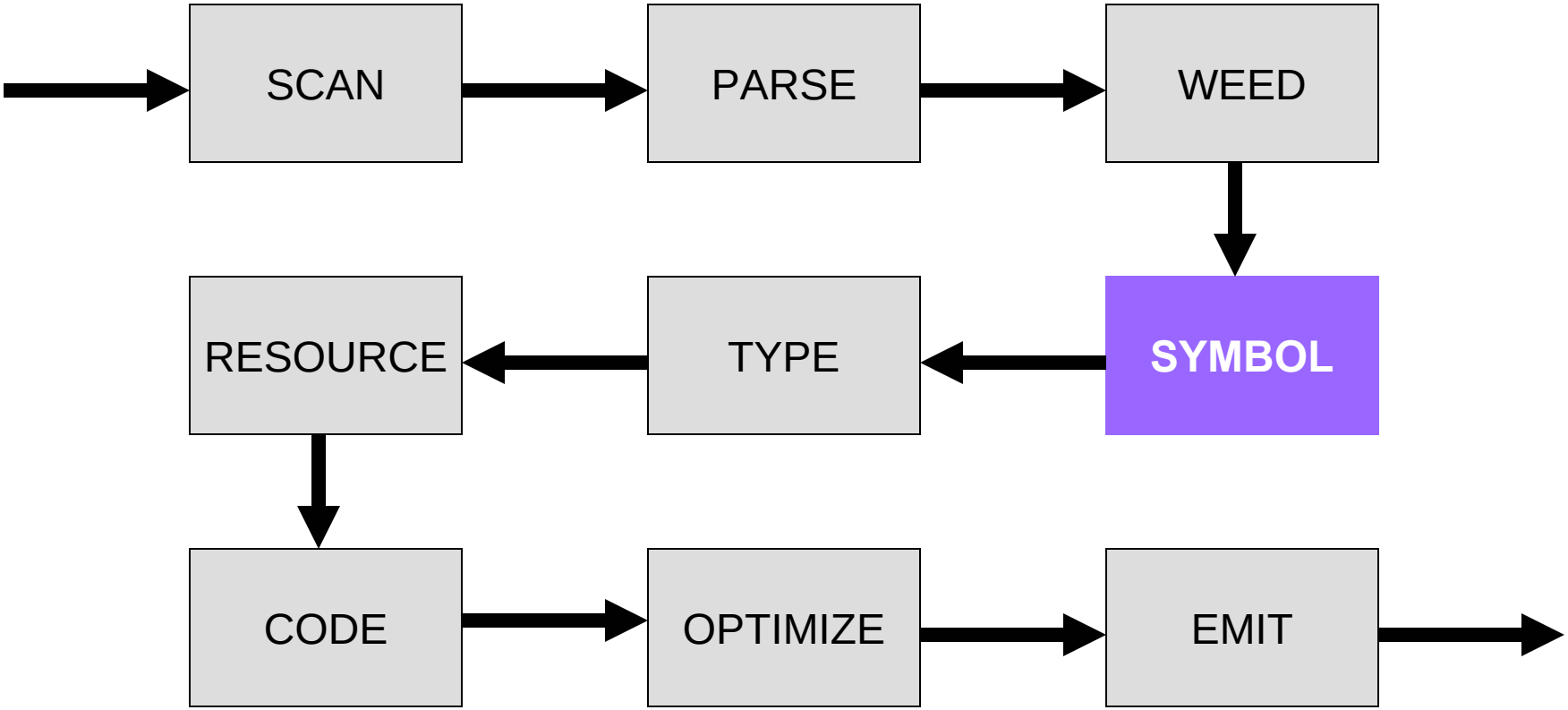
Compiler

Symbol Processing

Compiler Architecture



Analyzing Identifiers

- Lexical analysis defines *form* of identifiers
- Syntactic analysis defines *where* identifiers can appear
- Symbol analysis defines *correlation* of definition and uses of identifiers
- Grammars are too weak for this

$$\{w\alpha w | w \in \Sigma^*\}$$

is not context-free

Symbol Table

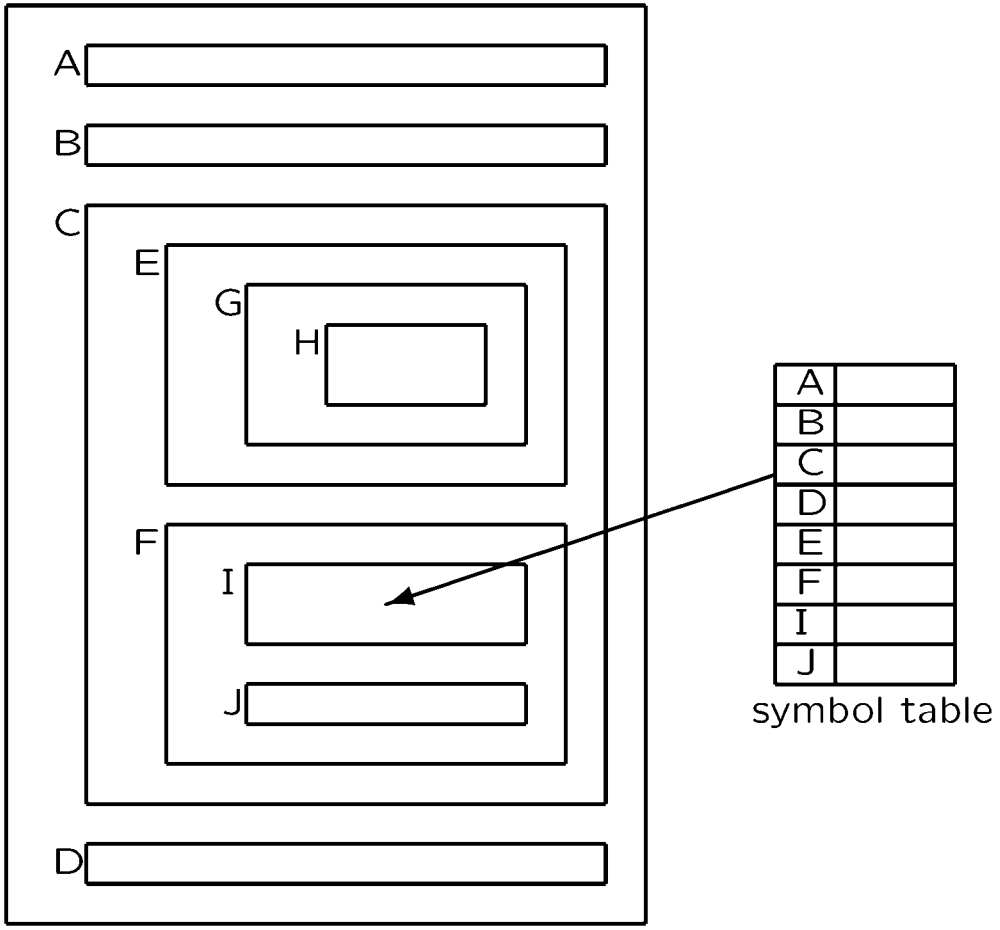
- Maps identifiers to their meaning (i.e., definition)

`i`	local	int
`done`	local	boolean
`insert`	method	ldots
`x`	formal	`List`
`List`	class	`...`
⋮	⋮	⋮

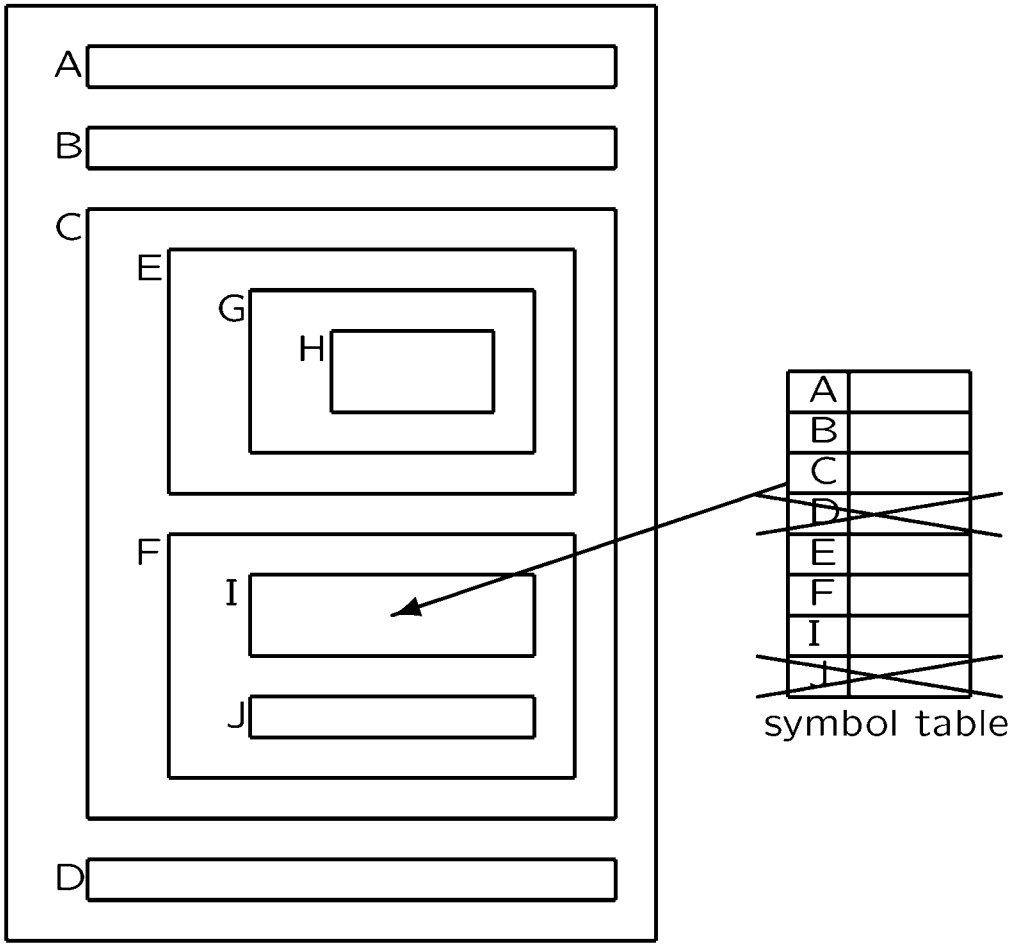
SJ Symbol Table Uses

- which classes are defined;
- which fields are defined;
- which methods are defined;
- what are the signatures of methods;
- are identifiers defined twice;
- are identifiers defined when used; and
- are identifiers used properly?

Static Nested Scope Rules

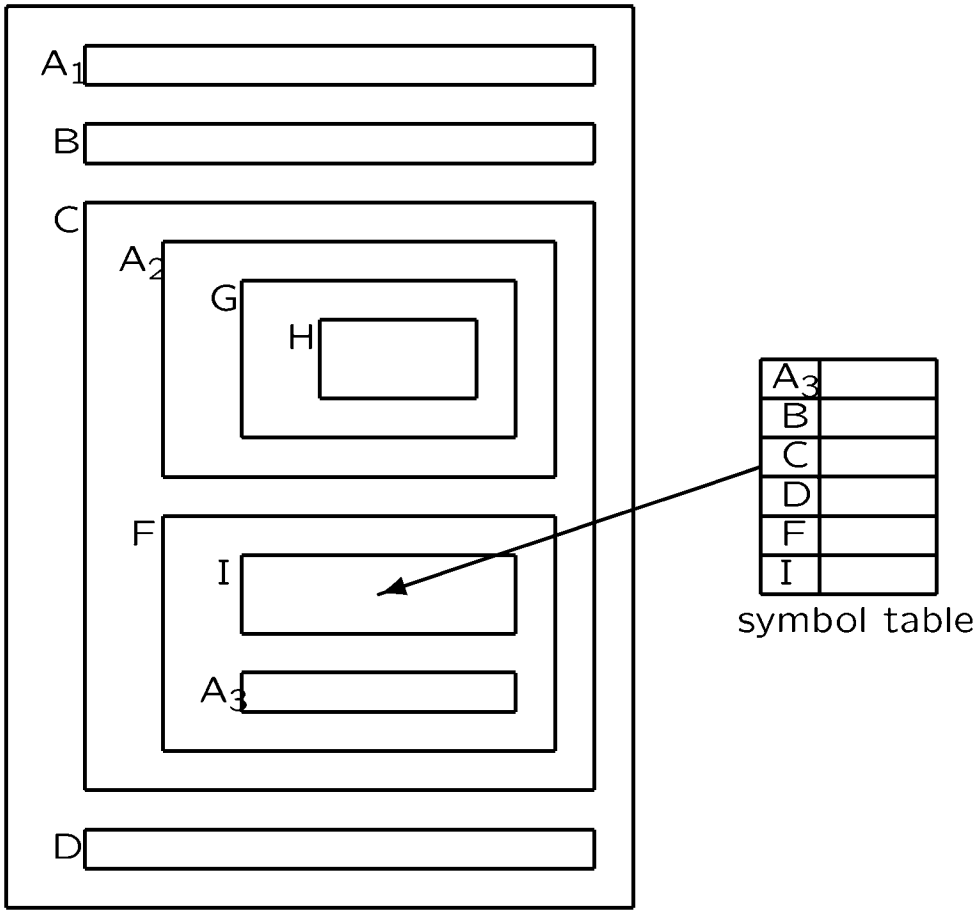


Nesting vs. Ordering



Multiple passes are required to eliminate the need for *forward declarations*

Most Closely Nested Definition

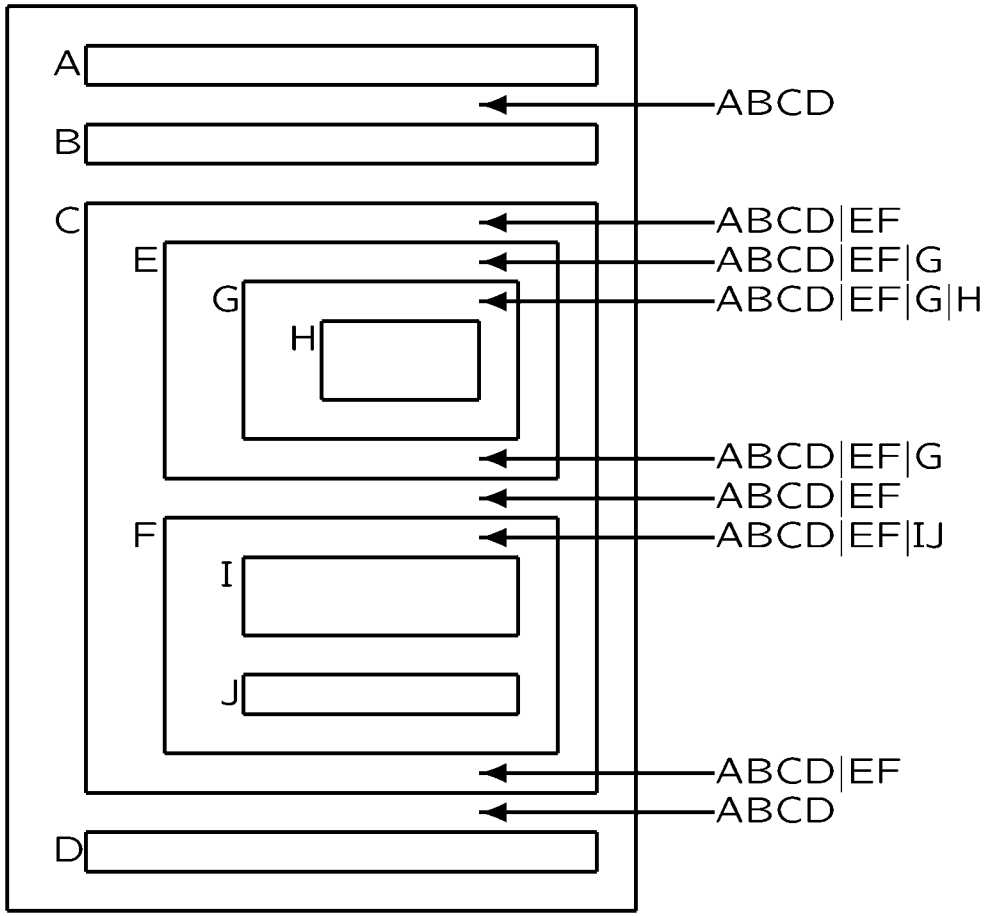


A2 shadows A1

A3 shadows A2, A1

Identifiers at same level must be unique

Symbol Table acts like a Stack



Implementation

Symbol table is *stack* of *hash tables*

- each hash table contains the identifiers in a level;
- push a new hash table when a level is entered;
- each identifier is entered in the top hash table;
- it is an error if it is already there;
- a use of an identifier is looked up in the hash tables from top to bottom;
- it is an error if it is not found;
- pop a hash table when a level is left.

SJ Symbol Table

- Strict rules on variable naming and variable declarations simplify symbol resolutions
  - variables can only be declared in the beginning of method bodies
  - parameter and local variable names are not allowed to shadow field names
- Other rules
  - method overloading is disallowed
  - allows only one class, etc.

SJ SymbolTable API

Overview Package **Class** Tree Deprecated Index Help

PREV CLASS NEXT CLASS | FRAMES NO FRAMES
SUMMARY: NESTED | FIELD | CONSTR | METHOD | DETAIL: FIELD | CONSTR | METHOD

sjc.symboltable

Class SymbolTable

```
java.lang.Object
  └ sjc.symboltable.SymbolTable
```

```
public class SymbolTable
extends java.lang.Object
```

This class represents a symbol table for a StaticJava `CompilationUnit`.

Author:
Robby

Field Summary	
`java.util.Map<org.eclipse.jdt.core.dom.ASTNode,java.lang.Object>`	`symbolMap` Holds the map of: a `SimpleName` expression, i.e., a reference to a field, a method parameter, or a local variable, to its corresponding `FieldDeclaration`, `SingleVariableDeclaration`, or `VariableDeclarationStatement`, respectively, and a `MethodInvocation` expression to its corresponding `MethodDeclaration` or `Method`.

Method Summary	
`java.lang.String`	`toString()` Returns the `String` representation of this symbol table.

Mutually Recursive Definitions

- A single traversal of the parse tree is not enough.
- Make two passes:
  - collect definitions of field and method identifiers; and
  - analyze uses of field and method identifiers.
- In cases like recursive types, the definition is not completed before the second traversal.

SymbolTableBuilder API

Overview Package **Class** Tree Deprecated Index Help

PREV CLASS NEXT CLASS | FRAMES NO FRAMES

SUMMARY: NESTED | FIELD | CONSTR | METHOD | DETAIL: FIELD | CONSTR | METHOD

sjc.symboltable

Class SymbolTableBuilder

```
java.lang.Object
  └ sjc.symboltable.SymbolTableBuilder
```

```
public class SymbolTableBuilder
extends java.lang.Object
```

This class is used to build symbol table for a StaticJava `CompilationUnit`. Note that the algorithm assumes that the JDT AST tree was built using the `ASTParser`. That is, it assumes certain structures on the AST, e.g., a class does not have an instance method.

Author:

Robby

Nested Class Summary	
static class	`SymbolTableBuilder.Error` This class is used to signal an error in the process of building a symbol table.

Method Summary	
static SymbolTable	`build(org.eclipse.jdt.core.dom.CompilationUnit cu)` Builds a `SymbolTable` for the given StaticJava `CompilationUnit`.

AST Visitor Code for SymbolTableBuilder

```
protected static class Visitor extends ASTVisitor {
  public @NonNull Map<ASTNode, Object> result =
                                              new HashMap<ASTNode, Object>();

  protected @NonNull Map<String, ASTNode> nameMap =
                                              new HashMap<String, ASTNode>();

  protected @NonNull Map<String, MethodDeclaration> methodMap =
                                       new HashMap<String, MethodDeclaration>();

  protected @NonNull Set<String> localNames = new HashSet<String>();

  protected String className;

  …

  protected void dispose() { … }
}
```

SymbolTableBuilder Code — TypeDeclaration (1)

```
@Override public boolean visit(TypeDeclaration node) {
  // remembers the class name
  className = node.getName().getIdentifier();

  if ("java.lang.String".equals(className)) {
    throw new Error(node, "Cannot redeclare the String class");
  }

  // visit field declarations and process method names first
  // because we want to be able to resolve field and method names
  // in the method bodies later on
  for (Object o : node.bodyDeclarations()) {
    if (o instanceof FieldDeclaration) {
      ((ASTNode) o).accept(this);
    } else if (o instanceof MethodDeclaration) {
      MethodDeclaration md = (MethodDeclaration) o;
      String methodName = md.getName().getIdentifier();
      if (methodMap.containsKey(methodName)) { // throw error }
      methodMap.put(methodName, md);
    } else { // throw error }
  }
```

SymbolTableBuilder Code — TypeDeclaration (2)

```
  // visit method declarations
  for (Object o : node.bodyDeclarations()) {
    if (o instanceof MethodDeclaration) {
      ((ASTNode) o).accept(this);
    }
  }
  return false;
}
```

SymbolTableBuilder Code — FieldDeclaration

```
@Override public boolean visit(FieldDeclaration node) {
  VariableDeclarationFragment vdf = (VariableDeclarationFragment) node
                                                    .fragments().get(0);
  String name = vdf.getName().getIdentifier();
  if (nameMap.containsKey(name)) {
    throw new Error(node,
                    "Error in field declaration '" + name
                            + "' : the field name has been used in:\n"
                            + nameMap.get(name));
  }
  nameMap.put(name, node);
  return false;
}
```

SymbolTableBuilder Code — MethodDeclaration

```
@Override public boolean visit(MethodDeclaration node) {
  for (Object o : node.parameters()) {
    SingleVariableDeclaration svd = (SingleVariableDeclaration) o;
    String name = svd.getName().getIdentifier();
    if (nameMap.containsKey(name)) { // throw error }

    localNames.add(name);
    nameMap.put(name, svd);
  }
  node.getBody().accept(this);

  for (String name: localNames) {
    nameMap.remove(name);
  }
  localNames.clear();
  return false;
}
```

SymbolTableBuilder Code — VariableDeclarationStatement

```
@Override public boolean visit(VariableDeclarationStatement node) {
  VariableDeclarationFragment vdf = (VariableDeclarationFragment) node
                                                        .fragments().get(0);
  String name = vdf.getName().getIdentifier();
  if (nameMap.containsKey(name)) {
    throw new Error(node,
                    "Error in local variable declaration '" + name
                           + "' : the variable name has been used in:\n"
                           + nameMap.get(name));
  }
  localNames.add(name);
  nameMap.put(name, node);
  return false;
}
```

SymbolTableBuilder Code — MethodInvocation

```
@Override public boolean visit(MethodInvocation node) {
  // Note that we don't visit the MethodInvocation's simple name
  // because we want visit(SimpleName) to resolve variable references
  // instead of method names.
  String methodName = node.getName().getIdentifier();

  if (node.getExpression() == null
      || className.equals(node.getExpression().toString())) {
    if (methodMap.containsKey(methodName)) {
      result.put(node, methodMap.get(methodName));
    } else { // throw error }
  } else {
    // lib call, delay until type checking phase
  }

  for (Object e : node.arguments()) {
    ((Expression) e).accept(this);
  }
  return false;
}
```

SymbolTableBuilder Code — SimpleName

```
@Override public boolean visit(SimpleName node) {
  String varName = node.getIdentifier();
  ASTNode parent = node.getParent();

  // Note that we have to make sure that at this point, the node
  // only corresponds to a variable reference
  if (parent instanceof Expression || parent instanceof Statement) {
    if (nameMap.containsKey(varName)) {
      result.put(node, nameMap.get(varName));
    } else {
      throw new Error(node, "Cannot resolve symbol '" + varName
                            + "' in:\n" + parent);
    }
  }
  return false;
}
```

Testing Strategy

- The testing strategy for the symbol tables involves an extension of the pretty printer.
- A textual representation of the symbol table is printed once for every scope area.
- These tables are then compared to a corresponding manual construction for a sufficient collection of programs.
- Furthermore, every error message should be provoked by some test program.