(IJID) International Journal on Informatics for Development
Vol. 7, No. 1, 2018, Pp. 13-19

A Scanner and Parser for Z Specifications

Maria Ulfah Siregar
Informatics Department
Faculty of Science and Technology, UIN Sunan Kalijaga
Yogyakarta, Indonesia
maria.siregar@uin-suka.ac.id

John Derrick
Department of Computer Science
The University of Sheffield
United Kingdom

***Abstract*—Coding either a scanner or a parser from beginning has many disadvantages such as tedious, could raise many errors, needs much times and effort, etc. All of these could result less scanner or parser. This paper describes our research on implementing a scanner and parsers for Z specifications. Rather to code them from scratch, we use tools that have specialities on creating such tasks. These tools generate several Java files which can be integrated with a main program in Java. Our research produces a scanner and parser for Z specifications. These tools may benefit Z specifications to be studied further.**

Keywords-Z2SAL; Scanner; Parser; JFlex; BYACC/J

I. INTRODUCTION

Z2SAL is a translator for Z specification documents into SAL specification documents. It has been known also as a scanner and parser for Z specifications, specifically a hand-written scanner and a hard-coded parser. It is since Z2SAL researchers wrote their scanner and parser by using Java language; it is a language that is not specialized for writing scanners and parsers.

Thus, it is one reason for us not to reuse the Z2SAL scanner and parser to implement our Z scanner and parser. Other reason is that it will take time and be an effort to hand-write such a scanner and hard-code such a parser, such as to define regular expressions, and Z operators' precedencies and associativity, to match a sequence of tokens to one of the Z rules, and others.

Another reason is that a JFlex lexer has been known to be faster than a hand-written scanner/ lexer. Although a BYACC/J parser is not as fast as a hard-coded parser, the BYACC/J parser is easy to write and modify.

Moreover, to learn code of somebody else is more difficult rather than to write code from scratch. More importantly, Z2SAL scanner and parser were integrated into the design of other parts of Z2SAL.

II. LITERATUR REVIEW

A. A Z Scanner

A Z scanner will scan Z tags in Z specifications. A successful scanning will pass tokens, which are obtained from accepted Z tags, to a parser for further process. Otherwise, a lexical error on an involved line will be reported.

One of scanner generator that can be used to produce a scanner for Z specification documents is JFlex. JFlex is a Java lexical analysis generator (scanner generator) [1]. JFlex 1.6.1 is the current stable version which was released on 16 March 2015. It is free software which is published under a BSD-style license.

JFlex will generate a **.java** file from a JFlex specification which has an extension **.flex**. Thus, this generator has an input which is the JFlex specification.

The Java file has one class that consists of code for the scanner. This code is a lexer that reads input, matches the input against the regular expression, and runs an associated action. A lexer is a part of a compiler, specifically the first front-end of it.

The lexer will match keywords, comments, operators, etc. Then it will generate a stream of input tokens for a parser. However, it can also be used for other intentions.

Built on a deterministic finite automaton (DFA), a JFlex lexer is fast since backtracking is not performed. Several parser generators can be integrated with this lexer. For example: the LALR parser generator Construction of Useful Parsers (CUP) by Scott Hudson, the Java modification of Berkeley Yet Another Compiler Compiler (YACC), BYACC/J, by Bob Jamison [2].

To interface a generated scanner with BYACC/J, the command **%byacc** is used. It is will be discussed later on.

B. A Z Parser

This section describes the Z parser. The parser will read tokens passed by the scanner, and try to process whether these tokens match any rule in Z grammar specified in the Z parser.

This section contains several sub-sections, which begins with an introduction to the BYACC/J parser generator.

An extension to the Berkeley 1.8 YACC-compatible parser generator can be used to implement a parser for Z specification documents [2]. It is BYACC/J, which is available in Microsoft, Linux, Macintosh, and SUN Solaris platforms [3], with version 1.15.

By a flag "**-J**", the standard YACC tool will generate one or more Java parser files from a YACC source file **.y**. However, BYACC/J can also generate C/C++ parsers [3]. These Java files can be compiled to produce a LALR-grammar parser.

One of these files, which is usually generated, is the **Parser.java**. By reading this file, a user can see how a parsing algorithm of YACC works. This Java file generates a class which is an extension of **Thread**.

Another Java file is the **ParserVal.java**. The current version of BYACC/J allows a user to define an **int**, a **double**, a **String**, or an **Object** values.

C. Z2SAL

The idea of translating Z into the SAL input language was due to Smith and Wildman [4] at the University of Queensland, Australia. However, since the basic idea given in [4], the idea was implemented in a tool set, and the current Z2SAL is extended in a different direction. In doing so, it has also had to tackle optimization issues [5], and thus is quite different from the ideas as originally envisaged.

Z2SAL translates a Z specification into a SAL module. This module will group a number of definitions including types, constant and modules for describing a Z states transition system [6].

Currently, the tool has two operating modes which it will either translate a single Z specification into the input format of SAL for model checking purposes, or translate a pair of Z specifications for refinement checking purposes [7]. The translated output is placed in the same directory as the source.

Regarding model checking, it is possible to add theorems at the end of this automaton, to check whether certain properties always hold, or eventually hold. However, Z2SAL is able also to translate properties which are added on the Z specification.

III. METHOD

This section describes briefly our method on this research. After study literature, we designed a system which has functions such as a scanner and a parser. Next is to implement such a system. This implementation is described here.

A. The Implementation of Our Scanner

After two above sub-sections about a brief introduction to JFlex, a scanner generator which was used to implement our Z scanner, and a brief description on lexical specifications, this sub-section discusses our Z scanner.

Thus, our Z scanner was implemented using JFlex. Our Z scanner was implemented so it can scan several Z tags. In other words, our Z scanner does not support all Z tags. For a complete list of Z tags which can be scanned by our Z scanner, please see [8].

As mentioned earlier, our scanner does not scan all Z tags as well as not all of our Z tags were accompanied by actions. Reasons behind the first statement are to be in line with Z2SAL as the translator does not support all Z tags.

Thus, there is no point here to be able to scan a token represents a Z tag which is not supported by Z2SAL and sometimes it is not available also on [9].

Other reason for us not to include all Z tags is that it is not difficult to add a new token. Another one is our Z specifications could be scanned by our Z scanner, though this scanner does not support all Z tags.

In other words, this scanner has implemented a list of Z tags which are suitable to our Z specifications.

Several Z tags which were not specified in our Z scanner are:

- \nexi, \nexists for representing "∃";
- \bool for "$\mathbb{B}$";
- \iter for "*iter*";
- \pred for "*pred*";
- \post for "*post*";
- \items for "*items*";
- \bagcount for "*count*";
- \buni for "⊎";
- \varsdef for "≙";
- R^+ for transitive closure;
- R* for reflexive-transitive closure.

Let us move to three parts of our scanner. There was no user code which was put in the first part of our JFlex specification. The name of our JFlex specification is **Lexer.flex**.

For the second part, the **%byacc** directive was added. Another directive was added in this part, a directive to indicate a name of the generated Java file. In this scanner, it was defined as **ScannerCl**.

At first, **Scanner** was chosen, but then it turned out that the latter is one of Java class names. There are two methods specified in the second part.

The first method is a constructor for the generated Java class. The second one is a method to get the line number of a particular line of our Z specification. This method is called by actions of "**.**" of our regular expression to indicate a lexical error.

There were also declarations of two variables in this part. Both methods and these two variables were enclosed by "**%{**" and "**%}**".

Z tags were specified in the third part. Several Z tags that have actions in them, these actions are quite similar in all these tags. The first action is to assign a matched tag which is returned

```
yyparser.yylval = new
ParserVal(yytext());
```

by **yytext()** as a semantic value for the associated parser, shown as follows:

The JFlex must store this value in **yylval** before it is returned. The routine **yyparser()** is the parser generated by YACC.

The second action is to return a token of the matched tag to the parser.

```
return Parser.BZED;
```

The above is an example of the action to return the **BZED** token to the parser. This token indicates **\begin{zed}** tag. Among these tags, not all of them were implemented with the first action.

JFlex matches input texts to patterns constructed by regular expressions based on a set of simple disambiguating rules as follows [10]:

- JFlex patterns only match a given input character or string once.
- JFlex executes the action of the longest possible matched input texts.

Thus, if our scanner returns a lexical error while scanning a particular Z specification, this error can inform us several cases after a further check on this Z specification.

The first case, the error means that the associated Z tag has not been specified in our scanner. Having this error, a solution is to add this tag to our scanner.

As an example is shown by the below output:

```
run:
file          parse:          E:\Google
Drive\Tesis\program\JavaCode\Thesis\src
\carspark.tex
Lexical error on line: 1 : \
C:\Users\MUS\AppData\Local\NetBeans\Cac
he\8.2\executor-snippets\run.xml:53:
Java returned: 1
BUILD FAILED (total time: 3 minutes 5
seconds)
```

It was generated by our system during running the modified Cars Park specification (see [8]).

The first line of this specification has been changed to:

```
\documentstyle[11pt,oz]{article}
```

Our scanner only recognizes **12pt** as the font size.

The second case, the error means the tag, which is available in our scanner, has been written wrongly. Thus, the associated tag will be rewritten precisely.

Using the same example as above, below is the output generated by our scanner:

```
run:
file          parse:          E:\Google
Drive\Tesis\program\JavaCode\Thesis\src
\carspark.tex
Lexical error on line: 4 : \
C:\Users\MUS\AppData\Local\NetBeans\Cac
he\8.2\executor-snippets\run.xml:53:
Java returned: 1
BUILD FAILED (total time: 1 minutes 2
seconds)
```

It is because the example has also been modified. Its fourth line is misspelt into:

This lexical error should be fixed since the error will push the system to stop immediately.

In order to proceed to the Z parsing, it indicates no lexical error which means all Z tags on associated Z specification are recognized as true Z tags and specified in our Z scanner.

By using the JFlex scanner generator, there were 1,746 states during a Non-Deterministic Finite Automaton (NFA) construction of our scanner. This large number of states was reduced to 778 states in a DFA construction before minimization and it was reduced again to 566 states in minimized DFA.

There was neither error nor warning detected by the JFlex scanner generator.

This is shown by Fig. 1.

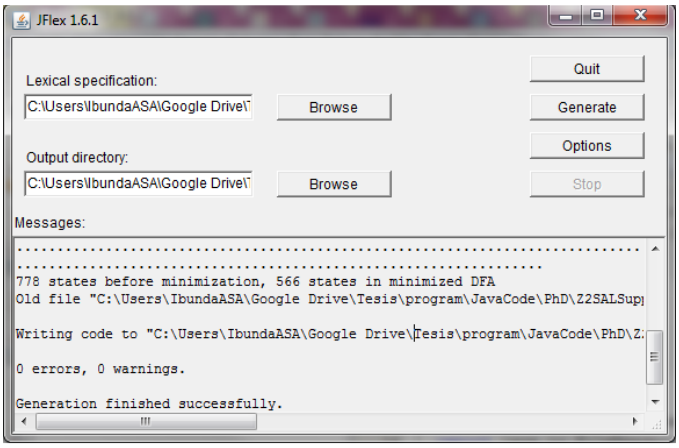

Figure 1 The JFlex scanner generator

In a case the scanner generation is successful; the generated Java file will be generated.

This Java file is located in the same place as the JFlex specification. This generation will generate the **ScannerCl.java** file from our scanner.

B. The Implementation of Our Parser

Our Z parser can be seen in [8]. It was represented by a BYACC/J specification, which was named as **Parser.y**.

Our parser does not implement all Z rules. The Z grammar in our BYACC/J specification refers to [9].

Several Z rules that were not specified by our BYACC/J specification are given in Table 1. Z rules, which were listed in Table 1, have not been implemented because of several reasons.

The first reason is our examples do not contain any declaration or predicate which match one of those rules. Another reason is several of those rules caused the number of shift/ reduce or reduce/ reduce conflicts is even higher. Since then they were not included in our Z grammar.

Table 1 A list of unspecified Z rules

LHS	RHS
schema.exp1	pre **schema.exp1**

pred	let **Let-Def.list • pred**
Let-Def.list	**Let-Def Let-Def.list**
Let-Def	**ident == expr** (**op.name**) == **expr**
op.name	**_in-sym decor_ pre-sym decor_ _post-sym decor _(_)decor _decor**
pred1	**PREREL decor expr** pre **schema.ref**
expr0	**μ spot.tail μ word.schema.text** let **Let-Def.list • expr**
expr	if **pred** then **expr** else **expr**
expr4	**expr4**$^{\textbf{expr}}$
in-sym	INFUN | INGEN | INREL
pre-sym	PREGEN | PREREL
post-sym	POSTFUN

If a Z specification, which has a declaration or predicate statement does not match any of our Z rules, is given to our parser, then our Z parser will generate a syntax error. This error can fall into several sources.

The first source is our grammar does not have a rule of such a declaration or predicate statement. Solving this problem is by adding this new rule. This might be necessary to check also relevant tokens to specify the rule since it can be such a token has not been specified in our scanner.

The second source is indeed the rule of the declaration or predicate statement has been specified in our Z grammar. However, there were conflicts on either declaration or predicate. For this case, a solution requires a further check on available grammars and solve any shift/ reduce or reduce/ reduce conflict if it exists.

For example, is given by the output shown in the right column. It was generated from the same example used in [8].

This time, the statement in line 5 has been modified incorrectly into:

```
count \nat \\
```

Our scanner counts the number of line from 0. Thus, the real number of line should be added with 1.

```
run:
file          parse:          E:\Google
Drive\Tesis\program\JavaCode\Thesis\sr
c\carspark.tex
syntax error
Please check line: 4
\nat with length: 4
BUILD   SUCCESSFUL   (total   time:   34
seconds)
```

The suspicious line is a declaration part in a schema. Our parser expected that there is "**:**" between name and type of a variable.

In the first part of our BYACC/J specification, several imported packages were declared. The first half of tokens were declared having string values, whereas the second ones have not had any values. Tokens were specified using capital letters. All types of terminal symbols in our parser have also string values.

Precedencies and associativity of several Z operators, which were formulated in our parser in this first part, can be seen in Table 2.

These precedencies and associativity follow ones specified in [9], but not the last two lines. Both these lines were specified by us.

Table 2 Precedencies and associativity of Z operators

The second part of our parser contains almost Z rules which were obtained from [9]. However, several of them have been rewritten to avoid shift/ reduce and reduce/ reduce conflicts. Although these conflicts exist on our parser, the numbers are less than the numbers of the same conflicts on original Z grammar.

Not all our rules were accompanied by actions. These actions store information which will be used on further process.

One example of our Z rules is discussed here. It is a Z rule to parse a schema calculus definition. Our parser supports also many lines in one schema calculus definition box. It is since our parser passes information about a separator on each different schema calculus definition.

A separator which separates each line containing different schema calculus will be put on the **llSchCal** list. This list will be used later in the schema calculus operation.

The associated Z rules to process the separator is shown as follows:

```
schema.def.horz: WORD SDEF
                {
                      schCal = true;
                      if (separator){
             llSchCal.add("separator");
                 separator = false;
                      }
                }
                  schema.exp
                | WORD gen.formals SDEF
                {
                  // has the same code
            as for WORD SDEF
                }
                 schema.exp
                      ;
```

schema.exp non-terminal can be matched by two rules. One of them is **word.schema.exp1** and it will match with either **schema.exp1** or **WORD**. The latter is a terminal, in this parser, it is a Z token which a firing on it will store the token information on **llSchCal** list.

On the other hand, **schema.exp1** non-terminal will store information to the list shown in the right column. "..." can be seen in [8].

The third part of our parser consists of declarations of several variables which were used on our actions. There is also a reference to our scanner.

There are several functions specified in this part. The first function is to establish an interface to our scanner. The second one is to report any syntax error that has been found.

```
word.schema.exp1: schema.exp1
                | WORD
                    {
                     llSchCal.add($1);
                    }
                ;
```

```
schema.exp1: LSBRACK
             {
                llSchCal.add($1);
             }
             word.schema.text RSBRACK
             {
                llSchCal.add($4);
             }
    |   schema.ref
        ...
    |   NOT word.schema.exp1
        ...
    |   word.schema.exp1 AND
word.schema.exp1
        ...
    |   word.schema.exp1 OR
word.schema.exp1
        ...
    |   word.schema.exp1 IMPLIES
word.schema.exp1
        ...
    |   word.schema.exp1 BIMPLIES
word.schema.exp1
        ...
    |   word.schema.exp1 PROJECT
word.schema.exp1
    |   word.schema.exp1 HIDE '('WORD
"'"')'
        ...
    |   word.schema.exp1 HIDE
'('word.decl.name.list')'
        ...
    |   word.schema.exp1 SEMI
word.schema.exp1
        ...
    |   word.schema.exp1 PIPE
word.schema.exp1
    |   '(' schema.exp ')'
        ...
    ;
```

This function will call another function to perform this job. Another function is a constructor of the generated Java file later.

Before a parser generation is performed, a BYACC/J specification must be copied to the place at which JFlex generator is located. The command to generate our parser is as follows:

```
C:\jflex-1.6.1\bin>yacc -J Parser.y
```

Our Z parser generated two Java files at the end of this generation. In our case, they are **Parser.java** and

ParserVal.java. Then, both these files are copied again to the place at which the BYACC/J specification is defined.

In addition to both Java files, inevitably, our Z parser generated also several warnings. These warnings relate to conflicts with our parsed Z grammar.

The warnings are:

```
yacc: 1 shift/reduce conflict, 5
reduce/reduce conflicts.
```

These warnings have not been solved. It requires time and an effort to an elaborate check on the grammar and a rewriting in it.

However, all of our examples could be parsed by our parser. Based on information gathered from actions defined in our parser, the way our parser was designed is sufficient and it could be said that our parser parses the input correctly. Thus, these warnings are left as future works.

Fortunately, YACC provides also an output file during the parser generation. To obtain the output file, the above generation command is modified as follows:

```
C:\jflex-1.6.1\bin>yacc -v -J Parser.y
```

A file named **y.output** as default is generated after the above command is executed. This file contains the parse table of the parser. The parsed table could be checked if there is conflict with the grammar. Our parsed table contains 382 states, 86 terminals, 95 non-terminals, and 220 grammar rules.

The above conflicts are informed also as follows:

- State 69 contains 1 shift/ reduce conflict.
- State 137 contains 1 reduce/ reduce conflict.
- State 151 contains 1 reduce/ reduce conflict.
- State 159 contains 1 reduce/ reduce conflict.
- State 199 contains 1 reduce/ reduce conflict.
- State 202 contains 1 reduce/ reduce conflict.

However, it is possible that there are errors in gathered information if further type-checker or processing is added to our parser. Furthermore, it might these conflicts make our system fails to run other Z specifications. This case is beyond our expectation now.

IV. CONCLUSION AND FUTURE WORK

Our research has been able to produce a scanner and parser for Z specifications. We have also integrated them with our main program to redefine and expand Z specifications. Although our scanner does not recognize all of Z tags, all of the implemented Z tags are suitable for our research. Nevertheless, our parser, warning that it has, could be ignored for the running of our program.

ACKNOWLEDGMENT

The first author would like to thank John Derrick, Siobhan North, and Anthony J.H. Simons for giving the author a chance to work with their Z2SAL, discussions in this tool, and supervision in the first author doctoral study. A lot of thanks are dedicated to MORA The Republic of Indonesia for its financial support during this study.

REFERENCES

[1] G. Klein, "JFlex – The Fast Scanner for Java," Accessed from http://www.jflex.de/index.html, 2015.

[2] T. Hurka, "BYACC/J," Accessed from http://byaccj.sourceforge.net, 2008.

[3] A. J. D. Reis, "Compiler Construction Using Jva, JavaCC, and YACC," Wiley-IEEE Press, 2012.

[4] G. Smith and L. Wildman, "Model Checking Z Specifications Using SAL," in ZB 2005: Formal Specifications and Devvelopment in Z and B, Springer, 2005, pp. 85–103.

[5] J. Derrick, S. North, and A. J. H. Simons, "Z2SAL: A Translation-based Model Checker for Z," Formal Aspect of Computing, Springer, 2011, pp. 43--71.

[6] J. Derrick, S. North, and A. J. H. Simons, "Issues in Implementing a Model for Z," Formal Methods and Software Engineering, Springer, 2006, pp. 678--696.

[7] A. J. H. Simons, "The Z2SAL User Guide," Accessed from http://staffwww.dcs.shef.ac.uk/people/A.Simons/z2sal/userguide.html, 2012.

[8] M. U. Siregar, "Support for Model Checking Z Specifications," A PhD Thesis of the University of Sheffield, Accessed from etheses.whiterose.ac.uk/17776/1/thesis_acp12mus_rev.pdf, 2017.

[9] J. M. Spivey, "The Z Notation,"Prentice-Hall: New York, 1989.

J. R. Levine, T. Mason, and D. Brown, "Lex & YACC," O'Reilly & Associates, Inc., 1992.