

COMPONENT-BASED LANGUAGE IMPLEMENTATION WITH  
OBJECT-ORIENTED SYNTAX AND ASPECT-ORIENTED SEMANTICS

by

XIAOQING WU

BARRETT R. BRYANT, COMMITTEE CHAIR

JEFF GRAY

MARJAN MERNIK

ALAN SPRAGUE

MURAT TANIK

A DISSERTATION

Submitted to the graduate faculty of The University of Alabama at Birmingham,  
in partial fulfillment of the requirements for the degree of  
Doctor of Philosophy

BIRMINGHAM, ALABAMA

2007

Copyright by  
XIAOQING WU  
2007

# COMPONENT-BASED LANGUAGE IMPLEMENTATION WITH OBJECT-ORIENTED SYNTAX AND ASPECT-ORIENTED SEMANTICS

XIAOQING WU

## ABSTRACT

Providing proper modularity is one of the major challenges in software development. In the context of programming language implementation, two key factors impede the modularization process in a compiler system: First, the parser, as the core of the language implementation, usually is specified in a non-decomposable style such that it must be built as a whole regardless of the complexity of the grammar. This usually yields poor comprehensibility, reduced changeability, limited reusability, and hampers independent development of the language implementation. Second, the interconnected nature of syntax and semantic analysis of a language propels the compiler implementation to have each phase tangled together. As a result, the construction of one phase always “pollutes” a different phase, which makes a system hard to develop, maintain and extend. These two modularization complications create the intricacy in compiler design known as a “dragon” task.

With an aim to address the modularity problems and alleviate the implementation complexity in language implementations, this thesis presents a new language implementation formalism and archetype that improves the modularity in compiler construction in a two-dimensional manner. Structure wise, a new parsing algorithm called Component-based LR parsing is utilized to decompose a language implementation into executable components at the byte-code level, which decreases the complexity of building a large language by constructing a set of smaller languages. Function wise, the framework employs object-oriented and aspect-oriented programming paradigms to describe syntax and

semantic entities separately, which facilitates separation of concerns by isolating syntax and semantics as well as semantic phases themselves into different modules. The techniques in these two dimensions work in a coherent manner, producing a solution that can solve both modularization obstacles introduced above. Consequently, the framework increases the reusability, changeability, extensibility and independent development ability of both syntax and semantics specification with less development workload required from compiler designers.

This dissertation elaborates the design, implementation, usage, and future directions of the overall framework. A number of case studies, including an implementation of the Java language, are provided to assess and demonstrate the benefits of using the framework in practice.

## DEDICATION

To Juan - everything is meaningless without you.

To mom and dad – your love is the debt that I can never pay off.

## ACKNOWLEDGEMENTS

To my advisor, Dr. Barrett Bryant, thank you for bringing me into the compiler world! Without your unwavering devotion, critical supervision, tireless support and enduring encouragement, I would not be able to gradually turn a small course project into a thesis during the last four years. I am grateful from the bottom of my heart that I have an advisor who can provide me unbounded opportunities and resources, yet is willing to give me independence in my academic study. In 2005, right before my leaving for a cross-country trip, you told me which route I should drive and pointed me to interesting viewpoints. I took that as a vivid reflection of your direction to my Ph. D. life. Pursing a Ph. D. degree could be a tough experience, but being your student is always joyful and memorable. Thanks, Dr. Bryant, for being such a personal role model for all of your students.

To Dr. Jeff Gray, taking your courses is one thing that I will miss for my school years. I can't remember how many of my research ideas originated from your courses. The fresh knowledge I learned there not only benefits this thesis, but even secured me job offers. When I am playing with design patterns on my daily work, when I was giving a speech on AOP in front of tens of top engineers at Google, I felt grateful that I was able to study such cool knowledge systematically in your classes. I was deeply stimulated when I received your revision of my paper for the first time, and I know every one of your students feels the same way. I will never forget those moments when I got your

emails after 2:00 am. Thank you, Dr. Gray, for your persistent academic guidance and altruistic help.

To Dr. Marjan Mernik, I always feel blessed that my research work can be directed by two experts in the field, you and Dr. Bryant. Please take my deepest gratitude for your keenness to dig into my research ideas and pointing out various improvements just in time. In my eyes, our hundreds of email discussions weigh much more than any of my published papers. Thanks so much for the European trip. That is definitely a memorable experience for both my wife and me.

To Dr. Alan Sprague and Dr. Murat Tanik, I greatly appreciate your precious time and effort in serving as my committee members. I'm grateful to your willingness to assist me in improving this work. To Stacye Fraser, Janet Sims, Kathy Baier and Fran Fabrizio, who have been so friendly and helpful during my study here, thank you for making UAB an unforgettable experience for me.

To former and present SoftCom students, Fei Cao, Faizan Javed, Jane Lin, Alex Liu, Suman Roychoudhury, Robert Tairas, Hui Wu, Jing Zhang and Wei Zhao, I enjoyed so much in our joint work and fun time together. I wish all of us can have a great future! Special thanks to Jane, Jing and Alex. I am indebted to you for taking care of my miscellaneous affairs when I was not at UAB. I can't imagine what I would do if I didn't have friends like you.

To my friends and former supervisors at Synergex, Inc., Ken Lidster, Rosanne Brill, Roger Andrews, Sudeep Sabnis and David Barron, I appreciate your help on granting me such a real-world experience on compiler development. My intern experience at Synergex significantly impacted the contents of this thesis. My gratitude also goes to Ken

and David for mentoring my work with great patience and respectfulness, I always feel what I took away from Synergex is much more than what I contributed.

To my mother and father, Fenglian Wang and Shouxin Wu, thank you for bringing me into this world. I would not come to this country if I knew my leaving will cause you such a pain today. You devoted all of your life to your two boys and both of them left you for another country. Mom and Dad, thank you for giving me intelligence, a unique character and an enviable family that I can always be proud of. No matter what it takes, I will stay at your side.

To my brother, Xiaqing Wu, life is much easier for me with someone as you to follow. No one takes advantage of having a big brother like I do. Thanks for being my big brother. With you, I never feel lonely on my life journey.

Last, to my wife, Juan Lou, I am always grateful to God that I could meet you on that day. Thanks for abandoning what you had and joining me here in 2004, otherwise it will be a totally different story. Thanks for enlightening my life with your love, happiness and elegant food. I never know the life journey can be so colorful and full of joys. Thanks for coming to this world, so that I know why I am here.

Thank you all who have truly believed in me.

## TABLE OF CONTENTS

	<i>Page</i>
ABSTRACT.....	iii
DEDICATION.....	v
ACKNOWLEDGMENTS.....	vi
LIST OF TABLES.....	xii
LIST OF FIGURES.....	xiii
LIST OF ABBREVIATIONS.....	xv
CHAPTER	
1 INTRODUCTION.....	1
1.1 Modularity in Language Implementation.....	2
1.1.1 Decomposition of Language Definition.....	3
1.1.2 Proper Separation of Compiler Construction Phases.....	5
1.2 Research Objectives.....	7
1.3 Organization.....	9
2 BACKGROUND.....	11
2.1 General Context in Language Implementation.....	11
2.1.1 Syntax and Semantics.....	11
2.1.2 Context-Free Grammar, BNF and EBNF.....	12
2.1.3 LL, LR and GLR parsing.....	12
2.1.4 Parser Generators/Compiler Compilers.....	14
2.1.5 Syntax Tree Construction.....	15
2.1.6 Attribute Grammar.....	15
2.2 Related Technologies.....	16
2.2.1 Component-Based Software Engineering.....	16
2.2.2 Object-Oriented Programming.....	17
2.2.3 Aspect-Oriented Programming.....	18
2.3 Survey and Assessment on Related Work.....	18

TABLE OF CONTENTS (Continued)

Page

CHAPTER

2.3.1	Decomposition of Language Syntax .....	19
2.3.2	Separation of Concerns between Syntax and Semantics .....	21
2.3.2.1	Syntax and Semantics in Parser Generators.....	21
2.3.2.2	Syntax and Semantics in Formal Specifications.....	22
2.3.3	Separation of Concerns in Semantics .....	23
2.3.3.1	The Visitor Pattern.....	23
2.3.3.2	Aspect-Oriented Semantics.....	25
2.3.4	Summary .....	26
3	COMPONENT-BASED LR PARSING.....	28
3.1	Component-based Context Free Grammar and LR Parsing .....	30
3.1.1	Component-based Context Free Grammar .....	30
3.1.2	Component-based LR parsing.....	33
3.2	Implementation of CLR Parsing .....	41
3.2.1	Implementation Challenges .....	43
3.2.2	Implementation Strategy .....	44
3.3	Case study of CLR Parsing.....	48
3.4	Evaluation of CLR Parsing.....	51
3.4.1	Software Engineering Benefits .....	51
3.4.2	Language Description Ability.....	53
3.4.3	Performance Measurement .....	57
3.4.4	Discussion.....	60
3.5	Modular Grammar vs. Compositional Parser .....	62
3.6	Summary .....	65
4	OBJECT-ORIENTED SYNTAX AND ASPECT-ORIENTED SEMANTICS....	66
4.1	Framework Architecture .....	67
4.2	Object-Oriented Syntax .....	69
4.2.1	OOCFG and Concrete Syntax Tree Generation.....	69
4.2.2	Macro Definitions .....	74
4.2.3	Templates.....	78
4.2.4	Abstract Syntax Tree Generation.....	81
4.3	Aspect-Oriented Semantics.....	86
4.3.1	Aspect-Oriented Semantics Development in AspectJ .....	87
4.3.1.1	Inter-Type Declarations .....	88
4.3.1.2	Pointcut-Advice Model.....	90
4.3.1.3	Aspect Fields and Methods.....	94
4.3.1.4	Aspect Inheritance .....	94
4.3.2	Object-Oriented Visitor Pattern vs. Aspect-Oriented Semantics...95	

TABLE OF CONTENTS (Continued)

*Page*

CHAPTER

4.4	Case Study .....	96
4.4.1	Syntax Analysis Phase .....	96
4.4.2	Name Analysis Phase.....	99
4.4.3	Pretty Print Phase.....	102
4.5	Integration with CLR Parsing.....	104
4.6	Summary .....	106
5	FRAMEWORK APPLICATIONS .....	107
5.1	Pam and BasicM .....	107
5.2	RelationJava.....	109
5.3	OOCFG Converter .....	112
5.4	Bootstrap Implementation.....	115
5.5	Google Query Language.....	115
5.6	Robot Language .....	119
6	FUTURE WORK.....	123
6.1	CLR Backtracking .....	123
6.2	Module Inclusion .....	124
6.3	Grammar Aspects.....	125
6.4	Support of Other Parsing Paradigms.....	125
6.5	Rich Client Platform based on Eclipse .....	126
7	CONCLUSION.....	129
	LIST OF REFERENCES.....	132
APPENDIX		
A	THE SPECIFICATION OF THE JAVA BINARY EXPRESSION COMPONENT .....	139
B	THE PRETTY PRINT ASPECT OF THE JAVA LANGUAGE IMPLEMENTATION.....	142
C	THE SYNTAX GRAMMAR OF OOCFG PARSER GENERATOR .....	145

## LIST OF TABLES

<i>Table</i>		<i>Page</i>
1	OOCFG nonterminal production patterns.....	72
2	OOCFG macro production patterns .....	74
3	Comparison between abstract syntax tree and concrete syntax tree .....	82
4	OOCFG specification features and their usage.....	86
5	Analysis data of JLS grammars .....	97
6	Specific features in each advanced Google search domain .....	116

## LIST OF FIGURES

<i>Figure</i>		<i>Page</i>
1	The Visitor pattern of an expression language .....	25
2	Illustration of production grouping in the JLS grammar .....	31
3	Flowchart of component-based LR parsing.....	36
4	Pseudo code of component parsing.....	37
5	Sample action sequence and stack evolution in LR parsing .....	39
6	A parse tree instance of Language X.....	40
7	The CLR specification for languages X, Y and Z .....	42
8	The implementation detail of the switch and return actions.....	46
9	CLR components of JLS (11 component version).....	49
10	Parsing illustration of a sample Java program using the CLR JLS implementation (4 component version). .....	50
11	CLR specification for the island grammar example .....	55
12	Parsing speed comparison among 11 versions of CLR implementation of JLS....	58
13	The number of external actions used to parse Sample4.....	59
14	Difference between modular parser generation and compositional parser genera- tion .....	63
15	The syntax and semantics implementation framework.....	68
16	Commonly used templates definitions.....	79

17	Class diagram for ASTNode.....	83
18	The generated node class for IfStmt .....	84
19	AST class hierarchy for statement related productions .....	85
20	Translation from regular Java class members to AspectJ Inter-type declarations.....	88
21	The Dfvisit algorithm and its AspectJ implementation .....	92
22	A sample aspect to trace the AST construction process .....	94
23	Operation redirection in the Visitor pattern vs. aspect weaving in AOP implementation .....	96
24	AspectJ code for symbol table loading phase.....	101
25	The integrated framework overview.....	105
26	Output of a sample BasicM program.....	108
27	The UnparseVisitor class .....	110
28	The Unparse aspect.....	111
29	Nonterminal and macro definitions of the OOCFG converter.....	112
30	The translation aspect in OOCFG converter.....	114
31	Syntax specification for WebQL .....	117
32	Syntax specification for GQL .....	117
33	Syntax specification for recursive GQL .....	118
34	The syntax of the Robot language .....	120
35	The graphic output of the robot language.....	122

## LIST OF ABBREVIATIONS

ANTLR	ANother Tool for Language Recognition
AOP	Aspect-Oriented Programming
AOS	Aspect-Oriented Semantics
ASF	Algebraic Specification Formalism
AST	Abstract Syntax Tree
BFS	Breadth First Search
BNF	Backus-Naur Form
CBLD	Component-Based Language Development
CBSE	Component-Based Software Engineering
CCFG	Component-based Context Free Grammar
CCFL	Component-based Context-Free Language
CFG	Context-Free Grammar
CLR	Component-based LR parsing
CST	Concrete Syntax Tree
DFA	Deterministic Finite Automata
DFS	Depth First Search
DSL	Domain-Specific Language
EBNF	Extended BNF
GLR	Generalized LR

GQL	Google Query Language
IDE	Integrated Development Environment
ITD	Inter-Type Declarations
JLS	Java Language Specification
JTB	Java Tree Builder
LALR	Look-Ahead LR
LHS	Left-Hand Side
OOCFG	Object-Oriented Context-Free Grammar
OOP	Object-Oriented Programming
OOS	Object-Oriented Syntax
PDA	Push Down Automata
PDE	Plug-in Development Environment
RHS	Right-Hand Side
SDF	Syntax Definition Formalism
SoC	Separation of Concerns
TXL	Turing eXtender Language
YACC	Yet Another Compiler-Compiler

## CHAPTER 1

### INTRODUCTION

*I have to respect the strictly limited size of my head and can deal with only one thing at a time.*

Edsger Dijkstra [Dijkstra 1988]

Everything in the universe is composed of smaller components, so is a software system. One of the main software engineering endeavors is to decompose a large system into primitive entities that can be specified and implemented separately. This not only helps a developer to divide-and-conquer the development complexity, but also reduces the cost of maintenance and promotes the reuse of existing constructs [Gauthier 1970, Parnas 1972].

However, not everything can be easily broken into pieces. Providing proper modularity is one of the major challenges in software development. In the context of programming language development, two key factors impede the modularization process in a compiler system: First, the parser, as the core of the language implementation, usually is produced by a parser generator in a non-decomposable style, which means it must be constructed as a whole regardless of the complexity of the grammar. This requirement further enforces that later phases must be built in the same fashion. Second, the interconnected nature of various semantic stages within the compiler makes it difficult to separate each phase clearly. As a result, the construction of multiple phases is often tangled together, which makes a system hard to develop, maintain and extend.

These two characteristics conceive the modularization complication of language development, which create the complexity in compiler design known as a “dragon” task [Aho 2007].

This dissertation describes a framework that can address the modularity problem in language implementations in a two dimensional manner (structure wise and function wise). The framework is composed of Component-Based Language Development (CBLD), Object-Oriented Syntax (OOS) and Aspect-Oriented Semantics (AOS), each of which will be developed fully in this thesis. CBLD decreases the development complexity by reducing the granularity of a language, whereas OOS and AOS facilitate Separation of Concerns (SoC) [Tarr 1999] by isolating syntax and semantics as well as semantic phases into different modules.

In the following sections of this chapter, first the modularity problems in language implementation will be discussed, followed by the description of the research objectives and contributions of this thesis. The general organization of the remaining chapters is outlined in Section 1.3.

## **1.1 Modularity in Language Implementation**

Overall, the first modularity problem of language implementation resides at the structure level. Due to the non-decomposable nature of the parser implementation, a language always has to be employed as a whole, which might be too complex to handle if the language has a complicated syntax. The second modularity problem is the lack of separation between syntax and semantics as well as among semantic phases, which reveals the poor modularity of language implementation at the function level.

### 1.1.1 Decomposition of Language Definition

In most parser generators such as YACC (Yet Another Compiler-Compiler) [Johnson 1975] and its derivatives, there is little compositionality provided at the syntax level. Since large grammars may run into several hundred or even thousands of productions (e.g., the GOLD grammar [Cook 2006] of Cobol 85 is 2500 lines<sup>1</sup>), compiler developers are forced to put a significant amount of production rules inside one module and develop the parser as a whole. This usually yields poor comprehensibility, reduced changeability, limited reusability, and hampers independent development of the language implementation, described as follows:

- **Comprehensibility.** Clearly, too many lines of productions intertwined together may result in poor readability. The lack of modularity means all variables have to share a single namespace, which leads to the generation of very verbose variable names. For instance, the IBM VS-COBOL grammar<sup>2</sup> contains 1028 variable names [Johnstone 2004a]. Identifying what each symbol stands for in a global scope is difficult.
- **Changeability and Reusability.** Change to any variable name may propagate errors to hundreds of lines and changes to any production may generate numerous conflicts with other productions across the specification file. On the other hand, the language specification is difficult to reuse for embedded languages and languages with many variants or extensions. For example, COBOL has extensions for embedded CICS, SQL or DB2. To change from one dialect to another, the whole grammar has to be rewritten [van den Brand 1998].

---

<sup>1</sup> Available at <http://www.devincook.com/GOLDParser/grammars/index.htm>

<sup>2</sup> Available at <http://www.cs.vu.nl/grammars/browsable/vs-cobol-ii/>

- Independent development. Since all the productions reside in one module, the grammar initially has to be written all together. Tangled relations between productions make it impossible for multiple developers to concurrently edit the same grammar, resulting in a lengthened development cycle for parser implementation. Because other parts of the language development depend heavily on the parsing results, the delay of the parser implementation impedes the overall development progress. In our previous development experience of extending a legacy language, it took a developer over six months to design a grammar with 600 nonterminals and implement the parser and tree structure, significantly deferring the progress of five other engineers working on semantic analysis and the editing environment.

The problems caused by modularity can become more severe when tree construction and semantic analysis are involved. As a result of the poor modularity of the syntax grammar, parse tree construction will be badly modularized as well. If a semantic phase is implemented using the Visitor pattern [Gamma 1995], a single visitor class may contain a large number of visiting methods for all node classes (e.g., a JavaCC [JavaCC 2006] + JJTree [JJTree 2006] sample program contains 85 visiting methods in a single unparse visitor<sup>3</sup>).

---

<sup>3</sup> Available at <https://javacc.dev.java.net/source/browse/javacc/examples/VTransformer/UnparseVisitor.java>.

### 1.1.2 Proper Separation of Compiler Construction Phases

*Life is a very complicated business if you want to do it well. This is because anything of any importance is always a many-sided affair and none of its different aspects may be neglected, while at the same time, in order to do the whole job well, the different concerns have to be separated as ruthlessly as possible.*

Edsger Dijkstra [Dijkstra 1988]

It is well-known that compiler development is a multi-stage process (e.g., syntax analysis, tree construction, static checking, and code generation) [Aho 2007]. Each phase has a clear goal and some of them often require an independent traversal of the syntax tree. Language implementation using traditional tools such as YACC [Johnson 1975] and its derivatives tangle these different phases of the implementation together as one module. There are few mechanisms available to isolate each phase cleanly, which contributes to the difficulty in hiding the data and methods that are only relevant to specific phases; as such, the phases of a compiler implementation are often tightly coupled. The compiler writer is forced to consider all phases simultaneously and the construction of one phase always “pollutes” a different phase, which makes a system hard to develop, maintain and extend.

The lack of SoC in language implementation is mainly shown in the following two aspects: 1) between syntax and semantics and 2) among various semantic phases themselves, with each detailed as below:

- In most YACC-like compiler generators, language syntax is described by a formal specification, and semantics is implemented by a general purpose programming language. Normally, semantics can be attached to each grammar rule in the parser specification as action codes. These action codes are copied into the generated parser and are invoked when a production is reduced or derived. However, the

mixing of grammar and code fragments generally yields a software engineering problem [van den Brand 1997]. First of all, the mixed specification usually results in a large file and the juxtaposition of formal specification and programming language code is hard to read. Moreover, since the generated code contains user-supplied code, debugging a program becomes quite difficult. Considering debugging an error in the action code – a programmer has to make a change in the specification file, regenerate the whole parser, locate the error in the generated parser and map it back to the action code. This process is quite cumbersome and the parser generation is unnecessary because the syntax grammar is actually not changed at all. Taking a shortcut by modifying the generated parser directly will make the code unsynchronized with the specification and is considered an error-prone practice [Gagon 1998].

- Except for the action codes directly inserted in the parser specification, most semantic analysis is implemented by traversing a syntax tree built during parsing the program. In classical object-oriented development, syntax tree nodes are usually defined as classes with various semantic operations (e.g., type-checking, symbol table loading, pretty printing, code generation) embedded as methods. Consequently, syntax tree traversal is achieved by recursively executing those semantic operations. However, an inherent problem in this approach is that each kind of semantic operation (defined as a method within each node class) crosscuts the various node class boundaries, thereby leading to a system that is hard to comprehend, maintain and extend. For example, mixing the logic for pretty print and code generation inside a single class will be confusing. Moreover, adding a

new operation requires an invasive change throughout the existing node class hierarchy. The problem reflects the drawback of object-oriented programming in modularizing crosscutting concerns. It would be better if each semantic function could be specified separately, so that the node classes were independent of the operations that apply to them. Although the Visitor pattern has provided a way to solve this problem, its unnatural implementation and restrictions make it difficult to adopt [Hachani 2002, Wu 2006], as detailed in Chapter 2.

## 1.2 Research Objectives

With an aim to address the modularity problems in language implementations without hurting the flexibility and language description ability in current development practices, this thesis presents a new language implementation formalism and framework. The goal of this new paradigm is to improve modularity in the language development process and alleviate the implementation complexity. During the development of this framework, the following criteria are used to measure this goal:

- Large languages should be developed based on small language implementations;
- Syntax specification must be physically separate with semantics;
- Semantic phases should be isolated from each other;
- Declarative formal specification should be separated with imperative programming language code;
- Hand-written code must be separated from auto-generated code.

In order to meet the above criteria, the framework utilizes component-based development to enable decomposition of a large language implementation into

manageable language components; based on the different characteristics of syntax and semantics, object-oriented formal specification and an aspect-oriented programming language are incorporated to describe syntax and semantics, respectively. A list of the specific techniques and features employed in this framework is provided below:

- Component-based development decreases the workload of developing a large language into developing a number of smaller languages, because each decomposed component can be implemented and executed independently;
- Object-oriented context free grammar automates the parser and tree construction processes directly from syntax definitions, eliminating the redundancy between syntax specification and tree structure description;
- Macros and templates used in syntax specification provide utilities to resolve parsing conflicts and specify syntax precisely, while at the same time keeping the tree structure comprehensible;
- Aspect-oriented semantic implementation built on top of an object-oriented syntax tree clearly encapsulates each phase of semantic analysis as an aspect and offers the flexibility in semantic pattern description, phase composition and tree traversal.

In summary, this framework is distinguished from all previous language implementation practices because it embeds useful constructs and techniques from software engineering and programming languages, such as component-based development, object-orientation, aspect-orientation, design patterns, namespace, macros and templates. Throughout the framework, the abstraction power of formal specifications is exploited and the flexibility provided by general purpose programming languages is

utilized. The overall paradigm will increase the reusability, changeability, extensibility and independent development ability of the syntax and semantics with less development workload required from compiler designers.

### **1.3 Organization**

Chapter 2 provides background context related to this thesis. It begins with introducing some related knowledge in language implementation and then briefly describes the key techniques utilized in the framework: component-based development, object-oriented programming and aspect-oriented programming. A survey of related literature can be found in Section 2.3, where various techniques used to support better modularization in language development are examined. These techniques are either targeted to add modularity into the syntax grammar or aimed at providing better separation of concerns.

Chapter 3 and Chapter 4 present the whole framework and are the core parts of this thesis. Chapter 3 introduces Component-based LR parsing (CLR), which is a novel parsing algorithm that supports CBLD. Object-oriented syntax and aspect-oriented semantics are explored in Chapter 4 to illustrate how the SoC problem is addressed in the framework. A case study on the Java language implementation is utilized throughout Chapter 3 and Chapter 4 to evaluate this framework.

A detailed outline of future extensions to this work can be found in Chapter 5. Most of the extending areas are focused on enhancements to the CLR parsing and providing flexibility of the overall language description.

Concluding remarks appear in Chapter 6, prefixed by additional case studies that further assess the benefits of using the described paradigm. An appendix is included at the end of the thesis, followed by a comprehensive bibliography.

## CHAPTER 2

### BACKGROUND

This chapter begins by exploring some general knowledge related to language implementation, followed by an introduction of the various programming language and software engineering techniques used in this dissertation. A broad survey of prior efforts towards language development modularization is encompassed afterwards. These background knowledge, techniques, and ideas form the basis of this dissertation.

#### **2.1 General Context in Language Implementation**

##### **2.1.1 Syntax and Semantics**

Unlike natural languages, programming languages are strictly stylized entities created to facilitate human communication with computers. In order to make programming languages recognizable by computers, the key object is to describe and implement language syntax and semantics such that a compiler could be built to generate machine-readable code. The syntax of a programming language is the representation of its programmable entities such as expressions, declarations and commands. The semantics is the actual meaning of the syntax entities. In a compiler implementation, syntax determines the structure of the system while semantics describes the behaviors.

Since the 1960s [Sebesta 2006], intensive research efforts have been made to formalize the language implementation process. Great success has been made in the

syntax analysis domain. As a result, Context-Free Grammars (CFGs) are widely used to describe the syntax of programming languages and generate parsers automatically [Slonneger 1995]. However, there is no universally accepted formal method for semantic description [Sebesta 2006], due to the fact that the semantics of programming languages are quite diverse, and it is difficult to invent a simple notation to satisfy all the computation needs of various kinds of programming languages.

### **2.1.2 Context-Free Grammar, BNF and EBNF**

In the 1950s, Noam Chomsky invented four levels of grammars to formally describe different kinds of languages [Chomsky 1959]. These grammars, from Type-0 to Type-3, are rated by their expressive power in decreasing order, which is known as the *Chomsky hierarchy*. The two weaker grammar types (i.e., regular grammars, Type-3; and context-free grammars, Type-2) are well-suited to describe the lexemes (i.e., the atomic-level syntactic units) and the syntactic grammar of programming languages, respectively. Backus-Naur Form (BNF) [Naur 1960] was introduced shortly after the Chomsky hierarchy. BNF has the same expressive power as context-free grammar and it was first used in describing ALGOL 60 [Naur 1960]. BNF has an extended version called Extended BNF, or simply EBNF, where a set of operators are added to facilitate expressing production rules.

### **2.1.3 LL, LR and GLR parsing**

Based on the study of CFG and BNF, a number of parsing algorithms have been developed. The two main categories among them are called top-down parsing and

bottom-up parsing. Top-down parsing recursively expands a non-terminal (initially the start symbol) according to its corresponding productions and matches the expanded sentences against the input program. Because it parses the input from **L**eft to right, and constructs a **L**eftmost derivation of the program [Aho 2007], a top-down parser is also called an LL parser. A typical implementation of an LL parser is to use recursive descent function calls for each expansion, which are easy to develop by hand. Bottom-up parsing, on the other hand, identifies terminal symbols from the input stream first, and combines them successively to reduce to non-terminals. Bottom up parsing also parses the input from **L**eft to right, but it constructs a **R**ightmost derivation of the program [Aho 2007]. Therefore, a bottom-up parser is also called an LR parser [Knuth 1965]. LR parsers are typically implemented by a pushdown automaton with actions to shift (i.e., push an input token into the stack) or reduce (i.e., replace a production right-hand side at the top of the stack by the nonterminal which derives it), which are difficult to code by hand. The table size of a canonical LR parser is generally considered too large to use in practice. Consequently, an optimized form of it, the LALR (Look Ahead LR) parser is widely used instead, which significantly reduces the table size [Deremer 1969, Aho 2007].

The grammars recognized by LL and LR parsers are called LL and LR grammars, respectively. They are both subsets of context-free grammars. LL grammars cannot have left-recursive references (a non-terminal has a derivation with itself as the leftmost symbol) and LR grammars cannot contain action conflicts (i.e., shift-reduce conflicts, reduce-reduce conflicts). Any LL grammar can be rewritten as an LR grammar, but not vice versa. Both LL and LR parsers can be extended by using  $k$  tokens of lookahead. The associated parsers are called LL( $k$ ) parsers and LR( $k$ ) parsers, respectively. Lookahead

can eliminate most of the action conflicts existing in an LR grammar, unless the grammar contains ambiguity (i.e., the same program can be parsed in different ways). To resolve action conflicts in a generic way, an extension of the LR parsing algorithm, called GLR (Generalized LR) parsing [Tomita 1986], has been invented to handle any context-free grammar, including ambiguous ones. The basic strategy of the algorithm is that, once a conflict occurs, the GLR parser will process all of the available actions in parallel. Hence, GLR parsers are also named parallel parsers. Due to its breadth-first search nature, the GLR parsing suffers from its time and space complexity. In general, GLR complexity is  $O(n^{p+1})$ , where  $p$  is the length of the longest right-hand side. Various attempts [Kipps 1991, Aycock 2001, McPeak 2004] have been made to optimize its performance. For example, in [Kipps 1991], it has been proved that GLR's time complexity can be optimized to  $O(n^3)$  by converting the grammar into Chomsky normal form, but the optimization comes with a cost of destroying the conceptual structure [McPeak 2004]. Currently GLR is still not widely used in programming language implementation, but its popularity is growing [Johnstone 2004a].

#### **2.1.4 Parser Generators/Compiler Compilers**

Language implementation is generally considered as one of the most appropriate software applications that can be implemented systematically. After context-free grammar and its variants (BNF, EBNF) were found well suited for formally defining syntax [Wexelblat 1981], quite a number of compiler/parser generator tools have been developed (e.g., YACC, Bison [Donnelly 1995], ANTLR (ANother Tool for Language Recognition) [Parr 2007], SableCC [Gagon 1998], CUP [CUP 1999], LISA [Mernik

2005b], and JavaCC [JavaCC 2006]) to free language designers from handcrafting a compiler from scratch<sup>4</sup>. These tools are generally referred to as parser generators or compiler-compilers [Aho 2007].

### 2.1.5 Syntax Tree Construction

In language implementation, after an input program is parsed, an intermediate result called a syntax tree is normally produced. Semantic analysis consults this tree structure to retrieve the information encompassed by the program and delivers the corresponding actions. Therefore, the syntax tree serves as the middle layer between syntax and semantics that links them together. There are two ways to build the syntax tree structure: construction by hand and automatic generation from tools. The first method requires the language developer to manually write tree node classes and embed code across productions to link the nodes together, which is quite tedious and error-prone. The second method works on a more abstract level where tree structure and construction logic are automatically generated from formal specifications provided by the developer. Some sample tree generation systems are introduced in Section 2.3.

### 2.1.6 Attribute Grammar

Attribute grammars [Knuth 1968, Paakki 1995] were invented to specify the static semantics of programming languages (those properties that can be acquired from the source program directly) as an extension of context-free grammars. Each symbol has an associated set of attributes that carry semantic information, and each production has a set of semantic rules associated with attribute computation. The attributes are divided into

---

<sup>4</sup> A longer list is available at [http://en.wikipedia.org/wiki/List\\_of\\_compiler-compilers](http://en.wikipedia.org/wiki/List_of_compiler-compilers).

two groups: synthesized attributes and inherited attributes. The synthesized attributes are used to pass the semantic information up the syntax tree and the inherited attributes are passed down from parent nodes. An evaluation scheme walks a tree one or more times to compute all the attributes.

Attribute grammar is normally considered not powerful enough to express dynamic semantics, which are those properties that “reflect the history of program execution or user interactions with the programming environment” [Kaiser 1989].

Some compiler generators such as LISA and JastAdd [Hedin 2001] use attribute grammar as a formalism to describe both the syntax and semantics of a language.

## **2.2 Related Technologies**

### **2.2.1 Component-Based Software Engineering**

Component-Based Software Engineering (CBSE) [Heineman 2001] is a discipline leading software engineering into a new generation. The goal of CBSE is to build a software system by assembling small components in the same manner as hardware composition. Based on the achievement of object-orientation, CBSE offers a promising way to promote software reuse and decrease the complexity of the development process, via proper information encapsulation and separation of concerns at various levels [Cao 2005]. Although various definitions of “component” are available, a software component is generally considered to have the following properties:

- **Information hiding.** A component should totally hide its implementation from the party who uses it. In other words, components are used as black boxes.

- **Explicit interface.** A component must have a well-defined interface such that component users know how to compose it.
- **Context Independency.** A component should be independent of other components, which guarantees that change to any other components won't break the component.

The basic unit in CBSE generally has a large granularity. Therefore, CBSE enhances the modularity of a software system at a much higher level.

### 2.2.2 Object-Oriented Programming

Object-Oriented Programming (OOP) [Wikipedia 2006a] is an important programming paradigm that was born in the 1960s. The design scheme behind OOP is that a program is composed of objects. The basic type in OOP is called a class and objects are instances of classes. OOP's core features include encapsulation, inheritance and polymorphism: encapsulation is a language feature that accumulates methods and fields into classes, whose accessibility is controlled by modifiers; inheritance enables a class to reuse the interface or implementation of other classes; polymorphism makes possible the same operation applying on different types of data. All these language features provide high quality information hiding and reusability for software development. Consequently, programming in OOP significantly improves the modularity of software implementation by encapsulating related data and behaviors together.

### **2.2.3 Aspect-Oriented Programming**

Aspect-Oriented Programming (AOP) [Kiczales 1997] provides special language constructs called aspects that modularize crosscutting concerns in conventional program structures (e.g., a concern that is spread across class hierarchies of object-oriented programs). AOP offers two new types of functionalities to conventional programming languages, namely, introduction to existing data structure and insertion of group behaviors (i.e., join point model). The first one is implemented by Inter-Type Declarations (ITD), which can add new class members to an existing class or alter the class inheritance hierarchy without any modification in the source code level. For the second one, a language element called advice is used to represent the crosscutting behavior, and a pattern language (e.g., pointcut expressions [Kiczales 2001]) is used to specify the locations in the base program where the advice should be applied. In AOP, a translator called a weaver is responsible for merging the additional code specified in an aspect language into the base language at compile time.

In addition to the object-oriented dimension, AOP in fact provides a mechanism to support modularity in a second extension. A general aspect-oriented language for supporting separation of crosscutting concerns is AspectJ [Kiczales 2001, Colyer 2004], which is an extension of Java.

## **2.3 Survey and Assessment on Related Work**

In this section, the modularity aspects of current language implementation methods will be evaluated from three categories: the decomposition of the language

syntax, the separation between syntax and semantics and the separation among semantic phases.

### **2.3.1 Decomposition of Language Syntax**

Modularity in syntax analysis is not a new problem. There have been a number of attempts to provide grammars with modular constructs, which are supported by parser generators based on various parsing algorithms.

The first category of related LR parser generators is represented by LISA [Mernik 2005] and PPG [Nystrom 2003]. Both tools allow a new grammar module to use different operators to override, inherit and extend productions from an existing module. However, despite their flexibility in incremental development, these grammar modules cannot be composed directly. The conflicts across modules have to be resolved manually either by modifying the grammar or assigning priority to tokens [van den Brand 1998]. Another LR parser generator that supports modular development is BtYacc (Backtracking Yacc [BtYacc 2006]). It uses backtracking to resolve shift-reduce or reduce-reduce conflicts. Once a conflict is encountered, BtYacc will try all possible paths until one of them succeeds. Therefore, BtYacc's modules are compositional. On the other hand, SDF [Visser 1997] and the Design Maintenance System (DMS) [Baxter 2004] use GLR parsing, where grammars are allowed to contain LR conflicts. When shift-reduce or reduce-reduce conflicts occur, a GLR parser forks the parse stack and tries all the possible paths in parallel. Consequently, their modules can be freely composed, provided that the same nonterminal does not coexist in multiple modules. To some extent, the backtracking technology can be treated as a Depth First Search (DFS) solution and GLR

can be treated as Breadth First Search (BFS). DFS finds the first successful path and returns, but BFS will retrieve all the successful ones. Therefore, GLR is well-known for handling ambiguous grammars, but backtracking does not serve that purpose. Due to its BFS implementation nature and the overhead of maintaining a graph-structured stack, SDF parsers are typically slower by a factor of ten or more than their LALR counterparts on non-ambiguous input [McPeak 2004].

Beyond the LR grammar family, there are also other parser generators and transformation systems that support modular language development, including ANTLR and TXL (Turing eXtender Language) [Cordy 2004], which are based on LL(k) parsing, and *Rats!* [Grimm 2006], which is built on recent research on parsing expression grammars (PEGs) [Ford 2004]. ANTLR has to manually resolve the possible ambiguities generated from composition using predicates, but TXL and *Rats!* do not need to resolve any ambiguity because their production rules are inherently ordered.

However, merely enabling textual modularity at the grammar level is not strong enough to address the problems mentioned in Chapter 1. The nature of module inclusion of these tools is pure text copying and the main goal is to support incremental language development [Mernik 2005b], rather than decompose the development complexity. For each language, the parser is still implemented as a singleton that forces the remaining phases to follow the same track (detailed in Chapter 3). Despite the fact that modularity was addressed in these practices at various levels, the state of the automated parsing has changed little [McPeak 2004]. To develop a programming language, YACC and its variants are still the first choice of many compiler developers; hence, the development will employ no compositionality at all.

### 2.3.2 Separation of Concerns between Syntax and Semantics

Current popular practice and research effort towards language implementation can be categorized into two groups by their semantic implementation strategy. The first category includes most practical parser generators, where semantics are implemented directly by imperative programming languages. The second category, on the other hand, relies on formal specification on both syntax and semantic descriptions, which are mostly used in research works. In the following sections, the modularization characteristics between syntax and semantics of language implementation strategies will be explored within these two categories of parser generators.

#### 2.3.2.1 *Syntax and Semantics in Parser Generators*

The first generation of these tools is represented by YACC, where syntax specifications and semantic actions (including tree construction) are juxtaposed in the specification file. YACC was developed during the time when procedural languages were the dominant paradigm, so the modularity in the programming language level is very limited. As described in Chapter 1, the tangling of formal syntax rules specification with informal semantic rules written in programming language reduces the readability, and makes the programming complicated by shifting between different programming paradigms.

The second generation of tools includes LL(k) parser generators JavaCC + JJTree, ANTLR, and the LALR(1) generator SableCC. They move one step forward by allowing automatic syntax tree generations. All these tools are based on object-oriented programming languages, where syntax tree nodes are defined as classes that can

encapsulate tree links and semantic operations. Consequently, a developer may delegate most semantic analysis tasks to the programming entities defined outside the grammar specification, which provides better separation between syntax and semantic analysis. As inherited from YACC, JavaCC and ANTLR still allow embedding semantic actions inside the grammar productions. On the contrary, SableCC specification files do not contain any action code, which fully separates syntax and semantics definitions. In this case, the semantic analysis is completely specified as tree traversing logic afterwards. The syntax tree serves as the sole middle layer that links syntax and semantic analysis together.

#### *2.3.2.2 Syntax and Semantics in Formal Specifications*

Formal specification based compiler-compilers include most rewrite rule systems and attribute grammar based applications. In contrast to other parser generators, these systems describe both the syntax and semantics in a declarative manner by different formalisms, which enable the semantic part to be separated from syntax definitions.

A well-known framework in this category is ASF+SDF [van den Brand 2002], which combines the syntax definition formalism SDF (Syntax Definition Formalism) with the term rewriting language ASF (Algebraic Specification Formalism). SDF is supported with GLR parsing technology. ASF is a rather pure executable specification language that allows rewrite rules to be written in concrete syntax. However, although ASF is good for the prototyping of language processing systems, the mathematical based SDF lacks some features to build mature semantic implementations. For instance, ASF

does not come with a strong library mechanism, I/O capabilities, or support for generic term traversal [Kuipers 2001].

Another sample framework is called JastAdd [Hedin 2001], which is built on top of JavaCC and adds facilities for specifying and generating object-oriented syntax trees. In contrast to ASF+SDF, JastAdd allows the use of both declarative behavior (e.g., Rewritable Reference Attributed Grammars – ReRAGs [Ekman 2004]) as well as imperative behavior (ordinary Java code), which are all woven together into syntax tree classes. However, based on the fact that JastAdd totally relies on outside parser generators for syntax specification and tree building, and the different types of semantic specifications it requires, there are five kinds of description files that need to be hand-written during language development. As all these files are coupled with syntax tree classes, significant consistency checks are required for language developers.

### **2.3.3 Separation of Concerns in Semantics**

In Chapter 1, it has been discussed that directly adding semantic operations into tree node classes results in a system that is hard to maintain and extend. Some recent efforts targeted to address this problem are listed in the next section, which includes the Visitor design pattern and some endeavors based on aspect-orientation.

#### *2.3.3.1 The Visitor Pattern*

The Visitor pattern [Gamma 1995] is an object-oriented pattern used to separate functional operations with object structure. Most practical parser generators such as JavaCC and SableCC support generating visitor interfaces along with the tree node

classes. The benefit of using this pattern is that each tree traversal phase is isolated as a class, which is independent of other node classes and can be freely modified or extended.

In applying this pattern, all the methods pertaining to one semantic pass are encapsulated inside a visitor class. Each node class has a general *accept* method associated with it, which can redirect a semantic evaluation request to the appropriate method in the provided visitor class. This mechanism is called double-dispatch. Figure 1 provides an illustration of the Visitor pattern used in implementing an expression language.

However, since object-orientation describes a system by a collection of objects rather than a collection of operations, it is clear that object-orientation is not a natural specification of programs based on the Visitor pattern. The complicated implementation of this design pattern introduces a lot of extra code in the element classes and makes the code hard to understand and maintain [Hachani 2002, Wu 2006]. Particularly, if a Visitor pattern has not been incorporated into the code from the beginning, the whole syntax tree hierarchy has to be modified to implement it. To support the double-dispatch mechanism, the abstract visitor forces the return types, number of parameters and parameter types of various visiting methods of a certain node to be the same. This is a very inflexible limitation, because different semantic operations have different computing needs. This tends to make the programs difficult to understand and introduces dependencies that can impede evolution of the compiler.

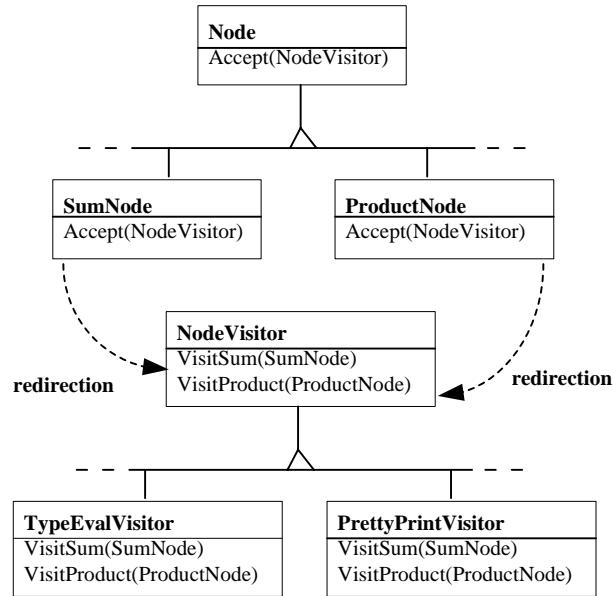


Figure 1: The Visitor pattern of an expression language

### 2.3.3.2 Aspect-Oriented Semantics

Current research in AOP and design patterns has indicated that the visitor has basic AOP characteristics: without it the structure and behavior characteristics are scattered throughout the code base instead of being isolated in single separate classes [Gamma 1995, Hachani 2002, Hannemann 2002].

Applying AOP concepts to compiler design was first proposed by de Moor et al. [de Moor 2000], at a time when AOP tool support was not mature and when various definitions of aspects existed. Essentially, they introduced an aspect-oriented implementation of an attribute grammar, where each aspect represented a semantic attribute (e.g., the environment). Their notion of aspect is highly restrictive and it is a slight deviation from the general notion and terminology of aspects as described in this thesis.

Similarly, TreeCC [Weatherley 2006] invented special notations to facilitate compiler writers to specify semantic computation outside of a syntax tree structure. These aspects are translated by a preprocessor into C functions with those tree nodes as parameters. Therefore, the advantages of object-orientation are not utilized in the framework.

In JastAdd II [Hedin 2003], static aspect-oriented programming technology was utilized to assist some of the computation of attributes, which can add features to the syntax tree classes in a way analogous to the use of inter-type declarations in AspectJ that will be described in this thesis. However, JastAdd II does not support any dynamic aspects that can be easily specified by AspectJ.

#### **2.3.4 Summary**

To summarize the above survey, although the modularization problem has been addressed at different levels by various compiler generation frameworks, there are no complete and thorough solutions existing to attack the problem in a comprehensive way.

- For language decomposition, although various syntax development practices support modular syntax specification, the coupling among modules is still quite tight due to their textual composition nature.
- For separating syntax and semantics, except SableCC, there is no practical parser generator that offers a clear isolation between syntax specification and semantic behaviors. Although some research tools do use separated formalism modules to describe syntax and semantics, the mathematical based semantic specification is not good enough for mature semantic implementation. On the other hand,

- allowing both declarative and imperative semantic specification generates additional complexity due to their intricate communication.
- For separating semantic phases, the widely used Visitor pattern generates a number of side-effects due to the fact that object-orientation is not a perfect scheme for describing crosscutting behaviors. Although some aspect-related solutions have been provided to take over the Visitor pattern, the full power of AOP is not well utilized and there is still much room available for further extensions towards this area. Overall, none of the current compiler generation tool is targeted to handle all the three aspects of the modularization problem described in Chapter 1.

## CHAPTER 3

### COMPONENT-BASED LR PARSING

As discussed in Chapter 1, the first modularity problem of current language implementation practices resides at the lack of decomposition of language definitions. In 2.3.1, it has been pointed out that modularity at the grammar level is not strong enough to address the problem. The coupled nature of those textual modules cannot be used to decompose the development complexity.

Therefore, to avoid the coupling between different modules, the ideal solution is to decompose a large language into several smaller language components (e.g., components for expressions and statements), with each component generating its own individual parser. Because the LR grammars of the various components do not co-exist in the same parser, composition at the code level does not generate any internal conflicts. In this way, a language could be implemented simply by a plug-and-play mechanism for parser components. The language developer will not have to worry about the dependency issues among different components. This approach is common in component-based programming [Szyperski 2002] where components can be simple plug-ins.

In order to implement the above mechanism, a new parsing algorithm is introduced in this thesis, Component-based LR (CLR) parsing, which provides code-level compositionality for language development by producing a separate parser for each grammar component. In addition to shift and reduce actions, the CLR algorithm extends

general LR parsing by introducing switch and return actions to empower the parsing action to jump from one parser to another. CLR decreases the workload of developing a large language parser into developing a number of smaller language parsers, where each decomposed parser component can be implemented and tested independently. This further benefits the development of the following semantic phases. Experimental evaluation demonstrates that CLR increases the reusability, changeability and independent development ability of the language implementation. Moreover, the loose coupling among parser components enables CLR to describe grammars that contain LR parsing conflicts or require ambiguous token definitions, such as embedded languages. It is also possible to extend component-based parsing into other parsing algorithms, where a language can be implemented by multiple parsers each of which employs the most suitable parsing algorithm.

In this chapter, Component-based LR parsing is presented as a stand-alone technique without the presence of OOS and AOS. The integration with OOS and AOS will be introduced in Chapter 4. The chapter is organized as follows. Section 3.1 details the definition of Component-based context free grammar and introduces the CLR algorithm. Section 3.2 describes the difficulties and strategies of implementing the algorithm and a CLR parser generator. A case study of CLR parsing is introduced in Section 3.3, where sample CLR implementations of the Java language are presented. Section 3.4 demonstrates the evaluation of using CLR parsing and Section 3.5 discusses the difference between the compositionality employed in CLR and the modular grammar utilized in other language implementation practices. A summary of the contributions of CLR parsing is presented in Section 3.6.

### 3.1 Component-based Context Free Grammar and LR Parsing

In order to enable language implementation in a component-based manner, we have invented a new type of context free grammar called *Component-based Context Free Grammar* (CCFG), which is as expressive as conventional CFG, yet has a different representation format. A component-based LR parsing algorithm is presented to conduct parsing on top of CCFG specifications.

#### 3.1.1 Component-based Context Free Grammar

Within a grammar specification, it is common to find a group of productions that are tightly cohesive with one another. Inside the group of productions, most grammar symbols are not used by the outside productions, whereas one symbol may be constantly referred to by other groups. This symbol can be regarded as the point of entry to this group, in other words, the start grammar symbol of this sub-language. Figure 2 illustrates the grouping situation in the grammar of the Java Language Specification (JLS) [Gosling 1996]. A star (\*) in position (i, j) represents that there exists a production where the  $i^{\text{th}}$  nonterminal is the Left-Hand Side (LHS) symbol and the  $j^{\text{th}}$  nonterminal is one of the Right-Hand Side (RHS) symbols. The symbol names are attached to the indices along the right-most column. Notice that those expression-related productions form a sub-matrix, inside which each symbol is mainly referred to by symbols inside the matrix, with one exception, the nonterminal `expression`. We call these expression related productions a *grammar component* and treat the nonterminal `expression` as the start symbol of the component. The language produced from a grammar component is called a *language*

*component*. A language component can be reused to build other languages (e.g., the expression component can be utilized to build the statement component).

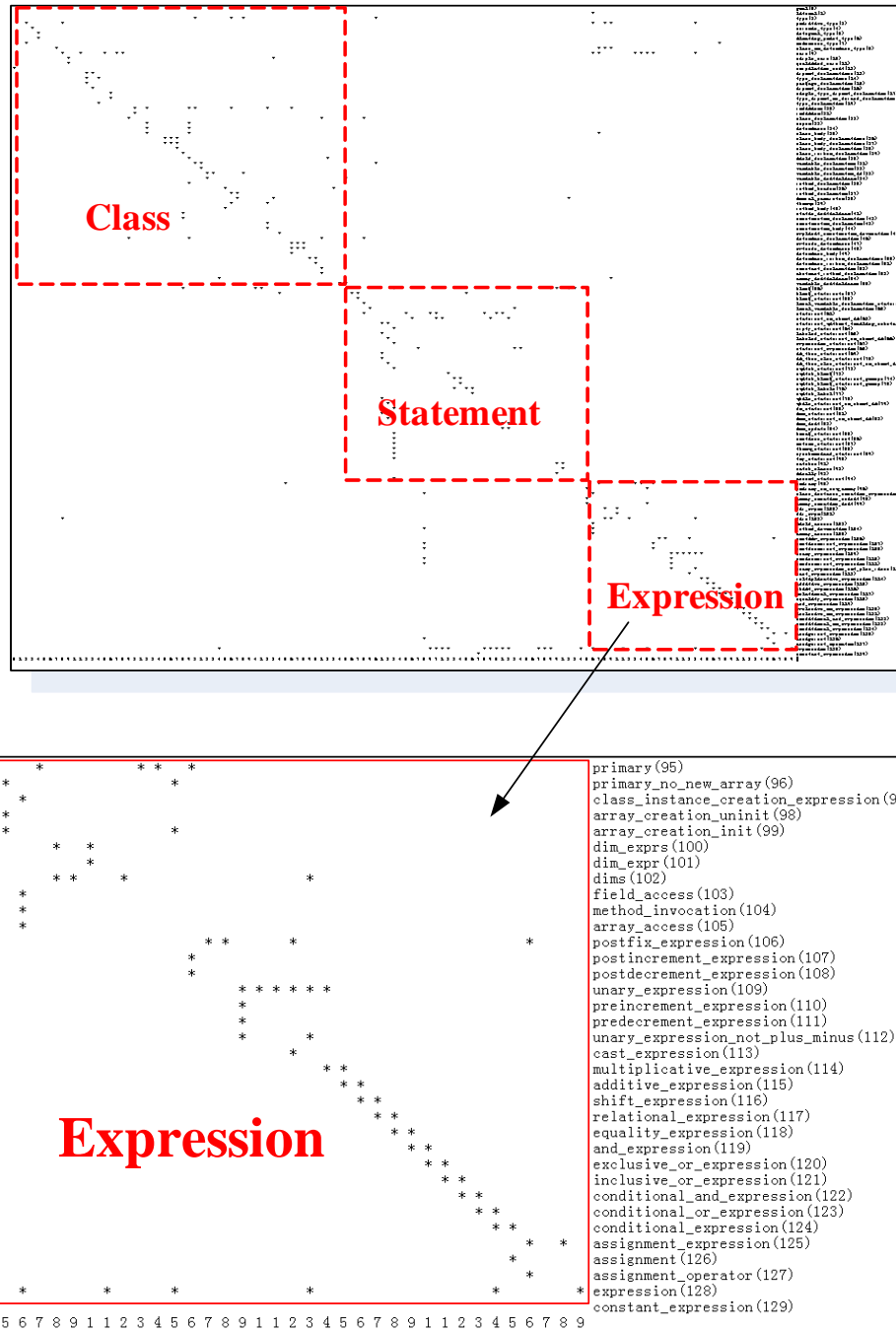


Figure 2: Illustration of production grouping in the JLS grammar

These groups of productions suggest the basic idea of language components. We formally define *Component-based Context Free Grammar* as follows:

**Definition:** A CCFG component  $G$  is a quintuple  $(N, T, C, P, S)$ , where  $N$  is a set of nonterminal symbols,  $T$  is a set of terminal symbols,  $C$  is a set of symbols representing grammar components (called  $G$ 's *child components*, which may include  $G$  itself) with  $T \cap N = T \cap C = N \cap C = \emptyset$ ; the relation  $P \subseteq N \times (N \cup T \cup C)^*$  is a finite set of production rules;  $S$  is the start symbol with  $S \in N$ . A production of the form  $A \rightarrow \alpha$  means  $A$  derives  $\alpha$ , where  $A \in N$  and  $\alpha \in (N \cup T \cup C)^*$ . The *Component-based Context-Free Language* (CCFL) produced from grammar  $G$  is denoted by  $L(G)$ . The union of CCFLs produced from  $G$ 's child components is denoted by  $L(C)$ . Let  $\sigma \in (N \cup T \cup C \cup L(C))^*$ ,  $\tau \in (N \cup T \cup C \cup L(C))^*$ , and let  $\gamma_1 = \sigma B \tau$  and  $\gamma_2 = \sigma \beta \tau$ . Then  $\gamma_1$  directly derives  $\gamma_2$ , denoted  $\gamma_1 \Rightarrow \gamma_2$ , if one of the two conditions is met: 1)  $B \rightarrow \beta$  is a production in  $P$ ; 2)  $B \in C$  and  $\beta \in L(B)$ . Furthermore, if there is a sequence of (zero or more) strings  $\delta_1, \dots, \delta_n$  such that  $\gamma_1 \Rightarrow \delta_1 \dots \Rightarrow \delta_n \Rightarrow \gamma_2$ , we say that  $\gamma_1$  derives  $\gamma_2$ , denoted  $\gamma_1 \Rightarrow^* \gamma_2$ . If  $S \Rightarrow^* \alpha$ , we say that  $\alpha$  is a *sentential form* of  $G$ . A sentential form with no nonterminal and component symbols is called a *sentence*. Therefore,  $L(G)$  is the set of all sentences that can be derived from the start symbol  $S$  by sequential application of production rules and replacing grammar components with their corresponding languages (i.e.,  $L(G) = \{x \mid S \Rightarrow^* x, x \in (T \cup L(C))^*\}$ ).

CCFG is the specification language for describing a grammar component. Inside a grammar component, the component preferably should have appropriate granularity. In other words, its derived sentences should be rich enough to be treated as a language, such

as an expression language. This enables the decomposed unit to be developed and tested independently.

A single language can be described by a set of CCFG components together, where the component containing the entry point to the language serves as the *root component*. The set of CCFG components must be closed (i.e., components being referenced in any other component must be defined within the set). Circular reference is allowed across CLR's component definitions. The restriction is that the component set should not contain "left-recursive" references, namely, a component symbol should never be the first symbol of its own sentential forms; otherwise it may cause infinite loops. Notice that unlike LL grammars, this restriction is placed at the component level, not the production level. As CCFG components themselves are LR grammars, there is no problem to have left-recursive productions in CCFG.

### 3.1.2 Component-based LR parsing

In CLR parsing, each grammar component in a CCFG set is generated as a separate parser. Each parser has a unique identifier which serves as the symbolic link when parsers are composed at the code level. These parsers are applied sequentially to parse different segments of the same input stream. The first invoked parser, called the *root parser*, is generated from the root component of the CCFG set. It invokes some sub-parsers that will recursively invoke other parsers as needed. Therefore, except for the root parser, each component parser will have a *parent parser* (which invoked this parser) and zero or more *child parsers* (which will be invoked by this parser). Those parsers having

no child parsers are called *leaf parsers*. This makes all the parser instances compose in a tree-like structure when a program is parsed.

CLR is an extension to LR parsing. In addition to shift and reduce actions employed in LR parsing, the algorithm adds two new external actions: *return* and *switch*. A switch action transfers the parsing task from a parser to one of its child parsers and a return action returns it back. Although shift and reduce actions are determined by the current stack state and the next lookup (in case of LR(1)), whether to switch or return solely depends on the current state configuration, regardless of the next input. This is because the parser components are independent of each other, so the parent parser cannot pre-compute what is the first token of the child parser; therefore, it cannot make a decision upon external actions based on lookups. As the shift and reduce information is kept in a table, the switch and return conditions are stored in data structures called the *switch map* and *return map*, respectively. Given a particular state as the key, the switch map returns a small set of parser components (usually zero or one) that can be switched to at this state, and the return map simply returns a true or false value indicating if the parser is returnable. By consulting these two maps, the external actions are executed to switch the parsing task from one parser to another when internal actions are found ineffective.

Although there are no internal conflicts generated when LR parsers are composed, composition may generate new conflicts between internal actions and external actions, as well as conflicts among external actions themselves. To ensure the correct action is selected, the following rules are applied in the CLR parsing algorithm. By enforcing these rules, there is at most one action that can be selected at any particular parsing state.

Hence, CLR always produces a deterministic parsing result for any set of CCFG components.

- For conflicts between internal actions and external actions, the internal actions will take place first until an error state is encountered, when external actions will be checked.
- For conflicts between external actions (i.e., return-switch conflicts and switch-switch conflicts), switch actions have higher priority than return actions; the priorities among all possible switch actions are specified at the grammar level by the appearance order of related component symbols .
- Any incorrect selection of internal or external actions should be corrected by recovering the stack and executing other external actions until one of them ultimately succeeds.

Figure 3 demonstrates the flowchart of CLR parsing from one parser's perspective. It shows that CLR parsing extends regular LR parsing by introducing parser dispatching actions. The extended part is invoked when the executing parser reaches a point that there are no internal actions available for the next input token (i.e., an error state). The detailed logic used in parser dispatching is presented below as an algorithm.

**Algorithm:** Component parsing

**Input:** The unparsed part of the input program and the stack of the executing parser.

**Output:** A flag to indicate whether the general LR parsing should be resumed or terminated, or simply flag an error.

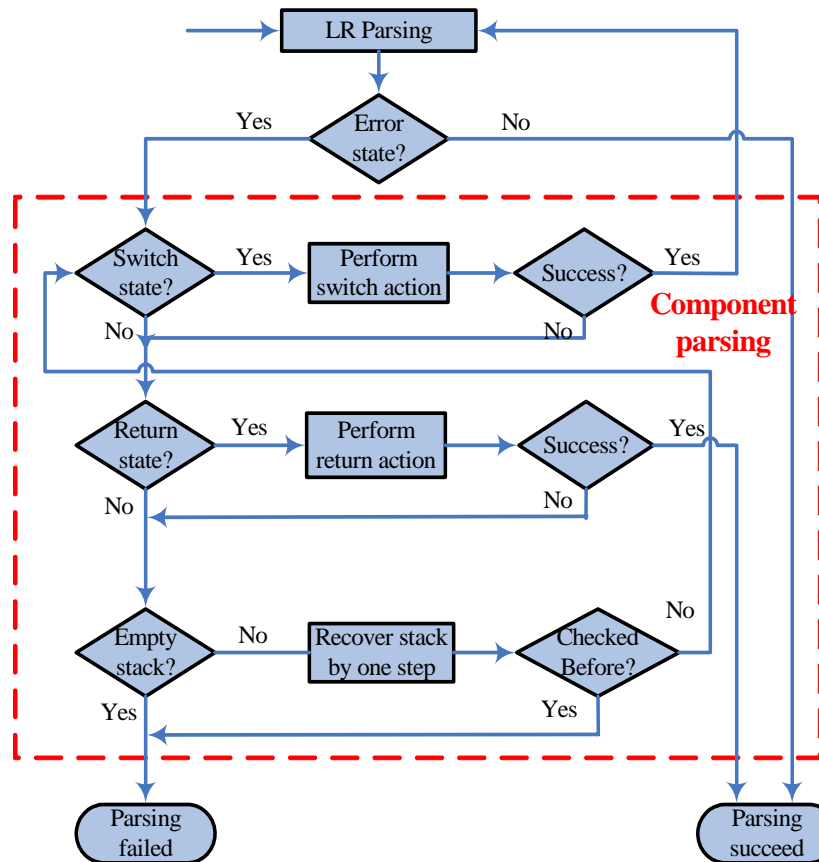


Figure 3: Flowchart of component-based LR parsing

**Description:** Once the component parsing is invoked, the switch condition will be checked first. If the available components returned by the switch map are not empty, the switchable components will be invoked one after another as child parsers to take over the parsing task, until one of them successfully parses the next segment of stream and returns. The component parsing in this case will return a continue signal to resume the LR parsing of the current parser. Otherwise, if no child parsers can parse the remaining stream, or if there was no switch condition available, the return condition will be examined. Consequently, if the parser is in a return state, the parsing task will be returned to this parser's parent parser. If the return action is successful, it can be claimed

that this component has finished its parsing job and should be terminated. Otherwise, as all external actions fail, the current parsing stack will be recovered to the previous stage and external conditions will be re-checked. The same procedure will be repeated until a stage is reached where the external actions have already been examined before, or the parsing stack simply becomes empty, in which case the algorithm claims that this component's parsing fails by throwing an error flag. The pseudo-code presented in Figure 4 provides a more formal description of this algorithm.

```

Pseudo code:
flag component_parse(program, stack):
  repeat
    state := stack.top()
    if switch_map (state) ≠ ∅
      ∀ component ∈ switch_map (state)
        if component.lr_parse(program) == true
          record stack configuration
          return continue_flag
        end if
    end if
    if return_map (state) == true
      if return action success
        return termination_flag
      end if
    end if
    if stack ≠ ∅
      recover stack by one step
    else
      return error_flag
    end if
  until reach the stack configuration when
    last switch action happened
  return error_flag

```

*Figure 4: Pseudo code of component parsing*

Below, a simple language illustrates how the algorithm works. Suppose there is a language  $X$  that only takes three strings: “aa”, “ab”, and “ac”. By using CCFG, this language is built on top of two language components  $Y$  and  $Z$  that each contains one string “b” and “ac”, respectively. The composition of language  $X$  is illustrated by the CCFG below:

$$\begin{aligned} G_x &= ( \{S_x\}, \{\mathbf{a}\}, \{G_y, G_z\}, \{S_x \rightarrow \mathbf{a a} \mid \mathbf{a} G_y \mid G_z\}, S_x ) \\ G_y &= ( \{S_y\}, \{\mathbf{b}\}, \emptyset, \{S_y \rightarrow \mathbf{b}\}, S_y ) \\ G_z &= ( \{S_z\}, \{\mathbf{a, c}\}, \emptyset, \{S_z \rightarrow \mathbf{a c}\}, S_z ) \end{aligned}$$

Suppose we have four programs to parse, namely, “aa”, “ab”, “ac” and “ad”. The parser  $X$  (together with sub-parsers  $Y$  and  $Z$ ) should be able to successfully parse the first three programs and reject the last one. Figure 5 illustrates the action sequences (internal and external) and the stack evolution for all of the four input instances, with each parsing procedure detailed as follows:

For program “aa”, parser  $X$  is able to parse the program by two shift and one reduce actions, without the help of other parser components. In this case, it is simply LR parsing.

For program “ab”, parser  $X$  will first shift the input character ‘a’ onto the stack. However, when parser  $X$  encounters input ‘b’, there are no internal actions available in its parsing table because ‘b’ is not even in  $X$ ’s terminal set. Therefore, parser  $X$  enters an error stage. At this time, component parsing is invoked to check if there are any external actions available. The switch map returns component  $Y$  for this state and parser  $Y$  will be tried. After parser  $Y$  shifts the second character ‘b’ onto the stack, again we reach an error state. Since parser  $Y$  is a stand-alone parser, there is no child parser  $Y$  can switch to. However, from the return action map, it is seen that  $Y$  is currently in a valid return stage.

Therefore, parser Y will reduce its token into  $S_y$  and return it to parser X as  $G_y$ . The parser X then reduce  $aG_y$  into  $S_x$  and claims the success of parsing. Figure 6 illustrates the resulting parse tree, where a circle represents a regular node (in this case they are all terminal nodes) and a square represents a root node for a parser component.

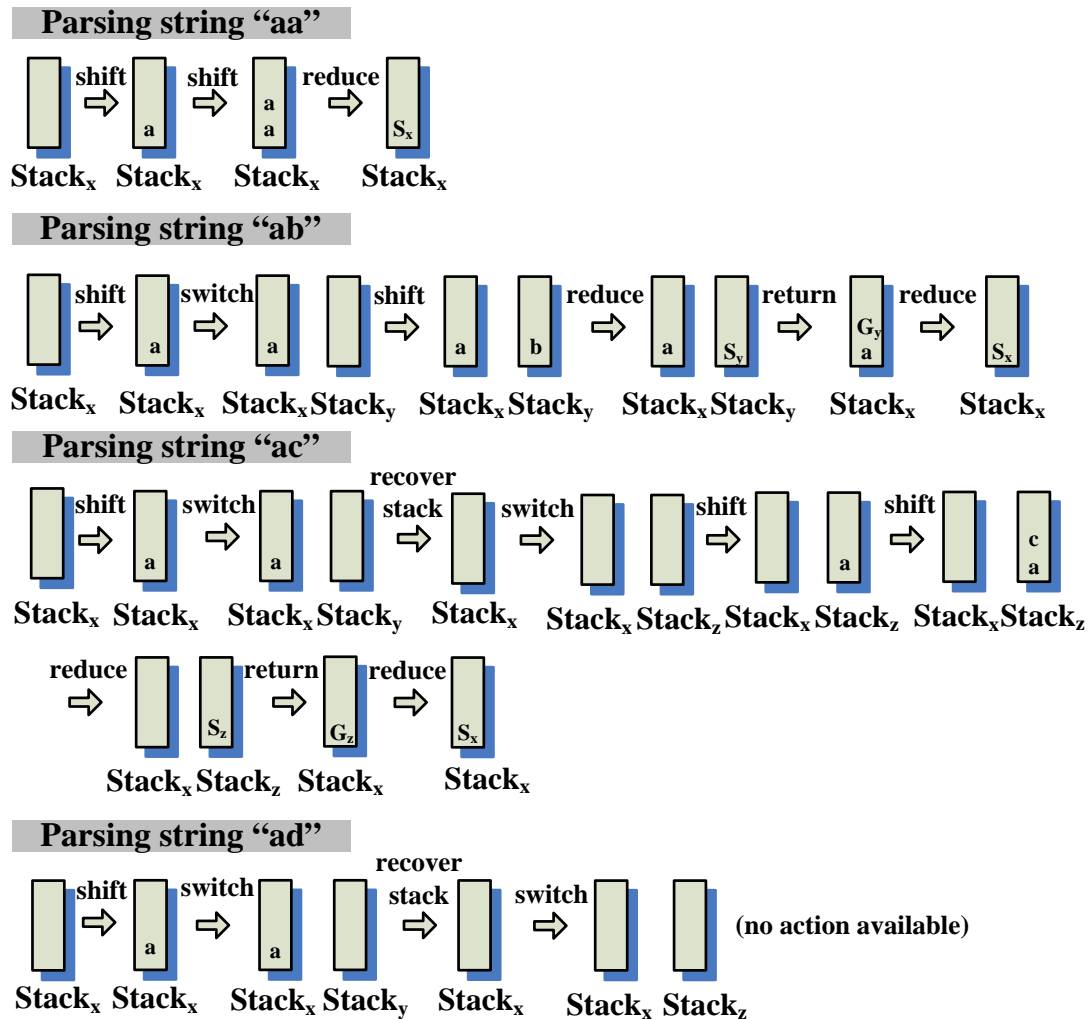


Figure 5: Sample action sequence and stack evolution in LR parsing

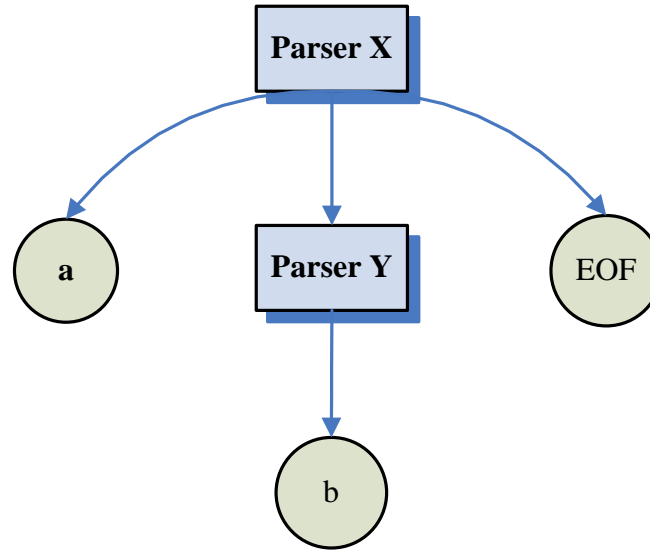


Figure 6: A parse tree instance of Language  $X$

For program “ac”, similarly, after parser X shifts ‘a’, it enters an error state and attempts to switch to parser Y. However, this time parser Y also cannot parse the next input token ‘c’. Because there is no switch or return state available and its parsing stack is empty, parser Y immediately fails and is discarded. As parser X is the root parser, there is no return action for it. Therefore, now the only choice for parser X is to recover the stack by one step (i.e., pop the ‘a’ character from the stack) and recheck the external actions. The switch map returns component Z, so parsing is switched to parser Z. Similar to parser Y recognizing ‘b’ in case 2, parser Z recognizes “ac” and returns to parser X before parser X reduces  $G_z$  into  $S_x$  and claims the parsing successful.

For program “ad”, similar to case 2, parser Y fails in parsing the ‘d’ character and parser X’s stack is recovered. Then parser Z is able to parse the first character ‘a’, but it fails at the ‘d’ character as well. As parser Z has no component symbols and it hasn’t reached a return point yet, there is no external action that can be used, so the parser Z

fails. At this point, because there are no more external actions available and the stack is empty, parser X totally fails the parsing. Because it is the root parser, a syntax error is declared at this point.

### 3.2 Implementation of CLR Parsing

The CLR parsing algorithm and its associated parser generator have been fully implemented in Java by extending and modifying parser and lexer generators CUP and JLex [JLex 2006] respectively. The external actions have been added and the priority rules for resolving conflicts (i.e., internal - external action conflicts and conflicts among external action themselves) are properly enforced. For each CCFG component, a parser and lexer are generated as a Java package.

Figure 7 provides a sample CLR specification for the example languages mentioned in 3.1.2. It can be seen that a general CLR component generally contains four parts: component name, import declaration, production definitions and terminal definitions, which correspond to symbols S, C, N & P, T in the CCFG definition, respectively. The component name serves as both the default start symbol of the component and the language identifier used for composition. The identifier following the keyword “as” (e.g., `y_language`) provides an alias of the imported component, which can be directly used inside the current component. The order of the import declarations indicates the trial sequence when there are switch-switch conflicts, in which case the most frequently occurred sub-language should have higher priority. In order to make each specification module partially reusable, additional start symbols can be declared using an `export` declaration. Each exported symbol will generate a separate parser that uses it as the start

symbol. In this case, the component name and a start symbol together represent the unique identification of a sub-language of the component, which can be imported by other components as child parsers. For example, if a component only wants to reuse the ‘c’ character of language z, it can achieve that by importing id z.z1 (the grammar of the Z component is tailored for demonstration purposes). Consequently, from Figure 7, a total of four parsers and three lexers are generated in three Java packages (parser z and z.z1 share the same lexer and package), among which the parser x will use parser y and z to conduct its parsing functionality.

<pre> <b>Component X</b> // Component name. <b>language</b> x; // Imported components. <b>import</b> y <b>as</b> y_language; <b>import</b> z <b>as</b> z_language; // Product definitions. x ::= A A   A y_language      z_language; // Terminal definitions. A   'a'; </pre>	<pre> <b>Component Y</b> <b>language</b> y; y ::= B; B   'b'; <b>Component Z</b> <b>language</b> z; // Exported symbols. <b>export</b> z1; z ::= A z1; z1 ::= C; A   'a'; C   'c'; </pre>
---	---

*Figure 7: The CLR specification for languages X, Y and Z*

In the following two sections, the difficulties in implementing this system will be first discussed, followed by the corresponding solutions.

### 3.2.1 Implementation Challenges

To use the CLR parsing algorithm correctly, there are several questions to consider. First, how are the external action maps constructed? As described in Section 3.1.2, to transfer the parsing task from one parser to another once an error state occurs, the switch map and return map have to be consulted. The question is how to determine if a particular state is a switch or return state and where to store such information. Particularly, it is difficult to validate a return state because an LR parser normally identifies the termination of the parsing by shifting the final token EOF, which is less likely reached by a component parser.

Second, how is it determined if a component's parsing is truly successful? In LR(1) parsing, a valid shift or reduce action is secured by the next lookup. However, in CLR parsing, since each parser has its own parsing table, it is not easy to conduct lookups for the external actions. Moreover, because LR (1) grammars are not closed under composition, the overall "composed grammar" in CLR might not be a valid LR (1) grammar, which means a single lookup cannot always solve the problem. It is possible that a segment of the input stream can be parsed by more than one component, but only one of them is the ultimate correct choice. Therefore, a return action of a particular parser can only be treated as a temporary success of its parsing. Further processing is needed to guarantee it is indeed the correct choice.

Third, how is the parsing stream synchronized? In CLR, each parser has its own lexer and definition of tokens, so synchronization is needed to enable multiple lexers to work on the same source stream sequentially. Because each LR(1) parser requires its lexer to look ahead one token before an action is executed, the stream's state has to be

reset after the parsing is switched. Furthermore, there may be wrong actions that are executed. To retry other actions, the lexical stream should be reset to the previous stage along with the parser's stack.

Fourth, how are semantic actions incorporated with syntax specification? Normally, an LR parser generator is able to embed semantic actions with each production, which will be executed once the production is reduced. However, because CLR parsing involves backtracking, which means there could be false reductions, the side-effects caused by executing related semantic actions must be revoked in a certain way.

### 3.2.2 Implementation Strategy

To solve the obstacles mentioned above, the following techniques have been employed. First, external actions are implemented by transforming them into internal actions. In the implementation of CLR parsing, instead of providing separate data structures to host the external action maps, switch and return information is integrated into the original shift-reduce parsing table. For the switch action, this is achieved by introducing a dummy terminal symbol (i.e., a token that doesn't exist in the input parse stream) to represent a child component<sup>5</sup>. Whenever it is needed to consult the *switch map*, we simply return those dummy symbols under which there are shift actions available (i.e., assuming the next input token is a dummy symbol). If it is not empty, the component symbols that are able to switch to are recorded. For the return condition, we simply return whether the current state has shift or reduce actions available under the EOF token. In summary, to check an external action, we examine the current stack state against predefined dummy tokens

---

<sup>5</sup> This idea is borrowed from the error-recovery mechanism utilized in most LR parsers, where a special nonterminal "error" is utilized as the dummy token.

instead of the next real input token. For example, in the implementation of language X, Y is represented as a dummy terminal symbol. When an “a” character is shifted onto the stack of X and external actions are checked, it will be shown that Y language can be switched to regardless what the next input symbol is. After Y language recognizes a “b” character, it will reach a return state even if the next input is not an EOF token.

Second, in order to make sure a wrong switch or return action can be corrected eventually, backtracking is employed in CLR. For switch actions, as illustrated in the left part of Figure 8, the corresponding child parsers are created one after another to try parsing the remaining stream until one of them succeeds. Each time a switch fails, the parsing stack will be recovered to make sure the next child parser begin its experiment from the same point. If none of the child parsers can successfully parse the stream, the whole switch parsing fails. For return actions, two types of components are defined: *perfect components* and *regular components*. A perfect component is one that has no ambiguity with its surroundings (i.e., using this component is the only way to interpret any sentence in that language). This type of component usually comes with special guards, such as statement (ending with ‘;’ or ‘}’), or class definition (beginning with ... `class` and ending with ‘}’) in the Java language. For these components, a valid return action can guarantee the previous parsing with this component parser was the correct choice. In the current CLR specification, a perfect component is indicated by prefacing the keyword `perfect` to the language id or the start symbol of an export declaration, as in `perfect language foo` or `export perfect foo`. Regular components are those components whose sentences may fully or partially overlap with other component sentences,

which are usually the small components that do not have clear guards. Parsing with such components hinders determining if the return action is correct.

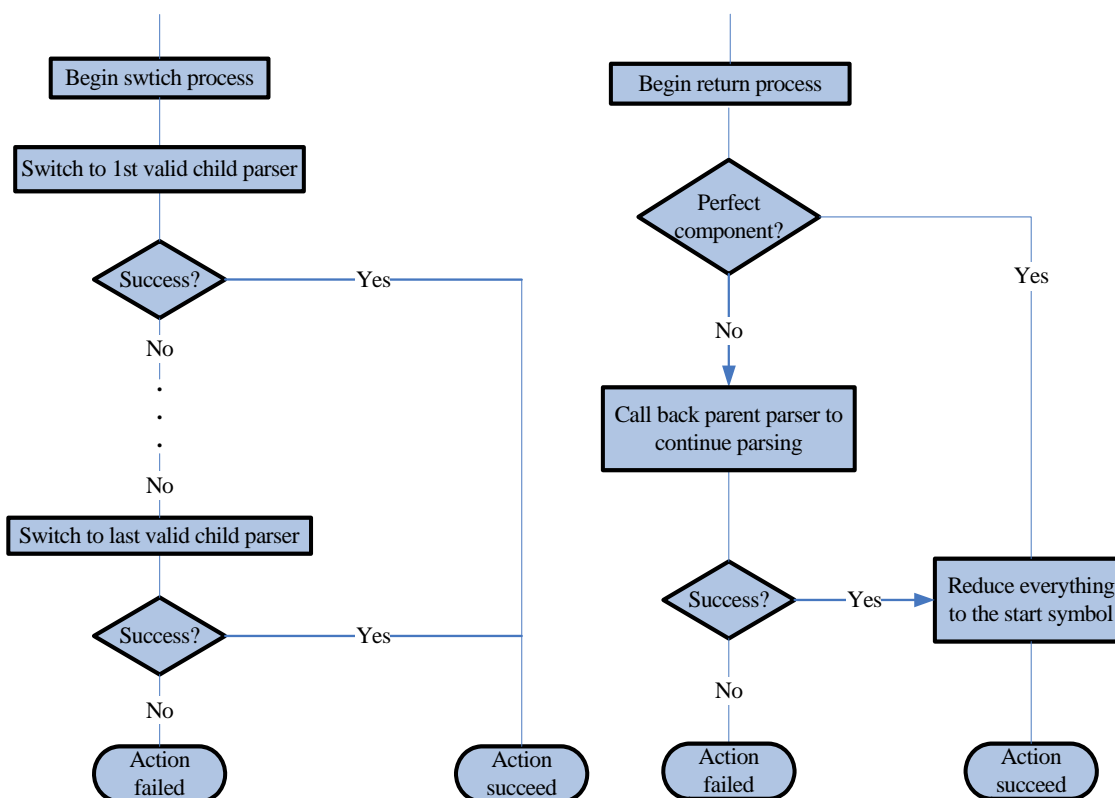


Figure 8: The implementation detail of the switch and return actions

As described in the right part of Figure 8, for a perfect component, the return actions will simply iteratively reduce all the symbols in the stack to the start symbol and then return the control to the parent parser to continue its parsing. The parser object is immediately thrown away after the return. However, for a regular component, instead of returning to the parent parser directly, the child parser will *call-back* the parent's parsing method to resume the parse. In this case, the success of this return action is actually determined by the success of the parent parser's subsequent parsing result. If the parent

parser fails at a certain place, this parser's stack has to be backtracked to a place where another external action is available. Meanwhile, the parent parser's stack will be recovered to the stage where the return action has not been executed. For example, in a programming language such as Java, if the grammars of the cast expression and the parenthesized expression are specified in two separate components, the prefix `( int )` of a cast expression like `( int ) var` might be first successfully parsed by an expression parser as a parenthesized expression. However, after it returns, its parent parser will not be able to continue its parsing because an expression can not be followed by a variable (e.g., `var`). Therefore, the return action will be recovered such that a different parsing path can be tried.

Third, to solve the synchronization problem, a static stream reader equipped with mark and reset functions is shared by all the lexers to harmonize and recover the input stream processing. Each time the parser is switched or returned, the static reader will be reset so that it always points to the unrecognized input stream. Supplying separate lexers for each parser is a significant contribution of CLR parsing (detailed in Section 3.4). However, in the case that multiple parsers do want to share the same lexical definition, we simply replace the keyword `import` with `importNL` (meaning no lexer importation) and then the parser's own lexer will be used directly by its child parsers.

Fourth, to address the problem related to semantic actions, the CLR specification is designed to be purely declarative. Given a CLR grammar component, a strongly-typed parse tree structure is automatically generated along with the parser. The tree construction actions are embedded within the generated parser. After one child parser returns or calls back its parent parser, its parse tree is silently plugged into the parent parser's parse

tree. Because all the tree nodes are stored in the parsing stack, any tree node obtained from a false return or reduce will be eventually popped during the parse stack backtracking. Consequently, the tree structure will no longer hold a reference to this node as its parent node is already popped (last in first out). Eventually, all the parse tree components will result in a single parse tree. Further semantic analysis will be made through the generated tree classes instead of embedding them directly inside the parser. This avoids the need to undo semantic actions once a false reduction is detected.

### 3.3 Case study of CLR Parsing

To validate the benefits of using CLR, we have utilized CCFG to rewrite the Java language grammar in a component-based manner. The overall grammar follows the JLS introduced in [Gosling 1996] with extension to JDK 1.4, which contains 156 nonterminals and 359 productions. Depending on the scale of each component, eleven versions of the Java grammar have been developed, with each version composed of one to eleven grammar components, respectively. Development experience shows that the version that has four components (i.e., `Java language`, `Class`, `Statement` and `Expression`) already offers high-quality compositionality. Other versions that contain more than four components are provided in this section for exploration purposes, with the eleven-component implementation pushing the language modularity to an extreme.

Figure 9 demonstrates the version that contains 11 language components, where an arrow denotes the end component is a child component of the start component. Initially, a `Java language` component is created to describe an instance of the Java program (i.e., a compilation unit). A compilation unit is composed of a sequence of package,

import and class declarations. The definition of package and import declarations is short enough to be specified in the `Java` component directly. However, production rules related to a class declaration tend to be lengthy and closely coupled, so a class declaration is extracted out as an individual language for compositional development. Applying the same strategy recursively, productions related to variable declarations, statements and expressions are factored out as grammar components, among which the `Expression` component is further decomposed into `Binary`, `Unary`, `Primary` and `Postfix` components. Type related productions are also encapsulated as `Type` and `Primitive type` components, where the latter is a child of the former component.

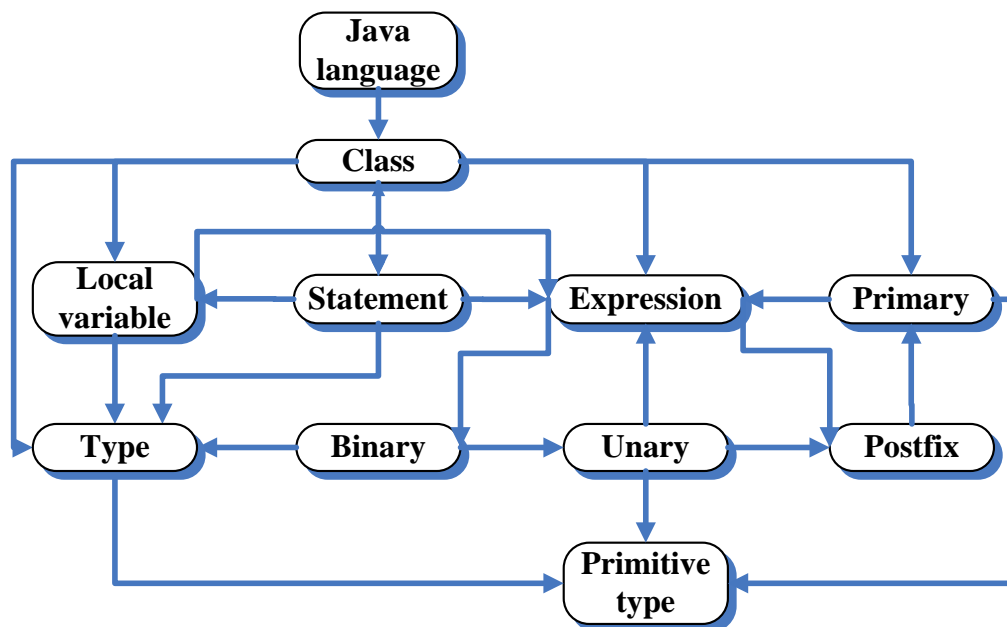


Figure 9: CLR components of JLS (11 component version)

Figure 10 provides an illustration on how the four component implementation parses the text of a sample Java program<sup>6</sup>. A total of five parser instances are created sequentially to parse the input stream, which is segmented into eight corresponding sections. Compared to single parser implementation (generated from 800 lines of grammar), each parser component focuses on a relative small scope, which is more modular. For example, the statement parser is only used to parse the keywords and separators, whereas the parsing of concrete expressions is left to the expression parsers.

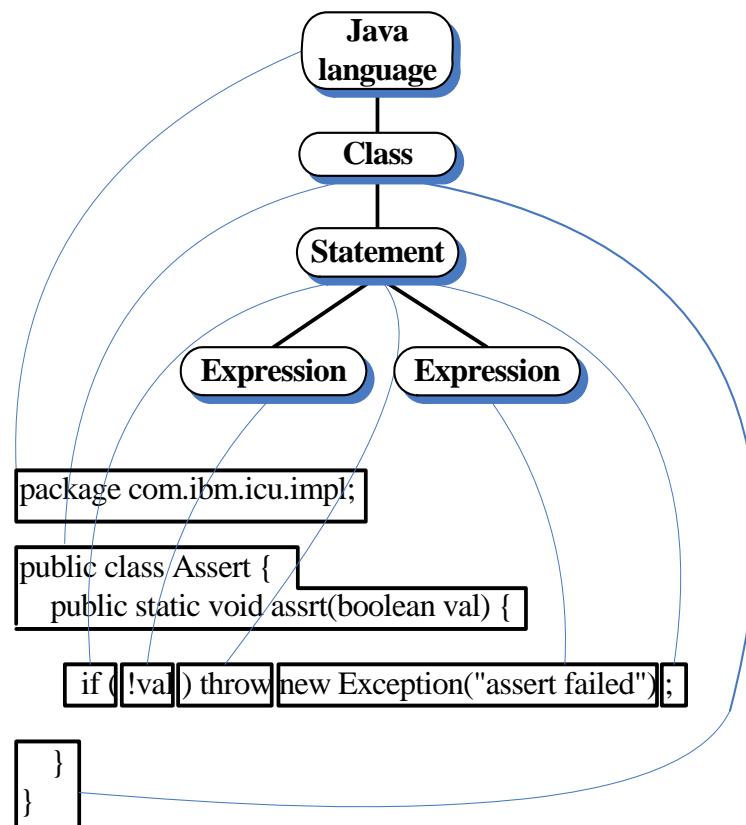


Figure 10: Parsing illustration of a sample Java program using the CLR JLS implementation (4 component version).

<sup>6</sup> This is adapted from a Java program of Eclipse SDK-3.2 (available at <http://dev.icu-project.org/cgi-bin/viewcvs.cgi/icu4j/src/com/ibm/icu/impl/Assert.java?revision=1.2&view=markup>).

### 3.4 Evaluation of CLR Parsing

Based on our experience with applying CLR to JLS, together with various other CLR practices described below, we have observed positive results in using CLR parsing technology. This section evaluates the implementation of the CLR parsing algorithm against regular LR parsing using three criteria: compatibility with software engineering principles, language description ability and performance. A discussion of CLR's drawbacks and limitations is also provided.

#### 3.4.1 Software Engineering Benefits

CLR parsing provides benefits toward changeability, reusability and independent development of a language implementation. Moreover, the CLR specification is more comprehensible than conventional LR grammars

- **Changeability.** CLR parsing is naturally equipped with information hiding. A change to a component is isolated inside the component to avoid propagation. Furthermore, since each component generates a separate parser, any change inside a grammar component will only require the recompilation of itself. This is a major difference with other modular grammar development approaches such as PPG and SDF, where any change inside a component will require recompilation of all the related modules. This is discussed further in Section 3.5.
- **Reusability.** As each component is an independent entity, it is free to compose and reuse different components to build new languages. For example, to extend the Java language to SQLJ [Melton 2000], we simply need to add an SQL parser as the child parser of the `Statement` parser to integrate SQL statements as legal

statements in Java. Moreover, using CLR parsing, after a parser for a language is developed, a set of sub-language parsers are also ready to reuse. Specifically, they might be reusable to build other Domain-Specific Languages (DSLs), especially those designed using the piggyback pattern (i.e., building a new language by partially reusing existing languages) [Mernik 2005a]. For example, the Binary component in the Java specification, which has more than 50 lines of CFG productions, can be reused to build the binary expression part of C or C++ with little change (illustrated in Appendix A). The Java statement parser can be reused to implement compiler-compilers such as CUP, where Java semantic actions are embedded in the BNF specifications. The sub-components are also reusable in development of Integrated Development Environments (IDEs), where various entities such as the outline view and debugger require parsers for class definitions and expressions.

- **Independent development.** In CLR, each grammar component has its own namespace and start symbol. Dependencies among components are handled at the code-level instead of the grammar level. Left-recursion is the only thing that needs to be concerned with globally. In the current implementation, the generated Java class of each parser maintains a set of symbols that indicate the “first set” of the corresponding grammar component, and a left-recursion check is made when a new parser is loaded. Notice that this analysis is conducted dynamically at parsing time. Therefore, the global analysis is achieved beyond the grammar level. As a result, the loose coupling in CLR enables each language component to be individually developed, debugged and tested against different portions of a source

program. Particularly, to debug a component whose child components are not developed yet, an empty component can be used to represent the child components. Therefore, multiple developers can work on the grammar of the same language concurrently and produce separate parser components. This reduces the development cycle of syntax analysis and benefits other compiler phases that follow.

- **Comprehensibility.** By decomposing the large grammar into small grammar components, the number of intertwined symbols and productions inside a single component are reduced, resulting in a specification that is easy to understand and maintain. In the original JLS specification, explicit long names have to be created to capture the necessary information for improved comprehensibility. In CLR, because the role of each symbol is defined by the component name and nonterminal name pair, the name of each symbol tends to be short and precise. For example, in the `statement` component, the nonterminals `local_variable_declaration_statement`, can be shortened as `local_variable_declaration`. Moreover, inside a grammar component all the symbols and productions are related to certain language entities (e.g., expressions, statements), which makes the definition more cohesive.

### 3.4.2 Language Description Ability

In terms of language description abilities, a CLR grammar is more expressive than a regular LR grammar due to its backtracking mechanism. Essentially, the expressiveness of CLR is a superset of LR and a subset of arbitrary context-free grammars. It can describe any unambiguous context-free language. Additionally, by enabling multiple

lexers to tokenize the same stream, CLR parsing is able to solve the problems caused by ambiguous tokens, which normally occur in conventional LR parsing.

- **Expressive power.** Although the CLR approach generates LR parsers, the CLR grammar is richer than a pure LR grammar. In principle, the backtracking logic employed in CLR parsing is equivalent to infinite lookahead [McPeak 2004]. By extracting the shift symbol and its associated productions into a separated component, CLR's backtracking can resolve the traditional shift-reduce or shift-shift conflicts in LR parsers. Therefore, it is able to produce a deterministic result as long as the grammar is not inherently ambiguous. An example of using this advantage is the processing of *island grammars* [van Deursen 1999, Moonen 2001], where the structure of interesting parts of a language (the *islands*) is described in detail and the remaining part (the *water*) is simply mentioned as character streams. This type of language normally is difficult to handle by regular parsing methods due to the conflicts between the island and water. Figure 11 provides an island grammar example (adapted from [Visser 2000]) that is used to extract email addresses from an arbitrary text document. The overall grammar is difficult to specify using LR grammars because it is ambiguous (e.g., terminal @ can be either recognized as an email symbol or one of the water characters). However, in CLR, by extracting the water definition as an external component, the ambiguity is naturally resolved. As in Figure 11, the symbol `email` is defined by internal productions and `water` is defined as an imported component. This gives priority to the email concern. Consequently, the `island` parser will always first parse the

stream as email addresses until there is no internal action available, when the water parser will be used to consume a single token and return.

```

language island;
import water as other;
// Syntax definition
island ::= unit | unit island;
unit ::= email | other;
email ::= SEGMENT AT hostname;
hostname ::= SEGMENT DOT SEGMENT | SEGMENT DOT hostname;
//Lexical definition
DOT ".";
AT "@"|"("at)"|"at";
SEGMENT [A-Za-z][A-Za-z0-9\-\_\]*;

language water;
water ::= ANY;
ANY .; // \.' means any character

```

*Figure 11: CLR specification for the island grammar example*

- **Ambiguous tokens.** Allowing an independent lexer for each parser permits the same token to be interpreted differently inside various component scopes. This is very helpful for processing embedded languages where the host language has totally different reserved words from the guest language. For example, SQLJ could be implemented by a Java parser that hosts an SQL parser, with each having its own lexer to tokenize input programs. Therefore, a token like `count` will be naturally recognized as a variable in the Java scope and as a keyword in the SQL scope. If we use a conventional communication mechanism between a single parser and single lexer, all the reserved keywords will be prohibited, regardless of the scope. Similarly, multiple-lexers can also benefit those languages that have no

reserved keywords, such as PL/I, where a statement like `IF IF = THEN THEN IF = THEN;` is legal. In a proper CLR implementation where expressions and statements are described in two separate components, `IF` and `THEN` are only treated as keywords in the statement component, but not in the expression component. Consequently, only the first `IF` and second `THEN` will be parsed as statement keywords, but others are parsed as identifiers in the expression. Similar usage can also be applied to a stand alone language that does have reserved words. For instance, an annoying fact for C++ developers is that most C++ parsers surprisingly fail to parse a template reference like `vector<vector<int>>` although it looks like a legal syntax. The problem is caused by the fact that the two consecutive right angle brackets are falsely recognized by the C++ lexer as a right shift token. To avoid this problem, the above expression has to be rewritten as `vector<vector<int> >`, where one or more empty spaces are inserted between the two brackets. This problem also can be solved by the CLR implementation easily if type-related productions have their own component, as the right shift token `>>` is only preserved in the `Expression` component (or some of its child components) as operators. In the `Type` component, which should be used to parse the above clause, it will be simply recognized as two template reference guards.

Notice that CLR is designed to handle unambiguous grammars. If there is actually more than one right choice, it means the grammar is ambiguous. In that case, CLR will only find the first correct pass and stop.

### 3.4.3 Performance Measurement

In this thesis, CLR's parsing performance is not directly compared with other parsing tools, due to the fact that the development language (e.g., C vs. Java) and the input grammar can affect the parsing speed significantly. Instead, the performance of CLR is measured with its LALR counterpart and itself with different numbers of components. In general, CLR parsing is slower than LR parsing if there is more than one parser component (CLR is equivalent to LR parsing if there is only one component).

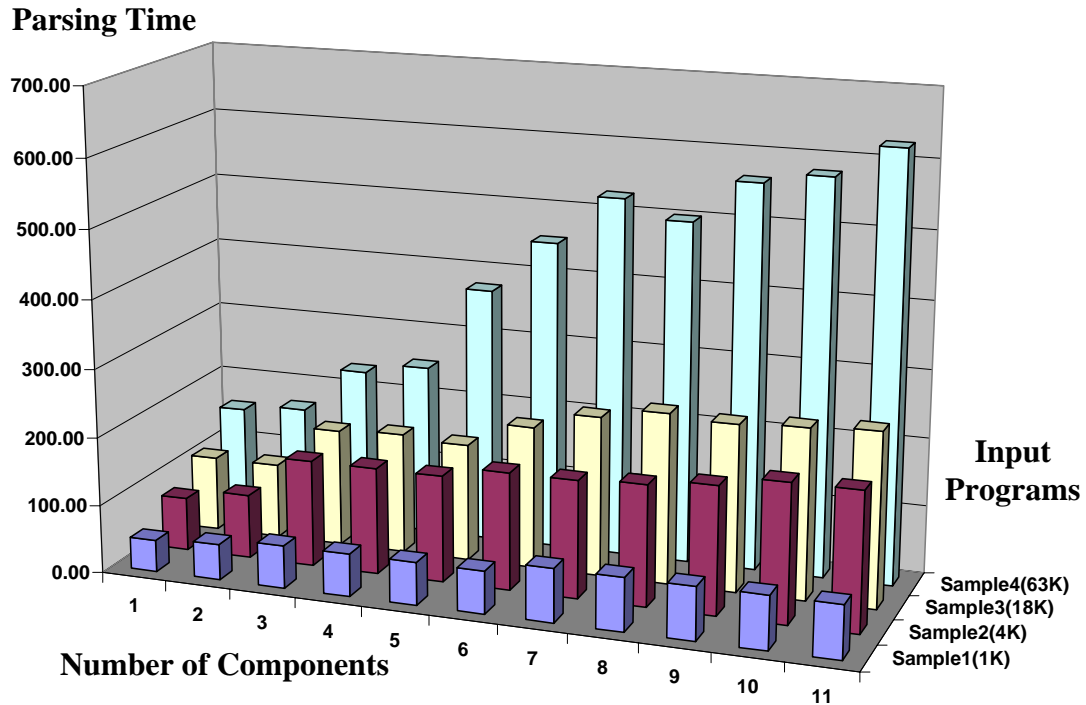
We have tested the parsing speed of the 11 versions of the JLS implementation (the single component version is the regular LALR parser) by using them to parse four randomly selected Java programs from the Eclipse SDK of size 1k, 4k, 18k and 63k<sup>7</sup>. Figure 12 illustrates the results. It is clear that as the number of components increases (1 to 11), the parsing time increases in a sub-linear manner (156ms to 625ms in the case of Sample4). The table provides the actual value of the parsing time and the increasing ratio when a new component is extracted and added to the implementation. Notice that by adjusting the import order of child components in each component, a different set of testing results can be produced. However, in this case the difference is very minor because there are few switch-switch conflicts occurring in these implementations.

There are two reasons attributed to the time increase. First, when a parser is switched or returned, the overhead of the function call is not ignorable, especially when the component is small and parser switching happens frequently. For example, in Figure 13 it is shown that there are totally 3885 switch actions and 2648 return actions executed when parsing Sample4 using the 11 component version. These thousands of extra calls

---

<sup>7</sup> Available at <http://dev.icu-project.org/cgi-bin/viewcvs.cgi/icu4j/src/com/ibm/icu/impl/>. The four programs are named Assert.java, CalendarCache.java, Trie.java and CalendarAstronomer.java, respectively.

are unavoidable because they are needed by the backtracking mechanism, and hence are attributed to the extra overhead of CLR parsing over regular LR parsing.



	1	2	3	4	5	6	7	8	9	10	11
<b>Sample1 (1k)</b>	47	52	62	62	62	62	78	78	78	78	78
		11%	19%	0%	0%	0%	26%	0%	0%	0%	0%
<b>Sample2 (4K)</b>	78	94	156	156	156	172	172	177	187	203	203
		20%	67%	0%	0%	10%	0%	3%	6%	8%	0%
<b>Sample3 (18K)</b>	109	109	172	177	172	209	234	250	245	250	255
		0%	57%	3%	-3%	21%	12%	7%	-2%	2%	2%
<b>Sample4 (63K)</b>	156	166	234	250	375	453	526	500	563	578	625
		6%	41%	7%	50%	21%	16%	-5%	13%	3%	8%

Figure 12: Parsing speed comparison among 11 versions of CLR implementation of JLS

Another factor is that in LR parsing, the program is always parsed deterministically, with the development cost of making the grammar LR, whereas CLR employs

backtracking to resolve conflicts across parser components, which means the same program piece could be tried by multiple parsers until one succeeds. As seen in Figure 13, the number of switches is always greater or equal to the number of returns. Because a success switch is always followed by a corresponding return, the gap actually indicates the number of failed switch actions. As some internal actions may be executed after the parser is switched, failed switches consume a considerable amount of parsing overhead.

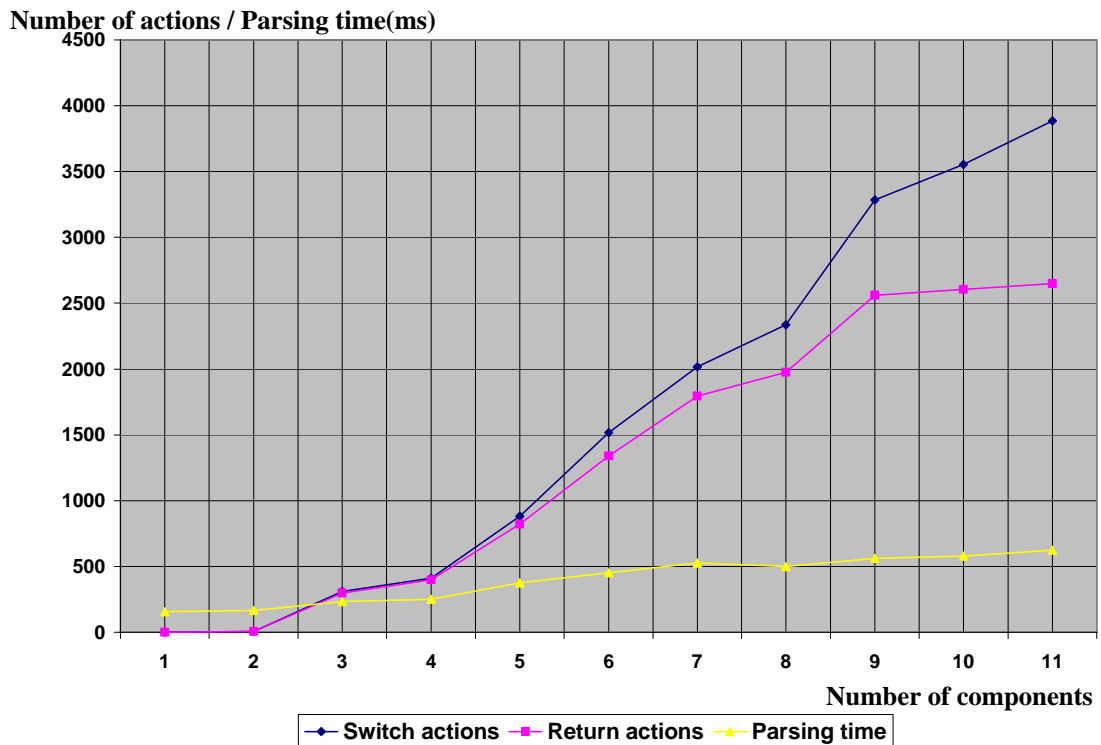


Figure 13: The number of external actions used to parse Sample4

However, because each component parser uses LR parsing internally, most parts of CLR parsing are still deterministic. If the number of components is small (e.g.,  $\leq 4$ ), the dominant overhead of CLR parsing is still tokenization and internal actions. In Figure

13, the corresponding parsing time of each implementation is also provided to reflect the relationship between parsing time and number of external actions. It shows that the increase in the number of external actions is not proportional to the increase in parsing time (e.g., from 1 component to 11 components, the number of external actions increases by several orders of magnitude, but parsing time only increases 2-4 times). Moreover, it is not always the case that the parsing time goes up as the number of components increases. For instance, for input program Sample4, the eight component parser is actually 5% faster than the seven component version. This phenomenon could be explained by the following analysis. As the number of components increases, the size of the parsing table decreases, which optimizes the table lookup time. If the savings on the table lookup overcomes the overhead on extra function calls and backtracking, the overall parsing speed will increase, otherwise it will decrease.

#### **3.4.4 Discussion**

The current CLR implementation has two limitations caused by the backtracking used in CLR. In general, backtracking has the risk of exponential parsing time [van den Brand 1998], so it is possible to develop a set of CLR components that are configured in a hostile manner such that it causes exponential behavior. However, such risk is much less than in general LR parsing:

- In CLR, backtracking only occurs when there is an external action available at the state and the action conflicts with an internal action or another external action. There is no backtracking for shift-reduce or reduce-reduce conflicts as found in regular LR backtracking. In practice, each component only imports a limited

number of other components, each of which is only switchable under a limited number of states. It is rare that there are both internal and external actions available for a certain state. Particularly, parsing by *leaf parsers* (the components that have no imported components) is guaranteed to be linear bounded. Even if they are tried multiple times, the complexity will still be polynomial, not exponential.

- Employing perfect components can significantly prune the paths during backtracking, because once a “perfect language” is recognized, the parsing with this parser will not need to be backtracked (i.e., a perfect component guarantees the interpretation was right).

Overall, CLR gives the developer control over the whole language grammar by resolving most conflicts manually inside a small scope, so that backtracking is seldom needed. In contrast, it is very easy to write a grammar that causes exponential parsing in regular backtracking LR parsers [Johnstone 2004b].

The other problem caused by backtracking is error-recovery. In YACC-like LR parsers, the error recovery process is invoked immediately after an error state occurs. However, in CLR, an error state does not mean the program actually contains invalid syntax until all external actions have been tried. This breaks the error-recovery mechanism of regular LR parsing. The same problem also exists in other indeterminate parsing technologies, such as GLR and backtrack LR. CLR does not currently support error-recovery. A possible solution will be discussed in Chapter 5.

### 3.5 Modular Grammar vs. Compositional Parser

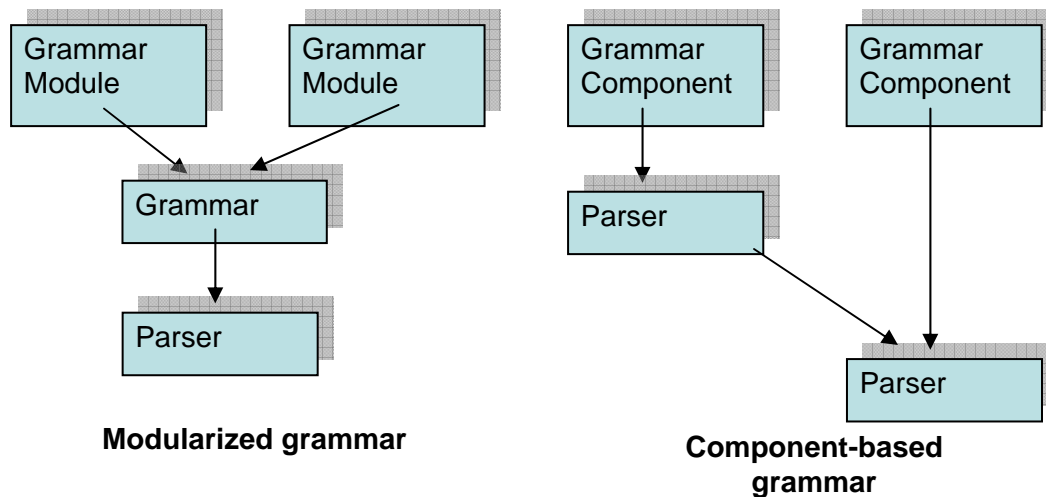
In Chapter 2, a number of parsing technologies that support modular grammar development were introduced. In comparison to these parsing technologies, the significance of CLR parsing comes from the fact that the composition occurs among parsers instead of grammars. Figure 14 demonstrates the difference between modular parser generation and compositional parser generation. Parser-level composition is superior in several aspects:

- The coupling between components resides at the parser level instead of the grammar level. Any update to a particular component does not require a recomposition of all the components in the package and regeneration of a large parse table as in PPG, BtYacc or SDF. For an implementation of a complex language with a number of modules, composition and generation overhead is not negligible. For example, for SDF modules of the Java language<sup>8</sup>, module composition time alone took 8.2 seconds<sup>9</sup>.
- The decomposition of the single parser leads to the decomposition of its parsing table and parser code size. A smaller table sometimes speeds up the table lookup time, as illustrated in Section 3.4.3. A smaller parser can avoid certain problems caused by large code size. For example, our attempt to build a one component version of the JLS parser with a concrete syntax tree construction cannot be compiled by javac 1.5 due to a “code too large” problem (the CUP generated parser contains a 7000-line switch statement).

---

<sup>8</sup> The SDF modules were obtained from <http://www.stratego-language.org/Stratego/JavaFront>.

<sup>9</sup> The test is conducted on a machine with the following specifications: AMD Athlon(tm) XP 2800+ (Cache size: 512 KB), 1 GB RAM, Linux (Fedora Core 2). 8.2 is the average time of 10 rounds of tests.



*Figure 14: Difference between modular parser generation and compositional parser generation*

- As we have discussed in Section 3.4, each parser component can maintain its own lexer, which provides support for ambiguous tokens. To solve the same problem, SDF uses scannerless parsing, which essentially takes each character as a token and leaves the GLR parser to recognize the terminal symbols, at which level ambiguity is allowed. However, this implementation essentially assigns a task originally handled by a Deterministic Finite Automata (DFA) to a more expensive Pushdown Automata (PDA), which definitely hurts performance due to the overhead on stack operations. Some lexer generators such as Lex [Lesk 1975 ] and JLex [JLex 2006] allow the same lexeme to represent different tokens in different states. However, to switch lexer states, the developer is required to manually trigger it as lexer actions, which sometimes is difficult to use as the grammar context is not known [Parr 1995]. Notably, ANTLR allows the same parser to work with multiple lexers. The programmer still needs to invoke the action manually, but the

action resides at the grammar level. The lexer switching process is controlled by a token stream multiplexor, which has the same role as the static reader employed in the CLR implementation.

- Parser composition actually happens at the Java byte code level, which means within the CLR family, a component parser is still reusable by a third party if the source grammar is not released, even if the Java source code is not available.

The only previous effort on code-level parser composition we are aware of is from the functional language community. Notably, parser combinators allow runtime manipulation of parsers [Hutton 1998]. Specifically, a parser is coded directly using a functional language such as Haskell [Jones 2003] and it can be passed around as a parameter. In [Dijkstra 2001], implementing parser combinators in Java is introduced. The essence of this strategy is to handcraft a parser instead of using declarative grammar specifications, which has its obvious drawbacks. Mapping it manually to an imperative language makes it more verbose and inefficient [Hutton 1998]. As a result, its performance is not comparable to table-driven LR parsers [Dijkstra 2001].

For parser generators in which the lexer is accessible in parser actions (e.g., ANTLR), it is possible to invoke a sub-parser by manually inserting semantic actions in the syntax definition, but this comes with a limitation and a development cost. The language module must have clear guards (e.g., JavaDoc has “/\*” and “\*/”). The grammar must be divided in such a way that the open guard is specified in the parent parser to indicate the switch and the closing guard is used as the end token of the child parser to indicate the return. Moreover, besides the parser switching actions, the parsing result from the sub-parser also has to be connected to the main parser manually.

### 3.6 Summary

In this chapter, CLR parsing is presented as a novel parsing technology that is designed to support component-based development in language implementations. It adds the regular LR parsing switch and return actions to dispatch parsing tasks from one parser to another. CLR parsing decreases the complexity of building a large language by constructing a set of smaller language parsers from grammar components, which at the same time strengthens software engineering principles. CLR is more expressive than regular LR as it employs backtracking and multiple lexers to resolve the conflicts at the syntax and lexical levels. Compared to other parser generation tools that support modular grammars, the CLR parser generator produces loosely coupled parser components with more manageable code size and code-level reusability. Notably, each parser component can be individually developed and tested, which helps to reduce the development cycle of the overall language implementation. CLR is an ideal platform to develop programming languages and DSLs with complex and hybrid language constructs. Its usage can be much extended after combining with other parsing technologies, such as LL and handcrafted parsers.

Therefore, CLR successfully solves the first modularity problem in language implementation from a structure perspective. The parsing algorithm can be easily integrated with OOS and AOS, which are designed to address the modularity problem relying on the functional standpoint. Chapter 4 has the detail of OOS and AOS, as well as their integration with CLR.

The source code of the CLR parser generator and some of its sample specifications are available at <http://www.cis.uab.edu/wuxi/research/>.

## CHAPTER 4

### OBJECT-ORIENTED SYNTAX AND ASPECT-ORIENTED SEMANTICS

As explained in Chapter 1, syntax essentially represents the structure of a program, whereas semantics describe the program behavior and intended meaning. In terms of implementation, syntax is typically represented by a data structure called a syntax tree, and the inherited nature of semantics is that each analysis phase crosscuts various syntax tree node classes. Recall the discussion in Chapter 2 that OOP is suitable to specify data structures and AOP is designed for encapsulating crosscutting behaviors, so the design principle of this framework is to handle syntax by object-orientation and perform semantic analysis in an aspect-oriented manner. Moreover, it has also been diagnosed in Chapter 2 that syntax can be easily specified by formal specification but semantics cannot due to its arbitrary nature. Therefore, based on the discussion above and various research practices presented in this chapter, in this framework syntax is described by object-oriented formal specification and semantics are expressed via aspect-oriented programming languages.

The syntax tree serves as the middle layer that connects the syntax analysis and the aspect-oriented semantics together. It has been discussed in Chapter 2 that a syntax tree can be either built by hand or built by tree generators during parsing. Semantics then consult this tree structure to deliver desired behaviors. Normally, syntax grammar processing and syntax tree construction are two separate phases in compiler design.

Because the syntax tree is a direct reflection of syntax grammar, another important design strategy in this framework is to generate syntax trees automatically from syntax specification, instead of separating them as two phases.

In this framework, Component-based LR parsing is integrated with OOS and AOS. However, OOS + AOS can be also used independently in conventional LR parsing paradigms. Therefore, in this chapter, object-oriented syntax and aspect-oriented semantics will be first introduced as a stand-alone development framework, followed by their integration with CLR parsing.

This chapter is organized as follows. The overall architecture in describing syntax and semantics is introduced first. Detailed exploration of OOS and AOS methodologies are presented next as two separated sections. A case study on Java language implementation has been provided to show how object-orientation and aspect-orientation would work together for clear separation of syntax and semantic construction. Integrating CLR into OOS and AOS is presented at the end of the chapter.

#### **4.1 Framework Architecture**

Figure 15 provides the framework of language implementation by OOS and AOS. Tools are shown in ellipses. Shaded boxes contain generated code. Solid lines denote input and output, whereas dashed lines stand for references. To describe a language, language developers first specify the lexical and syntactic rules for each grammar symbol (terminal or non-terminal) in Object-Oriented Context-Free Grammar (OOCFG). The specification will serve as the input to a specification compiler that extracts lexical rules and syntax rules, which can be processed by the lexer generator JLex and parser

generator CUP to generate the corresponding lexer and parser in Java. Additionally, tree nodes are also generated from the syntax grammar as Java classes and interfaces, without any extra construction effort. During the parsing phase, the parser uses the embedded action code in a CUP specification to call the constructors of these node classes to build two tree structures simultaneously, Concrete Syntax Tree (CST) and Abstract Syntax Tree (AST). Each tree serves a different purpose and AST initially shares most of the CST's tree nodes with a different type of node links. Afterwards, AspectJ code for semantic analysis is added without any change to the automatically generated Java classes. After the semantics in AspectJ are weaved into the parser and those node classes, a compiler is produced.

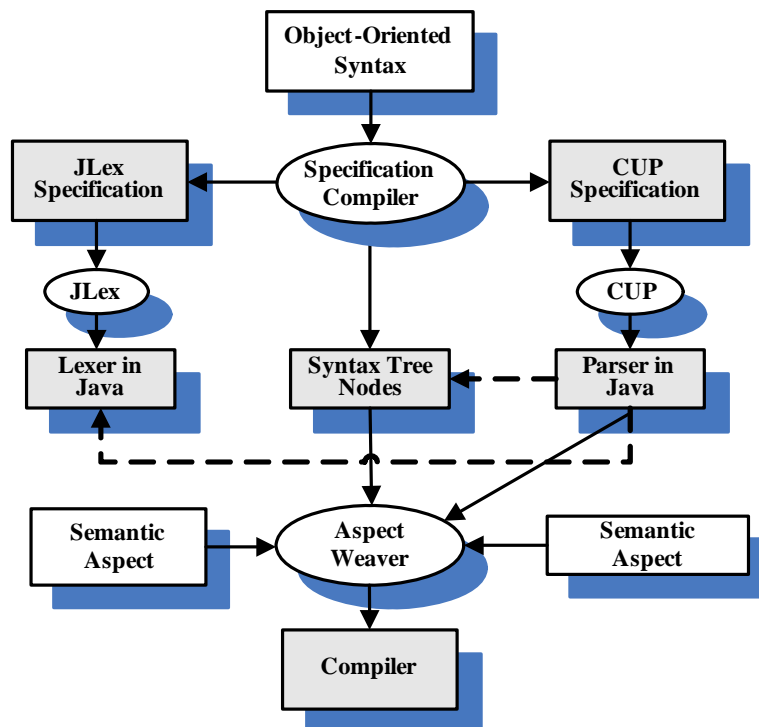


Figure 15: The syntax and semantics implementation framework

## 4.2 Object-Oriented Syntax

In the syntax specification, a language's grammar is composed of terminal, nonterminal, macro and template definitions, with one of the nonterminals acting as the start symbol. In the overall grammar, each symbol appears exactly once as an LHS symbol and it can be optionally typed. The corresponding right side is treated as its definition. Definitions of terminal symbols are specified by regular expressions, whereas all other three definitions are described by OOCFG. Macros and templates are two kinds of syntactic sugar [Wikipedia 2006b] added to the OOCFG specification to facilitate parser and tree generation. Their implementation relies on a preprocessing phase that can expand templates and merge macros to corresponding regular nonterminal definitions and tree creation actions. A simplified syntax grammar of the specification is illustrated below.

```
grammar ::= symbol_definition +;
symbol_definition ::=
    nonterminal_definition |
    terminal_definition |
    macro_definition |
    template_definition;
```

The specification compiler has three argument options to control the generated syntax tree results: `-ntr` (no tree generation, only parser), `-ctr` (only concrete syntax tree), and `-atr` (only abstract syntax tree). By default, no argument indicates that the parser, concrete syntax tree and abstract syntax tree will all be constructed.

### 4.2.1 OOCFG and Concrete Syntax Tree Generation

In this framework, a syntax tree structure is built in an object-oriented way, which means each syntax tree node is represented by a class such that inheritance and

polymorphism can be utilized in the development. However, although syntax tree structure is derived from syntactic grammar, clearly there is no direct mapping between context-free grammar and object-orientation. Therefore, a tree generation system has to provide a certain mapping from the grammar specification to the tree specification. Most object-oriented compiler generation systems model each nonterminal as an abstract superclass and its corresponding RHS alternatives as specialized subclasses. For example, JastAdd uses an additional tree composition specification to indicate this relationship along with the grammar specification. Its specification for a set of statement definitions is illustrated as follows.

```
// Syntax definition
Stmt ::= Block
      | "if" Expr "then" Stmt
      | Id "!=" Exp

// Tree definition
abstract Stmt;
BlockStmt : Stmt ::= Block;
IfStmt : Stmt ::= Expr Stmt;
AssignStmt : Stmt ::= Id Exp;
```

The specification explicitly uses keyword `abstract` to indicate that `Stmt` is a superclass in the AST and explicitly attaches different types to each production to specify the type of node that it is going to generate, which results in a complex specification that requires extra effort to build. Moreover, because most items (i.e., grammar symbols and node class names) co-exist in both specifications, this tends to lead to a system with considerable redundancy. As a result, the tight cohesion between two specifications makes the system hard to maintain.

To address this problem, OOCFG, a restricted form of regular CFG, is created to make uniform the specifications for both syntax definition and concrete tree construction.

This specification can be utilized to describe the syntax in the same manner as using a regular CFG. The difference is that unlike a regular CFG, OOCFG can be directly used to present an object-oriented class hierarchy.

In OOCFG, the restriction applied to regular CFG is that there are only two kinds of patterns that a production can follow, namely, the right-hand side symbols should be listed either in a concatenation form (e.g.,  $A ::= B C D$ , including  $A ::= B$ ) or in an alternation form (e.g.,  $A ::= B \mid C \mid D$ )<sup>10</sup>. Combination of these two forms (e.g.,  $A ::= B \mid C D$ ) is not allowed. This restriction enables an object-oriented relationship between an LHS symbol and RHS symbols. For concatenation, the LHS symbol can be treated as an aggregation of RHS symbols, whereas for alternation, the RHS symbols can be seen as inherited from the LHS symbol.

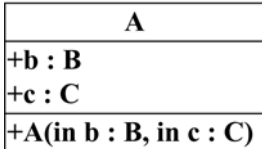
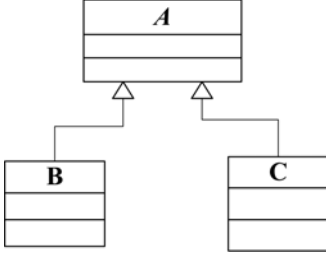
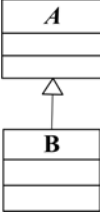
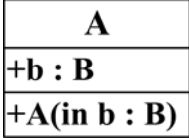
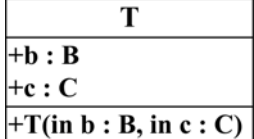
Although OOCFG looks like a subset of regular CFG, it is easy to see that its functionality is still equal to CFG. Any non-OOCFG production can be rewritten in OOCFG. For example, production  $A ::= B C \mid D$  can be rewritten as  $A ::= A1 \mid D$ ;  $A1 ::= B C$ . Note that if the original grammar is a valid LR grammar, this conversion does not change its LR property. It basically only adds an additional reduce action (e.g., reduce  $A1$  to  $A$ ) to the parsing process. Because  $A1$  is a new symbol and is not referenced elsewhere, reducing  $A1$  to  $A$  is the only action to take when  $B C$  is recognized as  $A1$ . Therefore, this addition will not generate reduce-reduce or shift-reduce conflicts with other actions. Overall, without considering syntax tree construction, a parser can be generated from an OOCFG specification in the same manner as from a regular CFG.

---

<sup>10</sup> In both forms, one of the RHS symbol can be the empty token, which is represented as  $\epsilon$ .

In OOCFG, the relationship between an LHS and RHS symbols can be directly mapped into syntax tree node structures. Table 1 summarizes how CUP specification and tree node classes can be generated from each pattern, including some special cases.

Table 1: OOCFG nonterminal production patterns

Production	LHS-RHS Relationship	Generated Cup Specification	Node Class Diagram
$A ::= B C;$	aggregation	$A ::= B : B C : C$ {: Result = new A(B, C);}	
$A ::= B   C;$ (B and C are nonterminals)	inheritance	$A ::= B : B$ {: Result = B; ;}   $C : C$ {: Result = C; ;}	
$A ::= B   \epsilon;$ (B is nonterminal)	inheritance	$A ::= B : B$ {: Result = B; ;}   {: Result = null; ;}	
$A ::= B;$	aggregation	$A ::= B : B$ {: Result = new A(B);}	
$A ::= \epsilon;$	aggregation	$A ::= \{ : \text{Result} = \text{null}; \}$	N/A
$(T)A ::= B C;$	aggregation	$A ::= B : B C : C$ {: Result = new T(B, C);}	
$(T)A ::= B   C;$	N/A	$A ::= B : B \{ : \text{Result} = B; ; \}$   $C : C \{ : \text{Result} = C; ; \}$	N/A

The OOCFG grammar for the same set of statements as previously specified in JastAdd is provided below:

```

Stmt ::= Block | IfStmt | AssignStmt.
IfStmt ::= "if" Exp "then" Stmt.
AssignStmt ::= Id "!=" Exp.

```

Because each nonterminal will occur exactly once as an LHS symbol, each nonterminal can use its own symbol name as a default class type to define the corresponding node (terminal nodes are defined by a pre-defined class `Token`). The type of the node can be either a concrete class or an interface. The LHS symbols whose definitions are in the form of concatenation generate concrete classes for their corresponding syntax tree nodes, with the RHS symbols as their class fields, which are also typed by their own node classes. The class fields are declared as constants by the `final` keyword and are initialized in the constructor. This ensures the tree links to be immutable. The LHS nonterminal that has alternative productions is typed as a superclass with its RHSs as its subclasses. Because a symbol may appear as an RHS alternative in multiple productions, the superclass is implemented as an interface in Java to support multiple inheritance.

If an LHS symbol is prefixed by a type variable `T` and its RHSs are concatenations, its corresponding node will reuse the type `T` instead of creating a new class. Notice that `T` must be either a symbol defined elsewhere in the grammar or a user supplied Java class. On the other hand, if the LHS symbol is typed and the corresponding RHSs are alternatives, they should be the same type as the LHS symbol instead of being its subtypes. This feature is designed to permit reuse of existing classes to avoid

generating too many nodes. It is also commonly used to cast the LHS symbol into a terminal symbol when the RHS is a set of terminal alternatives.

#### 4.2.2 Macro Definitions

Macro definitions are used to rename grammar symbols in syntax definitions. As with nonterminal definition, the LHS of a macro is a unique symbol that can be optionally typed and the RHSs are composed of symbols connected as alternatives or concatenations. Macro definitions generate the same kind of node class hierarchy as nonterminal definitions. The difference between a nonterminal and a macro resides in the CUP specification generation, namely, when a macro symbol is referenced in a production, the preprocessor will replace its occurrence with its RHS counter parts in the syntax specification but keep its original form in the tree creation actions, as demonstrated in Table 2. Therefore, a macro is a symbol that only affects tree construction but does not exist in the parser.

*Table 2: OOCFG macro production patterns*

Pattern #	Production	Generated Cup Specification
1	M = M1 M2; A ::= B M;	A ::= B : B M1 : M1 M2 : M2 {: Result = new A(B, new M(M1, M2));}
2	M = M1   M2; A ::= B M;	A ::= B : B M1: M1 {: Result = new A(B, M1); :}   B : B M2: M2 {: Result = new A(B, M2); :}
3	M = M1 M2; A ::= B   M;	A ::= B : B {: Result = B; :}   M1 : M1 M2 : M2 {: Result = new M(M1, M2); :}
4	M = M1   M2; A ::= B   M;	A ::= B : B {: Result = B; :}   M1 : M1 {: Result = M1; :}   M2 : M2 {: Result = M2; :}

The design rationale for macros is that the grammar used in syntax and tree definition is not necessary to be exactly the same grammar used for parser generations. Because the parser generation process is transparent to language developers, its corresponding grammar can be less readable for resolving conflict and producing efficient parser purposes. The syntax grammar specified by the user can be more high-level and less restricted, which will be translated to the low-level grammar for parser generations. This is analogous to compiling a high-level programming language into target machine code. It improves the readability of the grammar as well as tree notations, and meanwhile keeps the parsing efficient and conflict free. The major usage of macros lies in several aspects, with each associated with a development pattern of using nonterminal and macro definitions.

- **Resolving parsing conflicts.** One of the main usages of macro definitions is to reduce conflicts. For example, there are a number of shift-reduce conflict problems introduced in the JLS grammar [Gosling 1996] and most of them could be solved in OOCFG by using macros. The original solutions provided by the authors are to massage the grammar to eliminate conflicts. For instance, consider a sample partial input and consider the productions for field declaration and method declaration as listed below:

```
// Sample partial input
class C { int foo

// Grammar productions
field_declaration ::= modifiers_opt type
                    variable_declarators ;
method_header ::= modifiers_opt result_type
                 method_declarator throws_opt;
result_type ::= type | void;
```

After the parser reduces the token `int` to `type` and reads the next token `foo`, the parser is not able to tell whether `type` should be further reduced to `result_type` or shift next token in as part of the `variable_declarators`. Hence, a shift-reduce conflict occurs. The solution in [Gosling 1996] is to eliminate the definition of `result_type`. As shown in the listing below, it is clear that the rewritten grammar is not as comprehensive as the original one.

```
// Rewritten productions
field_declaration ::= modifiers_opt type
                    variable_declarators ;
method_header ::= modifiers_opt type
                  method_declarator throws_opt
                  | modifiers_opt void
                  method_declarator throws_opt;
```

By using macros, the conflict problem can be resolved by converting `result_type` into a macro and let the preprocessor do the expansion seamlessly, as in `result_type = type | void;`

- **Allowing the same syntax entity to have multiple roles.** In a syntax grammar of a particular language, it is a normal case that one syntactic entity may have different roles in different productions. For example, an identifier can act as a type-id or a class-id or a variable-id according to the different context. In traditional grammar design methods, the designer has to either simply use the

same variable (e.g., identifier) for all the different roles, or employ some unit productions to rename the same variable. The first approach eliminates the beneficial exposition of the grammar because the same variable name can not reveal different syntactic roles. The second approach increases the complexity of the grammar by adding unnecessary unit-productions, which may generate parsing conflicts. Using macro definitions (e.g., `class_id = identifier`) can eliminate the drawback of both methods introduced above by hiding the parsing grammar details. It enables one syntactic entity to have different representations at the grammar level and in the tree structure, while keeping the same instance in the generated parser.

- **Removing unnecessary reductions.** A side-effect of translating a regular CFG into an OOCFG specification is that a number of unit productions [Hopcroft 2001] are generated. This increases the number of productions resulting in a larger parsing table and more reduce actions, which may hurt the parsing performance. The problem can be addressed by using macros. The development pattern is that, for each nonterminal whose definition is an alternation (i.e., composed by a set of unit productions), rewrite the RHS' definition as macro definitions, as shown in patterns 3 and 4 in Table 2. In this way, all the RHS symbols will be expanded by their own respective RHS symbols for parser generation, where the unit productions are removed. Notice that an exception to this pattern is that, if the RHS symbol itself is also defined by alternation and this symbol is referenced multiple times across the overall grammar, it is better to define this symbol as a nonterminal instead of a macro. For example, in pattern 4, if M is also referenced

elsewhere besides in the definition of A, using M as a macro will increase the overall number of productions, because each reference to M will eliminate the production containing M by introducing two new productions.

There is one restriction of macro definitions. Macro definitions should be acyclic, in other words, a macro symbol should not appear in the derivation path of its own definition.

### 4.2.3 Templates

Syntax grammar usually contains certain patterns among all the productions, but in regular context-free grammar there is no mechanism provided to describe these patterns and simplify the specification. Although some parser generators support EBNF operators on the right side, none of the existing parser generators support user-defined patterns. A template is an abstraction mechanism created to facilitate OOCFG to support generic production definition in a grammar specification. The idea and implementation are borrowed from C++ templates [Vandevoorde 2002], where a preprocessor generates a separate copy of the template definition for each distinct template reference. In C++, templates are designed for function and class definitions, whereas in OOCFG templates are applied to nonterminal or macro definitions.

Figure 16 illustrates some common template definitions that are needed in syntax specification. As seen in this figure, similar to nonterminal and macro symbols, an OOCFG template symbol is also defined by alternatives or concatenations. The difference is that the LHS template symbol contains parameter variables in the form of  $T\langle\#P_1, \#P_2, \dots\rangle$ . Besides other regular symbols that can appear as a production's

RHS, a template definition's RHS symbols can also include the parameter variables (e.g., #P<sub>1</sub>, #P<sub>2</sub>) and other template symbols that contain the same parameter variables (e.g., T<sub>1</sub><#P<sub>1</sub>>, T<sub>2</sub><#P<sub>2</sub>>), as seen in production 3 and production 4 of Figure 16. Notice that the RHS parameter variables can be used in a recursive way, namely, the variable itself can be a template symbol that contains a parameter variable (e.g., T<sub>1</sub>< T<sub>2</sub><#P<sub>2</sub>>>). For example, the second template definition in Figure 16 reuses itself as a template parameter of the first template to describe a plain list structure recursively. Moreover, overloading is allowed in template definitions in the sense that two templates with the same name can co-exist in the same specification as long as they have a different number of arguments. Two template definitions are only considered as duplicated if they have the same template name and same number of arguments.

```

// Optional item
1. opt<#Item> ::= #Item | ε;
// Plain list
2. list<#Item> ::= #Item opt<list<#Item>>;
// List with separators
3. sep_list<#SEP, #Item>
   ::= m_sep_list<#SEP, #Item> | #Item;
4. m_sep_list<#SEP, #Item>
   ::= sep_list<#SEP, #Item> #SEP #Item;

```

*Figure 16: Commonly used templates definitions*

A reference to a template symbol is of the form X<R<sub>1</sub>, R<sub>2</sub>, ...>, where R<sub>i</sub> is a parameter value that can be a terminal, a nonterminal, a macro or even another template reference. Once a template reference occurs, the preprocessor will consult the corresponding template definition to generate related productions by substituting the

parameter variables with the reference values. Because of these generated productions, a template reference can be used just like other nonterminals. The production below shows two sample template references used in a Java grammar.

```
method_header ::= opt<list<modifier>>
                return_type
                method_declarator
                opt<throws>;
```

The productions that the preprocessor generate from the template reference symbols are as follows:

```
opt_list_modifier ::= list_modifier | ε;
list_modifier ::= modifier opt_list_modifier;
opt_throws ::= throws | ε;
```

Note that if the same template reference with the same parameter values occurs more than once, the template definition will be expanded only once.

Templates provide great flexibility for language developers to describe whatever pattern they want to build. Particularly, all EBNF operators can be easily implemented using templates, such as a list or an optional element. Other examples using grammar templates include XML statement definitions such as `xml_markup<#expr><#id> ::= "<" #id ">" #expr "</" #id ">"`.

The consistent substitution rule of templates ensures that the first `id` and the second one are identical. Consequently, a reference such as `xml_markup<float_literal><price>` reduces the redundancy at the grammar level. Moreover, generally it is easy to use a conventional context-free grammar to describe ordered finite lists (e.g., `A ::= B1 B2`) or unordered infinite lists, (e.g., `A ::= B A | ε; B ::= B1 | B2;`), but it is hard to describe those unordered finite lists (e.g., permutation and composition of a set of symbols) unless all the

possibilities are presented. In programming language implementation, sometimes such kind of logic is needed (e.g., field modifiers in Java). In the case that the same pattern is needed multiple times, it will be quite cumbersome to provide them all, which is a good indication that a template is needed. The listing below shows the template definition for the composition of two symbols. This first RHS alternative recognizes  $\epsilon$ , A, B, AB and the second recognizes BA. A template reference like `composition<static, abstract>` not only avoids redundancy in grammar definitions but also is more descriptive.

```
composition<#A, #B> ::= opt_concat<#A, #B>
                    | concat<#B, #A>;
opt_concat <#A, #B> ::= opt<#A> opt<#B>;
concat <#A, #B> ::= A B;
```

By default, a template reference has a corresponding class generated as its node type. However, if the LHS of the template definition is already typed, that type will be used for all the associated template references.

#### 4.2.4 Abstract Syntax Tree Generation

After a syntax tree is generated, various analysis phases (also called clients) will consult this tree structure to deliver their functionality. However, because these clients may serve widely different purposes, it is an intricate problem to have a single tree structure to satisfy all the computation needs. For example, in an Integrated Development Environment (IDE), various tools such as the editor, pretty printer and error reporter will need a concrete syntax tree to retrieve the detailed information of the source program, whereas other clients such as compilation-related semantic analysis will only care about the data values and program logic, in which case an abstract syntax tree is more useful.

Moreover, the first type of clients would prefer a tree that is immutable so that concrete information can be obtained at any time, whereas the second type of client sometime requires transformation and optimization of the tree structure for semantic computation needs. Table 3 summarizes the characteristics of the two types of tree structures.

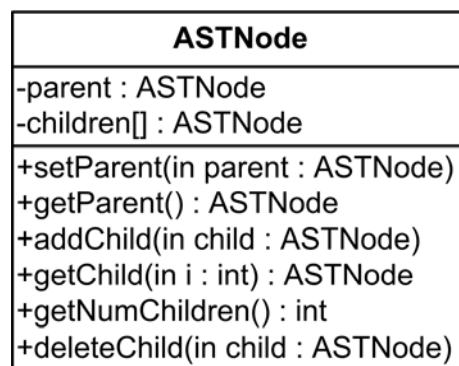
*Table 3: Comparison between abstract syntax tree and concrete syntax tree*

	<b>Concrete Syntax Tree</b>	<b>Abstract Syntax Tree</b>
<b>Co-relation to syntax and semantics</b>	Syntax-oriented	Semantics-oriented
<b>Level of details</b>	Low level syntax details.	High level abstraction
<b>Type of tree nodes</b>	Strongly-typed	Weakly-typed
<b>Type of tree links</b>	Immutable	Programmable
<b>Usage mode</b>	Read-only	Read and write enabled

Traditional language development will select one of the two kinds of trees as the implementation strategy. Because all the clients depend on the same tree, the conflicts of interest among two types of clients usually generate considerable complexity in tree building and semantic phases. For example, after certain clients modify the tree structure, some syntax-related information is no longer available to those clients who need them. This makes it difficult to determine the correct order of the tree traversal and sometimes complicated approaches have to be employed to solve the problem. For example, if a node is no longer available at a certain stage but its attributes are needed, the program has to trace back to the token stream to get the detailed syntactic information.

To address the stated problem, besides the concrete syntax tree introduced in Section 4.2.1, an additional abstract tree is also generated simultaneously based on the same set of tree nodes. For CST generation, the goal is to create a strongly typed and

immutable tree with a one-to-one mapping between grammar symbols and tree nodes, whereas for AST tree generation, the goal is to create a weakly typed, programmable tree structure with a high-level of abstraction. To make this happen, an `ASTNode` class is defined as in Figure 17 and all AST tree nodes will be instances or sub-instances of this class. Each node is able to access its parent or children through methods provided by the `ASTNode` class. Inside the class, the fields are not declared as `final` and there are methods defined to add new children or delete old children, which makes an `ASTNode` instance programmable.



*Figure 17: Class diagram for ASTNode*

In order to reuse the parse tree node to build the initial AST, all concrete syntax tree nodes are required to subclass the `ASTNode` class. Inside their constructors, besides setting their own strongly typed class fields, their child nodes will also be added through the provided APIs of `ASTNode`. As an example, Figure 18 shows the generated node class for the `IfStmt` symbol described in the previous section. Notice that besides the constructor, there is also a `toString` method generated for each node class that returns the string representation of all the tokens enclosed by this node (including its sub-nodes

but excluding all white spaces). This is designed to facilitate reprinting program source from a CST node.

```

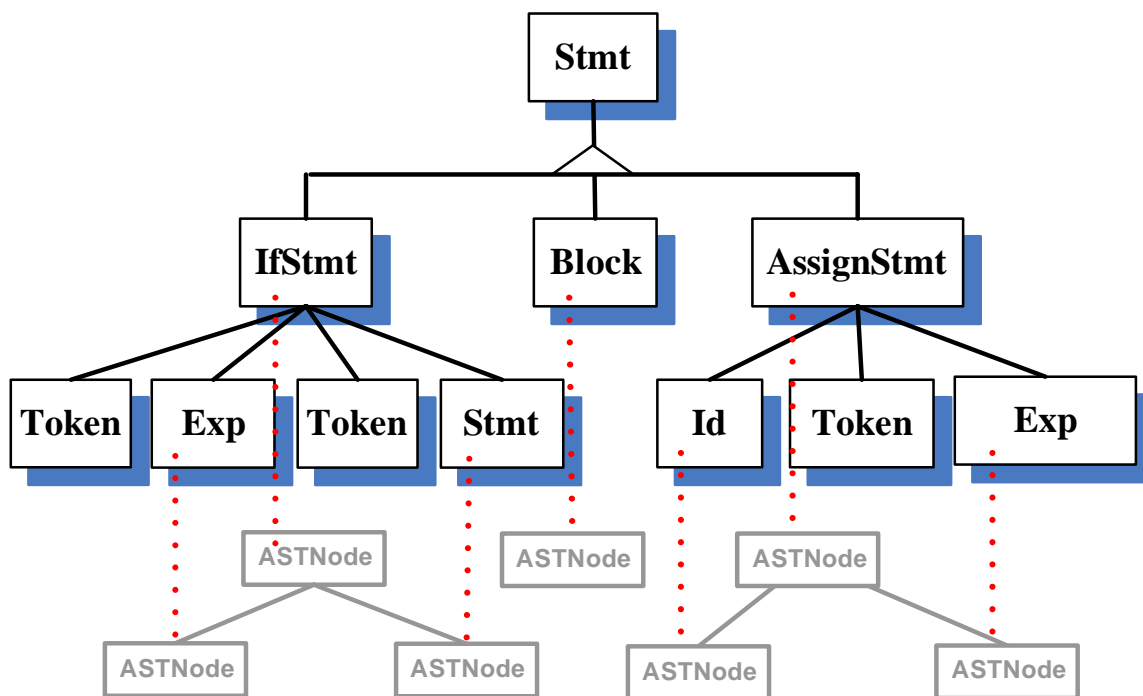
class IfStmt extends ASTNode implements Stmt {
    public final Token _IF;
    public final Exp exp;
    public final Token _THEN;
    public final Stmt stmt;
    public IfStmt (Token _IF, Exp exp, Token _THEN,
                  Stmt stmt) {
        // Concrete syntax tree linkage
        this._IF = _IF;
        this.exp = exp;
        this._THEN = _THEN;
        this.stmt = stmt;
        // Abstract syntax tree linkage
        addChild((ASTNode)exp);
        addChild((ASTNode)stmt);
    }
    // Program reprinting
    public String toString() {
        return _IF + exp + _THEN + stmt;
    }
}

```

*Figure 18: The generated node class for IfStmt*

Because an AST has a higher level of abstraction than a CST, those symbols that carry no semantic values should automatically be neglected in the AST tree building process. This is achieved through specifying those trivial symbols by their quoted lexeme (for terminal only) or prefacing the symbol with an underscore symbol “\_”. Figure 19 illustrates the class hierarchy of the AST tree and the parse tree generated from the specification for the statement related productions. The nodes in gray represent the AST nodes. The initial AST appears to be a floating clone of the parse tree, with less nodes

and weakly typed links. Any AST node itself is also a CST node. It can access its children either by its class fields or by the `GetChild(int)` method defined in `ASTNode`. Gradually, some semantic phases will create new nodes and add them into the AST or they may remove some AST links to existing CST nodes, but the CST structure will be kept intact due to the immutable nature of the alternative CST tree node links.



*Figure 19: AST class hierarchy for statement related productions*

In summary, describing syntax in an object-oriented manner enables a parser and tree structure to be specified by the same specification, which eliminates the redundancy in some parser and tree generators. Using macros and templates further increases the abstraction level of the OOCFG specification by hiding the production details used in real parsing. Generating an additional AST gives different clients flexibility to manipulate the

tree structure to meet their computation needs. Table 4 summarizes various features employed in OOCFG and their corresponding usage.

In addition, the overall syntax specification is purely declarative, which avoids the complexity of mixing formal specification and programming code. The only limitation of this is that user supplied semantic actions are not allowed to insert into parser construction. However, this limitation is fixed by weaving aspect code directly in the tree construction pointcut, which will be introduced in the next section.

*Table 4: OOCFG specification features and their usage*

<b>Specification features</b>	<b>Utilization benefits</b>
Object-oriented grammar definition	Enabling an object-oriented relationship between LHS symbol and RHS symbols, therefore removing the need to provide a separated specification for syntax tree construction.
Typed LHS symbol	Promoting the reuse of existing node classes.
Macros	Reducing parsing conflicts while providing richer description of the grammar and distinct syntax tree nodes.
Templates	Facilitating OOCFG to support generic production definition in a grammar specification.
AST and CST	Providing semantic analysis the flexibility in tree selection and ensuring all analysis needs can be easily computed.

### **4.3 Aspect-Oriented Semantics**

In Chapter 2, it has been discussed that the Visitor pattern, one of the most popular semantic implementation strategies, has several drawbacks due to the fact that OOP is not an ideal orientation to encapsulate functionalities. It was also mentioned that AOP offers an alternative to specify semantics.

In this framework, after the syntax tree is generated as an object-oriented hierarchy, it is natural to explore the effect of AOP on semantic analysis [Mernik 2004], because a semantic phase in compiler design often traverses and crosscuts various AST nodes. In aspect-oriented semantic implementation, each semantic phase can be implemented as an individual aspect. Inside each aspect, the concrete semantic actions of specific nodes can be implemented as inter-type declarations of AST classes, from which the tree iteration logic and common behaviors can be extracted out as join points. The remaining global fields and utility routines are represented as static aspect members.

Our experience results on AspectJ-based semantics implementation show that AOP can significantly improve the separation of concerns in compiler construction [Wu 2005a] [Wu 2005b] [Wu 2006]. As presented in the following sub sections, various language features provided by AspectJ (e.g., pointcuts, advice, inter-type declarations and aspect methods/fields) are well-suited to describe the various analysis needs in compiler design, which cannot be easily achieved by classical object-oriented programming.

#### **4.3.1 Aspect-Oriented Semantics Development in AspectJ**

Aspect-oriented programming is an ideal programming technology to solve the separation of concerns problem in compiler development. By using AOP, all the operations that belong to one semantic phase can be encapsulated as a separated aspect, providing much better modularization and abstraction at the source code level. The aspects can be selectively composed and weaved into AST node classes at compile time, without abandoning the desirable properties of object-orientation such as polymorphism and overloading. The following sub-sections summarize the aspect-oriented semantic

analysis methodology in AspectJ, categorized by the language features that help model the compiler development process.

#### 4.3.1.1 Inter-Type Declarations

AspectJ's Inter-Type Definitions (ITDs) are able to introduce class fields and interfaces into an existing class hierarchy. The semantic artifacts that are represented as methods and fields of AST classes in traditional object-oriented design can be removed to the specific semantic aspect as ITDs, which will be introduced to the AST classes at compile time. Figure 20 shows the corresponding translation from regular Java methods/fields to AspectJ's ITDs. As compared to declaring these operations directly inside the Java classes, AspectJ ITDs have the following benefits:

```

class ClassId {
    Modifiers Type MethodId (Parameters) [MethodBody] [;]
    Modifiers Type FieldId [= Expression];
}
→
aspect AspectId {
    Modifiers Type ClassId.MethodId (Parameters) [MethodBody]
[;]
    Modifiers Type ClassId.FieldId [= Expression];
}

```

*Figure 20: Translation from regular Java class members to AspectJ Inter-type declarations*

- Aspect-orientation can isolate crosscutting semantic behavior in an explicit way. Each semantic segment is encapsulated as one or more physically separated aspects. The semantic code of different phases is not tangled together.

- Each semantic aspect can be freely attached to generate AST nodes without “polluting” the parser or AST node structure. Using tree generation systems such as JJTree and Java Tree Builder (JTB) [JTB 2000], it is a common development practice to add user-supplied semantic code to generated node classes. However, the mixed generated code and handwritten code lead to a system that is hard to maintain and evolve. There are situations when the regenerated code can overwrite the user-supplied code, or the generated code is modified by users accidentally. By using ITDs, all the hand-written code exists as aspects, and hence is visibly separated from the tree classes generated from object-oriented syntax.
- Since each aspect is separated with other aspects, developers can always come back to the previous phase while developing a later phase. As compiler design is a multi-stage process, it is beneficial to have newly added phases independent of existing phases: if an error occurs, it would be easier to narrow the problem scope using aspects; the failure of the new phase will not affect the success of previous phases.
- Different aspects can be selectively plugged in for different purposes. For example, the pretty print operation could be either defined to return a `String` that has the desired code format or defined as a `void` method that prints the formatted code directly. One of these two operations defined as aspects can be selectively plugged in by aspect weaving.

AspectJ’s ITDs can also extend the inheritance hierarchy among classes. This is also very critical in aspect-oriented semantic development, as it provides the ability to

alter the syntax-originated relationship between tree node classes. The syntax of the `declare parents` clause is shown below.

```
// Declare C as the subclass of D.
declare parents : C extends D;

// Declare C implements I and J
declare parents : C implements I, J;
```

The first statement replaces `C`'s original superclass with `D`, with a constraint that `D` must also be a subclass of `C`'s original superclass [AspectJ 2003]. This essentially adds a middle layer (e.g., `D`) between a sub class (e.g., `C`) and a superclass (`C`'s original superclass). By declaring the same class as their parent, multiple node classes can share the semantic operations of the parent class. Considering that all concrete classes by default are `ASTNode`'s subclasses, modifying their relationship satisfies the constraint that a new superclass must be the child of the old superclass. The second statement declares interface inheritance. In aspect-oriented semantics, this can be utilized to group a set of tree classes to facilitate pointcut definition or generic processing, which will be further explored in Section 4.4.

#### 4.3.1.2 Pointcut-Advice Model

AspectJ provides language constructs for defining pointcuts and advice around join points, which enables the same semantic logic to be distributed all over the syntax tree hierarchy. The following areas are some sample facets facilitated by the abstraction power of join-points:

- **Common behavior description.** In semantic analysis, it is a common phenomenon that several tree nodes have the same kind of semantic behavior. If

the behavior is identical, the code can be reused by node inheritance, as introduced in the last section. However, if the behavioral is partially identical, the same piece of code can be encapsulated as an advice, whereas the locations to distribute the code can be described by a pointcut definition. This eliminates the redundancy in the code level compared to traditional programming paradigm. Section 4.4 will show a usage example of this.

- **Tree traversal.** Inside one semantic aspect, tree traversal algorithms are easy to implement with pointcuts and advice, especially for the attribute grammar traversal strategy. For example, the depth-first evaluation algorithm `dfvisit` for L-attributed grammars [Aho 2007] can be directly implemented in AspectJ as shown in Figure 21, provided that the method `synthesized` handles synthesized attribute computations and the method `inherited` handles inherited attribute computations. The main program simply calls the `synthesized` method of the root node to activate whole tree traversal.
- **Phase combination.** Using the pointcut-advice model, each semantic aspect can be glued together with other aspects as one phase, which eliminates unnecessary passes of the AST. One key requirement of the gluing is there should be no traversal conflicts, which means only one aspect should contain the iteration logic, with others containing standalone semantic actions attached to it.

The above items list the benefits of modularizing and weaving code fragments into the AST class hierarchy. The following will further demonstrate the flexibility of using pointcut-advice for weaving codes directly into the parser, which compliments the limitation of OOS.

**Dfvisit algorithm:**

```

procedure dfvisit(n:node);
begin
  for each child m of n, from left to right
  do begin
    evaluate inherited attributes of m;
    dfvisit(m);
  end;
  evaluate synthesized attributes of n
end

```

**AspectJ implementation:**

```

pointcut synAttrCall(Node node):
  target(node) && call (* *.synthesized());
before(Node node):synAttrCall (node){
  Iterator iter = node.getChildren();
  while(iter.hasNext()){
    Node child = (Node)iter.next();
    child.inherited();
    child.synthesized();
  }
}

```

*Figure 21: The Dfvisit algorithm and its AspectJ implementation*

- **Semantic action weaving.** In order to separate imperative semantics with declarative syntax, the object-oriented syntax specification does not allow attaching actions with productions. It is encouraged to describe the actions as an independent iteration over the tree structure. However, recall that for every concatenation production, a new node object is created after the production is reduced. By using this event as a pointcut, any semantic code can be invoked as an after advice, which is functionally equivalent to embedding the semantic actions in syntax specifications used in other parser generators. Describing these semantic actions in an aspect-oriented way supersedes the traditional YACC

approach not only because it clearly separates generated code with user-supplied code, but also because it can describe the same actions for multiple nodes. As illustrated in the listing below, the advice can weave the same actions into the parser once any of `NodeA`, `NodeB` and `NodeC` is created. In parser generators like YACC, this same piece of code has to be specified three times.

```
// The following advice serves as an action in YACC
after(ASTNode n) returning(): target(n) && execution((
    NodeA || NodeB || NodeC).new(..)){
    System.out.println("you can insert action here!");
}
```

- **Parser Tracing.** One of the canonical uses of aspects is to facilitate tracing. For example, it is often required to add print statements during application development to debug certain functionality. Using AspectJ, such tracing or debugging statements can be easily introduced without manually scattering them in several places in the source code. Such a debugging facility is especially useful in compiler development, as a syntax tree usually contains hundreds of node types. It would be quite tedious to put tracing information in every node class to keep track of the node processing sequence. Using the join point model, the construction and traversal information of the syntax tree can be easily specified and displayed. For example, in order to track if the syntax tree is created as expected, the code shown in Figure 22 prints the syntax tree node creation sequence.

```

aspect PrintNodeCreation {
  pointcut construction(Node n): target(n)
    && execution((Node+ && !Node).new(..));
  after(Node n) returning() : construction(n) {
    System.err.println(
      thisJoinPointStaticPart.getSignature().
      getDeclaringType().getName()+"is created");
  }
}

```

*Figure 22: A sample aspect to trace the AST construction process*

#### 4.3.1.3 Aspect Fields and Methods

During one pass of the syntax tree traversal, there are certain fields and methods that need to be shared by all node operations related to this specific phase. This type of field includes constants and accumulated states (e.g., a symbol table). The methods in this category include utility routines or a common processing method that can be reused by various nodes. Within an aspect, these elements can be directly declared as the aspect's own static fields and methods, which are accessible by all ITD methods. Without aspects, the fields or methods must be passed as extra arguments to the semantic operations or they might appear as global elements in non-visitor implementations.

#### 4.3.1.4 Aspect Inheritance

Similar to classes in OOP, aspects are extendable entities. OOP allows new functionality to be added to an object-oriented system without modifying the existing classes. Aspect inheritance further allows new functionality to be added as a separated aspect without modifying the existing aspects. This improves the reusability of aspects. Typically, multiple semantic phases may share several common fields and routines.

These constructs can be defined in a parent aspect that is reusable among other aspects. There are also cases when the implementations are separated with interfaces, where an abstract aspect can be implemented by concrete aspects.

In summary, because AOP focuses on modularizing concerns that crosscut multiple classes, aspects can be helpful in separating the compiler stages that require traversals on multiple AST nodes. Since the semantics of AspectJ is compatible with Java, any compiler written in Java (either using or not using the Visitor pattern) could be reimplemented with AspectJ.

#### **4.3.2 Object-Oriented Visitor Pattern vs. Aspect-Oriented Semantics**

Aspect-oriented semantic implementation is superior to the Visitor pattern because of its unrestricted parameters and return types for different semantic operations, with additional benefits coming from the ability to introduce any field and method into an existing class, as well as increased flexibility in phase integration and tree iteration using pointcut-advice models. As all the semantic actions are introduced to the AST node classes, there is only one class hierarchy created. Therefore, during tree iteration, object-oriented polymorphism is sufficient to determine which node's processing method should be invoked at runtime. Explicit double-dispatch is removed and the `accept` methods defined in each node class are no longer needed, which makes the AST nodes totally oblivious to semantic operations. A visual comparison between operation redirection in the Visitor pattern and the aspect weaving in AOP implementation is illustrated in Figure 23.

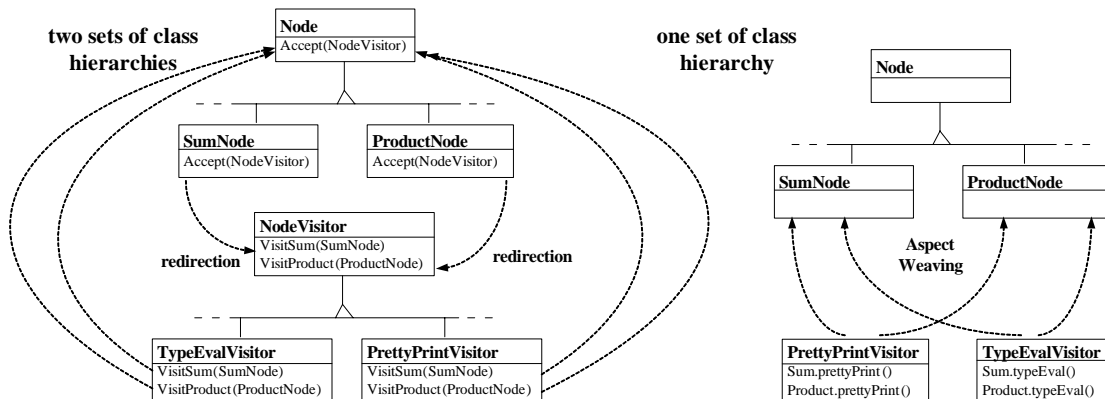


Figure 23: Operation redirection in the Visitor pattern vs. aspect weaving in AOP implementation

## 4.4 Case Study

The overall OOS + AOS methodology has been utilized to implement the Java language. The syntax grammar is based on the same JLS grammar used in Section 3.3. A parser and an object-oriented tree structure are generated from the OOCFG specification, based upon which name analysis and pretty print of the Java language are developed with AspectJ. This example demonstrates how object-oriented syntax can nicely work together with aspect-oriented semantics.

### 4.4.1 Syntax Analysis Phase

For demonstration purposes, the syntax analysis of the Java language is developed by four steps, each of which results in a different version of the grammar named as JLS1, JLS2, JLS3 and JLS4, respectively. JLS1 is the original JLS grammar provided in [Gosling 1996]. JLS2 converts JLS1 into OOCFG. JLS3 adds template definitions to

JLS2 and rewrites a number of productions using template references. JLS4 further replaces some nonterminal definitions with macros.

Table 5 shows the analysis data about each grammar. The first three rows illustrate the properties of the OOCFG specification, whereas the last three rows reveal the characteristics of the generated CUP parser. Notice that the number of terminal definitions is 102, which is a constant number throughout four types of grammars. The detailed description of each conversion phase is presented below.

*Table 5: Analysis data of JLS grammars*

		<b>JLS1</b>	<b>JLS2</b>	<b>JLS3</b>	<b>JLS4</b>
<b>OOCFG Specification</b>	Nonterminal Definitions	156	274	212	99
	Template Definitions	0	0	7	7
	Macro Definitions	0	0	0	113
<b>Parser</b>	Nonterminals	156	274	274	151
	Productions	359	477	477	354
	States	614	732	732	609

- **JLS1 to JLS2.** This step is straightforward. Any illegal OOCFG nonterminal definition is converted into the object-oriented form, while the rest are kept the same. A nonterminal definition is considered as illegal if the following condition is met: its RHS is in the alternative form and at least one of the alternatives contains more than one symbol. The conversion process was done automatically by an OOCFG converter. The conversion rule is that, for any alternative that have more than one symbol, replace it by a new nonterminal that is named by its LHS

symbol and a unique number; the original RHS alternative is specified as the definition of this new nonterminal. For example, nonterminal definition  $A ::= B C \mid D E$ ; can be rewritten as  $A ::= A1 \mid A2$ ;  $A1 ::= B C$ ;  $A2 ::= D E$ ; Following this rule, totally 118 new nonterminals are created in JLS2, each is attributed to a new production and a new parsing state.

- **JLS2 to JLS3.** There are plenty of optional and list structures used in the JLS grammar, which are all abstracted as template definitions in JLS3. With these templates, the original definitions of the optional and list structures are removed from the OOCFG specification. As a result, the number of nonterminal definitions in JLS3 is reduced to 212. Consequently, template references are utilized intensively in JLS3, below is a typical example, which uses three template references and two of them use recursive references. Notice that there are two template definitions for optional nonterminals (i.e.,  $\text{opt}\langle\#P\rangle$  and  $\text{opt}\langle\#P1, \#P2\rangle$ ). One takes a single argument and the other takes two.

```
class_declaration ::=
    opt<list<modifier>> CLASS IDENTIFIER
    opt<EXTENDS, class_type>
    opt<IMPLEMENTS, sep_list<COMMA, interface_type>>
    java_class_body;
```

Although templates reduce the number of nonterminal definitions in the specification level, the generated parser of JLS3 still contains the same number of symbols, productions and states as the JLS2 grammar.

- **JLS3 to JLS4.** It has been illustrated in Section 4.2.2 that utilizing macros can benefit syntax analysis in several ways. In JLS4, JLS3 is rewritten with 113 macro definitions, which resolves all conflicts mentioned in Section 4.2.2 and

reduces unnecessary unit productions. Consequently, the generated parser contains much fewer productions and states than JLS3.

Overall, the syntax definition of the Java language is achieved by converting the original JLS grammar into OOCFG and employing template and macro definitions inside it. Compared to the original grammar (i.e., JLS1), the final grammar (i.e., JLS4) is conflict free and it contains slightly fewer states and symbols in the generated parser. Most important, JLS4 can be used directly to generate a CST class hierarchy and a weakly typed AST, based on which the name analysis and pretty print analysis can be developed in an aspect-oriented way.

#### **4.4.2 Name Analysis Phase**

The implementation of the name analysis phase was a two-step development process: first, loading all the declared types and variables into a symbol table structure; second, evaluating all the referenced type identifiers and variable identifiers against the symbol tables to see if they are declared somewhere. Since Java allows forwarding references, which means a type can be referenced before it is defined, the symbol loading requires a full pass over the program preceding the verification phase. One straightforward way to develop the symbol loading phase is that, using inter-type declarations to add to every node class a default `loadSymbolTable` method containing traversing logic and symbol table actions. However, it can be observed that symbol table loading is not an aspect that needs all the nodes to participate. Introducing a `loadSymbolTable` method to every node and iteratively invoking all of them is too much of an effort. Ideally, only those related nodes (i.e., those nodes that contain a scope)

should define a corresponding method and be visited. However, since those related nodes are scattered all over the tree and consume arbitrary linkage relations, visiting them in a generic way is not an easy job in the traditional object-oriented paradigm. Benefiting from the power of AspectJ's introduction construct, this could be achieved easily with the help of a `declare parents` clause.

As seen in the upper part of Figure 24, an interface `ScopeType` with an abstract method `loadSymbolTable` is defined. All the scope related nodes can implement this interface by using the `declare parents` clause and specify their own `loadSymbolTable` semantics by introducing methods. At this point, the symbol loading process can be easily implemented by the traversing logic defined in the static aspect method `loadSymbols`, provided that the main program will call this function with the root node as the input parameter. Note that the traversing is applied to the AST, which avoids the processing of semantically irrelevant nodes.

The above implementation explores how ITDs can provide great flexibility in semantic analysis. In the following the modularization power of the pointcut-advice model will be further illustrated. During the development of the symbol loading aspect, it has been observed that the same type of actions repeatedly occur whenever entering and leaving a definition scope (e.g., a class, a function or a `for` statement). Specifically, once entering a new scope, the current symbol table should be pushed onto a global stack and a new table will be created and linked to the previous table. After leaving the scope, the previous table should be popped from the stack and serve as the executing table. To avoid duplicating the same piece of code multiple times, the two pieces of code are specified as two pieces of advice. As the code shown in the lower part of Figure 24, the advice is

applied before and after a pointcut called `scopeEvaluate`. The variables `symTabs` and `currentSymTab` are two static aspect members acting as global states that can be accessed by all syntax tree nodes.

```

interface ScopeType {
    public void loadSymbolTable();
}
public static void loadSymbols(ASTNode node){
    if (node == null || node.isLeave()) return;
    else if (node instanceof ScopeType) {
        ((ScopeType)node).loadSymbolTable();
    }
    for(int i = 0; i < node.getNumChildren(); i++){
        loadSymbols(node.getChild(i));
    }
}
declare parents : Method_declaration implements ScopeType;
public void Method_declaration.loadSymbolTable(){
    // ...
}
declare parents : Class_declaration implements ScopeType;
public void Class_declaration.loadSymbolTable(){
    // ...
}

pointcut scopeEvaluate():
    call (* ScopeType.loadSymbolTable()) ;
before() : scopeEvaluate(){
    symTabs.push(currentSymTab);
    SymbolTable tmp = currentSymTab;
    currentSymTab = new SymbolTable();
    currentSymTab.parentScope = tmp;
}
after() : scopeEvaluate(){
    currentSymTab = (SymbolTable)symTabs.pop();
}
// ... ITDs for other scope related nodes

```

Figure 24: AspectJ code for symbol table loading phase

This example shows the abstraction power of aspect-oriented semantic implementation. In our previous compiler implementation of a Cobol-like commercial language, it has been shown that the code specified in the above two pieces of advice occurred 46 times in its YACC specification.

Once the symbols are all loaded in the tables, the second step is to verify if each referenced type identifier or variable identifier has been stored in the symbol tables, which requires a separate traversal. Similar to the symbol loading aspect, this is achieved by combining the use of aspect introduction and join point models. Particularly, when entering or leaving a scope, the corresponding symbol table should be switched. This logic is again implemented by pointcuts and advice.

#### **4.4.3 Pretty Print Phase**

The pretty print aspect was built based on the `toString` methods declared in each CST node class. The proper formatting of the code is achieved by inserting white spaces between the tokens.

In the object-oriented paradigm, the implementation of pretty print would be a typical example of the Visitor pattern. Each node will have a visiting method defined to specify how the node's underlying tokens should be printed. Once every node is visited, the whole program is properly formatted. However, a deep analysis of the most common formatting guidelines reveal that pretty print is not necessarily analyzed in a node by node manner. Instead, the formatting guidelines are composed of several patterns that are applied to a set of nodes, with a few of them listed as follows:

- Increasing the number of indents whenever entering a new block; decreasing it whenever leaving a block.
- Inserting a new line character at the end of a statement or any kind of declaration (e.g., `package`, `import`, `class`, etc.).
- A blank space should be placed after seeing a keyword such as `if`, `for` and `switch`.

These regulations are clearly crosscutting behaviors. As a result, the whole pretty print aspect can be implemented by a set of well-defined pointcut-advice models. Appendix B shows how the formatting rules were implemented by pointcuts and advice. The `around` keyword is utilized there to insert a new line character, a blank space and proper indents to the plain text that a `toString` method returns.

As shown in this example, by using AOP's dynamic weaving feature, specifications are directly mapped into crosscutting aspects, which is more precise and manageable. As a contrast, the same pretty print functionality was implemented as one thousand lines of code in the JastAdd specification [Hedin 2005], which only allows static introduction behaviors.

With the aspects described beforehand, the name analysis and pretty print functionality of the Java language was fully implemented without manually modifying the generated syntax tree classes. Because the extension to the compiler is achieved using aspects, the compiler can always return to its original state (e.g., only syntax analysis) by not weaving the new aspects. As the compiler evolves, additional semantic functionality can be added as separated aspects in the same manner, without any change in the existing code.

The OOS + AOS implementation of the whole Java language is available at <http://www.cis.uab.edu/wuxi/research/>.

#### **4.5 Integration with CLR Parsing**

Merging component-based LR parsing with OOS+AOS is straightforward. For syntax specification, the restrictions of OOCFG can be applied to CCFG without generating any side-effects; macro and template definitions are able to be added into CLR directly; import and export declarations are extended to accept template references, but macros cannot be exported or imported because their symbols do not exist in real parsers and hence cannot serve as the start symbol.

Figure 25 provides an illustration of the whole framework after CLR is merged with OO syntax and AO semantics.

For syntax tree construction, CLR's parse tree generation process is inlined with OOCFG tree generation, so the OOCFG tree construction strategy can be directly used in CLR. As discussed in Section 3.4.2, each component generates its own tree structure and small parse trees will be ultimately connected into a single one after the parsing is finished.

For semantic analysis, once the syntax tree node classes are generated into different packages, semantic aspects can be supplied to deliver corresponding functionalities. The aspects can be defined either as "global" aspects (the ones that crosscut all syntax tree nodes) or "local" aspects (the ones that are only weaved into a particular language component).

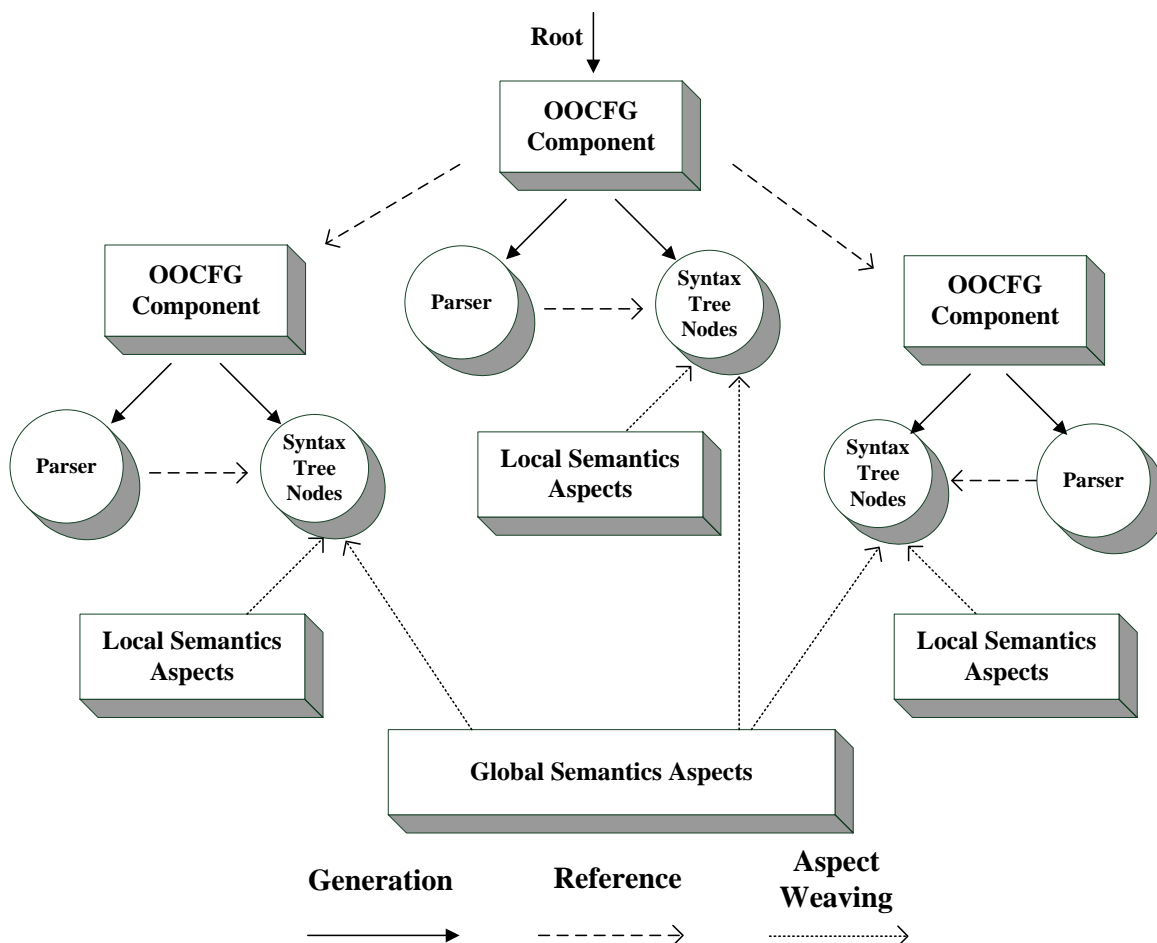


Figure 25: The integrated framework overview

There is one limitation of applying AO semantics in component-based development, namely, an aspect should not be directly applied on parser reduce actions as presented in Section 4.3.2.2. This is due to the fact that backtracking is employed in component-based LR parsing, where a reduction does not necessarily mean that path is the correct one. In this case, a semantic action immediately following a reduce action can generate undesirable side-effects. Therefore, when CLR parsing and AOS are used

together, all semantic analysis must be performed in the form of tree traversing logic after correctly building up the tree.

#### **4.6 Summary**

This chapter has presented object-oriented syntax and aspect-oriented semantics construction, which yields a clear separation of concerns between syntax and semantics as well as among semantic phases themselves, thereby improving the overall modularization of the system and providing flexibility for future evolution of the compiler. The object-oriented syntax specification seamlessly integrates parser and tree construction into a single description. The employed macro and template definitions further increase the abstraction level of the overall specification and hide the implementation details. The aspect-oriented semantics supersede the object-oriented Visitor pattern by its unrestricted method definitions and transparent nature to node classes, as well as the flexibility in phase integration and tree walking using join points.

The framework presents syntax and semantics separately by formal specifications and programming languages, which properly matches the needs of each phase. The generated code is totally separated from user-supplied code, which avoids the problems caused by mixing them together.

Therefore, OOS + AOS successfully solve the second modularity problem in language implementation. By integrating OOS + AOS with the CLR parsing algorithm, the modularity problems mentioned at the beginning of this thesis are fully addressed in a two-dimensional manor (structure-wise and functional-wise).

## CHAPTER 5

### FRAMEWORK APPLICATIONS

In this chapter, in order to further illuminate the benefits derived from applying the framework defined in this thesis to language development, six case studies besides the Java language implementation will be discussed. Each case study either entirely or partially uses OOS, AOS or CLR as the key techniques in the language implementation process. Many individual ideas of the framework have been experimented with in these case studies.

For most case studies presented in this chapter, their source code, testing programs and demos are available at <http://www.cis.uab.edu/wuxi/research/>.

#### **5.1 Pam and BasicM**

OOCFG and its generation system have been utilized to prototype Pam (an Algol-like language) [Pagan 1981]. Both the parser and tree structure were automatically generated from the OOCFG compiler, with the semantics written as node class methods in a pure object-oriented way. The whole implementation was later successfully reused to develop a language for matrix calculation called BasicM [Wu 2004], which essentially replaces integers by matrices as the first class elements of the language. Figure 26 shows the output of a sample BasicM program.

Matrix	Row 1	Row 2	Row 3	Row 4	
Matrix: E =	1	0	0	0	
	-1.5	1	0	0	
	-1	0	1	0	
	-1	0	0	1	
Matrix: E~ =	1	-0	0	-0	
	1.5	1	-0	0	
	1	-0	1	-0	
	1	0	-0	1	
Matrix: E.(E~) =	1	0	0	0	
	0	1	0	0	
	0	0	1	0	
	0	0	0	1	
Matrix: E* =	1	-0	0	-0	
	1.5	1	-0	0	
	1	-0	1	-0	
	1	0	-0	1	
Matrix: (E)' =	1	0	0	0	
	-1.5	1	0	0	
	-1	0	1	0	
	-1	0	0	1	
Matrix: (E~)~ =	1	-0	0	-0	
	-1.5	1	-0	0	
	-1	-0	1	-0	
	-1	0	-0	1	
Matrix: G =	-0.91014	-0	0	-0	0
	-0	10.3389	-0	-0	0
	0	-0	-10.01...	0	-0
	-0	-0	0	-3.22931	-0
	0	0	-0	-0	6.29258

Figure 26: Output of a sample BasicM program

The experience shows that OOCFG can significantly reduce the development cycle of syntax analysis and tree construction. The grammar of Pam is given as an LALR(1) grammar, so it took little time to convert it into OOCFG and produce a parser and a set of tree nodes from it. The BasicM language was implemented in three days by a single developer, with 30% of the time to design the language, 10% of the time to write the grammar (including parser and tree generation) and 60% of the time to describe the semantics. The only negative side is that, because the semantics were directly implemented in the generated node classes, they will be removed each time the tree structure is regenerated

as a result of syntax extension. This problem was avoided in later case studies that employ aspect-orientation in semantic implementations.

This case study justifies that OOS provides efficient implementation of a language by automating the parser and tree construction processes directly from syntax definitions. Furthermore, it shows that OOS supports good extensibility in language description. It also demonstrated that an object-oriented implementation of semantics is not inlined with automatic tree generation and an AOS-like approach is needed.

## 5.2 RelationJava

The aspect-oriented semantics have been utilized to implement a language called RelationJava, which provides *relations* (methods that can access private members of multiple classes) as first-class computation elements along with Java classes and interfaces [Wu 2006]. The nature of the language is irrelevant with the topic, but it can help to understand how the compiler is built. Essentially, a relation is an external method owned by multiple interfaces/classes and it can use fields and methods of those interfaces/classes directly. The language was implemented by compiling it into regular Java and the key part was to translate a relation into a static Java method defined in a default class. The compiler implementation reused an existing JavaCC + JJTree sample program that can take a Java program and reproduce its source code by using an unparse visitor<sup>11</sup>. Consequently, the RelationJava compiler can be achieved based on this by providing additional syntax tree nodes and their translation rules for a relation definition.

---

<sup>11</sup> Available at <https://javacc.dev.java.net/source/browse/javacc/examples/VTransformer/UnparseVisitor.java>.

The implementation of the compiler was a two-step development process: first, in order to facilitate further aspect-oriented development, the `UnparseVisitor` class was rewritten as an `Unparse` aspect. Then, the `unparse` methods of new nodes were introduced in a `Translation` aspect derived from the `Unparse` aspect to translate each relation locally into a static Java method, with related class names converted to method parameters. The key significance shown in this practice occurs in the process of converting the unparse visitor into an unparse aspect. The original JJTree implementation contains 500 lines of handwritten code for the visiting methods and 90 automatically generated `accept` methods inside each AST node, as demonstrated in Figure 27. The contents of each visiting method are identical, namely, returning the call results of a `print` method that takes a tree node as a parameter.

```

class UnparseVisitor {
    protected PrintStream out = System.out;
    public Object print(SimpleNode node, Object data){
        // ...
    }
    // Other utility routines
    // ...
    public Object visit(SimpleNode node, Object data){
        return print(node, data);
    }
    public Object visit(CompilationUnit node, Object data){
        return print(node, data);
    }
    // same visit methods for another 83 nodes.
    // ...
}

```

*Figure 27: The UnparseVisitor class*

The reason that the Visitor pattern has so much redundancy in this case is that the abstract visitor, which is necessary for double-dispatch, forces each concrete visitor to implement a visiting method for every single node, even if their contents are identical. However, in the aspect-oriented implementation, because all the semantic operations can be naturally weaved into the class hierarchy, there is no need to define the generic abstract visitor. After rewriting the `UnparseVisitor` class as an `Unparse` aspect (demonstrated in Figure 28), *all of those redundancies (500 lines of code) are removed by a three line ITD in AspectJ* [Wu 2006].

```

aspect Unparse{
    protected static PrintStream out = System.out;
    public static void print(SimpleNode node){
        // ...
    }

    // other utility routines
    // ...
    public void SimpleNode.unparse(){
        Unparse.print(this);
    }
}

```

*Figure 28: The Unparse aspect*

In summary, this case study reveals that the AOS implementation built on top of an object-oriented syntax tree can clearly encapsulate each phase of semantic analysis as an aspect and offers the flexibility in semantic description. It also shows that AOS supercedes the widely used Visitor pattern by its straightforward implementation and simplified class hierarchy.

### 5.3 OOCFG Converter

The OOS and AOS combination has been utilized to build an OOCFG converter, which facilitates translating a regular CFG into OOCFG. Besides the basic translation rule introduced in Section 4.4, this converter can also process CUP-style specifications where an RHS symbol can be optionally labeled (e.g., `expression : expr`). The converter is implemented by one OOS definition and one translation aspect.

Figure 29 shows the nonterminal and macro definitions of this simple translator. The use of template references again significantly reduces the number of definitions (from 15 definitions to 9 definitions). The same syntax entity ID acts as three different roles in the syntax grammar, so three macros (`nt_id`, `symbol_id`, `label_id`) are defined to satisfy this need. These macro symbols will be generated as three different classes in the syntax tree structure, but their occurrences in the parsing part are replaced by the same terminal ID.

```

cfg ::= list<production>;
production ::= nt_id COLON_COLON_EQUALS sep_list<rhs> SEMI;
rhs ::= list<prod_part> | ;
prod_part ::= symbol_id opt_label;
opt_label ::= label | ;
label ::= COLON label_id;
nt_id = ID;
symbol_id = ID;
label_id = ID;

```

*Figure 29: Nonterminal and macro definitions of the OOCFG converter*

Figure 30 shows the translation aspect. The `toString` methods are used as the base of transformation, with two pointcut-advice pairs defined to adjust the output. In case a production is of the form  $A ::= B \ C \mid D$ , one advice can replace  $B \ C$  by  $A_n$  ( $n = 1, 2, \dots$ ) as the output of the `rhs` node and stores the new production  $A_n ::= B \ C$  into a global string variable. The other advice adds this global string into the output and clears its contents after a `production` node is printed.

In summary, this case study proves that OOS and AOS can coherently work together to describe syntax and semantics respectively. The macros and templates used in syntax specification provide utilities to resolve parsing conflicts and specify syntax precisely, while at the same time keeping the tree structure more comprehensible.

```

aspect Translation {
  String around():
    execution(String Production.toString()){
      Global.lhsName = nt_id.id.toString();
      Global.rhsCount = 0;
      Global.newProductions = new String("");
      Global.multiRhs = (rhs_list instanceof Rhss);
      String rslt = nt_id + colon_colon_equals +
                    rhs_list + semi;
      // Add the new production to the output
      rslt += Global.newProductions;
      // Clear the contents
      Global.newProductions = new String("");
      return rslt;
    }

  String around() :
    execution(String Rhs.toString()) {
      String str = prod_part_list == null ?
                  new String("") :
                  prod_part_list.toString();
      if (Global.multiRhs &&
          prod_part_list instanceof Prod_parts) {
        // Add a new production in the form of
        // A1 ::= B C;
        Global.rhsCount++;
        String newName = Global.lhsName +
                          Global.rhsCount;
        if (Global.rhsCount == 1) {
          Global.newProductions += "\n";
        }
        Global.newProductions +=
          newName + " ::= " + str + ";\n";
        return "\t" + newName;
      }
      else {
        return str;
      }
    }
}

```

*Figure 30: The translation aspect in OOCFG converter*

#### **5.4 Bootstrap Implementation**

The OOS and AOS combination has also been used to bootstrap the whole OOCFG component generator (i.e., the CLR parser generator integrated with the OOCFG compiler). The overall syntax specification is provided in Appendix C. The semantics are implemented by six aspects. The first aspect has a user-defined class and uses ITDs to introduce this class as the common super class for nonterminal, macro and template definition nodes. It is designed in such a way to enable code reuse among the three nodes as they share many common semantics. To facilitate forward references, one aspect is built to create symbol table entries before the table is loaded and a second aspect loads all the necessary information into the symbol tables and provides static checking. The last three aspects are utilized to generate the lexer, parser and tree node classes respectively. The whole compilation is built upon an AST instead of a CST due to tree transformation required by template references. In the first aspect, whenever a new template reference node is created, a new nonterminal or macro definition node will be created and added to the tree structure. All of the following phases will work on this AST to deliver the required logic.

This case study again confirms that the incorporation of AOS and OOS can provide high-level of modularity for syntax and semantic descriptions. Particularly, it exposes the programming need of using an AST in compiler construction.

#### **5.5 Google Query Language**

The case study that completely employs all of OOS, AOS and CLR is the implementation of a language called Google Query Language (GQL), which is a domain-

specific language developed to provide a user-friendly facility to support advanced Google search [Wu 2005b]. GQL supports advanced search for several different domains such as general web search and image search. Each of these domains has its own set of constraints to narrow down the search results, as shown in Table 6. For example, the keyword constraint is supplied by all the search domains, whereas the language constraint is only supported by web search and group search.

*Table 6: Specific features in each advanced Google search domain*

<b>Search Features</b>	<b>Web Search</b>	<b>Image Search</b>	<b>Group Search</b>	<b>News Search</b>
<b>Keyword</b>	√	√	√	√
<b>Language</b>	√		√	
<b>Filetypes</b>	√	√		
<b>Date</b>	√		√	√
<b>Occurrences</b>	√			√
<b>Domain</b>	√	√		
<b>SafeSearch</b>	√			
<b>Size</b>		√		
<b>Coloration</b>		√		
<b>SafeSearch</b>		√	√	
<b>Group</b>			√	
<b>Subject</b>			√	
<b>Author</b>			√	
<b>News source</b>				√
<b>Location</b>				√

Using component-based development, it is a straightforward idea to decompose the GQL as various query languages in different domains (e.g., web\_ql, img\_ql), which

share a number of constraints in common. Because each constraint is syntactically represented by a group of productions, it can be modularized as a small language component that could be developed and tested independently. Consequently, one domain-specific search language can be developed simply by plugging and playing these constraint components together. As a result, 15 constraint components are developed, which further compose into four query languages. Figure 31 shows the syntax specification for WebQL. The union of the four query languages is GQL, whose specification is shown in Figure 32.

```

language web_ql;

import keyword, occurrence, file_format, language, date,
domain, safe;

web_ql ::= SEARCHTYPE list<keyword><COMMA>
          opt<IN><occurrence>
          WHERE list<constraint>;
constraint = file_format | language | date | domain | safe;

SEARCHTYPE "web";
COMMA ",";
IN "in";

```

*Figure 31: Syntax specification for WebQL*

```

language GQL;
import web_ql, img_ql, news_ql, group_ql;
GQL ::= web_ql | img_ql | news_ql | group_ql;

```

*Figure 32: Syntax specification for GQL*

A domain-specific language is always desirable to be extendable. One possible direction of GQL is to allow recursive queries (i.e., one search query can be built on top of other queries to make it easy to read). To implement this feature, there is an important restriction to be considered: if query A is built on top of query B, then A and B should be in the same domain, in other words, it is not allowed to have a web query to reuse the search result of an image query because it will simply return nothing. To specify such syntactic requirements in normal CFG, at least two productions should be added for every domain-specific query language. For example, the following productions should be added to extend WebQL:

```
RecursiveGQL ::= RecursiveWebQL.
RecursiveWebQL ::= WebQL "within" WebQL.
```

The number of productions that should be added to GQL will be multiplied by N when there are N domain-specific query languages. However, by using template definition, the problem could be done with ease. Figure 33 shows the recursive GQL specification in OOCFG.

```
language recursive_gql;
import gql, web_ql, img_ql, group_ql, news_ql;
recursive<#Item> ::= #Item WITHIN #Item;
recursive_gql ::= gql recursive<web_ql> | recursive<img_ql>
| recursive<news_ql> | recursive<group_ql>;
WITHIN "within";
```

*Figure 33: Syntax specification for recursive GQL*

The semantics of GQL contain two phases: static checking and generating Google recognizable search tokens. The code generation pass is done by an independent traversal

of the syntax tree, which was implemented as an aspect. Based on this aspect, a pointcut-advice model is utilized to invoke static checking of each tree node before the translation method being called, which avoids traversing the tree twice [Wu 2005b].

Overall, like the Java language example, the GQL example above exhibits how component-based language development can provide appropriate modularity such that the workload of developing a large language is decomposed into developing a number of smaller languages. It also comprehensively explores how different pieces of this framework (e.g., CLR, macro and templates, ITDs and pointcut-advice) can work together to enhance the reusability and changeability of the overall language specification.

## **5.6 Robot Language**

For demonstration purpose, a toy language has been developed to precisely present the flexibility of using the framework in language implementation. The language is used to control the actions of a robot. A robot can move in four directions from the initial position (0,0) and occasionally it can speak some simple sentences. An example of the program written in this language is `begin up right up right hello right down end.`

For better modularity, the language is decomposed into two components: a main component that controls the moving of a robot and a child component that manages the sentences that a robot can speak. The two components can be developed separately by different developers.

The syntax of the Robot component is shown in Figure 34. Initially, since the Speak component is not developed yet, a dummy component with the following syntax will be used as a place holder.

```

language speak;
export program;
program ::= HELLO;
HELLO <<"hello">>;

```

With the provided syntax specification, a parser can be generated and a user-supplied driver can invoke the parser to successfully parse the sample program mentioned beforehand.

```

language robot;
import speak.program as speak;
export program;

program ::= BEGIN list<robot_action> END;
robot_action ::= move | speak;
move ::= move_left | move_right | move_up | move_down;

move_left ::= LEFT;
move_right ::= RIGHT;
move_up ::= UP;
move_down ::= DOWN;

BEGIN <<"begin">>;
END <<"end">>;
LEFT <<"left">>;
RIGHT <<"right">>;
UP <<"up">>;
DOWN <<"down">>;

```

*Figure 34: The syntax of the Robot language*

For semantic analysis, each generated syntax tree node class will have a `doAction` method defined as an AspectJ ITD. The `doAction` methods recursively invoke one another to traverse the whole syntax tree and produce a graphic output demonstrating a robot's movements. Notice that, for the dummy speak component, the corresponding ITDs will simply be empty methods.

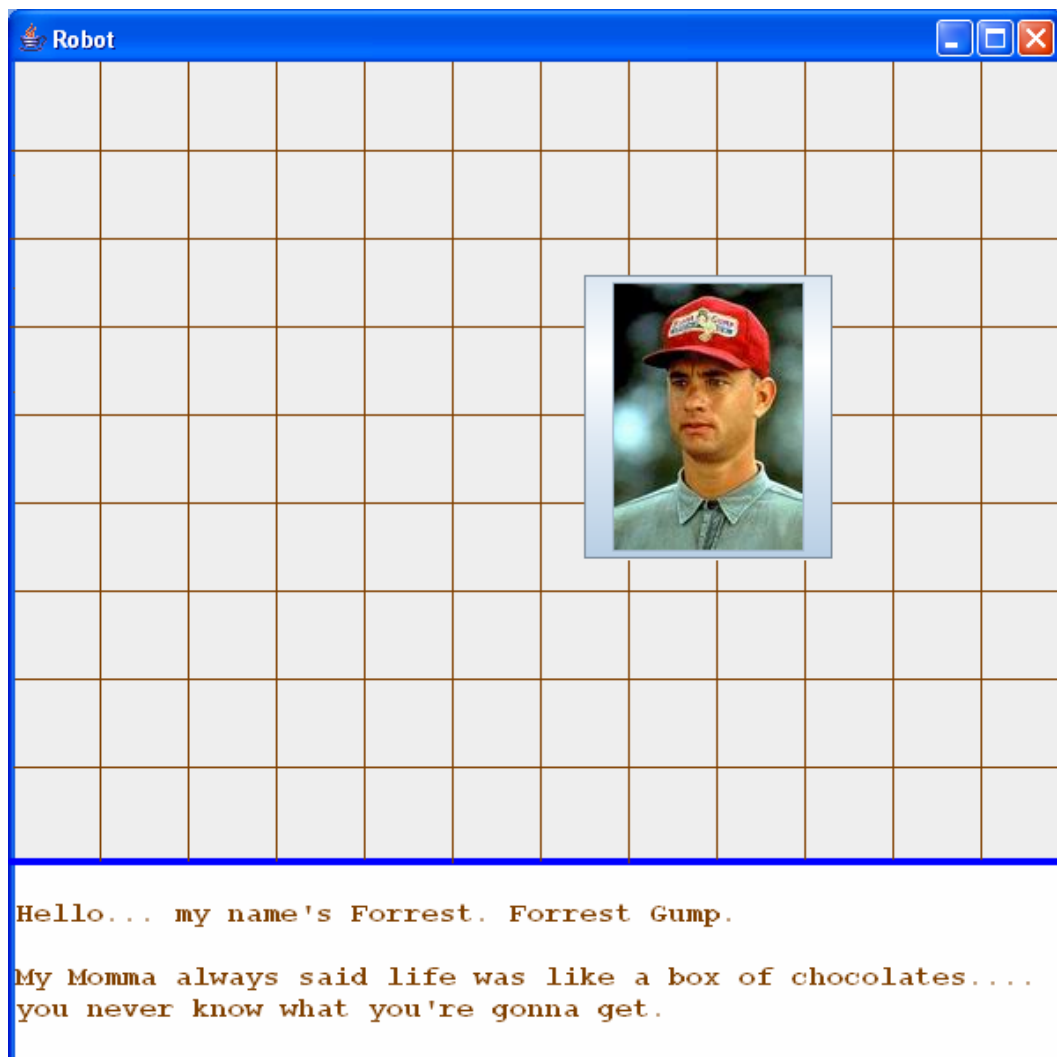
After the semantics are fully described as ITDs, the following aspect can be utilized to invoke the `doAction` methods immediately after the root node is constructed. In this way, the semantic analysis is seamlessly glued with the syntax analysis without modifying any existing code. This completes the development of the main component.

```
// Invoke the actions as soon as the syntax tree is
// built up
pointcut treeConstrucion(Program n):
    target(n) && execution(Program.new(..));

after(Program n) returning(): treeConstrucion(n) {
    n.doAction();
}
```

The child component (i.e., the real `Speak` component implemented by another person) is developed in the same manner, except that it does not need specifying a dummy child component. This component supports several speak commands, each of which invokes an audio player to play the sound and types the speaking sentence on the screen. Once this component is ready, it can be composed with the main component at class loading time, which is achieved simply by altering the class path. In this way, when a speak command is encountered during parsing, the second `Speak` component will be used instead of the dummy one to parse the token and build the syntax tree. Figure 35 provides a screen shot of the robot movements and the sentences it speaks.

In summary, the development of this toy language demonstrates the integration process of syntax and semantic analysis and it illustrates how different language components can be plugged together on the fly. At the same time, it shows that the component-based approach facilitates the independent development ability of a language implementation.



*Figure 35: The graphic output of the robot language*

## CHAPTER 6

### FUTURE WORK

The framework is still considered to be in its beginning stage and there are many opportunities for improvement. This chapter includes several detailed proposals to extend the work presented in Chapter 3 and Chapter 4.

#### 6.1 CLR Backtracking

In previous chapters we discussed several problems in CLR parsing which are caused by backtracking. Below are possible solutions.

- It has been discussed in Chapter 3 that, by using backtracking, it is possible to have exponential time parsing complexity. To solve this problem, certain restrictions can be invented to constrain the backtracking behavior, such as setting a maximum backtracking length or limiting the number of non-perfect components in use. These restrictions should be designed in such a way that it will not significantly affect the language description ability of CLR grammars.
- For error handling, a possible strategy is to try all possibilities and record the path that goes the furthest. The path in this case includes all the involved parser components and their corresponding parsing stack configurations at the time the furthest point is reached. When all possibilities fail, the error recovery methods of all parsers in the longest path will be invoked in a bottom-up manner until one of

them handles the error, as in the Chain of Responsibility design pattern [Gamma 1995].

## 6.2 Module Inclusion

In component-based language development, it is often desirable to reuse only a small number of nonterminals or template definitions from other modules. As the scale of such reuse is quite small, it would be too costly to produce a standalone parser for that purpose. Moreover, a module may need to be tailored before being reused, which is difficult to achieve by direct parser composition.

Therefore, the syntax specification could be improved by borrowing some strategies from incremental development approaches to copy and modify production rules from other grammar components. Besides using the keyword `import` to declare the component reuse at the code-level, another keyword `include` can be used to indicate module reuse at the textual level. These two types of declarations can be seen as language constructs borrowed from general purpose programming languages, namely, `import` is borrowed from the Java language and `include` is copied from C++. New types of operators should be also supported to inherit, extend, overwrite or modify the included modules.

Once the above mechanism is supported, commonly used terminals, nonterminals, macros and templates can be built into libraries to further facilitate syntax reuse. Particularly, the template definitions presented in Chapter 4 should be provided as library definitions, which can significantly shorten the length of syntax notations.

Notice that the module inclusion is still different from pure module-based development. Once a change is made inside a component, although multiple included modules might need to be read, there is just one sub-parser regenerated, with other component parsers remaining intact. In other parser generators that support modular development, the change would require the whole parser being regenerated.

### **6.3 Grammar Aspects**

There are certain language constructs (e.g., arrays and pointers) that have a global effect throughout a language grammar. Their productions are functionally related to each other but physically distributed all over the grammar. Such constructs are difficult to extract as a general component. Therefore, the aspect-oriented paradigm can be utilized here to provide a second dimension for grammar modularization. For instance, an aspect component can be included in the framework to add new language artifacts whose related productions crosscut other components, which is analogous to the inter-type declaration provided in AspectJ. Aspect components can be very useful for incremental language design. The new productions to be added to a language can be specified in one aspect, which are weaved into existing components for language update.

### **6.4 Support of Other Parsing Paradigms**

In this thesis, the discussion of object-oriented syntax and component-based parsing are both based on LR grammars. It is worthwhile to explore the possibilities to extend the ideas to other parsing paradigms, as different constructs of a language may have different parsing needs.

OOCFG can be directly used for LL parser generation, provided that the grammar is left-factored and contains no left-recursion [Aho 2007]. To make this extension available, a flag can be attached to each language id to indicate what type of parser is generated based on this grammar component.

The component based parsing schema is not limited to LR parsers. The independent nature of each component allows the idea to be extended to any other parsing algorithms, such as LL, GLR, and even handcrafted parsers. Specifically, the requirements for a parser to become a valid component are the following:

- It must have the return action defined to be an eligible child component;
- It must have the switch action defined to be an eligible parent component;
- If a component is neither a perfect component nor a leaf component, or it contains internal-external conflicts, the parser must support backtracking logic as return or switch actions could be wrong.

These components can coordinate together by sharing a global interface. This enables each language construct to be implemented by the most suitable approach. For example, if one part of the language is not LR but other parts are, we could build a hand-coded parser for that particular component and compose it with the main parser built by a LR parser generator.

## **6.5 Rich Client Platform based on Eclipse**

To carry out the whole language implementation process in a user-friendly manner, an integrated development environment can be constructed. Such an IDE can be built within the Eclipse [Eclipse 2003] framework, which is a Java IDE offering a platform for

building and integrating application development tools delivered via plug-ins. By using the Eclipse Plug-in Development Environment (PDE), we can extend the existing plug-ins instead of building an IDE from scratch. There are many existing plug-ins in the Eclipse platform that could be reused to build this IDE and the most important one is the AspectJ plug-in, AJDT [AJDT 2007], which can provide a rich development feature for aspect-oriented programming language for semantics implementation. The IDE should have the following features:

- **Project based.** All components of a language implementation should be encapsulated in the same project, which includes grammar specification for syntax and tree grammar as well as aspect-oriented semantic code. AspectJ code should be put into separate folders with automatically generated code to make sure there are no multiple entrances for the same code (i.e., code should be either generated or user-supplied). Users are not supposed to edit the contents of the generated code.
- **Integration with AJDT.** The IDE should include the AJDT such that aspect-oriented semantics can be easily specified with rich editing features provided by AJDT.
- **Syntax coloring and dynamic checking.** The IDE should have syntax coloring and dynamic checking for syntax specifications. Specification checking should be invoked whenever the file is loaded, changed, or saved in the editing environment.
- **Refactoring.** Refactoring [Fowler 1999] should be provided across the specification and the programming language. For example, by using refactoring, renaming

an LHS symbol inside a production will trigger changes to all the related symbol references, as well as the corresponding class name used in syntax tree nodes and semantic aspects.

## CHAPTER 7

### CONCLUSION

Compiler development is generally known as a “dragon” task due to the intrinsic complexity engaged in a language implementation. This complexity is revealed by the fact that compiler entities in a language implementation cannot be easily modularized. Although the adoption of parser generators frees language developers to handcraft a parser from scratch, these tools force the whole language to be built as a whole regardless of its granularity. Moreover, the interconnected nature of syntax and semantic analysis of a language propels the compiler implementation to have each phase separated clearly.

With an aim to provide improved modularity and abstraction in the language implementation process, a component-based language development framework with object-oriented syntax and aspect-oriented semantics is presented in this thesis. The major design policy of the overall system is to decrease the workload of developing a large language into developing a number of smaller languages and clearly separating each phase of compiler construction, with the suitable combination of formal specification and general purpose programming languages. The framework addresses the above modularity problems in language implementation in a two-dimensional manner. Structure wise, the framework employs component-based language development to decompose a large language into a set of smaller languages, each of which is first implemented separately and then assembled together. Function wise, object-oriented syntax specification and aspect-

oriented semantic programming are used jointly within each individual language implementation to facilitate separation of concerns, where syntax and semantics as well as semantic phases themselves are isolated into different modules.

The paradigm is distinguished from all previous approaches because it embeds useful constructs and techniques from software engineering and programming languages in the compiler design domain, such as component-based development, object-orientation, aspect-orientation, inheritance, design patterns, namespace, and macros and templates. Specifically, there are four major contributions in this framework that distinguish it from other related research work:

- A new parsing algorithm called Component-based LR parsing is utilized to decompose a language implementation into executable components at the byte-code level. CLR parsing is the first algorithm that offers syntax composition at the parser level, which avoids the coupling between grammar productions. It decreases the complexity of building a large language by constructing a set of smaller language parsers from grammar components, which at the same time strengthens software engineering principles. CLR is more expressive than regular LR as it employs backtracking and multiple lexers to resolve potential conflicts at the syntax and lexical levels.
- By applying suitable restrictions on context-free grammars, object-orientation and syntax notation are integrated into a unified formalism. Consequently, both the parser and object-oriented tree hierarchy are automatically generated from a single type of syntax grammar, which eliminates the redundancy between syntax

specification and tree structure description. The two types of tree structures (CST and AST) can be selectively used to satisfy different semantic needs.

- Macros and templates are employed in the syntax grammar to improve the abstraction and reusability of the syntax definition and hide the parser implementation details from language developers. The use of macros can further resolve certain parsing conflicts existing in regular LR parsing.
- Aspect-oriented semantics implementation is able to clearly encapsulate each semantic phase that originally crosscuts various syntax tree nodes as an aspect. It further assists the object-oriented syntax by enabling direct insertion of action code into the parser. The aspect-oriented semantics supersede the object-oriented Visitor pattern by its unrestricted method definitions and transparent nature to node classes, as well as the flexibility in phase selection, tree traversal and semantic sharing among different tree nodes.

The framework has been utilized in practice to implement the Java language and various domain-specific languages. As elaborated throughout Chapters 3-6, it can be seen that the introduced framework provides a practical solution to the modularity problems raised in Chapter 1 in both the structure dimension and the functional dimension. It increases the reusability, changeability, extensibility and independent development ability of both syntax and semantics specification with less development workload required from compiler designers.

## LIST OF REFERENCES

- [Aho 2007] A. V. Aho, M.S. Lam, R. Sethi, J. D. Ullman. *Compilers: Principles, Techniques, and Tools*, 2<sup>nd</sup> edition. Addison-Wesley, 2007.
- [AJDT 2007] *The AspectJ Development Tools (AJDT)*. <http://www.eclipse.org/ajdt/>, 2007.
- [AspectJ 2003] *The AspectJ Programming Guide*.  
<http://www.eclipse.org/aspectj/doc/released/progguide/>, 2003.
- [Aycock 2001] J. Aycock, R. N. Horspool, J. Janoušek, and B. Melichar. Even Faster Generalized LR Parsing. *Acta Informatica*, Vol. 37, No. 9, pp. 633-651, 2001.
- [Baxter 2004] I. Baxter, P. Pidgeon, and M. Mehlich. DMS: Program Transformations for Practical Scalable Software Evolution. In *Proc. International Conference on Software Engineering*, pp. 625-634, May 2004.
- [van den Brand 1997] M.G.J. van den Brand, M.P.A. Sellink, and C. Verhoef. Obtaining a COBOL Grammar from Legacy Code for Reengineering Purposes. In *Proc. Second International Workshop on the Theory and Practice of Algebraic Specifications, Electronic Workshops in Computing*, Springer, Berlin, 1997.
- [van den Brand 1998] M. G. J. van den Brand, M.P.A. Sellink, and C. Verhoef. Current Parsing Techniques in Software Renovation Considered Harmful. In *Proc. International Workshop on Program Comprehension*, pp. 108-117, June 1998.
- [van den Brand 2002] M. G. J. van den Brand, J. Heering, P. Klint and P.A. Olivier. Compiling Language Definitions: The ASF+SDF Compiler. *ACM Transactions on Programming Languages and Systems*, Vol. 24, No. 4, pp. 334-368, 2002.
- [BtYacc 2006] Siber Systems Inc. *BtYacc: BackTracking Yacc*.  
<http://www.siber.com/btyacc/>, 2006.

- [Cao 2005] F. Cao. *Model Driven Development and Dynamic Composition of Web Services*. PhD thesis, University of Alabama at Birmingham, August 2005.
- [Chomsky 1959] N. Chomsky. On Certain Formal Properties of Grammars. *Information and Control*, Vol. 2, No. 2, pp. 137-167, 1959.
- [Colyer 2004] A. Colyer, A. Clement, G. Harley and M. Webster. *Eclipse AspectJ: Aspect-Oriented Programming with AspectJ and the Eclipse AspectJ Development Tools*. Addison-Wesley, 2004.
- [Cook 2006] D. Cook. *Gold Parser Builder*. <http://www.devincook.com/goldparser>, 2006.
- [Cordy 2004] J. R. Cordy. TXL - A Language for Programming Language Tools and Applications. *Electronic Notes in Theoretical Computer Science*, Vol. 110, pp. 3–31, 2004.
- [CUP 1999] *CUP: Parser Generator for Java*. <http://www.cs.princeton.edu/~appel/modern/java/CUP/>, 1999.
- [Deremer 1969] F. L. Deremer. *Practical Translators for LR(k) Languages*. Ph.D. dissertation, Massachusetts Institute of Technology, Cambridge, 1969.
- [van Deursen 1999] A. van Deursen and T. Kuipers. Building Documentation Generators, In *Proc. 1999 International Conference on Software Maintenance*, pp. 40–49, 1999.
- [Dijkstra. 1988] E. W. Dijkstra. Position Paper on “Fairness”. *SIGSOFT Software Engineering Notes*, Vol. 13, No. 2, pp. 18-20, April 1988.
- [Dijkstra 2001] A. Dijkstra and D. S. Swierstra. Lazy Functional Parser Combinators in Java. In *Proc. 1<sup>st</sup> Workshop on Multiparadigm Programming with Object-Oriented Languages*, pp. 11–42, 2001.
- [Donnelly 1995] C. Donnelly and R. Stallman. *Bison: The YACC-compatible Parser Generator*. Free Software Foundation, Boston, MA. Part of the Bison distribution, November 1995.
- [Eclipse 2003] Object Technology International, Inc. *Eclipse Platform Technical Overview*. <http://www.eclipse.org/articles/Whitepaper-Platform-3.1/eclipse-platform-whitepaper.html>, February 2003.

- [Ekman 2004] T. Ekman and G. Hedin. Rewritable Reference Attributed Grammars. In *Proc. 18th European Conference on Object-Oriented Programming*, LNCS Vol. 3086, pp. 147-171. Springer, June 2004.
- [Ford 2004] B. Ford. Parsing Expression Grammars: A Recognition-based Syntactic Foundation. In *Proc. 31<sup>st</sup> ACM Symposium on Principles of Programming Languages*, pp. 111–122, January 2004.
- [Fowler 1999] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.
- [Gagon 1998] E. Gagnon. *SableCC, An Object-Oriented Compiler Framework*. Master's thesis, McGill University, Montreal, Quebec, March 1998.
- [Gamma 1995] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns, Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.
- [Gauthier 1970] R. Gauthier, and S. Pont. *Designing Systems Programs (C)*, Prentice-Hall, Englewood Cliffs, N.J., 1970.
- [Gosling 1996] J. Gosling, B. Joy, and G. L. Steele. *The Java Language Specification*, Addison-Wesley, 1996
- [Grimm 2006] R. Grimm. Better Extensibility through Modular Syntax. In *Proc. 2006 ACM Conference on Programming Language Design and Implementation*, pp. 38-51, June 2006.
- [Hachani 2002] O. Hachani and D. Bardou. Using Aspect-Oriented Programming for Design Patterns Implementation. In *Proc. Workshop Reuse in Object-Oriented Information Systems Design*, 2002.
- [Hannemann 2002] J. Hannemann and G. Kiczales. Design Pattern Implementation in Java and AspectJ. In *Proc. Object-Oriented Programming, Systems, and Applications (OOPSLA)*, pp. 161–173, 2002.
- [Hedin 2001] G. Hedin and E. Magnusson. JastAdd-a Java-based System for Implementing Front Ends. In M. van den Brand and D. Parigot, eds, *Electronic Notes in Theoretical Computer Science*, Vol. 44, Elsevier Science Publishers, 2001.
- [Hedin 2003] G. Hedin and E. Magnusson. JastAdd-An Aspect-Oriented Compiler Construction System. In M. van den Brand, M. Mernik and D. Parigot, eds, *Science of Computer Programming*, Vol. 47, No. 1, pp. 37-58, 2003.

- [Hedin 2005] G. Hedin. *The JastAdd Extensible Java Compiler*. JastAdd Project Website. <http://jastadd.cs.lth.se/web/extjava/index.shtml>, 2005.
- [Heineman 2001] G. T. Heineman and W. T. Councill. *Component Based Software Engineering: Putting the Pieces Together*. Addison-Wesley, 2001.
- [Hopcroft 2001] J. E. Hopcroft, R. Motwani, and J. D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, 2001.
- [Hutton 1998] G. Hutton and E. Meijer. Monadic Parsing in Haskell, *Journal of Functional Programming*, Vol. 8, No. 4, pp. 437-444, July 1998.
- [JavaCC 2006] *JavaCC: Java Compiler Compiler*, Sun Microsystems, Inc. <https://javacc.dev.java.net/>, 2006.
- [JLex 2006] *JLex: Java Lexical Analyzer Generator*. <http://www.cs.princeton.edu/~appel/modern/java/JLex/>, 2006.
- [JJTree 2006] *Introduction to JJTree*. <http://www.j-paine.org/jjtree.html>, 2006.
- [Johnson 1975] S.C. Johnson. *YACC: Yet Another Compiler Compiler*. Technical Report 32, AT&T Bell Laboratories, Murray Hill, NJ, 1975.
- [Johnstone 2004a] A. Johnstone, E. Scott, and G. Economopoulos. The Grammar Tool Box: a Case Study Comparing GLR Parsing Algorithms. *Electronic Notes in Theoretical Computer Science*, Vol. 110, pp. 97-113, 2004.
- [Johnstone 2004b] A. Johnstone, E. Scott, and G. Economopoulos. Generalised Parsing: Some Costs. In *Proc. 13<sup>th</sup> International Conference on Compiler Construction*, pp. 89–103, March 2004.
- [Jones 2003] S. P. Jones, editor. *Haskell 98 Language and Libraries: The Revised Report*. Cambridge University Press, 2003.
- [JTB 2000] *JTB: Java Tree Builder*. <http://www.cs.purdue.edu/jtb/releasenotes.html>, 2000.
- [Kaiser 1989] G. E. Kaiser. Incremental Dynamic Semantics for Language-based Programming Environments. *ACM Transactions on Programming Languages and Systems*, Vol. 11, pp. 169-193, 1989.

- [Kiczales 1997] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J. Loingtier, and J. Irwin. Aspect-Oriented Programming. In *Proc. 11<sup>th</sup> European Conference Object-Oriented Programming (ECOOP)*, pp. 220-242, 1997.
- [Kiczales 2001] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An Overview of AspectJ. In *Proc. 15th European Conference on Object-Oriented Programming (ECOOP)*, Springer-Verlag, LNCS 2072, pp. 327–355, 2001.
- [Kipps 1991] R. J. Kipps. GLR Parsing in Time  $O(n^3)$ . In *Tomita, M., ed., Generalized LR Parsing*, pp. 43-60. Kluwer Academic Publishers, 1991.
- [Kuipers 2001] T. Kuipers, J. Visser. Object-Oriented Tree Traversal with JForester. *Electronic Notes in Theoretical Computer Science*, Vol. 44, pp. 59-87, 2001.
- [Knuth 1965] D. E. Knuth. On the Translation of Languages from Left to Right. *Information and Control*, Vol. 8, pp. 607-639, 1965.
- [Knuth 1968] D. E. Knuth. Semantics of Context-Free Languages. *Mathematical Systems Theory*, Vol. 2, No. 2, pp. 127-145, 1968.
- [Lesk 1975] M. E. Lesk and E. Schmidt. *Lex - A Lexical Analyser Generator*. Technical Report, AT&T Bell Laboratories, Murray Hill, NJ, 1975.
- [McPeak 2004] S. McPeak and G. C. Necula. Elkhound: A Fast, Practical GLR Parser Generator. In *Proc. 13<sup>th</sup> International Conference on Compiler Construction*, pp. 73–88, March 2004.
- [Melton 2000] J. Melton and A. Eisenberg. *Understanding SQL and Java Together: A Guide to SQLJ, JDBC, and Related Technologies*. Morgan-Kaufmann, 2000.
- [Mernik 2004] M. Mernik, X. Wu, B. R. Bryant. Object-Oriented Language Specification: Current Status and Future Trends, In *Proc. ECOOP Workshop on Evolution and Reuse of Language Specifications for DSLs*, June 2004.
- [Mernik 2005a] M. Mernik, J. Heering, and A. Sloane. When and How to Develop Domain-Specific Languages. *ACM Computing Surveys*, Vol. 37, No. 4, pp. 316–344, 2005.
- [Mernik 2005b] M. Mernik, and V. Žumer. Incremental Programming Language Development. *Computer Languages, Systems and Structures*, Vol. 31, pp. 1–16, 2005.

- [Moonen 2001] L. Moonen. Generating Robust Parsers Using Island Grammars, In *Proc. IEEE 8th Working Conference on Reverse Engineering*, pp. 13–22, 2001.
- [de Moor 2000] O. de Moor, S. Peyton-Jones, and E. Van Wyk. Aspect-oriented Compilers. In *Proc. Generative and Component-Based Software Engineering (GCSE)*, pp. 121–133, 2000.
- [Naur 1960] P. Naur. Report on the Algorithmic Language ALGOL 60. *Communications of the ACM*, Vol. 3, pp. 299-314, 1960.
- [Nystrom 2003] N. Nystrom, M. R. Clarkson, and A. C. Myers. Polyglot: An Extensible Compiler Framework for Java. In *Proc. 12<sup>th</sup> International Conference on Compiler Construction*, pp. 138-152, April 2003.
- [Paakki 1995] J. Paakki. Attribute Grammar Paradigms - A High-Level Methodology in Language Implementation. *ACM Computing Surveys*, Vol. 27, No. 2, pp. 196-255, 1995.
- [Pagan 1981] F. G. Pagan. *Formal Specification of Programming Languages: A Panoramic Primer*. Prentice Hall, 1981.
- [Parnas 1972] D. Parnas. On the Criteria to be Used in Decomposing Systems into Modules. *Communications of the ACM*, pp. 1053-1058, December 1972.
- [Parr 1995] T. J. Parr and R. W. Quong. ANTLR: A Predicated-LL(k) Parser Generator. *Software, Practice and Experience*, Vol. 25, No. 7, pp. 789–810, July 1995.
- [Parr 2007] T. Parr. *The Definitive ANTLR Reference: Building Domain-Specific Languages*. The Pragmatic Programmers, LLC, 2007.
- [Sebesta 2006] R. W. Sebesta. *Concepts of Programming Languages*, 7<sup>th</sup> edition. Boston, MA: Addison-Wesley Longman Publishing Co., Inc, 2006.
- [Slonneger 1995] K. Slonneger, and B.L. Kurtz. *Formal Syntax and Semantics of Programming Languages*. Addison-Wesley, 1995.
- [Szyperski 2002] C. Szyperski. *Component Software: Beyond Object-Oriented Programming*, 2<sup>nd</sup> edition. ACM Press, 2002.
- [Tarr 1999] P. Tarr, H. Ossher, W. Harrison, and S. Sutton. N Degrees of Separation: Multi-Dimensional Separation of Concerns. In *Proc. International Conference Software Engineering (ICSE)*, pp. 107-119, 1999.

[Tomita 1986] M. Tomita. *Efficient Parsing for Natural Language*. Kluwer Academic Publishers, 1986.

[Vandevoorde 2002] D. Vandevoorde and N. M. Josuttis. *C++ Templates: The Complete Guide*. Addison-Wesley, November 2002.

[Visser 1997] E. Visser. *Syntax Definition for Language Prototyping*. PhD thesis, University of Amsterdam, September 1997.

[Visser 2000] J. Visser and J. Scheerder, *A Quick Introduction to SDF*, CWI, April 2000.

[Weatherley 2006] R. Weatherley. *TreeCC: An Aspect-Oriented Approach to Writing Compilers*. <http://www.southern-storm.com.au/treec.html>, 2006.

[Wexelblat 1981] L. Wexelblat, ed. *History of Programming Languages*. New York: Academic Press, 1981.

[Wikipedia 2006a] Wikipedia. *Object-oriented programming*. [http://en.wikipedia.org/wiki/Object-oriented\\_programming](http://en.wikipedia.org/wiki/Object-oriented_programming), 2006.

[Wikipedia 2006b] Wikipedia. *Syntactic Sugar*. [http://en.wikipedia.org/wiki/Syntactic\\_sugar](http://en.wikipedia.org/wiki/Syntactic_sugar), 2006.

[Wu 2004] X. Wu, B. R. Bryant, and M. Mernik. Object-Oriented Pattern-Based Language Implementation. *Acta Electrotechnica et Informatica*, No.4, Vol.4, pp.5-12, 2004.

[Wu 2005a] X. Wu, S. Roychoudhury, B. Bryant, J. Gray, and M. Mernik. A Two-Dimensional Separation of Concerns for Compiler Construction. In *Proc. ACM Symposium on Applied Computing (SAC)*, pp. 1365-1369, 2005.

[Wu 2005b] X. Wu, Bryant, B., Gray, J., and Mernik, M. 2005. Applying Object-Oriented and Aspect-Oriented in Teaching Domain-Specific Language Implementation. *Journal of Computing Science in Colleges*, Vol. 21, No. 2, pp. 335-340, December 2005.

[Wu 2006] X. Wu, B.R. Bryant, J. Gray, S. Roychoudhury, and M. Mernik. Separation of Concerns in Compiler Development Using Aspect-Oriented. In *Proc. ACM Symposium on Applied Computing (SAC)*, pp. 1585-1590, 2006.

## APPENDIX A

## THE SPECIFICATION OF THE JAVA BINARY EXPRESSION COMPONENT

Note: after removing the two productions in the bold font, the remaining productions will result in a binary expression component for the C and C++ languages.

```

multiplicative_expression ::=
    unary_expression
    | multiplicative_expression MULT unary_expression
    | multiplicative_expression DIV unary_expression
    | multiplicative_expression MOD unary_expression
    ;

```

```

additive_expression ::=
    multiplicative_expression
    | additive_expression PLUS multiplica-
      tive_expression
    | additive_expression MINUS multiplica-
      tive_expression
    ;

```

```

shift_expression ::=
    additive_expression
    | shift_expression LSHIFT additive_expression
    | shift_expression RSHIFT additive_expression
    | shift_expression URSHIFT additive_expression
    ;

```

```

relational_expression ::=
    shift_expression
    | relational_expression LT shift_expression
    | relational_expression GT shift_expression
    | relational_expression LTEQ shift_expression
    | relational_expression GTEQ shift_expression
    | relational_expression INSTANCEOF reference_type
    ;

```

```

equality_expression ::=
    relational_expression
    | equality_expression EQEQ relational_expression
    | equality_expression NOTEQ relational_expression
    ;

```

```

and_expression ::=
    equality_expression
    | and_expression AND equality_expression
    ;

```

```

exclusive_or_expression ::= and_expression

```

```
|    exclusive_or_expression XOR and_expression
;

inclusive_or_expression ::=
    exclusive_or_expression
    |    inclusive_or_expression OR
        exclusive_or_expression
    ;

conditional_and_expression ::=
    inclusive_or_expression
    |    conditional_and_expression ANDAND
        inclusive_or_expression
    ;

conditional_or_expression ::=
    conditional_and_expression
    |    conditional_or_expression OROR
        conditional_and_expression
    ;
```

## APPENDIX B

### THE PRETTY PRINT ASPECT OF THE JAVA LANGUAGE IMPLEMENTATION

```

aspect Indent {

    declare precedence : Indent, PrettyPrint;
    static int indentLevel = 0;

    pointcut blockPrint():
        execution(String (Class_body|Block).toString());

    before() : blockPrint() {
        indentLevel++;
    }

    after () returning : blockPrint() {
        indentLevel--;
    }
}

aspect PrettyPrint {
    static HashSet keywordSet = new HashSet();
    static String[] keywords = {
        "package", "import", "public", "if",
        "else", ..., "while", "do"};

    static void loadKeywords(){
        for (int i = 0; i < keywords.length; i++){
            keywordSet.add(keywords[i]);
        }
    }

    pointcut newlinePrint():execution(String (
        Class_declaration ||
        Method_declaration ||
        Import_declaration ||
        Package_declaration ||
        // ...
        Statement).toString());

    String around() : newlinePrint() {
        return "\n" + indents() + proceed();
    }

    String around(Token tok) : target(tok) &&
        execution(String Token.toString()){

        if (keywordSet.contains(tok.lexeme()))
            return proceed(tok) + " ";
        else if (tok.lexeme().compareTo("}")==0){
            return "\n" + indents() + proceed(tok);
        }
    }
}

```

```
    }  
    return proceed(tok);  
}  
  
String indents() {  
    String blanks = "";  
    for (int i = 0; i < Indent.indentLevel - 1; i++) {  
        blanks += "  ";  
    }  
    return blanks;  
}  
  
// ... ..  
}
```

## APPENDIX C

## THE SYNTAX GRAMMAR OF OOCFG PARSER GENERATOR

```

//
// Section 1: Template definitions
//

// Optional item
opt<#Item> ::= #Item | ;

// Plain list
list<#Item> ::= #Item opt<list<#Item>>;

// List with separators
sep_list<#SEP, #Item> ::= m_sep_list<#SEP, #Item>
                        | #Item
                        ;

m_sep_list<#SEP, #Item> ::= sep_list<#SEP, #Item>
                          #SEP
                          #Item
                          ;

//
// Section 2: Nonterminal and macro definitions
//

compilation_unit ::= language_declaration
                   opt<include_declaration>
                   opt<list<import_declaration>>
                   export_declaration
                   list<production>
                   ;

language_declaration ::= LANGUAGE
                     language_name
                     PRODUCT_END
                     ;

include_declaration ::= INCLUDE
                    sep_list<COMMA, parser_name>
                    PRODUCT_END
                    ;

import_declaration ::= IMPORT
                   sub_language_name
                   AS ID PRODUCT_END
                   ;

language_name ::= sep_list<DOT, ID>;

```

```

sub_language_name ::= m_sep_list<DOT, ID>;

export_declaration ::= EXPORT
                    sep_list<COMMA, export_id>
                    PRODUCT_END
                    ;

export_id ::= opt<perfect> ID;

production ::= nonterminal_definition
              | token_definition
              | macro_definition
              | template_definition
              ;

nonterminal_definition ::= opt<type> ID
                        LHS_RHS_SEP
                        rhs
                        PRODUCT_END
                        ;

type ::= LPAREN item RPAREN;

rhs ::= list<item> | m_sep_list<ITEM_SEP, item>;

item ::= identifier | template_reference;

identifier = ID opt<postfix>;

postfix ::= LBRACKET NUMBER RBRACKET;

template_reference = identifier
                   LARROW
                   sep_list<COMMA, param>
                   RARROW
                   ;

param ::= value_param | generic_param;

value_param = ID;

generic_param = SHARP ID;

token_definition ::= opt<type> ID
                   QSTRING
                   PRODUCT_END

```

*i*

```
macro_definition ::= opt<type> ID  
                  EQUAL rhs  
                  PRODUCT_END  
                  ;
```

```
template_definition ::= opt<type> template_reference  
                       LHS_RHS_SEP rhs  
                       PRODUCT_END  
                       ;
```