Syntactic Typology of Programming Languages

Luke Palmer

May 5, 2006

Abstract: *Programming languages, interfaces for telling computers what to do designed in a Human-friendly way, exhibit many of the same phenomena as natural Human languages. This paper categorizes programming languages along two orthogonal axes. It then describes some of the properties that each programming language family has and similarities between programming language culture and natural language culture.*

1 Paradigm Axes

It is widely believed throughout the programmer's community that there are four fundamental paradigms of programming: Procedural, Object-oriented, Functional, and Logical. However, [2] provides an analysis enumerating six different paradigms as a combination of two axes, called the *control style* and the *abstraction style*. These are semantic styles and in theory they have nothing to do with the syntax, but in practice, they are highly correlated to syntax.

Computer programs are built inductively, by building a complex structure out of smaller forms of the same structure. The first axis, control style, indicates what kind of structure is being built. It takes on three values: imperative, functional, and logical.

Imperative programs build a sequence of instructions for the computer to execute like a cooking recipe. This is done inductively using *procedures*, which are smaller sequences of instructions that are combined together to make the final program. For example, they may issue an instruction "ask for the user's age", then do some computation and decide to issue the instruction to "display 'you are too old'". Imperative programs are so called because every statement is read as though it were in the imperative mood.

Functional programs build a large mapping from input to output, out of smaller such mappings called *functions*. As a contrast to the imperative paradigm, a functional program would implement a function which takes a number as input (representing the user's age) and returns a string (the description of their age) as output.

Logical programs build a complex boolean (true or false) condition that the computer tries to make true by assigning to variables. The logical languages do not make up an interesting syntactic category, so this is the last we will hear of them.

The other axis, abstraction style, indicates how the smaller pieces are layed out to be combined. There are two common abstraction styles: procedural and object-oriented.

Procedural abstraction is essentially a substitution filter. For example, we could build a procedural English sentence like so: Let $L(x) =$ "I like x"; I told Mark that $L(\text{Jane})$.

Object-oriented abstraction, instead, defines names as conceptual entities to be associated to some object, whose behavior may differ based on the type of that object. English is already object-oriented in a sense, where we can say "I rode my horse", and "I rode my bike". The speaker is riding something, but the way in which a horse and a bike are ridden are quite different, even though they have the same grammatical structure.

Figure 1 presents table of common languages, most of which we will discuss in this paper, categorized by their control and abstraction styles. Many of these languages are multi-paradigm, so they have been categorized by their most common use.

	Procedural	**Object-oriented**
Imperative	C, Pascal, ALGOL	C++, Java, Smalltalk
Functional	Scheme, ML	O'Caml, Haskell, Lisp
Logical	Prolog, Curry	*none*

Figure 1: Common languages categorized by their control and abstraction styles.

C/Java/C++	**Python**	**Perl**	**PHP**
`int x = 1;`	`x = 1`	`my $x = 1;`	`$x = 1`
`x++;`	`x = x + 1`	`$x++;`	`$x++;`
`array`	`array`	`@array`	`$array`
`array[4]`	`array[4]`	`$array[4]`	`$array[4]`
`foo(4)`	`foo(4)`	`foo 4`	`foo 4`
`obj.member`	`obj.member`	`$obj->member`	`$obj->member`
`obj.method()`	`obj.method()`	`$obj->method`	`$obj->method()`

Figure 2: Contrast in the C language family.

2 Control Style

2.1 Imperative Syntax

An extreme majority of languages used today are imperative. According to `freshmeat.net` statistics[4], the top six languages in use in open source projects are C, Java, C++, Perl, PHP, and Python, which account for 95% of all projects. *All six* of these are imperative languages[1]. C, Java, and C++ have very similar syntax to each other (we will call these C languages), Perl and PHP are syntactic derivatives of the C languages. Python is the only outlier, having a very different "first glance" look from the other five, but some analysis shows that it is still very close.

These six languages are almost precisely the same in terms of syntax. Their differences could be considered morphological, or even literal. The comparison chart in Figure 2.1 shows some of the differences.

There seems to be a fair amount of variation, but the word order is always the same, and the operators, when not identical, can easily be transliterated. Also keep in mind that, except for constructs present in one language and

[1]Perl could be considered on the border between imperative and functional, but common use is in its imperative form.

completely missing in another, these are almost the only the variations.

There are some key differences. The largest variation is demonstrated on the first line where the programmer declares a new variable. Each of the languages does something different here. The C-languages declare a new variable by stating its type (`int` (for integer) in this case, but it could easily be `float`, `class String`, `double (* const)(int, char)`, etc.). None of the other languages declare a type. Perl uses the placeholder `my` to indicate that a variable is being declared. Python uses the fact that a variable is being assigned to declare the variable. PHP uses any reference of a variable to be an implicit declaration. The following example clarifies the distinction between the last two:

```
# Python code
x = 4
def foo():
    x = x + 1  # make x 5

# PHP code
$x = 4
function foo() {
    $x = $x + 1;  # declare a new $x (defaulting to 0)
                  # and add 1 to it
}
```

Another interesting distinction is that Python uses indentation to determine where blocks are, whereas all the other languages use curly braces. The above example demonstrates that as well. This is where Python gets it unique look.

2.2 Functional Syntax

Functional languages account for even less than the remaining 5% of the most common languages. Nonetheless, there is much more variety in the syntax of functional languages than that of the imperative languages. There are two possible reasons for this:

1. All of the functional languages covered here emerged out of academic research, whereas most of the imperative languages above emerged out

Lisp	**ML**	**Haskell**
`(let ((x 4)) x)`	`let val x = 4 in x end`	`let x = 4 in x`
`(+ x 1)`	`x + 1`	`x + 1`
`(if 'nil 0 1)`	`if false then 0 else 1`	`if False then 0 else 1`
`(lambda (x) (+ x 1))`	`fn x => x + 1`	`(+1)`
`(foo x y)`	`foo x y`	`foo x y`

Figure 3: Contrast in the functional languages.

of industrial research. Academics are more willing to experiment with strange new ideas, because there is not nearly as much on the line if they get it wrong.

2. Imperative languages have a fairly straightforward compositional approach: do this, then do that. Complex units are combined in simple ways. Functional languages instead combine simple units in complex ways. That is, imperative programming is about content words; functional programming is about function words.

We will focus on two language families: Lisp/Scheme (the Lisp family) and ML/Haskell/O'Caml (the ML family). Lisp and Scheme have precisely the same syntax (they differ in their semantic models and libraries). ML and O'Caml have similar syntax, but O'Caml adds some features. Figure 2.2 gives a comparison of some syntactic features of Lisp, ML, and Haskell.

It would take more space than we have here to explain all the differences demonstrated in that table. One thing to notice is that Lisp is minimalist in a sense. All function calls look like `(func arg1 arg2 ...)`, and everything is a function. ML and Haskell provide a lot more "syntactic sugar", to make programs easier for Humans to read.

3 Abstraction Syntax

Figures 3 and 3 compare features across the procedural and functional languages in different control styles. There is vast variation here: word and clause order, naming conventions, braces, argument passing styles, etc. In all of these languages, classes and objects (the units that make up object-oriented programming) behave approximately the same, but the programmers from different languages think about these concepts very differently.

C	Scheme	ML
`func(x)`	`(func x)`	`func x`
`&fptr`	`(lambda (x) (fptr x))`	`fptr`
`int foo() {...}`	`(define (foo) (...))`	`fun foo ():int`

Figure 4: Contrast across procedural languages.

C++	Smalltalk	Haskell
`obj.meth(arg)`	`obj meth: arg`	`meth arg obj`
`class Foo : Bar`	`Bar subclass: #Foo`	`class (Bar a) => Foo a`
`int foo(int x) {...}`	`foo | x | ...`	`foo x self = ...`

Figure 5: Contrast across object-oriented languages.

For example, the second line defines `Foo` as a subtype of `Bar` (for example we would define `Dog` as a subtype of `Animal`). However, C++ programmers would translate into English as "Define a class `Foo` derived from `Bar`". Smalltalk programmers would translate "Tell `Bar` to subclass itself into `Foo`". Haskell programmers would say "Any `Foo` must also be a `Bar`". The end meaning is the same, but the mental space when dealing with these concepts is very different.

4 Relation to Natural Language

Constructs in programming languages are usually inspired by constructs in Human language. Every language described here has been developed by native English speakers[2].

There is one very interesting languages to look at when considering natural language. This is Perl, created by linguist Larry Wall and modeled after English.

Perl was designed as an "idiomatic" language. That is, Perl gives the programmer a lot of syntactic freedom, and relies on the culture to enforce certain styles to maintain readability. In this way, Perl gathers a very Human-like culture. "Foreign" speakers—those who come from other programming language backgrounds—write Perl as if it were a transliteration of the language they are comfortable with. The more a person codes in Perl, the more

[2]With the exception of C++, created by Danish computer scientist Bjarne Stroustrup. But C++ is such a close derivative of C that it doesn't really count.

it starts to look “native”, but programmers may always maintain some kind of “accent”. Here are three ways in Perl to compute the factorial of a number `$n`.

Perl with a C accent

```
my $result = 1:
for (my $i = 1; $i <= $n; $i++) {
    $result *= $i;
}
```

Perl with a functional accent

```
sub fact {
    my ($n) = @_;
    if ($n == 0) { 1 }
    else         { $n * fact($n-1) }
}
my $result = fact($n);
```

Idiomatic Perl

```
my $result = reduce { $a * $b } 1..$n;
```

Another phenomenon to explore in programming languages is the sentential complexity present in functional programs. Consider the following “Schwartzian Transform” in Haskell:

```
map snd (sort (map (\x -> (f x,x)) list))
```

The two important things in that sentence are `sort` and `list`, both of which are hidden in the depths of the expression. Also note the following “center embedded” sentence:

```
maybe Nothing (maybe Nothing (maybe Nothing (lookup locations))
        (lookup addresses)) (lookup names) x
```

Such expressions are common in functional programs, and notoriously hard to read. That is to be expected, as it is equally hard to read the following example from [1]: “The boy the girl the teacher flunked kissed cried”. This may be one of the hidden causes preventing functional programming from making it into mainstream code culture.

5 Conclusions

The control style of a programming language correlates with its syntactic structure, but the abstraction style shows no correlation. This could indicate that the control style determines the way people think a more heavily than the abstraction style. Categorizations of the world vary in usefulness across domains, and Humans are good at understanding new categorizations because they depend on it to survive. The abstraction style of a programming language is a categorization of the concepts of a program. However, Humans are typically not very good at coming up with new interpretations of how the universe as a whole behaves (quantum mechanics, mathematically simple at its core, continues to haunt and perplex even accomplished physicists). Rethinking how your program executes—the control style—is analogous to rethinking how the universe behaves. This is hard for Humans, so new concepts in that area are rarely defined, so new language structures don't have any way to get in.

Even in explicitly-designed, special-purpose programming languages, we find a lot of the syntactic phenomena found in natural languages leaping out of anyone's specific intention. From the Mandarin Lisp programmer who can not comprehend the Cantonese Haskell programmer to the Californian C++ programmer who just thinks New York Java is rude, language boundaries and attitudes form inside programming cultures. C programmers speak an awkward Perl accent and seek help from their experienced, native friends in the culture. It seems that such linguistic phenomena appear whenever humans have to learn any syntax-semantic mapping.

References

[1] Abe Y., Kazumi H., Cowan J R. (1988). Investigating universals of sentence complexity. *Typological Studies in Language 17*, 77–92.

[2] Palmer L. (2005). *What is the fourth paradigm?* Retrieved 2005-05-05 from `http://use.perl.org/~luqui/journal/27651`.

[3] Randal A., Sugalski D., Tötsch L. (2003). *Perl 6 Design Philosophy.* Retrieved 2005-05-05 from `http://www.perl.com/pub/a/2003/06/25/perl6essentials.html`.

[4] Welton D. (2004). *Programming Language Popularity.* Retrieved 2005-05-05 from `http://www.dedasys.com/articles/language_popularity.html`.

[5] Wikipedia. *C programming language.* Retrieved 2005-05-05 from `http://en.wikipedia.org/wiki/C_programming_language#History`