Domain-Specific Languages: An Example

Kathi Fisler and Shriram Krishnamurthi

2003-11-17

1 The Design and Programming Perspectives

To start to understand what comprises a programming language, let's consider sample programs in four different languages (see Figure 1). What differences do you observe?

1. Syntax: each uses rather different notations for writing down programs. Quite true, but syntax is only so interesting; let's look for deeper differences.
2. What kinds of data they inherently support. Fortran supported various kinds of numbers, C includes arrays, while Scheme included lists (define-struct came in later), Java has classes.
3. Program organization techniques: The original Fortran didn't even have functions or basic blocks of code. Modern languages all provide functions, many now provide classes and objects, as well as larger organizational constructs like packages and modules.
4. What kinds of control constructs the programmer can use to express computations. Fortran provided only branches (conditionals) and goto-statements. Later languages provided loops, function calls, exceptions, continuations and other ways for programmers to control how a program executes.
5. What the language will do for the programmer. Languages like C require programmers to declare variables and free memory. Languages like Scheme and Java use garbage collection (programs that reclaim memory when a program is no longer using it). The different degrees of typechecking in modern languages (and there are lots of interesting differences) also fall into this category.

Even though these samples are from large, mainstream languages, the same differences define the boundaries of small, domain specific languages (such as those you implement to create a particular software system). Thus, our quick contrast of these languages suggests the questions you need to ask when you're about to define a new language:

1. What kind of data is this language designed to process?
2. What operations can someone perform on the data?
3. What control operators do I need to sequence operations?
4. What work is the language trying to save the programmer from doing?
5. What decisions should be postponed until run-time?

2 But What Is a Programming Language, Really?

More abstractly, 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. When you write down a while loop, you don't write down all of these steps. The loop is a common and useful pattern in programming that the designers of the language gave you in the form of a language construct. The loop is an abstraction: a reusable pattern where the language executes

FORTRAN (mid 1950's;1964)[a]

```
10 IF (X.GT.0.000001) GO TO 20
11 X = -X
   IF (X.LT.0.000001) GO TO 50
20 IF (X*Y.LT.0.000001) GO TO 30
   X = X*Y-Y
30 X = X+Y
   ...
50 CONTINUE
   X = A
   Y = B-A
   GO TO 11
   ...
```

[a]sample from slides for John Mitchell's PL course at Stanford

Scheme (1975)

```
(define-struct boa (name length eats))

(define (all-boa-foods ani-lst)
  (map boa-eats (filter boa? ani-lst)))
```

C (1978)[a]

```
#include <stdio.h>
#include <malloc.h>

/* Linked list structure */
struct c_linked_list
{
   char data[20]; /* String */
   struct c_linked_list *next;
};

/* Type def for linked list */
typedef struct c_linked_list link;

void deallocate_list(link *element) {
   link *current;

   /* Cycle through list */
   while (element != NULL)
   {
      /* Copy pointer */
      current = element;
      /* Get next pointer */
      element = (link *) element->next;
      /* Free old pointer */
      free (current);
   }
}
```

[a]Code adapted from http://www.stat.cmu.edu/ hseltman/c/Reilly.html

Java (mid 1990s)[a]

```
class Body {
  public long idNum;
  public String name=''<unnamed>'';
  public Body orbits = null ;
  private static long nextID = 0;

  Body() {
    idnum = nextID++;
  }
}

Body sun = new Body();
sun.idNum = Body.nextID++;
sun.name = ``Sol'';
sun.orbits = null ;
```

[a]Code from *The Java Programming Language*. Ken Arnold and James Gosling. Addison Wesley 1996

Figure 1: Sample code fragments from four well-known programming languages.

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 and debug.

Similarly, *map* and *filter* from Scheme are abstractions, patterns of programs that share a common structure: you write your code in terms of them and those constructs execute the common pattern. The main difference between them and loops in this context is that you can yourself define these functions in Scheme, while Java needs to build loops into the language because it doesn't give you good support for defining your own loops.

> *Languages are collections of abstractions: collections of common patterns that programmers can combine into working programs.*

When you implement a language, you implement those patterns (abstractions) that make up the language. You provide programmers with the tools they need to express the computations they want to perform. One task of a language designer is therefore to understand what programmers want to express (according to questions like those presented in Section 1), and then to implement the abstractions (using interpreters or compilers) corresponding to those patterns so someone can program in the new set of constructs.

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, says, 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.

```
;; Stage -1 : PowerPoint without a language

;; print-string : string → void
;; prints string and a newline to the screen
(define (print-string str)
  (printf "~a~n" str))

;; await-click : → void
;; mimics a mouse click by waiting for the user to type a character
(define (await-click) (read))

(begin
  (print-string "————————————————")
  (print-string "Hand Evals in DrScheme")
  (print-string "Hand evaluation helps you learn how Scheme reduces programs to values")
  (print-string "————————————————")
  (await-click)
  (print-string "————————————————")
  (print-string "Example 1")
  (print-string "(+ (* 2 3) 6)")
  (await-click)
  (print-string "(+ 6 6)")
  (await-click)
  (print-string "12")
  (print-string "————————————————")
  (await-click)
  (print-string "————————————————")
  (print-string "Summary: How to Hand Eval")
  (print-string "Find the innermost expression")
  (await-click)
  (print-string "Evaluate one step")
  (await-click)
  (print-string "Repeat until have a value")
  (print-string "————————————————")
  )
```

Figure 2: A naïve implementation of a slideshow.

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.

4 A Concrete Example: Implementing a Slideshow Package

Let's implement a slide-presentation system, similar in spirit to PowerPoint. What would a naïve implementation of the slideshow look like (using a simple, text-based interface)? Look at Figure 2: what's wrong with this implementation (aside from the unappealing text-based interface)?

- The programmer manually inserts the begin/end of slide lines each time.

- The programmer has to explicitly add the *await-click* commands, even though those are standard between slides.
- The programmer has to call *print-string* all the time, even though all the slide data are strings.

In short, there's little here that the programmer could reuse when writing another slideshow presentation.

This example is crying out for a language for writing slideshows, and for an interpreter to run those slideshows. To start defining the language, we need to answer the five questions from Section 1:

1. Slides are the data. Slides have titles and content. The content may be a bunch of text, or lists that are either bulleted or numbered. We may also have overlays, which are slides that sit on top of other slides, so that we can present content in stages.
2. The main operation on slides seems to be displaying them on the screen.
3. We sequence slides by putting them in some (linear) order.
4. The language should save the programmer from waiting for clicks to move between slides, from printing titles manually, and from manually specifying/tracking item numbers in itemized lists.
5. It's not clear at the moment what decisions should be postponed until run-time, so we'll come back to that one later.

Let's start by implementing a simple prototype of our slideshow system. A *prototype* is a working version of a scaled-down version of the system. The prototype lets us figure out how to implement the key concepts, and we'll build on those concepts later as we refine the language. For our first prototype, we'll implement basic sequences of slides (no overlays), and use a simple text display rather than a fancier graphics display.

4.1 Modeling the Data

We've already observed that our data consists of slides, so we need a data definition for slides:

```
;; a slide is a (make-slide string slide-body)
(define-struct slide (title body))

;; a slide-body is either
;; - a string (paragraph), or
;; - a (make-pointlist list[string] boolean)
(define-struct pointlist (points numbered?))

;; Examples
  (make-slide
    "Hand Evals in DrScheme"
    "Hand evaluation helps you learn how Scheme reduces programs to values")

  (make-slide
    "Example 1"
    (make-pointlist (list "(+ (* 2 3) 6)" "(+ 6 6)" "12") false))
```

4.2 Modeling the Operations

At this stage, we are not *implementing* the operations, we are instead trying to *represent programs as data*, so we can later write an implementation. We must develop a data definition for the set of operations in our language. Right now, we only have a display operation (though we might have more later). Let's write a data definition for operations (here called *commands*):

```
;; A cmd is
;; - (make-display slide)
(define-struct display (slide))
```

```
;; Example
  (make-display
    (make-slide
      "Hand Evals in DrScheme"
      "Hand evaluation helps you learn how Scheme reduces programs to values"))
```

This is just the abstract syntax for our PowerPoint interpreter.

4.3 Modeling Programs

In the slideshow example, a "program" is just a talk, which is itself a sequence of operations on slides. We need to model talks as data. The word "sequence" suggests lists. So, we'll view talks/programs as containing lists of operations:

```
;; A talk is a (make-talk list[cmd])
(define-struct talk (cmds))

;; A program is a talk
```

Now that we've modeled programs, operations, control, and data through data definitions, we are ready to write our first program (even though we can't run it yet):

```
(define talk1
  (let ([intro-slide
         (make-slide
          "Hand Evals in DrScheme"
          "Hand evaluation helps you learn how Scheme reduces programs to values")]
        [arith-eg-slide
         (make-slide
          "Example 1"
          (make-pointlist (list "(+ (* 2 3) 6)" "(+ 6 6)" "12") false))]
        [func-eg-slide
         (make-slide
          "Example 2"
          (make-pointlist (list "(define (foo x) (+ x 3))" "(* (foo 5) 4)" "(* (+ 5 3) 4)" "(* 8 4)"
                                "32") false))]
        [summary-slide
         (make-slide
          "Summary: How to Hand Eval"
          (make-pointlist (list "Find the innermost expression"
                                "Evaluate one step"
                                "Repeat until have a value")
                          true))])
    (make-talk
     (list (make-display intro-slide)
           (make-display arith-eg-slide)
           (make-display func-eg-slide)
           (make-display summary-slide)))))
```

(This code uses **let**, which is a bit more compact, instead of **local**, because we presume you've grown quite comfortable with Scheme by now.)

5 Implementing the Prototype

To implement the language, we need an interpreter for it:

```
;; run-talk : talk → void
```

;; executes the commands in a talk then displays end-of-show message

We know that a talk contains a list of commands, so we'll need a function that runs all the commands in a list:

```
;; run-talk : talk → void
;; executes the commands in a talk then displays end-of-show message
(define (run-talk a-talk)
  (begin
    (run-cmdlist (talk-cmds a-talk))
    (end-show)))

;; run-cmdlist : list[cmd] → void
;; executes every command in a list
(define (run-cmdlist cmd-lst)
  (cond [(empty? cmd-lst) void]
        [(cons? cmd-lst)
         (begin
           (run-cmd (first cmd-lst))
           (run-cmdlist (rest cmd-lst)))]))

;; run-cmd : cmd → void
;; executes the given command
(define (run-cmd cmd)
  (cond [(display? cmd) ...]))
```

Note we still haven't figured out how to implement the display command; we're just writing down templates and creating new functions to process all of the functions involved in the datatype for programs. Now that we're down to *run-cmd*, though, we need to figure out how to implement the display command.

What should the display command do? Print out the slide contents, then wait for a click from the user before proceeding. We can easily fill in the code with these operations:

```
;; run-cmd : cmd → void
;; executes the given command
(define (run-cmd cmd)
  (cond [(display? cmd)
         (begin (print-slide (display-slide cmd))
                (await-click))]))
```

Finally, we just need to implement *print-slide*. For that, we use the library of printing routines, and follow the data definition for slides so we know what we need to print;

```
;; print-slide-title : string → void
;; displays title of slide on screen
(define (print-slide-title title-str)
  (print-string (string-append "Title: " title-str))
  (print-newline))

;; print-slide-body : slide-body → void
;; displays contents of body on screen
(define (print-slide-body body)
  (cond [(string? body) (print-string body)]
        [(pointlist? body)
         (cond [(pointlist-numbered? body)
                (print-numbered-strings (pointlist-points body) 1)]
               [else (print-unnumbered-strings (pointlist-points body))])]))

;; print-slide : slide → void
```

```
;; displays contents of slide on screen
(define (print-slide aslide)
  (begin
    (print-string "————————————————————")
    (print-slide-title (slide-title aslide))
    (print-slide-body (slide-body aslide))
    (print-string "————————————————————")))
```

The *run-talk* program and all of its subprograms collectively form an *interpreter*: they interpret programs (talks) by executing them. Unlike our past interpreters, however, the output of this one is not a value in a datatype, but rather some behavior of pixels on a screen.