International Journal of Engineering Research and Development
e-ISSN: 2278-067X, **p**-ISSN: 2278-800X, www.ijerd.com
Volume 7, Issue 6 (June 2013), PP. 25-41

Minimal Java

[1]K.Sundaramoorthy, [2]Dr.S.Srinivasa Rao Madane
[1]Research Scholar, St.Peter's University,Chennai ,Tamilnadu, India,
[2]Principal , Adhiparasakthi college of Engineering, Kalavai Tamilnadu, India

Abstract:- "Minimal Java", focuses on reducing the verbosity present in java programming language's syntax specification by designing a new syntax specification (called as 'MiVa' henceforth) and a compiler designed specifically for recognizing the MiVa syntax specification.

Keywords:- MiVa-Minimal java, verbosity-wordiness.

I. INTRODUCTION

The java programming language is a general-purpose concurrent class-based object-oriented programming language. Java is also been designed to have few dependency implementation as possible. In addition to that it has a syntax specification which likely reflects the syntax specification of languages such as C, C++ and also highlights its verboseness.

Basically, Java programming language involves much of syntax statements irrespective of the kind of task the code is being built for. By designing a new specification of language with syntax specification having less verbosity and creating a fresh compiler which will recognize the new language and produce the targeted source code as an output of compilation. Because of the size of the project it is being broken up into two phases. As a first step of a potentially huge project, this phase supports minimal keyword/ syntax or even, no keyword/ syntax - alternative to each and every keyword/ syntax that exists in the current Java Language Specification. The minimal keyword/ syntax will be converted in to the equivalent/ original Java source code internally, which will then be treated with the existing Java compilers to produce the byte code that can run on any JVM. When speaking about the compiler it is a high-level to high-level language compiler that can transform a new minimal source code into a Java source code. The acronym of Minimal Java is 'MiVa'.
The MiVa compiler will be programmed using Java programming Language.
Advantages:

- The time required to codify a task can be trimmed down.
- Syntax level errors can be minimized and can be easily debugged.
- The compilation time can be improved since the syntax specification is being improved.
- It a platform independent because of using java as source code.
- Easy understanding of the code
- Reducing the file size of the source files

II. COMPILER SPECIFICATION

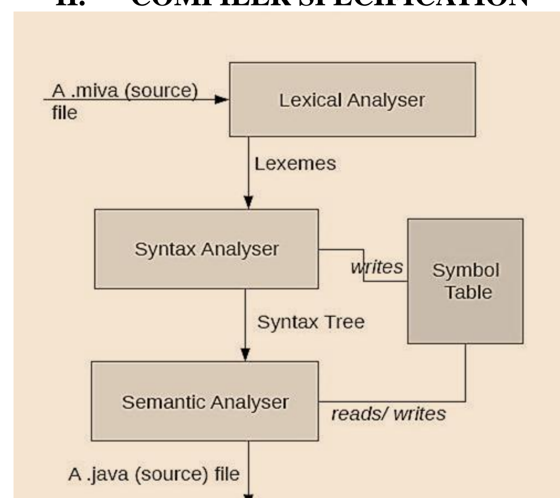

Fig. 1.MiVa Compiler architecture

1) **Lexical Analyzer:** The .miva file is read and the input streams of characters are processed to separate the lexemes out of it. Lexemes are the atomic elements of the language, they are Identifiers, Names, Variables, Constants, keywords, statement terminators, operators, special characters that convey special meaning to the language (like comments) etc. Lexemes are separated by white spaces (new line, space, tab), special characters etc.

2) **Syntax Analyzer:**The lexemes are processed and the category of it is identified by referring to a dictionary that already has the keywords, operators and other special symbols. Based on *the grammar** defined for the language the lexeme's position relative to other lexemes are analyzed and if any discrepancies are identified they are reported back to the user asking for correction. Syntax analysis will continue on the entire file irrespective of the presence of thediscrepancies.

A symbol table is being generated based oneachand every lexeme to help identify the meaning the meaning of each of them in the syntax and to assist constructing the target language with references to the symbols in the table. Having references to the symbol tablemakes it possible to have all the lexeme has acentral location and to avoid duplications oflexemes scattered around.

A syntax tree is being constructed as an intermediate representation so that any target representation can be derived out of this representation. In our case, the target language is the Java high level language.

* A set of rule that is being defined to validatethe correctness of the syntax.

3) **Semantic Analyzer**The syntaxtreerepresentation is being parsed and the semantics being the syntax are identified to write down the corresponding target languagestatements/ codes. The symbol table, syntax tree are destroyed after the semantic analyzer iscompleted. As of the completion of this phase a .java is being generated which will be compatible with any already built Java Compiler.

III. LANGUAGE SPECIFICATION

In MiVa language specification the verbosity is reduced by creating a minimal syntax or sometimes no keywords. Alternative to each and every keyword/ syntax that exists in the current Java Language Specification. For e.g. import is been replaced with '#'symbol.

#java.util.*

Here no delimiters like ';' are used, for that we consider the white space as the delimiters

1) Syntax Specification- MiVa's syntax is the crux of the concept we are chasing after, that is, to minimize the verbosity of the original Java language. The following information is the syntax specification of MiVa Language. The list is containing the minimal syntax for the fundamental Java language elements like for, while loop

/**How to read this document: The grammar followed to describe the Semantics, Syntactic structure w/ example and the grammar of the syntax with respect to other syntax in MiVa is as follows,

[~ Bold 14pt First line is the **Semantics** of the syntax that follows

Regular 12pt group of statements are the **Syntactic Structure** with example

Bold Italics 12pt paragraph/s is/are the **grammar** that particular will follow]+ ****/**

~ Package declaration

~org.example.me

~ org.example.me

Package name should appear before any non-comment element in the source file. Package name ends with a new line or any white space character. There can be any white space character between ~ and the package name.

~ Import statement

#java.util.*

#java.one #java.two

Appears just next to package declaration and before class definition. Comment/s can appear between them. Delimited by a new line or any white space character.

~ Class signature

eg.

ClassName (package scope class)

+ClassName (public class)

#ClassName (protected class)

- ClassName(private class)

eg. for a class subclass and implementing an interface

+ClassName<SuperClass Interface1, Interface2

optional:

+**c**ClassName

Class name can appear just by the user given name alone that is without any keyword. Optionally, the character c or C can be given to indicate that this is class definition. This is useful when interface or enumeration is being defined instead of a class.
The symbol + indicates that this is a public class, and – indicates as private class, whereas # says it is protected.
Class name, the superclass, the interface/s and its flags/ modifiers such as -, +, #, c should be separated by a new line or any white space character.

~ Interface signature

eg.
+**i**InterfaceName
iInterfaceName
#**i**InterfaceName
InterfaceName
iInterfaceName<SuperInterface

The flag/ modifier i/I is mandatory to indicate that the following definition is for an interface. This enables us to verify the defined syntax/ semantics with the grammar of that of an interface.
-, +, # and other grammars are same as that specified in the Class name

~ Enum signature

EEnumName
eEnumName
+**e**EnumName
+**e**EnumName interface1, Interface2

E/e flag is mandatory to indicate that the following defintion is for aenum. Other flags and grammar is same as defined for Class name/ Interface name.

~Method signature

eg. for a simple method that returns nothing (void)
+methodName{
stmt 1; stmt 2
stemt 3
}
eg. for a method signature with 1 argument
+methodNameint a {
....
}
eg. for a method signature with return type (int)
-intmethodName**}**
....
}
eg. for method signature with exceptions that are thrown by it
#intmethodNameint a, byte b
CheckedException }
....
}

Method signature is similar to Class name etc. But since this is a block structure, the | character is used as the start and end of the block.
Flags/ modifiers, method name, arguments and exception list should be separated by new line or white space characters.

-, +, # plays the same role here as with the case of Class name, Interface name and Enum name.

~ Class variable

+**s** type variableName
+**s** type variableName = 10

The flag s indicates the class level scope of the declared variable. Variables can also be initialized. Terminated by a new line or a white space character.

~ Instance variable

-typevariableName
#typevariableName = null

Same as the class variable except the flag s is not included.

~ Printing to the default console

eg. to print Hello, World

“Hello, World”
to print i + 3 where i = 10
i + 3
eg. to print an incrementing variable
printi++
The compiler takes any statement that are hanging around like the ones shown in the example will be taken as print statements it can occur anywhere where a statement is valid (that is inside a method, constructor, static block, anonymous block).

But it also aware of statements that seems like hanging around but with meaning behind it. Take the example of i++, the increment operator, in the example, it will not be printed as the compile knows that it is incrementing and the programmer is not intending to print it. So, in this case, if the programmer wants to print, a print keyword should be used as shown in the example. Another example for explicit print keyword is the method call that doesn't assign its return type to variables (LHS).
~ For loop
i=0, j=0; ;i++ {
....
}
The three components, initializer, expression, incrementer, of the loops are given without any keyword along with a block structure delimiters. All the three components are optional.
~ While loop
I < 3 {
.....
}
One expression is followed by a block structure.

~ Do-while loop
{
...
} j<5 || k<0
A block structure followed by an expression.
~ If-else
If expression {
....
}**else** {
....
}
The keyword if followed by the expression and a block structure. If structure can optionally be followed by a else keyword and a block struture.

For the basic java output system

System.out.print(”abcd..”);

We need to code alone the statement which we need to print as the output.

MiVa e.g. “abcd..”
It will automatically print on the screen
MiVa will consider this statement as a output statement and automatically it will print that statement in the screen.
For that we have to code that statement using double quotes
For println use ‘/n’ before the statement
E.g. “\nabcd..”
e.g.2Forloop in MiVa
“i=0, j=0; ;i++{
}
....“

Consider the sample program coded using java:

```
import java.io.*;
classPrimeNumber {
	public static void main(String[] args) throws Exception{
		inti, n=50;
		System.out.println("Prime number: ");
		for (i=1; i<num; i++ ){
			int j;
			for (j=2; j<i; j++){
				int n = i%j;
				if (n==0){
					break;
				}
			}
			if(i == j){
				System.out.print("  "+i);
			}
		}
	}
}
```

Now the same program is coded using MiVa(new syntax specification)

```
#java.io.*;
- PrimeNumber {
	+ S (String[] args) throws Exception {
		inti,n=50;
		"/n Prime number: ";
			i=1; i<num; i++ {
			int j;
				j=2; j<i; j++ {
				int n = i%j;
				if n==0 {
					break;
				}
			}
			ifi == j{
				+i;
			}
		}
	}
}
```

IV. GRAMMAR VALIDATION FOR MIVA' SYNTAX

The grammar validation for the newly defined syntax specification is done using a Context-Free Grammar checker tools available in the web. As discussed earlier the new syntax is defined using symbols andcharacters. Here' the example of import' syntax specification.

Following is the input for the grammar checker :-

```
IMPORT -> # WHITESPACES CLASSNAME
WHITESPACE DELIMITER.
DELIMITER -> \n |; .
WHITESPACES -> tab WHITESPACES | space
WHITESPACES | empty.
CLASSNAME -> STRING dot CLASSNAME | * |
STRING.
STRING -> CHAR STRING | char.
CHAR-> a |b c |d e |f |g |h |i |j |k |l |m |n |o |p |q |r |s
|t |u |v |w |x |y |z |A |B |C |D |E |F |G |H |I |J |K
```

|L |M |N |O |P |Q |R |S |T |U |V |W |X |Y |Z.

Following table is the output for the import grammar :-

nonterminal	first set	follow set	nullable	endable
IMPORT	#	Ø	no	yes
WHITESPACE	Ø	Ø	no	no
DELIMITER	n ;	Ø	no	yes
WHITESPACES	tab space empty	* char a b d f g h i j k l m n o p q r s t u v w x y z	no	no
CLASSNAME	* char a b d f g h i j k l m n o p q r s t u v w x y z	Ø	no	no
STRING	char a b d f g h i j k l m n o p q r s t u v w x y z	dot	no	no
CHAR	a b d f g h i j k l m n o p q r s t u v w x y z	char a b d f g h i j k l m n o p q r s t u v w x y z	no	no

*This can be verified in the following link: **http://tinyurl.com/dxh9qew**

V. PHASES OF EXECUTION

1) Loading the Grammar:

In this initial stage, the defined grammar are codified into the java file (Grammar.java) which comprises of the grammar production rules for the MiVa' syntax specification.

Secondly, the syntax is taken into the memory by another java code (Parser2.java) which loads the input code into the system memory.

```
*************Grammar.java**************
packageorg.miva.compiler.micro;
importjava.util.ArrayList;
importjava.util.HashMap;
importjava.util.List;
importjava.util.Map;
/**
 * @author binu
 *
 */
@SuppressWarnings("serial")
public class Grammar {
/**
* The EBNF productions that defines the MiVa language syntax
 * specifications. TODO:The hard coded EBNF production rules should be moved
 * to a separate grammar file.
 */

private static final String[] G_PRODUCTIONS = {
"s-tilde<-->'~'",
"s-dot<-->'.'",
"s-semicolon<-->';'",
"s-colon<-->':'",
"newline<-->'\n'",
"empty-character<-->''",
"whitespace<-->{' '|'\t'}",
"alpha-character<--
>'A'|'B'|'C'|'D'|'E'|'F'|'G'|'H'|'I'|'J'|'K'|'L'|'M'|'N'|'O'|'P'|'Q'|'R'|'S'|'T'|'U'|'V'|'W'|'X'|'Y'|'Z'|'a'|'b'|'c'|'d'|'e'|'f'|'g'|'h'|'i'|'j'|'k'|'l'|'
m'|'n'|'o'|'p'|'q'|'r'|'s'|'t'|'u'|'v'|'w'|'x'|'y'|'z'",
"numeric-character<-->'1'|'2'|'3'|'4'|'5'|'6'|'7'|'8'|'9'|'0'",
"delimiter<-->s-semicolon",
"character<-->alpha-character|numeric-character",
"string<-->{character}|empty-character",
"non-empty-alpha-begin-string<-->alpha-character__string",
"group-dot-and-non-empty-alpha-begin-string<-->(s-dot,numeric-character)",
"package<-->s-tilde__[{whitespace}]__non-empty-alpha-begin-string__[{group-dot-and-non-empty-alpha-
begin-string}]__[{whitespace}]__[delimiter]",
// "package<-->[{group-dot-and-non-empty-alpha-begin-string}]__[{whitespace}]__[delimiter]"
};
/**
* Data structure to hold all the 'EBNF - Java Object' interpretations*/
```

```
public static final Map<String, Component> G_PRODUCTION_NAMES_N_RULES = new HashMap<String,
Component>();
/**
* EBNF terminal(literal) start/ end character
 */
private static final char TERMINAL_START_N_END = '\"';
/**
 * EBNF 'group' start character that indicates a group of non-terminals and/
 * or terminals start point
 */
private static final char GROUP_START = '(';
/**
 * EBNF 'group' end character that indicates a group of non-terminals and/
 * or terminals end point
 */
private static final char GROUP_END = ')';
/**
 * EBNF 'optional' start character that indicates a non-terminals and/ or
 * terminal's optional status start point
 */
private static final char OPTIONAL_START = '[';
/**
 * EBNF 'optional' end character that indicates a non-terminal's and/ or
 * terminal's optional status end point
 */
private static final char OPTIONAL_END = ']';
/**
 * EBNF 'repeat' start character that indicates a non-terminal's and/ or
 * terminal's repeatable status start point
*/
private static final char REPEATABLE_START = '{';
/**
 * EBNF 'repeat' end character that indicates a non-terminal's and/ or
 * terminal's repeatable status end point
 */
private static final char REPEATABLE_END = '}';
/**
 * EBNF 'alternate' indicator of a sub-component in a production rule
 */
private static final char ALTERNATOR = '|';
/**
 * Mapping of start and end indicators of grouped, repeatable, optional sub
* components
 */
private static final Map<Character, Character> GRAMMAR_SYMBOL_MAPPINGS;
static
 {GRAMMAR_SYMBOL_MAPPINGS = new HashMap<Character, Character>() {
{
put(TERMINAL_START_N_END, TERMINAL_START_N_END);
put(GROUP_START, GROUP_END);
put(OPTIONAL_START, OPTIONAL_END);
put(REPEATABLE_START, REPEATABLE_END);
put(ALTERNATOR, ALTERNATOR);
                                    }
                        };
            }
/**
 * Interprets and loads the MiVa EBNF to memory
 */
public static void load() {
```

```
intp_name = 0;
intp_rule = 1;
for (String g_production : G_PRODUCTIONS)
{ String[] g_production_name_n_rule = g_production.split("<-->");
// The parent node of the production rule being parsed
Component parent_component = new Component();
for (String p_rule_component : g_production_name_n_rule[p_rule].split("__")) {
// Parse the production rule's subcomponents and map it to //the parent node in the grammar tree
parseComponentGrammar(p_rule_component, parent_component,Boolean.FALSE);
}
// Grammar tree holder
G_PRODUCTION_NAMES_N_RULES.put(g_production_name_n_rule[p_name],parent_component);
System.out.println(parent_component);
                    }
            }
private static void parseComponentGrammar(String p_rule_component,Componentparent_component, Boolean
alternate)
{
// Contains a list of flags indicating whether a //subcomponent isoptional/ repeatable/ grouped
List<Character>p_rule_component_symbols = new ArrayList<Character>();
// The terminal/ non-terminal that forms the subcomponent
List<Character>p_rule_sub_component = new ArrayList<Character>();
// When alternations are found in a component of the //production
// rule, they are built into a child node's alternations. Every
// node in the production rule tree has a collection of one of
// more child nodes and every child node has a collection of //one of more alternations.
for (intpos = 0; pos<p_rule_component.length(); pos++) {
// Parse through the code one character a time
char token = p_rule_component.charAt(pos);
switch (token)
{case TERMINAL_START_N_END:
if (!p_rule_component_symbols.isEmpty()
&&p_rule_component_symbols.get(
p_rule_component_symbols.size() - 1).equals(
TERMINAL_START_N_END))
 {// A subcomponent has been identified wrapped within
// terminal symbols
String sub_component = convertToString(p_rule_sub_component);
// Clear the subcomponent buffer to enable identifying the
// next subcomponent
p_rule_sub_component.clear();
// Create the memory structure 'Component' to reflect the
// EBNF's subcomponent
Component child = createComponentObject(sub_component,p_rule_component_symbols);
// Attach the child node, just created, to its parent node
attachChildToParent(parent_component, child, alternate);
// A subcomponent has been processed, clear the symbols
// cache as well
p_rule_component_symbols.remove(p_rule_component_symbols.size() - 1);
} else
 {
// A start of a terminal is identified, cache it now and
// look for the end of this terminal subcomponent
p_rule_component_symbols.add(TERMINAL_START_N_END);
}
break;
case ALTERNATOR:
charp_alternate_component_start = p_rule_component
.charAt(pos + 1);
String alternate_component;
```

```
// An alternator is found, check if the subcomponent is a
// terminal, grouped, optional or repeatable
if (GRAMMAR_SYMBOL_MAPPINGS.keySet().contains
(p_alternate_component_start))
{// The subcomponent is not a pointer to another //production rule
alternate_component = p_rule_component.substring(pos + 1, p_rule_component.indexOf(
GRAMMAR_SYMBOL_MAPPINGS.get(p_alternate_component_start),pos + 2) + 1);
} else
 {
// The subcomponent is a pointer to another production rule
alternate_found: {
for (inti = pos + 1; i<p_rule_component.length(); i++) {
if(GRAMMAR_SYMBOL_MAPPINGS.keySet().contains(p_rule_component.charAt(i))) {
alternate_component = p_rule_component.substring(pos + 1, i + 1);
breakalternate_found;
}
alternate_component = p_rule_component.substring(
pos + 1, p_rule_component.length());
}
}
parseComponentGrammar(alternate_component, parent_component,Boolean.TRUE);
pos = pos + alternate_component.length();
break;
case GROUP_START:
String p_group_component = p_rule_component.substring(pos + 1,p_rule_component.indexOf(GROUP_END,
pos + 1));
Component group_component = new Component();
group_component.attribute.setGroup(Boolean.TRUE);
String[] p_sub_group_components = p_group_component.split(",");
for (String p_sub_group_component : p_sub_group_components)
{parseComponentGrammar(p_sub_group_component,
group_component, Boolean.FALSE);
}
attachChildToParent(parent_component, group_component,alternate);
skip position of pointer to the end of p_group_component,
// that is beyond the ')' parenthesis
pos = pos + p_group_component.length() + 1;
break;
case OPTIONAL_START:p_rule_component_symbols.add(OPTIONAL_END);
break;
case
OPTIONAL_END:p_rule_component_symbols.remove(p_rule_component_symbols.indexOf(OPTIONAL_EN
D));
case REPEATABLE_START:p_rule_component_symbols.add(REPEATABLE_END);
break;
case
REPEATABLE_END:p_rule_component_symbols.remove(p_rule_component_symbols.indexOf(REPEATABL
E_END));
default:p_rule_sub_component.add(token);
String p_rule_sub_component_as_production = convertToString(p_rule_sub_component);
if (null != G_PRODUCTION_NAMES_N_RULES
.get(p_rule_sub_component_as_production)) {
Component production_component =
createComponentObjectFromProduction(G_PRODUCTION_NAMES_N_RULES.get(p_rule_sub_component_
as_production),p_rule_component_symbols);
attachChildToParent(parent_component, production_component,alternate);
p_rule_sub_component.clear();
}
}
}
```

```
}
private static Component createComponentObject(String sub_component,List<Character> symbols)
{Component component = new Component();
for (Character symbol : symbols) {
switch (symbol) {
case TERMINAL_START_N_END:
component.attribute.setTerminal(Boolean.TRUE);
component.body.addToTerminals(sub_component);
break;
case GROUP_END:
component.attribute.setGroup(Boolean.TRUE);
break;
case OPTIONAL_END:
component.attribute.setOptional(Boolean.TRUE);
break;
case REPEATABLE_END:
component.attribute.setRepeatable(Boolean.TRUE);
break;
}
}
return component;
}
private static Component createComponentObjectFromProduction(
Component production_component, List<Character> symbols)
{Component component = new Component();
for (Character symbol : symbols) {
switch (symbol)
{case TERMINAL_START_N_END:
component.attribute.setTerminal(Boolean.TRUE);
break;
case GROUP_END:
component.attribute.setGroup(Boolean.TRUE);
break;
case OPTIONAL_END:
component.attribute.setOptional(Boolean.TRUE);
break;
case REPEATABLE_END:
component.attribute.setRepeatable(Boolean.TRUE);
break;
}
}
/* * component.attribute.setTerminal(production_component.attribute
 * .getTerminal());
*component.attribute.setGroup(production_component.attribute
* .getGroup());
*/
component.setBody(production_component.getBody());
return component;
}
private static String convertToString(List<Character>p_rule_sub_component)
{
Character[] sub_component_buffer = new Character[p_rule_sub_component.size()];
p_rule_sub_component.toArray(sub_component_buffer);
char[] pca = new char[sub_component_buffer.length];
for (inti = 0; i<sub_component_buffer.length; i++) {
pca[i] = sub_component_buffer[i];
}
returnString.valueOf(pca);
}
private static void attachChildToParent(Component parent, Component child,Boolean append)
```

```
{
if (!append)
 {
parent.body.addNewChild(child);
} else
{
parent.body.appendChildToLastChildAlternations(child);
}
}}
****************Parser2.java*****************
packageorg.miva.compiler.micro;
importjava.util.List;
public class Parser2 {
public static void main(String[] args)
 {
Grammar.load();
Component parent_component = Grammar.G_PRODUCTION_NAMES_N_RULES
.get("package");
displayGrammar(parent_component, 0);
}
public static void displayGrammar(Component parent_component, int depth)
{
System.out.println(depth);
System.out.println("GROUP: " + parent_component.attribute.getGroup());
System.out.println("OPTIONAL: "
+ parent_component.attribute.getOptional());
System.out.println("REPEATABLE: "
+ parent_component.attribute.getRepeatable());
System.out.println("START_SYMBOL: "
+ parent_component.attribute.getStartSymbol());
System.out.println("TERMINAL: "
+ parent_component.attribute.getTerminal());
for (List<Component>child_node : parent_component.body.getChildren())
{
for (Component alternate_node : child_node)
{
if (alternate_node.attribute.isAttributeEmpty())
{
displayGrammar(alternate_node, depth + 1);
} else
if (null != alternate_node.attribute.getTerminal()
&&alternate_node.attribute.getTerminal())
{
System.out.println(alternate_node.body.getTerminals());
} else
if (null != alternate_node.attribute.getRepeatable()
&&alternate_node.attribute.getRepeatable())
 {
displayGrammar(alternate_node, depth + 1);
} else
if (null != alternate_node.attribute.getOptional()
&&alternate_node.attribute.getOptional())
 {
displayGrammar(alternate_node, depth + 1);
} else
if (null != alternate_node.attribute.getGroup()
&&alternate_node.attribute.getGroup())
{
displayGrammar(alternate_node, depth + 1);
}
```

2) Parsing the Code:

The code is now analyzed using a java file (Parser1.java) which parse the input code loaded into the memory by checking it in reference with the Grammar production rules defined in the Grammar file.

```
*****************Parser1.java*******************
packageorg.miva.compiler.micro;
importjava.util.List;
public class Parser1
{
public static String CODE = "~3org.3;";
public static Integer CHARACTER_COUNTER = 0;
public static void main(String[] args)
{
Grammar.load();
Component parent_component = Grammar.G_PRODUCTION_NAMES_N_RULES
.get("package");
if (parseCode(parent_component, 0))
{
if (CHARACTER_COUNTER <CODE.length())
{
System.out.println("incomplete");
} else
 {
System.out.println("success");
}
} else
{
System.out.println("failure");
}
System.out.println("column " + CHARACTER_COUNTER + " CHAR: "
+ CODE.charAt(CHARACTER_COUNTER));
}
public static booleanparseCode(Component parent_component, int depth)
 {
System.out.println(depth);
System.out.println("GROUP: " + parent_component.attribute.getGroup());
System.out.println("OPTIONAL: "
+ parent_component.attribute.getOptional());
System.out.println("REPEATABLE: "
+ parent_component.attribute.getRepeatable());
System.out.println("START_SYMBOL: "
+ parent_component.attribute.getStartSymbol());
System.out.println("TERMINAL: "
+ parent_component.attribute.getTerminal());
children: for (List<Component>child_node : parent_component.body.getChildren())
 {
for (Component alternate_node : child_node)
{
if (alternate_node.attribute.isAttributeEmpty())
{
if (parseCode(alternate_node, depth + 1))
continue children;
} else
if (null != alternate_node.attribute.getTerminal()
&&alternate_node.attribute.getTerminal())
{
System.out.println(alternate_node.body.getTerminals());
if (CHARACTER_COUNTER == CODE.length())
{
return false;
} else
```

```
if (alternate_node.body.getTerminals().contains(
String.valueOf(CODE.charAt(CHARACTER_COUNTER))))
{
CHARACTER_COUNTER++;
continue children;
}
} else
if (null != alternate_node.attribute.getRepeatable()
&&alternate_node.attribute.getRepeatable())
{
boolean optional = null != alternate_node.attribute
.getOptional()&&alternate_node.attribute.getOptional();
intrepeat_counter = 0;
while (true)
{
if (parseCode(alternate_node, depth + 1))
{
repeat_counter++;
} else
 {
if (repeat_counter> 0 || optional)
continue children;
else
return false;
}
}
} else
if (null != alternate_node.attribute.getOptional()
&&alternate_node.attribute.getOptional())
{
parseCode(alternate_node, depth + 1);
continue children;
} else
if (null != alternate_node.attribute.getGroup()
&&alternate_node.attribute.getGroup())
{
intstart_counter = CHARACTER_COUNTER;
if (parseCode(alternate_node, depth + 1))
{
continue children;
} else
{
CHARACTER_COUNTER = start_counter;
return false;
}
}
}
return false;
}
return true;
}
}
```

VI. SAMPLE OUTPUT

- Checking for Package syntax

~org;//checking for a valid package name
Console output for that example:
0

GROUP: null
OPTIONAL: null
REPEATABLE: null
START_SYMBOL: null
TERMINAL: null
1
GROUP: null
OPTIONAL: null
REPEATABLE: null
START_SYMBOL: null
TERMINAL: null
[~]
1
GROUP: null
OPTIONAL: true
REPEATABLE: true
START_SYMBOL: null
TERMINAL: null
[ ]
[ ]
1
GROUP: null
OPTIONAL: null
REPEATABLE: null
START_SYMBOL: null
TERMINAL: null
2
GROUP: null
OPTIONAL: null
REPEATABLE: null
START_SYMBOL: null
TERMINAL: null
[A][B][C][D][E][F][G][H][I][J][K][L][M][N][O]
[P][Q][R][S][T][U][V][W][X][Y][Z][a][b][c][d]
[e][f][g][h][i][j][k][l][m][n][o]
2
GROUP: null
OPTIONAL: null
REPEATABLE: null
START_SYMBOL: null
TERMINAL: null
3
GROUP: null
OPTIONAL: null
REPEATABLE: true
START_SYMBOL: null
TERMINAL: null
4
GROUP: null
OPTIONAL: null
REPEATABLE: null
START_SYMBOL: null
TERMINAL: null
[A][B][C][D][E][F][G][H][I][J][K][L][M][N][O]
[P][Q][R][S][T][U][V][W][X][Y][Z][a][b][c][d]
[e][f][g][h][i][j][k][l][m][n][o][p][q][r]
3
GROUP: null
OPTIONAL: null
REPEATABLE: true

START_SYMBOL: null
TERMINAL: null
4
GROUP: null
OPTIONAL: null
REPEATABLE: null
START_SYMBOL: null
TERMINAL: null
[A][B][C][D][E][F][G][H][I][J][K][L][M][N][O]
[P][Q][R][S][T][U][V][W][X][Y][Z][a][b][c][d]
[e][f][g]
3
GROUP: null
OPTIONAL: null
REPEATABLE: true
START_SYMBOL: null
TERMINAL: null
4
GROUP: null
OPTIONAL: null
REPEATABLE: null
START_SYMBOL: null
TERMINAL: null
[A][B][C][D][E][F][G][H][I][J][K][L][M][N][O]
[P][Q][R][S][T][U][V][W][X][Y][Z][a][b][c][d]
[e][f][g][h][i][j][k][l][m][n][o][p][q][r][s]
[t][u][v][w][x][y][z]
4
GROUP: null
OPTIONAL: null
REPEATABLE: null
START_SYMBOL: null
TERMINAL: null
[1]
[2]
[3]
[4]
[5]
[6]
[7]
[8]
[9]
[0]
1
GROUP: null
OPTIONAL: true
REPEATABLE: true
START_SYMBOL: null
TERMINAL: null
2
GROUP: true
OPTIONAL: null
REPEATABLE: null
START_SYMBOL: null
TERMINAL: null
3
GROUP: null
OPTIONAL: null
REPEATABLE: null
START_SYMBOL: null

TERMINAL: null
[.]
1
GROUP: null
OPTIONAL: true
REPEATABLE: true
START_SYMBOL: null
TERMINAL: null
[ ]
[ ]
1
GROUP: null
OPTIONAL: true
REPEATABLE: null
START_SYMBOL: null
TERMINAL: null
2
GROUP: null
OPTIONAL: null
REPEATABLE: null
START_SYMBOL: null
TERMINAL: null
[;]
success

~3org.3;//checking for an invalid package name

Console Output for that example:
0
GROUP: null
OPTIONAL: null
REPEATABLE: null
START_SYMBOL: null
TERMINAL: null
1
GROUP: null
OPTIONAL: null
REPEATABLE: null
START_SYMBOL: null
TERMINAL: null
[~]
1
GROUP: null
OPTIONAL: true
REPEATABLE: true
START_SYMBOL: null
TERMINAL: null
[ ]
[ ]
1
GROUP: null
OPTIONAL: null
REPEATABLE: null
START_SYMBOL: null
TERMINAL: null
2
GROUP: null
OPTIONAL: null

REPEATABLE: null
START_SYMBOL: null
TERMINAL: null
[A][B][C][D][E][F][G][H][I][J][K][L][M][N][O]
[P][Q][R][S][T][U][V][W][X][Y][Z][a][b][c][d]
[e][f][g][h][i][j][k][l][m][n][o][p][q][r][s]
[t][u][v][w][x][y][z]
failure
column 1 CHAR: 3

VII. CONCLUSION

Minimal Java (MiVa), in this paper the analysis is made on the verbosity of the java language's syntax specification and complexity in debugging the java source code in sense of syntax errors. So, the MiVa language specification will be developed in such a way it reduces the syntax verbosity and brings comfort in codifying the task. A new transpiler has been developed specifically to recognize the syntax specification of Minimal Java and performs tasks such as analyzing and parsing. The program will be written in file with .miva as a file extension and compiled using MiVatranspiler which produces an output consisting of java' syntax specification accordingly. MiVaTranspiler will notify the warnings and errors produced while compiling the .miva file. Once the Transpiler produces the desired Java code for the given MiVa code, the existing java compiler can be used to generate the machine runnable code. MiVa' syntax specification working together with the Transpiler will supply the advantages in the programming task like reducing the complexity present in syntax specification due to verbosity, increasing the understandability of the program structure and reducing the occurrence of errors in syntax level.

VIII. FUTURE ENHANCEMENTS

Regarding the future development, the system could be powered for generating the machine runnable code by itself by taking in the MiVa code which avoids the need for Java compiler or any other compilers for generating the target code. The system will be supplied with better debugging solutions available in the IDEs.

REFERENCES

[1]. Alfred V. Aho, Ravi Sethi, Jeffrey D. Ullman (2007), "Compilers: principles, techniques, and tools." Pearson Education Inc.
[2]. D. Grune, H. Bal, C. Jacobs, K. Langendoen (2000), "Modern Compiler Design (1st edition)" WileyPublishing.
[3]. Mikael Olsson (2011), "Handbook of Java Syntax: A Reference to the Java Programming Language." CreateSpace Independent Publishing Platform.
[4]. Oracle (2013) "The JavaTM Tutorials" [Online] http://docs.oracle.com/javase/tutorial/index.html.
[5]. Richard E. Pattis, Carnegie Mellon University (2005), "Extended Backus-Naur Form or Backus-Normal Form (EBNF) "[Online] http://www.cs.cmu.edu/~pattis/misc/ebnf.pdf.
[6]. Robert Lafore (2002), "Data Structures and Algorithms in Java (2nd Edition)", Sams Publishing
[7]. Salmon (1992), "Backus-Naur Forms", Irwin Professional Publishing