INTRODUCTION TO

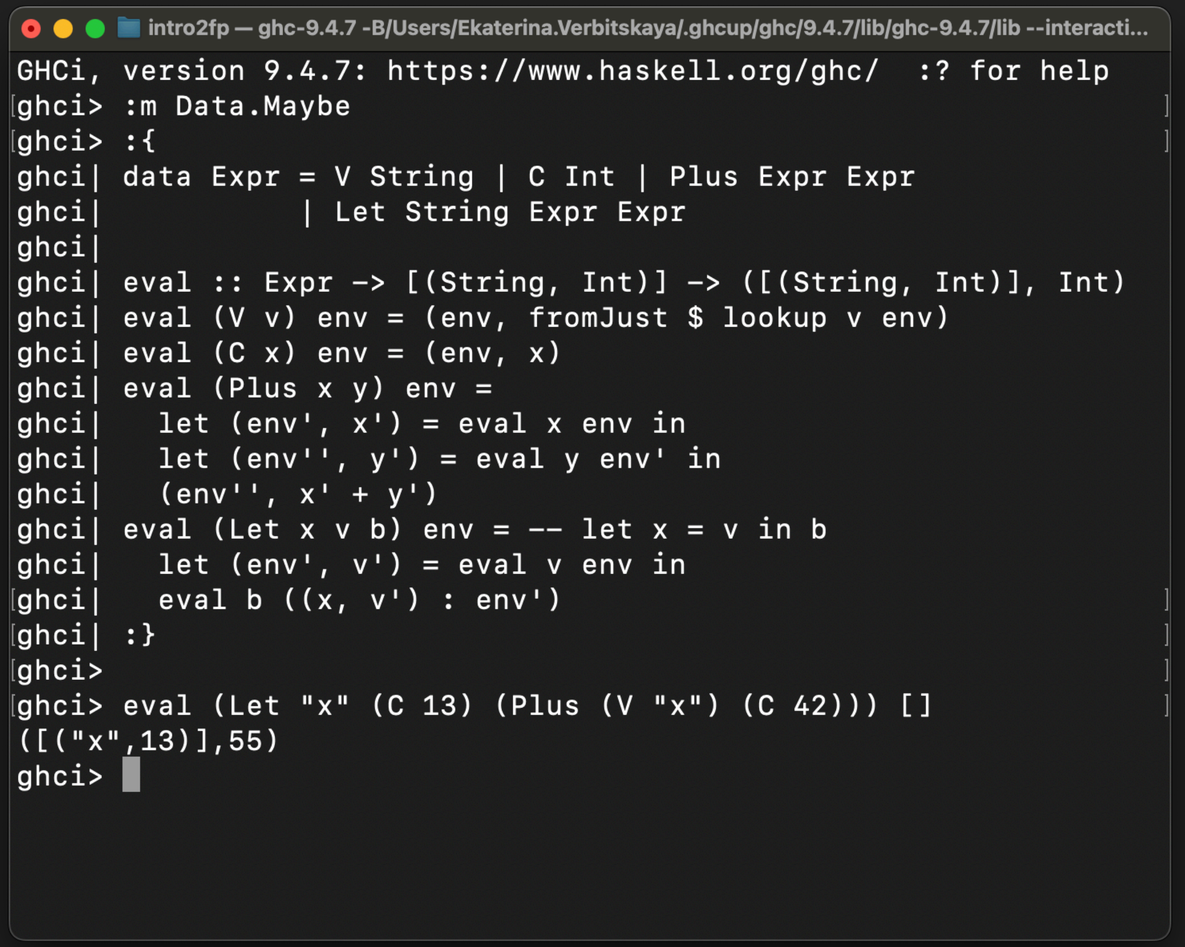
FUNCTIONAL PROGRAMMING

OUTLINE

- State Monad
- Crash Course on Parsing
- Monadic Parsers

EFFECTS: STATE

- How do we combine reading and writing?
- f :: a -> b – pure function (no effects)
- f :: a -> s -> (s, b) – function can modify state
  - s -> _ is for reading
  - (s, _) is for writing



IMPLEMENTATION

src — vim StateDemo.hs — 65×25

```haskell
newtype State s a = State { runState :: s -> (s, a) }

instance Functor (State s) where
    fmap :: (a -> b) -> State s a -> State s b
    fmap f x = State $ \s ->
      let (s', y) = runState x s in
      (s', f y)

instance Monad (State s) where
    (>>=) :: State s a -> (a -> State s b) -> State s b
    m >>= k = State $ \s ->
      let (s', x) = runState m s in
      runState (k x) s'

instance Applicative (State s) where
    pure :: a -> State s a
    pure x = State $ \s -> (s, x)

    (<*>) :: State s (a -> b) -> State s a -> State s b
    f <*> x = State $ \s ->
      let (s', f') = runState f s in
      let (s'', x') = runState x s' in
      (s'', f' x')
```

src — vim StateDemo.hs — 65×25

```haskell
execState :: State s a -> s -> a
execState m x = snd (runState m x)

get :: State s s
get = State $ \s -> (s, s)

put :: s -> State s ()
put s = State $ \_ -> (s, ())

modify :: (s -> s) -> State s ()
modify f = do
  s <- get
  put (f s)
```

EXAMPLES

src — vim StateDemo.hs — 65×25

```haskell
execState :: State s a -> s -> a
execState m x = snd (runState m x)

get :: State s s
get = State $ \s -> (s, s)

put :: s -> State s ()
put s = State $ \_ -> (s, ())

modify :: (s -> s) -> State s ()
modify f = do
  s <- get
  put (f s)
```

src — vim StateDemo.hs — 65×25

```haskell
increment :: State Int Int
increment = do
  n <- get
  put (n+1)
  return n

-- replicateM :: Applicative m => Int -> m a -> m [a]
-- replicateM n act performs the action act n times, and then ret
urns the list of results:
efficientFib :: Int -> Int
efficientFib n =
    last $ execState (replicateM n step) (0,1)
  where
    step = do
      (a, b) <- get
      put (b, a+b)
      return b
```

EXPRESSIONS

intro2fp — ghc-9.4.7 -B/Users/Ekaterina.Verbitskaya/.ghcup/ghc/9.4.7/lib/ghc-9.4.7/lib --interacti...

```
GHCi, version 9.4.7: https://www.haskell.org/ghc/  :? for help
ghci> :m Data.Maybe
ghci> :{
ghci| data Expr = V String | C Int | Plus Expr Expr
ghci|           | Let String Expr Expr
ghci|
ghci| eval :: Expr -> [(String, Int)] -> ([(String, Int)], Int)
ghci| eval (V v) env = (env, fromJust $ lookup v env)
ghci| eval (C x) env = (env, x)
ghci| eval (Plus x y) env =
ghci|   let (env', x') = eval x env in
ghci|   let (env'', y') = eval y env' in
ghci|   (env'', x' + y')
ghci| eval (Let x v b) env = -- let x = v in b
ghci|   let (env', v') = eval v env in
ghci|   eval b ((x, v') : env')
ghci| :}
ghci>
ghci> eval (Let "x" (C 13) (Plus (V "x") (C 42))) []
([("x",13)],55)
ghci>
```

src — vim Expr.hs — 65×25

```haskell
data Expr = V String | C Int | Plus Expr Expr
          | Let String Expr Expr

type ExprState = [(String, Int)]

eval :: Expr -> State ExprState Int
eval (V v) = do
  env <- get
  return $ fromJust $ lookup v env
eval (C x) = return x
eval (Plus x y) = do
  x <- eval x
  y <- eval y
  return $ x + y
eval (Let x v b) = do
  v <- eval v
  modify ((x, v) :)
  eval b

-- runEval (Let "x" (C 13) (Plus (V "x") (C 42)))
runEval :: Expr -> Int
runEval expr =
  execState (eval expr) []
```

EXERCISE

- Implement a stateful generator of pseudo-random numbers
  - Use the series:
    $x_n = (6364136223846793005 * x_{n-1} + 1442695040888963407) \bmod 2^{64}$
  - API:
    - type Random a = State Int a
    - fresh :: Random Int
    - runPRNG :: Random a -> Int -> a

OUTLINE

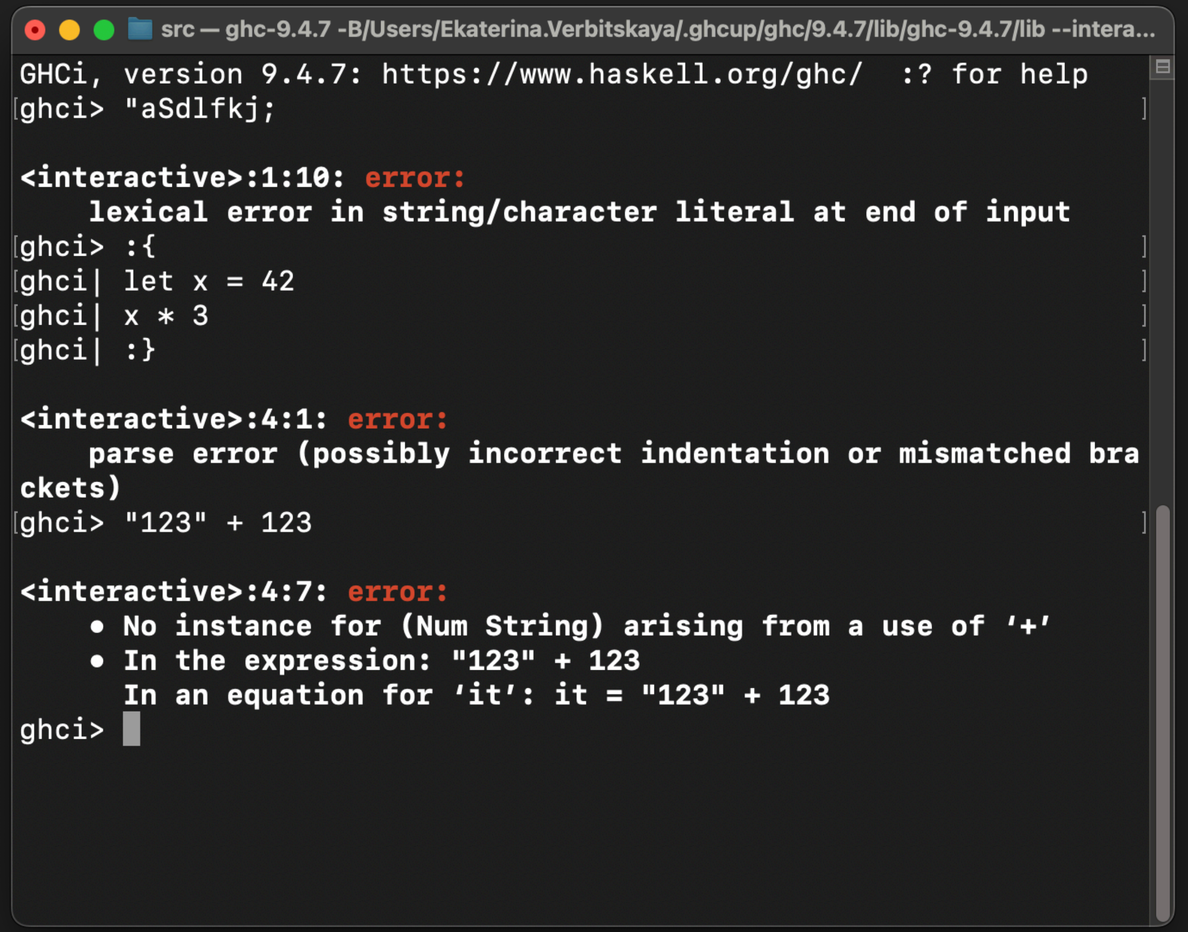
- State Monad
- Crash Course on Parsing
- Monadic Parsers

COMPUTERS ARE DUMB

- Computers have no idea:
  - What is a word, a number, a string
    - x123 vs 123 vs “123”
  - How a program is formed from words and numbers
    - x123 = “123” + 123
  - Where parentheses go in an expression
    - 1+2*3 is 1+(2*3) and not (1+2)*3

PARSERS DO INITIAL ANALYSIS

- **Lexer** figures out what are words, symbols…
- **Parser** checks grammatical correctness
- Parser creates a **data structure** that represents a program
- Then, **static analyses** are run and **compilation/interpretation** is done over that structure

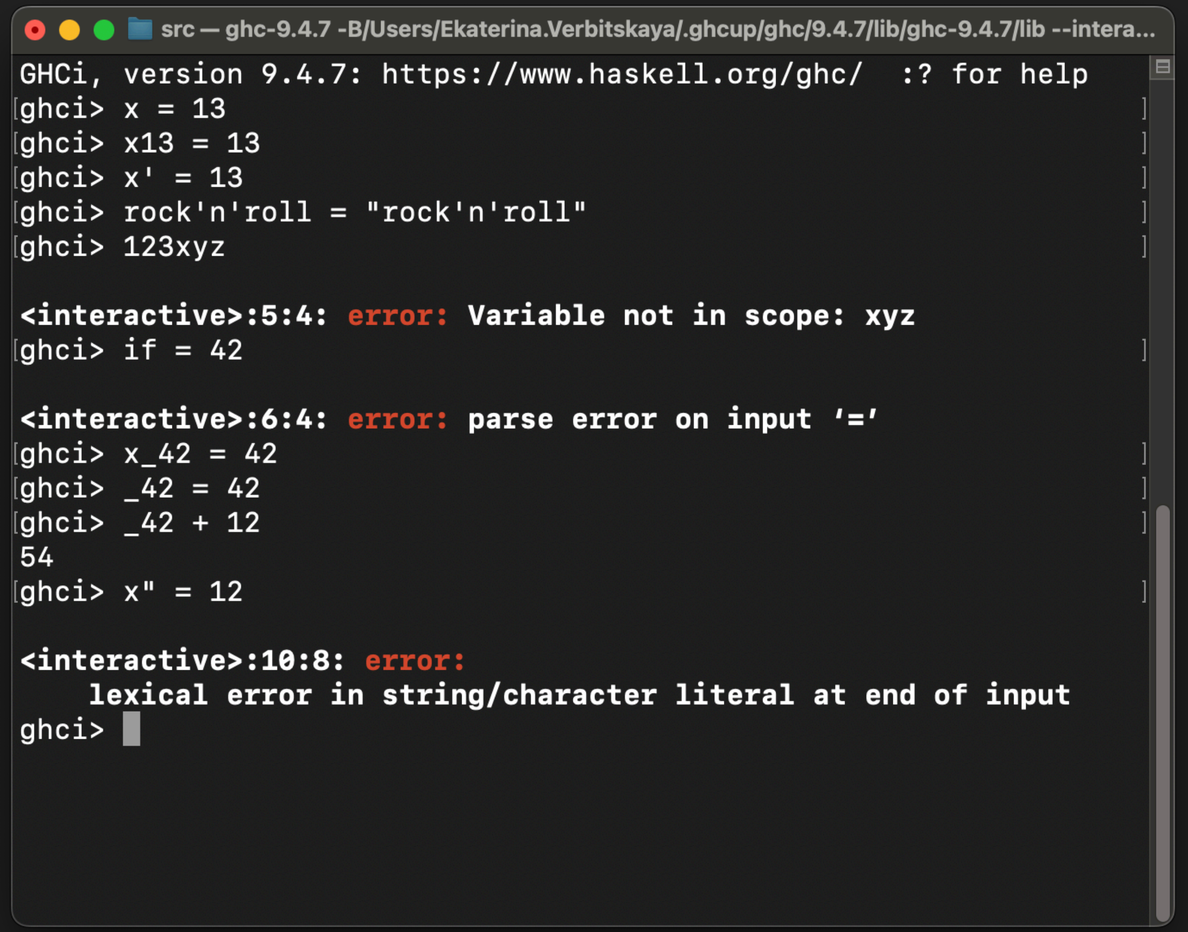


SOME FORMALITIES

- **Alphabet** – finite set of symbols
  - $\Sigma = \{a, b, c, \ldots, z\}$
  - $\Sigma = \{0,1,2,\ldots,9, -, ., e\}$
- **Universal language** – set of all strings that can be created from the symbols of the alphabet
  - $\Sigma^* = \{\varepsilon, a, b, \ldots, aa, ab, \ldots, ba, bb, \ldots\}$
  - $\Sigma^* = \{\varepsilon,0,1,\ldots,10,11,\ldots,123.45e-67,\ldots\}$
- **Language** – any subset of the universal language

LANGUAGE OF INDENTIFIERS

- Alphabet $\Sigma = \{a, b, c, \ldots, z, 0, 1, \ldots, 9, \_, '\}$
- Language L:
  - An identifier is a non-empty sequence of symbols of the alphabet
  - A keyword is not an identifier



PARSING INDENTIFIERS

- Let’s take a string and check if it’s an integer
- Can return an error
- If it’s a correct identifier, return its String value

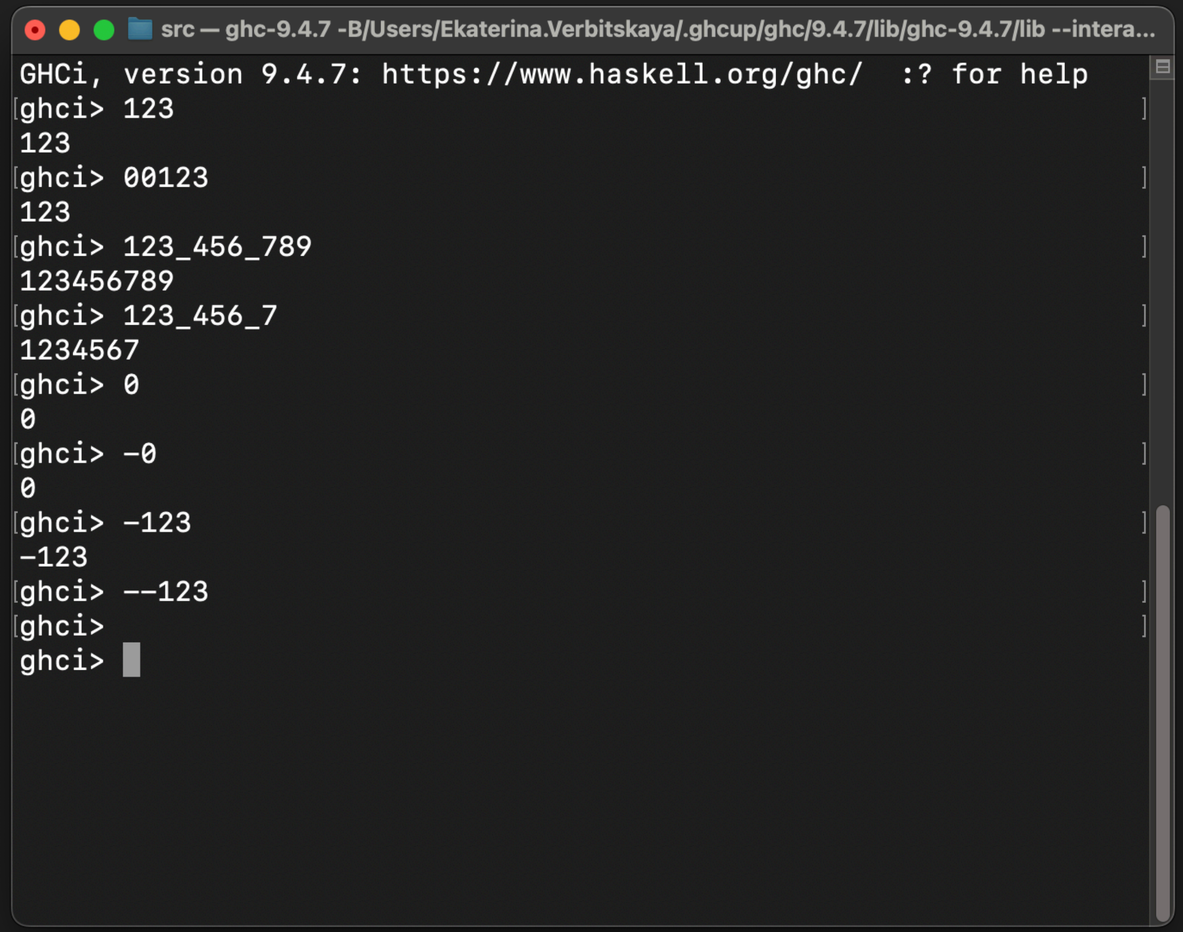
src — vim Parser.hs — 65×25

```
module Parser where

import Data.Char ( isAlpha, isAlphaNum )

keywords = ["if", "then", "else"]

parseIdent :: String -> Either String String
parseIdent str@(h:t)
    | isValidFirst h && all isValid t =
        if str `notElem` keywords
        then Right str
        else Left "Keyword cannot be an identifier"
    | otherwise = Left "Lexical error: inadmissible character"
  where
    isValidFirst x = x == '_' || x == '\'' || isAlpha x
    isValid x = isAlphaNum x || isValidFirst x
parseIdent [] = Left "Empty string is not an identifier"
```

"Parser.hs" [noeol] 26L, 622B

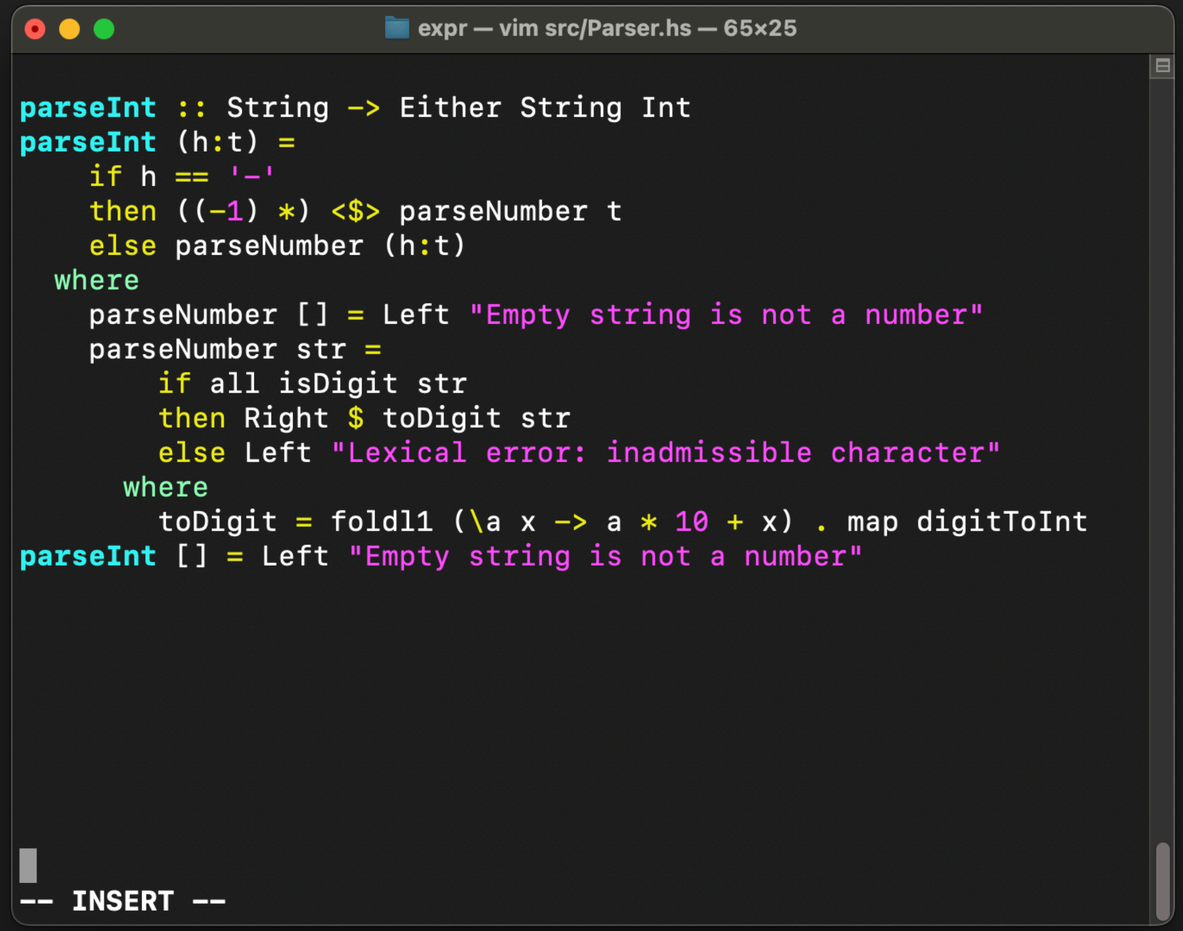
LANGUAGE OF INTEGER NUMBERS

- Alphabet $\Sigma = \{0,1,2,\ldots,9,-\}$
- Language L:
  - Digit is a one-symbol string $\{0,1,2,\ldots,9\}$
  - A number is a non-empty sequence of digits
  - An integer number is a number or a number with - in front of it.



LANGUAGE OF INTEGER NUMBERS

- Alphabet $\Sigma = \{0,1,2,\ldots,9,-\}$
- Language L:
  - Digit is a one-symbol string $\{0,1,2,\ldots,9\}$
  - A number is a non-empty sequence of digits
  - An integer number is a number or a number with - in front of it.



EXERCISE

- Implement a parser for Doubles
  - Use read to make a String into a Double

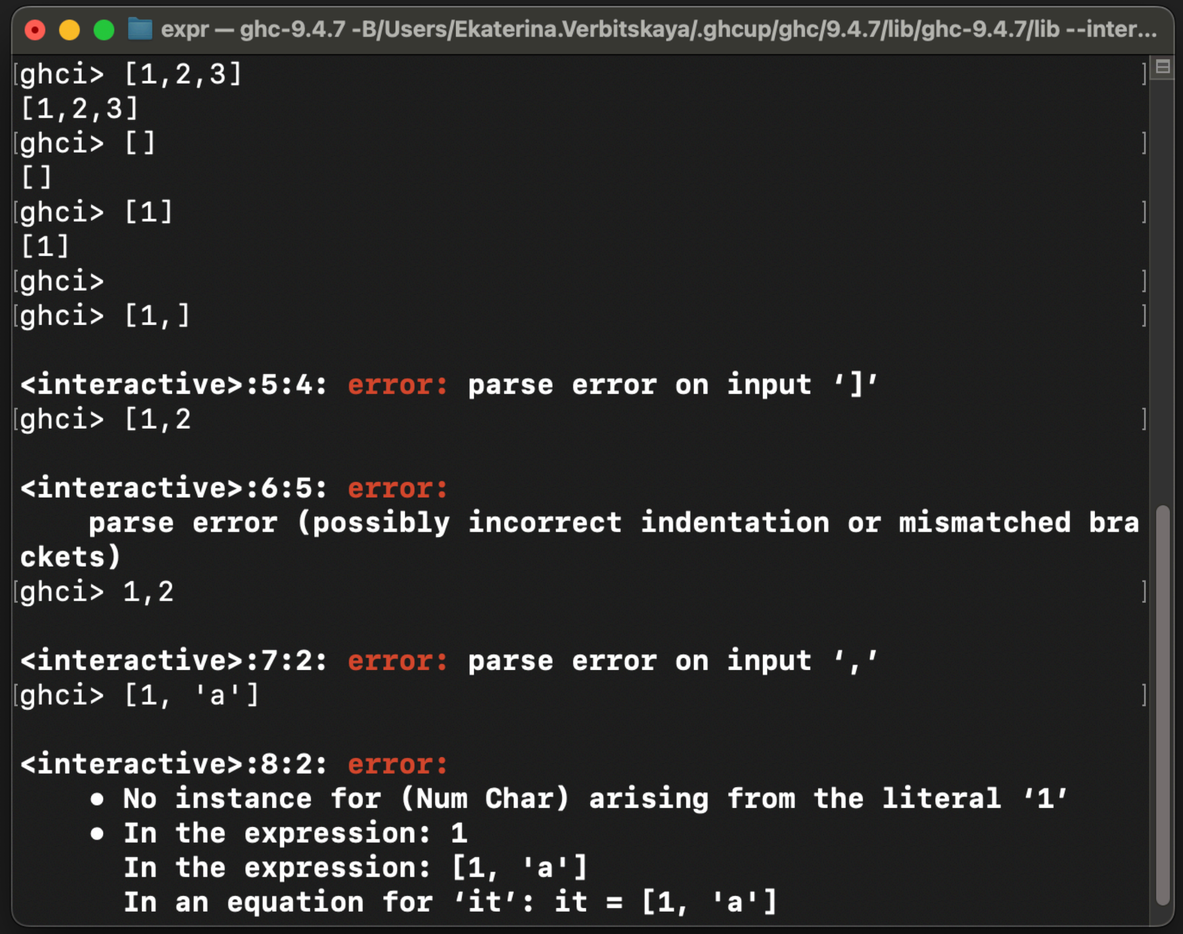
OUTLINE

- State Monad
- Crash Course on Parsing
- Monadic Parsers

PARSING OF A LIST OF NUMBERS

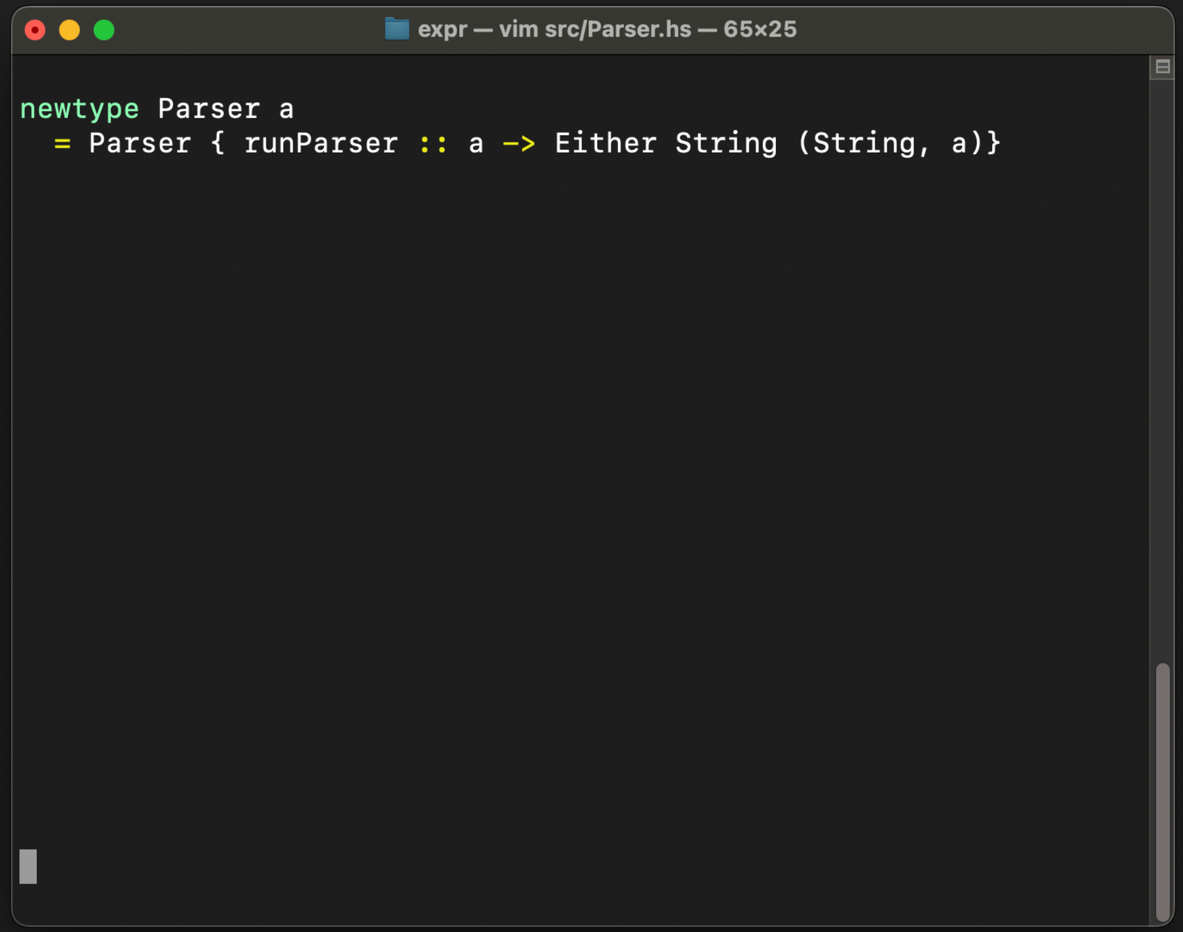
- A list is a possibly empty sequence of numbers
  - Each number is separated by ,
  - A list is surrounded by [and]

- We have a parser for numbers
- We can write parsers for ,, [and]
- There is no good way to compose them



BETTER PARSER ABSTRACTION

- A parser consumes a prefix of a string
- It returns the result and the leftover string suffix
- This abstraction allows us to easily compose simpler smaller parsers together
- A parser is a monad: what is the instance?

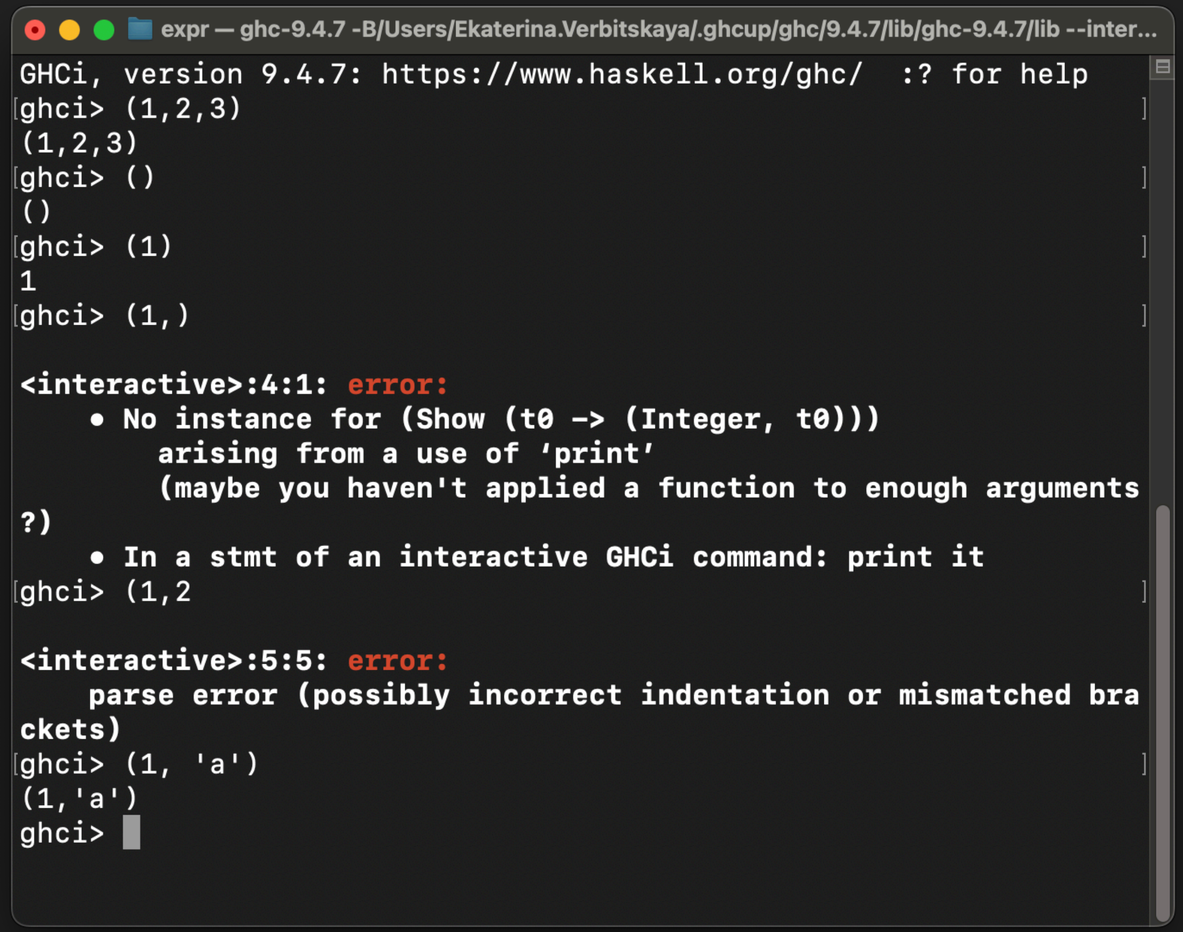


EXERCISE

- Implement monadic instances for Parser newtype

PARSING OF A TUPLE

- A tuple is a possibly empty sequence of values
  - Each value is separated by `,`
  - A list is surrounded by `(` and `)`
- This is very similar to lists: need for higher order parsers



PARSING OF AN PREFIX EXPRESSION

- Prefix expressions:
  - Either a number
  - Or a sequence:
    - Operator
    - Left operand
    - Right operand

PARSING OF AN INFIX EXPRESSIONS

- Expression is
  - A number, or
  - Expression, operator, Expression
    - Left recursion makes the parser hang
- Alternative point of view:
  - Expression is a list of operands separated by operators
  - We know how to parse this!