Factor Interpreter : Arithmetic and Definitions

```
-------------------------------------------------------------------------------
--                    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, definitions, invocations,                                    --
--    and the operators . + - * / % drop dup lift sink                       --
-------------------------------------------------------------------------------

type Token = String

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

type Stack = [ Integer ]

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

type Environment = [ ( Token, [ Token ] ) ]

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

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

factor :: String -> IO ( )

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

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

-- format source : the result of indenting each line of 'source',
--                 after the first one, by nine spaces

format :: String -> String

format "\n"          = "\n"
format ( '\n' : cs ) = '\n' : "         " ++ format cs
format ( c    : cs ) = c    :              format cs

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

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

eval :: String -> String

eval source = eval' ( words source ) [ ] [ ]

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

```
-- eval' tokens stack env : the result of interpreting the token list 'tokens'
--                          using the stack 'stack' and the environment 'env'

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

eval' [ ]       _     _   = ""

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

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

-- 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 )

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

-- splitDef tokens : a 3-tuple consisting of
--                      the first token in 'tokens'
--                      all remaining tokens before the first ";"
--                      all remaining tokens after  the first ";"

splitDef :: [ Token ] -> ( Token, [ Token ], [ Token ] )

splitDef ( name : ts ) = ( name, def, rest ) where ( def, rest ) = split ";" ts

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

Factor Interpreter : Arithmetic and Definitions

```
-- split token tokens : a 2-tuple consisting of
--                      all tokens in 'tokens' before the first 'token'
--                      all tokens in 'tokens' after  the first 'token'

split :: Token -> [ Token ] -> ( [ Token ], [ Token ] )

split t ( t' : ts ) = if t' == t then ( [ ], ts )
                                 else ( t' : ts1, ts2 )
                                      where ( ts1, ts2 ) = split t ts

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

-- getDef t env : the definition of 't' in 'env'

getDef :: Token -> Environment -> [ Token ]

getDef t ( ( name, def ) : es ) = if name == t then def
                                               else getDef t es

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

-- 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 "prog1"

SOURCE = : double 2 * ;

         : square dup * ;

         3 double square .

RESULT = 36


> factor "prog2"

SOURCE = : square dup * ;

         : power4 square square ;

         2 power4 .

RESULT = 16


> factor "prog3"

SOURCE = : onetwothree 1 2 3 ;

         : add3 + + ;

         onetwothree add3 .

RESULT = 6
```