
15. Depth of Inheritance
16. Number of Interfaces

2.5 Tests of metrics

In parallel with the work on the grammar we prepared the test for 16 metrics. These test can be found in the file TestMetrics.java in the github repository [13].

Results

In results of our work we developed Command Line Application for extracting metrics from program on Ruby. Our application can calculate 16 types of metrics, and was successfully tested on the 8 sample files with different Ruby programs. The most difficult part of the work was creating of the BNF grammar for Ruby, because of this language has very flexible syntax, and there was no any accessible grammar for it in the Internet. When the grammar was finished and translated to the JavaCC format we generated the parser and added to it functionality for data extracting and data output, as well as Command Line Interface for convenient using.

References

[1] Ruby language official site. `https://www.ruby-lang.org/`.

[2] KK Aggarwal, Yogesh Singh, Arvinder Kaur, and Ruchika Malhotra. Empirical study of object-oriented metrics. *Journal of Object Technology*, 5(8):149–173, 2006.

[3] JoAnn Brereton. Use javacc to build a user friendly boolean query language. `http://www.ibm.com/developerworks/data/library/techarticle/dm-0401brereton/index.html`, 2004.

[4] Lionel C Briand, John W Daly, and Jurgen Wust. A unified framework for cohesion measurement in object-oriented systems. In *Software Metrics Symposium, 1997. Proceedings., Fourth International*, pages 43–53. IEEE, 1997.

[5] Shyam R Chidamber and Chris F Kemerer. A metrics suite for object oriented design. *IEEE Transactions on software engineering*, 20(6):476–493, 1994.

[6] Michael Furr, Jong-hoon David An, Jeffrey S Foster, and Michael Hicks. The ruby intermediate language. *ACM Sigplan Notices*, 44(12):89–98, 2009.

[7] Rachel Harrison, Steve J Counsell, and Reuben V Nithi. An evaluation of the mood set of object-oriented software metrics. *IEEE Transactions on Software Engineering*, 24(6):491–496, 1998.

[8] Brian Henderson-Sellers. *Object-oriented metrics: measures of complexity*. Prentice-Hall, Inc., 1995.

[9] Yen-Sung Lee, Bin-Shiang Liang, and Feng-Jian Wang. Some complexity metrics for object-oriented programs based on information flow: A study of c++ programs. *J. Inf. Sci. Eng.*, 10(1):21–50, 1994.

[10] YS Lee, BS Liang, SF Wu, and FJ Wang. Measuring the coupling and cohesion of an object-oriented program based on information flow. In *Proc. International Conference on Software Quality, Maribor, Slovenia*, pages 81–90, 1995.

[11] Laurie Williams Michele Marchesi, Giancarlo Succi. Traditional and agile software engineering, 2003.

[12] EF Norman, SL Pfleeger, et al. Software metrics: a rigorous and practical approach. *Tsinghua University Press & PWS Publishing Company*, 1997.

[13] Yuliya Palamarchuk Pavel Sozonov. Github repository of our project. `https://github.com/PavelSozonov/JavaCC/`.

[14] Di Wu, Lin Chen, Yuming Zhou, and Baowen Xu. A metrics-based comparative study on object-oriented programming languages. In *SEKE*, pages 272–277, 2015.