Modul Expr

```
module Expr where
data Expr = Plus Expr Expr | Minus Expr Expr |
            Mul Expr Expr  |   Div Expr Expr |
            Mod Expr Expr  |   Num Integer   |
            Var Variable   |                    {-pro Eval3.hs-}
            Assign Variable Expr |              {-pro Eval4.hs-}
            Output Expr    |                    {-pro Eval5.hs-}
            Try Expr Expr                       {-pro Eval6.hs-}
           deriving (Show)

type Variable = String
type Values = [(Variable, Integer)]
```

Modul Eval3

```
import Expr
```

Přidáme ohodnocení proměnných

```
data Result x = Chyba String | Hodnota x deriving (Show)
instance Monad Result where
  Chyba s >>=  _ = Chyba s
  Hodnota a >>= f = f a
  return x = Hodnota x
  fail s = Chyba s

data Vypocet x = V {unV :: Values -> Result x}
instance Monad Vypocet where
  V vyp >>= f = V (\vals->vyp vals >>= \x->unV (f x) vals)
  return x = V (\_ -> return x)
  fail ch  = V (\_ -> fail ch)
get :: Vypocet Values
get = V (\vals -> return vals)

runVypocet :: Vypocet x -> Values -> Result x
runVypocet (V vyp) vals = vyp vals

eval::Expr->Vypocet Integer
eval (Plus e1 e2) = do  r1 <- eval e1;  r2 <- eval e2;   return (r1 + r2)
eval (Minus e1 e2) = do r1 <- eval e1;  r2 <- eval e2;   return (r1 - r2)
eval (Mul e1 e2) = do   r1 <- eval e1;  r2 <- eval e2;   return (r1 * r2)
eval (Div e1 e2) = do   r1 <- eval e1;  r2 <- eval e2;
                        if r2==0 then fail"Deleni nulou"else return(r1 `div` r2)
eval (Mod e1 e2) = do   r1 <- eval e1;  r2 <- eval e2
                        if r2==0 then fail"Deleni nulou"else return(r1 `mod` r2)
eval (Num n) = return n
eval (Var s) = do ohodnoceni <- get
                  case lookup s ohodnoceni of
                    Just x -> return x
                    Nothing -> fail ("Neznama promenna " ++ s)
```

Modul Eval4

```
import Expr
```

Přidáme změnu proměnných

```
data Result x = Chyba String | Hodnota x deriving (Show)
instance Monad Result where
  Chyba s >>=  _ = Chyba s
  Hodnota a >>= f = f a
  return x = Hodnota x
  fail s = Chyba s

data Vypocet s x = V {unV::s->(s, Result x)}
instance Monad (Vypocet s) where
  V vyp1 >>= f = V (\s->let (s', val) = vyp1 s in case val of
                                                 Chyba ch -> (s', Chyba ch)
                                                 Hodnota x->unV (f x) s')
  return x = V (\s -> (s,return x))
  fail ch =  V (\s -> (s,fail ch))
```

```haskell
get :: Vypocet s s
get = V (\s -> (s, s))

put :: Vypocet s ()
put s = V (\_ -> (s, return ()))

runVypocet :: Vypocet s x -> s -> (s, Result x)
runVypocet (V f) state = f state

eval::(Monad m, MonadRead m Values, MonadState m Values) => Expr->m Integer
eval (Plus e1 e2) = do  r1 <- eval e1;  r2 <- eval e2;   return (r1 + r2)
eval (Minus e1 e2) = do r1 <- eval e1;  r2 <- eval e2;   return (r1 - r2)
eval (Mul e1 e2) = do   r1 <- eval e1;  r2 <- eval e2;   return (r1 * r2)
eval (Div e1 e2) = do   r1 <- eval e1;  r2 <- eval e2;
                        if r2==0 then fail"Deleni nulou"else return(r1 `div` r2)
eval (Mod e1 e2) = do   r1 <- eval e1;  r2 <- eval e2
                        if r2==0 then fail"Deleni nulou"else return(r1 `mod` r2)
eval (Num n) = return n
eval (Var s) = do ohodnoceni <- get
                  case lookup s ohodnoceni of
                    Just x -> return x
                    Nothing -> fail ("Neznama promenna " ++ s)
eval (Assign s e) = do r <- eval e
                       ohodnoceni <- get
                       put (update ohodnoceni s r)
                       return r
update::Values->Variable->Integer->Values
update [] s v = [(s,v)]
update ((s1,v1):t) s v | s == s1   = (s,v) : t
                       | otherwise = (s1, v1) : update t s v
```

PreludeIO

```haskell
data IO
type FilePath = String

putChar     :: Char -> IO ()
putStr      :: String -> IO ()
putStrLn    :: String -> IO ()
print       :: Show a => a -> IO ()
getChar     :: IO Char
getLine     :: IO String
getContents:: IO String
interact    :: (String -> String) -> IO ()
readFile    :: FilePath -> IO String
writeFile   :: FilePath -> String -> IO ()
appendFile  :: FilePath -> String -> IO ()
readIO   :: Read a => String -> IO a
readIO s =  case [x | (x,t) <- reads s, ("","") <- lex t] of
              [x] -> return x
              []  -> ioError (userError "Prelude.readIO: no parse")
              _   -> ioError (userError "Prelude.readIO: ambiguous parse")
readLn :: Read a => IO a
readLn =  do l <- getLine
             r <- readIO l
             return r

-- IO exceptions
data IOError
ioError     :: IOError -> IO a
userError   :: String -> IOError
catch       :: IO a -> (IOError -> IO a) -> IO a
```

```
                         Module IO
                         ---------
data Handle = ... -- implementation-dependent
data HandlePosn = ... -- implementation-dependent
data IOMode      =  ReadMode | WriteMode | AppendMode | ReadWriteMode
                    deriving (Eq, Ord, Ix, Bounded, Enum, Read, Show)
data BufferMode  =  NoBuffering | LineBuffering |  BlockBuffering (Maybe Int)
                    deriving (Eq, Ord, Read, Show)
data SeekMode    =  AbsoluteSeek | RelativeSeek | SeekFromEnd
                    deriving (Eq, Ord, Ix, Bounded, Enum, Read, Show)
stdin, stdout, stderr :: Handle

openFile              :: FilePath -> IOMode -> IO Handle
hClose                :: Handle -> IO ()

hFileSize             :: Handle -> IO Integer
hIsEOF                :: Handle -> IO Bool
isEOF                 :: IO Bool
isEOF                 =  hIsEOF stdin

hSetBuffering         :: Handle  -> BufferMode -> IO ()
hGetBuffering         :: Handle  -> IO BufferMode
hFlush                :: Handle -> IO ()
hGetPosn              :: Handle -> IO HandlePosn
hSetPosn              :: HandlePosn -> IO ()
hSeek                 :: Handle -> SeekMode -> Integer -> IO ()

hWaitForInput         :: Handle -> Int -> IO Bool
hReady                :: Handle -> IO Bool
hReady h              =  hWaitForInput h 0
hGetChar              :: Handle -> IO Char
hGetLine              :: Handle -> IO String
hLookAhead            :: Handle -> IO Char
hGetContents          :: Handle -> IO String
hPutChar              :: Handle -> Char -> IO ()
hPutStr               :: Handle -> String -> IO ()
hPutStrLn             :: Handle -> String -> IO ()
hPrint                :: Show a => Handle -> a -> IO ()
hIsOpen, hIsClosed, hIsReadable, hIsWritable, hIsSeekable :: Handle -> IO Bool
```