JGroovy – an Extensible Java Programming Language with Groovy

Siwadol Sateanpattanakul*, Aranya Walairacht*

*Department of Computer Engineering, King Mongkut's Institute of Technology Ladkrabang, Chalongkrung Rd., Ladkrabang, Bangkok 10520 Thailand

sidol.sat@gmail.com, kwaranya@kmitl.ac.th

Abstract— **Java is Object-Oriented Programming Languages (OOPL) that widely used for software development. But Java has a limitation for working with Domain-Specific Languages (DSLs). Java language structure and syntax has not more support for working with DSLs and including type of Java language. Static language likes Java does not flexible more for DSLs handle. This limitation has to solve by adding new language structure and syntax into Java language. Groovy is a dynamic programming languages that support DSLs with internal structure. In this paper, we introduce "JGroovy" which is extended Java programming language with Groovy programming. We are built JGroovy compilers that fully support Java programming language.**

Keywords — **Object-Oriented Programming, Domain-Specific Language, Compiler, Automata, Language Oriented Programming**

I. INTRODUCTION

Language Oriented Programming (LOP) [1] is computer programming paradigm for solving a specific problem with domain-specific language (DSL) and general-purpose languages (GPL). A language specification of DSL can be seen as any language what is not general purpose. The LOP can consists of a set of DSLs which manages group of specific problem. The famous of Object–Oriented programming language (OOPL) like Java has enabled to support programmers who want to written LOP paradigm. The Java Specification Language (JSL) [9] was prepared some specification to working with DSL like this.

Fluent Interface

Annotations

Generics

These language specifications have been allowed programmer to write a domain language on top of Java with easy technique. But it is not easy likes that because there are two kinds of DSLs: internal and external. And DSLs that define in JSL is internal DSLs. The Java language is not a better fit for creating internal DSLs because static languages like Java cannot take all advantages of metaprogramming technique to writing internal DSLs. On the other word, Java language is an appropriate for creating external DSLs, which are more depend on the host language syntax.

There are many programming languages what are based on Java language [18]. And those languages are compatible with Java Virtual Machine such as JRuby [13], Groovy [11], and Scala [5]. The programming language as Groovy is designed for working with internal DSLs, because that programming language adds powerful metaprogramming features for support internal DSLs development. Groovy programming language is appropriated for developing internal DSLs because it is dynamic language. The power of dynamic methods of the Groovy language allows developers to add and invoke methods dynamically. So developers and programmers can add methods at runtime because Groovy maintains a metaclass for each Java class.

Java is a programming language that supports an external DSL. It is easy when we want to extend a new programming language into the Java programming language because an external DSL is supported in easy way. The advantage of new language structure is the way for working with another DSL. Moreover, the language structure is not always change when programmers are working with another DSL. The Groovy programming language has some proper structure for manage DSL in kind of internal. So, the new language is using Groovy language features for handle problem. The extensible language is supported an internal DSL for manage specific problems.

In this paper, we extended the Java programming language with the Groovy programming language for improving Java programming with external DSLs technique. We extended Groovy programming language features on top of Java programming language where Groovy incorporates Java construct as parts.

In the rest of this paper, we describe the related work in section 2, and describe system overview in section 3. In section 4, 5 and 6, we demonstrate all sub-modules in our compiler. Section 7, we represent preliminary experiment of the JGroovy compiler. Finally, the future work and concludes is explained in Section 8.

II. RELATED WORK

There have been many efforts to build tool for extensible compiler for Java and other languages. We demonstrate those extended languages and compilers.

JMatch [3] is an extension Java programming language with support for data abstraction and iteration abstraction. JMatch

has mechanism for pattern matching that support data abstraction features of Java .It handle iteration abstractions to convenient for use and implement.

MetaBorg [2] is an embedding tool for adding domain-specific languages. It allows one to extend a host language by adding concrete syntax for objects. MetaBorg is handling the specification of conditional abstract syntax tree with Stratego/XT rewriting system.

AspectJ [7] is an aspect-oriented extension to Java. AspectJ is a new kind of programming that used Aspect-oriented programming [15] (AOP) for resolve cross-cutting concern.

JaCo [4] is an extensible Java 1.4 compiler for algebraic types with defaults. JaCo concept is used to design Keris [17] programming language which adds an expressive module system to Java.

III. AN OVERVIEW OF JGROOVY COMPILER

The JGroovy compiler system comprises of several sub-modules which work together. There are three extensive sub-modules in JGroovy compiler; lexical analyzer, syntactic and semantic analyzer (Figure 1). The JGroovy compiler is an extension Java compiler with Groovy specification language. Therefore,the JGroovy compiler is fully supported Java 5 language specification.

The JGroovy compiler use Java programming language as a host language, and use Groovy programming language for extended language. There are many different features between Java and Groovy. We described what Groovy features are necessary for extended to Java in section 3.1 and how to extended Groovy feature to host language in section 3.2.

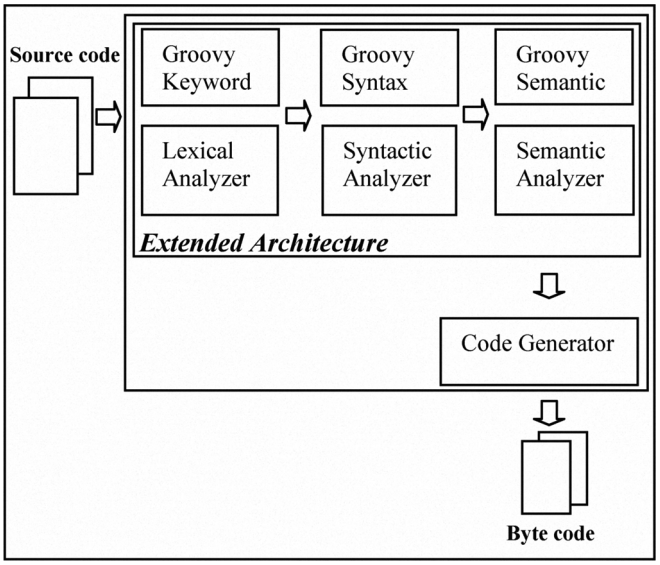

Figure 1. JGroovy compiler achitechture

A. Extending Groovy features in Java

Groovy is a dynamic language programming for the JVM, offers a lot of features that allow programmer to create embedded DSLs also called a "mini language" for a specific set of tasks. We will discuss in more detail why and how we carried out an extending of Groovy in Java.

First, Groovy is object-oriented language that runs on the JVM. Groovy is supported Java 1.4 or newer. Moreover, the programming syntax is familiar to Java programmers. As a Java programmer, you c an use almost of Java programming skill for Groovy. Then in aspect of language designer, you will build a language simply by reusing different language definition modules, such as modules for expressions, declarations, etc. Finally, it supports DSLs and smart syntax. So the source code is easy to maintain.

However, there are some features of Groovy that are not available in Java as follows.

- Closure
- Native syntax for lists and maps
- Native support for regular expression
- Dynamic and Static typing
- Embed expression inside strings
- Switch statement and polymorphic iteration
- Smart syntax for writing bean
- And safe navigation

To support Groovy features, the language needs to be extended in several ways. At the semantic analyzer, the language needs to be extended with notation for dynamic type, safe navigation, and native of regular expression. At the syntactic analyzer, the language needs to be extended production rules of Groovy that are not available in Java language.

B. Extensible Java Compilers with Groovy Specification

With the techniques developed and extended language specification in previous section we are now able to build extensible compilers. The main program is consists of two main components: Java component and Groovy component. There are two modules in Java component: Java1.4 and Java5. Each module contains many sub-modules for support each JSL. Groovy components reused and extended Java components (Figure 3). Therefore, the JGroovy compiler is supported all Java API because it extended on top of Java API (Figure 2).

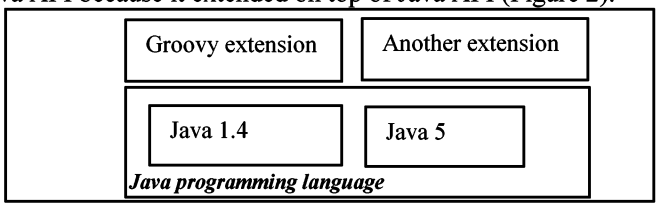

Figure 2. JGroovy: Java extended compilers architecture

In Groovy component is consisted of Groovy frontend and backend. The frontend are semantic checker such as error checking and prettyprintter: they parse Groovy source files together with Java source file and read dependently class file.

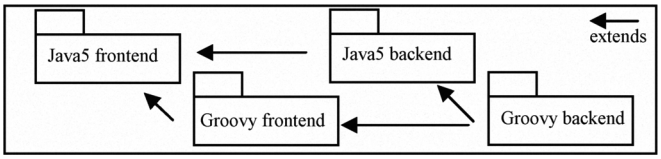

Figure 3. JGroovy: Groovy extension components

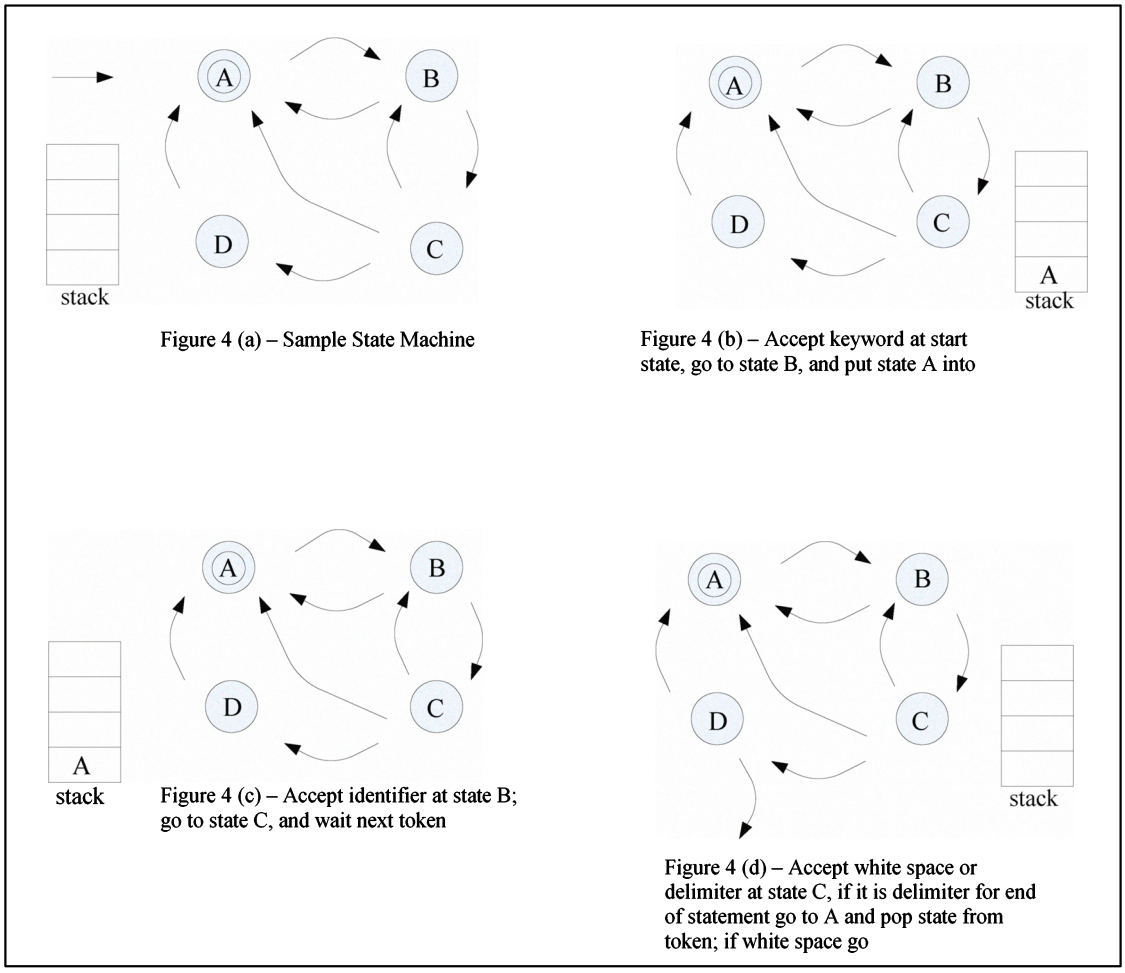

Figure 4 (a) – Sample State Machine

Figure 4 (b) – Accept keyword at start state, go to state B, and put state A into

Figure 4 (c) – Accept identifier at state B; go to state C, and wait next token

Figure 4 (d) – Accept white space or delimiter at state C, if it is delimiter for end of statement go to A and pop state from token; if white space go

Figure 4. Stack-Machine Scanner Architechture

IV. STACK MACHINE LEXICAL ANALYZER

The characters of the source program are scanned by lexical analyzer to discover tokens. Hence, the lexical analyzer must be defined the regular expression (RE) of both languages (Java and Groovy) for describe the token.

The Groovy programming language is almost like the Java programming language. So there are several ambiguous syntaxes when we introduce Groovy Specification Languages (GSL) to JSL. For example delimiter; Java programming language is defined delimiter as semi-colon for statement terminator but Groovy is not. It uses both of semi-colon and line-terminator for end of statement, etc.

We use the stack-machine technique for resolving these problems. The stack-machine scanner will read next-token .Then the stack-machine scanner will approve the input token and assign right state to scanner (Figure 4). The next-token and stack-machine function should be straightforward. The basic steps are as follows:

1. Read first token and put current state into stack-machine (Figure 4(a) and 4(b)).

2. See what the next character is (Figure 4(c)). (Ignore comments and white spaces), and then return the token with the appropriate, and pop or put state to proper state.

-- If the next character is an operator, or delimiter, return the appropriate kind (Figure 4(d)). (e.g.,, if the character is "(", return the token with the "kind" value of "LPAREN", go to another state which is upon current state and put current state);

-- If it is a number (integer), return the token with the "kind" value of "INTEGER" and the "val" value of the whole number, but not just the first digit; and

-- If it is a word (string/character array), return the token with the "kind" value of "IDENTIFIER" and the "id" value of the whole string, but not just the first character. If the string is a keyword (e.g., "if", "else", etc.), return the token with the proper "kind" value.

In addition, the lexical analyzer has to common operator for handle many DSLs. We use a closure that is an operator to show and accept any number of strings. An operator * is used to representing closure L* to denote the concatenation of language L with itself any number of times, that is

$$L^* = \bigcup_{i=0}^{\infty} L^i \quad (1)$$

With stack-machine technique, the lexical analyzer can describe particular pattern of both languages.

V. SYNTAX ANALYZER EXTENSIONS

The syntax analyzer determines whether the stream of tokens from the lexical analyzer from a valid sentence in programming language grammar.

To support Groovy features in Java, the parsing grammar must be extended the Groovy keyword and delimiters such as def, in, and including productions rule for dynamic declaration, literal, and groovy for-loop etc. An example that represents declaration and instantiation between host language and extended languages is shown below.

```
public class Demo {
        public static void main(String args) {

        Demo demo1 = new Demo() ;

        def demo2 = new Demo() }
}
```

Figure 5. Variable declaration of Java and Groovy

In a grammar, the strings that returned by the lexer and symbols which appear in the input sentence to the parser are terminal symbols or tokens, while those that appear on the left-hand of the some rule are nonterminal symbols. We extended the production rule of the dynamic declaration that consists of terminal and nonterminal symbols (Figure 6).

```
Access dynamic_type
    = DEF {: return new DynamicTypeAccess("def"); :} ;

BodyDecl def_declaration
    = modifiers.m? dynamic_type.t def_declarators.v SEMICOLON
        {: return new FieldDecl(new Modifiers(m), t, v); :}
```

Figure 6. JGroovy : The production rule of dynamic varible

The JastAdd uses abstract context-free grammar for basic specification. Each production rule has specific abstract context-free grammar. Each abstract grammar has a specific Abstract Syntax Tree (AST). The AST is used to design model of programming language structure as a class hierarchy (Figure 7) with abstract class like statement and expression. In the other word, the AST is an augmented class hierarchy with subcomponent information corresponding to production on the right-hand sides.

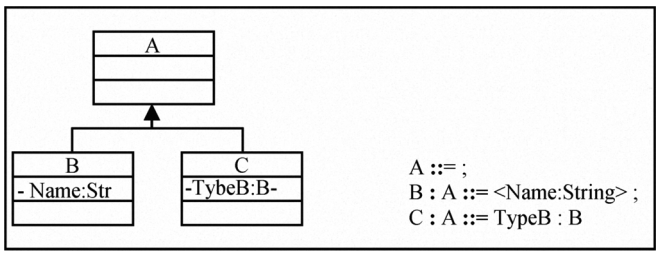

Figure 7. Class heirachy of AST

A. Java Syntax Definitions

Of cause, we use Java programming language as a host language. Therefore, Java programming language is firstly language that design for using with compiler. The AST is used to design compiler structure. The AST structure is used to a guide line to design parsing grammar. For example; The AST of Java variable declaration is showed in figure 8. On the left-hand side of the AST (before "::="),there must be a single variable. The single variable is nonterminal symbol. While the right-hand side has a special form that describe the production rule or terminal symbol.

```
VarAccess : Access ::= <ID:String>;

TypeAccess : Access ::= <Package:String> <ID:String>;

FieldDecl : MemberDecl ::= Modifiers TypeAccess:Access
VariableDecl*;
```

Figure 8. An AST of Java varible declaration

B. Groovy Syntax Definitions

The AST of Groovy language is Groovy language structure definition likes Java language. The AST of Groovy is extended from Java's AST in kind of inheritance. Because the characteristic of both languages are resemble. We can reuse AST structure from Java language such as variable declaration etc.

The syntax of Groovy is defined and created on JastAdd [16] specification, a Java-based system for compiler implementation. A JastAdd component consist of an object-oriented abstract grammar that defines the AST, reference attributes for behavior specifications, and the context free grammar for parsing java and extensions. The JastAdd system is transformed all Groovy and Java syntax definition to JGroovy compiler source code.

VI. SEMANTIC ANALYZER EXTENSIONS

In the previous section we describe how the Groovy is syntactically extended in Java using the modular syntax definition formalism that is defined by JastAdd. From this extending we can generate parser. The parser can parse program which written in Java and Groovy form. The JastAdd is a prepared module for checking semantic as static-semantic checking. We can extend these modules or create new module for managing system behavior. The Reference Attribute Grammars [14] (RAGs) is used to handle and access class in semantic module. The semantic module is created for guarantee the correctness of source code before transform to byte code.

RAGs are specific file that use in JastAdd. RAG is specific language for handle compiler behavior. It composes of ordinary Java language and specific language. The specific language is a kind of aspect-oriented programming.

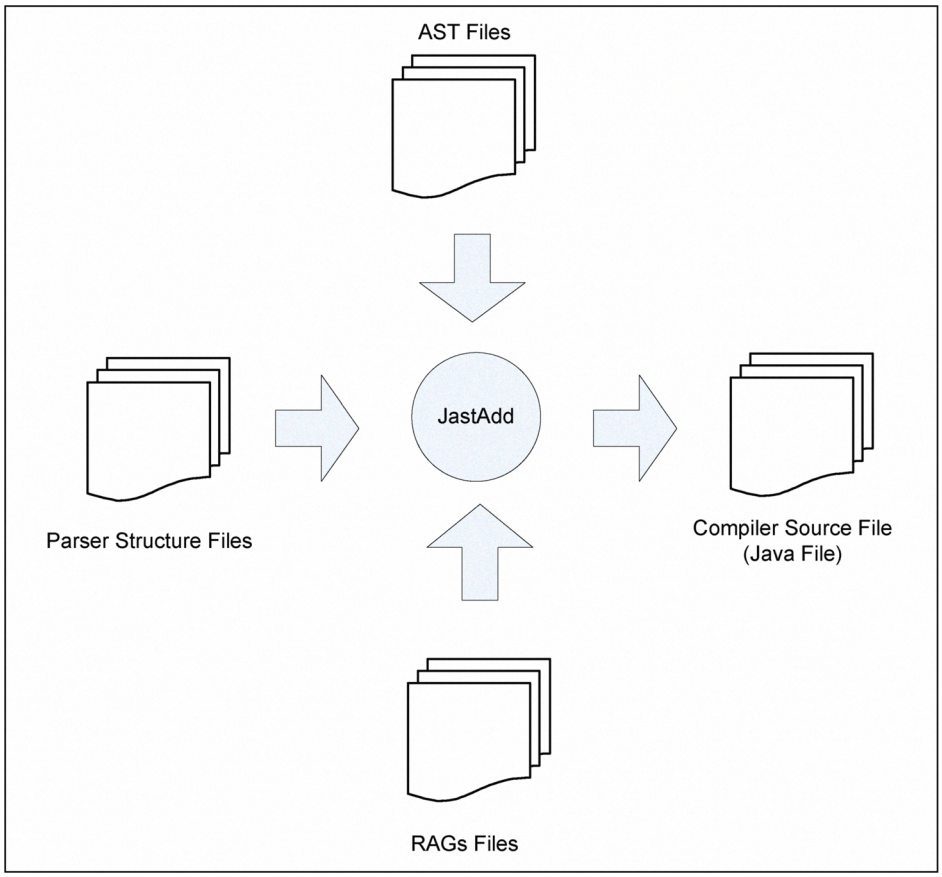

Figure 9. JastAdd compiler construction system

There are several modules which design in previous section, and all sections must be verified and approved by the JastAdd compiler construction system. The JastAdd is checked AST, syntactic and semantic module. Finally, the Jastadd is transformed all modules to compiler source code (Figure 9).

VII. PRELIMINARY EXPERIMENT

All tests were carried out on an Asus A8jseries Duo T2350 with 1 Gbyte memory, running Microsoft Window Xp Sp2.

To evaluate affects of our implementation technique, we have evaluated by comparing it in to Java compiler, Groovy Compiler and JastAddJ5 [10] compiler. We have used a test suite of Jacks [6] compiler to evaluate all compilers. Jacks with 4619 test cases were conducted to test performance of three compilers. Each compiler was tested for three times. Jacks test cases are designed and maintained by the Mauve project. A test suite is designed from JLS. A goal of each test case is a compile result corresponding to an expected result of a test case. For example, if we want to test a unicode-escape. A test case is shown in figure 10. Compiler must show up an error while working on bodies of source code.

The Jacks test suite prepares many modules for testing a compiler. Lists of testing module are shown below.

```
tcltest::test 3.3-invalid-1 { A unicode sequence must

have four hex digits } {
compile [saveas T33i1.java {class T33i1

{ char c = '\u20'; }}]

} {FAIL}
```

Figure 10. An example of Jacks test case

- Arrays
- Blocks and statements
- Classes
- Conversions and promotions
- Definite assignment
- Expressions
- Interfaces
- Lexical structure
- Names
- Packages
- Types values and variables
- Etc.

We use Javac compiler version jdk1.5.0_06 and including JastAddJ5 for testing and comparing with the JGroovy compiler. This experiment is conducted to evaluate an error of

JGroovy compiler with Java5 specification after Groovy features have been extended. The result of experiment is shown below.

Table 1. RESULT OF COMPILERS EXPERIMENT

Compiler	Pass	Skip	Fail	% Pass
Javac 5	4540	0	79	98.3
JastAddJ5	4569	0	50	98.9
JGroovy	4557	0	62	98.6

The results of experiment show that the JGroovy compiler with test cases found error less than Javac 5 compiler. And, the JGroovy compiler found error more than JastAddJ5.

Because JGroovy compiler supports more syntax than Java compiler, it found error more than JastAddJ5. However, percentages of pass for three compilers are almost same. It shows that JGroovy has performance as other two compilers.

However, there are some errors that happen in JGroovy compiler. They affect to the JGroovy performance. The Lists of error are shown below.

- Floating point literals
- Syntactic classification

These errors happen from unclearly dot (.) operator definition and it can be fixed in semantic level. We will fix it on the next testing.

VIII. CONCLUSIONS

In this paper we have presented an extensible Groovy programming language to Java 5.

We have built JGroovy compiler with Java languages specification and Groovy language specification. The both of languages can work together with Stack Machine Lexical Analyzer. We also have tested performance of JGroovy. The experiment result of JGroovy shows that it supports almost Java language specification 5.A conclusion may review the main points of the paper. Please do not replicate the abstract as the conclusion. A conclusion might elaborate on the importance of the work or suggest applications and extensions.

ACKNOWLEDGMENT

We are grateful to Somsak Walairacht and to the anonymous reviewers for their constructive comments. Financial Support from the Aun/Seed-Net is gratefully acknowledged. We also thank Ms. S. Chokpaiboon who provided valuable feedback.

REFERENCES

[1] M. Ward, "Language Oriented Programming," Software---Concepts and Tools 15 (1994), 147—161.

[2] M. Bravenboer, R. de Groot, and E. Visser. MetaBorg in Action: Examples of Domain-specific Language Embedding and Assimilation using Stratego/XT. In R. Lämmel and J. Saraiva, editors, Proceedings of the Summer School on Generative and Transformational Techniques in Software Engineering (GTTSE 2005), volume 4143 of Lecture Notes in Computer Science, pages 297--311, Braga, Portugal, 2006.Springer Verlag.

[3] Jed Liu, Andrew C. Myers. JMatch: Iterable Abstract Pattern Matching for Java, Proceedings of the 5th International Symposium on Practical Aspects of Declarative Languages (PADL'03), pp. 110–127, New Orleans, LA, Jan. 2003. LNCS 2562

[4] Matthias Zenger, Martin Odersky. Implementing Extensible Compilers, ECOOP 2001 Workshop on Multiparadigm Programming with Object-Oriented Languages, Budapest, June 2001.

[5] David Pollak. Beginning Scala, 1st edition. Apress Berkely, CA, USA. May 2009.

[6] The Jacks compiler test suite, 2008.http://sources.redhat.com/mauve/.

[7] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, MikKersten, Jeffrey Palm, and William G. Griswold.An overview of AspectJ. In Proceedings of ECOOP2001, volume 2072 of LNCS, pages 327–355.Springer, 2001.

[8] Gorel Hedin and Eva Magnusson. JastAdd: an aspect-oriented compiler construction system.Science of Computer Programming, 47(1):37–58,2003.

[9] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. The Java Language Specification Third Edition. Addison-Wesley, Boston, Mass., 2005.

[10] Torbjorn Ekman and Gorel Hedin. The JastAdd Extensible Java Compiler. OOPSLA'07, Montreal, Quebec, Canada

[11] Venkat Subramaniam. Programming Groovy: Dynamic Productivity for the Java Developer. http://pragprog.com.2008

[12] Groovy Language Specification, 2008. Available: http://groovy.codehaus.org/jsr/spec/

[13] JRuby. www.jruby.codehaus.org/

[14] Hedin, G., Reference Attributed Grammars, in D.Parigot and M. Mernik,eds., Second Workshop on Attribute Grammars and their Applications,WAGA'99, Amsterdam, The Netherlands, (1999), 153{172. INRIA Roc-quencourt

[15] Kiczales, G., et al. Aspect-Oriented Programming, ECOOP'97, LNCS 1241(1997), 220{242. Springer Verla

[16] Gorel Hedin and Eva Magnusson. JastAdd: an aspect-oriented compiler construction system. Science of Computer Programming, 47(1):37–58, 2003.

[17] The Programming Language Keris .http://zenger.org/keris/

[18] The Programming Language for JVM. http://www.is-research.de/info/vmlanguages/http://zenger.org/keris