

Functional Calculation 5 : Operations

Neville Holmes

Email: Neville.Holmes@utas.edu.au

ABSTRACT

This article is the sixth in a series expounding the joys of functional calculation. Functional calculation does with operations applied to functions and numbers what numerical calculation does with functions applied to numbers. The functional notation used as the vehicle in this series is provided by a freely available calculation tool called \mathcal{J} . This article makes a start to introducing those functional calculation capabilities, in particular the use of certain operators which can be applied to functions and values to produce new functions.

FUNCTIONAL CALCULATION

The description so far has been of numerical calculation, that is, of functions which can be applied to numbers to produce other numbers, though there has been some consideration of structures of characters and boxes. Much is possible using the \mathcal{J} notation through such simple numerical calculation, because the notation provides a rich variety of primitive functions, that is, of functions that have symbols like $+$ and $>.$ and $\%:$ instead of names given by the user, names like x and foo and $Herbert$.

What remains to be described is how functional calculation can be built, uniformly and consistently, upon the numerical calculation provided by \mathcal{J} .

There are two ways in which functional expressions can be built up—by juxtaposing functions in *trains*, and by applying *operations* to functions and values. Of course the two methods may be combined. Here operations are reviewed while the use of trains is deferred.

In \mathcal{J} the definition of functions is exactly the same as the definition of results—it's simply a matter of naming. Naming is done using the $=.$ or $=:$ copulas, of which the latter is more general in providing a definition which holds globally. Thus the square root of 999 is a numerical result and is named by

$sr =: \% : 999$

while the natural log of the difference¹ is a function and is named by

$ld =: ^ . @ -$

¹Properly speaking, the *difference* is the magnitude of the subtraction, but English doesn't have a convenient unambiguous word for the result of a subtraction, so *difference* has its loose meaning here.

where the $\hat{\cdot}$ and the $-$ are functions, and the $@$ is an operation, as is about to be explained.

Operations

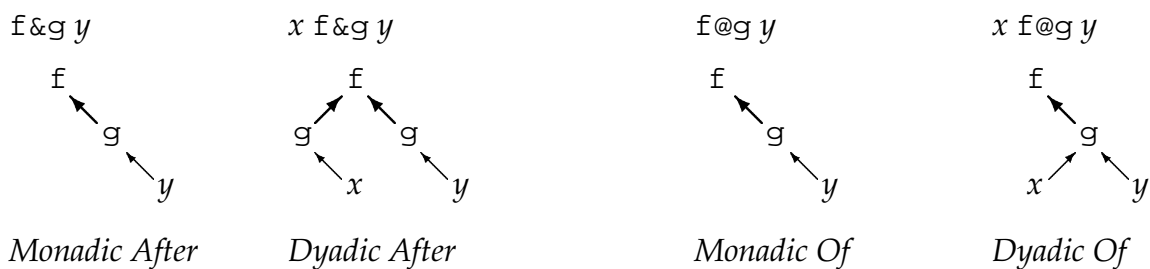
Operations are to functions what functions are to values. Primitive operations are given special symbols, and can be monadic or dyadic, but not both. In this they differ from primitive functions which can be used both monadically or dyadically.

The point about operations is that they are applied to produce functions, whereas functions are applied to produce values, that is, numbers or characters or boxes. As higher level entities, operations apply themselves to their operands more strongly than functions apply themselves to their arguments. Two primitive operations have been briefly considered in previous articles.

The symbol \sim stands for a primitive monadic operation, and its operand (the function to its left) is always applied dyadically. Thus, the argument of the function it produces when used monadically is used both as the left argument and the right argument. Otherwise, the arguments of the function it produces when used dyadically are reversed, or *commuted*, the right argument being used as the left and the left argument being used as the right.

The symbol $/$ stands for a primitive monadic operation, and the function it produces when used monadically applies its operand dyadically between all the items of its argument. So $+/$ applied monadically to a list of numbers will add them up.

There are two dyadic operations which can be used to combine functions to make new ones, as illustrated in the following diagram.



Syntactically, what holds for primitive functions also holds for composed functions. Where a primitive function can be used so also can a function produced by an operation. A function, primitive or composed, can be used monadically or dyadically, and this is independent of whether any component operation is monadic or dyadic.

Note particularly that an expression like

$$x \ f @ g \ y$$

can be keyed in directly for evaluation, but that the expression

$$x \ f \ g \ y$$

does *not* have the same meaning, even though it might give the same result.

In some of the literature, dyadic operations are called *conjunctions*, while monadic operations are called *adverbs*, by analogy with the conventional names for parts of speech in natural language. This is a dubious analogy, but the names are useful and will be adopted here.

ADVERBS

The simpler operations are the adverbs. They only have one operand, to their left; conjunctions have two. Most primitive adverbs are structural, that is, their operand is a function, and the adverb controls how the operand is applied amongst the items of the composed function's argument or arguments. The table lists the adverbs discussed in the following.

~	both	swap				
/	across	between	/ .	diagonals	sequester	
\	prefixes	infixes	\ .	suffixes	exfixes	
}	extract	amend	b .	basic		
			f .	fix	fix	

Moving Arguments

The simplest primitive adverb has ~ for its symbol. It always uses its operand as a dyadic function, but the function it produces may of course be used monadically or dyadically.

If its result, say $f \sim$, is used monadically, then the argument is used as both arguments of the operand function. The expression $f \sim x$ is equivalent to $x \ f \ x$. For example, $+ \sim x$ will double x , while $* \sim x$ will square it.

If $f \sim$ is used dyadically, then the arguments are swapped for the operand function. The expression $x \ f \sim y$ is equivalent to $y \ f \ x$. For example, $x - \sim y$ will subtract x from y , not y from x , while $x \% \sim y$ will divide x *into* y , not *by* it.

The dyadic use of this adverb is convenient to streamline thought by removing parentheses. Thus $(expression) \ f \ x$ may be rewritten $x \ f \ expression$.

Inserting Functions

The primitive adverb with symbol / is structural at a lower level, causing its operand to be inserted between the items of the argument or arguments of the function it produces.

If its result, say $f /$, is used monadically, then it is as though f is inserted between the items of the argument of $f /$ however many items there might be. In this case

$/$ is often pronounced *across* or *insert*. For example, $+/x$ will add up the items of x , $*/x$ will multiply them up, and $;/x$ will put each in a box.

If $f/$ is used dyadically, then it is as though f were inserted between the items of the left argument of $f/$ and the items of its right argument, each and every one of them at least for scalar functions. In this case $/$ is often pronounced *between* or *table*. For example, $x+/y$ will, if x and y are lists, make a table of the sums of the items of x and of y , while $x*/y$ will make a table of their products. Conveniently, $*/>:i.12$ will produce a 12×12 multiplication table, and, using functions other than $*$ as operand, other tables may be similarly produced.

The function $f/$ is often spoken of as *the f reduction* when used monadically because its effect is normally to reduce the rank of its argument, at least when f is a scalar function. The aspect of most interest here is that a list is reduced to a scalar, so that means an empty list like $i.0$ or $\$9$ must reduce to a scalar. That scalar must be the identity value for the reducing function, so that $+/i.0$ will yield 0, while $*/\$7$ will yield 1.

More complex ways of inserting functions use symbols that look like the $/$ symbol. However, in all cases that follow, the operand is applied monadically by the adverb, unlike the operand of $/$ which is applied dyadically.

- Used monadically, $f\backslash$ will apply its operand to successive prefixes of its argument. For example, $<\backslash$ will show those successive prefixes boxed, and $+/\backslash$ will yield progressive sums.
- Used dyadically, $f\backslash$ will apply its operand to successive subsequences of its right argument of the size specified by its left argument. When its left argument is negative, the subsequences are consecutive within their argument, if positive their heads are consecutive. Thus $_2+/\backslash y$ will yield the sums of distinct pairs of y giving a result with half as many items as y , but $2-\sim/\backslash y$ will yield the first differences, that is, it will subtract each item except the last from its immediately following item giving a result with one item fewer than y .
- The adverb $\.$ is just like \backslash except the subsequences *included* by \backslash are *excluded* by $\.$ so that $\#i.3$ yields 1 2 3 while $\#\i.3$ yields 3 2 1.
- Monadic $/.$ applies its operand to diagonals of its argument, while dyadic $/.$ applies its operand to subsequences of the right argument selected according to the key given by the left argument.

Other Adverbs

The *amend* adverb with symbol $\}$ behaves in a more complex way. In the first place, its operand may be a function or it may be a value. In the second place, the $\}$ adverb is closely associated with the $\{$ function, an unlikely association.

Superficially, monadic $x\}$ looks like monadic $x\&\{$ when their argument is of rank one. Thus, $4\}i.7$ yields the same as $4\{i.7$ but their behaviour diverges when more complex arguments and operands are used.

Dyadically, after $w = :x \ z\} \ y$ is carried out, $z\{w$ will yield x in simple cases. The basic idea of dyadic amendment is that, using the example just given, the operand z specifies which elements of y the elements of x are to replace.

The amend adverb is too complex to explain further here, except that, where the operand is a function, that function is applied to the overall function's argument or arguments to give a result that becomes the effective operand as already described.

A couple of housekeeping adverbs are *basic* and *fix*, spelt $b.$ and $f.$ respectively. The basic adverb produces a monadic function which, for an argument of $_1$ yields a character string showing the obverse of $b.$'s operand, for an argument of 0 a numeric list of the ranks of the operand, and for an argument of 1 a character string showing the identity function of the operand. The obverse is needed for the $\sim :$ and $\&.$ conjunctions, and can be defined by the $:.$ conjunction. The *fix* adverb yields its operand function, but redefined entirely in terms of primitives. All contained definitions are eliminated, so once a function is fixed it can no longer be changed by changing other definitions.

CONJUNCTIONS

The more complex operations are the conjunctions, more complex because they have two operands. Some primitive conjunctions are strictly compositional, their operands being functions, and the conjunction controlling how the operands are applied to the composed function's argument or arguments. Others primitive conjunctions are structural, that is, they have one operation that is a function, and one that is a value which modifies how the other operand, the function, is applied to the conjunction's argument or arguments.

Here is a table of the simpler conjunctions.

		$;$	cut	cut	$\wedge :$	power	power	
"	rank	rank	!	fit	fit	!	foreign	foreign
@	of	of				L	level	level
&	after	after	&.	dual	dual	@:	of	of
						&:	after	after

But, before discussing more general primitive conjunctions, it's useful to review one of the simplest of them, *value bonding*, because it is perhaps the most versatile.

Value Bonding

The symbol for bonding is $\&$, the ampersand. In value bonding, one of the operands is a function, and one is a value.

For example, used as a monadic function $3\&+$ will add 3 to its argument, while $\%&5$ will divide its sole argument by 5. The value operand doesn't have to be a

scalar, nor does it have to be numeric.

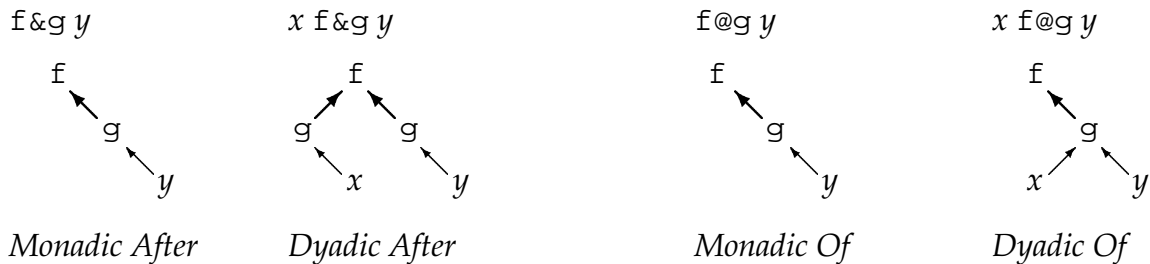
On the other hand, used as a dyadic function $a \ 3\&+$ will add 3 to its right argument a times, while $a \ \%5$ will divide its right argument by 5 a times. If a is zero, nothing will happen, but if a is negative, the inverse function will be carried out $-a$ times.

Compositions

Bonding can also be used with two functional operands. For a function composed in this way, the right operand is always used monadically, being applied to each argument when there are two, and the left operand being applied to the result or results from the right operand. Thus $\wedge.\%:y$ is the same as $\wedge.\%:y$ (the \ln of the square root of y), while $x\wedge.\%:y$ is the same as $(\wedge.x)\%:(\wedge.y)$ (in which the second pair of parentheses are included for æsthetic reasons), the $\ln x$ root of the \ln of y .

The other simple function-combining primitive conjunction uses the $@$ as its symbol, and applies its left operand to the result of its right operand, which is supplied with whatever argument or arguments the composed function is given. Thus $\wedge.@%:y$ is the same as $\wedge.\%:y$, but $x\wedge.@%:y$ is the same as $\wedge.(x\%:y)$ the \ln of the x root of y .

The following diagram shows the nature of these two most common conjunctions.



The conjunctions which use the $\&$ and $@$ are the most commonly used, but there are several related ones that are of interest.

- The *under* conjunction uses the symbol $\&$. and applies the obverse (which is usually its inverse) of its right operand to the result of the bond of its operands. This is very useful with a $>$ (unbox) right operand to allow the contents of boxes to be worked on then put back into boxes.
- For primitive functions the obverse is usually the inverse, but the $:\ .$ conjunction yields its left operand with its obverse defined as its right operand. The obverse is used by the $\wedge:$ conjunction described below, as well as by $\&.$ as just described.
- The symbol $::$ stands for the *adverse* conjunction, very like obverse but using its right operand for the replacement upon error of its left operand rather than for its obverse.

- The symbols $\cdot\cdot$ and $\cdot:$ stand for the *even* and *odd* conjunctions, though the other uses of the \cdot symbol means that usually a blank character must precede these conjunctions to make their meaning unambiguous and plain. In brief, $f \cdot\cdot g$ is $-:@(f+f&g)$ while $f \cdot:g$ is $-:@(f-f&g)$. The names describe the effect when $g = \cdot -$ since they then yield the even and odd parts of the function f .

Modifying Functions

The primitive conjunctions that modify functions typically use the value of their right operand to change the application of their functional left operand to the arguments of the function produced.

Perhaps the most interesting, and at the same time the most complex, of modifying primitive conjunctions is the *rank* conjunction, expressed by the ditto (" \cdot ") symbol. When its left operand is a function, its right operand specifies the rank to be used for the items of the modified function's argument or arguments.

Summary

This article is like a list of ingredients that can be used for combining functions using notation provided by \mathcal{J} . At their simplest, these ingredients can be used to define functions that are more complex than the primitive functions but which combine those primitive functions in a variety of ways.

The operation described here, together with those not yet considered, provide the basic means of functional programming. Yet to be considered are *trains* of various kinds, which will soon be described.

However, the next article in this series will illustrate how operations can be used with functions as a preliminary to treatment of trains.