Chapter 1
Extensible Languages: Blurring the Distinction between DSL and GPL

Didier Verna
EPITA Research and Development Laboratory, France

ABSTRACT

Out of a concern for focus and concision, domain-specific languages (DSLs) are usually very different from general purpose programming languages (GPLs), both at the syntactic and the semantic levels. One approach to DSL implementation is to write a full language infrastructure, including parser, interpreter, or even compiler. Another approach however, is to ground the DSL into an extensible GPL, giving you control over its own syntax and semantics. The DSL may then be designed merely as an extension to the original GPL, and its implementation may boil down to expressing only the differences with it. The task of DSL implementation is hence considerably eased. The purpose of this chapter is to provide a tour of the features that make a GPL extensible, and to demonstrate how, in this context, the distinction between DSL and GPL can blur, sometimes to the point of complete disappearance.

INTRODUCTION

Domain-specific language (DSL) design and implementation is inherently a transverse activity (Ghosh, 2010; Fowler, 2010). It usually requires from the product team knowledge and expertise in both the application domain and language design and implementation, two completely orthogonal areas of expertise. From the programming language perspective, one additional complication is that being an expert developer in one specific programming language does not make you an expert in lan-
language design and implementation - only in using one of them. DSLs, however, are most of the time completely different from the mainstream languages in which applications are written. A general-purpose programming language (GPL), suitable to write a large application, is generally not suited to domain specific modeling, precisely because it is too general. Using a GPL for domain-specific modeling would require too much expertise from the end-users and wouldn’t be expressive enough for the very specific domain the application is supposed to focus on.

As a consequence, it is often taken for granted that when a DSL is part of a larger application, it has to be completely different from the application’s GPL. But what if this assumption was wrong in the first place?

**Taxonomy of DSLs**

Following Fowler (2005) and Tratt (2008, section 2), a taxonomy of DSLs is provided in Figure 1.

**Standalone DSLs**

A *standalone*, or *external* DSL is written from scratch, as a completely autonomous language. In such a case, a whole new compiler / interpreter chain needs to be implemented, presumably with tools like Lex, Yacc, ANTLR etc. As noted by Vasudevan and Tratt (2011):

*Such an approach provides the DSL author with complete control over the DSL, from its syntax to its style of execution, but leads to high development costs as each implementation tends to be engineered from scratch.*

In addition to that, the use of several languages in parallel, within the same application, may be regarded as a disadvantage (Ghosh, 2011):

*Unless carefully controlled, this polyglot programming can lead to a language cacophony and result in bloated design.*

Indeed, every time a standalone DSL needs to be extended, the whole language production chain may need to be modified in ways that were not anticipated.

**Embedded DSLs**

A second approach to DSL implementation consists in reusing the capabilities (both syntactic and semantic) of a host GPL, hence avoiding the need for rewriting a complete infrastructure. This approach leads to so-called *embedded* or *internal* DSLs, and idea which is probably at least 50 years old (Landin, 1966). In order to implement an embedded DSL, one needs to express it in terms of the differences from the host GPL. To that aim, two approaches currently exist.