CS444/644 Assignment 1 Technical Report

Ho-Yi Fung, Jianchu Li, Zhiyuan Lin

March 16, 2016

Abstract

Contents

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.

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 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 examined.

3 Implementation

This chapter discusses the actual implementation of our program and the issues encountered during the implementation. This project was implemented in Java 8. *Git* was used for version control and the code repository was hosted on University of Waterloo server with *Gitlab*.

Traversal Visitor

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.