Chapter 8
Operator Overloading as a DSL Parsing Mechanism

Alberto Simões
Polytechnic Institute of Câvado and Ave, Portugal

Rui Miguel da Costa Meira
Polytechnic Institute of Câvado and Ave, Portugal

ABSTRACT

This chapter describes an approach for the implementation of embedded domain-specific languages by using operator overloads and the creation of abstract syntax trees in run-time. Using the host language parser, an AST is created stating the structure of the DSL expression that is later analyzed, simplified, and optimized before the evaluation step. For the illustration of this process, the chapter proposes a domain-specific language for a basic linear algebra system dealing with matrices algebra and its optimization.

INTRODUCTION

Domain Specific Languages (DSL) are, undoubtedly, a great approach for the acceleration of processes (Kosar et al, 2015). These processes can be at different level: either for a specific purpose, completely outside of the scope of the programming paradigm (e.g. a specific language for describing a taxonomy of a thesaurus) or to help in the development process (e.g. the well-known flex and bison languages, designed to help in the development of compilers). While the first require a syntax specifically designed for that purpose, as the end users are not necessarily programmers, the
second usually take a hybrid approach, where some details are described in a new syntax, but a lot of the syntax is from the target language.

Usually DSL are classified as proper languages, when they are developed as a standard new language, where parsing follows the traditional approach (lexical and syntactic analysis, abstract syntax tree creation, tree manipulation and code generation or evaluation) or as an embedded language.

For this second situation, some DSL implement code generation, creating code in the target language that will be compiled and integrated with other host language files, or allowing its evaluation on run time (mainly for interpreted or other languages with reflection or meta-programming support (Bracha & Ungar, 2015)).

As parsing mechanism, this second case uses language constructs to define a dialect of the host language or, in some other situations, a hybrid parsing approach where some high order function transforms parts of the DSL syntax in the host language syntax (see (Simões & Almeida, 2010) for such a DSL implementation).

In this chapter we present another way for the development of embedded DSL through the use of operator overloading. While operator overloading is a common functionality on recent object-oriented languages, like C++, Java, C#, Python or Perl, the way these operators are used is, in most situations, the simple replacement of the default operator behavior (for example, the sum of two numeric values) with a similar one (for example, the sum of two vectors).

There are other situations where these operator overloads can create an abstract syntax tree (AST) instead of trying to evaluate the operator semantic. Thus, this behavior would be very similar to what a traditional parser would do when analysing the language. The main different is that it will be done in run time.

Note that this is not a new approach, as the way some libraries work show that similar approaches are used. An example is the way TensorFlow (Abadi et al, 2016) is able to compute what they call a computation graph, and later infer this computation derivative automatically. Therefore, this chapter does not claims a new methodology, but rather the clear definition of the structure of such a DSL implementation. Another example is MXNet (Chen et al, 2015) library. While the authors refer the usage of embedding a DSL in a host language, no references are made. Authors describe their systems results rather the way their implementation was done.

A few other references (Corliss & Griewank, 1993; Phipps & Pawlowski, 2012) were found, where the idea of operator overloading is discussed in order to enable differentiation and integration of expressions. But no details on DSL embedding are given.
Evaluating the Refactoring Index Using Entropy Approach
www.igi-global.com/chapter/evaluating-the-refactoring-index-using-entropy-approach/219982?camid=4v1a

White-Box Testing Automation With SonarQube: Continuous Integration, Code Review, Security, and Vendor Branches
www.igi-global.com/chapter/white-box-testing-automation-with-sonarqube/219976?camid=4v1a