Software Composition and Modeling Laboratory
SoftCom
Department of Computer and Information Sciences
University of Alabama at Birmingham

A Framework for Component-based Compiler Development

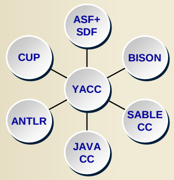
Xiaoqing Wu
Advisors: Barrett R. Bryant, Jeff Gray, Marjan Mernik
Department of Computer and Information Sciences
The University of Alabama at Birmingham
http://www.cis.uab.edu/wuxi/research/ Email: wuxi@cis.uab.edu

UAB Department of Computer & Information Sciences

1. Problem Statement

Current compiler development practices suffer from lack of modularity.

- No decomposition of language definitions
  - Large grammars may run into thousands of productions (e.g., Cobol 85 has 2500 lines of specification with more than 1000 variables).
  - There is no modular grammar support in most parser generators (e.g. YACC).



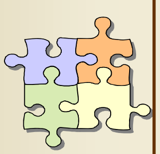
- No clear separation of compiler construction phases
  - *Syntax and semantics:* The communication between syntax (in formal specification) and semantics (in programming languages) makes the specification and code tangled together.
  - *Among different semantic phases:* In pure object-oriented design, code scatters throughout the syntax tree class hierarchy

Lack of modularity usually yields poor comprehensibility, reduced changeability, limited reusability, and hampers independent development of the language implementation.

2. Background

Component-Based Software Engineering (CBSE)

- The goal of CBSE is to build a software system by assembling small components in the same manner as hardware composition. A software component is generally considered to have the following properties:
  - Information hiding
  - Explicit interface
  - Context independency



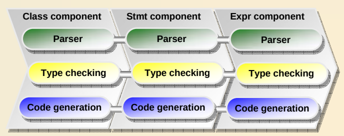
Aspect-Oriented Programming (AOP)

- Aspect-Oriented Programming provides special language constructs called aspects to modularize crosscutting concerns in conventional program structures. AOP offers two new types of functionalities:
  - introduction to existing structure
  - insertion of group behaviors (i.e., join point model).
- In AOP, a translator called a weaver is responsible for merging the aspects into the base language at compile time.

3. Research Objectives and Major Contributions

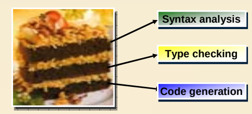
Research Objectives

- The goal of this new language implementation framework is to break the compiler constructs into pluggable components, which helps to improve the modularity in the language development process and alleviate the development and maintenance complexity.
- By utilizing CBSE strategy and aspect-oriented development paradigm, this framework is targeted to address the modularity problem in compiler design in both structural and functional dimensions.



Objective I: Large languages should be developed based on small language implementations.

Objective II: Inside a single language implementation, each compiler phase should be separated with each other.



Syntax ⟷ Semantics
Semantic phase1 ⟷ Semantic phase2
Auto-generated code ⟷ Hand-written code
Declarative formal specification ⟷ Imperative programming language code

Major Contributions

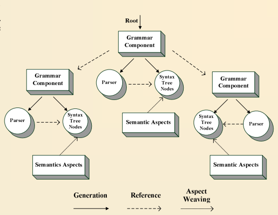
- The framework employs a new parsing algorithm called Component-based LR (CLR) parsing.
  - CLR is the first algorithm that offers syntax composition at the parser level.
  - CLR decreases the complexity of building a large language by constructing a set of smaller language parsers from grammar components.
  - CLR is more expressive than regular LR as it employs backtracking and multiple lexers to resolve potential conflicts at the syntax levels.
  - The object-oriented CLR specification language generates both parser and syntax tree, which eliminates the redundancy between syntax specification and tree structure description.
- The aspect-oriented semantics employed in the framework supersede the object-oriented Visitor pattern by unrestricted method definitions and transparent nature of node classes, as well as the flexibility in tree walking and phase composition using join points.

Modularity ↑ Complexity ↓

4. Component-based development framework

Framework Overview

- **Structure decomposition:** CLR decomposes a large language into a set of smaller languages, each of which is first implemented separately and then assembled together.
- **Function decomposition:** Object-Oriented Syntax (OOS) and Aspect-Oriented Semantics (AOS) are used jointly within each individual language implementation to facilitate Separation of Concerns (SoC), where syntax and semantics as well as semantic phases themselves are isolated into different modules.



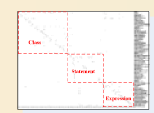
Language Decomposition and Syntax Implementation

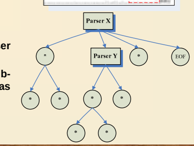
Grammar Components

- A grammar component G is a quintuple (N, T, C, P, S)
  - N: nonterminal symbols
  - T: terminal symbols
  - C: other grammar components
  - $P \subseteq N \times (N \cup T \cup C)^*$ is a finite set of production rules, a production of the form $A \rightarrow \alpha$ means A derives α.
  - $S \in N$: the start symbol

Parser components

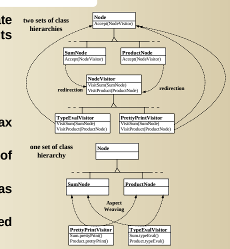
- Each grammar component generates a parser component and an object-oriented syntax tree
- To conduct parsing, the root parser invokes its sub-parsers that will recursively invoke other parsers as needed.





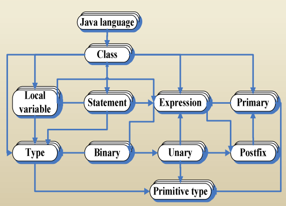
Aspect-Oriented Semantics

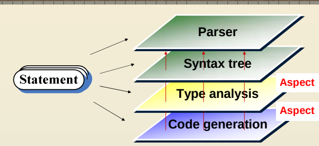
- AOS implementation can clearly encapsulate each semantic phase that originally crosscuts various syntax tree nodes as an aspect.
  - An independent semantic pass
  - A group of action codes
- Semantic pass
  - Implemented as introductions to the syntax tree classes
- Crosscutting actions applied to a group of nodes
  - Weaved into syntax tree classes as interceptions
- AOS is less complex than the object-oriented Visitor pattern.



5. Case study: the Java language implementation

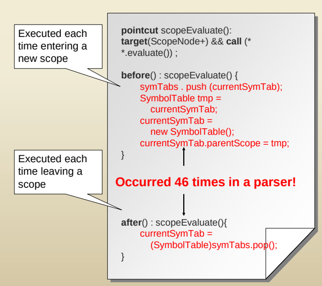
- The framework has been utilized in practice to implement the Java language. By using language decomposition, the workload is decreased into developing a number of smaller languages that can be implemented and executed independently.



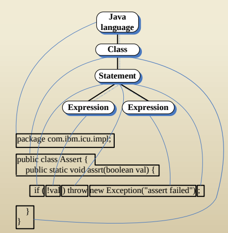


- For each individual language implementation, the built-in CLR specification automates the parser and tree construction processes.
- With an object-oriented syntax tree, Aspect-Oriented Programming (AOP) is able to clearly encapsulate each semantic phase as an aspect. These semantics are integrated into the object-oriented syntax tree through aspect weaving.

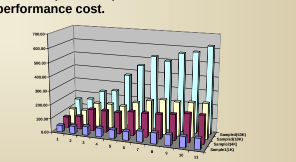
- AOP offers the flexibility in semantic pattern description.



- By using CLR parsing, these components are assembled together to deliver the overall language implementation.



- Besides the Java language, the framework has also been utilized to implement various domain-specific languages. These practices have proved that the overall paradigm provides exceptional modularity for compiler implementation with a minor performance cost.



- Consequently, the framework increases the comprehensibility, reusability, changeability, extendibility and independent development ability of the syntax and semantics with less development workload required from compiler designers.