Direct Functions in Dyalog APL

John Scholes – Dyalog Ltd.
john@dyalog.com

A Direct Function (dfn) is a new function definition style, which bridges the gap between named function expressions such as `rank←⍴∘⍴` and APL's traditional 'header' style definition.

Simple Expressions

The simplest form of dfn is: **{expr}** where **expr** is an APL expression containing `'⍺'`s and `'⍵'`s representing the left and right argument of the function respectively. For example:

```
      {(+/⍵)÷⍴⍵} 1 2 3 4          ⍝ Arithmetic mean
2.5
      2 {(⍺-⎕io)+⍳1+⍵-⍺} 5        ⍝ Sequence ⍺ ... ⍵
2 3 4 5
      {{⍵*0.5}+/⍵*2} 3 4          ⍝ Pythagorus (Note inner D-Fn).
5
```

A dfn can be used in any function context ...

```
      {⍺+÷⍵}/25⍴1                 ⍝ Repeated fraction.
1.618033989
```

... and of course, assigned a name:

```
      root←{⍵*÷⍺}                 ⍝ ⍺'th root.
```

Dfns are ambivalent. Their right (and if present, left) arguments are evaluated irrespective of whether these are subsequently referenced within the function body.

```
      2{⍵}3
3
      4{⍺}5
4
```

Guards

A guard is a boolean-single valued expression followed by `':'`. A simple expression can be preceded by a guard, and any number of guarded expressions can occur separated by `'⋄'`s.

```
     ⍺<0:                     ⍝ Left arg negative.

     0=≡⍵:                    ⍝ Right arg simple scalar.
```

Guards are evaluated in turn (left to right) until one of them yields a 1. Its corresponding **expr** is then evaluated as the result of the dfn. A guard is equivalent to an If-Then-Else or Switch-Case construct.

```
      {⍵=0:'zero' ⋄ ⍵≠0:'non zero'} 3
non zero
```

A final simple **expr** can be thought of as a default case:

```
      {2|⍵:'odd' ⋄ 'even'} 3                  ⍝ Parity.
odd
      {⍵<0:'neg' ⋄ ⍵=0:'zero' ⋄ 'pos'} 0      ⍝ Signum.
zero
```

The `'⋄'`s can be replaced with newlines. For readability, extra null phrases `'⋄ ⋄ ⋄ ...'` can be included. The parity example above becomes:

```
      {                            ⍝ Parity
          2|⍵: 'odd'               ⍝ Odd case
               'even'              ⍝ Even case
      } 4
even
```

Named dfns can be reviewed using the system editor: `)ed` or `⎕CR/⎕FX`, and note how you can comment them in the normal way using `'⍝'`.

The following example interprets a dice throw:

```
      dice←{
          ⍵≡6 6: 'Box Cars'
          ⍵≡1 1: 'Snake Eyes'
          =/⍵:   'Pair'
          7=+/⍵: 'Seven'
                 'Unlucky'
      }
```

Local Definition

The final piece of dfn syntax is the local definition. An expression whose principal function is a simple or vector assignment, introduces a name that is local to the dfn. The name is localised dynamically as the assignment is executed. As we shall see later, this mechanism can be used to define local nested dfns.

```
      mean←{                             ⍝ Arithmetic mean.
          sum←+/⍵
          num←⍴⍵

          sum÷num
      }

      roots ← {                          ⍝ Real roots of quadratic.
          a b c←⍵                        ⍝ Coefficients.
          d←(b*2)-4×a×c                  ⍝ Discriminant.

          d<0: ⍬                         ⍝ No roots
          d=0: -b÷2×a                    ⍝ One root
          d>0: (-b+¯1 1×d*0.5)÷2×a       ⍝ Two roots
      }
```

A nuance: the special syntax **⍺←expr** is executed only if the dfn is called with no left argument. This is equivalent to, but neater than: `⍎(0=⎕NC'Larg')/'Larg←...'` in a conventionally defined function.

```
        root←{                          ⍝ ⍺'th root
            ⍺←2                         ⍝ Default to sqrt.
            ⍵*÷⍺
        }

        root 64                         ⍝ Square root.
8
        3 root 64                       ⍝ Cube root.
4
```

Recursion

Dfns use recursion in place of iteration. Reference to the current function can of course use the function’s name explicitly. However because we allow unnamed dfns, we need a special symbol to represent self-reference. `'∇'` does the job.

`'∇'` has a further small advantage in that a recursive function can be renamed without having to locate and change explicit self references within the function body.

```
        fact←{                      ⍝ Factorial.
            ⍵=0: 1
            ⍵×∇ ⍵-1                                         ⍝ !
        }


        gcd←{                       ⍝ Euclid's greatest common divisor.
            ⍵=0: ⍺
            ⍵ ∇ ⍵|⍺                                         ⍝ *
        }


        fib←{                       ⍝ ⍵'th Fibonacci number.
            1≥⍵: ⍵
            +/∇¨⍵-1 2                                       ⍝ !
        }


        ack←{                       ⍝ Ackermann's function.
            ⍺=0: ⍵+1
            ⍵=0: (⍺-1)∇ 1                                   ⍝ *
                 (⍺-1)∇ ⍺ ∇ ⍵-1                             ⍝*!
        }


        osc ← {                     ⍝ Oscillate - probably returns 1.
            1=⍵: 1
            2|⍵: ∇ 1+3×⍵                                    ⍝ *
                 ∇ ⍵÷2                                      ⍝ *
        }
```

These examples show that dfns contain a minimum of syntactic baggage - allowing the underlying algorithms to ‘shine through’. For example, compare the `osc` example above with the equivalent coding using:

Control structures

```
    ∇ z←osc n
[1]    :If 1=n
[2]        z←1
[3]    :ElseIf 2|n
[4]        z←osc 1+3×n
[5]    :Else
[6]        z←osc n÷2
[7]    :End
    ∇
```

Traditional Branching

```
    ∇ z←osc n
[1]    →(1=n)/0,z←1
[2]    →(2|n)/odd
[3]    →0,z←osc n÷2
[4]   odd:z←osc 1+3×n
    ∇
```

A happy consequence of reducing syntax is that the interpreter has less work to do. Dfns typically out-perform their conventionally defined counterparts.

Tail Calls

In the previous examples, the recursive calls marked (`⍝*`) are said to be **tail calls** or **tail-recursive calls** whereas those marked (`⍝!`) are not. A function call is a tail call if its result is passed back unchanged to its own calling environment.

Tail calls can be implemented in the interpreter by re-entering the current instance of the function, rather than creating a new one. This delivers a substantial performance benefit, both in terms of execution time and workspace used.

It is often possible to turn a ‘stack-call’ into a tail call by accumulating the result on the way into, rather than on the way out of, the recursion. In general, this technique employs an **auxiliary function** that takes an **accumulating argument**. APL's infix syntax is particularly suited to this technique as we can use the left argument exclusively to accumulate result-so-far, and the right argument to specify work outstanding. Evaluation consists of moving work from the incoming right argument into the left argument of the next recursive call. Note that in these and following examples, `⎕ml` migration level is set to 3. In particular, `↑⍵` means first.

```
fact←{                      ⍝ Tail recursive Factorial.
    1 {
        ⍵=0: ⍺
        (⍺×⍵)∇ ⍵-1                                      ⍝ *
    } ⍵
}

fib←{                       ⍝ Tail recursive Fibonacci.
    0 1 {
        ⍵=0: ↑⍺
        (1↓⍺,+/⍺)∇ ⍵-1                                  ⍝ *
    } ⍵
}
```

We can simplify the coding by expressing the auxiliary function as a dyadic case of the outer monadic function:

```
    fact←{
        ⍺←1
        ⍵=0: ⍺
        (⍺×⍵)∇ ⍵-1                                      ⍝ *
    }

    sieve←{                         ⍝ Sieve of Eratosthenes.
        ⍺←⍬                         ⍝ No primes yet.
        next←1↑⍵                    ⍝ Next prime and ...
        mask←×next|⍵                ⍝ Mask of non-multiples.
        ∧/1↓mask:⍺,⍵                ⍝ No multiples, finished.
        (⍺,next)∇ mask/⍵            ⍝ Sieve out further primes.⍝ *
    }

    sieve 1↓⍳20                     ⍝ Sieve primes from 2 .. 20
2 3 5 7 11 13 17 19
```

It's easy to check if a call is a tail call by tracing it. Tail calls appear to re-use the current trace window, whereas stack-calls pop up a new one.

D-Ops: Direct Operators

The operator equivalent of a dfn refers to its left and right **operand** using `'⍺⍺'` and `'⍵⍵'` respectively. A dop is distinguished from a dfn by the presence of either of these symbols anywhere in the definition. A dop containing only `'⍺⍺'`s is monadic, whereas one that contains at least one `'⍵⍵'`, is dyadic. The Dyalog system editor distinguishes these cases and colours braces that surround an operator accordingly.

```
    else←{                          ⍝ Condition f else g ...
        ⍺: ⍺⍺ ⍵                     ⍝ True: apply left operand.
           ⍵⍵ ⍵                     ⍝ False: apply right operand.
    }

    inner←{                         ⍝ Inner product.
        ⍺⍺/⊃(↓⍺)∘.⍵⍵↓⍉⍵
    }
```

For recursive dops, there are two kinds of self reference: `'∇'` refers to the current derived function, that is, the operator bound with its operand(s). When the operands are functions, this is the most frequently used form of self reference. However, if the operands are arrays, we often need a recursive reference to the operator itself and then we would use: `'∇∇'`.

```
    pow←{                           ⍝ Explicit function power.
        ⍺=0: ⍵                      ⍝ Apply function operand ⍺ times.
        (⍺-1)∇ ⍺⍺ ⍵                 ⍝ ⍺⍺ ⍺⍺ ⍺⍺ ... ⍵             ⍝ *
    }

    until←{                         ⍝ Conditional function power.
        ⍵⍵ ⍵: ⍵                     ⍝ Until ⍵⍵,
        ∇ ⍺⍺ ⍵                      ⍝ Apply ⍺⍺ ⍺⍺ ...           ⍝ *
    }
```

```
      limit←{                     ⍝ Function power limit.
          z←αα ω                  ⍝ Apply operand until
          z≡ω: ω                  ⍝ ... it has no effect.
          ∇ z                     ⍝ Else, repeat.               ⍝ *
      }

      perv←{                      ⍝ Scalar pervasion, apply ...
          1=≡α ω: α αα ω          ⍝ ... fn between simple scalars.
                  α ∇¨ω           ⍝ ... else, traverse each arg.  ⍝ !
      }

      saw←{                       ⍝ Simple-array-wise.
          1≥≡ω: αα ω              ⍝ Simple: apply operand.
          ∇¨ω                     ⍝ Nested: visit each sub-array  ⍝ !
      }
```

Re-writing `pow` to take its function as right operand and its count as left *operand* requires that the recursion refer to the operator (`∇∇`) rather than its derived function:

```
      pow←{                       ⍝ Explicit function power.
          αα=0: ω                 ⍝ Apply function operand αα times.
          (αα-1)∇∇ ωω ωω ω        ⍝ ωω ωω ωω ... ω              ⍝ *
      }
```

Example: Pythagorean triples

The following sequence shows an example of combining Direct Functions and Operators in an attempt to find Pythagorean triples: `(3 4 5)(5 12 13)` ...

```
      sqrt←{ω*0.5}                     ⍝ Square root.

      sqrt 9 16 25
3 4 5

      hyp←{sqrt+/⊃ω*2}                 ⍝ Hypoteneuse of triangle.

      hyp(3 4)(4 5)(5 12)
5 6.403124237 13

      intg←{ω=⌊ω}                      ⍝ Whole number?

      intg 2.5 3 4.5
0 1 0

      pyth←{intg hyp ω}                ⍝ Pythagorean pair?

      pyth(3 4)(4 9)(5 12)
1 0 1

      pairs←{,⍳ω ω}                    ⍝ Pairs of numbers 1..ω.

      pairs 3
 1 1  1 2  1 3  2 1  2 2  2 3  3 1  3 2  3 3
```

```
      filter←{(⍺⍺ ⍵)/⍵}                ⍝ Op: ⍵ filtered by ⍺⍺.

      pyth filter pairs 12             ⍝ Pythagorean pairs 1..12
 3 4  4 3  5 12  6 8  8 6  9 12  12 5  12 9
```

So far, so good, but we have some duplicates: For example, `(6 8)` is just double `(3 4)`.

```
      rpm←{                            ⍝ Relatively prime?
          ⍵=0:⍺=1                      ⍝ C.f. Euclid's gcd.
          ⍵ ∇ ⍵|⍺
      }/¨                              ⍝ Note the /¨

      rpm(2 4)(3 4)(6 8)(16 27)
0 1 0 1

      rpm filter pyth filter pairs 20
 3 4  4 3  5 12  8 15  12 5  15 8
```

We can use an operator to combine the tests:

```
      and←{                            ⍝ Lazy parallel 'And'.
          mask←⍺⍺ ⍵                    ⍝ Left predicate selects...
          mask\⍵⍵ mask/⍵               ⍝ args for right predicate.
      }

      pyth and rpm filter pairs 20
 3 4  4 3  5 12  8 15  12 5  15 8
```

Better, but we still have some duplicates: `(3 4) (4 3)`.

```
      less←{</⊃⍵}
      less(3 4)(4 3)
1 0

      less and pyth and rpm filter pairs 40
 3 4  5 12  7 24  8 15  9 40  12 35  20 21
```

And finally, as promised, triples:

```
      {⍵,hyp ⍵}¨less and pyth and rpm filter pairs 40
 3 4 5  5 12 13  7 24 25  8 15 17  9 40 41  12 35 37  20 21 29
```

A larger example

`Tokens` uses nested local dfns to split an APL expression into its constituent tokens. Note that all calls on the inner functions: `lex`, `acc`, and the unnamed dfn in each token case, are *tail calls*. In fact, the *only* stack calls are those on function: `all`, and the unnamed function: `{⍵∨¯1⌽⍵}`, within the ‘Char literal’ case.

```
    tokens←{                                    ⍝ Lex of APL src line.
        alph←⎕A,⎕A̲,'_∆⍙',26↑17↓⎕AV               ⍝ Alphabet for names.
        all←{+/∧\⍺∊⍵}                           ⍝ No. of leading ⍺∊⍵.
        acc←{(⍺,↑/⍵)lex⊃↓/⍵}                    ⍝ Accumulate tokens.

        lex←{
            0=⍴⍵:⍺ ⋄ hd←↑⍵                      ⍝ Next char else finished.

            hd=' ':⍺{                           ⍝ White Space.
                size←⍵ all' '
                ⍺ acc size ⍵
            }⍵

            hd∊alph:⍺{                          ⍝ Name
                size←⍵ all alph,⎕D
                ⍺ acc size ⍵
            }⍵

            hd∊'⎕:':⍺{                          ⍝ System Name / Keyword
                size←⍵ all hd,alph
                ⍺ acc size ⍵
            }⍵

            hd='''':⍺{                          ⍝ Char literal
                size←+/∧\{⍵∨¯1⌽⍵}≠\hd=⍵
                ⍺ acc size ⍵
            }⍵

            hd∊⎕D,'¯':⍺{                        ⍝ Numeric literal
                size←⍵ all ⎕D,'.¯E'
                ⍺ acc size ⍵
            }⍵

            hd='⍝':⍺ acc(⍴⍵)⍵                   ⍝ Comment
            ⍺ acc 1 ⍵                           ⍝ Single char token.
        }

        (0⍴⊂'')lex,⍵
    }

      display tokens'xtok←size↑srce  ⍝ Next token'
.→---------------------------------------------------.
| .→---. .→. .→---. .→. .→---. .→-. .→-----------. |
| |xtok| |←| |size| |↑| |srce| |  | |⍝ Next token| |
| '----' '-' '----' '-' '----' '--' '------------' |
'∊---------------------------------------------------'
```

Summary

The syntax of a dfn/ dop is shown in the railway diagram:

```
         .-→-------→----.              ⍝ Null phrase,
         |-→--LDEF←EXPR-|              ⍝ Local definition,
         |-→-GUARD:EXPR-|              ⍝ Guarded result expression,
         |-→-----⍺←EXPR-|              ⍝ Default left argument,
-→-{-+-→-------EXPR-+-}-→-             ⍝ Unguarded result expression,
         '-←-------⋄--←-'              ⍝ ... separated by ⋄-s.
```

LDEF:	Definition local to the dfn using simple or vector assignment.
GUARD:	Expression which evaluates to a Boolean single.
EXPR:	Arbitrary APL expression.

Following special characters are substituted:

⍺:	Function Left Argument.
⍵:	Function Right Argument.
∇:	Function (and Derived Function) self reference.
⍺⍺:	Operator Left Operand.
⍵⍵:	Operator Right Operand.
∇∇:	Operator self reference.

Phrases are evaluated in turn until an unguarded EXPR, or one whose guard evaluates to 1, is encountered. The corresponding EXPR is then evaluated as the result of the dfn.

Conclusion

Dfns give us the opportunity to combine the power of expression of APL with an alternative and in some circumstances, more convenient functional framework.

They encourage (but don't compel), the programming of 'pure' functions. There is evidence to suggest that coding in such a style produces more reliable software.

Dfns provide true local functions. They encourage the use of sub-functions that may otherwise be avoided owing to a (perceived) degradation in performance or code readability. For example, it is often tempting to replace a call on a small function applied under each (¨) by embedding the code directly in the calling line and peppering it with `'¨'`s or `'∘'`s.

Using dfns in this situation improves both speed and readability. For example:

```
    ↑∘⍴∘⍴¨vec          => {↑⍴⍴⍵}¨vec          ⍝ Number of axes.
    +/¨∧\¨CVEX∊¨⊂⎕tc   => {+/∧\⍵∊⎕tc}¨CVEX    ⍝ Leading TC chars.
```

To coin a phrase: "Dfns provide a new Lo-Cal wrapper for our familiar APL expressions."

Implementation Status

These ideas were first presented in the Dyadic Vendor Forum at APL96 where they appeared to meet with general approval. Dfns were introduced with APL/W version 8.1 release 1 in early 1997. Coding examples such as those included in this article show that they have a reasonably wide application and measurements of their performance have been particularly pleasing