

INTRODUCTION TO

---

# FUNCTIONAL PROGRAMMING

## OUTLINE

---

- ▶ State Monad
- ▶ Crash Course on Parsing
- ▶ Monadic Parsers

## EFFECTS: STATE

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

```
● ○ ● intro2fp — ghc-9.4.7 -B/Users/Ekaterina.Verbitskaya/.ghcup/ghc/9.4.7/lib/ghc-9.4.7/lib --interact...
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>
```

# IMPLEMENTATION

```
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')
```

```
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)
```

## STATE MONAD

# EXAMPLES

```
src — vim StateDemo.hs — 65x25

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 — 65x25

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

```

introduction.fp — ghc-9.4.7 -B/Users/Ekaterina.Verbitskaya/.ghcup/ghc/9.4.7/lib/ghc-9.4.7/lib --interact...
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> eval (Let "x" (C 13) (Plus (V "x") (C 42))) []
([("x",13)],55)
ghci>

```

```

src — vim Expr.hs — 65x25
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 :: 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

```
src - ghc-9.4.7 -B/Users/Ekaterina.Verbitskaya/.ghcup/ghc/9.4.7/lib/ghc-9.4.7/lib --intera...
GHCI, version 9.4.7: https://www.haskell.org/ghc/ :? for help
ghci> "aSdlfkj;

<interactive>:1:10: error:
    lexical error in string/character literal at end of input
ghci> :{
ghci| let x = 42
ghci| x * 3
ghci| :}

<interactive>:4:1: error:
    parse error (possibly incorrect indentation or mismatched bra
ckets)
ghci> "123" + 123

<interactive>:4:7: error:
  • No instance for (Num String) arising from a use of '+'
  • In the expression: "123" + 123
    In an equation for 'it': it = "123" + 123
ghci>
```

# SOME FORMALITIES

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

# LANGUAGE OF IDENTIFIERS

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

```
src - ghc-9.4.7 -B/Users/Ekaterina.Verbitskaya/.ghcup/ghc/9.4.7/lib/ghc-9.4.7/lib --intera...
GHCI, version 9.4.7: https://www.haskell.org/ghc/ :? for help
[ghci]> x = 13
[ghci]> x13 = 13
[ghci]> x' = 13
[ghci]> rock'n'roll = "rock'n'roll"
[ghci]> 123xyz

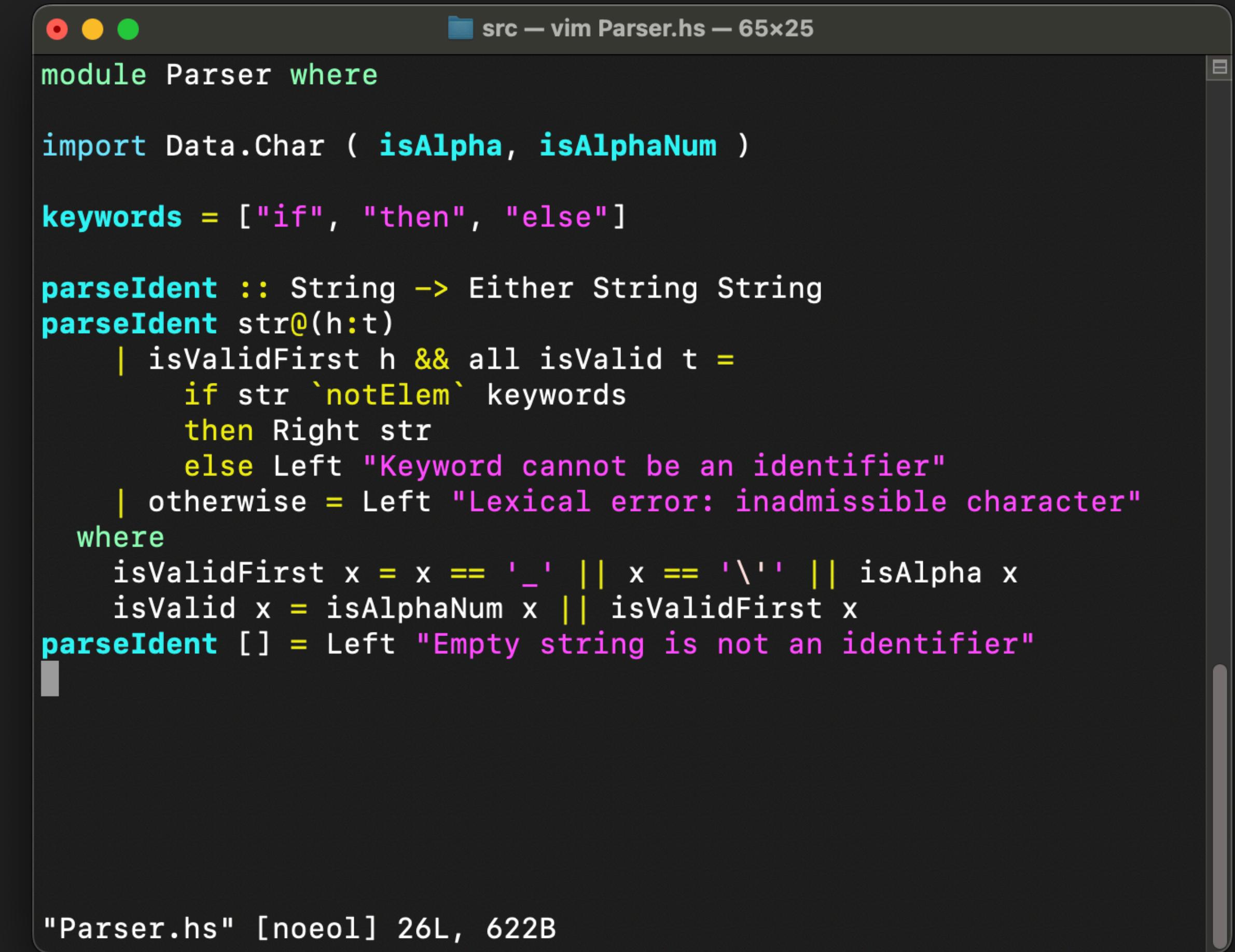
<interactive>:5:4: error: Variable not in scope: xyz
[ghci]> if = 42

<interactive>:6:4: error: parse error on input '='
[ghci]> x_42 = 42
[ghci]> _42 = 42
[ghci]> _42 + 12
54
[ghci]> x" = 12

<interactive>:10:8: error:
    lexical error in string/character literal at end of input
ghci> █
```

## PARSING IDENTIFIERS

- ▶ 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



A screenshot of a terminal window titled "src — vim Parser.hs — 65x25". The window contains Haskell code for a parser module. The code defines a module Parser with imports for Data.Char (isAlpha, isAlphaNum) and a list of keywords ("if", "then", "else"). It includes a parseIdent function that takes a string and returns either a Right (String) or Left (String) error message. The function checks if the string starts with a valid character (either underscore, backslash, or alpha), then checks if it's a keyword or a valid identifier. A where clause provides the definitions for isValidFirst and isValid functions. An empty list is also handled as an error.

```
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

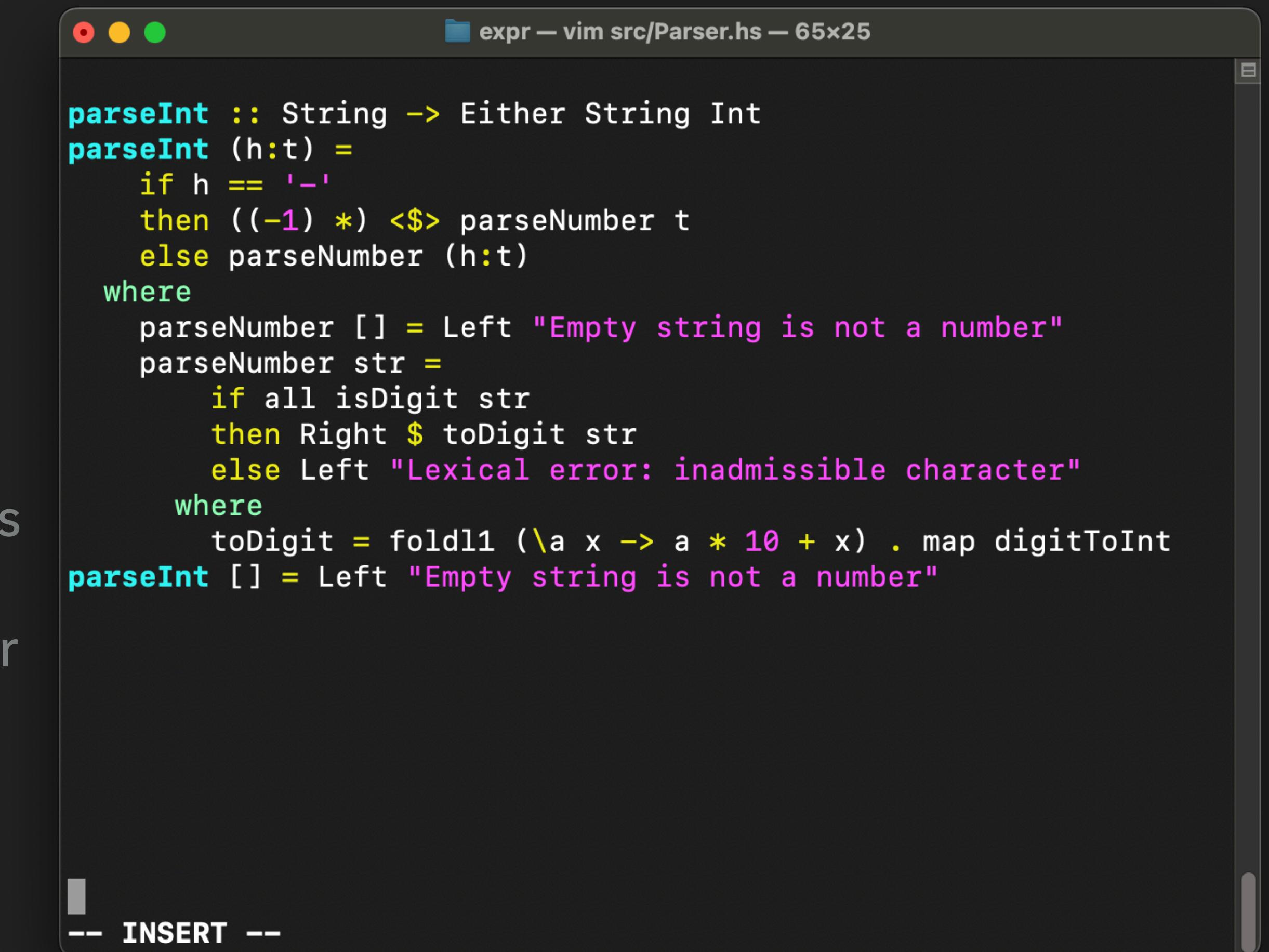
# LANGUAGE OF INTEGER NUMBERS

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

```
src — ghc-9.4.7 -B/Users/Ekaterina.Verbitskaya/.ghcup/ghc/9.4.7/lib/ghc-9.4.7/lib --intera...
GHCI, version 9.4.7: https://www.haskell.org/ghc/  ?: for help
[ghci]> 123
123
[ghci]> 00123
123
[ghci]> 123_456_789
123456789
[ghci]> 123_456_7
1234567
[ghci]> 0
0
[ghci]> -0
0
[ghci]> -123
-123
[ghci]> --123
[ghci]>
[ghci]>
```

# LANGUAGE OF INTEGER NUMBERS

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



A screenshot of a terminal window titled "expr — vim src/Parser.hs — 65x25". The window displays Haskell code for parsing integers. The code defines a function `parseInt` that takes a string and returns either a string or an integer. It handles a leading minus sign by multiplying the result of `parseNumber` by -1. The `parseNumber` function handles an empty string by returning an error. It also defines a `toDigit` function using `foldl1` and `map digitToInt`. The code uses the `Left` and `Right` constructors from the `Either` type.

```
parseInt :: String -> Either String Int
parseInt (h:t) =
  if h == '-' then ((-1) *) <$> parseNumber t
  else parseNumber (h:t)
where
  parseNumber [] = Left "Empty string is not a number"
  parseNumber str =
    if all isDigit str
    then Right $ toDigit str
    else Left "Lexical error: inadmissible character"
  where
    toDigit = foldl1 (\a x -> a * 10 + x) . map digitToInt
parseInt [] = Left "Empty string is not a number"

-- INSERT --
```

# EXERCISE

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

## OUTLINE

---

- ▶ State Monad
- ▶ Crash Course on Parsing
- ▶ Monadic Parsers

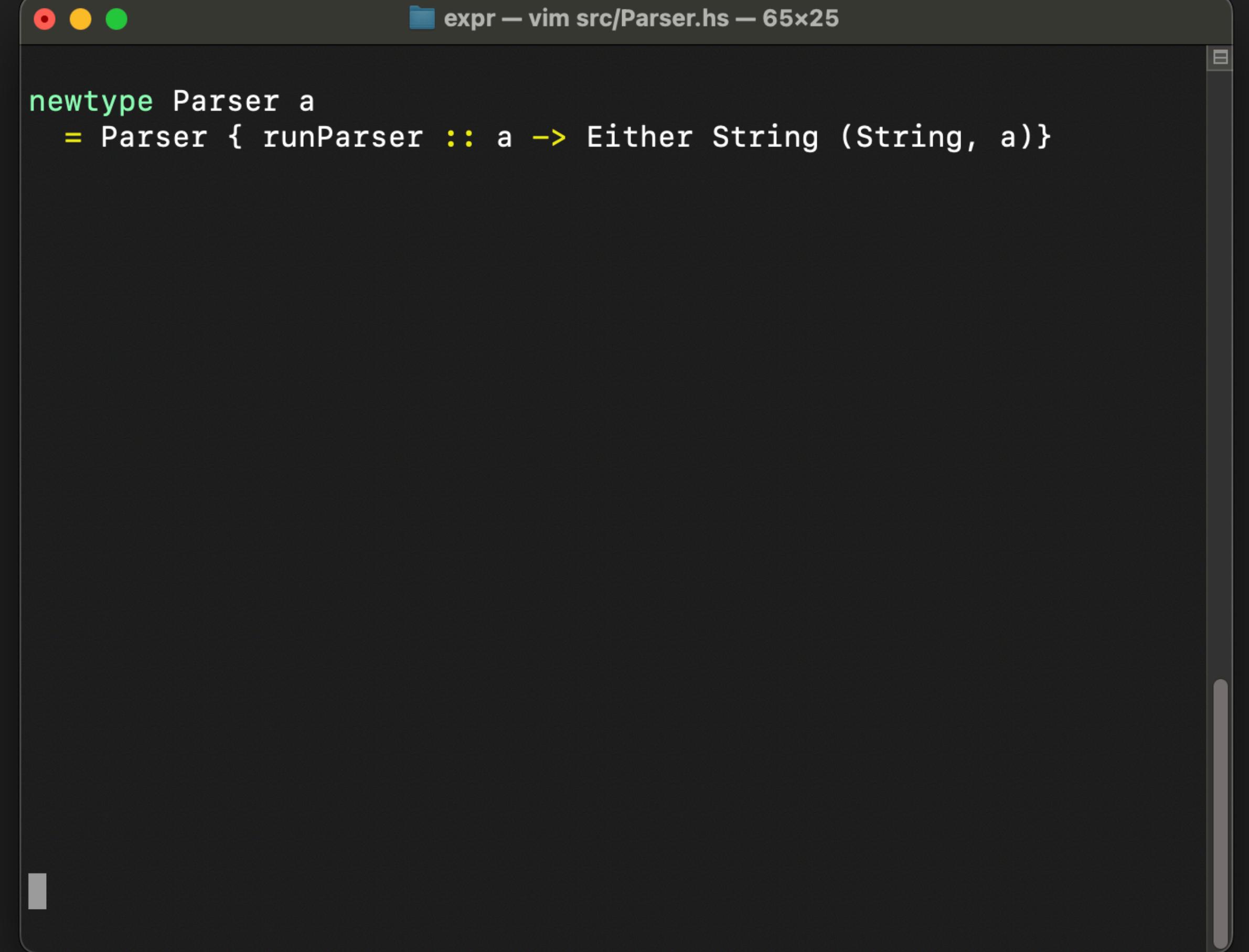
# PARSING OF A LIST OF NUMBERS

- ▶ 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

```
expr – ghc-9.4.7 -B/Users/Ekaterina.Verbitskaya/.ghcup/ghc/9.4.7/lib/ghc-9.4.7/lib --inter...  
ghci> [1,2,3]  
[1,2,3]  
ghci> []  
[]  
ghci> [1]  
[1]  
ghci>  
ghci> [1,]  
  
<interactive>:5:4: error: parse error on input ''  
ghci> [1,2  
  
<interactive>:6:5: error:  
    parse error (possibly incorrect indentation or mismatched brackets)  
ghci> 1,2  
  
<interactive>:7:2: error: parse error on input ','  
ghci> [1, 'a']  
  
<interactive>:8:2: error:  
• No instance for (Num Char) arising from the literal '1'  
• In the expression: 1  
  In the expression: [1, 'a']  
  In an equation for 'it': it = [1, 'a']
```

# 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?



A screenshot of a terminal window titled "expr — vim src/Parser.hs — 65x25". The window shows the following Haskell code:

```
newtype Parser a
  = Parser { runParser :: a -> Either String (String, a)}
```

# 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

```
expr -> ghc-9.4.7 -B/Users/Ekaterina.Verbitskaya/.ghcup/ghc/9.4.7/lib/ghc-9.4.7/lib --inter...  
GHCi, version 9.4.7: https://www.haskell.org/ghc/ :? for help  
[ghci> (1,2,3)  
(1,2,3)  
[ghci> ()  
()  
[ghci> (1)  
1  
[ghci> (1,)  
  
<interactive>:4:1: error:  
  • No instance for (Show (t0 -> (Integer, t0)))  
    arising from a use of 'print'  
    (maybe you haven't applied a function to enough arguments  
?)  
  • In a stmt of an interactive GHCi command: print it  
[ghci> (1,2  
  
<interactive>:5:5: error:  
  parse error (possibly incorrect indentation or mismatched brackets)  
[ghci> (1, 'a')  
(1,'a')  
ghci>
```

## 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!