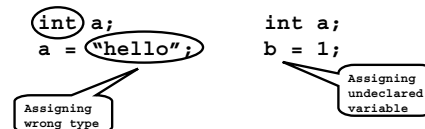


Winter 2007-2008 Compiler Construction T6 – semantic analysis part I scopes and symbol tables

Mooly Sagiv and Roman Manevich
School of Computer Science
Tel-Aviv University

Semantic analysis motivation

- Syntactically correct programs may still contain errors
 - Lexical analysis does not distinguish between different variable names (same ID token)
 - Syntax analysis does not correlate variable declaration with variable use, does not keep track of types



4

Some notes

- PA1 submissions will be returned soon
 - Use comments to fix scanner if needed
- Notes on PA2
 - Don't use CUP's **expect** switch
 - Join Library AST to main AST
- New electronic submission guidelines
 - Please keep stable code in submission directory (under PA2/PA3/PA4)
 - Unstable code in your own temp directory
- Read and understand solution to TA1
- Exam date: 30/4

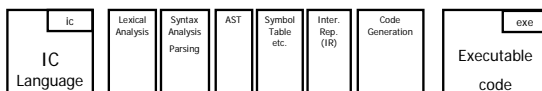
2

Goals of semantic analysis

- Check "correct" use of programming constructs
- Provide information for subsequent phases
- Context-sensitive – beyond context free grammars
 - Lexical analysis and syntax analysis provide relatively shallow checks of program structure
 - Semantic analysis goes deeper
- Correctness specified by semantic rules
 - Scope rules
 - Type-checking rules
 - Specific rules
- Note: semantic analysis ensures only partial correctness of programs
 - Runtime checks (pointer dereferencing, array access)

5

Today



- Today:
 - Scopes
 - Symbol tables
 - (Type table)
- Next week:
 - Types
 - Type-checking
 - More semantic analysis

3

Example of semantic rules

- A variable must be declared before used
- A variable should not be declared multiple times
- A variable should be initialized before used
- Non-void method should contain return statement along all execution paths
- **break/continue** statements allowed only in loops
- **this** keyword cannot be used in static method
- **main** method should have specific signature
- ...
- Type rules are important class of semantic rules
 - In an assignment statement, the variable and assigned expression must have the same type
 - In a condition test expression must have boolean type

6

Scope and visibility

- Scope (visibility) of identifier = portion of program where identifier can be referred to
- Lexical scope = textual region in the program
 - Statement block
 - Method body
 - Class body
 - Module / package / file
 - Whole program (multiple modules)

7

Scope hierarchy in IC

- Global scope
 - The names of all classes defined in the program
- Class scope
 - Instance scope: all fields and methods of the class
 - Static scope: all static methods
 - Scope of subclass nested in scope of its superclass
- Method scope
 - Formal parameters and local variables in code block of body method
- Code block scope
 - Variables defined in block

10

Scope example

```

class Foo {
  int value;
  int test() {
    int b = 3;
    return value + b;
  }
  void setValue(int c) {
    value = c;
    { int d = c;
      c = c + d;
      value = c;
    }
  }
}

class Bar extends Foo {
  int value;
  void setValue(int c) {
    value = c;
    test();
  }
}
    
```

Diagram illustrating scope boundaries with curly braces:

- scope of local variable b (points to `int b = 3;`)
- scope of field value (points to `int value;`)
- scope of local variable in statement block d (points to `{ int d = c; ... }`)
- scope of formal parameter c (points to `int c`)
- scope of method test (points to `int test() { ... }`)
- scope of c (points to `int c` in `Bar.setValue`)
- scope of value (points to `int value;` in `Bar`)

8

Scope rules in IC

- “When resolving an identifier at a certain point in the program, the enclosing scopes are searched for that identifier.”
- “local variables and method parameters can only be used after they are defined in one of the enclosing block or method scopes.”
- “Fields and virtual methods can be used in expressions of the form *e.f* or *e.m()* when *e* has class type *C* and the instance scope of *C* contains those fields and methods.”
- “static methods can be used in expressions of the form *C.m()* if the static scope of *C* contains *m*.”
- ... (Section 10 in IC specification)
- How do we check these rules?

11

Scope nesting

- Scopes may be enclosed in other scopes


```
void foo() { int a; ... {int a;} }
```
- Name disambiguation
 - same name but different symbol
- Generally scope hierarchy forms a tree
- Scope of subclass enclosed in scope of its superclass
 - Subtype relation must be acyclic

9

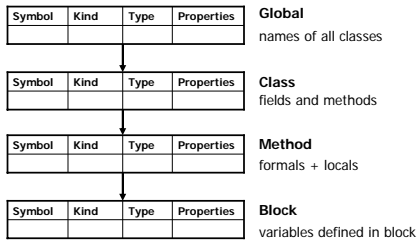
Symbol table

- An environment that stores information about identifiers
- A data structure that captures scope information
- Each entry in symbol table contains
 - The name of an identifier
 - Its kind (variable/method/field...)
 - Type
 - Additional properties, e.g., `final`, `public` (not needed for IC)
- One symbol table for each scope

12

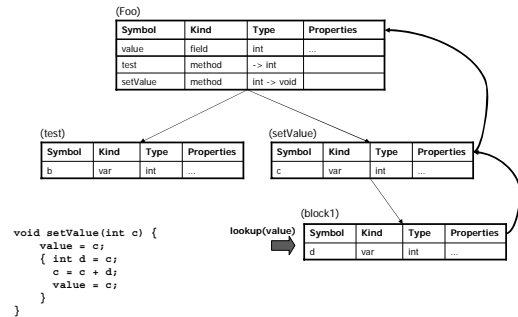
Scope nesting in IC

Scope nesting mirrored in hierarchy of symbol tables



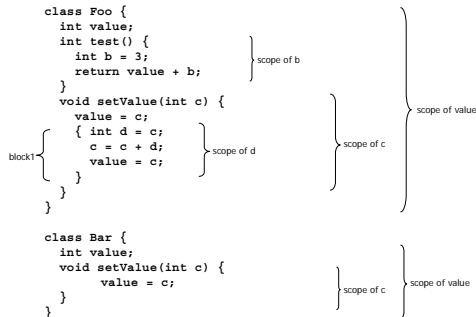
13

Checking scope rules



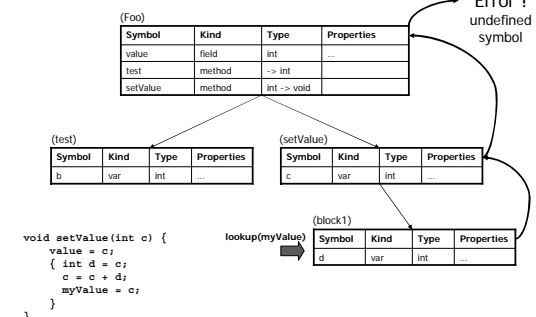
16

Symbol table example



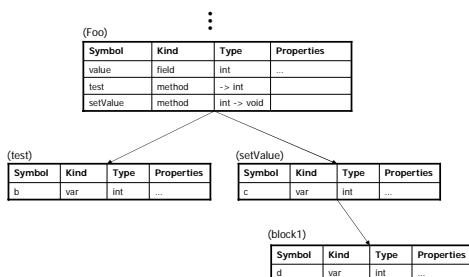
14

Catching semantic errors



17

Symbol table example cont.



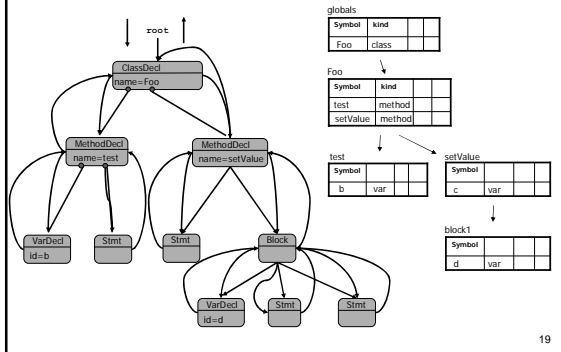
15

Symbol table operations

- insert
 - Insert new symbol (to current scope)
- lookup
 - Try to find a symbol in the table
 - May cause lookup in parent tables
 - Report an error when symbol not found
- How do we check illegal re-definitions?

18

Symbol table construction via AST traversal



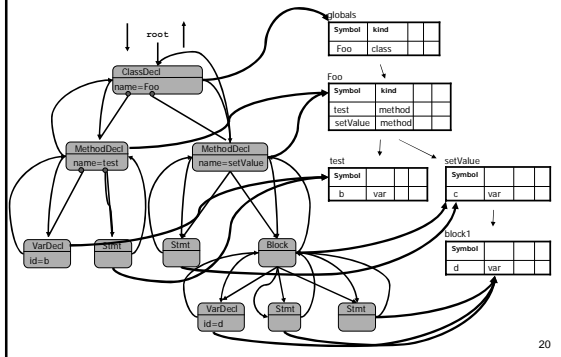
19

Symbol table implementation

- Each table in the hierarchy could be implemented using `java.util.HashMap`
- Implement a hierarchy of symbol tables
- Can implement a `Symbol` class
 - Use in subsequent phases instead of id name
- A programming note:
 - `HashMap` keys should obey `equals/hashcode` contracts
 - Safe when key is symbol name (`String`)

22

Linking AST nodes to enclosing table



20

Symbol table implementation

```
public class SymbolTable {
    /** map from String to Symbol */
    private Map<String, Symbol> entries;
    private String id;
    private SymbolTable parentSymbolTable;
    public SymbolTable(String id) {
        this.id = id;
        entries = new HashMap<String, Symbol>();
    }
    ...
}

public class Symbol {
    private String id;
    private Type type;
    private Kind kind;
    ...
}
```

(this is only a suggestion)

23

What's in an AST node – take 2

```
public abstract class ASTNode {
    /** line in source program */
    private int line;

    /** reference to symbol table of enclosing scope */
    private SymbolTable enclosingScope;

    /** accept visitor */
    public abstract void accept(Visitor v);

    /** accept propagating visitor */
    public abstract <D,U> U accept(PropagatingVisitor<D,U> v, D context);

    /** return line number of this AST node in program */
    public int getLine() {...}

    /** returns symbol table of enclosing scope */
    public SymbolTable enclosingScope() {...}
}
```

21

Implementing table structure

- Hierarchy of symbol tables
 - Pointer to enclosing table
 - Can also keep list of sub-tables
- Symbol table key should include id and kind
 - Can implement using 2-level maps (kind->id->entry)
 - Separating table in advance according to kinds also acceptable

24

Implementation option 1

```
public class SymbolTable {
    /** Map kind->(id->entry)
        Kind enum->(String->Symbol)
    **/
    private Map<Kind, Map<String, Symbol> > entries;
    private SymbolTable parent;
    ...
    public Symbol getMethod(String id) {
        Map<String, Symbol> methodEntries = entries.get(METHOD_KIND);
        return methodEntries.get(id);
    }
    public void insertMethod(String id, Type t) {
        Map<String, Symbol> methodEntries = entries.get(METHOD_KIND);
        if (methodEntries == null) {
            methodEntries = new HashMap<String, Symbol>();
            entries.put(METHOD_KIND, methodEntries);
        }
        methodEntries.put(id, new Symbol(id, t));
    }
    ...
}
```

25

Solution 1 – multiple phases

- Multiple phase solution
 - Building visitor
 - Checking visitor
- Building visitor
 - On visit to node – build corresponding symbol table
 - class, method, block (and possibly nested blocks)
 - Can maintain stack of symbol tables
 - Push new table when entering scope
 - Pop when exiting scope
 - Link AST node to symbol table of corresponding scope
 - Do not perform any checks

28

Implementation option 2

```
public class SymbolTable {
    /** Method Map id->entry **/
    private Map<String, Symbol> methodEntries;
    ...
    private Map<String, Symbol> variableEntries;
    ...
    private SymbolTable parent;
    public Symbol getMethod(String id, Type t) {
        return methodEntries.get(id);
    }
    ...
    public void insertMethod(String id, Type t) {
        methodEntries.put(new Symbol(id, METHOD_KIND, t));
    }
    ...
}
```

Less flexible, but acceptable

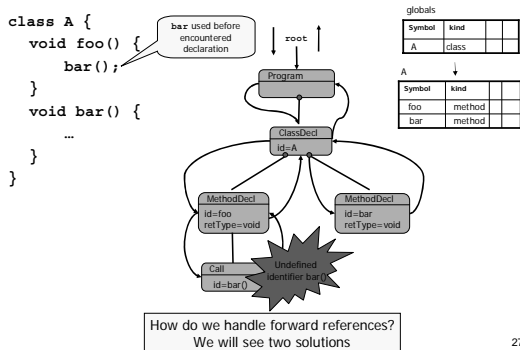
26

Building and checking

- Building visitor
 - A propagating visitor
 - Propagates reference to the symbol table of the current scope (or use table stack)
 - In some cases have to use type information (**extends**)
 - Populate tables with declarations/definitions
 - Class definitions, method definitions, field definitions, variable declarations, formal arguments
 - `class A {void foo() {int B; int[] C;}}`
- Checking visitor
 - On visit to node – perform check using symbol tables
 - Resolve identifiers
 - Look for symbol in table hierarchy

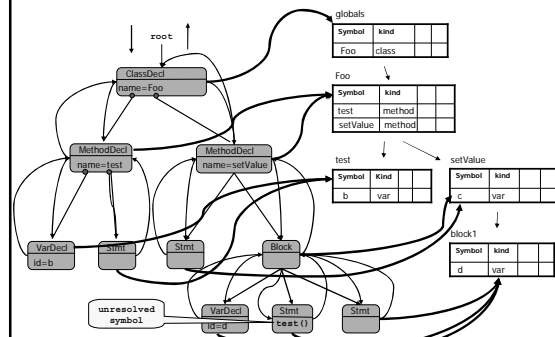
29

Forward references

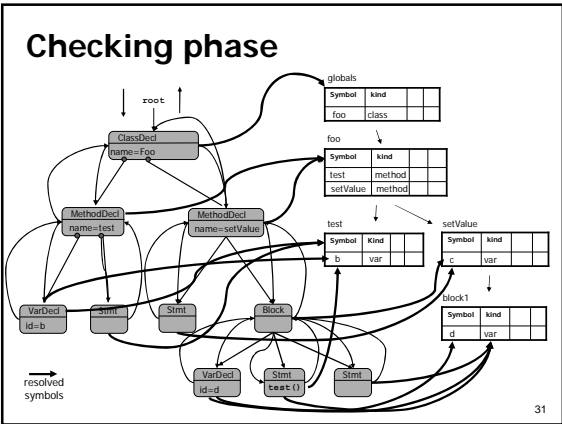


27

Building phase



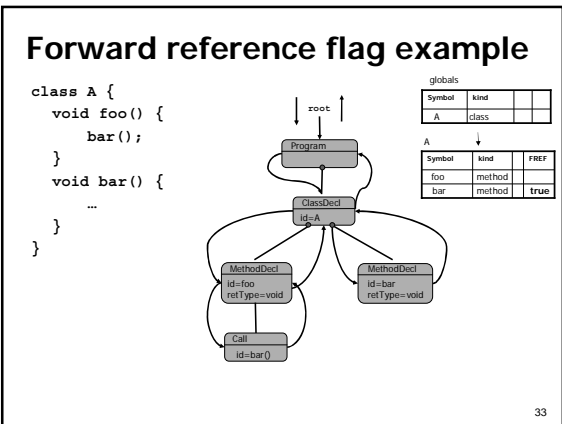
30



- ### Class hierarchy
- **extends** relation should be acyclic
 - Avoid creating cyclic symbol table hierarchy (infinite looping)
 - Can check acyclicity in separate phase (**ClassHierarchyVisitor**)
 - Build symbol tables for classes first and check absence of cycles
 - In IC this is even simpler – each class has to be defined before used in **extends**
- 34

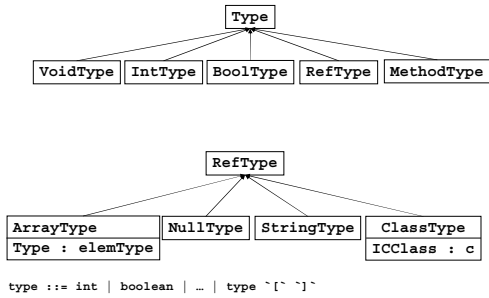
- ### Forward references – solution 2
- Use forward reference marker (flag)
 - Optimistically assume that symbol will be eventually defined
 - Update symbol table when symbol defined
 - Remove forward-reference marker
 - Count unresolved symbols and upon exit check that #unresolved=0
 - And/or construct some of the symbol table during parsing
- 32

- ### Next phase: type checking
- First, record all pre-defined types (**string, int, boolean, void, null**)
 - Second, record all user-defined types (classes, methods, arrays)
 - Recording done by storing in type table
 - Now, run type-checking algorithm
- 35



- ### Type table
- Keeps a single copy for each type
 - Can compare types for equality by ==
 - Records primitive types: int, bool, string, void, null
 - Initialize table with primitive types
 - User-defined types: arrays, methods, classes
 - Used to record inheritance relation
 - Types should support **subtypeOf(Type t1, Type t2)**
 - For IC enough to keep one global table
 - Static field of some class (e.g., **Type**)
 - In C/Java associate type table with scope
- 36

Possible type hierarchy



37

See you next week

38