Component-Based Language Implementation with Object-Oriented Syntax and Aspect-Oriented Semantics

Barrett Bryant
Outline

- Background knowledge
- Problem statement
- Related work
- Framework overview
- Component-based language development
- OOS and AOS
  - Object-oriented syntax
  - Aspect-oriented semantics
- Framework usage
- Future work
- Conclusion
Outline

- **Background knowledge**
- Problem statement
- Related work
- Framework overview
- Component-based language development
- OOS and AOS
  - Object-oriented syntax
  - Aspect-oriented semantics
- Framework usage
- Future work
- Conclusion
Background knowledge

- Syntax and semantics
- Component-Based Software Engineering (CBSE)
  - Promote software reuse
  - Essential properties
    - Information hiding
    - Explicit interface
    - Context Independency
- Object-oriented programming
- Aspect-Oriented Programming (AOP) and AspectJ
  - Aspects: special language constructs to modularize crosscutting concerns.
  - Introduction (inter-type declarations)
  - Interception (join-points)
Outline

- Background knowledge
- **Problem statement**
- Related work
- Framework overview
- Component-based language development
- OOS and AOS
  - Object-oriented syntax
  - Aspect-oriented semantics
- Framework usage
- Future work
- Conclusion
Compiler construction vs. cooking a wedding cake

- **Cooking facilities**: YACC, JavaCC, CUP, ...

- **Cooking complexity**
  - Compiler design is known as a “dragon” task
  - Good modularity enables you to divide-and-conquer the complexity
  - As long as the pieces can be assembled together
No decomposition of language definitions

Most parser generators don’t support modular grammar definitions at all.

Cobol 85 is 2500 lines of specification, more than 1000 variables.

Comprehensibility

Changeability

Reusability

Independent development
No clear separation of compiler construction phases

- **Syntax and semantics**
  - Syntax analysis -- formal specification
  - Semantic analysis -- programming languages
  - The communication between syntax and semantics makes the specification and code tangled together

- **Among different semantic phases**
  - Pure object-oriented design, code scatter all over the syntax tree class hierarchy

*Hard to maintain and evolve!!*
Ideal separation objectives

Syntax analysis

Type checking

Code generation

Syntax

Semantic phase1

Auto-generated code

Declarative formal specification

Semantics

Semantic phase2

Hand-written code

Imperative programming language code
Outline

- Background knowledge
- Problem statement
- **Related work**
- Framework overview
- Component-based language development
- OOS and AOS
  - Object-oriented syntax
  - Aspect-oriented semantics
- Framework usage
- Future work
- Conclusion
Related work – decomposition of language syntax

Modular grammar

- The nature is pure text copying
- Modules are still tightly coupled \(\rightarrow\) conflicts

<table>
<thead>
<tr>
<th>Tools</th>
<th>Conflict solving solution</th>
</tr>
</thead>
<tbody>
<tr>
<td>LISA,PPG</td>
<td>Manually</td>
</tr>
<tr>
<td>BtYacc</td>
<td>Backtracking</td>
</tr>
<tr>
<td>SDF,DMS</td>
<td>GLR</td>
</tr>
</tbody>
</table>

- Any update to a particular module requires a re-composition of all the modules and regeneration of a large parse table
Related work – separation of compiler phases

- **Separation between syntax and semantics**
  - Semantics by formal specification
    - Good separation but not rich enough to fully describe semantics

- **Separation between semantic phases**
  - The Visitor pattern
    - Introduces a lot of extra code
    - Forces all concrete visitors to share the same interface
    - New semantics are always introduced by traversal of the whole tree
    - Cannot access private members of a node class
  - Aspect-oriented semantics
    - JastAdd II – only supports static introduction
Outline

- Background knowledge
- Problem statement
- Related work
- **Framework overview**
- Component-based language development
- OOS and AOS
  - Object-oriented syntax
  - Aspect-oriented semantics
- Framework usage
- Future work
- Conclusion
Component-based LR (CLR) parsing decomposes a large language into a set of smaller languages.

Object-Oriented Syntax (OOS) and Aspect-Oriented Semantics (AOS) facilitate separation of different phases.
OOS + AOS implementation
**Contribution**

- CLR decreases the development complexity by reducing the granularity of a language
  - Syntax composition at the parser level ➔ reduced coupling between grammar modules
  - More expressive than regular LR parsing

- OOS + AOS isolates syntax and semantics as well as semantic phases themselves into different modules
  - Separation of declarative and imperative behavior
  - Separation of generated code and handwritten code
  - OOS - generation of both parser and syntax tree
  - AOS - transparent to node classes, flexible in tree walking and phase composition.

The overall paradigm increases the comprehensibility, reusability, changeability, extendibility and independent development ability of the syntax and semantic analysis with less development workload required from compiler designers.
Outline

- Background knowledge
- Problem statement
- Related work
- Framework overview
- **Component-based language development**
- OOS and AOS
  - Object-oriented syntax
  - Aspect-oriented semantics
- Framework usage
  - Demo
- Future work
- Conclusion
Component-based Context-Free Grammar (CCFG)
Java language components (11 components)
CCFG & CCFL & parser components

- A CCFG component $G$ is a quintuple $(N, T, C, P, S)$
  - $N$: a set of nonterminal symbols
  - $T$: a set of terminal symbols
  - $C$: a set of symbols representing 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 $\alpha$.
  - $S \in N$: the start symbol

- CCFL
  - Let $\sigma \in (N \cup T \cup C \cup L(C))^*$, $\tau \in (N \cup T \cup C \cup L(C))^*$, $\gamma_1 = \sigma \beta \tau$ and $\gamma_2 = \sigma \gamma \beta \tau$, 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 $\beta \in L(B)$
  - $L(G) = \{x | S \Rightarrow^* x, x \in (T \cup L(C))^*\}$

- Parser components
  - Grammar component $\rightarrow$ parser component.
  - The root parser invokes its sub-parsers that will recursively invoke other parsers as needed.
Component-based grammar vs. modularized grammar

Modularized grammar

Grammar Module
Grammar Module
Grammar
Parser

Component-based grammar

Grammar Component
Grammar Component
Grammar Component
Parser
Parser

Code-level composition, less coupled definition, smaller parsing table, multiple lexers, etc...
CLR parsing algorithm – switch and return

Pseudo code:
flag component_parse(program, stack):
    repeat
        state := stack.top()
        if switch_map(state) ≠ ∅
            for ∀ 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
CLR parsing example (4 components)

```java
package com.ibm.icu.impl;

public class Assert {
    public static void assert(boolean val) {
        if (!val) throw new Exception("assert failed");
    }
}
```
Software engineering benefits

Comprehensibility

Intertwined symbols and productions are reduced

Changeability

Changes are isolated inside individual components
Only local recompilation needed

Reusability

Components can be plugged and played

Independent development

Dependencies are handled at the code-level instead of the grammar level
Language description ability

- **Expressive power**
  - CLR’s backtracking can resolve the traditional shift-reduce or shift-shift conflicts in LR parsers

- **Ambiguous tokens**
  - Benefited by multiple lexers
  - Useful for embedded languages, languages with no reserved words, etc.
    - SQLJ: `count`
    - PL/I example `IF IF = THEN THEN IF = THEN;`
Performance measurement

Parsing speed comparison among 11 versions of CLR implementation of JLS

The increase of external actions as the number of components increases
Outline

- Background knowledge
- Problem statement
- Related work
- Framework overview
- Component-based language development
- **OOS and AOS**
  - Object-oriented syntax
  - Aspect-oriented semantics
- Framework usage
- Future work
- Conclusion
Syntax can be easily specified by formal specification but semantics cannot due to its arbitrary nature.
# Object-oriented syntax

\[ A ::= B \ C \ | \ D; \]

| OOS specification | A ::= B C | A ::= B | C |
|-------------------|-----------|-----------|
| LHS and RHS relationship | Aggregation | Inheritance |
| Generated Cup specification | CUP: A ::= B : B C : C {: Result = new A(B,C);:} | A ::= B : B {: Result = B; :} | C : C {: Result = C; :} |
| Node class diagram | ![Node class diagram](image) |

- **Aggregation**: A new object is created from its components. The components maintain their identity and are available to the created object. The components cannot be changed once they have been added to the composed object.
- **Inheritance**: A new instance is created from its super class reference. The super class is then discarded and the objects of the classes are not available from their hierarchy up.

![Node class diagram](image)
// Syntax definition
Stmt ::= Block
   | “if” Expr “then” Stmt
   | Id “:=” Exp

// Tree definition
abstract Stmt;
BlockStmt : Stmt ::= Block;
IfStmt : Stmt ::= Exp Stmt;
AssignStmt : Stmt ::= Id “:=” Exp;

Stmt ::= Block | IfStmt | AssignStmt.
IfStmt ::= “if” Exp “then” Stmt.
AssignStmt ::= Id “:=” Exp.

Object-oriented syntax

JastAdd specification
OOCFG specification features and their usage

- **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.

- **AST and CST**
  - Providing semantic analysis the flexibility in tree selection and ensuring all analysis needs can be easily computed.

- **Typed LHS symbol – no node class generation**
  - Promoting the reuse of existing node classes.

- **Macros – only occur in syntax trees, transparent to parser**
  - Reducing parsing conflicts while providing richer description of the grammar and distinct syntax tree nodes.

- **Templates – generic production definitions**
  - Facilitating OOCFG to support generic production definition in a grammar specification.
## Abstract syntax tree vs. concrete syntax tree

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

OOS specification generates both concrete syntax tree and abstract syntax tree to fully satisfy various semantic analysis requirements
Aspect-oriented semantics implementation

- Each semantic concern is modularized 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
Introductions
Aspect-oriented introduction vs. object-oriented 
Visitor pattern
500 lines of redundant code have been removed!
pointcut scopeEvaluate(): target(ScopeNode+) && call(* *.evaluate()) ;

before() : scopeEvaluate(){
    symTabs.push(currentSymTab);
    SymbolTable tmp = currentSymTab;
    currentSymTab = new SymbolTable();
    currentSymTab.parentScope = tmp;
}

after() : scopeEvaluate(){
    currentSymTab = (SymbolTable)symTabs.pop();
}

Occurred 46 times in a parser!

Executed each time entering a new scope

Executed each time leaving a scope
before() : scopeEvaluate(){
    symTabs.push(currentSymTab);
    SymbolTable tmp = currentSymTab;
    currentSymTab = new SymbolTable();
    currentSymTab.parentScope = tmp;
}

after() : scopeEvaluate(){
    currentSymTab = (SymbolTable)symTabs.pop();

    // node is an instance of ScopeNode
    node.evaluate();

    // node is an instance of ScopeNode
    node.evaluate();

    symTabs.push(currentSymTab);
    SymbolTable tmp = currentSymTab;
    currentSymTab = new SymbolTable();
    currentSymTab.parentScope = tmp;

    ...
AOS advantages

- Aspect-orientation can isolate crosscutting semantic behavior in an explicit way
  - Each semantic aspect can be freely attached to (generated) AST nodes without “polluting” the parser or AST node structure.
  - Different aspects can be selectively plugged in for different purposes at compile time.
  - Since each aspect is separated with other aspects, developers can always come back to the previous phase while developing a later phase.
AOS advantages (cont’d)

- **Inter-type declaration**
  - Defined within class scope, direct access to syntax tree node class members
  - Extend the object-oriented inheritance relationship

- **Join-point model**
  - Provide flexibility to insert semantic behaviors into AST nodes or parser
  - Avoid code duplication
  - Trace facility

- **Introduction + Interception**
  - Tree traversal
  - Phase combination

```java
aspect PrintNodeCreation {
  pointcut construction(Node n): target(n)
    && execution((Node+ && !Node).new(..));
  after(Node n) returning():construction(n) {
    System.out.println(
      thisJoinPointStaticPart.getSignature()
        .getDeclaringType().getName()+"is created");
  }
}
```
Integration with CLR parsing

- **Syntax specification** ➔ The restrictions of OOS can be applied to CCFG without generating any side-effects
- **Syntax tree construction** ➔ CLR’s parse tree generation process is inlined with OOS tree generation
- **Semantic analysis** ➔ Semantic composition follows syntax tree composition
OOS + AOS implementation
Outline

- Background knowledge
- Problem statement
- Related work
- Framework overview
- Component-based language development
- OOS and AOS
  - Object-oriented syntax
  - Aspect-oriented semantics
- **Framework usage**
- Future work
- Conclusion
Case studies

- Pam and BasicM
- RelationJava
- OOCFG Converter
- Bootstrap Implementation
- Google Query Language
- Java

- Changeability
- Reusability
- Independent development ability
- Readability
- Flexible semantics
- Flexible syntax/tree
Outline

- Background knowledge
- Problem statement
- Related work
- Framework overview
- Component-based language development
- OOS and AOS
  - Object-oriented syntax
  - Aspect-oriented semantics
- Framework usage
- Future work
- Conclusion
Future work

- CLR backtracking
  - Time complexity
  - Error recovery
- Module inclusion
- Grammar aspects
- Support of other parsing paradigms
- Rich client platform based on eclipse platform
Outline

- Background knowledge
- Problem statement
- Related work
- Framework overview
- Component-based language development
- OOS and AOS
  - Object-oriented syntax
  - Aspect-oriented semantics
- Framework usage
- Future work
- Conclusion
Conclusion

- Compiler design is an intricate task because it is hard to be modularized (structure wise and function wise).
- The presented framework presents a solution that can attack the modularity problems in two dimensions
  - CLR decreases the complexity of building a large language by constructing a set of smaller language parsers from grammar components
  - OOS + AOS provides a clean separation of concerns between syntax and semantics as well as semantic phases themselves
  - The framework also supersedes conventional language implementation practices by its description power, reduced specification, and support to describe crosscutting semantic behaviors, etc.
- Various experiments prove that the methodology increases the comprehensibility, reusability, changeability, extendibility and independent development ability of both syntax and semantics specification with less development workload required from compiler designers.
Further Information

Contact:

bryant@cis.uab.edu

Web Page:

http://www.cis.uab.edu/softcom/cde