# The Factor Interpreter

```
--------------------------------------------------------------------------------
--                     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, booleans, quotations, definitions, invocations,               --
--    and the operators . + - * / % = # < <= > >= drop dup lift sink if       --
--------------------------------------------------------------------------------

type Token = String

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

data Sitem = Sinteger   Integer
           | Sboolean   Bool
           | Squotation [ Token ]

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

type Stack = [ Sitem ]

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

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 ( Sinteger ( read t ) : stack ) env
                             else
                             if isBoolean t then
                                eval' ts ( Sboolean ( toBool t ) : stack ) env
                             else
                             if t == "[" then
                                let ( quot, rest ) = splitQuotation ts in
                                   eval' rest ( Squotation quot : stack ) env
                             else
                             if t == ":" then
                                let ( name, def, rest ) = splitDef ts in
                                   eval' rest stack ( ( name, def ) : env )
                             else
                             if t == "if" then
                                eval'if ts stack env
                             else
                             case t of
                                "."    -> let ( s1 : ss ) = stack in
                                             showS 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
                                "="    -> eval' ts ( apply'eq    stack ) env
                                "#"    -> eval' ts ( apply'ne    stack ) env
                                "<"    -> eval' ts ( apply'lt    stack ) env
                                "<="   -> eval' ts ( apply'le    stack ) env
                                ">"    -> eval' ts ( apply'gt    stack ) env
                                ">="   -> eval' ts ( apply'ge    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 )

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

-- isBoolean t : does 't' represent a Boolean constant ?

isBoolean :: Token -> Bool

isBoolean t = ( t == "t" ) || ( t == "f" )

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

-- toBool t : the Boolean value corresponding to 't'

toBool :: Token -> Bool

toBool "t" = True
toBool "f" = False

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

-- splitQuotation tokens : a 2-tuple consisting of
--                              all tokens in 'tokens' before
--                                 the first unmatched "]"
--                              all tokens in 'tokens' after
--                                 the first unmatched "]"

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

splitQuotation ts = let ( b1, d, a1 ) = split [ "[", "]" ] ts in
                        case d of
                           "[" -> let ( b2, a2 ) = splitQuotation a1 in
                                     let ( b3, a3 ) = splitQuotation a2 in
                                        ( b1 ++ [ "[" ] ++ b2 ++ [ "]" ] ++ b3,
                                          a3 )
                           "]" -> ( b1, a1 )

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

-- 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

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

-- split delims tokens : a 3-tuple consisting of
--                          all tokens in 'tokens' before the first occurrence
--                             of an element of 'delims'
--                          the element of 'delims' to occur first in 'tokens'
--                          all tokens in 'tokens' after  the first occurrence
--                             of an element of 'delims'

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

split ds ( t : ts ) = if elem t ds then ( [ ], t, ts )
                                   else ( t : b, d, a )
                                        where ( b, d, a ) = split ds 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

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

-- eval'if tokens stack env : the result of interpreting the token list 'tokens'
--                            using the stack 'stack' and the environment 'env',
--                            where the top of the stack holds two quotations
--                            over a Boolean value; all three items are removed,
--                            and the lower / upper quotation is applied,
--                            according as the Boolean is True / False

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

eval'if ts ( _ : Squotation qT : Sboolean True  : ss ) env =
   eval' ( qT ++ ts ) ss env

eval'if ts ( Squotation qF : _ : Sboolean False : ss ) env =
   eval' ( qF ++ ts ) ss env

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

## The Factor Interpreter

```
-- showS s : a string representation of the stack item 's'

showS :: Sitem -> String

showS ( Sinteger   n ) = show n
showS ( Sboolean   b ) = if b then "t" else "f"
showS ( Squotation q ) = show q

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

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

apply'plus, apply'minus, apply'times, apply'div,  apply'mod,
            apply'eq,    apply'ne,    apply'lt,   apply'le,  apply'gt, apply'ge,
            apply'drop,  apply'dup,   apply'lift, apply'sink
   :: Stack -> Stack

apply'plus  ( Sinteger n2 : Sinteger n1 : ss ) = Sinteger ( n1   +   n2 ) : ss
apply'minus ( Sinteger n2 : Sinteger n1 : ss ) = Sinteger ( n1   -   n2 ) : ss
apply'times ( Sinteger n2 : Sinteger n1 : ss ) = Sinteger ( n1   *   n2 ) : ss
apply'div   ( Sinteger n2 : Sinteger n1 : ss ) = Sinteger ( n1 `div` n2 ) : ss
apply'mod   ( Sinteger n2 : Sinteger n1 : ss ) = Sinteger ( n1 `mod` n2 ) : ss

apply'eq    ( Sinteger n2 : Sinteger n1 : ss ) = Sboolean ( n1   ==  n2 ) : ss
apply'ne    ( Sinteger n2 : Sinteger n1 : ss ) = Sboolean ( n1   /=  n2 ) : ss
apply'lt    ( Sinteger n2 : Sinteger n1 : ss ) = Sboolean ( n1   <   n2 ) : ss
apply'le    ( Sinteger n2 : Sinteger n1 : ss ) = Sboolean ( n1   <=  n2 ) : ss
apply'gt    ( Sinteger n2 : Sinteger n1 : ss ) = Sboolean ( n1   >   n2 ) : ss
apply'ge    ( Sinteger n2 : Sinteger n1 : ss ) = Sboolean ( n1   >=  n2 ) : ss

apply'drop  ( s : ss )                         = ss
apply'dup   ( s : ss )                         = s : s : ss
apply'lift  ( Sinteger n : ss )                = lift n ss
apply'sink  ( Sinteger n : ss )                = sink n 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 "max"

SOURCE = : max 2 lift dup 3 sink
               2 lift dup 3 sink
               > [ drop ] [ 2 lift drop ] if ;

         2 3 max .
         5 4 max .

RESULT = 3 5


> factor "factorial"

SOURCE = : ! dup 0 = [ drop 1 ] [ dup 1 - ! * ] if ;

          0 ! .
          1 ! .
          2 ! .
          3 ! .
          4 ! .
          5 ! .
         40 ! .

RESULT = 1 1 2 6 24 120 815915283247897734345611269596115894272000000000


> factor "power"

SOURCE = : ^ 1 3 sink ^' ;

         : ^' dup 0 =
              [ drop drop ]
              [ 2 lift dup 4 lift * 3 sink 2 lift 1 - ^' ]
              if ;

          1  0 ^ .
          1  1 ^ .
          1  2 ^ .
          2  0 ^ .
          2  1 ^ .
          2  2 ^ .
          2  3 ^ .
         -2  0 ^ .
         -2  1 ^ .
         -2  2 ^ .
         -2  3 ^ .
         10 10 ^ .
         13 25 ^ .

RESULT = 1 1 1 1 2 4 8 1 -2 4 -8 10000000000 705641001486681666030739693
```