Factor Interpreter : Arithmetic

```
-------------------------------------------------------------------------------
--                    F A C T O R   I N T E R P R E T E R                    --
--                    ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~                    --
-- An interpreter for a subset of the language Factor, comprising:           --
--    integers, and the operators . + - * / % drop dup lift sink             --
-------------------------------------------------------------------------------

type Token = String

-------------------------------------------------------------------------------

type Stack = [ Integer ]

-------------------------------------------------------------------------------

-- factor fileName : interpret the Factor program in 'fileName'

factor :: String -> IO ( )

factor fileName = do
                     source <- readFile fileName
                     putStr ( "SOURCE = " ++ source )
                     putStr ( "RESULT = " ++ eval source )

-------------------------------------------------------------------------------

-- eval source : the result of interpreting the program in 'source'

eval :: String -> String

eval source = eval' ( words source ) [ ]

-------------------------------------------------------------------------------

-- eval' tokens stack : the result of interpreting the token list 'tokens'
--                      using the stack 'stack'

eval' :: [ Token ] -> Stack -> String

eval' [ ]        _     = ""

eval' ( t : ts ) stack = if isInteger t then
                            eval' ts ( read t : stack )
                         else
                         case t of
                            "."    -> let ( s1 : ss ) = stack in
                                           show s1 ++ " " ++ eval' ts ss
                            "+"    -> eval' ts ( apply'plus  stack )
                            "-"    -> eval' ts ( apply'minus stack )
                            "*"    -> eval' ts ( apply'times stack )
                            "/"    -> eval' ts ( apply'div   stack )
                            "%"    -> eval' ts ( apply'mod   stack )
                            "drop" -> eval' ts ( apply'drop  stack )
                            "dup"  -> eval' ts ( apply'dup   stack )
                            "lift" -> eval' ts ( apply'lift  stack )
                            "sink" -> eval' ts ( apply'sink  stack )

-------------------------------------------------------------------------------
```

```
-- isInteger t : does 't' represent an integer constant ?

isInteger :: Token -> Bool

isInteger ( '-' : t ) = isNonNegInteger t
isInteger t           = isNonNegInteger t

-------------------------------------------------------------------------------

-- isNonNegInteger t : does 't' represent a non-negative integer constant ?

isNonNegInteger :: Token -> Bool

isNonNegInteger t = ( t /= "" ) && ( all ( \c -> c >= '0' && c <= '9' ) t )

-------------------------------------------------------------------------------

-- apply'* stack : apply the corresponding operator to the top of stack 'stack'

apply'plus, apply'minus, apply'times, apply'div,  apply'mod,
            apply'drop,  apply'dup,   apply'lift, apply'sink
   :: Stack -> Stack

apply'plus  ( s1 : s2 : ss ) = ( s2   +   s1 ) : ss
apply'minus ( s1 : s2 : ss ) = ( s2   -   s1 ) : ss
apply'times ( s1 : s2 : ss ) = ( s2   *   s1 ) : ss
apply'div   ( s1 : s2 : ss ) = ( s2 `div` s1 ) : ss
apply'mod   ( s1 : s2 : ss ) = ( s2 `mod` s1 ) : ss

apply'drop  ( s1 : ss )      = ss
apply'dup   ( s1 : ss )      = s1 : s1 : ss

apply'lift  ( k : ss )       = lift k ss
apply'sink  ( k : ss )       = sink k ss

-------------------------------------------------------------------------------

-- lift k stack : the stack 'stack', with its 'k'th item now up on top

lift :: Integer -> Stack -> Stack

lift 1 ss          = ss
lift k ( s1 : ss ) = s' : s1 : ss' where ( s' : ss' ) = lift ( k - 1 ) ss

-------------------------------------------------------------------------------

-- sink k stack : the stack 'stack', with its top item now down in position 'k'

sink :: Integer -> Stack -> Stack

sink 1 ss              = ss
sink k ( s1 : s2 : ss ) = s2 : sink ( k - 1 ) ( s1 : ss )

-------------------------------------------------------------------------------
```

Factor Interpreter : Arithmetic

```
> factor "prog1"
SOURCE = 2 3 + 4 * .
RESULT = 20

> factor "prog2"
SOURCE = 2 3 4 * + .
RESULT = 14

> factor "prog3"
SOURCE = 7 3 / 7 3 % - .
RESULT = 1

> factor "prog4"
SOURCE = 4 3 2 1 dup 4 lift 5 sink drop . . . .
RESULT = 1 2 4 3
```