

# Compiler Construction

## Semantic Analysis I

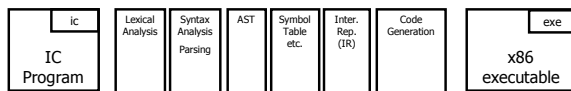
Rina Zviel-Girshin and Ohad Shacham  
School of Computer Science  
Tel-Aviv University

## TA1

- LR Parsing
- Submission is not in groups
- Submission either in class or to **box** פז גרימברג

2

## IC compiler Compiler

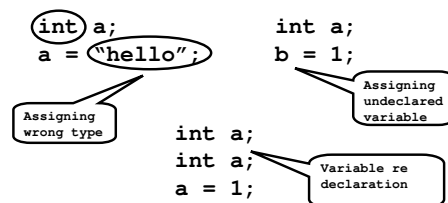


- Scopes
- Symbol tables
- Type checking

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();
  }
}
```

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` in `setValue`)
- scope of method test (points to `test()`)
- scope of c (points to `int c` in `Bar.setValue`)
- scope of value (points to `value` in `Bar.setValue`)

9

## Scope nesting

- Scopes may be enclosed in other scopes

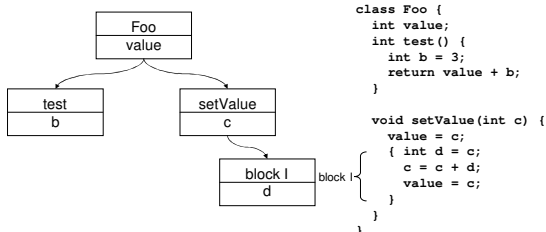
```
void foo() {
  int a;
  ...
  {
    int a;
  }
}
```

Diagram illustrating nested scopes with arrows and a callout box: "same name but different symbol" (points to the inner `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 {
}
```

Diagram illustrating inheritance with a callout box: "Bar sees 'a' as well" (points to `int a;` in `Class 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 method parameters can only be used after they are declared in 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  $C$ . 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

Symbol	Kind	Type	Properties	
				<b>Global</b> names of all classes
				<b>Class</b> fields and methods
				<b>Method</b> formals + locals
				<b>Block</b> variables defined in block

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 c

scope of value

scope of d

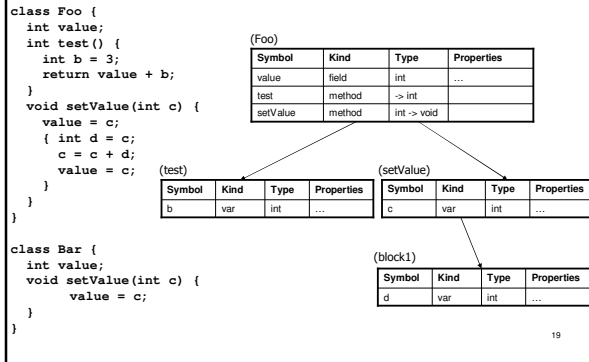
scope of c

scope of value

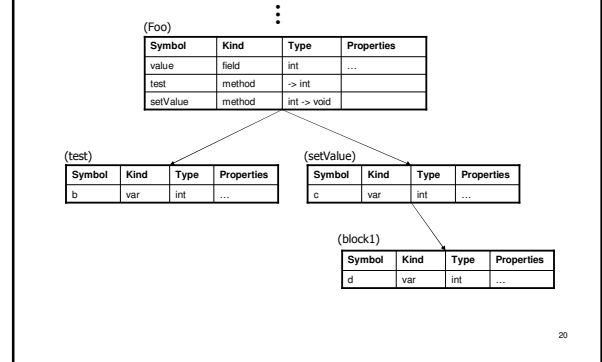
block1

18

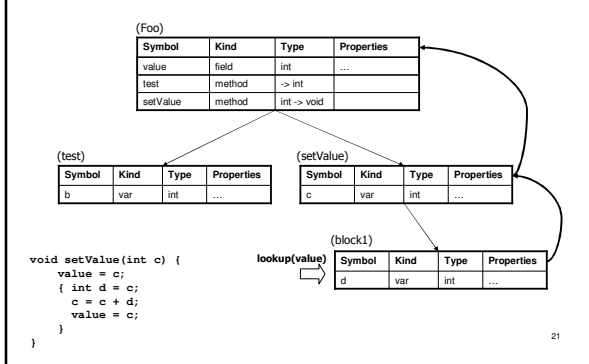
## Symbol table example



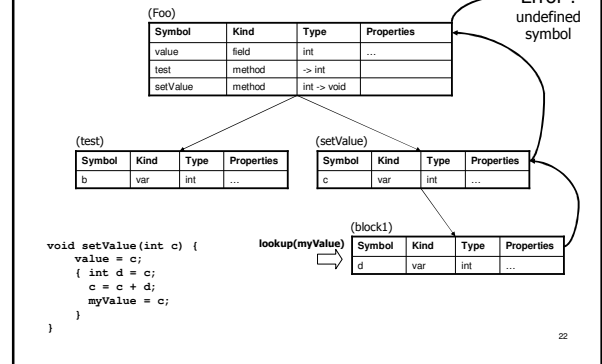
## Symbol table example cont.



## Checking scope rules



## Catching semantic errors

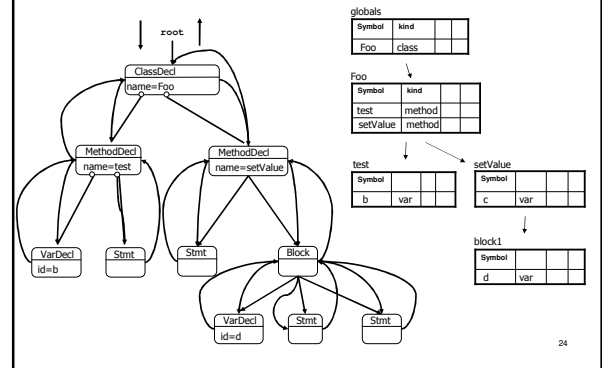


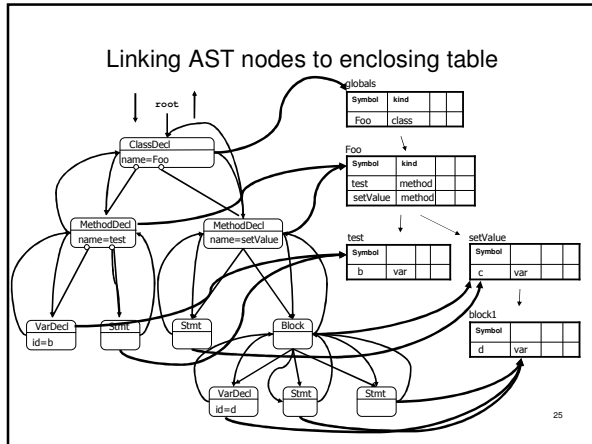
## Symbol table operations

- 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





### 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> 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

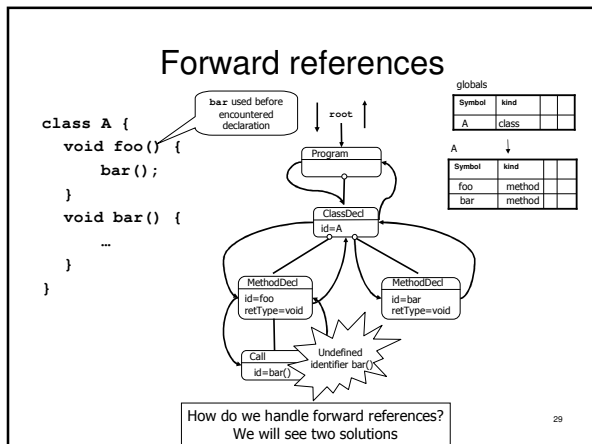
```

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>();
    }
    ...
}

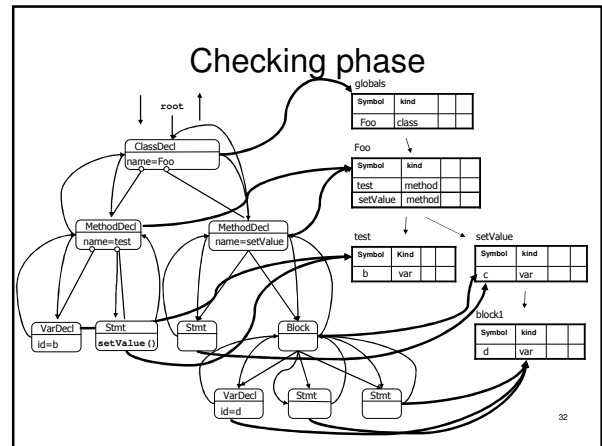
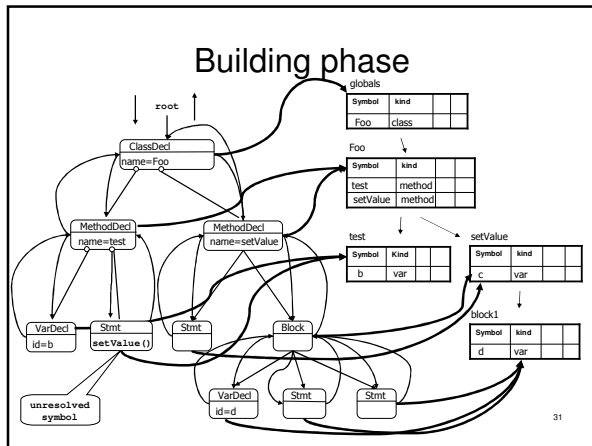
public class Symbol {
    private String id;
    private Type type;
    private Kind kind;
    ...
}

```

28



- ### 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
- 30



- ### 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

- ### 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

