

Compiler Construction

Semantic Analysis I

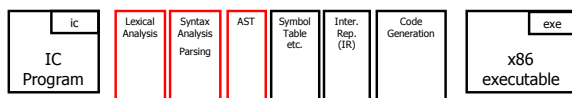
Ran Shaham and Ohad Shacham
School of Computer Science
Tel-Aviv University

TA1

- Parsing
- Submission is not in groups
- Deadline January 5th 2009
- Submission either in class or to box פז גרימברג

2

IC compiler Compiler

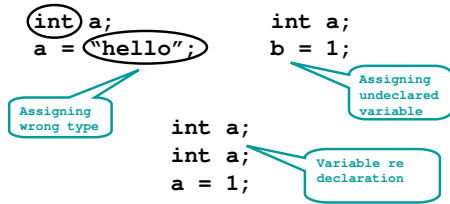


- Scopes
- Symbol tables
- Type checking

3

Semantic analysis motivation

Syntax analysis is not enough



4

Goals of semantic analysis

- Check "correct" use of programming constructs
- Context-sensitive
 - Beyond context free grammars
 - Deeper analysis than lexical and syntax analysis
- Semantic rules for checking correctness
 - Scope rules
 - Type-checking rules
 - Specific rules
- Guarantee partial correctness only
 - Runtime checks
 - pointer dereferencing
 - array access

5

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
- ...

6

Example of semantic rules

- Type rules are important class of semantic rules
 - In an assignment RHS and LHS must have the same type
 - In a condition test expression must have boolean type

7

Scope

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

8

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();  
  }  
}
```

scope of local variable b

scope of formal parameter c

scope of local variable d in statement block d

scope of field value

scope of method test

scope of c

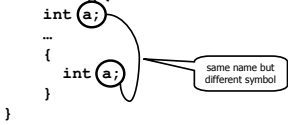
scope of value

9

Scope nesting

- Scopes may be enclosed in other scopes

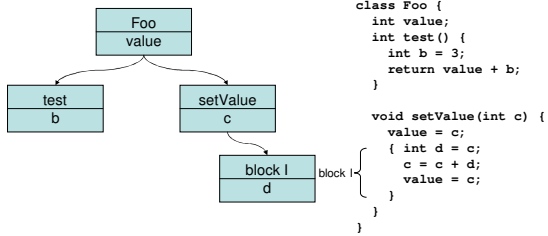
```
void foo() {  
  int a;  
  ...  
  {  
    int a;  
  }  
}
```



10

Scope tree

- Generally scope hierarchy forms a tree

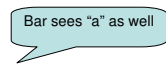


11

Subclasses

- Scope of subclass enclosed in scope of its superclass
- Subtype relation must be acyclic

```
Class Foo {  
  int a;  
}  
Class Bar extends Foo {  
}
```



12

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
 - Variables defined in block
- Code block scope
 - Variables defined in block

13

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 declared in an 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 if 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)

14

Symbol table

- An environment that stores information about identifiers
- A data structure that captures scope information

Symbol	Kind	Type	Properties
value	field	int	...
test	method	-> int	private
setValue	method	int -> void	public

15

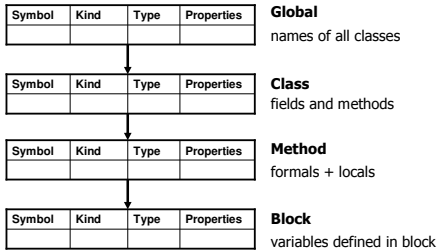
Symbol table

- Each entry in symbol table contains
 - name of an identifier
 - kind (variable/method/field...)
 - Type (int, string, myClass...)
 - Additional properties, e.g., final, public (not needed for IC)
- One symbol table for each scope

16

Scope nesting in IC

Scope nesting mirrored in hierarchy of symbol tables



17

Symbol table 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 {
  int value;
  void setValue(int c) {
    value = c;
  }
}
    
```

scope of b

scope of d

scope of c

scope of value

scope of c

scope of value

18

Catching semantic errors

Error !
undefined symbol

```

void setValue(int c) {
    value = c;
    { int d = c;
      c = c + d;
      myValue = c;
    }
}

```

22

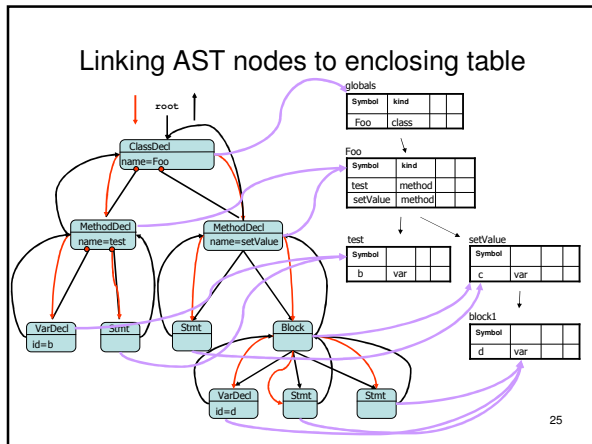
Symbol table operations

- 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?

23

Symbol table construction via AST traversal

24



What's in an AST node

```

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() {...}
}

```

26

Symbol table implementation

- Each table could be implemented using `java.util.HashMap`
- Implement a hierarchy of symbol tables
- Can implement a `Symbol` class
- `HashMap` keys should obey `equals/hashcode` contracts
- Safe when key is symbol name (`String`)

27

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;
    ...
}
```

28

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

29

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));
    }
    ...
}
```

30

Implementation option 2

```

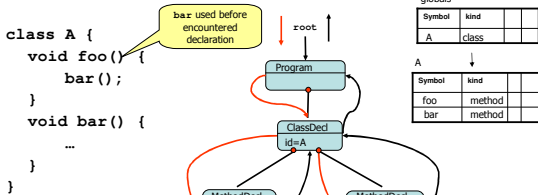
public class SymbolTable {
    /** Method Map id->entry **/
    private Map<String, Symbol> methodEntries;
    ...
    private Map<String, Symbol> variableEntries;
    ...
    private SymbolTable parent;
    public Symbol getMethod(String id, Type t) {
        return methodEntries.get(id);
    }
}

public void insertMethod(String id, Type t) {
    methodEntries.put(new Symbol(id, METHOD_KIND, t));
}
}

```

31

Forward references



globals

Symbol	kind		
A	class		

A ↓

Symbol	kind		
foo	method		
bar	method		

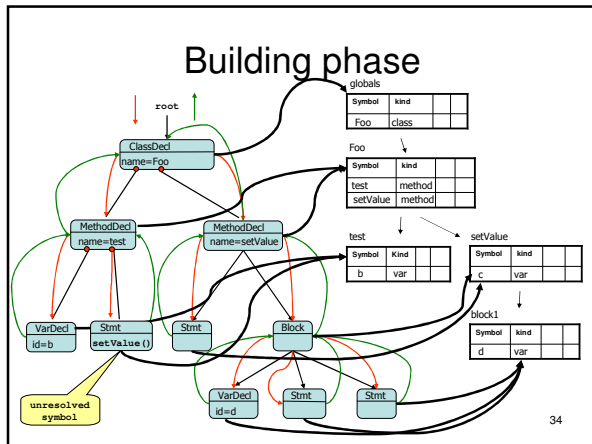
How do we handle forward references?
 We will see two solutions

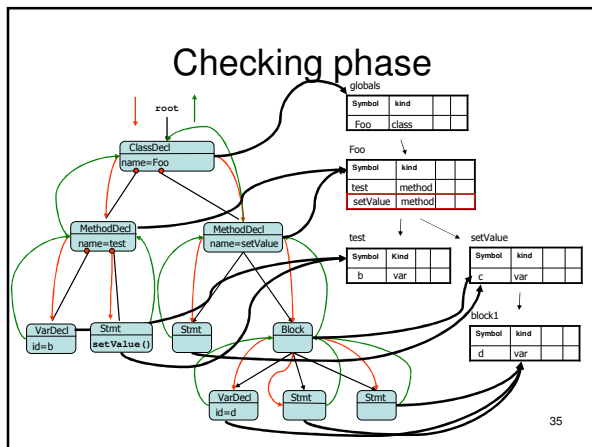
32

Solution 1 – multiple phases

- Building visitor
 - A propagating visitor
 - Propagates reference to the symbol table of the current scope
- Checking visitor
 - On visit to node
 - perform check using symbol tables
 - Resolve identifiers
 - Look for symbol in table hierarchy

33





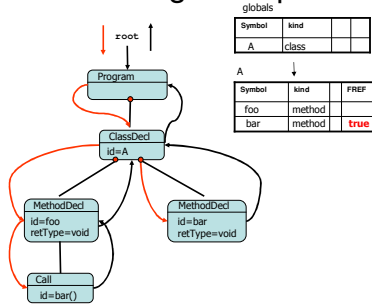
Forward references – solution 2

- Use forward reference marker
- Update symbol table when symbol defined
 - Remove forward-reference marker
- Count unresolved symbols
- Upon exit check that #unresolved=0

36

Forward reference flag example

```
class A {
  void foo() {
    bar();
  }
  void bar() {
    ...
  }
}
```



37

Class hierarchy

- **extends** relation should be acyclic
- Can check acyclicity in separate phase
- Build symbol tables for classes first and check absence of cycles

38

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

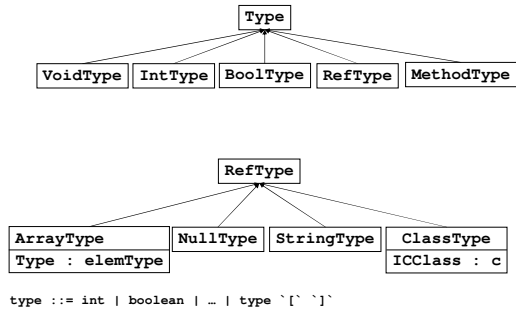
39

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

40

Possible type hierarchy



41
