CS412/CS413

Introduction to Compilers
Tim Teitelbaum

Lecture 12: Visitors; Symbol Tables
February 18, 2005

Abstract Syntax Trees

- Separate AST construction from semantic checking phase
- Traverse the AST and perform semantic checks (or other actions) only after the tree has been built and its structure is stable
- This approach is less error-prone
  - It is better when efficiency is not a critical issue

Visitor Methodology for AST Traversal

- Visitor pattern: useful OO programming pattern that separates data structure definition (e.g., the AST) from code that traverses the structure (e.g., the name resolution code and the type checking code).
- Define a Visitor interface for all traversals of the AST
- Extend each AST class with a method that accepts any Visitor
- Code each traversal as a separate class that implements the Visitor interface

AST Data Structure

```
abstract class Expr { …  }
class Add extends Expr { …
   Expr e1, e2
   }
class Num extends Expr { …
   int value
   }
class Id extends Expr { …
   String name
   }
```


Visitor Interface

```
interface Visitor {
   void visit(Add e);
   void visit(Num e);
   void visit(Id e);
}
```


Accept methods

```
abstract class Expr { …
    abstract public void accept(Visitor v);
    }
class Add extends Expr { …
    public void accept(Visitor v) {
        v.visit(this); }
    }
class Num extends Expr { …
    public void accept(Visitor v) {
        v.visit(this); }
    }
class Id extends Expr { …
    public void accept(Visitor v) {
        v.visit(this); }
    }
```

The declared type of **this** is the subclass it which it occurs.

Overload resolution of v.visit(**this**); invokes appropriate visit function in the Visitor.

Visitor Methods

- For each kind of traversal, implement the Visitor interface, e.g.,

```
class PostfixOutputVisitor implements Visitor {
    void visit(Add e)  {
        e.e1.accept(this); e.e2.accept(this); System.out.print( "+" );
        }
    void visit(Num e)  {
        System.out.print(value);
        }
    void visit(Id e)  {
        System.out.print(id);
        }
}
```

Dispatch in the visit methods eliminates case analysis on AST subclasses

- To traverse expression e:

```
PostfixOutputVisitor v = new PostfixOutputVisitor();
e.accept(v);
```


Inherited and Synthesized Information

- So far, OK for traversal and action w/o communication of values
- But we need a way to pass information
  - Down the AST (inherited)
  - Up the AST (synthesized)
- To pass information down the AST
  - add parameter to visit functions
- To pass information up the AST
  - add return value to visit functions

Visitor Interface (2)

```
interface Visitor {
  Object visit(Add e, Object inh);
  Object visit(Num e, Object inh);
  Object visit(Id e, Object inh);
}
```


Accept methods (2)

```
abstract class Expr { …
    abstract public Object accept(Visitor v, Object inh);
    }
class Add extends Expr { …
    public Object accept(Visitor v, Object inh) {
        return v.visit(this, inh); }
     }
class Num extends Expr { …
    public Object accept(Visitor v, Object inh) {
        return v.visit(this, inh);
    }
class Id extends Expr { …
    public Object accept(Visitor v, Object inh) {
        return v.visit(this, inh); }
    }
}
```


Visitor Methods (2)

- For kind of traversal, implement the Visitor interface, e.g.,

```
class EvaluationVisitor implements Visitor {
    Object visit(Add e, Object inh)  {
        int left = (int) e.e1.accept(this, inh);
        int right = (int) e.e2.accept(this, inh);
        return left+right;
        }
    Object visit(Num e, Object inh)  {
        return value;
        }
    Object visit(Id e, Object inh)  {
        return Lookup(id, (SymbolTable)inh);
        }
}
```

- To traverse expression e:

```
EvaluationVisitor v = new EvaluationVisitor ();
e.accept(v);
```


Incorrect Programs

- Lexically and syntactically correct programs may still contain other errors!
- Lexical and syntax analysis are not powerful enough to ensure the correct usage of variables, objects, functions, statements, etc.

Incorrect Programs

- Example 1: lexical analysis does not distinguish between different variable or function identifiers (it returns the same token for all identifiers)

int a;	int a;
a = 1;	b = 1;

- Example 2: syntax analysis does not correlate the declarations with the uses of variables in the program:

int a;	
a = 1;	a = 1;

- Example 3: syntax analysis does not correlate the types from the declarations with the uses of variables:

int a;	int a;
a = 1;	a = 1.0;

Goals of Semantic Analysis

- Semantic analysis = ensure that the program satisfies a set of rules regarding the usage of programming constructs (variables, objects, expressions, statements)
- Examples of semantic rules:
  - Variables must be declared before being used
  - A variable should not be declared multiple times in the same scope
  - In an assignment statement, the variable and the assigned expression must have the same type
  - The condition of an if-statement must have type boolean
- Some categories of rules:
  - Semantic rules regarding types
  - Semantic rules regarding scopes

Type Information

- Type information = describes what kind of values correspond to different constructs: variables, statements, expressions, functions

variables:	int a;	integer
expressions:	(a+1) == 2	boolean
statements:	a = 1.0	floating-point
functions:	int pow(int n, int m)	int x int $\rightarrow$ int

Type Checking

- Type checking = set of rules that ensure the type consistency of different constructs in the program
- Examples:
  - The type of a variable must match the type from its declaration
  - The operands of arithmetic expressions (+, *, -, /) must have integer types; the result has integer type
  - The operands of comparison expressions (==, !=) must have integer or string types; the result has boolean type

Type Checking

- More examples:
  - For each assignment statement, the type of the updated variable must match the type of the expression being assigned
  - For each call statement foo(v_1, …, v_n), the type of each actual argument v_i must match the type of the corresponding formal argument f_i from the declaration of function foo
  - The type of the return value must match the return type from the declaration of the function
- Type checking: next two lectures.

Scope Information

- Scope information = characterizes the declaration of identifiers and the portions of the program where it is allowed to use each identifier
  - Example identifiers: variables, functions, objects, labels
- Lexical scope = textual region in the program
  - Statement block
  - Formal argument list
  - Object body
  - Function or method body
  - Module body
  - Whole program (multiple modules)
- Scope of an identifier: the lexical scope in which it is valid

Scope Information

- Scope of variables in statement blocks:

```
{ int a;
  ...
    { int b;
    }
  ...
}
```

scope of variable a

scope of variable b

- In C:
  - Scope of file static variables: current file
  - Scope of external variables: whole program
  - Scope of automatic variables, formal parameters, and function static variables: the function

Scope Information

- Scope of formal arguments of functions/methods:

```
int factorial(int n) {
    ...
}
```

scope of formal parameter n

- Scope of labels:

```
void f() {
    ... goto l; ...
    l: a =1;
    ... goto l; ...
}
```

scope of label l

Scope Information

- Scope of object fields and methods:

```
class A {
    private int x;
    public  void g() { x=1; }
    ...
}
class B extends A {
    ...
    public int h() { g(); }
    ...
}
```

scope of field x

scope of method g

Semantic Rules for Scopes

- Main rules regarding scopes:
  Rule 1: Use an identifier only if defined in enclosing scope
  Rule 2: Do not declare identifiers of the same kind with identical names more than once in the same lexical scope
- Can declare identifiers with the same name with identical or overlapping lexical scopes if they are of different kinds

```
class X {
  int X;
  void X(int X) {
    X: for(;;)
       break X;
  }
}
```

```
int X(int X) {
  int X;
  goto X;
  { int X;
    X: X = 1; }
}
```

Not Recommended!

Symbol Tables

- Semantic checks refer to properties of identifiers in the program -- their scope or type
- Need an environment to store the information about identifiers = symbol table
- Each entry in the symbol table contains
  - the name of an identifier
  - additional information: its kind, its type, if it is constant, …

NAME	KIND	TYPE	ATTRIBUTES
foo	fun	int x int $\rightarrow$ bool	extern
m	arg	int	
n	arg	int	const
tmp	var	bool	const

Scope Information

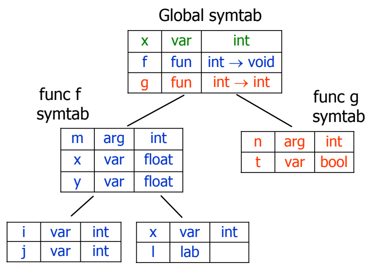
- How to capture the scope information in the symbol table?
- Idea:
  - There is a hierarchy of scopes in the program
  - Use a similar hierarchy of symbol tables
  - One symbol table for each scope
  - Each symbol table contains the symbols declared in that lexical scope

Example

```
int x;

void f(int m) {
   float x, y;
   ...
   { int i, j; ...; }
   { int x; l: ...; }
}

int g(int n) {
   bool t;
   ...;
}
```



Identifiers With Same Name

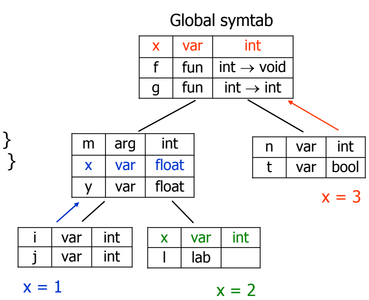
- The hierarchical structure of symbol tables automatically solves the problem of resolving name collisions (identifiers with the same name and overlapping scopes)
- To find which is the declaration of an identifier that is active at a program point:
  - Start from the current scope
  - Go up in the hierarchy until you find an identifier with the same name

Example

```
int x;

void f(int m) {
   float x, y;
   ...
   { int i, j; x = 1; }
   { int x; l: x = 2; }
}

int g(int n) {
   bool t;
   x = 3;
}
```

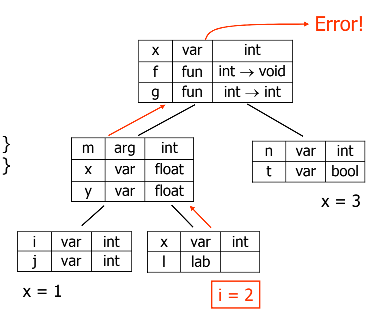


Catching Semantic Errors

```
int x;

void f(int m) {
   float x, y;
   ...
   { int i, j; x = 1; }
   { int x; l: i = 2; }
}

int g(int n) {
   bool t;
   x = 3;
}
```



Symbol Table Operations

- Two operations:
  - To build symbol tables, we need to insert new identifiers in the table
  - In the subsequent stages of the compiler we need to access the information from the table: use a lookup function
- Cannot build symbol tables during lexical analysis
  - hierarchy of scopes encoded in the syntax
- Build the symbol tables:
  - while parsing, using the semantic actions
  - After the AST is constructed

Array Implementation

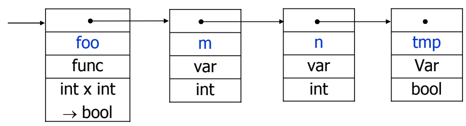
- Simple implementation = array
  - One entry per symbol
  - Scan the array for lookup, compare name at each entry

foo	fun	int x int → bool
m	arg	int
n	arg	int
tmp	var	bool

- Disadvantage:
  - table has fixed size
  - need to know in advance the number of entries

List Implementation

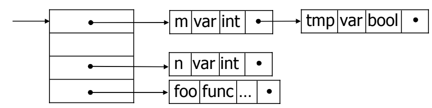
- Dynamic structure = list
  - One cell per entry in the table
  - Can grow dynamically during compilation



- Disadvantage: inefficient for large symbol tables
  - need to scan half the list on average

Hash Table Implementation

- Efficient implementation = hash table
  - It is an array of lists (buckets)
  - Uses a hashing function to map the symbol name to the corresponding bucket: hashfunc : string → int
  - Good hash function = even distribution in the buckets



- hashfunc("m") = 0, hashfunc("foo") = 3

Forward References

- Forward references = use an identifier within the scope of its declaration, but before it is declared
- Any compiler phase that uses the information from the symbol table must be performed after the table is constructed
- Cannot type-check and build symbol table at the same time
- Example:

```
class A {
  int m() { return n(); }
  int n() { return 1; }
}
```


Summary

- Semantic checks ensure the correct usage of variables, objects, expressions, statements, functions, and labels in the program
- Scope semantic checks ensure that identifiers are correctly used within the scope of their declaration
- Type semantic checks ensures the type consistency of various constructs in the program
- Symbol tables: a data structure for storing information about symbols in the program
  - Used in semantic analysis and subsequent compiler stages
- Next time: type-checking