Chapter 34

Domain-Specific Languages

34.1 Language Design Variables

Programming languages differ in numerous ways:

1. Each uses rather different notations for writing down programs. As we’ve observed, however, syntax is only partially interesting. (This is, however, less true of languages that are trying to mirror the notation of a particular domain.)

2. Control constructs: for instance, early languages didn’t even support recursion, while most modern languages still don’t have continuations.

3. The kinds of data they support. Indeed, sophisticated languages like Scheme blur the distinction between control and data by making fragments of control into data values (such as first-class functions and continuations).

4. The means of organizing programs: do they have functions, modules, classes, ...?

5. Automation such as memory management, run-time safety checks, and so on.

Each of these items suggests natural questions to ask when you design your own languages in particular domains.

34.2 Languages as Abstractions

Languages are abstractions: ways of seeing or organizing the world according to certain patterns, so that a task becomes easier to carry out. More concretely, think about a loop in Java. When you write a loop, you expect the machine to carry out certain tasks for you automatically: testing the termination condition, running the loop code if the test passes, exiting the loop if it doesn’t, etc. The loop is an abstraction: a reusable pattern where the language executes part of the pattern automatically, and you supply the parts that are different. You could write down all of those steps manually, but then your program would be longer, harder to read, and more painful to write, debug and maintain. Scheme’s map and filter are also abstractions,
which differ from Java’s loops in one significant way: you can define Scheme’s loops as abstractions in user programs.

34.3 Domain-Specific Languages

Based on the above description, it becomes clear that some domains may be served better by programming in a language specialized to that domain. While we are familiar with such languages (often bundled with software packages that blur the boundary between the package and the language) such as Mathematica and Matlab, this principle is not new. Indeed, study the names of four of the oldest popular programming languages, and you spot a pattern:

**Fortran** Stands for “formula translator”.

**Algol** An “algorithmic language”.

**COBOL** An abbreviation for “COmmon Business-Oriented Language”.

**LISP** Short for a “list-processing” language.

Notice the heavy emphasis on very concrete domains (or, in the case of LISP, of a language construct)? Indeed, it was not until the late 1960s and 1970s that programming languages really became liberated from their domains, and the era of general-purpose languages (GPL) began. Now that we know so much about the principles of such languages (as we’ve been seeing all semester long), it is not surprising that language designers are shifting their sights back to particular domains.

Indeed, I maintain that designing GPLs has become such a specialized task—well, at least designing good GPLs, without making too many mistakes along the way—that most lay efforts are fraught with peril. In contrast, most people entering the programming workforce are going to find a need to build languages specific to the domains they find themselves working in, be they biology, finance or the visual arts. Indeed, I expect many of you will build one or more “little languages” in your careers.

Before you rush out to design a domain-specific language (DSL), however, you need to understand some principles that govern their design. Here is my attempt to describe them. These are somewhat abstract; they will become clearer as we study the example that follows in more detail.

First and foremost—define the domain! If your audience doesn’t understand what the domain is, or (this is subtly different) why programming for this domain is difficult, they’re not going to pay attention to your language.

Justify why your language should exist in terms of the current linguistic terrain. In particular, be sure to explain why your language is better than simply using the most expressive GPLs around. (Small improvements are insufficient, compared with the odds that the considerably greater resources that are probably going into language implementation, library support, documentation, tutorials and so on for that GPL compared with your language.) In short, be very clear on what your DSL will do that is very difficult in GPLs. These reasons usually take on one or more of the following forms:

- Notational convenience, usually by providing a syntax that is close to established norms in the domain but far removed from the syntax of GPLs. (But before you get too wrapped up in fancy visual notations,
keep in mind that programs are not only written but also edited; how good is your editor compared with vi or Emacs?)

- Much better performance because the DSL implementation knows something about the domain. For instance, some toolkits take limited kinds of programs but will, in return, automatically compute the derivative or integral of a function—a very useful activity in many kinds of high-performance scientific computing.

- A non-standard semantics: for instance, when neither eager nor lazy evaluation is appropriate.

There are generally two kinds of DSLs, which I refer to as “enveloping” and “embedded”. Enveloping languages are those that try to control other programs, treating them as components. Good examples are shell languages, and early uses of languages like Perl.

Enveloping languages work very well when used for simple tasks: imagine the complexity of spawning processes and chaining ports compared with writing a simple shell directive like `ls -l | sort | uniq`. However, they must provide enough abstraction capabilities to express a wide variety of controls, which in turn brings data structures through the back door (since a language with just functions but without, say, lists and queues, requires unreasonable encodings through the lambda calculus). Indeed, invariably programmers will want mapping and filtering constructs. The net result is that such languages often begin simple, but grow in an unwieldy way (responding to localized demands rather than proactively conducting global analysis).

One way to improve the power of an enveloping language without trying to grow it in an ad hoc way is to embed another language inside it. That is, the enveloping language provides basic functionality, but when you want something more powerful, you can escape to a more complete (or another domain-specific) language. For instance, the language of Makefiles has this property: the Makefile language has very limited power (mainly, the ability to determine whether files are up-to-date and, if not, run some set of commands), and purposely does not try to grow much richer (though some variants of `make` do try). Instead, the actual commands can be written in any language, typically Unix shell, so the `make` command only needs to know how to invoke the command language; it does not itself need to implement that language.

The other kinds of languages are embedded in an application, and expose part of the application’s functionality to a programmer who wants to customize it. A canonical example is Emacs Lisp: Emacs functions as a stand-alone application without it, but it exposes some (most) of its state through Emacs Lisp, so a programmer can customize the editor in impressive ways. Another example may be the command language of the `sendmail` utility, which lets a programmer describe rewriting rules and custom mail handlers.

Any time one language is embedded inside another language (as opposed to an application), there are some problems with this seemingly happy symbiosis:

1. The plainest, but often most vexing, is syntactic. Languages that have different syntaxes often don’t nest within one another very nicely (imagine embedding an infix language inside Scheme, or XML within Java). While the enveloping language may have been defined to have a simple syntax, the act of escaping into another language can significantly complicate parsing.

2. Can the embedded language access values from the language that encloses it? For example, if you embed an XML path language inside Java, can the embedded language access Java variables? And
even if it could, what would that mean if the languages treat the same kinds of values very differently? (For instance, if you embed an eager language inside a lazy one, what are the strictness points?)

3. Often, the DSL is able to make guarantees of performance only because it restricts its language in some significant way. (One interesting example we have seen is the simply-typed lambda calculus which, by imposing the restriction of annotations in its type language, is able to deliver unto us the promise of termination.) If the DSL embeds some other language, then the analysis may become impossible, because the analyzer doesn’t understand the embedded language. In particular, the guarantees may not longer hold!

In general, as a DSL developer, be sure to map out a growth route. Anticipate growth and have a concrete plan for how you will handle it. No DSL designer ever went wrong predicting that her programmers might someday want (say) closures, and many a designer did go wrong by being sure his programmers wouldn’t. Don’t fall for this same trap. At the very least, think about all the features you have seen in this course and have good reasons for rejecting them.

You should, of course, have thought at the very outset about the relationship between your DSL and GPLs. It doesn’t hurt for you to think about it again. Will your language grow into a GPL? And if so, would you be better off leveraging the GPL by just turning your language into a library? Some languages even come with convenient ways of creating little extension languages (as we will see shortly), which has the benefit that you can re-use all the effort already being poured into the GPL.

In short, the single most important concept to understand about your DSL is its negative space. Language designers, not surprisingly, invariably have a tendency to think mostly about what is there. But when you’re defining a DSL remember that perhaps the most important part of it is what isn’t there. Having a clear definition of your language’s negative space will help you with the design; indeed, it is virtually a prerequisite for the design process. It’s usually a lot easier to argue about what shouldn’t (and should) be in the negative space than to contemplate what goes in. And to someone studying your language for the first time, a clear definition of the negative space will greatly help understand your rationale for building it, and perhaps even how you built it—all of which is very helpful for deciding whether or not one finds this the right language for the task, both now and in the future.