Functional Perl Programming

Frank Antonsen
frankantonsen@netscape.net

Functional programming (FP) is based on the definition and evaluation of functions. You do everything by evaluating functions, and you rarely use variables or typical iteration constructions such as `for` or `while` loops. This style of programming minimizes side effects, is easy to optimize and refactor, and allows the you to build up complex functions out of simpler ones.

Perl, however, is a multiparadigm programming language, meaning it does not force you to program according to a specific paradigm. It has elements of functional, procedural, object-oriented, and other sorts of designs in it. You can program any way that you like, and you can use the good parts of each style to their greatest advantage. Perl isn't a FP language since it isn't based on the definition and evaluation of functions, but you can still use some of the ideas of FP.

A common feature of FP is list processing. In fact, Lisp, the very first FP language, was invented in 1959 for precisely that purpose. It's name stands for "list processing". In academia, the most widely used functional languages are ML (for Meta Language), [3], and Haskell.

Functional languages can in general achieve a higher level of optimisation than non-pure FP languages, and this is one of the major reasons for their importance in academic circles. The syntax of these languages is easier to master than that of Lisp. One could say, that Lisp has a pure functional *syntax*, whereas ML and Haskell borrow the notation from the Pascal/Ada/Eiffel branch of languages.

FP is characterised by a number of features, but I only have room to cover some here.

- Lack of side effects
- Iteration instead of recursion
- Higher order functions
- Composition of functions

No side effects

A side-effect is something causing a change in the program's environment, such as writing to a file or the screen or modifying certain global variables. Many statements in Perl, and most other languages, have side-effects. Perl is full of side effects since so many functions set variables. For instance, the match operator can set several variables, including `` $` ``, `$&`, `$'`, and the memory match variables `$1`, `$2`, and so on. A global match in scalar context also sets the value returned by `pos().` An assignment to a variable is a side effect.

Pure functional languages are a subset of FP and have no *side-effects*. since side-effetcs have been eliminated everywhere else, the compiler can make a large number of optimisations. What makes optimisation of non-FP programs difficult is the fact that variables can have non-trivial scopes.

In this way, many side effects have been avoided—you can be assured that any variable you may declare anywhere in your program also has the same meaning or value anywhere else. This makes FP-programs "safer" than procedural ones, and is one of the main reasons why it easier to *prove* the correctness of an FP-program than a procedual one. This also accounts for the greater interest among academics for FP languages.

Recursion, not iteration

FP languages use recursion instead of iteration. Functions call themselves to repeatedly do something. Most Perl idioms use iteration instead of recursion, but you can turn an iterative solution into a recursive one.

All of Perl's iteration commands can be built up from a **while** loop. The standard **for** loop

```
for( INIT; CONDITION; INCREMENT ) {
     ... # some code
}
```

is just another way you write this **while** loop.

```
INIT;
while( CONDITION ) {
     ... # some code
     INCREMENT;
}
```

To see how you can begin to turn this into a program using recursion instead of iteration, you can rewrite it once more as

```
INIT;
LOOP: {
     ... # some code
     INCREMENT;
     redo LOOP if CONDITION;
}
```

Now, this almost has the form of a recursive subroutine, so you make the final alterations to turn it into an FP-style "loop by recursion":

```
sub loop {
     my $i = shift;
     ... # some code
     return if CONDITION;
     loop( $i+1 );
}
```

Another feature of pure FP is the lack of conditionals. You can easily get rid of the **if** with a short circuit operation. You replace

```
return if CONDITION;  # conditional return
```

with the condition as the first part of the short circuit and the return as the second part. The **return** happens only if the condition is true.

```
CONDITION && return;
```

My **loop** function becomes

```
sub loop {
     my $i = shift;
     ... # some code
     CONDITION && return;
     loop( $i+1 );
}
```

This is essentially how the loop-construct above would be written in Prolog or Miranda (except, of course, that the notation is different).

But, in fact, you can do much more. Suppose the body of the loop is written as a single function (which you can always arrange).

```
foreach $i ( LIST ) {
     body($i);
}
```

You can rewrite that using the **foreach** as a statement modifier.

```
body( $i ) foreach $i ( LIST );
```

But it could also be written even more "functionally" by using **map** to apply the function to each item in the list. This is a purely functional way of doing loops.

```
map body( $_ ), LIST;
```

Furthermore, if you think of a program as a list of statements, then you can even run the entire program by a call to **map** combined with **eval**:

```
map eval( $_ ), @program;
```

Just like in Lisp, you can consider a program as just another list, and you can run it by applying a special function to it. Indeed, that is what FP is all about.

Higher order functions

A higher order function takes another function as an argument. Perl comes with some higher order functions built-in: **map**, **grep**, **sort**, and **eval** take code blocks as arguments. These go by different names in different FP languages.

There is one more important higher order function found in most FP languages but absent in Perl (interestingly enough, it has been added to Python). This function is often known as **fold** (ML and Haskell has two versions, **foldl** and **foldr**, for "folding" from the left and right respectively), elsewhere in the FP literature, it may also be known as **`reduce`**, which is the name Python uses.

To fold a list, you take two arguments and apply a function to them. You take the result of that operation and another element from the list and do it again, and go on until you are done.

Using the **`add`** routine from earlier, you can add the list of numbers **`(1,2,3,4,5)`**. You can write this out as a series of chained function calls.

```
my $sum = add( 1, 2 );
   $sum = add( $sum, 3 );
   $sum = add( $sum, 4 );
   $sum = add( $sum, 5 );
```

You can write this without the temporary variable **`$sum`**.

```
add( add( add( add(1,2), 3 ), 4), 5 )
```

and so on for more than five elements in the list. That's too much typing though.

You can write your own **`fold`** routine in Perl, although you need a few utility functions first. You want to process a list, so you need a way to break that list into the first item, or "head", and the remaining items, or tail. You can't modify the list, so you have to work with copies of it. My **`head`** function returns the first item in `@_`, and the tail function returns the rest of the items, if any. Neither of them affect any variable or have a side effect. You add a **`len`** subroutine to return the length of a list.

```
sub head { shift }
sub tail { shift; return @_ }
sub len  { scalar @_ }

sub fold {
     my( $f, @a ) = @_;
     return head( @a )
     if len( tail( @a )  ) < 1;

     my ($a, $b) =  ( head( @a ),
                      head( tail( @a ) ) );

     fold( $f, ( $f->($a,$b),
             tail( tail( @a ) ) ) );
}
```

You can write this even more tersely by replacing the occurences of the dummy variables **`$a, $b`** by their definitions **`(head @a, head tail @a)`** in the recursive step. This would make the program harder to read, though, and, in any case, the use of dummy variables in this way is entirely consistent with FP practice.

To calculate the factorial using **`fold`** you can now define a function **`mult`** and use this as its first argument.

```
sub mult { $_[0] * $_[1] };
print "Result is ",
       fold( \&mult, 0 .. 10 ), "\n";
```

Since this is really just a utility function you are not likely to need elsewhere in the program, it is better to take advantage of another FP-construction also present in Perl: the *anonymous functions*. Whereas ordinary functions are globally defined, and hence "stick around" for the entire lifetime of the program, anonymous functions are "use once and throw away" functions.

Most FP languages introduces a special keyword for defining anonymous functions (typically called **`lambda`**), but Perl uses a much more simple, and natural approach: you simply omit the function name in the definition. Ironically, in this sense, Perl treats functions much more on an equal footing with other data structures than do most FP-languages.

Hence, you can rewrite the factorial computer with an anonymous subroutine.

```
my $n = fold( sub{ $_[0] * $_[1] }, 1..10 );
```

You can get even better by hacking on the definition of **`fold`** so you can make it work like **`grep`** which takes a block of code. The **`fold`** functionality stays the same, but you add a prototype saying that the first argument is a code reference. The perlsub man page explains prototypes.

```
sub fold(&@) { ... }
```

Once you have the prototype in place, you can make the statement even shorter. It even looks more like Perl now. Be careful that you don't have a comma after the code block. Perl will warn you about this for **grep** or **map**, but not for your own defined functions.

```
my $n = fold { $_[0] * $_[1] } 1 .. 10;
```

Composition and currying

Probably one of the easiest way of generating new functions from old ones is through *composition*. Composition turns two or more functions into a single call.

You can combine the functions **foo** and **bar** into a single function **baz**. In this example, the result of **bar** is the argument list for **foo**, but that is hidden behind **baz**.

```
sub foo { ... }
sub bar { ... }
sub baz { foo( &bar ) }
```

You could repeatedly have **foo(&bar)** statements in your code, but when you decide to change it, you have to change all of them. When you hide all that behind another function, not only is your code shorter but you have a single point of maintenance.

You can do this with anonymous functions too. The **f** function returns the square of its arguments, and the **g** function returns the value of its argument increased by one.

```
my $f = sub { $_[0] ** 2 };
my $g = sub { $_[0] + 1  };
```

If you want to square the argument plus one (given 5, for instance, square 6), you can compose the functions into one as before. The function **h** does it all in one step.

```
my $h = sub { $f->( &$g ) };
```

It is this possibility of combining functions (similar to how Unix combines processes through pipes) which can make FP very powerful. In most practical programming problems, you divide the problem into smaller sub-problems solved in functions. You build up more complex behavior by combining functions.

Another common way of creating new functions from old ones is through *partial instantiation*. A function of two variables may be turned into a function that takes one argument by fixing one of the arguments. This is also often referred to as "currying" of functions, after mathematician Haskell Curry, who has really maximised his impact on FP terminology: his first name is used for an programming language and his last name for a common technique!

Suppose you have a function to add two numbers.

```
sub add { $_[0] + $_[1] }
```

Now you want a function to just add one to a number, like the earlier anonymous function **$g**. You already have a function to add numbers, and you can reuse it by currying it. You fix the second argument at **1** by wrapping it in another

```
my $g = sub { add( 1, $_[0] ) }
```

A common perl idiom for this is a closure, which is just an anonymous subroutine that references a lexical variable that has gone out of scope. For this example, you create the anonymous sub in a scope defined by the **do** block and decide the value for partial instantiation at run time.

```
my $g = add_curry(1);
my $h = add_curry(5);
my $plus1 = $g->(3);  # 3 + 1 = 4
my $plus5 = $h->(4);  # 4 + 5 = 9

sub add_curry {
     my $offset = $_[0];
     sub { add( $offset, $_[0] ) };
}
```

Conclusion

Although Perl is not a functional programming language, it is flexible enough to let you fake it, at least partially. I couldn't cover everything, but functional programming depends on functions without side effects and uses them to build up more complex behavior. You don't have to program like this, but you can do what Perl does: steal what you find useful.