

# CS444/644 Assignment 1 Technical Report

Ho-Yi Fung, Jianchu Li, Zhiyuan Lin

March 16, 2016

**Abstract**

## Contents

<b>1</b>	<b>Introduction</b>	<b>2</b>
<b>2</b>	<b>Design</b>	<b>2</b>
2.1	Name resolution . . . . .	2
2.2	Static Analysis . . . . .	4
<b>3</b>	<b>implementation</b>	<b>4</b>
3.1	Name Resolution . . . . .	4
<b>4</b>	<b>Testing</b>	<b>6</b>
<b>5</b>	<b>Conclusion</b>	<b>7</b>

# 1 Introduction

## 2 Design

### 2.1 Name resolution

In this stage of the program, it was decided we would follow closely the 7 steps specified in class:

1. Constructing a global environment
2. Resolving uses of syntactically identifiable type names
3. Making sure class hierarchy is correct
4. Resolving ambiguous names
5. Resolving uses of local variables, formal parameters and static fields
6. Type checking
7. Resolving uses of methods and non-static fields.

In name resolution step, we connect names to their declarations.

The first step involves construction of a symbol table and resolution of syntactically identifiable names.

To construct the class environment, we would first need to build the global class environment. The global class environment contains information of all the types (classes and interfaces) defined, including those from the standard library. Once the global environment is in place, we would then construct the scopes within each class. This process needs to go over the declaration of all the types, methods, fields, and variables, and place these declarations in the proper scope.

The next step is to resolve the use of type names. This step deals with the following use of type names.

1. In a single-type-import
2. As super class of a type
3. As interface of a type
4. As a Type node in the AST, including:
  - (a) In a field declaration or variable declaration (including the declaration of formals)

- (b) As the result type of a method
- (c) As the class name used in class instance creation expression, or the element type in an array creation expression
- (d) As the type in a cast expression
- (e) In the right hand side of the instanceof expression

The above list is derived from the Java Language Specification, which defines syntactically identifiable type names in Section 6.5.1, and modified based on the Joos1W specification.

The resolution of type names would need to deal with both qualified names and simple names. As Java (and thus Joos) allows for use of fully qualified type name without import, and names cannot be partially qualified in Java, the resolution of qualified type name in Java is straightforward. One simply looks out the type from global environment. On the other hand, to resolve simple type names, we would need to look up the name, in the enclosing type declaration (the class or interface in which this type is used), in the single-type-import of the current compilation unit, in the package of the current class, and in the import-on-demand packages, exactly in this order. An error would be raised at this stage if no type under the name could be found. Besides if the name is resolved to more than one type (and thus is ambiguous), error would be thrown too. We also checks for prefix requirement in this process of resolving type name, that is for any qualified type name, its prefix should not be a valid type name in the class environment.

After type names are resolved, the class hierarchy needs to be built and checked for well-formedness. When building the class hierarchy, all classes that do not inherit from a super class implicitly inherit from the *java.lang.Object* class. Similarly every interface that does not have a super interface would inherit from an abstract version of the Object class. Checking the class hierarchy is straightforward. Besides the constraints given in class, it is also checked at this stage that no cycle exists in the class hierarchy.

The next step is to disambiguate ambiguous names. The Java Language Specification defines an *Ambiguous Name* to be the prefix of an *Expression Name*, a *Method Name*, or another *Ambiguous Name*, where an *Expression Name* could be the reference in an array access expression, a postfix expression, or the left-hand side of an assignment. The disambiguation process follows the simple steps in 1. Every use of local variable or a static field is resolved at this stage.

---

**Algorithm 1** Disambiguation of Field Name ( $A_1.A_2....A_n$ )

---

**Require:** Qualified Field Name ( $A_1.A_2....A_n$ )

- 1: If lookUp( $A_1$ ) is a local variable, then the rest are all non-static fields
  - 2: If lookUp( $A_1$ ) is a class field, then the rest are all non-static fields
  - 3: If lookUp( $A_1$ ) is a simple type, then  $A_2$  is a static field,  $A_3$  and onwards (if they exist) would be non-static fields.
  - 4: if lookUp( $A_1....A_k$ ) is a qualified type, then  $A_{k+1}$  is a static field, and every thing after that if exist would be non-static fields.
- 

The resolution of instance fields and methods are slightly more complicated, because it requires static type information of the prefix which would be provided by the type checking step. The type checking step associates every expression (including name) with a static type, and checks that the program is statically type correct. Being statically type correct means that the program satisfies all the type rules, which were provided in class and also available in JLS.

Once the type information is in place, we only need to look up the field in the field environment of its object's static type to resolve the name. For example, given instance field a.b where the object a has type A, we only need to look for b in the field environment of A. The process for resolving method name is similar. The only difference is that method overloading needs to be taken into consideration, therefore given a method invocation, the static types of its arguments are needed to find out which method is actually being called.

## 2.2 Static Analysis

The major part of our static analysis is the reachability analysis. In terms of reachability, Java requires that:

1. all statements are reachable (no dead code); and
2. all methods whose return type is not *void* must eventually return a value

These properties are non-trivial and thus undecidable by Rice's theorem. However our program provides a conservative approximation to the problem. This means that the reachability analysis implemented would not detect all occurrences of unreachable statements, but the analysis is always correct when it identifies statements as unreachable.

The reachability analysis follows in general the rules provides in Section 14.20 of JLS. The reachability rules define whether a statement is *reachable* and whether it could *complete normally*. A compile-time error would be raised if any statement is identified as unreachable according to the rules.

In Java, a local variable must be assigned before its value could be used. A definite assignment analysis is required to ensure this property. In Joos1W, this analysis is greatly simplified by requiring that all local variables are initialized when they are declared. Besides it is required that a variable cannot be used in its own initializer.

# 3 implementation

## 3.1 Name Resolution

It was discussed in the last report that, when constructing the abstract syntax tree, a *Visitor* interface was created so that future steps could follow the visitor design pattern. And the construction of symbol table is implemented following this pattern.

The *TopDeclVisitor* constructs the symbol table. This class extends the class *TraversalVisitor* which as its name suggests is a visitor that traverses the AST. This inheritance allows *TopDeclVisitor* to focus on the AST nodes that are relevant to its function while skipping the nodes that are useless to the process. Without the *TraversalVisitor* the *TopDeclVisitor* would have to implement every abstract visit method specified in the *Visitor* interface. The use of *TraversalVisitor* avoids repetition and results in clearer code, therefore some of the other visitors implemented also inherit from *TraversalVisitor*.

The *TopDeclVisitor* builds 4 types of scopes:

1. the *CompilationUnit* scope, which contains information of the imports; and
2. the *Inherit* scope, which contains fields and methods inherited by this class; and
3. the *Type* scope, which contains fields and methods declared in the current type; and
4. the *Block* scope, which is used within and as the method body.

The *Environment* class provides an implementation of scopes. The functions for managing scopes are implemented in the *SymbolTable* class. The *Environment* class also provides methods for looking up names, which would be used in later steps.

The `TopDeclVisitor` traverses the AST, creates and fills the scopes when appropriate. To be more specific, the *CompilationUnit* scope is first created and then the *Inherit* and *Type* scope. It was decided that the *Inherit* scope would not be populated until later steps where we build the class hierarchy. The *Type* scope however is immediately populated with fields and methods declared in this type. Then every method body is visited, and a new *Block* is created whenever a new block is opened.

In Java, a local variable must be declared before it is used. To ensure this, we create a new scope whenever a local variable declaration is created.

Type Linking Visitor

TopDecl visitor

Hierarchy Class

## 4 Testing

The unit testing method was used to ensure the robustness of our program, especially during implementation of scanner and parser. The JUnit 4 framework provides excellent support for this method. Following the principle of unit testing, each small unit in our program was tested separately. For example, the scanning of each token type was tested individually using parameterized tests. Both positive and negative test cases, derived from the language specification, were used in this process. For example, each error that should be detected by the weeder has its own test cases and was verified.

Because of the continuous nature of our project, the program was also tested incrementally, e.g. after each module (scanner, parser, etc.) was completed it was tested against all test cases that had been created even for previous modules. This approach ensures that any valid input could be run on all modules, while invalid input would fail at some point.

We've also created test case based on the example code from Joos 1W language specification [1]. These tests ensure that the program satisfies the requirements for the language.

Parse trees created by the parser was checked visually to ensure that it follows the grammar design. The weeder also provides confidence that the parse tree was constructed correctly. The effectiveness and correctness of the AST module was partly based on the correctness of parse tree due to their similarities. The AST will also be tested more extensively in later stages where it would be used as an input.

A JUnit test suite was also created so that after each addition/modification all modules could be tested again conveniently to prevent bugs being introduced to the program. After all parsing module of the program was completed, the tests cases from Marmoset was introduced and tested against. Parameterized testing was again used at this step to make sure the program passes every test case.

The program so far successfully passes all the test case devised by the team. This demonstrates the robustness and correctness of our implementation.

## 5 Conclusion

## Bibliography

- [1] “The Joos language.” <https://www.student.cs.uwaterloo.ca/~cs444/joos.html>. Accessed: 2016-02-03.