

Lecture 12: AST visitors, ad-hoc semantic analysis

David Hovemeyer

October 4, 2025

601.428/628 Compilers and Interpreters



Agenda

- ▶ Semantic analysis
- ▶ AST visitors
- ▶ Ad-hoc semantic analysis, symbol tables
- ▶ An example

Semantic analysis

- ▶ Parser establishes whether or not the input source is *syntactically* value
- ▶ This does not guarantee that the input is *semantically* valid
 - ▶ E.g., `int x = "hello";`
- ▶ Semantic analysis:
 - ▶ Check that names refer to something valid
 - ▶ Check that operations performed are consistent with the source language's semantics

Formal vs. ad-hoc techniques

- ▶ With lexical analysis and parsing, formal techniques are very effective
 - ▶ Lexical analysis: regular languages, regular expressions, finite automata
 - ▶ Parsing: context-free grammars, parsing algorithms
- ▶ Formal approach to semantic analysis: *attribute grammars*
 - ▶ Never widely used, we will (probably) not cover them
- ▶ *Ad-hoc semantic analysis*: write ad-hoc code to check semantic properties
 - ▶ Could execute during parsing
 - ▶ Could execute on a representation of the input source (i.e., the AST)

AST visitors

Doing a computation on a tree

// approach 1

```
void TreeComputation::process_tree(Node *n) {  
    switch (n->get_tag()) {  
        case NODE_TAG_1:  
            ...code to handle NODE_TAG_1...  
            ...recursively process children...  
            break;  
        case NODE_TAG_2:  
            ...code to handle NODE_TAG_2...  
            ...recursively process children...  
            break;  
        ...etc...  
    }  
}
```

Doing a computation on a tree

// approach 2

```
void TreeComputation::process_tree(Node *n) {  
    switch (n->get_tag()) {  
        case NODE_TAG_1:  
            visit_node_tag_1(n); // will also process children  
            break;  
        case NODE_TAG_2:  
            visit_node_tag_2(n); // will also process children  
            break;  
        ...etc...  
    }  
}
```

Observation

- ▶ Lots of repetitive code
- ▶ Second approach is nice in that each kind of tree node is handled by a dedicated function
 - ▶ But the big switch statement is still tedious and error-prone code
- ▶ Also: what if we have multiple tree computations?
 - ▶ Potential for duplicated code

Visitor design pattern

- ▶ Idea: abstract the traversal and dispatching to per-node-type functions into a base class
- ▶ Derived classes then only need to override the per-node-type member functions as necessary

- ▶ `ASTVisitor`: a base class for implementations of tree computations on the AST
 - ▶ Assignment 3: `SemanticAnalysis`
 - ▶ Assignment 4: high-level code generation

ASTVisitor

```
class ASTVisitor {
public:
    ASTVisitor();
    virtual ~ASTVisitor();

    virtual void visit(Node *n); // <-- switch statement is here
    virtual void visit_unit(Node *n);
    virtual void visit_variable_declaration(Node *n);
    ...many others...

    virtual void visit_children(Node *n); // <-- recursively visit children
    virtual void visit_token(Node *n);
};
```

General recursive treewalk

- ▶ The default behavior of each node-specific visit function is to call `visit_children`
- ▶ This means that the default behavior of any class derived from `ASTVisitor` is a general recursive treewalk of the AST
- ▶ Which is why a derived visitor class can just override the visit functions that it actually cares about

Defining a visit function

Note that if you override a node-specific visit function, then it's up to you to decide whether and how to visit children.

Example:

```
void SemanticAnalysis::visit_variable_declaration(Node *n) {  
    // visit the base type  
    visit(n->get_kid(1));  
    std::shared_ptr<Type> base_type = n->get_kid(1)->get_type();  
  
    // iterate through declarators, adding variables  
    // to the symbol table  
    Node *decl_list = n->get_kid(2);  
    for (auto i = decl_list->cbegin(); i != decl_list->cend(); ++i) {  
        Node *declarator = *i;  
        // ...handle the declarator...  
    }  
}
```

Where results go

- ▶ The most straightforward way to record results is to store them *in the visited tree node*
- ▶ For example:
 - ▶ Store a pointer to a symbol table entry in a node representing a reference to a variable or function
 - ▶ Store a (shared) pointer to the Type object representing the type of an expression
 - ▶ Store a boolean value indicating whether or not an expression yields an lvalue

The purpose of the `NodeBase` class is to give you a place to define new member variables and member functions for AST nodes.

The reason we don't recommend that you modify `Node` directly is that we might want to give you a new version. Putting your changes in `NodeBase` means you never need to modify `Node`.

Propagation of values

- ▶ Propagating values *upwards* in the tree is generally easy, because the parent has links to its children
 - ▶ Recursively visit children, then make use of computed values stored in them
- ▶ Propagating values *downwards* is more difficult because child nodes don't link back to the parent
- ▶ Fortunately, upwards tends to be the most natural direction
- ▶ For the rare cases of propagating values downwards, various strategies can work
 - ▶ E.g., store a value in the child node before visiting, then let the child update the value

Ad-hoc semantic analysis, symbol tables

Semantic analysis, symbol tables

Two of the main concerns of semantic analysis:

1. Determine what each name refers to
2. Determine a type for each expression

Building *symbol tables* is the classic approach to performing semantic analysis

SymbolTable = Environment

- ▶ If you're comfortable with the notion of “environment” from the interpreter project, a symbol table is more or less the same thing
 - ▶ Represents a scope in the program
 - ▶ Stores information about what names in that scope refer to
 - ▶ Can have a “parent” representing the enclosing scope
- ▶ The main difference is that `Environment` kept track of a runtime value for each name, while `SymbolTable` will keep track of information about a variable, function, or data type

Symbol class

```
// represents one symbol table entry
class Symbol {
private:
    SymbolKind m_kind;
    std::string m_name;
    std::shared_ptr<Type> m_type;
    SymbolTable *m_syntab;
    bool m_is_defined;

public:
    // constructor, member functions...
};
```

Symbol tables example

```
struct Point {  
    int x, y;  
};
```

```
void move_horiz(struct Point *p,  
               int dx) {  
    int u;  
    u = p->x + dx;  
    p->x = u;  
}
```

Global scope ☐	→	<table><thead><tr><th>Name</th><th>Kind</th><th>Type</th></tr></thead></table>	Name	Kind	Type
Name	Kind	Type			

Symbol tables example

```
struct Point {  
    int x, y;  
};
```

```
void move_horiz(struct Point *p,  
                int dx) {  
    int u;  
    u = p->x + dx;  
    p->x = u;  
}
```

Global scope ☐

Name	Kind	Type
struct Point	Type	struct { }

Name	Kind	Type
------	------	------

Create entry and symbol
table for the struct Point
data type

Symbol tables example

```
struct Point {  
    int x, y;  
};
```

```
void move_horiz(struct Point *p,  
                int dx) {  
    int u;  
    u = p->x + dx;  
    p->x = u;  
}
```

Global scope ☐

Name	Kind	Type
struct Point	Type	struct Point { }

Name	Kind	Type
x	Var	int
y	Var	int

Entries for members of
struct Point are added
to its symbol table

Symbol tables example

```
struct Point {  
    int x, y;  
};
```

```
void move_horiz(struct Point *p,  
                int dx) {  
    int u;  
    u = p->x + dx;  
    p->x = u;  
}
```

Global scope ☐

Name	Kind	Type
struct Point	Type	struct Point {x:int,y:int}

Name	Kind	Type
x	Var	int
y	Var	int

Full representation of
struct Point is now
known

Symbol tables example

```
struct Point {  
    int x, y;  
};
```

```
void move_horiz(struct Point *p,  
               int dx) {  
    int u;  
    u = p->x + dx;  
    p->x = u;  
}
```

Add entry for move_horiz
function, create symbol
table for its parameters
and body

Global scope ☐

Name	Kind	Type
struct Point	Type	struct Point {x:int,y:int}
move_horiz	Func	(ptr to struct Point × int) → void

Name	Kind	Type
x	Var	int
y	Var	int

Name	Kind	Type
p	Var	ptr to struct Point {x:int,y:int}
dx	Var	int

Symbol tables example

```
struct Point {  
    int x, y;  
};
```

```
void move_horiz(struct Point *p,  
               int dx) {  
    int u;  
    u = p->x + dx;  
    p->x = u;  
}
```

Visit function body,
add symbol table entries
for local variable(s)

Global scope ☐

Name	Kind	Type
struct Point	Type	struct Point {x:int,y:int}
move_horiz	Func	(ptr to struct Point × int) → void

Name	Kind	Type
x	Var	int
y	Var	int

Name	Kind	Type
p	Var	ptr to struct Point {x:int,y:int}
dx	Var	int
u	Var	int

Symbol tables example

```
struct Point {  
    int x, y;  
};
```

```
void move_horiz(struct Point *p,  
               int dx) {  
    int u;  
    u = p->x + dx;  
    p->x = u;  
}
```

Global scope ☐

Name	Kind	Type
struct Point	Type	struct Point {x:int,y:int}
move_horiz	Func	(ptr to struct Point × int) → void

Name	Kind	Type
x	Var	int
y	Var	int

Name	Kind	Type
p	Var	ptr to struct Point {x:int,y:int}
dx	Var	int
u	Var	int

Variable references can
be annotated with
pointers to symbol table
entries

Type checking

Type checking: based on the types of variables and literals, check each operation in the program to make sure the operand types are consistent with the language's semantic rules

Because C requires a declaration or definition to precede each use (for variables, functions, and types), the symbol table should have information about referenced names at the point of their use

Type checking examples

```
struct Point {  
    int x, y;  
};  
  
void foo(struct Point *p) {  
    int n;  
    n = 3;  
    q->x = n;  
}
```

'q' is not defined in any
currently-visible scope

Name	Kind	Type
struct Point	Type	struct Point {x:int,y:int}
foo	Func	(ptr to struct Point{x:int,y:int}) → void

Name	Kind	Type
p	Var	ptr to struct Point{x:int,y:int}
n	Var	int

Type checking examples

```
struct Point {  
    int x, y;  
};  
  
void foo(struct Point *p) {  
    int n;  
    n = 3;  
    p->z = n;  
}
```

Name	Kind	Type
struct Point	Type	struct Point {x:int,y:int}
foo	Func	(ptr to struct Point{x:int,y:int}) → void

Name	Kind	Type
p	Var	ptr to struct Point{x:int,y:int}
n	Var	int

'p' is a pointer to a struct type, but that struct type doesn't have a member named 'z'

Type checking examples

```
struct Point {  
    int x, y;  
};
```

```
void foo(struct Point *p) {  
    int n;  
    n = 3;  
    p->x = &n;  
}
```

Name	Kind	Type
struct Point	Type	struct Point {x:int,y:int}
foo	Func	(ptr to struct Point{x:int,y:int}) → void

Name	Kind	Type
p	Var	ptr to struct Point{x:int,y:int}
n	Var	int

Assignment of pointer
to int variable: 'p->x'
is an int lvalue, '&n'
is a pointer to int rvalue

Type checking examples

```
struct Point {  
    int x, y;  
};
```

```
void foo(struct Point *p) {  
    int n;  
    n = 3;  
    *p = n;  
}
```

Name	Kind	Type
struct Point	Type	struct Point {x:int,y:int}
foo	Func	(ptr to struct Point{x:int,y:int}) → void

Name	Kind	Type
p	Var	ptr to struct Point{x:int,y:int}
n	Var	int

Assignment of int rvalue
to struct Point lvalue
(types are not compatible)

Type checking examples

```
struct Point {  
    int x, y;  
};  
  
void foo(struct Point *p) {  
    int n;  
    n = 3;  
    p->x = n[0];  
}
```

Name	Kind	Type
struct Point	Type	struct Point {x:int,y:int}
foo	Func	(ptr to struct Point{x:int,y:int}) → void

Name	Kind	Type
p	Var	ptr to struct Point{x:int,y:int}
n	Var	int

'n' is neither a pointer
nor an array

Semantic analysis and type checking

To conclude:

- ▶ The semantic analyzer builds symbol tables recording the name and type of each variable, function, and struct type
- ▶ The symbol tables can be used to check that each operation in the code follows the source language's semantic rules
- ▶ The symbol tables will also be useful (and necessary) for storage allocation and code generation

An example

An example

```
int sq(int *p) {  
    int x;  
    x = *p;  
}
```

```
int main(void) {  
    int a;  
    a = 3;  
    sq(&a);  
    return a;  
}
```

