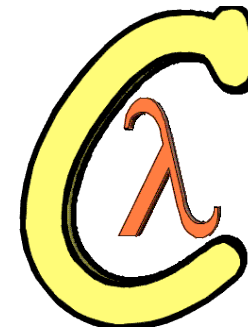
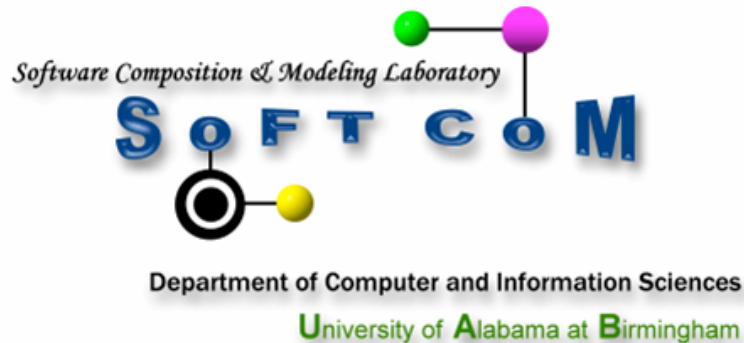


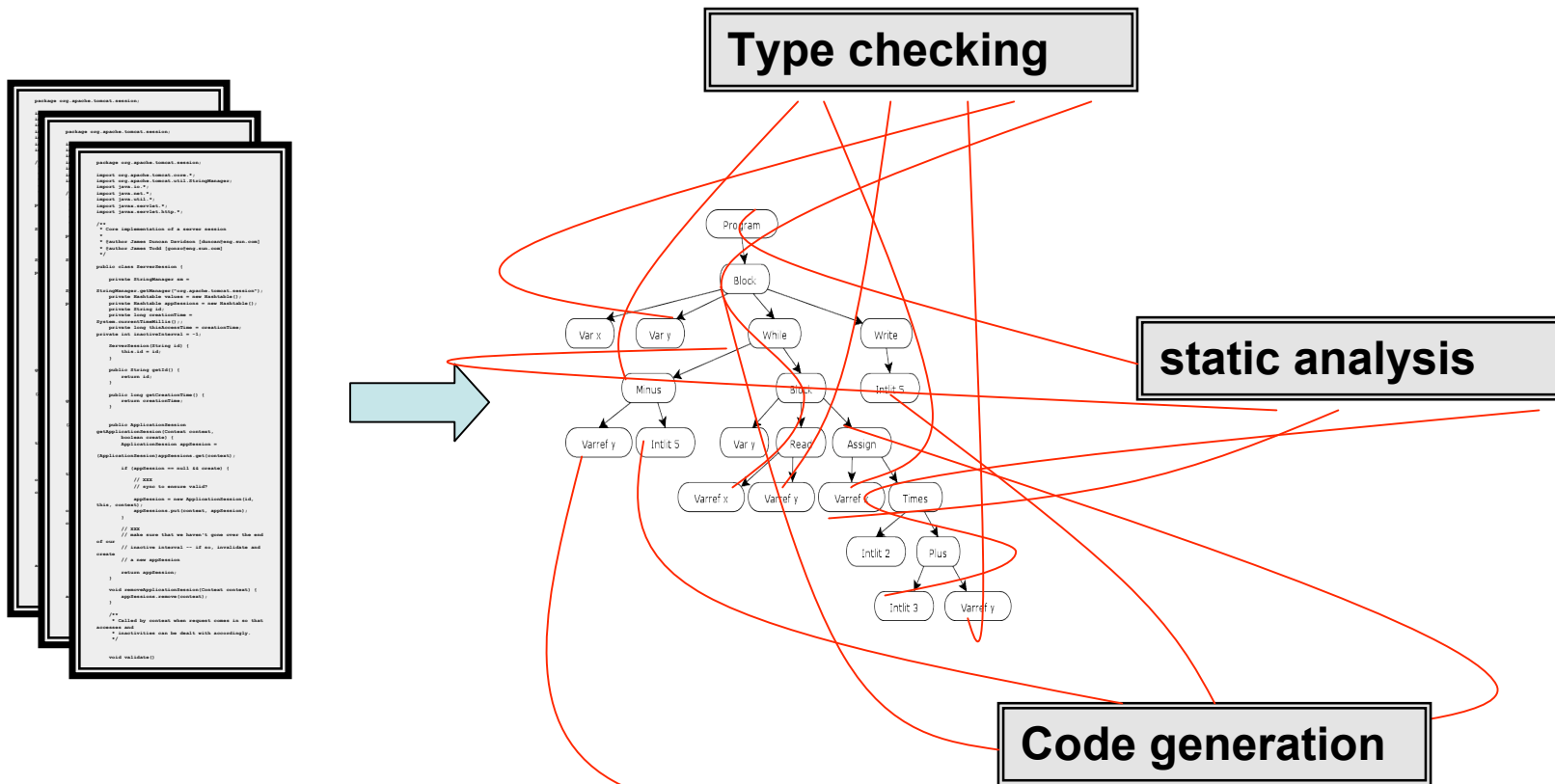
Separation of Concerns in Compiler Development Using Aspect-Orientation



**Carl (Xiaoqing) Wu, Barrett R. Bryant,
Jeff Gray and Suman Roychoudhury
University of Alabama at Birmingham**

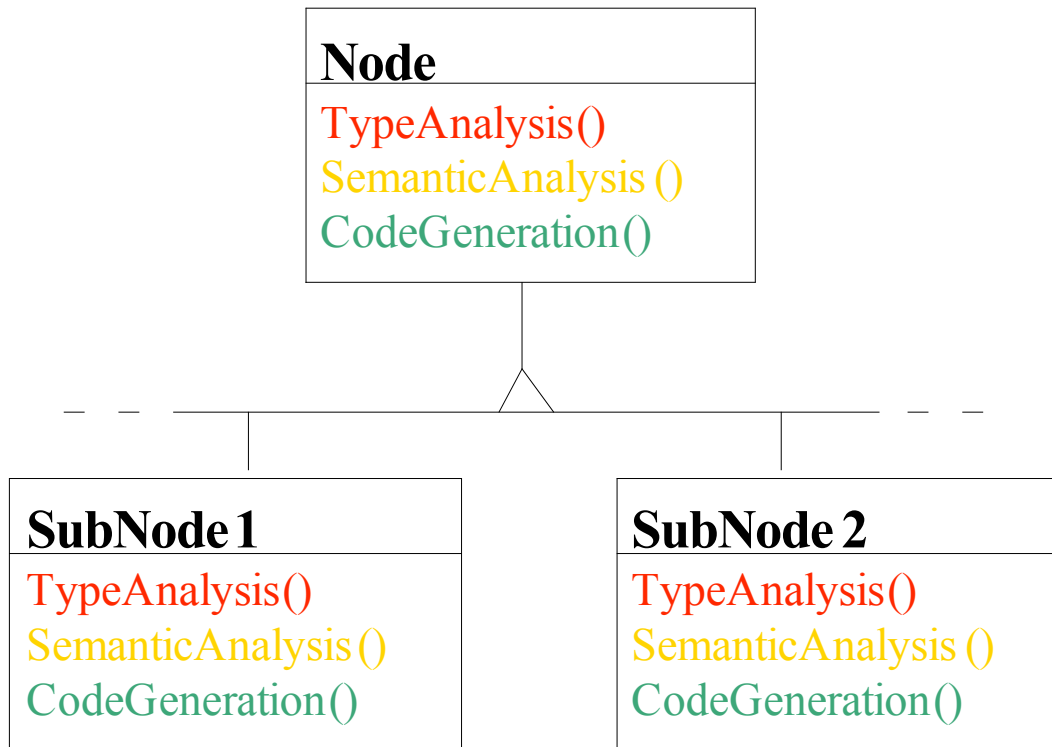
**Marjan Mernik
University of Maribor**

The Problem: Modularization of Compiler Phases



Compiler implementation is often an intricate task due to the complexity and interconnected nature of various stages within the compiler.

Pure Object-Oriented Design



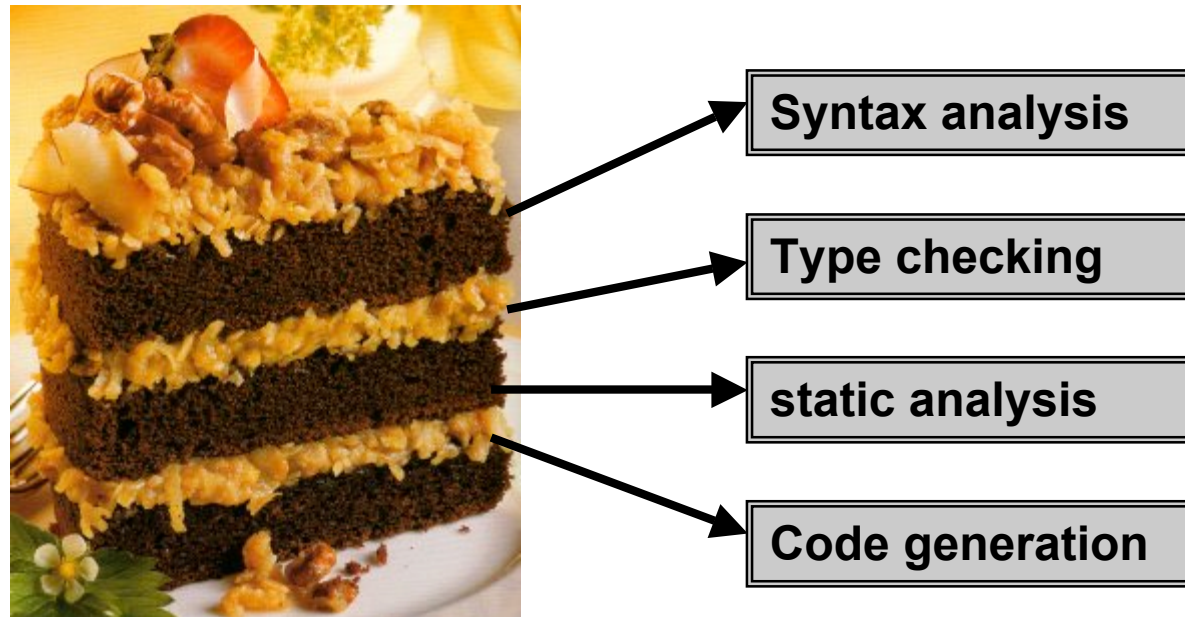
Problems

1. Code scatters all over the classes.
2. Hard to maintain and evolve.



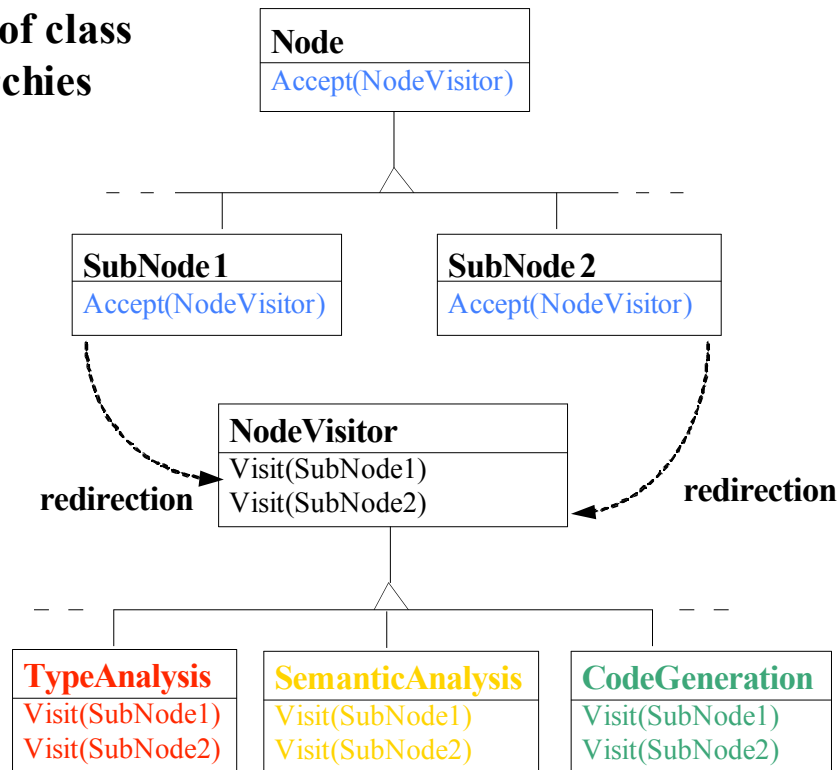
The Goal: Separation of Concerns

The key challenge is to provide exceptional modularity that assists in properly separating several crosscutting concerns, which not only helps the developer to divide-and-conquer the complexity, but also improves the readability, reusability and extensibility of the compiler implementation.



Popular Solution: the Visitor Pattern

two sets of class hierarchies



Problems

1. Introduce a lot of extra code.
2. Forces all concrete visitors share the same interface.
3. New semantics are always introduced by traversal of the whole tree.
4. Cannot access non-public members of a node class.

New solution: Aspect-Oriented Programming

- Aspect-Oriented Programming provides special language constructs called aspects to modularize crosscutting concerns in conventional program structures.
 - Introduction (inter-type declarations)
 - Interception (join-points)

**AOP is an excellent
programming technology to
apply to semantic analysis**

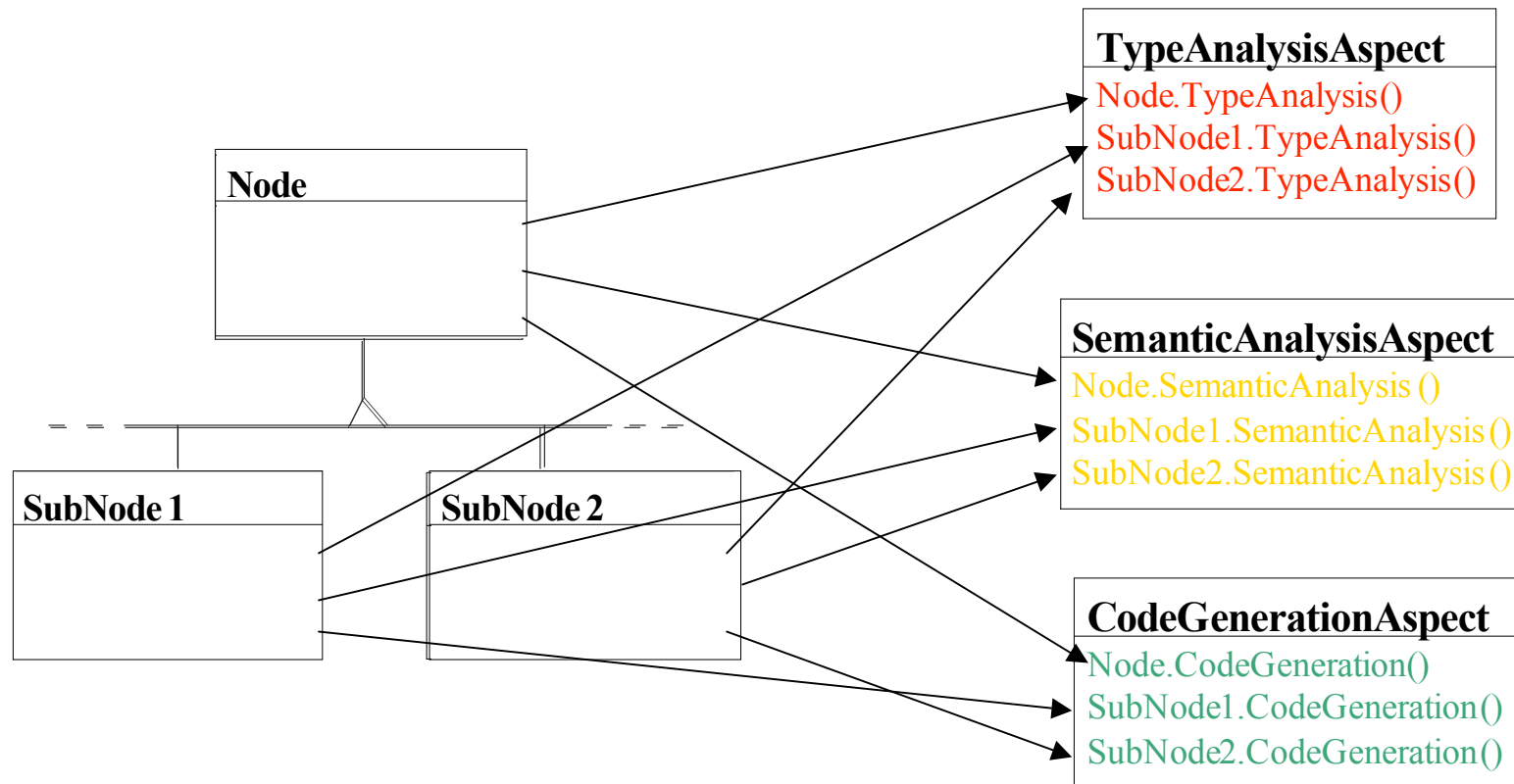
AOP: handling crosscutting concerns

Semantic phase: a concern that crosscuts various AST nodes

Aspect-Oriented Semantics Implementation

- Each concern is modularized as an aspect
 - An independent semantic pass
 - A group of action codes
- Semantic pass (i.e., a visitor)
 - Implemented as introductions of the AST classes
- Crosscutting actions applied to a group of nodes
 - Weaved into AST classes as interceptions.

Introduction



Interception

parser.cup

```

.....*/
parser_code_part ::=
.....
PARSER CODE
CODE_STRING:user_code
opt_semi
{
  parser_code_part
  ::=
}
PARSER CODE
CODE_STRING:u
/* ser_code opt_semi
init_code ::=
{
  INIT WITH
CODE_STRING:user_code
opt_semi /* ser_code!
{
  =null
}
lexer_emit_error("
Redundant parser
code (skipping)");
.....
else /* save the user
scan_code :=
string
SCAN WITH
CODE_STRING:user_code
opt_semi /* ser_code =
ser_code
{
}
{
}
.....
symbol_list ::= symbol_list
symbol;...
.....
symbol ::=
init_code ::=
TERMINAL
INIT WITH
TYPE:CODE, STRING:u
ser_code opt_semi

declares_term
|
if
(emit_init_code:=u
TERMINAL
declares_term
lexer_emit_error("
Redundant init
code (skipping)");
non_terminal
else /* save the user
type_id */
declares_non_term
emit_init_code :=
user_code;
non_terminal
declares_non_term
|
/* error recovery: ...
productions ... sync on
semification */
scan_code ::=
SCAN WITH
CODE_STRING:u
ser_code opt_semi
{
}

```

```

TERMINAL
error
%{emit_scan_code!=null}
%{user_emit_error("Redundant scan code
(skippping)");
SEMI
%{save the user code %/

non_terminal := user_code;
error
:;
:;
:;
:;

SEMI
symbol_list ::= symbol_list symbol |
symbol;
/*
.....
declares_term ::=
term_name_list
symbol ::=
{
TERMINAL
SEMI
type_id
/*
.....
declares_non_term ::=
non_term_name_list
TERMINAL
{
declares_term
SEMI
pop_terminal
/*
.....
term_name_list ::=
term_name_list COMMA
new_term_id | new_term_id;
declares_pop_term
/*
.....
|
non_terminal
declares_non_term
|
/* error recovery productions -- sync on
semicolon */

TERMINAL
error
{;
/* reset the accumulated multipart name
to the empty string
multipart_name = new String();
:;
SEMI

```

```

non_term_name_list ::=
non_term_name_list
{
  COMMA
  /* reset the accumulated multipart
  name */
  non_term_name_list := new String();
}
.....
SEMI
/* reset the accumulated multipart
name */
precedence_list ::=
precedence_list | empty;
{
  precedence_list ::=
precedence_list | preced |
preced; term ::=
PRECEDENCE LEFT
term_name_list
{
  /* reset the accumulated multipart
  name */
  terminal_list SEMI
  multipart_name := new String();
  PRECEDENCE RIGHT
}
{
  terminal_list SEMI
SEMI
}
.....
PRECEDENCE
NONASSOC
}
declares_non_term ::=
terminal_list SEMI
non_term_name_list
{
  /* reset the accumulated multipart
  name */
  multipart_name := new String();
}
.....
SEMI
}
/*
.....
term_name_list ::= term_name_list
COMMA non_term_id |
non_term_id;
/*
.....
non_term_name_list ::=
non_term_name_list

```

Action.aj

```

if (emit.parser_code!=null)
    lexer.emit_error("Redundant parser code (skipping)");
else /* save the user included code string */
    emit.parser_code = user_code;

if (emit.init_code!=null)
    lexer.emit_error("Redundant init code (skipping)");
else /* save the user code */
    emit.init_code = user_code;

if (emit.scan_code!=null)
    lexer.emit_error("Redundant scan code (skipping)");
else /* save the user code */
    emit.scan_code = user_code;

/* reset the accumulated multipart name */
multipart_name = new String();

/* reset the accumulated multipart name */
multipart_name = new String();

/* reset the accumulated multipart name */
multipart_name = new String();

/* reset the accumulated multipart name */
multipart_name = new String();

update_precedence(assoc.left);

update_precedence(assoc.right);

:}

{;

update_precedence(assoc.nonassoc);

```

Benefits (I)

- 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.

Benefits (II)

- Inter-type declaration (Introduction)
 - Defined within class scope, direct access to AST node class members (include private members)
 - Preserve the object-oriented inheritance relationship.
- Join-point model (Interception)
 - Provides flexibility to insert semantic behaviors into AST nodes or parser
 - Avoid code duplication
 - Trace facility
- Introduction + Interception
 - Tree traversal
 - Phase combination

Inter-Type Declaration Example

```
class UnparseVisitor extends Visitor{  
    protected PrintStream out =  
        System.out;  
    public Object print(Node node){  
        // ...  
    }  
    // Other utility routines  
    // ...  
    public Object visit(Node node){  
        return print(node);  
    }  
    public Object visit(  
        ASTCompilationUnit node){  
        return print(node);  
    }  
    // same visit methods for another 83  
    nodes.  
    // ...  
}
```

```
aspect Unparse{  
    protected static PrintStream out =  
        System.out;  
    public static void print(Node node){  
        // ...  
    }  
    // other utility routines  
    // ...  
    public void Node.unparse(){  
        Unparse.print(this);  
    }  
}
```

Join-Point Model Example

pointcut scopeEvaluate(): target(ScopeNode+) && call (* *.evaluate()) ;

before() : scopeEvaluate(){

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

}

Executed each
time entering a
new scope

Occurred 46 times in a parser!

after() : scopeEvaluate(){

```
currentSymTab = (SymbolTable)symTabs.pop();
```

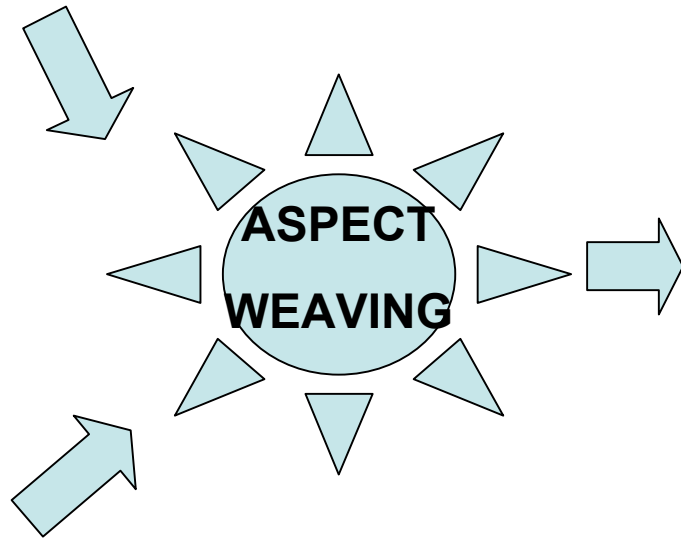
}

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

node.evaluate();

currentSymTab =
(SymbolTable)symTabs.pop();
...

```

```

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

```

Summary

- The major difficulty in compiler construction is separation of concerns.
- Traditional object-oriented design and the Visitor pattern cannot address the problem adequately.
- Using aspect-orientation to describe semantics supersedes both approaches without generating side-effects. Various AOP language features fulfill the computational needs of tree traversal.
- The proposed approach improves the overall modularization of the system and provides flexibility for future evolution of the compiler.



Thank you!

<http://www.cis.uab.edu/wuxi/>

wuxi@cis.uab.edu