WORK-IN-PROGRESS: - this material is still under development

Implementing an Internal DSL

Last significant update: 03 Sep 07

Contents

Internal DSLs are often the most approachable form of DSLs to write. Unlike external DSLs you don't need to learn about grammars and language parsing, unlike language workbenches you don't need any special tools. With internal DSLs you work in your regular language environment. As a result it's no surprise that there's been a lot of interest around internal DSLs in the last couple of years.

With internal DSLs you are very much constrained by your host language. Since any expression you use must be a legal expression in your host language, a lot of thought in internal DSL usage is bound up in language features. A good bit of the recent impetus behind internal DSLs comes from the Ruby community, whose language has many features which encourage DSLs. However many Ruby techniques can be used in other languages too, if usually not as elegantly. And the doyen on internal DSL thinking is Lisp, one of the world's oldest computer languages, which has a limited but very appropriate set of features for the job.

Another term you might hear for an internal DSL is **fluent interface**. This was a term coined by Eric Evans and myself to describe more language-like APIs. It's a synonym for internal DSL looked at from the API direction. It gets to the heart of the difference between an API and a DSL - the language nature. As I've already indicated, there is a very gray area between the two. You can have reasonable but ill-defined arguments about whether a particular language construction is language-like or not. The advantage of such arguments is that they encourage reflection on the techniques you are using and how readable your DSL is, the disadvantage is that they can turn into continual rehashes of personal preferences.

[TBD: Discuss how to refer to identifiers: symbol classes, strings, language symbols.]

Fluent and command-query APIs

For many people the central pattern for a fluent interface is that of Method Chaining. A normal API might have code like this:

```
Processor p = new Processor(2, Processor.Type.i386);
Disk d1 = new Disk(150, Disk.UNKNOWN_SIZE, null);
Disk d2 = new Disk(75, 7200, Disk.Interface.SATA);
return new Computer(p, d1, d2);
```

With Method Chaining we can express the same thing with

```
computer()
  .processor()
    .cores(2)
    .i386()
  .disk()
    .size(150)
  .disk()
    .size(75)
    .speed(7200)
    .sata()
  .end();
```

Method Chaining has attracted the attention it has because it leads to a very different style of programming than the one that is usually taught. In particular it assumes long sequences of method calls, which in other circumstances are derided as "train wrecks". But Method Chaining isn't the only way we can use combine functions to create more fluent fragments of program code.

The most obvious route to go is just to use a Function Sequence

```
computer();
  processor();
    cores(2);
    processorType(i386);
  disk();
    diskSize(150);
  disk();
    diskSize(75);
    diskSpeed(7200);
    diskInterface(SATA);
```

As you can see, if you try to lay out and organize a Function Sequence in an appropriate way, it can read in as clear a way as Method Chaining. The point here is that fluency isn't as much about the style of syntax you use, so much as it is about the way you name and factor the methods themselves.

In the early days of objects one of the biggest influences on me, and many others, was Bertrand Meyers book "Object-Oriented Software Construction". One of the analogies he used to talk about objects was to treat them as machines. In this view an object was a black box with its interface as a series of displays to see the observable state of the object and buttons that you could press to change the object. This effectively offers a menu of different things you can do with the object. This style of interface is the dominant way we think about interacting with software components and is so dominant that we don't even think of giving it a name. So for the purposes of this discussion, I'll give it one - a command-query interface.

The essence of fluent interfaces is that they approach thinking of using components in a different way. Instead of a box of objects each sporting lots of buttons, we think linguisticly of composing sentances using clauses that weave these objects together. It's this mental shift that is the core difference between an internal DSL and just calling an API.

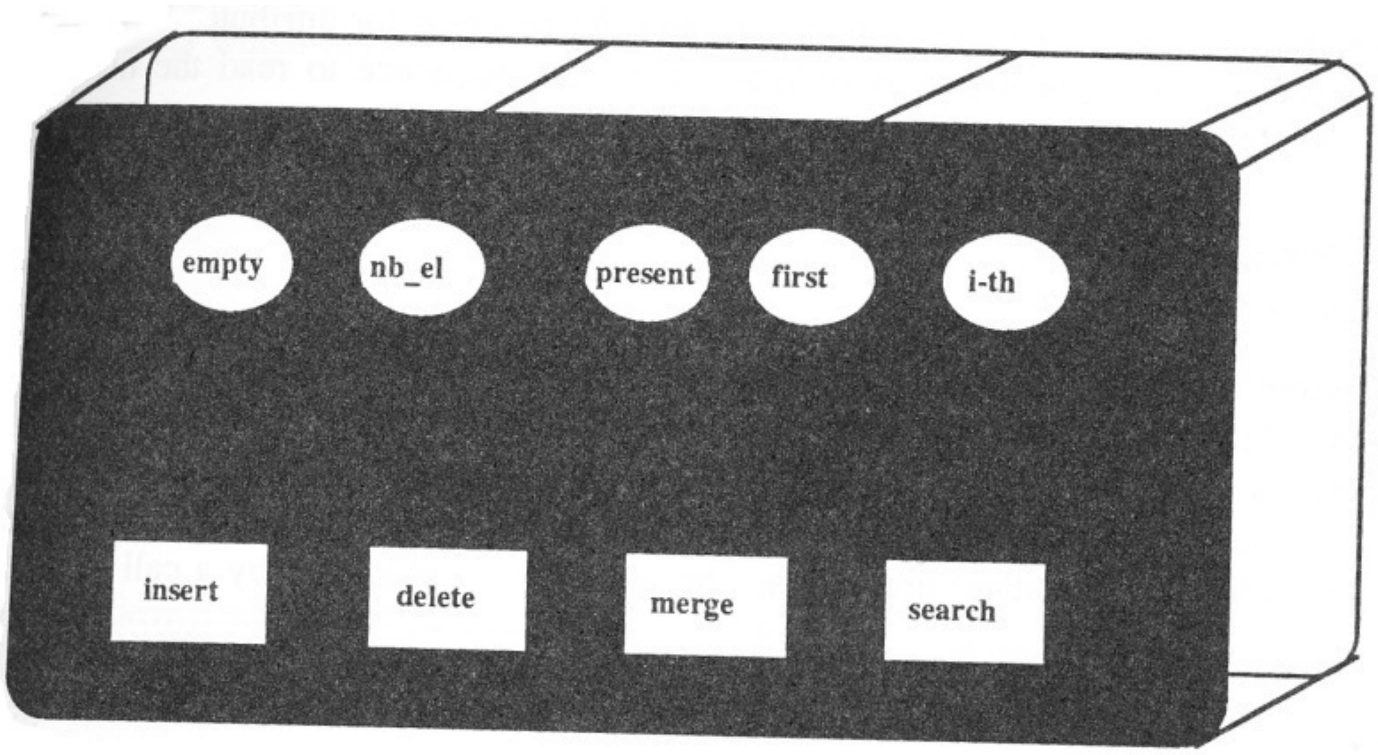

Figure 7.5 An object as machine

Figure 1: The original figure from OOSC that Bertrand Meyer used to illustrate the machine metaphor.

As I've mentioned earlier, it's a very fuzzy difference. Treating APIs as languages is also an old and well-regarded analogy that goes back before objects were the norm. There's plenty of cases that could be argued as command-query or fluent. But I do think that despite its fuzzyness it's a helpful distinction.

One of the consequences of the differences between the two styles of interface is that the rules about what makes a good interface are different. Meyer's original use of the machine metaphor is very apt here. The figure appeared in a section of OOSC that introduced the Command Query Separation. This principle is an extremely valuble one in programming and I strongly encourage teams to use it. One of the consequences of using Method Chaining in internal DSLs is that it often breaks this principle - you method alters state but returns an object to continue the chain. I have used many decibles disparaging people who don't follow command-query separation, and will do so again. But fluent interfaces follow a different set of rules, so I'm happy to allow it there.

Another important difference between a command-query and fluent interface is in the naming of the methods. When you're coming up with names for a command-query interface, you want the names to make sense in a stand-alone context. Often if people are looking for a method to do something, they'll run their eyes down the list of methods in web document page or in an IDE menu. As a result the names need to convey clearly what they do in that kind of context - they are the labels on the buttons.

Naming with fluent interfaces is quite difference. Here you concentrate less on the each individual element in the language, but instead on the overall sentances that you can form. As a result you can often have methods that make no sense in an open context, but read properly in the context of a DSL sentance. With DSL naming, it's the sentance that comes first, the elements are named to fit in with that context. DSL names are written with the context of the specific DSL in mind, while command-query names are written to work without any context (or in any context - the same thing here.)

The need for a parsing layer

The fact that a fluent interface is a different kind of interface to a command-query interface can lead to complications. If you mix both styles of interface on the same class, it's confusing. As a result I advocate keeping the lanauge handling elements of a DSL separate from regular command-query objects by building a layer of Expression Builders over regular objects.

The different nature of interfaces is one reason for Expression Builders, but the primary reason is a classic separation of concerns argument. As soon as you introduce some kind of language, even an internal one, you have to write code that understands that language. This code will often need to keep track of data that is only relevant while the language is being processed - parsing data. Understanding the way in which the internal DSL works is a reasonable amount of work, and isn't needed once you've populated the underlying model. You don't need to understand the DSL or how it works to understand how the underlying model operates, as a result it's worth keeping the language processing code in a separate layer.

This structure follows the general layout of DSL processing. The underlying model of command-query interface objects is the Semantic Model. The layer of Expression Builders is the parser. Although we are using the parser of the host language to turn text into statements of the host language, this isn't enough parsing to populate the Semantic Model - the Expression Builders add that extra element to complete the parsing process.

This separation also involves the usual advantages of a Semantic Model. You can test the Expression Builders and Semantic Model independently. You can have multiple parsers, mixing internal and external DSLs or supporting multiple internal DSLs with multiple Expression Builders.You can evolve the Expression Builders and Semantic Model independently. This is important as DSLs, like any other software, are hardly ever fixed. You need to be able to evolve the software, and often it's useful to change the underlying framework without changing the DSL scripts, or vice-versa.

Using Functions

Since the beginning of computing, programmers sought to package up common code into reusable chunks. The most successful packaging construct we've come up with is the function (also called subroutine, procedure, and method in OO-land). command-query APIs are usually expressed in functions, but DSL structures also are often built primarily on functions. The difference between a command-query interface and a DSL centers around how functions are combined.

There are a number of patterns in how you can combine functions to make a DSL. At the begining of this chapter I showed two. Let's recap, as I've forgotten what I wrote back there. First Method Chaining:

```
computer()
  .processor()
    .cores(2)
    .i386()
  .disk()
    .size(150)
  .disk()
    .size(75)
    .speed(7200)
    .sata()
  .end();
```

Then Function Sequence

```
computer();
  processor();
    cores(2);
    processorType(i386);
  disk();
    diskSize(150);
  disk();
    diskSize(75);
    diskSpeed(7200);
    diskInterface(SATA);
```

The are various factors to bear in mind when choosing between these two. The first issue is the scope of the functions. If you use Method Chaining the DSL methods need only be defined on the objects that take part in the chain, usually an Expression Builder. Using bare functions in a sequence means that you have to ensure the scope of the functions resolve properly. The most obvious way to do this is to use global functions, but using globals introduces two problems: complicating the global name-space and introducing global variables for parsing data.

Good programmers these days are nervous about any global stuff, as it makes it harder to localize changes. Global functions will be visible in every part of a program, not just the DSL processing, but you want the functions to only be available within the DSL processing. There are various language features that can remove the need to make everything global. A namespace capability can allow you to make functions look global only when you import a particular name-space. In the above examples I used the static import feature of java, which allows me to use static methods on classes as a name-space. Of course some languages don't allow this kind of global function access at all, including pre 1.5 java. In this case all you can do is use static methods explicitly, which will introduce extra noise to the DSL script.

The global parsing data is the more serious problem. Whichever way you do Function Sequence you'll need to manipulate Context Variables in order to know where you are in parsing the expression. Consider the calls to `diskSize`. The builder needs to know which disk's size is being specified, so it does that by keeping the track of the current disk - which it updates during the call to `disk`. Since all the functions are global, this state will end up being global too. There are things you can do to contain the globality - such as keeping all the data in a singleton - but you can't get away from global data if you use global functions.

Method Chaining avoids much of this because although you need some kind of bare function to begin the chain, once you've started all parsing data can be held on the Expression Builder object that the chaining methods are defined on.

You can avoid this globalness by using Object Scoping. In most cases this involves placing the DSL script in a subclass of an Expression Builder. With mainstream languages this allows bare function calls to resolve to instance method calls on the builder object, which handles both problems. All the functions in the DSL are defined only in the builder class, and thus localized. Furthermore since these are instance methods, they connect directly to data on the builder instance to store the parse data. That's a compelling set of advantages for the cost of placing the DSL script in a builder subclass, so that's my default option.

Both Function Sequence and Method Chaining require you to use Context Variables in order to keep track of the parse. Nested Function is a third function combination technique that can often avoid Context Variables. Using Nested Function the computer configuration example looks like this.

```
  computer(
    processor(
      cores(2),
      Processor.Type.i386
    ),
    disk(
      size(150)
    ),
    disk(
      size(75),
      speed(7200),
      Disk.Interface.SATA
    )
  );
```

Nested Functions have some powerful advantages with any kind of hierarchic structure, which is very common in parsing. One immediate advantage is that the hierarchic structure of the configuration is echoed by the language constructs themselves - the disk function is nested inside the computer function just as the resulting framework objects are nested. I don't have to rely on strange indentation conventions to suggest the overall structure - although I still format rather differently to how I'd format regular code.

The biggest difference however is the change in evaluation order. With a Nested Function the arguments to a function are evaluated before the function itself. This often allows you to build up framework objects without using a Context Variable. In this case, the `processor` function is evaluated and can return a complete processor object before the `computer` function is evaluated. The `computer` function can then directly create a computer object with fully formed parameters.

A Nested Function thus works very well when building higher level structures. However it isn't perfect. The punctuation of parenthesis and commas is more explicit, but can also feel like noise compared to the conventional indentation alone. A Nested Function also implies using bare functions, so runs into the same problems of globalness as Function Sequence - albeit with the same Object Scoping cure.

The evaluation order can also lead to confusion if you are thinking in terms of a sequence of commands rather than building up a hierarchic structure. My colleague Neal Ford likes to point out that if you want to write the song "Old Macdonald Had a Farm" with function applications, you have to write the memorable chorus phrase as `o(i(e(i(e()))))`. Both Function Sequence and Method Chaining allow you to write the calls in the order they'll be evaluated.

Nested Function also usually loses out in that the arguments are identified by position rather than name. This often leads to less readable expressions than Method Chaining. You can avoid this with languages that have keyword arguments, but sadly this useful syntactic feature is very rare. In many ways Method Chaining is a mechanism that helps to supply keyword arguments to a language that lacks them. Using a Literal Map is another way to overcome the lack of named parameters.

I've written about these patters so far as if they are mutually exclusive, but in fact you'll usually use a combination of these (and those I'll describe later) in any particular DSL. Each pattern has its strengths and weaknesses and different points in a DSL have different needs. Here's one possible hybrid.

```
computer(
  processor()
    .cores(2)
    .type(i386),
  disk()
    .size(150),
  disk()
    .size(75)
    .speed(7200)
    .iface(SATA)
  );
 computer(
   processor()
    .cores(4)
 );
```

This DSL script uses all three patterns that I've talked about so far. It uses Function Sequence to define each computer in turn, each `computer` function uses Nested Function for its arguments, each processor and disk is built up using Method Chaining.

The advantage of this hybrid is that each section of the example plays to the strengths of each pattern. A Function Sequence works well for defining each element of a list. It keeps each computer definition well separated into separate statements. It's also easy to implement as each statement can just add a fully formed computer object to a result list.

The Nested Function for each computer eliminates the need for a Context Variable for the current computer as the arguments are all evaluated before the computer function is called. If we assume that computer consists of a processor and a variable number of disks then the arguments lists of the function can capture that very well with its types. In general Nested Function makes it safer to use global functions, as it's easier to arrange things so the global function just returns an object and doesn't alter any parsing state.

If each processor and disk have multiple optional arguments, then that works well with Method Chaining. I can call whatever values I wish to set to build up the element.

But using a mix also introduces problems, in particular it results in punctuational confusion. Some elements are separated with commas, others with periods, others with semicolons. As a programmer I can figure it out - but it can also be difficult to remember which is which. A non-programmer, even one that is only reading the expression, is more likely to be confused. The punctuational differences are an artifact of the implementation, not the meaning of the DSL itself so I'm exposing implementation issues to the user - always a suspicious idea.

So in this case I wouldn't use exactly this kind of hybrid. I'd be inclined instead to use Method Chaining instead of Nested Function for the computer function. But I'd still use Function Sequence for the multiple computers, as I think that's a clearer separation for the user.

This trade off discussion is a microcosm of the decisions you'll need to make when building your own DSL. I can provide some indication here of the pros and cons of different patterns - but you'll have to decide the blend that works for you.

Literal Collections

Writing a program, whether in a general purpose language or a DSL, is about composing elements together. Programs usually compose statements into sequences and compose by applying functions. Another way to compose elements together is to use Literal List and Literal Map.

The most common collection to use here is Literal List. We can indicate the actions in a new state with separate function calls

```
//java
state("idle").
  action("unlockDoor").
  action("lockPanel")
  ...
```

Or we can use a single function call with both commands in a Literal List.

```
//java
state("idle").
  actions("unlockDoor", "lockPanel")
  ...
```

In java, like most C-like languages, we use a variable argument method in order to use a list. Languages with more flexible syntax for literal collections allow us a different style where we have a single list parameter to the function instead of a variable number of parameters.

```
#ruby
state("idle").
  actions(["unlockDoor", "lockPanel"])
  ...
```

However when putting a Literal List directly in a function call, I prefer to use varargs than a real list datatype as it reduces the punctuation.

C-like languages have a literal array syntax `{1,2,3}`, but you're usually quite limited as to where you can use it and what you can put in it. Other languages, like Ruby in this language, allow you to use literal lists much more widely. You can use variable argument functions to handle many of these cases, but not all of them.

Scripting languages also allow a second kind of literal collection: a Literal Map (aka hash or dictionary). This combination of key-value pairs would be handy for handling transitions in our state example

```
#ruby
state("idle",
  {"doorClosed" => "doorClosed"}
)
```

Indeed the combination of using literal lists and maps allows variations that could support the entire definition of a state.

```
state :name => "idle",
      :actions => [:unlockDoor, :lockPanel],
      :transitions => [:doorClosed => :doorClosed]
```

Ruby often allows you to omit uneccessary punctuation so we can omit the curly brackets when there's only a single literal map present.

This example also introduces another syntactic item that's not present in curly languages, the symbol data type. A **symbol** is a data type that on first sight is just like a string, but is there primarily for lookups in maps, particularly Symbol Tables. Symbols are immutable and are usually implemented so the same value of symbol is the same object to help with performance. Their literal form doesn't support spaces and they don't support most string operations, as their role is about symbol lookup rather than holding text. Elements above like `:name` are symbols - ruby indicates symbols with a leading colon. In languages without symbols you can use strings instead, but in languages with a symbol data type you should use it for this kind of purpose.

This is a good point to talk a little about why Lisp makes such an appealing language for internal DSLs. Lisp has a very convenient Literal List syntax: `(one two three)`. It also uses the same syntax for function calls `(max 5 14 2)`. As a result a lisp program is all nested lists. Bare words `(one two three)` are symbols, so the syntax is all about representing nested lists of symbols which is an excellent basis for internal DSL working - providing you're happy with your DSL having the same fundamental syntax. This simple syntax is both a great strength and weakness of lisp. It's a strength because it is very logical, making perfect sense if you follow it. It's weakness is that you have to follow what is an unusual syntactic form - and if you don't make that jump it all seems like lots of irritating, silly, parentheses.

Using Grammars to Choose Internal Elements

As you can see, there are a lot of different choices that you have for the elements of an internal DSL. One technique that you can use to help choose which one to use is to consider the logical grammar of your DSL. The kinds of grammar rules that you create when using Syntax Directed Translation can also make sense in thinking about an internal DSL. Certain kinds of expressions suggest certain kinds of internal DSL structure.

	BNF	**Consider...**
Mandatory List	parent ::= first second third	Nested Function
Optional List	parent ::= first maybeSecond? maybeThird?	Method Chaining, Literal Map
Homogenous Bag	parent ::= child*	Literal List, Function Sequence
Hetrogenous Bag	parent ::= (this | that | theOther)*	Method Chaining
Set	n/a	Literal Map

If you have a clause of mandatory elements (`parent ::= first second`) then Nested Function works well. The arguments of a Nested Function can match with the elements of rule directly. If you have strong typing then type-aware completion can suggest the correct items for each spot.

A clause with optional elements (`parent ::= first maybeSecond? maybeThird?`) is more awkward for Nested Function, as you can easily end up with a cominatorial explosion of possibilties. In this case Method Chaining usually works better as the method call indicates which element you are using. The tricky element with Method Chaining is that you have to do some work to ensure you only have one use of each item in the rule.

A clause with multiple items of the same sub-element (`parent ::= child*`) works well with Nested Function using a variable argument parameter. A Literal List is effectively the same thing, so also fits well. If the expression defines statements at the top level of a language, then this is one of the few places I'd consider Function Sequence.

With multiple elements of different sub-elements (`parent ::= (this | that | theOther)*`) then I'd move back to Method Chaining since again the method name is a good signal of which element you are looking at.

A set of sub-elements is common case that doesn't fit well with BNF. This is where you have multiple children, but each child can only appear at most once. You can think also of this as a mandatory list where the children can appear in any order. A Literal Map is logical choice here, the issue you'll normally run into is the inability to communicate and enforce the correct key names.

Grammar rules of the at-least-once form (`parent ::= child+`) don't lend themselves well to the internal DSL constructs. The best bet is to use the general multiple element forms and check for at least one call during the parse.

Closures

One of the main purposes for Method Chaining and Object Scoping is to narrow the scope of a function call to a single object. This narrowing both provides a limited namespace for the functions that are part of the DSL and targets all the DSL calls to a single object that can hold state relevant to the expression.

A third technique to do this is to use Closures. Closures (also called blocks, lambdas, and anonymous functions) are fragments of code that can be easily declared in expressions and passed around as objects. For DSL purposes they are very good at providing a limited context for a section of the code. Consider code like this:

```
#ruby
idle = state("idle")
idle.action("unlockDoor")
idle.action("lockPanel")
idle.transition("doorClosed", "doorClosed")
```

The variable 'idle' only makes sense during the build up of its state, we don't really want it to be visible elsewhere. A closure gives us a good way to do this.

```
state("idle") do |s|
  s.action("unlockDoor")
  s.action("lockPanel")
  s.transition("doorClosed", "doorClosed")
end
```

Not just does this Nested Closure limit the scope of the variable `s`, it also allows you to use deferred evaluation. This means that, in this case, the code within the `do...end` markers isn't necessarily evaluated when the `state` method is called. The code is passed to the state method as a parameter, the method may evaluate it then and there, or it may store it in a variable for later evaluation. Even if it evaluates it then and there, it may execute other code before or after the closure code. This ability to defer evaluation makes closures particularly suitable for use with Adaptive Models.

Nested Closure naturally fits hierarchic structure as we can nest the closures freely. Their simple syntax makes them less intrusive than using different methods for each building block.

[TBD: Maybe use rake as a better example here.]

A simple and even quite common situation where you need deferred evaluation is an assert method. A useful feature in programming is an assert facility that allows you to check that something is true in a section of the code.

```
# ruby
def bonusPoints customer
  assert {customer.inGoodStanding}
  return customer.balance * 0.15
end
```

You use assertions to check assumptions in your code. An assertion failure always indicates a programming error (so you don't use them to validate data). Assertions are useful for documenting assumptions in your code and providing a fail fast mechanism to help spot errors more easily. In particular assertions should be capable of being switched off, so that they don't slow down the performance of a system.

If you are working in a language that doesn't have assertions built-in, then you might want to add them yourself. Your first thought to do this would be to do it with something like this

```
def assert condition
  if assertions_enabled
    raise "Assertion Failure" unless condition
  end
end
```

The above code partially does the job, in that the assertion will only be checked and an exception thrown if assertions are enabled. However when assertions aren't enabled the condition will still be evaluated, because the arguments to a method are evaluated before being passed into the method.

Closures allow you to avoid this because their evaluation is under program control. So an equivalent example with closures would look like this.

```
def assert &cond
  if assertions_enabled
    raise "Assertion Failiure" unless cond.call
  end
end
```

In this case the code for the condition is passed to the assert method as a block rather than as a regular argument (indicated with the '&'). The block is only evaluated when the the call method is invoked on it. (Rubyists will notice that I haven't used the more usual and compact way of doing this using 'yield', but I did it this way as it shows more clearly what's going on.)

Using a closure in this way usually involves some syntax because you need to tell the system that you are passing a closure rather than the result of the expression. The difference in ruby is that using a closure means we have to write `assert {condition}` rather than `assert(condition)`. The extra syntax is potentially awkward, but pretty unobtrusive. Often, however, closures involve a lot more syntax, for example C# 2 needs something like `Assert(delegate() {return condition();});`, which rather messes up a DSL's fluency. One of the significant improvements in C# 3.0 was to fix this with the rather better `Assert(() => return condition(););`

[TBD: check C# 3 syntax]

Interestingly lisp has this problem. It uses closures very heavily as it is a minimal language which needs closures to implement imperative statements (conditionals and loops). It doesn't have the very minimal closure syntax that ruby (and Smalltalk) has so it uses macros to avoid the messy syntax.

Closures aren't the only way to handle deferred evaluation. You can use a function pointer in C or a command object in any OO language. You can also name a function and use a reflective call. While all these approaches are possible, they do involve a lot more intrusive syntax than using clean closures.

Macros

Since I've invoked the name of Lisp, it seems time to discuss another important element of Lisp DSL writing - macros. Macros are also used outside of Lisp, but they play a central role inside Lisp's DSL practice. The idea of macros is easiest to approach in the more common form of textual macros.

Most developers have probably come across textual macros in some form, possibly through the C programming language, through web templating systems, or through specific textual macro processors.

A good example of macro usage is C's assert macro. I talked earlier about how assertions are often a challenge because they need deferred evaluation so they can be switched off in production. C's way of handling this is to use a C macro.

A C macro is a text substitution mechanism with parameters. A very simplified statement of how it is defined is something like this:

```
#define assert(expr)
if (!(expr)) abort()
```

This tells the C pre-processor to look for any occurrences like

```
assert(c > 5);
```

and replace them with

```
if (!(c > 5)) abort();
```

The real macro is rather more involved than this, printing out the file and line where the assertion failed. But the basic idea is a text substitution with any parameters referenced in the parameter list copied over to the expanded form.

In the earlier days of programming macros were used pretty heavily, as widely as function calls. In the last few decades, however, macros have very much fallen out of favor. No mainstream modern languages include macro processors in the way that C does, and although it's easy to use macro processors with any language few people do. This is because macros are very difficult to use correctly, and can easily lead to quite subtle errors.

C's macros are **textual macros**: they perform a substitution of text without reference to the syntax of the language they are embedded in. Here the characters "assert" are replaced with the characters in the body of the macro. The macros in Lisp are different: they are **syntactic macros**. You define a macro with a valid lisp expression and the macro processor replaces the macro call with another valid lisp expression. This removes a big problem with textual macros where they can be expanded in such a way that mangles the syntax. Syntactic macros are thus a step forward, but they are still liable to many of the same problems such as multiple evaluation and variable capture.

Lisp and Scheme provide more techniques to deal with these problems, but the most fundamental problem still remains - most people find macro expansion much harder to compose than function calls. This composition occurs when a macro expression expands to code that itself contains macros, which themselves are expanded to code that contains further macros. We are used to composing long sequences of function calls on the stack, but in general people have found it harder to deal with similar compositions of macros.

Certainly some people are more comfortable dealing with composed macro expansions, and there may be an argument that these are difficult to most people only because they are unfamiliar. But there is a general view at the moment that composed macro expansions should be avoided because they are hard for most people to reason about. Much though I love to be a contrarian, I do support that view, as such I'm wary of using macros.

Macros play a central role in Lisp because the basic syntax for closures is awkward, so you have to use macros to get clean syntax. Probably most of the usage of lisp macros is for this. However another valuable usage of lisp macros is Parse Tree Manipulation. Macros allow you to dive into the source of an expression and manipulate it, allowing some quite powerful forms of code surgery. Lisp's simple structure, where everything (including code) is nested lists, helps make this work. Parse Tree Manipulation is, however, independent of macros. One the big features of C# 3.0 is its ability to return the parse tree of a closure. This is a crucial enabling feature for LINQ. You can express a query over a data structure as a closure in C#. A library can take this query and get hold of the parse tree of the query. It can then navigate the parse tree and translate it into SQL, or compose query code for XML or indeed any other target data structure. It's essentially a mechanism that allows application code to do run-time code translation, generating arbitrary code from C# expressions.

Parse Tree Manipulation is a powerful but somewhat complex technique. It's a technique that hasn't been supported much by languages in the past, but these days has been getting a lot more attention due support in C# 3 and Ruby. Since it's relatively new (at least outside the Lisp world) it's hard to evaluate how truly useful it is. My current perception is that it's something that is rarely needed but very powerful on the occasions that that need arises. Translating queries against multiple data targets in the way that LINQ does is a perfect example of its usefulness, time will tell on what other applications come to mind.

Annotations

When the C# language was launched, many programmers sneered that the language was really just a warmed over Java. They had a point, although there's no need to sneer at a well executed implementation of proven ideas. One example, however, of a feature that wasn't a copy of mainstream ideas was attributes, a language feature later copied by Java under the name of annotation. (I use the Java name since 'attribute' is such an overloaded term in programming.)

An Annotation allows a programmer to attach attributes to programming constructs such as classes and methods. These annotations can be read during compilation or at runtime.

For an example lets assume we wish to declare that certain integer fields can only have a limited valid range. We can do this with an annotation with code like this.

```
class PatientVisit...
  @ValidRange(lower = 1, upper = 1000)
  private int weight; // in lb
  @ValidRange(lower = 1, upper = 120)
  private int height; // in inches
```

The obvious alternative to this would be to put range checking code into the setter of the field. However the annotation has a number of advantages. It reads more clearly as a bound for the field, it makes it easy to check the ranges either when setting the attribute or in a later object validation step, it also specifies the validation rule in such a way that it could be read to configure a gui widget.

Some languages provide a specific language feature for number ranges like this (I remember Pascal did). You can think of Annotations as a way of extending the language to support new keywords and capabilities. Indeed even existing keywords might better have been done with Annotations - from a green field I'd argue that access modifiers (private, public etc) would be better that way.

Because Annotations are so closely bound to the host language, they are suited for fragmentary DSLs and not stand-alone DSLs. In particular they are very good providing a very intergrated feel of adding domain specific enhancements to the host language.

The similarities between Java's annotations and .NET's attributes are pretty clear, but there are other language constructs that look different while doing essentially the same thing. Here's Ruby on Rails's way of specifying an upper limit for the size of a string

```
class Person
  validates_length_of :last_name, :maximum => 30
```

The syntax is different in that the you indicate which field the validation applies to by providing the name of the field (:last_name) rather than placing the Annotations next to the field. The implementation is also different in that this is actually a class method that's executed when the class is loaded into the running system rather than a particular language feature. Despite these real differences, it's still about adding metadata to program elements and is used in a similar way to Annotations. So I think it's reasonable to consider it as essentially the same concept.

[TBD: Add something on open/partial classes and Literal Extension.]

Significant Revisions

03 Sep 07: First Draft

08 Apr 08: Added early section on command-query vs fluent interfaces