

Informatics 1

Functional Programming Lectures 15 and 16

Monday 17 and Tuesday 18 November 2014

IO and Monads

Don Sannella

University of Edinburgh

Mock exam: this week!

Each student has an assigned slot

Mon 17 Nov, 3:00-5:00pm, Appleton Tower room 5.05 (mostly)

Tue 18 Nov, 2:00-4:00pm, Appleton Tower room 5.05 (mostly)

Wed 19 Nov, 2:00-4:00pm, Appleton Tower room 5.05 (mostly)

Thu 20 Nov, 2:00-4:00pm, Appleton Tower room 5.05 (mostly)

Fri 21 Nov, 3:00-5:00pm, Appleton Tower room 5.05 (mostly)

Check the course webpage for your slot

Tutorials

No tutorial this week!

No revision tutorial this week!

Final tutorial next week, usual time/place

Revision tutorial next week:

Wednesday 2–3pm in AT 5.05

Do the revision tutorial exercise and bring your solution

The 2014 Informatics 1 Competition

- First prize: A bottle of champagne or equivalent in book tokens. And glory!
- Previous year entries are online:

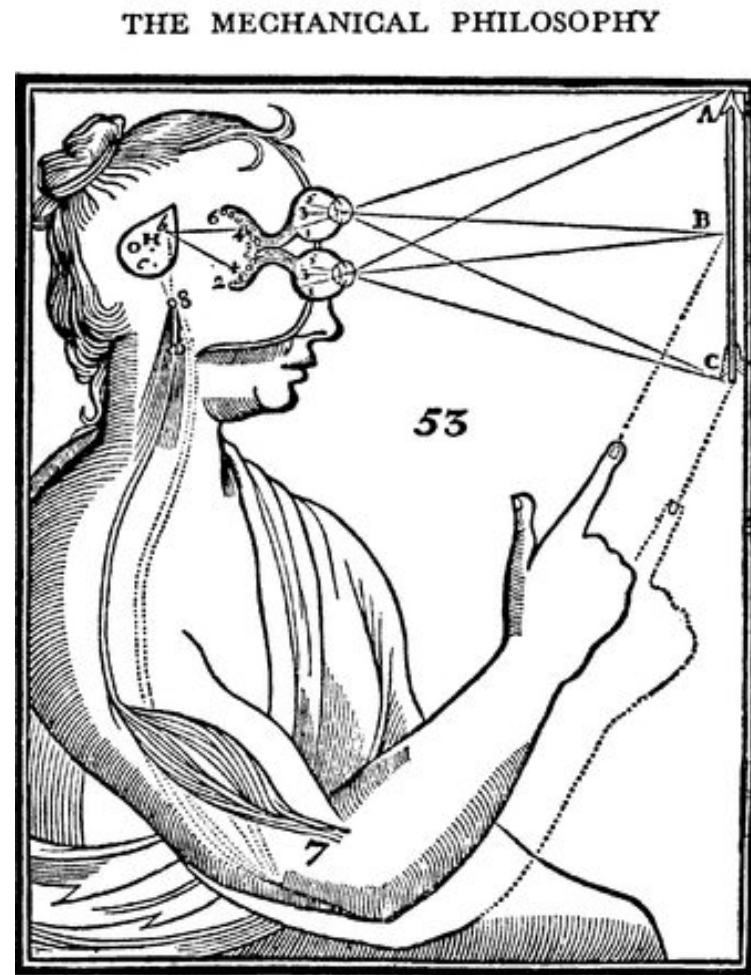
www.inf.ed.ac.uk/teaching/courses/inf1/fp/#competition

- Number of prizes depend on number/quality of entries.
- Sponsored by Galois (galois.com)
- Send code and image(s), list everyone who contributed.
- E-mail your entry to Karoliina Lehtinen
<M.K.Lehtinen@sms.ed.ac.uk>
- *Deadline: midnight, Friday 21 November 2014*

Part I

The Mind-Body Problem

The Mind-Body Problem



Part II

Commands

Print a character

```
putChar :: Char -> IO ()
```

For instance,

```
putChar '!'
```

denotes the command that, *if it is ever performed*, will print an exclamation mark.

Combine two commands

`(>>) :: IO () -> IO () -> IO ()`

For instance,

`putChar '?' >> putChar '!'`

denotes the command that, *if it is ever performed*, prints a question mark followed by an exclamation mark.

Do nothing

```
done :: IO ()
```

The term `done` doesn't actually do nothing; it just specifies the command that, *if it is ever performed*, won't do anything. (Compare thinking about doing nothing to actually doing nothing: they are distinct enterprises.)

Print a string

```
putStr :: String -> IO ()  
putStr []      = done  
putStr (x:xs)  = putChar x >> putStr xs
```

So `putStr "?!"` is equivalent to

```
putChar '?' >> (putChar '!' >> done)
```

and both of these denote a command that, *if it is ever performed*, prints a question mark followed by an exclamation mark.

Higher-order functions

More compactly, we can define `putStr` as follows.

```
putStr  :: String -> IO ()  
putStr  = foldr (>>) done . map putChar
```

The operator `>>` has identity `done` and is associative.

```
m >> done      = m  
done >> m       = m  
(m >> n) >> o  = m >> (n >> o)
```

Main

By now you may be desperate to know *how is a command ever performed?* Here is the file `Confused.hs`:

```
module Confused where
```

```
main :: IO ()
```

```
main = putStr "?!"
```

Running this program prints an indicator of perplexity:

```
[melchior]dts: runghc Confused.hs
```

```
?! [melchior]dts:
```

Thus `main` is the link from Haskell's mind to Haskell's body — the analogue of Descartes's pineal gland.

Print a string followed by a newline

```
putStrLn :: String -> IO ()  
putStrLn xs = putStr xs >> putChar '\n'
```

Here is the file ConfusedLn.hs:

```
module ConfusedLn where
```

```
main :: IO ()  
main = putStrLn "?!"
```

This prints its result more neatly:

```
[melchior]dts: runghc ConfusedLn.hs  
?!  
[melchior]dts:
```

Part III

Equational reasoning

Equational reasoning lost

In languages with side effects, this program prints “haha” as a side effect.

```
print "ha"; print "ha"
```

But this program only prints “ha” as a side effect.

```
let x = print "ha" in x; x
```

This program again prints “haha” as a side effect.

```
let f () = print "ha" in f (); f ()
```


Equational reasoning regained

In Haskell, the term

```
(1+2) * (1+2)
```

and the term

```
let x = 1+2 in x * x
```

are equivalent (and both evaluate to 9).

In Haskell, the term

```
putString "ha" >> putString "ha"
```

and the term

```
let m = putString "ha" in m >> m
```

are also entirely equivalent (and both print "haha").

Part IV

Commands with values

Read a character

Previously, we wrote `IO ()` for the type of commands that yield no value.

Here, `()` is the trivial type that contains just one value, which is also written `()`.

We write `IO Char` for the type of commands that yield a value of type `Char`.

Here is a command to read a character.

```
getChar :: IO Char
```

Performing the command `getChar` when the input contains `"abc"` yields the value `'a'` and remaining input `"bc"`.

Do nothing and return a value

More generally, we write `IO a` for commands that return a value of type `a`.

The command

```
return :: a -> IO a
```

is similar to `done`, in that it does nothing, but it also returns the given value.

Performing the command

```
return [] :: IO String
```

when the input contains `"bc"` yields the value `[]` and an unchanged input `"bc"`.

Combining commands with values

We combine command with an operator written `>>=` and pronounced “bind”.

$$(>>=) :: IO\ a \rightarrow (a \rightarrow IO\ b) \rightarrow IO\ b$$

For example, performing the command

```
getChar >>= \x -> putChar (toUpper x)
```

when the input is "abc" produces the output "A", and the remaining input is "bc".

The “bind” operator in detail

$$(>>=) :: \text{IO } a \rightarrow (a \rightarrow \text{IO } b) \rightarrow \text{IO } b$$

If

$$m :: \text{IO } a$$

is a command yielding a value of type a , and

$$k :: a \rightarrow \text{IO } b$$

is a function from a value of type a to a command yielding a value of type b , then

$$m >>= k :: \text{IO } b$$

is the command that, *if it is ever performed*, behaves as follows:

first perform command m yielding a value x of type a ;
then perform command $k \ x$ yielding a value y of type b ;
then yield the final value y .

Reading a line

Here is a program to read the input until a newline is encountered, and to return a list of the values read.

```
getLine :: IO String
getLine = getChar >>= \x ->
    if x == '\n' then
        return []
    else
        getLine >>= \xs ->
            return (x:xs)
```

For example, given the input "abc\ndef" This returns the string "abc" and the remaining input is "def".

Commands as a special case

The general operations on commands are:

```
return    :: a -> IO a
(>>=)     :: IO a -> (a -> IO b) -> IO b
```

The command `done` is a special case of `return`,
and the operator `>>` is a special case of `>>=`.

```
done      :: IO ()
done      =  return ()

(>>)      :: IO () -> IO () -> IO ()
m >> n    =  m >>= \() -> n
```


An analogue of “let”

Although it may seem odd at first sight, this combinator is reassuringly similar to the familiar Haskell “let” expression. Here is a type rule for “let”.

$$\frac{\begin{array}{l} E \vdash m :: a \\ E, x :: a \vdash n :: b \end{array}}{E \vdash \mathbf{let} \ x = m \ \mathbf{in} \ n :: b}$$

Typically, “bind” is combined with lambda expressions in a way that resembles “let” expressions. Here is the corresponding type rule.

$$\frac{\begin{array}{l} E \vdash m :: IO \ a \\ E, x :: a \vdash n :: IO \ b \end{array}}{E \vdash m \gg= \ \backslash x \ -> \ n :: IO \ b}$$

Echoing input to output

This program echoes its input to its output, putting everything in upper case, until an empty line is entered.

```
echo :: IO ()
echo =  getLine >>= \line ->
        if line == "" then
            return ()
        else
            putStrLn (map toUpper line) >>
            echo

main :: IO ()
main =  echo
```

Testing it out

```
[melchior]dts: runghc Echo.hs
```

```
One line
```

```
ONE LINE
```

```
And, another line!
```

```
AND, ANOTHER LINE!
```

```
[melchior]dts:
```

Part V

“Do” notation

Reading a line in “do” notation

```
getLine :: IO String
getLine =  getChar >>= \x ->
           if x == '\n' then
             return []
           else
             getLine >>= \xs ->
             return (x:xs)
```

is equivalent to

```
getLine :: IO String
getLine = do {
    x <- getChar;
    if x == '\n' then
      return []
    else do {
      xs <- getLine;
      return (x:xs)
    }
}
```

Echoing in “do” notation

```
echo :: IO ()
echo =  getLine >>= \line ->
        if line == "" then
            return ()
        else
            putStrLn (map toUpper line) >>
            echo
```

is equivalent to

```
echo :: IO ()
echo = do {
    line <- getLine;
    if line == "" then
        return ()
    else do {
        putStrLn (map toUpper line);
        echo
    }
}
```

“Do” notation in general

Each line `x <- e; ...` becomes `e >>= \x -> ...`

Each line `e; ...` becomes `e >> ...`

For example,

```
do { x1 <- e1;  
      x2 <- e2;  
      e3;  
      x4 <- e4;  
      e5;  
      e6 }
```

is equivalent to

```
e1 >>= \x1 ->  
e2 >>= \x2 ->  
e3 >>  
e4 >>= \x4 ->  
e5 >>  
e6
```

Part VI

Monads

Monoids

A *monoid* is a pair of an operator (\oplus) and a value u , where the operator has the value as identity and is associative.

$$\begin{aligned}u \oplus x &= x \\x \oplus u &= x \\(x \oplus y) \oplus z &= x \oplus (y \oplus z)\end{aligned}$$

Examples of monoids:

$(+)$ and 0

$(*)$ and 1

$(||)$ and `False`

$(\&\&)$ and `True`

$(++)$ and `[]`

$(>>)$ and `done`

Monads

We know that `(>>)` and `done` satisfy the laws of a *monoid*.

$$\begin{aligned}\text{done} >> m &= m \\ m >> \text{done} &= m \\ (m >> n) >> o &= m >> (n >> o)\end{aligned}$$

Similarly, `(>>=)` and `return` satisfy the laws of a *monad*.

$$\begin{aligned}\text{return } v >>= \backslash x \rightarrow m &= m[x := v] \\ m >>= \backslash x \rightarrow \text{return } x &= m \\ (m >>= \backslash x \rightarrow n) >>= \backslash y \rightarrow o &= m >>= \backslash x \rightarrow (n >>= \backslash y \rightarrow o)\end{aligned}$$

Laws of Let

We know that $(>>)$ and `done` satisfy the laws of a *monoid*.

$$\begin{aligned}\text{done} >> m &= m \\ m >> \text{done} &= m \\ (m >> n) >> o &= m >> (n >> o)\end{aligned}$$

Similarly, $(>>=)$ and `return` satisfy the laws of a *monad*.

$$\begin{aligned}\text{return } v >>= \lambda x \rightarrow m &= m[x:=v] \\ m >>= \lambda x \rightarrow \text{return } x &= m \\ (m >>= \lambda x \rightarrow n) >>= \lambda y \rightarrow o &= m >>= \lambda x \rightarrow (n >>= \lambda y \rightarrow o)\end{aligned}$$

The three monad laws have analogues in “let” notation.

$$\begin{aligned}\text{let } x = v \text{ in } m &= m[x:=v] \\ \text{let } x = m \text{ in } x &= m \\ \text{let } y = (\text{let } x = m \text{ in } n) \text{ in } o &= \text{let } x = m \text{ in } (\text{let } y = n \text{ in } o)\end{aligned}$$

“Let” in languages with and without effects

$$\begin{aligned}\text{let } x = v \text{ in } m &= m[x:=v] \\ \text{let } x = m \text{ in } x &= m \\ \text{let } y = (\text{let } x = m \text{ in } n) \text{ in } o \\ &= \text{let } x = m \text{ in } (\text{let } y = n \text{ in } o)\end{aligned}$$

These laws hold even in languages with side effects. For the first law to be true, v must be not an arbitrary term but a *value*, such as a constant or a variable (but not a function application). A value immediately evaluates to itself, hence it can have no side effects.

While in such languages one only has the above three laws for “let”, in Haskell one has a much stronger law, where one may replace a variable by any term, rather than by any value.

$$\text{let } x = n \text{ in } m = m[x:=n]$$

Part VII

Roll your own monad—IO

The Monad type class

```
class Monad m where  
  return :: a -> m a  
  (>>=)  :: m a -> (a -> m b) -> m b
```

My own IO monad (1)

```
module MyIO (MyIO, myPutChar, myGetChar, convert) where

type Input = String
type Remainder = String
type Output = String

data MyIO a = MyIO (Input -> (a, Remainder, Output))

apply :: MyIO a -> Input -> (a, Remainder, Output)
apply (MyIO f) inp = f inp
```

Note that the type `MyIO` is abstract. The only operations on it are the monad operations, `myPutChar`, `myGetChar`, and `convert`. The operation `apply` is not exported from the module.

My own IO monad (2)

```
myPutChar :: Char -> MyIO ()
myPutChar c = MyIO (\inp -> ((), inp, [c]))

myGetChar :: MyIO Char
myGetChar = MyIO (\(ch:rem) -> (ch, rem, ""))
```

For example,

```
apply myGetChar "abc" == ('a', "bc", "")
apply myGetChar "bc"  == ('b', "c",  "")
apply (myPutChar 'A') "def" == ((), "def", "A")
apply (myPutChar 'B') "def" == ((), "def", "B")
```


My own IO monad (3)

```
instance Monad MyIO where
  return x  = MyIO (\inp -> (x, inp, ""))
  m >>= k   = MyIO (\inp ->
                    let (x, rem1, out1) = apply m inp in
                    let (y, rem2, out2) = apply (k x) rem1 in
                    (y, rem2, out1++out2))
```

For example

```
apply
  (myGetChar >>= \x -> myGetChar >>= \y -> return [x,y])
  "abc"
==  ("ab", "c", "")
```

```
apply
  (myPutChar 'A' >> myPutChar 'B')
  "def"
==  ((), "def", "AB")
```

```
apply
  (myGetChar >>= \x -> myPutChar (toUpper x))
  "abc"
==  ((), "bc", "A")
```

My own IO monad (4)

```
convert :: MyIO () -> IO ()
convert m = interact (\inp ->
    let (x, rem, out) = apply m inp in
    out)
```

Here

```
interact :: (String -> String) -> IO ()
```

is part of the standard prelude. The entire input is converted to a string (lazily) and passed to the function, and the result from the function is printed as output (also lazily).

Using my own IO monad (1)

```
module MyEcho where
```

```
import Char
```

```
import MyIO
```

```
myPutStr :: String -> MyIO ()
```

```
myPutStr = foldr (>>) (return ()) . map myPutChar
```

```
myPutStrLn :: String -> MyIO ()
```

```
myPutStrLn s = myPutStr s >> myPutChar '\n'
```

Using my own IO monad (2)

```
myGetLine :: MyIO String
myGetLine = myGetChar >>= \x ->
    if x == '\n' then
        return []
    else
        myGetLine >>= \xs ->
            return (x:xs)
```

```
myEcho :: MyIO ()
myEcho = myGetLine >>= \line ->
    if line == "" then
        return ()
    else
        myPutStrLn (map toUpper line) >>
            myEcho
```

```
main :: IO ()
main = convert myEcho
```

Trying it out

```
[melchior]dts: runghc MyEcho
```

```
This is a test.
```

```
THIS IS A TEST.
```

```
It is only a test.
```

```
IT IS ONLY A TEST.
```

```
Were this a real emergency, you'd be dead now.
```

```
WERE THIS A REAL EMERGENCY, YOU'D BE DEAD NOW.
```

```
[melchior]dts:
```

You can use “do” notation, too

```
myGetLine :: MyIO String
myGetLine = do {
    x <- myGetChar;
    if x == '\n' then
        return []
    else do {
        xs <- myGetLine;
        return (x:xs)
    }
}
```

```
myEcho :: MyIO ()
myEcho = do {
    line <- myGetLine;
    if line == "" then
        return ()
    else do {
        myPutStrLn (map toUpper line);
        myEcho
    }
}
```

Part VIII

The monad of lists

The monad of lists

In the standard prelude:

```
class Monad m where
  return :: a -> m a
  (>>=)  :: m a -> (a -> m b) -> m b

instance Monad [] where

  return      :: a -> [a]
  return x    =  [ x ]

  (>>=)       :: [a] -> (a -> [b]) -> [b]
  m >>= k     =  [ y | x <- m, y <- k x ]
```

Equivalently, we can define:

```
[ ] >>= k      =  [ ]
(x:xs) >>= k   =  (k x) ++ (xs >>= k)
```

or

```
m >>= k  =  concat (map k m)
```


‘Do’ notation and the monad of lists

```
pairs :: Int -> [(Int, Int)]
pairs n = [ (i,j) | i <- [1..n], j <- [(i+1)..n] ]
```

is equivalent to

```
pairs' :: Int -> [(Int, Int)]
pairs' n = do {
    i <- [1..n];
    j <- [(i+1)..n];
    return (i,j)
}
```

For example,

```
[melchior]dts: ghci Pairs
GHCi, version 6.10.4: http://www.haskell.org/ghc/ :? for help
Pairs> pairs 4
[(1,2), (1,3), (1,4), (2,3), (2,4), (3,4)]
Pairs> pairs' 4
[(1,2), (1,3), (1,4), (2,3), (2,4), (3,4)]
```

Monads with plus

In the standard prelude:

```
class Monad m => MonadPlus m where
```

```
  mzero :: m a
```

```
  mplus :: m a -> m a -> m a
```

```
instance MonadPlus [] where
```

```
  mzero  :: [a]
```

```
  mzero  = []
```

```
  mplus  :: [a] -> [a] -> [a]
```

```
  mplus  = (++)
```

```
guard :: MonadPlus m => Bool -> m ()
```

```
guard False = mzero
```

```
guard True  = return ()
```

```
msum :: MonadPlus m => [m a] -> m a
```

```
msum = foldr mplus mzero
```

Using guards

```
pairs'' :: Int -> [(Int, Int)]  
pairs'' n = [ (i,j) | i <- [1..n], j <- [1..n], i < j ]
```

is equivalent to

```
pairs''' :: Int -> [(Int, Int)]  
pairs''' n = do {  
    i <- [1..n];  
    j <- [1..n];  
    guard (i < j);  
    return (i,j)  
}
```

For example,

```
[melchior]dts: ghci Pairs  
GHCi, version 6.10.4: http://www.haskell.org/ghc/ :? for help  
Pairs> pairs'' 4  
[(1,2), (1,3), (1,4), (2,3), (2,4), (3,4)]  
Pairs> pairs''' 4  
[(1,2), (1,3), (1,4), (2,3), (2,4), (3,4)]
```

Part IX

Parsers

Parser type

First attempt:

```
type Parser a = String -> a
```

Second attempt:

```
type Parser a = String -> (a, String)
```

Third attempt:

```
type Parser a = String -> [(a, String)]
```

*A parser for things
is a function from strings
to lists of pairs
Of things and strings*

—Graham Hutton

Module Parser

```
module Parser(Parser, apply, parse, char, spot, token,  
    star, plus, parseInt) where
```

```
import Char
```

```
import Monad
```

```
-- The type of parsers
```

```
data Parser a = Parser (String -> [(a, String)])
```

```
-- Apply a parser
```

```
apply :: Parser a -> String -> [(a, String)]
```

```
apply (Parser f) s = f s
```

```
-- Return parsed value, assuming at least one successful parse
```

```
parse :: Parser a -> String -> a
```

```
parse m s = head [ x | (x,t) <- apply m s, t == "" ]
```


Parser is a Monad with Plus

```
-- Some monads have additional structure

--   class MonadPlus m where
--       mzero :: m a
--       mplus :: m a -> m a -> m a

instance MonadPlus Parser where
    mzero      = Parser (\s -> [])
    mplus m n  = Parser (\s -> apply m s ++ apply n s)
```


Parsing characters

```
-- Parse a single character
char :: Parser Char
char = Parser f
  where
    f []      = []
    f (c:s)   = [(c,s)]

-- Parse a character satisfying a predicate (e.g., isDigit)
spot :: (Char -> Bool) -> Parser Char
spot p = Parser f
  where
    f [] = []
    f (c:s) | p c = [(c, s)]
              | otherwise = []

-- Parse a given character
token :: Char -> Parser Char
token c = spot (== c)
```

Parsing characters

```
-- Parse a single character
char :: Parser Char
char = Parser f
  where
    f []      = []
    f (c:s)   = [(c,s)]

-- Parse a character satisfying a predicate (e.g., isDigit)
spot :: (Char -> Bool) -> Parser Char
spot p = do { c <- char; guard (p c); return c }

-- Parse a given character
token :: Char -> Parser Char
token c = spot (== c)
```

Parsing a string

```
match :: String -> Parser String
match []      = return []
match (x:xs)  = do
    y <- token x;
    ys <- match xs;
    return (y:ys)
```

Parsing a sequence

```
-- match zero or more occurrences
star :: Parser a -> Parser [a]
star p = plus p `mplus` return []
```

```
-- match one or more occurrences
plus :: Parser a -> Parser [a]
plus p = do { x <- p;
              xs <- star p;
              return (x:xs) }
```

Parsing an integer

```
-- match a natural number
parseNat :: Parser Int
parseNat = do { s <- plus (spot isDigit);
               return (read s) }

-- match a negative number
parseNeg :: Parser Int
parseNeg = do { token '-';
               n <- parseNat
               return (-n) }

-- match an integer
parseInt :: Parser Int
parseInt = parseNat `mplus` parseNeg
```

Module Exp

```
module Exp where
```

```
import Monad
```

```
import Parser
```

```
data Exp = Lit Int
```

```
        | Exp :+: Exp
```

```
        | Exp **: Exp
```

```
        deriving (Eq, Show)
```

```
evalExp :: Exp -> Int
```

```
evalExp (Lit n)      = n
```

```
evalExp (e :+: f)    = evalExp e + evalExp f
```

```
evalExp (e **: f)    = evalExp e * evalExp f
```

Parsing an expression

```
parseExp :: Parser Exp
parseExp = parseLit `mplus` parseAdd `mplus` parseMul
  where
    parseLit = do { n <- parseInt;
                    return (Lit n) }
    parseAdd = do { token '(';
                    d <- parseExp;
                    token '+';
                    e <- parseExp;
                    token ')';
                    return (d :+: e) }
    parseMul = do { token '(';
                    d <- parseExp;
                    token '*';
                    e <- parseExp;
                    token ')';
                    return (d **: e) }
```

Testing the parser

```
[melchior]dts: ghci Exp.hs
GHCi, version 6.10.4: http://www.haskell.org/ghc/ :? for help
[1 of 2] Compiling Parser          ( Parser.hs, interpreted )
[2 of 2] Compiling Exp            ( Exp.hs, interpreted )
Ok, modules loaded: Parser, Exp.
*Exp> parse parseExp "(1+(2*3))"
Lit 1 :+: (Lit 2 *: Lit 3)
*Exp> evalExp (parse parseExp "(1+(2*3))")
7
*Exp> parse parseExp "((1+2)*3)"
(Lit 1 :+: Lit 2) *: Lit 3
*Exp> evalExp (parse parseExp "((1+2)*3)")
9
*Exp>
```