

One Programmer's Ideal Language

Table of Contents

Introduction.....	1
Basic Design	2
Syntactic Features.....	3
Variables As Static or Dynamic As Desired.....	4
Highly Constrained Parameters and Return Types.....	4
Less Explicit Parameters and Return Types.....	5
Mix and Match Constraints	6
Loops	7
Testing.....	9
Running the Code.....	10
Summary	11

Introduction

There are some readable programming languages out there, and there are powerful programming languages as well, but how often do the two meet? It would be ideal to have both readability and power in a language, but it seems as though one comes at the cost of another. Lisp is an incredibly powerful language, but it is not very readable due to its syntax. Ruby has some constructs that make code read almost like English, but its speed has been criticized when compared with other dynamic languages [1] [2]. Still other languages offer convenience to the programmer through their unique features.

Perhaps it is a pipe dream, but it would be nice if a single language could combine speed, expressiveness, readability of code, and power, all while being convenient and intuitive to write. Some would argue that these traits are already included in an existing language, and the degree to which these goals are met is certainly subjective. Since one person's elegant construct may be another's cluttered obfuscation, let this paper then reflect this particular developer's description of a more perfect language. The goal is to select the nicest features from existing languages as well as include other features that perhaps have not been seen before. The result might not be easily implementable, but that is the scope of this paper as well: to explore interesting ideas for a language beyond what is already known to work.

Basic Design

For this programmer, the ideal paradigm for a language is object-oriented. However, languages like Java where all code must be within a class feel artificially limiting; sometimes, a particular method does not seem to fit in a class. Situations arise where a class exists only as a utility class to hold static methods that do not fit anywhere else. Taking an idea from Ruby, the language will allow functions to exist outside of classes; likewise, code can exist outside of functions or classes. However, like Ruby, this hypothetical language could be entirely object-oriented. Ruby allows standalone functions by turning these functions into methods on the Object class:

```
>> (Object.new).respond_to? :foo
=> false
>> def foo      // Standalone function
>> puts 'hey'
>> end
=> nil
>> (Object.new).respond_to? :foo
=> true
```

This allows the programmer to organize code however he or she sees fit, and the language remains consistent with itself. Many conveniences, or shortcuts, may be allowed in the language to cater to the programmer's expectations, but it would be preferable for these conveniences not to violate the language's fundamental ideas.

As an object-oriented language, all operations should be methods called on an object. Part of the Python language is that some operations are functions that are available for any object, but are not declared as methods on an object; for example, the built-in function `len()` [3]. Having many operations as methods but a few as functions seems arbitrary and confusing; this should be avoided in the hypothetical language being described here. Such constructs disrupt the regularity of a language without seemingly providing any extra convenience: how is `len(myArray)` any more intuitive or concise than `myArray.len`?

A newline will separate statements, or a semi-colon if two statements are placed on the same line. While many programmers favor Python's whitespace idea, it is not appealing to this particular programmer; hence this language will not use whitespace for block structure. See the Syntactic Features section for a description of block structure syntax. To separate units of code, programmers can use classes and namespaces, as in C#. Errors will be handled using exceptions, as in C# and Ruby.

As in Smalltalk and Ruby, classes will be open such that new methods can be introduced at runtime to existing classes [4] [5]. That means new class and instance methods can be added as necessary to even core library classes. This is less limiting than the current version of C# wherein only new instance methods can be added to existing classes [6]. Built-in support for regular expressions has been very useful to this programmer when using Ruby and Perl, and so it should be included in this language as well.

Syntactic Features

One of the enjoyable things about programming in the Ruby language is its syntax. Often characters can be left out or added such that the code is more readable to the programmer, according to his or her preference. For example, parentheses are not necessary around conditions in loops and `if` statements, but they can be added without changing the functionality of the code. Sometimes for clarity parentheses are necessary, when it would be ambiguous to omit them; otherwise it is programmer's choice. Semicolons likewise are not necessary but can be added as line terminators. The lack of such characters is not a new idea but having the option to omit them helps make programming in Ruby feel more like writing pseudo-code. Nice syntax allows the programmer to focus on the algorithms and design of his or her project rather than struggling with the details of the language.

Another feature is the `unless` statement, which acts exactly like "if not" but can sometimes make more sense when reading the code. A person might think "do this unless that," and in a less expressive language that might have to be changed to fit the language, e.g. `if (!that) this;`. However, Ruby allows one-line `if` and `unless` statements to precede the `if/unless` condition, which fits more with the original human thought: `this unless that`. Such small details can make a programming language more comfortable to the programmer because it requires less thought to mentally translate the language and understand the intention behind the code.

One concept that seemed useful is that of pattern matching in functions in the ML language. Instead of explicitly having separate parameters that then must be checked to ensure they align in the way the programmer expects, the description of the parameter in the function header would provide such checks. Consider the following Standard ML: `fun loop (x::y::zs, xs, ys)`. This is a function header that requires a list with at least two elements, `x` and `y`, be passed in as the first parameter. This not only requires such a parameter that fits this pattern, but it also names the first two elements, as well as the tail of the list as `zs` (every element in the list following the first two elements, if there are any others). These are named variables to which the programmer can refer within the body of the function. Doing something similar in another language might involve taking in a generic parameter, or even a typed parameter of some `List` type, and then checking the contents of the variable in the body of the function via an `if` statement. The expressiveness and concision of the ML code is appealing.

Now is a good time to describe the ideal balance (recall that this the ideal for this particular programmer) between expressiveness and readability. Lest the language travel the same path as COBOL wherein it tries too much to be like English, it would be better to have a mixture of readability and concision. Concepts such as parameter constraints are used often and it would be convenient to have a concise way of expressing them. For other concepts in a language, the most concise way of expressing the concept while retaining readability is preferred.

Variables As Static or Dynamic As Desired

For this developer, one of the features missed when using a statically typed language like C# is the flexibility of a dynamically typed language like Ruby. However, the opposite is true when using a dynamic language. The ideal language might allow the programmer to specify exactly how specific the value placed into a variable must be. One could write `int myIndex = 3` or simply `myIndex = 3`, and the language would handle both cases by assigning the integer value three into the variable `myIndex`. Constraints other than just value type could also be available, e.g. `Numeric myValue <20 && >3.5` could be used to say that `myValue` can be initialized to any numeric type, such as `int` or `double`, so long as the value is less than 20 and greater than 3.5. Beyond variable declaration, this could be extended into method and function declarations as well.

Highly Constrained Parameters and Return Types

Statically typed languages such as C# and Java require that a method describe its parameters in the header via declaring their types. The same goes for local variables, fields in a class, return types, and so on. What if a language could describe more than just the type? Constraints on value would be helpful. For example, consider a method that returns an integer. That is easily enough expressed in existing languages, but if more rigorous validity checks are required, they are probably expressed via `if` statements or exception handling done within the body of the method. It seems reasonable to assume that many operations in a program have an expected or valid range for their values. Why then is this not a more ingrained part of the language? Consider the following code sample:

```
def myFunction(List list of x::y::_:n,  
               int maxIndex < list.size && > -1)  
  // body here  
end int < list.size && > -1
```

This is hypothetical code defining a function that takes two parameters. The function requires that the first parameter be an instance of class `List` and that it have:

1. A first element that we call `x` for the duration of the function body;
2. A second element called `y`;
3. A body of unknown length, signified by the underscore; and
4. A final element called `n`.

This syntax goes one beyond the ML pattern matching in that it gives a name to the entire list (namely, 'list'), but also specifies individual elements within the list. Because of how the 'list' parameter is described, it must be at least three elements long. Being able to reference the entire list is useful for operations that deal with the whole thing and not just its individual components, e.g. `list.length`, `list[4]` to get the fifth element (arrays and lists are indexed by zero). Accessing elements in the list at a certain index could be done by referring to the `_` parameter, which is also a list type, but then an offset is necessary since the two elements `x` and `y` are included.

The second parameter to `myFunction` is an integer that is bound not as the list was by its contents, but by its value. The function will take an integer whose value is less than the size of the given list parameter and greater than -1. Notice that 'int' is lowercase while the 'List' class of the preceding parameter is capitalized; this is inspired by C# wherein the lowercase 'int' refers to class `System.Int32`. The lowercased 'int' is provided as a convenience since integers are used frequently, and 'int' is familiar to those coming from a C/C++ background. This aligns with the stated goal for the hypothetical language to be concise while remaining readable. The language should be intuitive, and 'int' could be called intuitive to programmers with a C background, so why require the programmer to type more?

The function also sets bounds for the value it will return. It does not do this in the usual place, however, before the function name, but rather after the function body, directly following the end statement (which was taken from Ruby's block system). In this programmer's experience, a function header can get long and unwieldy when function names are descriptive, long class names are used for parameters or the return type, or several parameters are used. With the extra information added to the header to describe parameters with more detail than just their type, function headers can become especially long.

Many integrated development environments (IDEs) and editors provide ways of folding or collapsing the bodies of functions, and this could be done with the hypothetical language being described in this paper. That would place the return type near the function header for easy assessment by the programmer. This touches upon another idea that could be leveraged by the language: many programmers use IDEs or editors that offer features such as syntax highlighting and code folding (provided by Vim, Eclipse, Komodo Edit, and Visual Studio, for example). Knowing such details of how programmers will use the language can lead to different language design, such as describing the return type of functions at the end of the function rather than the beginning. This ideal language should be a modern one that makes use of how programmers commonly program.

The function will return an instance of type 'int' which, following the C# idea, would probably refer to an `Integer32` class. However, the function is guaranteed to return an integer whose value is less than the size of the parameter 'list' and greater than -1. The scope of variables referenced in the return type description is that of the parameters given in the function header. It is runtime checkable that the value being returned from the function fits these size restraints; if it does not, the language throws a runtime exception automatically. It is perhaps even compile-time checkable if the body of the function returns a constant value known at compile time.

Less Explicit Parameters and Return Types

The level of detail shown with the previous example does not necessarily have to be part of every function definition or variable declaration. Going along with the idea of letting the programmer choose what is necessary, the same function might be expressed thus:

```
def myFunction(List list, int maxIndex)
```

```
    // body here  
end int
```

Here we see a function taking two parameters where all that is required is that the first is an instance of class `List` and the second is an instance of class `int`. The function must return an instance of `int`. The body of the function could include checks to ensure the given parameters have values within an expected range, if the programmer so desired, and that the value to be returned is valid. There could also be flexibility in the syntax such that the return type could be listed before the function name instead of after the function body. Also like Ruby, there could be flexibility with block structure: instead of saying 'end', curly braces could be used to delineate the function body. Though this is not allowed in Ruby for function or method declarations, it is allowed for code blocks and anonymous functions.

To go along with being convenient for the programmer, this hypothetical language could take the same idea as Cobra when it comes to binding: "Cobra takes the position that when it can do so, it will bind method calls early, in the same way that C# chooses to, but when it can't, for whatever reason, it will choose instead to use late-bound semantics and resolve the method call at run time" [7]. This might allow a combination of static and dynamic typing such that a programmer can specify as many or as few details as is desired, or necessary, about parameters, variables, and return types. The same function example might be defined yet a different way as follows:

```
def myFunction(list, maxIndex)  
    // body here  
end
```

This looks more like Ruby or Python than a statically typed language. All that is required is that two parameters are given, and whether the body of the function checks the parameter values or not is up to the programmer.

Mix and Match Constraints

In Ruby and Python, one can pass any type of variable to a method. This kind of flexibility is nice, but sometimes the programmer may want to limit the types of allowed parameters but still allow a range of them. This sounds akin to C++ templates, or perhaps C# generics, wherein several different types of value can be passed to a method when the method requires a generic type. With C# generics, one can describe the unknown type as implementing a particular interface. With the ideal language being described here, it would be nice if it were not necessary to create a separate interface and then require an instance implementing that interface. Consider the following:

```
def myFunction2((string|bool) flag,  
               value has [:null?, :equals])  
end
```

This code defines a function named `myFunction2` taking two parameters. The first parameter can be either of type `string` or of type `bool`, and it is called `flag`. The second parameter is more interesting: it uses the built-in `has` operator to require that the parameter named `value` responds to methods `null?` and `equals`. The question mark at the end of `null?` is part of the method name; this is taken from Ruby where it is convention to name predicates with a question mark to indicate they return a Boolean value. The `has` operator can be passed either a single method name or a list of method names; a list literal in this language is like Ruby's: bounded by square brackets. Also taken from Ruby is the colon before the method names given to the `has` operator. The colon is used to declare a symbol in Ruby. A symbol is much like a string but only one copy of a symbol can exist in memory, whereas two string literals with the same value are different objects in Ruby. Further reading about Ruby symbols can be found on Ruby Doc [8].

Note that this function does not declare its return type, so any value can be returned. Also note that while the first parameter's type must be one of `string` or `bool`, the second parameter can be of any type so long as it has the two methods specified. This is akin to Ruby's duck typing [9] where the idea is if a variable walks like a duck and talks like a duck, treat it like a duck. That is, since the variable responds to the two methods we expect, we do not have to worry about what its type is, we just use the methods that are guaranteed to be there. In Ruby, parameters in a method cannot be constrained by the methods to which they respond: a programmer must do checks within the body of the method to ensure the given values respond to the methods he or she expects. The hypothetical language described herein takes Ruby's implementation of duck typing a step further by allowing constraints within the method header, such as shown above. This is a more concise way than the following Ruby alternative:

```
raise ArgumentError unless value.respond_to?(:null?) &&  
value.respond_to?(:equals)
```

Again, this hypothetical language is trying to make the language more usable by making concise a feature that is useful but otherwise lengthy.

Loops

There are a few ideas from other languages about loops that should be included in the hypothetical language, such as the use of iterators as in Ruby. One idea that should be avoided is PHP's `foreach` loop, which has the syntax `foreach (array as value), or foreach (array as key => value)` [10]. It is this programmer's belief that having the array variable come first in the loop condition is unintuitive, and that the 'as' keyword is not clear.

Most loops seen in the Ruby language are done through the use of iterators within a method on an object. Looping over values in an array is as straightforward as calling a method on the array. This goes along with the Principle of Least Astonishment [11] in that loops are not a special construct in the language but

rather are treated as any other method call. Here is an example use of a loop in Ruby, and how it might be in the hypothetical language as well:

```
[1, 2, :a, 'cat', 3.5].each do |myValue|  
  // do something with the current value, stored in myValue  
end
```

Here we see a heterogeneous array of values and the `each` method is called upon this array. The `each` method contains an iterator that, when asked, will supply the next value in the array to the given code block. As in Ruby, code blocks, or closures, are denoted by the `do` and `end` keywords or by curly braces. The current iteration's value is stored in the programmer-defined variable `myValue`. To access the current index in the array as well as the current value at that index, the method `each_with_index` in Ruby is used, and likewise could be implemented in this custom language. The method `each_with_index` requires a block taking two parameters: one for the index and one for the current value.

Regular `for` and `while` loops will be supported as well, as in the style of Ruby or C#. The syntax is as follows:

```
for value in array_of_values { ... } // foreach-style loop  
for i=0; i<10; i++ { ... } // for loop  
while condition { ... } // while loop
```

Here we see use of the `++` method which exists in C# and other C-like languages, though not in Ruby. It is a convenient method and more concise than Ruby's equivalent of the form `variable += 1`, so it should be included. Likewise the `--` method, but to avoid some of the confusion of C wherein the meaning of the method changes depending on its location relative to the variable, these two unary operators will only be valid when written after the variable. In fact it would make no sense to write them before the variable because no prefix methods will be allowed in this language: the pattern is `object[.]method`, and allowing syntax such as `--value` would represent `method[.]object`.

Also seen in the above code in the second `for` loop is the more C-like style for a `for` loop's condition. This is familiar to many programmers, at least to this programmer, and as an expected, concise construct it will be included in the language. It behaves as a programmer familiar with C might expect: the initialization statement, the condition to test before beginning an iteration, and the operation to happen at the end of every iteration. Unlike in C but like in Ruby, parentheses are not necessary.

It might be convenient to allow one-line loop bodies to precede the loop header. This also makes sense given the decision to allow one-line conditional bodies to precede the `if/unless` statement. Allowing this helps the regularity of the language since one might expect to be able to do this with loops if it is available with conditional statements. However, it might be considered feature creep if many developers do not use it. Here is an example of a one-line loop body preceding its loop condition:

```
print "hello " + i for i=1 to 10
```

This piece of code shows several new pieces of syntax that give insight into the details of our hypothetical language. The `print` statement looks to be a method called on no object, which violates the regularity of the language. However, as in Ruby, the standalone method `print` is provided as a convenience to the programmer, and is actually a method in the module `Kernel`, which is used as a mixin to the class `Object` [12].

We also see that the arguments to a method are not required to be surrounded by parentheses, and that string concatenation is done via the `+` method. This is also taken from Ruby. However, one difference from Ruby that is actually found in PHP and Java is the implicit conversion of the variable `i`, which is obviously an integer, to type string when it is passed as a parameter to `+`. This is provided as a convenience to the user; otherwise an explicit cast or perhaps a call to a `toString` method would be necessary before `i` could be appended to the string literal in the loop body.

The compiler or interpreter would know where the variable `i` is defined once it reaches the `for` statement, which is a reserved word in this language. This piece of code also illustrates a different type of `for` loop in which an initial value is given and a terminating, inclusive value is defined after the `to` keyword. String concatenation is done with the `+` method, as in Ruby, and also like in Ruby, the explicit period separating the object from the method name is omitted. This goes against the regularity of the language, but provides a convenience to the programmer, where `"hey" + " you"` might be more readable than `"hey".+ " you"`.

Testing

Testing code is an important part of writing code to ensure segments function as expected, and to prevent regression failures later when new code is added or modified. Many languages have testing suites as separate libraries, either provided by the original language developers or by a third party. However, it is an interesting and perhaps useful idea for something as basic as a unit test to be closer to the code it tests. Taking an idea from the Cobra programming language, tests could be part of the hypothetical ideal language. Tests could be written as part of the class that they test. A feature like Java's annotations could be used to associate the test with its method, like so:

```
def foo(int input1, int input2 < 5)
  // some code
end

@TestMethod(:foo)
def fooTest
  // code to test foo() above
end void
```

Here we see a method `foo` and one unit test called `fooTest`. The name of `fooTest` is irrelevant to the language, rather it is the `@TestMethod` attribute that 1) declares `fooTest` to be a test and 2) associates `fooTest` with the method being tested, i.e. `foo`. A symbol like in Ruby is again used to identify the name of the method being tested (see the Mix and Match Constraints section for more about Ruby symbols). As in the Visual Studio 2008 Test Suite, the annotation to denote a test method is 'TestMethod'. Also note that `fooTest` has no parentheses in its header; this again is taken from Ruby wherein a method that takes no parameters does not need to list empty parentheses after its name.

This method and its test method also show the commenting style of the hypothetical language. Single line comments are done with two forward slashes, and a multi-line comment would be done as in C++: `/*` followed by `*/`. This is a common commenting scheme in C-like languages as well as the web style sheet language CSS [13].

As with Cobra, when the developer wants to run the tests, a command-line option can be given to the interpreter/compiler. If the tests pass, the code will be compiled: "When run with the '-test' option at the command-line, Cobra will execute the unit tests contained in the class, and assuming they all work as expected, Cobra will report the tests pass, as well as compile the code" [14].

Running the Code

The language so far is sounding very expressive and customizable according to the programmer's taste, but speed might be a problem. It would be nice if it could be both interpreted for quick development, but also be compilable to increase speed when a program is run. If the most dynamic version of `myFunction` above were used incorrectly but the whole program was run as an interpreted program, then the misuse would be a runtime error. If the more constrained versions of `myFunction` were used incorrectly but run through an interpreter, the program would act as Ruby does and fail upon trying to execute the erroneous code. For the compiled version of the program, using any of the `myFunction` implementations, if the function were called incorrectly (i.e. one of its parameters did not match the runtime-knowable constraints), the code would compile but a runtime error would be thrown. If a programmer tried to pass `myFunction` a parameter that is compile-time checkable and invalid, the code would fail to compile.

Likewise the interpreter/compiler would fail if the constraints on a method parameter did not make sense with respect to the parameter. Consider the following:

```
def badConstraint(int value < "cat",  
                 double value2 > 1.5 && < 0.3)  
end
```

Here, the method `badConstraint` wants an integer whose value is less than the string "cat", which makes no sense and would cause the code not to compile. The compiler would know there is no instance of the `<` method on the `int` class such that

an argument of type `string` is accepted. The parameter constraints on `value2` would cause a runtime error because the compiler would not be able to determine if the actual parameter fit those constraints or not. Assuming the constraint on the first parameter were fixed, if the code was run and `badConstraint` was called, no matter what instance of `double` was given as the second parameter, a runtime error would occur.

The method `badConstraint` above *could* be made to work if the programmer overrode the `>` or `<` methods on the `double` class such that, for some values, when `>1.5` and `<0.3` are called upon those values, the result is true. See the Basic Design section for more on open classes and being able to override existing methods.

One might be concerned that having unit tests alongside regular code (see the Testing section) could cause unnecessary bloat in code size or decreases in speed when the code is deployed. The designers of Cobra were concerned with the same thing and allow a particular command-line option that strips out the tests in the compiled executable [15]. This seems like a good feature to have in this hypothetical language.

Summary

The hypothetical language that has been described herein has mainly been a conglomeration of the most convenient (to this developer, at least) features and syntax seen in other languages. Some features have been added to increase the regularity of the language (such as one-line loop bodies being allowed to precede the loop header) and others have been added to increase convenience (such as multi-line C++-style comments and the `++/--` unary methods). Some features found in other languages have been intentionally avoided in this language due to their perceived frustration or lack of clarity (such as Python's whitespace dependency and PHP's `foreach` loop syntax). One large feature that this programmer has heretofore never encountered in a language is the ability to constrain variables and method parameters with any number of type and value requirements, including no such requirements at all.

It is unknown whether such a language as has been described could be efficiently implemented, and in such a way that programs written in this language could execute in a reasonable amount of time. This programmer has never written any software as low-level as a compiler or interpreter and as such does not know what are the limits with respect to these tools. This language description comes as the result of a decade of programming in several languages, many of which were mentioned, and all of which were found lacking in some way. They were lacking in that they did not support features that seemed expected, convenient, or were found to be so in other languages that had them. It is this programmer's hope that gathering convenient syntax and functionality in one language would be helpful to programmers. Being able to write a program while thinking less about the syntax of the language and more about the design of the system at hand is ideal. If that program is then just as readable to future maintainers, better software can be produced.

References

- 1 Joel Spolsky, "Ruby Performance Revisited", *Joel on Software*, <http://www.joelonsoftware.com/items/2006/09/12.html> (accessed December 1, 2009).
- 2 Dhananjay Nene, "Performance Comparison – C++ / Java / Python / Ruby/ Jython / JRuby / Groovy", */var/log/mind*, <http://blog.dhananjaynene.com/2008/07/performance-comparison-c-java-python-ruby-jython-jruby-groovy/> (accessed December 1, 2009).
- 3 Python Software Foundation, "Built-in Functions", *Python v2.6.4 Documentation*, <http://docs.python.org/library/functions.html> (accessed December 6, 2009).
- 4 "Smalltalk Metaprogramming", *CS 2340 Summer 2007 Georgia Tech*, <http://coweb.cc.gatech.edu/cs2340/6243> (accessed December 7, 2009).
- 5 Werner Schuster, "Ruby's Open Classes - Or: How Not To Patch Like A Monkey", *InfoQ*, <http://www.infoq.com/articles/ruby-open-classes-monkeypatching> (accessed December 7, 2009).
- 6 Microsoft Corporation, "Extension Methods (C# Programming Guide)", *MSDN Visual C# Developer Center*, <http://msdn.microsoft.com/en-us/library/bb383977.aspx> (accessed December 7, 2009).
- 7 Ted Neward, "Reaping the Benefits of Cobra", *MSDN Magazine*, <http://msdn.microsoft.com/en-us/magazine/dd882513.aspx> (accessed December 7, 2009).
- 8 Members of the Ruby community, "Class: Symbol", *RubyDoc*, <http://ruby-doc.org/core/classes/Symbol.html> (accessed December 6, 2009).
- 9 Wikipedia contributors, "Duck typing," *Wikipedia, The Free Encyclopedia*, http://en.wikipedia.org/w/index.php?title=Duck_typing&oldid=329271533 (accessed December 7, 2009).
- 10 The PHP Group, "PHP: foreach – Manual", *PHP Manual*, <http://php.net/manual/en/control-structures.foreach.php> (accessed December 7, 2009).
- 11 Wikipedia contributors, "Principle of least astonishment," *Wikipedia, The Free Encyclopedia*, http://en.wikipedia.org/w/index.php?title=Principle_of_least_astonishment&oldid=326643979 (accessed December 7, 2009).
- 12 Members of the Ruby community, "Module: Kernel", *RubyDoc*, <http://ruby-doc.org/core/classes/Kernel.html> (accessed December 7, 2009).

13 Wikipedia contributors, "Cascading Style Sheets," *Wikipedia, The Free Encyclopedia*,
http://en.wikipedia.org/w/index.php?title=Cascading_Style_Sheets&oldid=329676798 (accessed December 7, 2009).

14 Neward, "Reaping the Benefits of Cobra".

15 *Ibid.*