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. Finally, the mechanisms of using Java annotations and of using an external preprocessor software are analysed.

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 [2].

```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 [2]

In analyzing this file, a lexical analyser may identify the tokens "ACCESS_MODIFIER" (public), "CLASS" (class), "IDENTIFIER" (HelloWorld), "LEFT_BRACKET" ({), "ACCESS_MODIFIER" (public), and so on. Notice that the exact contents of a token are irrelevant at this point in the analysis. This is because it is irrelevant to the correct syntax of the text. Whether the name of the class is "HelloWorld" or "ClassNameHere", the form of the code will remain the same. Also notice that blank spaces, such as the simple space (" "), tabs ("/t") or carriage returns ("/r") are generally not tokenized. This is because the blank space is usually used as a wildcard for the delimitation between two tokens, which means the number of blank spaces is not important to the tokenization, and also means that the simple fact of the tokenizer separating, say, "ACCESS_MODIFIER" and "CLASS" already acknowledges that there was indeed a space between them (the space is implicit in the tokenization).

The importance of understanding lexical analysis for the understanding of conditional compilation is that the conditional statements, such as #ifdef or #if, or the constant declaration #define, are in fact tokens to the compiler. Whether the token value expected by the compiler is #ifdef, #if, or something else entirely (say, #condition, for example) is not important: it is the meaning of these tokens that matters. For this reason, despite there being no specific preprocessor in Java, it is plausible to imagine a system in Java where a conditional compilation clause is created in the form of a token CONDITIONAL_CLAUSE or anything of the sort, where the value is whatever we choose for it.

Understanding lexical analysis on its own may appear confusing, as it is largely just a tool to be used by the syntactic analyser. For this reason, feel free to return to this section after you understand syntactic analysis, as it may make more sense to you then.

1.1.2 Syntactic Analyser

The syntactic analyser is charged with making sense of the tokens provided to it by the lexical analyser. What it does is check the order of these tokens against a set of rules, called a grammar, to see whether the input text makes sense or not. The importance of this is to guarantee that the input text is in the right format for the compiler, so that it be able to read the file properly. Think of

it as checking the structure of a sentence in natural language, and making sure that pronouns, nouns, verbs, adjectives, etc. are used correctly.

Following the same example from Listing 1, we can propose the following simplified grammar to compare the token list with:

$\langle class \rangle ::= \langle ACCESS\_MODIFIER \rangle \quad \langle CLASS \rangle \quad \langle IDENTIFIER \rangle$
$\langle LEFT\_BRACKET \rangle \langle class\_inside \rangle \langle RIGHT\_BRACKET \rangle$
$| \quad \langle class\_inside \rangle ::= \langle comments \rangle \langle declaration \rangle$
$| \quad \langle comments \rangle ::= \epsilon \mid ...$
$| \quad \langle declaration \rangle ::= \langle method \rangle \mid \langle attribute \rangle \mid \langle declaration \rangle \langle declaration \rangle \mid \epsilon$
$| \quad ...$

This grammar specifies that a class is comprised of an access modifier (public, private, protected, etc.), the keyword "class", an identifier (like HelloWorld or ClassNameHere), a left bracket ({), the contents of the class, and a right bracket (}). The contents of the class (class_inside), in turn, are formed of comments and declarations. Comments can either be empty (represented by ϵ) or made up of a group of rules specific for formatting comments (omitted here). Declarations can either be empty, a method or an attribute; in this special case, the sequence ¡declaration¿ ¡declaration¿ is used to demonstrate that a declaration may be formed of two declarations. Essentially, this means that there can be any number of declarations in a class.

What the syntactic analyser will do, then, is take the tokens given it by the tokenizer and check them against the grammar. It knows that the base rule is ¡class¿, so the first thing it looks for is a token ACCESS_MODIFIER. It goes from there until it reaches the rule ¡class_inside¿, which causes it to shift what it is looking for from the ¡class¿ rule to the ¡class_inside¿ rule.

In a more practical example, considering the problem of conditional compilation, a grammar might look like this:

$\langle conditional \rangle ::= \langle CONDITIONAL\_SYMBOL \rangle \quad \langle IDENTIFIER \rangle \quad \langle code \rangle$
$\langle END\_CONDITIONAL\_SYMBOL \rangle$
$| \quad \langle code \rangle ::= ...$

What this means is that the syntactic analyser will look for pieces of text that look like code that is placed inside a conditional block. Adapting the HelloWorld example, we could have:

The lexical analyser will have read "#if" as CONDITIONAL_SYMBOL, "PRINT" as IDENTIFIER and "#endif" as END_CONDITIONAL_SYMBOL, with everything in-between being analysed as normal code. The syntactic analyser will then be able to realize that the code in that section falls in a conditional compilation rule, and will set it aside for special treatment by the semantic analyser or code generator. In practice, we have established that our HelloWorld class will only print "Hello, World" on the screen if we define the constant PRINT.

```
#define boolean PRINT = true;

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

Listing 2: Adapted from HelloWorld.java from [2]

At this point, you should understand the basics of why we make tokens, and how they are interpreted by the syntactic analyser. What we have not yet covered is how the compiler takes this information and transforms it into code. That is the job of the semantic analyser.

1.1.3 Semantic Analyser

The semantic analyser is in charge of deciding what the input text means, after the syntactic analyser has assured it that the text is well-formed. The semantic analyser will essentially be a programming code itself, treating input text according to what it receives. For example, a semantic analyser for the previous example in Listing 2 would include a program, or a piece of code, stating that if PRINT is not defined, then the statement inside the conditional should not be compiled. Essentially, it says "if PRINT is false, ignore everything in this block".

This is where we meet our problem with the Java compiler. To be able to identify these kinds of semantic rules, C++ and C# possess what is called a preprocessor. What this preprocessor does is take in a number of constant definitions, like "int TWO = 2" or "boolean PRINT = true", and replace every instance of that constant in the code with the value of the constant. Essentially, after the preprocessor has done its work, Listing 2 would look like this:

As you can see, the constant definition is removed entirely from the program, since it has already served its purpose, and all of its appearances were replaced with its value. By doing this, the preprocessor allows the compiler to know that section of code should be compiled; if PRINT had been equal to false, then the compiler would know to ignore that section of code. What happens in Java is, without a preprocessor, we have no means by which to tell the compiler which parts of the code should or should not be compiled, because we cannot define constants to be read in that way. In fact, the preprocessor for C# already knows what #if and #endif mean, and if PRINT had been false, the preprocessor itself

```
1  public class HelloWorld {
2    public static void main(String[] args) {
3        #if true
4        // Prints "Hello, World" to the terminal window.
5        System.out.println("Hello, World");
6        #endif
7      }
8  }
```

Listing 3: Adapted from HelloWorld.java from [2]

would have removed that code. The compiler would never even know it existed.

This is where the Java preprocessor idea comes in: by creating a program that takes in the Java source code and transforms it, we are able to tell the compiler exactly what we want, and what we don't want.

1.2 Java Preprocessing with Static Final attributes

One way to get around the problem of Java not having a preprocessor is by abusing the optimization features that Java does have. In the standard Java compilers, any code that is unreachable is automatically removed by the compiler's optimizer, which means that we are able to remove certain parts of code by making sure they can never be executed.

To do this, one can use what is called a Static Final attribute. When a conditional loop evaluates itself with a single boolean attribute, and that attribute is always false, that loop will be removed by the optimizer due to being unreachable. Usually the compiler cannot know if the condition will be false or not, but since a Static Final attribute can never change its value, the optimizer will be able to tell. Listings 4 and 6 show an example of how this can be used.

```
1  public class Preprocessor
2  {
3   public static final boolean FULLACCESS = false;
4   ...
5  }
```

Listing 4: Preprocessor.java, encapsulating variability points

A Preprocessor class is declared in Listing 4 which contains all of the variability point constants for free access from the rest of the project's packages. The packages may then use these attributes in normal conditional loops to transform

```
1  import Preprocessor;
2
3  public class UmlAssociation extends UmlElement implements
   ↪  Serializable
4  {
5  ...
6    private UmlElement end1Element;
7  ...
8    public UmlElement getEnd1()
9    {
10     if(Preprocessor.FULLACCESS)
11     {
12       return this.end1Element;
13     }
14     else
15     {
16       return DeepCopy.copy(this.end1Element);
17     }
18   }
19 ...
20 }
```

Listing 5: Variability point being resolved by the Preprocessor.java attributes, taken from [1]

them into conditional compilation loops. In Listing 6, the method getEnd1 is conditionally compiled so that if the selected product has full access, a reference to end1Element is returned, whereas if the selected product does not have full access, only a copy is returned. This ensures that if the product does not have full access, the user will not be able to directly alter the end1Element variable, being forced to use other methods to do so.

This approach has one very important limitation: it can only be used where normal conditional loops may be used. Therefore, it is impossible to conditionally compile an entire method or class, as Java does not permit conditional loop statements to be outside method bodies. While it works in the example above, it does work in the example presented in Listing **??**.

In this example, an entire constructor should be conditionally compiled so that this particular overload is only available if the configured product has FULLACCESS. While it is possible to place the method's inner code inside a conditional loop, that would mean this overload would still exist, and a programmer using an IDE like NetBeans or Eclipse would be fooled into thinking this overload is available, when in fact it is not. The ideal way of solving this problem would be to remove the method entirely from the compiled .jar file, so

```
1  import Preprocessor;
2
3  public class UmlAssociation extends UmlElement implements
   ↪  Serializable
4  {
5  ...
6    //INCORRECT JAVA SYNTAX, DOES NOT COMPILE!!!!!
7    if(FULLACCESS)
8    {
9      public UmlAssociation()
10     {
11       BuildIt(null, null, null, null, null, null, null,
   ↪  null, null);
12     }
13   }
14 ...
15 }
```

Listing 6: Limitations of Static Final attributes, taken from [1]

that the programmer is not tricked into thinking the method exists. To be able to do this, we fall back into the idea of making our own Java preprocessor, with capabilities similar to the C++ and C# preprocessors.

1.3 Java Preprocessing with Annotations

Bibliography

[1] M. Schmitt Laser and L. Dumer. Autorest. https://github.com/autorest, 2017.

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