APL Interpreter

☠ пиздец всему фашизму ☠

M. Barney

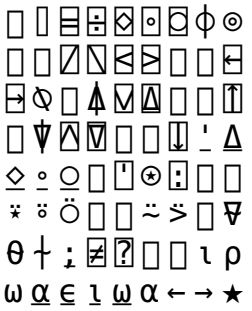
April 28, 2013

1 Introduction

This is a Haskell implementation of an APL interpreter. The paper, and code, is divided up into several sections, where I discuss issues in implementation, design decisions, and give function specifications.

2 Inputting APL

APL is famous for its non-standard character set, of which a few are:



Before beginning to write an APL parser, interpreter, etc., I had to first decide on how to even *input* APL. My eventual decision was to write an X11 xkb symbol file. For the uninitiated this is a text file for GNU/Linux Xorg distributions which allow the user to specify different keyboard input methods.

If you select a different keyboard setting via a GUI in gnome or KDE, it is modifying or referencing this file.

This was probably the most tedious part of my project — it took a substantial amount of time to look up the symbols unicode hexadecimal value; to make sure it was the correct symbol (Unicode is huge, and sometimes symbols look almost the same); and to manually type it into the keyboard configuration.

A typical line in an X11 keyboard configuration looks as follows:

```
key <AD10> { [     p,          P,     U22C6,       U235F ] };
```

In this case, the keyboard key for 'p' (and uppercase 'P' with a standard shift) is enhanced with two more possibilities via an `ISO_level3_Shift`. This key is sometimes the right alt key (also known as altgr) on many keyboards, but can be remapped to anything.

Mine is the menu key because I don't even know what that key is for anyway.

Assuming my keyboard configuration has been loaded[1], in our example above, holding down the `ISO_level3_Shift` and pressing the 'p' key will result in the symbol '⋆' (the APL function for exponentiation). Similarly, holding a regular shift and `ISO_level3_Shift` and finally the 'p' key will result in '⍟' (the APL function for logarithm).

Wasn't that fun?

3 Data Interface

In this module I introduce the APL data type; essentially a wrapper for a list datum.

$$\textbf{newtype}\ \mathit{APL}\ a = \mathit{APL}\ [a]\ \textbf{deriving}\ (\mathit{Read}, \mathit{Eq}, \mathit{Ord})$$

However, in order to better approximate the distinction that APL does *not* make between scalars and vectors, I install the newtype into a variety of different typeclasses.

I elaborate in more detail my decision to go the install-in-all-the-typeclasses route in the next section.

[1] In GNU/Linux, from the commandline, this can usually be accomplished by the command: `setxkbmap -layout us -variant apl`

3.1 The Bureaucracy

The first serious issue in implementing an APL interpreter in Haskell, and for any language, is what kind of data structure to use — and in a roundabout way, how to deal with APL scalars, and how to deal with APL lists/arrays, since no real distinctions are made between them in that venerable language.

For example, consider the following expression:

$$\alpha + \omega$$

If α and ω are scalars, then the type of "+" will be a function from scalar → scalar → scalar. But in APL, α and ω can also be vectors, and more importantly, the behavior of "+" is completely different depending on the type. In particular:

1. if the two are vectors, "+" returns a vector where each element of the first is added pointwise to the second;
2. if one is a scalar and the other a vector, then that scalar is added to each element of the vector;
3. if both are scalars we have normal addition;
4. and if both are vectors but have different lengths, we have an error.

A potential solution might be to introduce an APL data type which permits either scalars or vectors, i.e.:

```
data APL a = Scalar a | Vector [a]
```

But this doesn't prevent mixed types from occurring within a vector; i.e., we could have:

```
Scalar (Vector [1,2,3])
```

which is unusual, and undesirable.

My solution was to install the APL newtype into most of the usual type-classes; I thereby gained overloaded operators, and a blurred distinction between scalars and vectors (lists).

In effect then, an APL "scalar" would just be a list of one element, with the APL constructor acting as a wrapper. The result in my opinion is quite elegant; indeed, APL*ish* expressions are typeable (and computable) in ghci. E.g.:

```
*Main> (ϕ) $ (APL [1,2,3,4] >>= (*2)) ×: ((⍳) 4) / (fmap (○.) (4 ⍴: pi))
3.242277876554809 1.82378130556208 0.8105694691387022 0.20264236728467555
```

Furthermore, this makes the work of the parsing stage that much less — essentially it will be a one to one mapping from APL expressions to the internal Haskell versions of those functions.

```
undesirables c = (c ≢ '[') ∧ (c ≢ ']') ∧ (c ≢ ',')
putSpaces [] = []
putSpaces (']' : ',' : xs) = '\n' : putSpaces xs
putSpaces (']' : xs) = putSpaces xs
putSpaces (',' : xs) = ' ' : putSpaces xs
putSpaces (x : xs) = x : putSpaces xs
removeUndesirables = (filter undesirables) ∘ putSpaces
cheat [] = []
cheat ('.' : '0' : []) = []
cheat ('.' : '0' : x : xs) | ¬ ∘ isDigit $ x = x : cheat xs
cheat (x : xs) = x : cheat xs
aplShow (APL l) = cheat ∘ removeUndesirables ∘ show $ l
instance Show a ⇒ Show (APL a) where
  show p =      aplShow p
instance Functor APL where
  fmap f1 (APL p) = APL $ map f1 p
unwrap (APL x) = x
  -- container aware
aplZip f2 (APL p) (APL q) = APL (loop f2 p q [])
  where
    loop f2 [] (q : qs) acc      = error "Error: non equal length lists"
    loop f2 (p : ps) [] acc      = error "Error: non equal length lists"
    loop f2 [] [] acc            = reverse acc
    loop f2 (p : ps) (q : qs) acc = loop f2 ps qs ((p `f2` q) : acc)
  -- container aware
map2 :: (t → t → b) → APL t → APL t → APL b
map2 f2 (APL [p]) (APL [q]) = APL [f2 p q]
map2 f2 (APL [p]) q         = fmap (f2 p) q
map2 f2 p q                 = aplZip f2 p q
instance Num a ⇒ Num (APL a) where
```

```
  p + q      = map2 (+) p q
  p * q      = map2 (*) p q
  p - q      = map2 (-) p q
  abs p      = fmap abs p
  signum p   = fmap signum p
  fromInteger p = APL [fromInteger p]
instance Real a => Real (APL a) where
  toRational (APL p) = toRational . head $ p
instance Enum a => Enum (APL a) where
  toEnum i = APL [toEnum i]
    -- oh man this is just bad...
  fromEnum (APL (p : ps)) = fromEnum p
  succ p     = fmap succ p
instance (Integral a, Real a, Enum a) => Integral (APL a) where
  quot p q                = map2 (quot) p q
  rem p q                 = map2 (rem) p q
  div p q                 = map2 (div) p q
  mod p q                 = map2 (mod) p q
  quotRem p q             = ((quot p q), (rem p q))
  divMod p q              = ((div p q), (mod p q))
    -- truncating vectors for toInteger
  toInteger (APL [])      = error "toInteger: empty APL"
  toInteger (APL (p : ps)) = toInteger p
  -- instance (RealFrac a, Floating a) => RealFrac (APL a) where
  -- truncate (p) = fromIntegral p
instance Fractional a => Fractional (APL a) where
  p / q      = map2 (/) p q
  recip p    = fmap recip p
  fromRational p = APL [fromRational p]
instance Floating a => Floating (APL a) where
  pi         = APL [pi]
  exp p      = fmap exp p
  sqrt p     = fmap sqrt p
  log p      = fmap log p
  p ** q     = map2 (**) p q
  logBase p q = map2 (logBase) p q
```

```
  sin p      = fmap sin p
  tan p      = fmap tan p
  cos p      = fmap cos p
  asin p     = fmap asin p
  atan p     = fmap atan p
  acos p     = fmap acos p
  sinh p     = fmap sinh p
  tanh p     = fmap tanh p
  cosh p     = fmap cosh p
  asinh p    = fmap asinh p
  atanh p    = fmap atanh p
  acosh p    = fmap acosh p
concatAPL (APL p) = APL $ loop p
  where
    loop [] = []
    loop ((APL (x : xs)) : ps) =
      x : xs ++ (loop ps)
instance Monad APL where
  (APL p) >>= f = concatAPL $ fmap f (APL p)
  (APL p) >> (APL q) = APL (p >> q)
  return p       = APL [p]
instance Monoid (APL a) where
  mempty                  = APL []
  mappend (APL a) (APL b) = APL (a ++ b)
class Boolean a where
  land :: a -> a -> a
  lor  :: a -> a -> a
  lif  :: a -> a -> a
  lnot :: a -> a
  a `land` b = lnot ((lnot a) `lor` (lnot b))
  a `lor` b  = lnot ((lnot a) `land` (lnot b))
  a `lif` b  = (lnot a) `lor` b
  -- instance Foldable APL where
  -- foldMap g = mconcat . map g
  -- foldr f2 seed (APL []) = seed
  -- foldr f2 seed (APL x:xs) = f2 seed (foldr f2 seed (APL xs))
```

```
foo1 :: Num a ⇒ APL a
foo1 = APL [1, 2, 3, 4]
foo2 :: Num a ⇒ APL a
foo2 = APL [5, 6, 7, 8]
foo3 :: Fractional a ⇒ APL a
foo3 = APL [1.0, 2.0, 3.0, 4.0]
foo4 :: Fractional a ⇒ APL a
foo4 = APL [5.0, 6.0, 7.0, 8.0]
foo5 = APL [(APL [1, 2, 3, 4]), (APL [5, 6, 7, 8])]
```

4 Functions

It is probably fair to say that the interest (or notoriety) of APL lies in its functions. There are approximately 100 symbols in APL, each which can typically be interpreted either *monadically* (taking one argument), or *dyadically* (taking two arguments).

Therefore, we have approximately 200 functions to implement for what might be considered the "prelude" of APL.

I have separated the functions somewhat according to their behavior, and their "types" — in other words, whether they expect scalars or vectors, operate on vectors, and so on and so forth.

I have also partitioned the functions into their monadic and dyadic counterparts, for easier reference.

4.1 Unary Functions

```
independentFold f (x : [ ]) = x
independentFold f (x : xs) = inf_it f xs x
  where
    inf_it f (x : [ ]) acc = f x acc
    inf_it f (x : xs) acc = inf_it f xs (f x acc)
  -- + plus operator
(+.) :: Num a ⇒ APL a → APL a
(+.) ω = ω
  -- – minus operator
```

```
(-.) :: Num a ⇒ APL a → APL a
(-.) ω = fmap (*(-1)) ω
  -- × times operator
(×.) :: Num a ⇒ APL a → APL a
(×.) ω = fmap signum ω
  -- ÷ division operator
(÷.) :: Fractional b ⇒ APL b → APL b
(÷.) ω = fmap recip ω
  -- ★ power operator
( ★ ) :: Floating b ⇒ APL b → APL b
( ★ ) ω = exp ω
  -- ⌈ ceiling operator
( ⌈ ) :: (Num b, RealFrac a) ⇒ APL a → APL b
( ⌈ ) ω = fmap (fromIntegral ∘ ceiling) ω
  -- ⌊ floor operator
( ⌊ ) :: (Num b, RealFrac a) ⇒ APL a → APL b
( ⌊ ) ω = fmap (fromIntegral ∘ floor) ω
  -- ○ multiply by pi
( ○ ) :: Floating a ⇒ APL a → APL a
( ○ ) ω = pi * ω
  -- ⍟ natural logarithm
( ⍟ ) :: Floating a ⇒ APL a → APL a
( ⍟ ) ω = log ω
```

Vector based functions are as follows.

```
atIndex element (x : xs) count | x ≡ element = count
atIndex element (x : xs) count = atIndex element xs (count + 1)
iota :: (Num a, Ord a) ⇒ a → [a]
iota i = reverse ∘ loop $ i where
    loop i =
      if i ⩽ 0 then []
      else i : (loop (i - 1))
(⍳) ω = APL ∘ iota $ ω
( ⍳ ) :: (Num a, Ord a) ⇒ APL a → APL a
( ⍳ ) (APL (ω : rest)) = APL $ loop ω 1 where
  loop ω counter =
```

```
    if counter ⩾ (ω + 1) then []
    else counter : (loop ω (counter + 1))
  -- ρ rho
  -- X ←→ XρXρY
( ρ ) :: Num a ⇒ APL b → APL a
( ρ ) (APL ω) = fromIntegral (length ω)
  -- "monadic reversal..."
( ϕ ) :: APL a → APL a
( ϕ ) (APL ω) = APL ∘ reverse $ ω
(?.) :: (Num b, Random b) ⇒ b → IO b
(?.) ω = do
  r ← randomRIO (0, ω)
  return r
  -- problematic – will have type:
  -- turns something into a vector
( ∈ ) :: APL a → APL a
( ∈ ) (APL ω) = ⊥
  -- equally problematic – will have type:
  --  (^.) :: APL (APL a) -> APL a
  -- ... muurder
( , ) ω = ⊥
  -- head of vector
( ↑ ) ω = 1 ↑₂ ω
  -- tail of vector
( ↓ ) ω = 1 ↓₂ ω
```

4.2 Dyadic Functions

Basic dyadic arithmetic and trigonometric functions.

```
(+ :) :: Num a ⇒ APL a → APL a → APL a
α + : ω = α + ω
(- :) :: Num a ⇒ APL a → APL a → APL a
α - : ω = α - ω
(× :) :: Num a ⇒ APL a → APL a → APL a
```

```
α × : ω = α * ω
(÷ :) :: Fractional b ⇒ APL b → APL b → APL b
α ÷ : ω = α / ω
( ★₂ ) :: Floating b ⇒ APL b → APL b → APL b
α ★₂ ω = α ** ω
( ⌈₂ ) :: (Ord b) ⇒ APL b → APL b → APL b
α ⌈₂ ω = map2 max α ω
( ⌊₂ ) :: (Ord b) ⇒ APL b → APL b → APL b
α ⌊₂ ω = map2 min α ω
( ○₂ ) :: (Eq t, Floating a, Num t) ⇒ APL t → APL a → APL a
α ○₂ ω =
  case α of
  APL [1] → sin ω
  APL [2] → cos ω
  APL [3] → tan ω
    -- returning ω when no case match
  _ → ω
  -- caught non-commutativity bug in map2 after implementing ⊛
( ⊛₂ ) :: Floating a ⇒ APL a → APL a → APL a
α ⊛₂ ω = logBase α ω
```

Dyadic vector functions.

```
  -- container aware, will error if ⍴ ω > 1
( ⍳₂ ) :: (Eq b1, Num b, Ord b) ⇒ APL b1 → APL b1 → APL b
α@(APL ls) ⍳₂ (APL [ω]) =
  if ω ∈ ls then
    atIndex ω ls 0
  else
    1 + ((( ⌈₂ ) /* (( ⍳ ) $ ( ⍴ ) α))
  -- "The symbol ⍴ used for the dyadic function of
  -- replication..." (pg. 350, notation as thought)
  -- does not create array of shape α with data ω
  -- single dimension, container aware
( ⍴₂ ) :: (Num a, Ord a) ⇒ APL a → APL a1 → APL a1
α ⍴₂ (APL (ω: _)) = loop α ω [] where
    loop α ω acc =
```

```
        if α ⩽ 0 then
          ( ⌽ ) ∘ APL $ (reverse acc)
        else
          loop (α − 1) ω (ω : acc)
rotate :: (Num a, Ord a) ⇒ APL a1 → APL a → APL a1
rotate xs n = if n ⩾ 0 then
  (n ↓₂ xs) `mappend` (n ↑₂ xs)
  else let l = (((⍴) xs) + n) in
    (l ↑₂ xs) `mappend` (l ↓₂ xs)
  -- "dyadic rotation"
  -- 2 ⌽ ⍳ 5 ←→ 3 4 5 1 2
  -- ¯2 ⌽ ⍳ 5 ←→ 4 5 1 2 3
( ⌽₂ ) :: (Num a, Ord a) ⇒ APL a → APL a → APL a
α ⌽₂ ω = rotate ω α
(?:) :: (Num a, Random a) ⇒ Int → a → IO [a]
α ?: ω = sequence ∘ (replicate α) $ (?.) ω
roll :: (Num a, Random a) ⇒ a → IO a
roll x = (?.) x
  -- 1 for elements of α presnt in ω; 0 otherwise
( ∈₂ ) :: (Eq a, Num b) ⇒ APL a → APL a → APL b
α ∈₂ (APL ω) = fmap (λx → if x ∈ ω then 1 else 0) α
  -- , (comma) because , is reserved
( ,₂ ) :: APL a → APL a → APL a
α ,₂ ω = α `mappend` ω
  -- float based indexing — not awesome
( ↑₂ ) :: (Num a1, Ord a1) ⇒ APL a1 → APL a → APL a
α ↑₂ ω = APL $ loop α ω where
  loop n foo@(APL (p : ps)) =
    if n ⩽ 0 then [ ]
    else p : (loop (n − 1) (APL ps))
  -- ...at all
( ↓₂ ) :: (Num a1, Ord a1) ⇒ APL a1 → APL a → APL a
α ↓₂ (APL (ω : ωs)) = APL $ loop α (ω : ωs) where
  loop α (ω : ωs) =
    if α ⩽ 0 then (ω : ωs)
```

```
    else loop (α − 1) (ωs)
-- α /: ω
```

4.3 Special Functions: Operators

The special functions are operators when their left operand is a function. They are: "/" <add others>

$$f/* \; (APL\; \omega) = independentFold\; f\; \omega$$

4.4 Example and Unit Tests

$$\begin{aligned}
&foo1 :: Num\; a \Rightarrow APL\; a \\
&foo1 = APL\; [1,2,3,4] \\
&foo2 :: Num\; a \Rightarrow APL\; a \\
&foo2 = APL\; [5,6,7,8] \\
&foo3 :: Fractional\; a \Rightarrow APL\; a \\
&foo3 = APL\; [1.0,2.0,3.0,4.0] \\
&foo4 :: Fractional\; a \Rightarrow APL\; a \\
&foo4 = APL\; [5.0,6.0,7.0,8.0]
\end{aligned}$$

5 Parser and Evaluator for APL

The working parser for the interpreter is composed, essentially, of two parts: a symbol map that maps string APL functions to a tuple, containing the unary version and the dyadic version, and the scanner which tokenizes and marks the arity of functions.

During the parsing and evaluation stage, the arity of the operator is queried, and the appropriate side of the tuple is returned depending on that arity.

In this way, an evaluation stack is built up, and reduced, when moving across the token list.

```
symbolMap sym =
  case sym of
```

```
"+" → ((+.), (+ :))
"−" → ((−.), (− :))
"×" → ((×.), (× :))
"÷" → ((÷.), (÷ :))
"★" → (( ★ ), ( ★₂ ))
"⌈" → (( ⌈ ), ( ⌈₂ ))
"⌊" → (( ⌊ ), ( ⌊₂ ))
" " → (( ○ ), ( ○₂ ))
"⍟" → (( ⍟ ), ( ⍟₂ ))
"⍳" → (( ⍳ ), ( ⍳₂ ))
"⍴" → (( ⍴ ), ( ⍴₂ ))
"⍋" → (( ⍋ ), ( ⍋₂ ))
" " → (( ∈ ), ( ∈₂ ))
",," → (( , ), ( ,₂ ))
"↑" → (( ↑ ), ( ↑₂ ))
"↓" → (( ↓ ), ( ↓₂ ))
s → error ("Unknown symbol: " ++ s)
```

The token data structure has the obvious datums: a function which embeds the operator, and its arity; names for identifiers; numbers; left paren and right parens; and the minus sign, which has special precedence.

```
data TokenAPL a =
  Function String Int |
  Name String |
  Number a |
  LParen |
  RParen |
  Minus deriving (Show, Eq, Read)
```

The scanner uses a few functions to tokenize the string; it checks for whether the current object it is examining is a number, an identifier, or a symbol (neither of those); moreover, it supports vector input as one would expect, i.e.:

```
⍴ 1 2 3 4
```

The current scanner relies upon the Prelude's *words* function; a side effect is that the scanner is space sensitive for operators, which can simply be amended by writing a custom *words*.

```
isNum s = and (map isDigit s)
isIdent s = and (map isAlpha s)
isSym s = (¬ ∘ isNum $ s) ∧ (¬ ∘ isIdent $ s)
  -- eats the largest number or vector it can
obtainNumber ns = loop ns where
      loop [] = []
      loop (n : ns) | isNum n = (read n :: Double) : loop ns
      loop (n : ns) | and $ map isSpace n = loop ns
      loop (n : ns) = []
scan' [] = []
scan' ("(" : ss) = (LParen : scan' ss)
scan' (")" : ss) = (RParen : scan' ss)
scan' (s : ss) | isIdent s = ((Name s) : scan' ss)
scan' (s1 : ss) | (isSym s1) = (Function s1 1) : scan' ss
scan' t@(s : ss) | isNum s =
  let num = obtainNumber t
    ss' = (drop (length num) t)
      in
    if ss' ≡ [] then
      (Number num) : scan' ss'
    else
      let leftarg = Number num
          next = head ss'
          op   = (Function next 2) in
          leftarg : op : (scan' $ tail ss')
scan' _ = error "Unknown character in expression"
  -- need to write own words function
  -- to correctly deal with no spaces between parens, operators
scan = scan' ∘ words
```

The evaluator is *very* simple in the current version. In fact, it evaluates directly from the tokens, which is usually considered bad form. However, the structure is so simple right now, this seemed harmless, and so it wasn't necessary to construct an abstract syntaxt tree for the APL expressions.

Order of evaluation for an APL operator has "small" reach to the left (i.e., one datum to the left), and "long" reach to the right — as far as possible to the right.

As a result, one can evaluate as one goes left to right, computing the final result when one reaches the final token.

```
eval ((Number i) : []) = (APL i)
eval ((Number i) : (Function name arity) : tokens) =
  if arity ≡ 2 then
    ((snd ∘ symbolMap $ name)) (APL i) (eval tokens)
  else
    error "Arity error during parsing; expected arity 2, found 1."
eval ((Function name arity) : tokens) =
  if arity ≡ 1 then
    ((fst ∘ symbolMap $ name)) (eval tokens)
  else
    error "Arity error during parsing; expected arity 1, found 2."
eval _ = error "General unmatched parsing error during evaluation."
```

6 Main

This is the main module which starts a simple REPL loop that scans, parses and evaluates basic APL expressions.

```
simpleIgnoreLoop = do
  putStr "< >  "
  hFlush stdout
  expr ← getLine
  if (expr ≡ [] ∨ (and $ map isSpace expr)) then
    simpleIgnoreLoop
  else
    do
      let tokens = scan expr
      result ← try (evaluate (eval tokens)) ::
        IO (Either SomeException (APL Double))
      case result of
        Left errorMesg → print errorMesg
        Right answer → print answer
      simpleIgnoreLoop
main = do
  putStrLn "Atrociously Profound Language Interpreter, version 0.1"
  simpleIgnoreLoop
```