Contextual Analysis

- PA3 – Contextual Analysis assignment online
  - Due Thursday Mar 31
PA3 Contextual Analysis

• Implement contextual analysis in a subpackage
  – miniJava.ContextualAnalysis

• Contextual Analysis consists of
  1. Identification
  2. Type checking

• Also add null to miniJava!

• Sample PA3 project structure
  (within miniJava.ContextualAnalyzer
   choose classes and class names as you wish)

• Due Thu March 31
  – 21 days but 9 are spring break
  – Today’s lecture includes details on implementation
PA3 Implementation: Identification

• What needs to be done in Identification
  – Declarations need to be entered
    • ClassDecl, MemberDecl, LocalDecl
  
  – Identifiers need to be linked to their declaration
    • add field to Identifier class: public Declaration decl
    • work out a correct order to visit different parts of the AST to ensure all applicable declarations will have been seen before visiting an Identifier
    • link each identifier in the AST to its declaration using the appropriate idTable(s)

– What constructs need identification?
  • Basically all
    – Declarations
    – Statements, Expressions, References, TypeDenoters
      » anything that could contain an Identifier
Identification

- **IdTables**
  - enter(String s, Decl d)
    - associate s with Decl d
  - Decl retrieve(String s)
    - yields decl or null

- **Specific id tables**
  - is s a class name?
  - is s a member of class X?

- **Scoped id table**
  - enter or exit a scope
  - what declaration is associated with s in the current scope?
  - is s already declared in the current scope?
  - is s already declared in a scope with level ≥ 3?
  - enter a new <name,Decl> at the current scope level

---

<table>
<thead>
<tr>
<th>string</th>
<th>Decl</th>
<th>level</th>
</tr>
</thead>
<tbody>
<tr>
<td>class names</td>
<td>ClassDecl</td>
<td>1</td>
</tr>
<tr>
<td>member names</td>
<td>MemberDecl</td>
<td>2</td>
</tr>
<tr>
<td>parameter names</td>
<td>ParameterDecl</td>
<td>3</td>
</tr>
<tr>
<td>local var names</td>
<td>LocalDecl</td>
<td>4+</td>
</tr>
</tbody>
</table>
Identification in miniJava

- Parameters to the identification process
  - Class declarations
    - to identify uses of class names e.g.
      
      ```java
      Foo x = ...
      new Foo()
      ```
  - Member declarations in current class
    - to identify uses of fields or methods
  - Local declarations in current method
    - to identify uses of parameters or local variables
  - Member declarations in other classes
    - to identify qualified references, e.g.
      
      ```java
      Foo.field
      x.y.z
      ```
```java
class Foo {
    int x;

    int p(int y) {
        if (p(x)) {
            int x = 10;
            x = y;
        }
        return x;
    }
}
```
class Foo {
    int x;

    int p(int y) {
        if (p(x)) {
            int x = 10;
            x = y;
        }
        return x;
    }
}
Identification

• challenges
  – Access and Visibility restrictions of MemberDecls
    • Non-static members are not always accessible
    • private members are not always accessible
    • need a “context” for a reference to make a judgment

– Qualified references
  • example
    – x.y.z
  • what needs to be checked at each node of the Reference ast?
Implementation of Identification

- Use the Visitor pattern!

```java
package miniJava.ContextualAnalysis;

import miniJava.AbstractSyntaxTrees.*;
import miniJava.AbstractSyntaxTrees.Package;
import miniJava.ErrorReporter;

public class Identification implements Visitor<Object, Object> {
    public IdentificationTable table;
    private ErrorReporter reporter;

    public Identification(Package ast, ErrorReporter reporter) {
        this.reporter = reporter;
        table = new IdentificationTable(reporter);
        ast.visit(this, null);
    }
}
```
// Package
public Object visitPackage(Package prog, Object obj) {
    table.openScope();

    // add all the classes to the table.
    for(ClassDecl cd: prog.classDeclList) {
        table.enter(cd);
    }

    // then visit classes
    for(ClassDecl cd: prog.classDeclList) {
        cd.visit(this, null);
    }

    table.closeScope();
    return null;
}
// Declarations
public Object visitClassDecl(ClassDecl cd, Object obj) {
    currentClass = cd;

    // add members so all fields and methods are visible
    table.openScope();
    for(FieldDecl fd: cd.fieldDeclList) {
        table.enter(fd);
    }
    for(MethodDecl md: cd.methodDeclList) {
        table.enter(md);
    }

    // visit all members
    for(FieldDecl fd: cd.fieldDeclList)
        fd.visit(this, null);
    for(MethodDecl md: cd.methodDeclList) {
        md.visit(this, null);
    }
    table.closeScope();
    return null;
}
Identification – member declarations

```java
public Object visitFieldDecl(FieldDecl fd, Object obj) {
    fd.type.visit(this, null);
    return null;
}

public Object visitMethodDecl(MethodDecl md, Object obj) {
    md.type.visit(this, null);
    table.openScope();
    for(ParameterDecl pd: md.parameterDeclList) {
        pd.visit(this, null);
    }
    table.openScope();
    for(Statement st: md.statementList) {
        st.visit(this, null);
    }
    table.closeScope();
    table.closeScope();
    return null;
}
```
PA3 Type Checking

• Relatively simple
  – Bottom-up traversal of AST
  – Create a typeDenoter attribute in every Expression node (or possibly in every node)
    • The type rules for predefined functions are relatively simple
      +, -, *, etc : Int x Int → Int
      == : α x α → Boolean
      index : Array(α) x Int → α
      assign : α x α → Stmt
    • A single upwards pass suffices for miniJava type checking
  • Study type related classes in the AST
    – TypeDenoter, TypeKind, BaseType, ArrayType, Classtype
    – create an equality function between arbitrary instances of TypeDenoter

• run only if identification has completed successfully!
  – e.g. A x = new A();
Type Checking

• Special types
  – *Error* type
    • Error type is *equal* to any type
    • limits propagation of errors
    • gives most useful continuation of type checking after an error
  – *Unsupported* type
    • Unsupported type is *not equal* to any type
    • therefore a value of type unsupported is not type correct in any operation
    • predefined name String can have unsupported type
Logical order of Contextual analysis

1. Identification
   - check validity of declarations
     • is this declaration allowed in the current context?
   - link references to corresponding declarations
   - AST traversal order
     • top down, declarations before references

2. Type checking
   - assign types to AST nodes
     • start from leaves using decls installed in identification to determine parent nodes
   - check type agreement
     • operators and operands
     • assignment statements
   - AST traversal order
     • bottom up (assuming no overloading)
Contextual analysis in a single traversal

- For each node
  - *inherit* some information from parent
    - e.g. Identification table
  - traverse subtree rooted at node
  - *synthesize* some information to return to parent
    - e.g. type of expression computed by node
    - e.g. updated identification table

- Traversing the subtree rooted at a node
  - for each child in turn
    - apply contextual analysis on child
    - providing inherited data
    - receiving synthesized data