Universidade Federal do Pampa

Introduction to Variability Mechanisms in Java

Marcelo Schmitt Laser

March 31, 2017

Introduction

In order to properly implement the design decisions made for the PLeTs v4 project, it is necessary that the programmers involved be aware of certain mechanisms, both theoretical and practical, that will be used in its development. This document will serve both as a basis for the study and explanation of these techniques and as a guide for the appropriate implementation of them in PLeTs v4.

Chapter 1 will present the notion of using compilation directives for conditional compilation. First a basic theory of the concept is presented. Afterwards, the specific mechanism of using Static Final variables in Java is presented, along with its advantages and limitations.

Contents

Chapter 1

Compilation Directives

Compilation directives are well-known in C++, often identified as "#ifdef". While C# made a similar mechanism available to us (#if), Java is lacking in an in-built preprocessor, and therefore has no mechanism with the exact same purpose as the C++ compilation directives. This chapter presents a short introduction to compilers and the concept of compilation directives and preprocessors, as well as the reasons why these are used in PLeTs and how to use Static Final variables in Java to reach a similar result.

1.1 Compilers

Most are unaware of the exact way through which compilers work, but a basic understanding of their mechanisms is necessary to fully comprehend the uses of compilation directives. Compilers can generally be divided in three or four smaller tools that, together, execute the entire compilation process. These are the Lexical Analyser, the Syntactic Analyser, the Semantic Analyser, and, optionally, the Code Generator. Other features can be included in compilers, such as code optimizers, but these are not covered here.

1.1.1 Lexical Analyser

The Lexical Analyser is the part of a compiler that is responsible for tokenizing the input text. What this means is that the text, generally code, that is sent as input to the compiler is fully read and divided into its basic units, such as "identifiers", "reserved words", etc. This is important for the compilation process as the following steps (Syntactic and Semantic analysis) are not entirely concerned with the exact text included in the input, but rather with its structure. This tokenization process is generally executed in a single pass, and outputs an ordered list of tokens.

We can use the typical "Hello World" Java program as an example of Lexical Analysis, based on the code presented in Listing 1, taken from [1].

```java
public class HelloWorld {
  public static void main(String[] args) {
      // Prints "Hello, World" to the terminal window.
      System.out.println("Hello, World");
     }
}
```

Listing 1: HelloWorld.java from [1]

Bibliography

[1] R. Sedgewick and K. Wayne. Helloworld.java, 2011.