

Buddy, Can You Paradigm?

by Chuck Allison

Lee Copeland, managing technical editor of *Better Software* magazine, recently wrote of the Whorfian Hypothesis, which, according to Wikipedia.org, “argues that the nature of a particular language influences the habitual thought of its speakers.” As early as 1957, Kenneth Iverson, inventor of APL, prefaced his presentation of that interesting language with the phrase “Language is a tool of thought.” His goal was to efficiently process large arrays of data in multiple dimensions, so APL sports numerous operators, all very succinct and mutually orthogonal. When it’s time to cut code, the programming language we’re using tends to govern our thoughts. A C programmer will conjure up appropriate functions; a Java programmer will see nothing but objects; a developer using Haskell will traffic in lists and higher-order functions.

Each programming style, or paradigm, has its claim to fame, and one is not necessarily better than another. A message that needs to get out nowadays, both in industry and academia, is that object orientation is *just another paradigm*. It has its place and has certainly helped us write better-organized code for large projects, but it is not the One True Paradigm—because there isn’t one.

So, should developers be proficient in multiple languages? Certainly, but it is even more important to master multiple paradigms, which is not necessarily the same thing: A language may or may not force a particular paradigm upon you. Java pretty much forces you to do objects, so you’ll hurt yourself if you use it to write simple procedural code or to program in the functional style, à la Lisp. Likewise, you can do objects in C, but you have to build up so much scaffolding to support it that you end up feeling like you’re rubbing a cat backward. However, languages like C++, Python, Ruby, D, and CLOS support multiple paradigms naturally.

An Invitation to Functional Programming

The most commonly used programming styles nowadays are imperative (aka, procedural), object oriented, and functional, but many others exist (e.g., declarative, logic, constraint). Imperative languages closely mirror computer internals—they implement instructions that change machine state. The object-oriented (OO) style of programming is mostly imperative programming extended with the ability to package data and related functionality as classes. Most programmers seem to live in



GETTY IMAGES

the imperative/OO world.

Functional programming, first made accessible through Lisp, is another powerful programming paradigm that is older than you might think. Lisp wasn’t really usable until about 1960, but it is based on Alonzo Church’s lambda calculus, which came almost thirty years earlier, so the functional paradigm was fairly mature before it was realized on a computer. Functional languages treat functions like they treat built-in types—they can be passed and returned as values to and from other functions, and can even be created at runtime. Some very powerful constructs come to life this way.

```
# A Python example
>>> nums = [1,2,3,4]
>>> morenums = [x+1 for x in range(4)]
>>> morenums
[1, 2, 3, 4]
```

Listing 1

Since lists are so central in functional programming, you can easily create them by placing expressions inside brackets in Haskell and Python, as shown in listing 1.

The definition of `morenums` illustrates a *list comprehension*,

```
>>> set1 = 'abc'      # A string is a special sequence holding ASCII code points
>>> set2 = [1,2,3]    # A sequence of integers
>>> [(x,y) for x in set1 for y in set2 if x != 'b' if y < 3]
[('a', 1), ('a', 2), ('c', 1), ('c', 2)]
```

Listing 2

whereby a list is created through a complex formula. As shown in listing 2, list comprehensions can traverse multiple sequences just like nested loops do and can filter out what isn't wanted.

Several support functions are important in functional programming. The `map` function applies a function to each element of a list and returns the resulting list:

```
>>> map(len, ["a", "fine", "mess"])
[1, 4, 4]
```

The `map` function also accommodates functions whose arity matches the number of subsequent lists:

```
>>> map(operator.add, [1,2,3], [4,5,6])
[5, 7, 9]
```

The `reduce` function is used to summarize a list, usually reducing it to a single value. For example, to sum the elements of a list, you could use the following expression:

```
>>> reduce(operator.add, [1,2,3])
6
```

In the case of addition, however, it is better to use Python's built-in `sum` function:

```
>>> sum([1,2,3])
6
```

The `reduce` function is handy for other operations, though. The following expression forms the product of a list:

```
>>> reduce(operator.mul, [1,2,3])
6
```

An alternate form of `reduce` takes a third argument as an initializer. This comes in handy in case the list happens to be empty:

```
>>> reduce(operator.mul, [], 1)
1
```

As shown in listing 3, without the third argument, there is no

```
>>> reduce(operator.mul, [])
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: reduce() of empty sequence with no initial value
```

Listing 3

function `f`. The call `reduce(f, s, c)` will then compute the following expression:

$$f(f(f(\dots f(f(c, x_0), x_1)\dots), x_{n-3}), x_{n-2}), x_{n-1})$$

The insight that unlocks the power of `reduce` is to notice that the first argument in each call represents the *accumulated value* up to that point (which is `c` initially). With this in mind, here's a function that checks to see if any value in a sequence of Booleans is true:

```
def any(bools):
    return reduce(operator.or_, bools, False)
```

The function call `any([False, True])`, for example, computes `or_(or_(False, False), True)`. The accumulated value at each call will be `True` if any value encountered so far is `True`. The underlined argument above is the initial value from the third argument in the original call.

Higher-Order Functions

Now consider how to write a function that forms the union of two sets. The result will have everything from one set and everything in the other set that isn't already in the first. Therefore the initial value can be one of the sets, and the accumulating function inspects each element in the other set, discarding duplicates. Listing 4 is a first attempt.

```
def union(s,t):
    def form_union(sofar,x):
        return sofar + ([x] if x in s else [])
    return reduce(form_union, t, s)
```

Listing 4

Since `form_union` is a one-liner, we can use a *lambda expression* to create the function on-the-fly, instead, as shown in listing 5.

The `lambda` keyword takes a parameter list followed by a colon and a single expression.

As listing 6 shows, writing an intersection function is also a one-liner.

This time the default is the empty set, and elements of `t` are retained only if they are also elements of `s`.

A function that takes other functions as parameters or that returns a function as a result is known as a *higher-order function*. The `map` and `reduce` functions are examples that take a func-

“A precursor to generic programming, functional programming is especially powerful in constructing succinct, flexible solutions that process sequences of data regardless of type.”

```
def union(s,t):
    return reduce(lambda sofar, x: sofar + ([] if x in s else [x]), t, s)
```

Listing 5

```
def intersection(s,t):
    return reduce(lambda sofar,x: sofar + ([x] if x in s else []), t, [])
```

Listing 6

tion as a parameter but return a computed value, not a function. In an ambitious effort to convince you of the power of functional programming (and of Python!), the next example illustrates both.

Consider how you would formulate the composition of an arbitrary number of unary functions. Given the list of functions $[f, g, h]$, we want to create a new function that, when given the argument x , computes $f(g(h(x)))$ —but for any number of functions. Since `reduce` traverses sequences left to right, we’ll need to reverse the list of functions so that h is applied first. The accumulator function should take the current accumulated result as its first parameter, the next function in the list as a second parameter, and apply the latter to the former as the new accumulated result. The return value should itself be a function that takes a single parameter (x above). All the above verbiage can be reduced to the one-line function shown in listing 7.

```
def compose(funs):
    return lambda x: reduce(lambda z,f: f(z), list(reversed(funs)), x)
```

Listing 7

```
def add1(stuff):
    return [x+1 for x in stuff]

def mult2(stuff):
    return [x*2 for x in stuff]

c = compose([add1, mult2])
print c(range(10)) # [1, 3, 5, 7, 9, 11, 13, 15, 17, 19]
```

Listing 8

If there are no functions in the list `funs`, then `compose` returns the identity function. Otherwise, it returns a series of nested invocations on the free parameter x —just what we wanted, and all in one line! An example that validates `compose` is shown in listing 8, which prints the first ten odd numbers by applying $2x+1$ to


the list $[0, 1, \dots, 9]$.

Summary

The object paradigm is just one of many that a software developer should have in his arsenal. A precursor to generic programming, functional programming is especially powerful in constructing succinct, flexible solutions that process sequences of data regardless of type. I have shown examples in Python, but many languages support functional programming. Languages—such as C++, Python, and the increasingly popular D—that support multiple styles of programming tend to get you to an acceptable software solution more directly than those that don’t. **{end}**

Chuck Allison developed software for twenty years before becoming a professor of computer science at Utah Valley State

College. He was senior editor of the C/C++ Users Journal and is founding editor of The C++ Source. Chuck is also the author of two C++ books and gives on-site training in C++, Python, and design patterns.



Which paradigms do you regularly use? Have you noticed that the language you use influences the solutions you create?

Follow the link on the StickyMinds.com homepage to join the conversation.