Basics of Haskell

Bartosz Milewski

Hands On

- No installation, just an internet browser, School of Haskell site
- Go to https://FpComplete.com/user/bartosz/boh-code
- Run code online
- Edit code online
- Solve exercises online

- Implementing a software project in Haskell
- Open 1. Symbolic Calculator under user/bartosz/boh-code

Contents

- Basic syntax: functions, main
- Laziness, I/O, a taste of monads
- Symbolic calculator. Designing with types, recursion, conditionals
- Tokenizer. Defining data types, lists and recursion
- Single-character tokenizer. Function types, currying, guards
- Tokenizing numbers and identifiers. Higher order functions, lambdas, map, filter, fold
- Parser. Grammar, parse tree, threading state (token list)
- Evaluator. Threading symbol table
- Evaluator. Error handling, Either data type, case/of
- Monads for error handling and state
- Parser combinators

Basic Syntax

- Minimal function call syntax: `a b c d`
- The meaning of:
  - `f x y`
  - `(f x y)`
  - `f (x, y)`
- Precedence
  - `a b * c d` vs. `a (b * c) d`
  - `f (-1)`
  - `f g x` vs. `f (g x)`
- Dollar notation
  - `f $ g $ 1 + 2`
  - `Not: f g 1 + 2`
- Exercises: **2. Function Syntax**

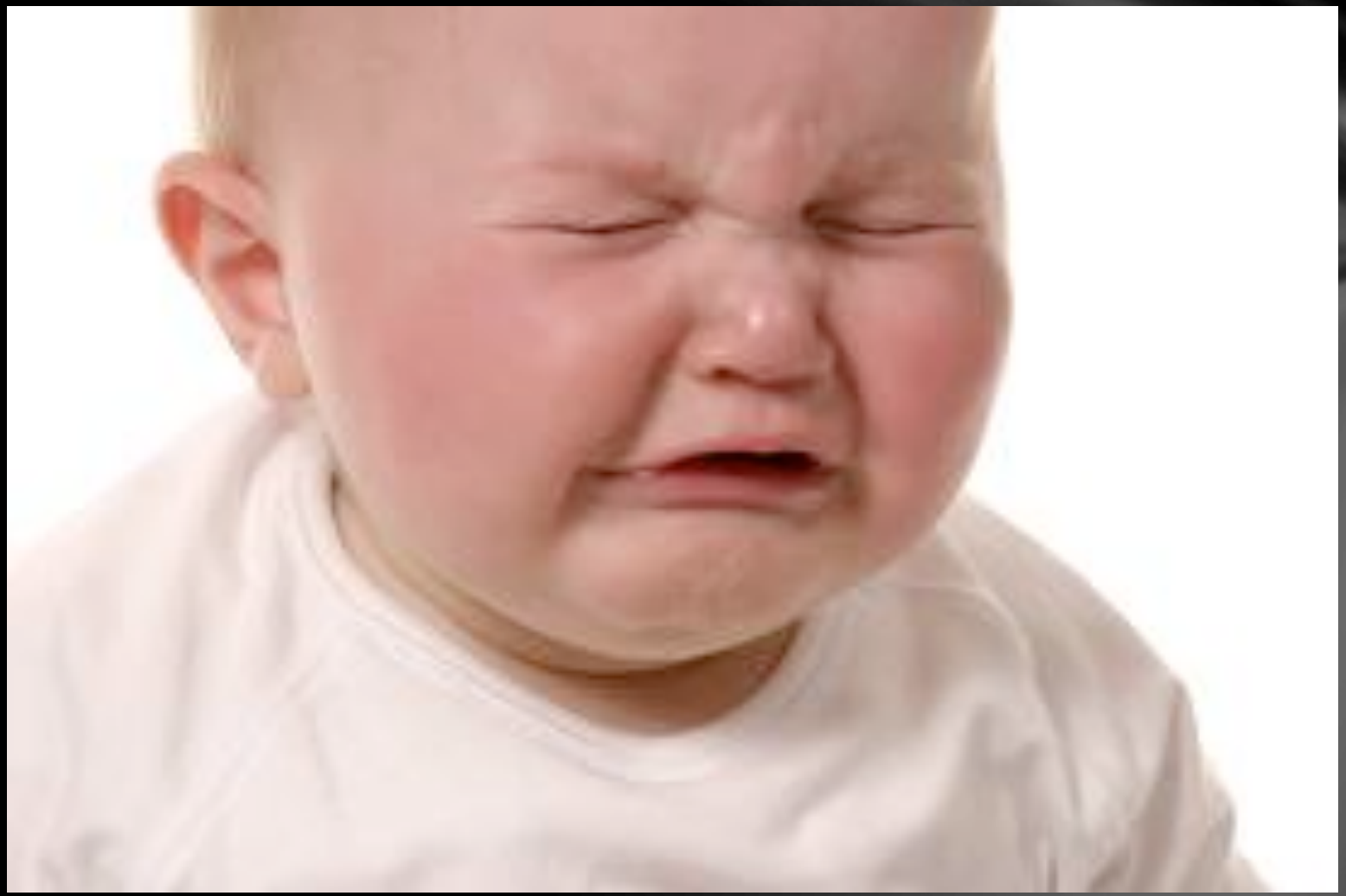
Function Definition

- Minimal syntax: `a b c d = expression`
- Definition is equation
  - `sq x = x * x`
- Substitution principle
  - `pyth a b = a * a + b * b`
  - `main = print $ pyth (1 + 2) (4/2)`
  - `main = print $ (1 + 2) * (1 + 2) + (4/2) * (4/2)`
- Main
  - `main = print $ sq 12`
- Pattern matching and pairs (tuples)
  - `pyth' (a, b) = a * a + b * b`
  - `main = print $ pyth’ (1, -1)`
- More exercises: **2. Function Syntax**

Pure Functions

- A function returns exactly the same result every time it's called with the same set of arguments. No state! Referential Transparency.
- A function has no side effects. Calling a function once is the same as calling it twice and discarding the result of the first call.
- I/O is side effects. Main is an IO action
  - `main = putStrLn "Hello World!“`
- Laziness and what makes a program run
- Sequencing using do notation (meaningful formatting)
  - `main = do`
  - `    putStrLn "The answer is: "`
  - `    print 43`
- User input
  - `main = do`
  - `    str <- getLine`
  - `    putStrLn str`

What? No side effects?



Symbolic Calculator

Design

To Level Design

- Main loop
  - Get a line of text from user
  - Calculate
  - Display result
- Calculation
  - Lexical analysis: tokenizer
  - Parser
  - Evaluator

```
data Token
data Tree

tokenize :: String -> [Token]
tokenize = undefined

parse :: [Token] -> Tree
parse = undefined

evaluate :: Tree -> Double
evaluate = undefined
```

Recursion

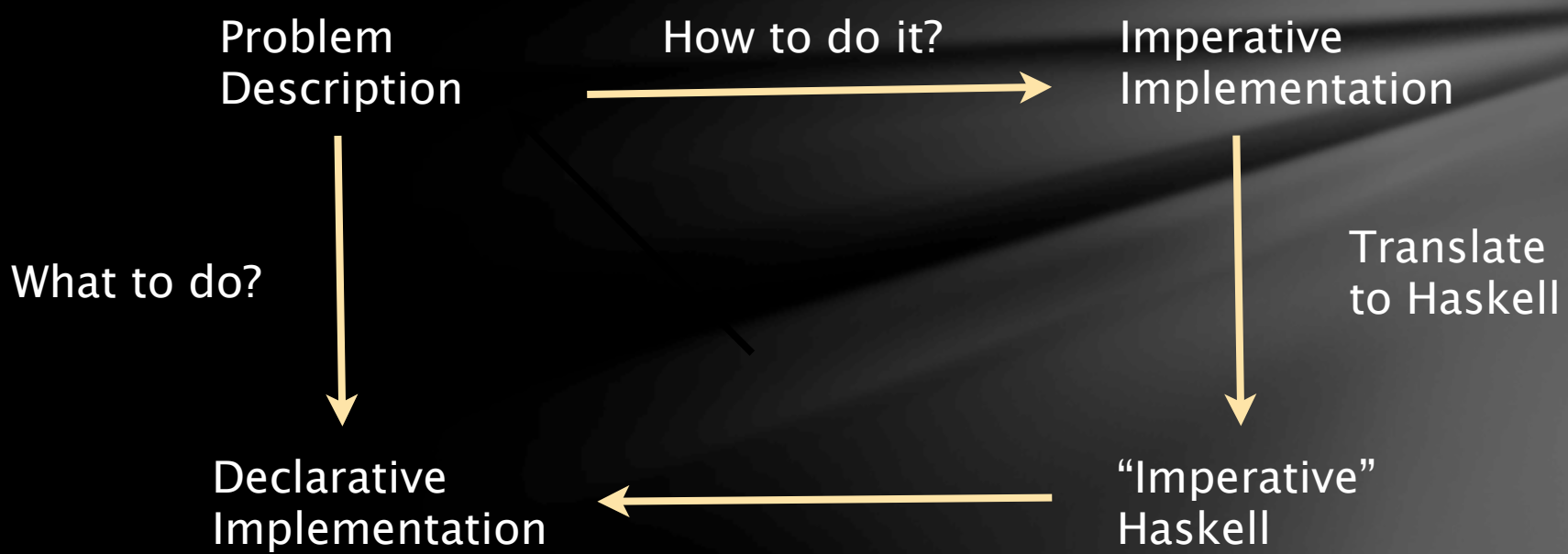
```
loop :: Int -> IO ()
loop n = do
    if n < 5
    then do
        putStrLn (show n)
        loop (n + 1)
    else
        return ()

main :: IO ()
main = loop 0
```

```
main = print [0..4]
```

More Exercises

Imperative vs. Declarative



Tokenizer

From string to list of tokens

Data Constructors

```
data Bool = True | False


boolToInt :: Bool -> Int
boolToInt b = if b then 1 else 0
```

- Pattern matching

```
boolToInt :: Bool -> Int
boolToInt True  = 1
boolToInt False = 0

main = print $ boolToInt False
```

- Exercise: Tokenizer

Token, Immutability

```
data Operator = Plus | Minus | Times | Div
    deriving (Show, Eq)
```

- Data constructors with arguments

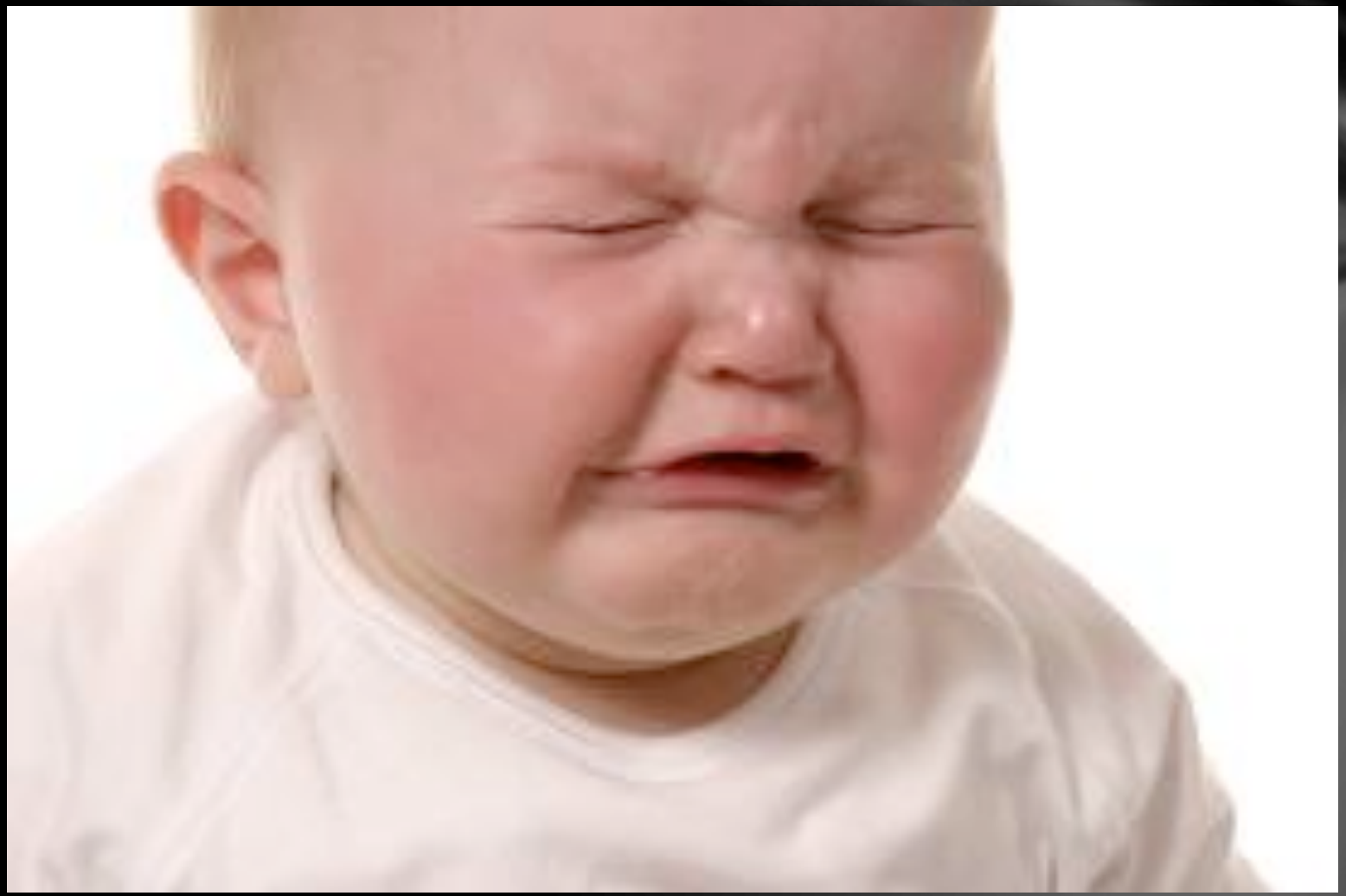
```
data Token = TokOp Operator
           | TokIdent String
           | TokNum Int
    deriving (Show, Eq)
```

- Pattern matching

```
prettyPrint :: Token -> String

prettyPrint (TokOp op)      = show op
prettyPrint (TokIdent str)  = str
prettyPrint (TokNum i)      = show i
```

What? No mutation?



List, Recursive Data

```
data List = Empty | Cons Int List
```

- Building lists

```
lst0, lst1, lst2 :: List
lst0 = Empty          -- empty list
lst1 = Cons 1 lst0    -- one-element list
lst2 = Cons 2 lst1    -- two-element list
```

- Built-in lists: Type list of `a`: `[a]`
  - Empty list: `[]`
  - Cons: `x : xs`
  - List literals: `[1, 2, 3, 4]`, `["Hi", "Hello", "Ciao"]`
  - Pattern matching: empty: `[]`, non-empty: `(c : cs)`
  - `type String = [Char]`
  - String literal: `"Hello"`
- **Exercises**

Categorizing Characters

- Simple categorizer: digit vs. alpha

```
import Data.Char

data Token = Digit | Alpha
    deriving (Show, Eq)

tokenize :: String -> [Token]
tokenize (c : rest) =
    if isDigit c
    then Digit : tokenize rest
    else Alpha : tokenize rest
tokenize [] = []

main = print $ tokenize "passwd123"
```

Multi-Argument Functions,

- Defining a function that calls `elem`

```
is3elem :: [Char] -> Bool
is3elem str = elem '3' str
```

- Currying function `elem`

```
is3elem :: [Char] -> Bool
is3elem = elem '3'
```

- Type signature

```
elem :: Char -> ([Char] -> Bool)
```

```
elem :: Char -> [Char] -> Bool
```

Guards

```
data Operator = Plus | Minus | Times | Div
    deriving (Show, Eq)

operator :: Char -> Operator
operator c | c == '+' = Plus
           | c == '-' = Minus
           | c == '*' = Times
           | c == '/' = Div
```

```
tokenize :: String -> [Token]
tokenize [] = []
tokenize (c : cs)
    | elem c "+-*/" = TokOp (operator c) : tokenize cs
    | otherwise     = error $ "Cannot tokenize " ++ [c]
```

- Code study: Single-character tokenizer

Break?

Separation of Concerns

- Traversal of the data structure (like C++ iterators)
- Operations on elements

```
tokenize :: String -> [Token]
tokenize [] = []
tokenize (c : cs)
    | elem c "+-*/" = TokOp (operator c) : tokenize cs
    | isDigit c  = TokNum (digitToInt c) : tokenize cs
    | isAlpha c  = TokIdent [c]          : tokenize cs
    | isSpace c  = tokenize cs
    | otherwise  = error $ "Cannot tokenize " ++ [c]
```

- Transforming one character at a time

Map and Filter

```
map :: (a -> b) -> [a] -> [b]
map _ [] = []
map f (a : as) = f a : map f as

main = print $ map toUpper "hello world!"
```

- Polymorphic in a and b
- std::transform in C++

```
filter :: (a -> Bool) -> [a] -> [a]
filter _ [] = []
filter p (a : as) = if p a
                    then a : filter p as
                    else filter p as

main = putStrLn $ filter isDigit "1x+3y"
```

- Polymorphic in a
- std::copy_if in C++

Map in Action

```
tokenize :: String -> [Token]
tokenize = map tokenizeChar
```

```
tokenizeChar :: Char -> Token
tokenizeChar c | elem c "+-*/" = TokOp (operator c)
               | isDigit c  = TokNum (digitToInt c)
               | isAlpha c  = TokIdent [c]
               | isSpace c  = TokSpace
               | otherwise  = error $ "Cannot
tokenize " ++ [c]
```

- Map can't change the shape (e.g. shorten the list)

Lambda

- Using helper function

```
deSpace :: [Token] -> [Token]
deSpace = filter notSpace

notSpace :: Token -> Bool
notSpace t = t /= TokSpace
```

- Using anonymous function (lambda)

```
deSpace :: [Token] -> [Token]
deSpace = filter (\t -> t /= TokSpace)
```

- Exercise

Accumulation

```
tokenize (c : cs)
    ...
    | isAlpha c = identifier c cs
    ...
```

```
identifier c cs = let (str, cs') = alnums cs in
                  TokIdent (c:str) : tokenize cs'
```

- Alnum returns a pair of lists

```
alnums :: [Char] -> (String, [Char])
alnums str = als "" str
  where -- String is the accumulator
    als :: String -> [Char] -> (String, [Char])
    als acc [] = (acc, [])
    als acc (c : cs) | isAlphaNum c =
                              let (acc', cs') = als acc cs
                              in (c:acc', cs')
                     | otherwise = (acc, c:cs)
```

Folding

```
foldl :: (a -> b -> a) -> a -> [b] -> a
```

```
foldr :: (b -> a -> a) -> a -> [b] -> a
```

- std::accumulate in C++

```
template <class InputIterator, class T, class BinaryOperation>
   T accumulate ( InputIterator first,
                  InputIterator last,
                  T init,
                  BinaryOperation binary_op );
```

- Exercise

Refactoring

```
digits :: String -> (String, String)
digits str = digs "" str
  where
    digs acc [] = (acc, [])
    digs acc (c : cs) | isDigit c =
                              let (acc', cs') = digs acc cs
                              in (c:acc', cs')
                      | otherwise = (acc, c:cs)
```

```
span :: (a -> Bool) -> [a] -> ([a], [a])
span pred str =
  let -- define a helper function 'spanAcc'
    spanAcc acc [] = (acc, [])
    spanAcc acc (c : cs) | pred c =
                             let (acc', cs') = spanAcc acc cs
                             in (c:acc', cs')
                         | otherwise = (acc, c:cs)
  in
    spanAcc [] str
```

Parser

From tokens to parse tree

Grammar

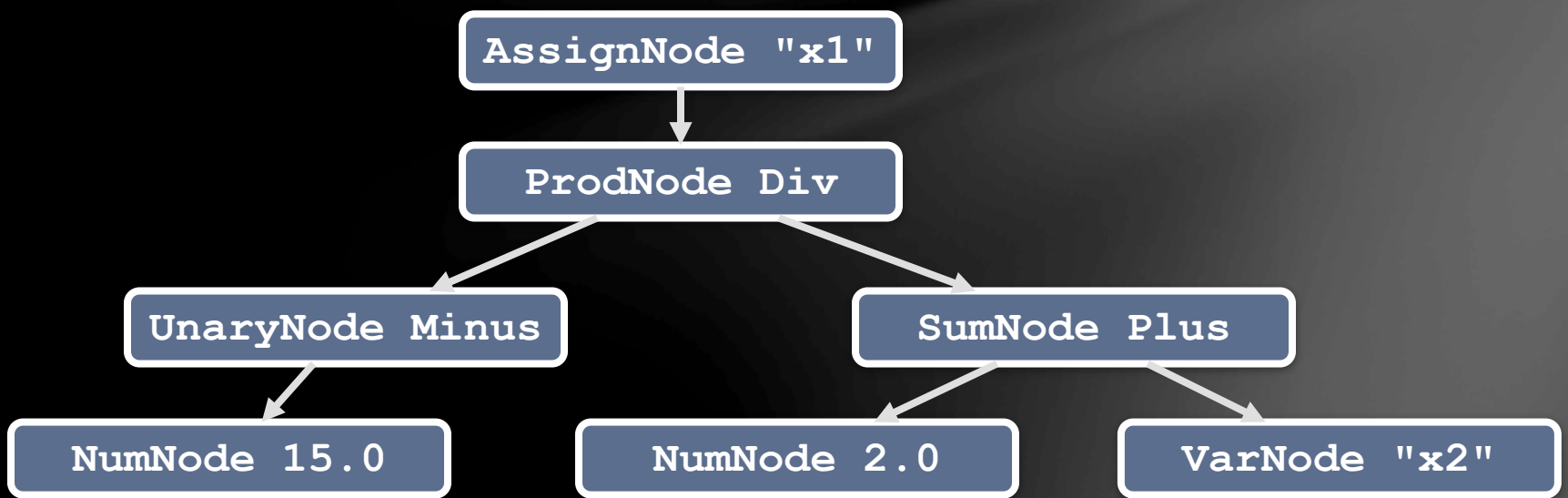
```
Expression <- Term [+-] Expression
            | Identifier '=' Expression
            | Term
Term       <- Factor [*/] Term
            | Factor
Factor     <- Number
            | Identifier
            | [+-] Factor
            | '(' Expression ')'
```

- Top-down recursive parser
- Line of user input is treated as an Expression

Parse Tree

```
data Tree = SumNode  Operator  Tree Tree
          | ProdNode Operator  Tree Tree
          | AssignNode String  Tree
          | UnaryNode Operator Tree
          | NumNode Double
          | VarNode String
  deriving Show
```

```
"x1 = -15 / (2 + x2)"
```



Threading Tokens

```
expression :: [Token] -> (Tree, [Token])
term       :: [Token] -> (Tree, [Token])
factor     :: [Token] -> (Tree, [Token])
```

- Token primitives

```
lookAhead :: [a] -> a
lookAhead [] = TokEnd
lookAhead (c:cs) = c

accept :: [a] -> [a]
accept [] = error "Nothing to accept"
accept (_:ts) = ts
```

- List implementation, persistent data structures, immutability

Expression

```
Expression <- Term [+-] Expression
            | Identifier '=' Expression
            | Term
```

```
expression :: [Token] -> (Tree, [Token])

expression toks =
   let (termTree, toks') = term toks
   in
      case lookAhead toks' of
         (TokOp op) | elem op [Plus, Minus] ->
            let (exTree, toks'') = expression (accept toks')
            in (SumNode op termTree exTree, toks'')
         TokAssign ->
            case termTree of
               VarNode str ->
                  let (exTree, toks'') =
                                 expression (accept toks')
                  in (AssignNode str exTree, toks'')
               _ -> error "Only variables can be assigned to"
         _ -> (termTree, toks')
```

- Exercise: Implement term

Evaluator

From parse tree to single number

Simple Evaluator

- Ignoring symbolic variables
- Pattern match all Tree constructors (nodes)
- Evaluate children first

```
evaluate :: Tree -> Double

evaluate (SumNode op left right) =
    let lft = evaluate left
        rgt = evaluate right
    in
        case op of
          Plus  -> lft + rgt
          Minus -> lft - rgt

evaluate (ProdNode op left right) = …
```

Symbol Table

- Threading symbol table for lookup and insertion/modification

```
evaluate :: SymTab -> Tree -> (Double, SymTab)
```

- Scope of symbol table

```
main = do
   loop (M.fromList [("pi", pi)])

loop symTab = do
   str <- getLine
   if null str
   then
      return ()
   else
      let toks = tokenize str
          tree = parse toks
          (val, symTab') = evaluate symTab tree
      in do
          print val
          loop symTab'
```

Symbol Table Primitives

- Data.Map

```
import qualified Data.Map as M
```

```
type SymTab = M.Map String Double
```

- Primitives

```
lookUp :: SymTab -> String -> (Double, SymTab)
lookUp symTab str =
    case M.lookup str symTab of
      Just v -> (v, symTab)
      Nothing -> error $ "Undefined variable " ++ str

addSymbol :: SymTab -> String -> Double -> (Double, SymTab)
addSymbol symTab str val =
    let symTab' = M.insert str val symTab
    in (val, symTab')
```

```
data Maybe a = Nothing | Just a
```

Evaluator

- Threading the symbol table

```
evaluate symTab (SumNode op left right) =
    let (lft, symTab')  = evaluate symTab  left
        (rgt, symTab'') = evaluate symTab' right
    in
        case op of
          Plus  -> (lft + rgt, symTab'')
          Minus -> (lft - rgt, symTab'')
```

- Nodes that use or modify the symbol table

```
evaluate symTab (VarNode str) = lookUp symTab str
```

```
evaluate symTab (AssignNode str tree) =
    let (v, symTab')  = evaluate symTab tree
    in addSymbol symTab' str v
```

- Exercise: Implement evaluate for UnaryNode

Error Handling

The Either monad, Exceptions

Break?

Functions that Fail

- Return Maybe: Only success or failure
- Return Either: Failure with error message

```
data Either a b = Left a | Right b
```

- Left on error: a usually fixed to String

```
lookUp :: SymTab -> String -> Either String (Double, SymTab)
lookUp symTab str =
    case M.lookup str symTab of
      Just v  -> Right (v, symTab)
      Nothing -> Left ("Undefined variable " ++ str)
```

Threading Errors

- Messy code, deeply nested (in any language, unless using

```
evaluate symTab (SumNode op left right) =
    case evaluate symTab left of
    Left msg -> Left msg
    Right (lft, symTab') ->
        case evaluate symTab' right of
        Left msg -> Left msg
        Right (rgt, symTab'') ->
            case op of
            Plus  -> Right (lft + rgt, symTab'')
            Minus -> Right (lft - rgt, symTab'')
```

- Can we abstract flow of control?

Abstracting Flow of Control

- This is the pattern with two holes

```
evaluate symTab (UnaryNode op tree) =
    let ev = evaluate symTab tree
    in
        case ev of
        Left msg -> Left msg
        Right result  -> ... call the rest with result
```

- This is the continuation that fits into second hole

```
\(x, symTab') ->
    case op of
    Plus  -> Right (x, symTab')
    Minus -> Right (-x, symTab')
```

Bind

- This is how bind is called

```
evaluate symTab (UnaryNode op tree) =
   bind (evaluate symTab tree)
        (\(x, symTab') ->
            case op of
            Plus  -> Right (x, symTab')
            Minus -> Right (-x, symTab'))
```

- This is the generic function bind. Short circuits the

```
bind :: Either String a -> (a -> Either String b)
        -> Either String b
bind eitherValue continueWith =
    case eitherValue of
    Left msg -> Left msg
    Right result  -> continueWith result
```

Monadic Types

- Return this “enriched” a instead of plain a

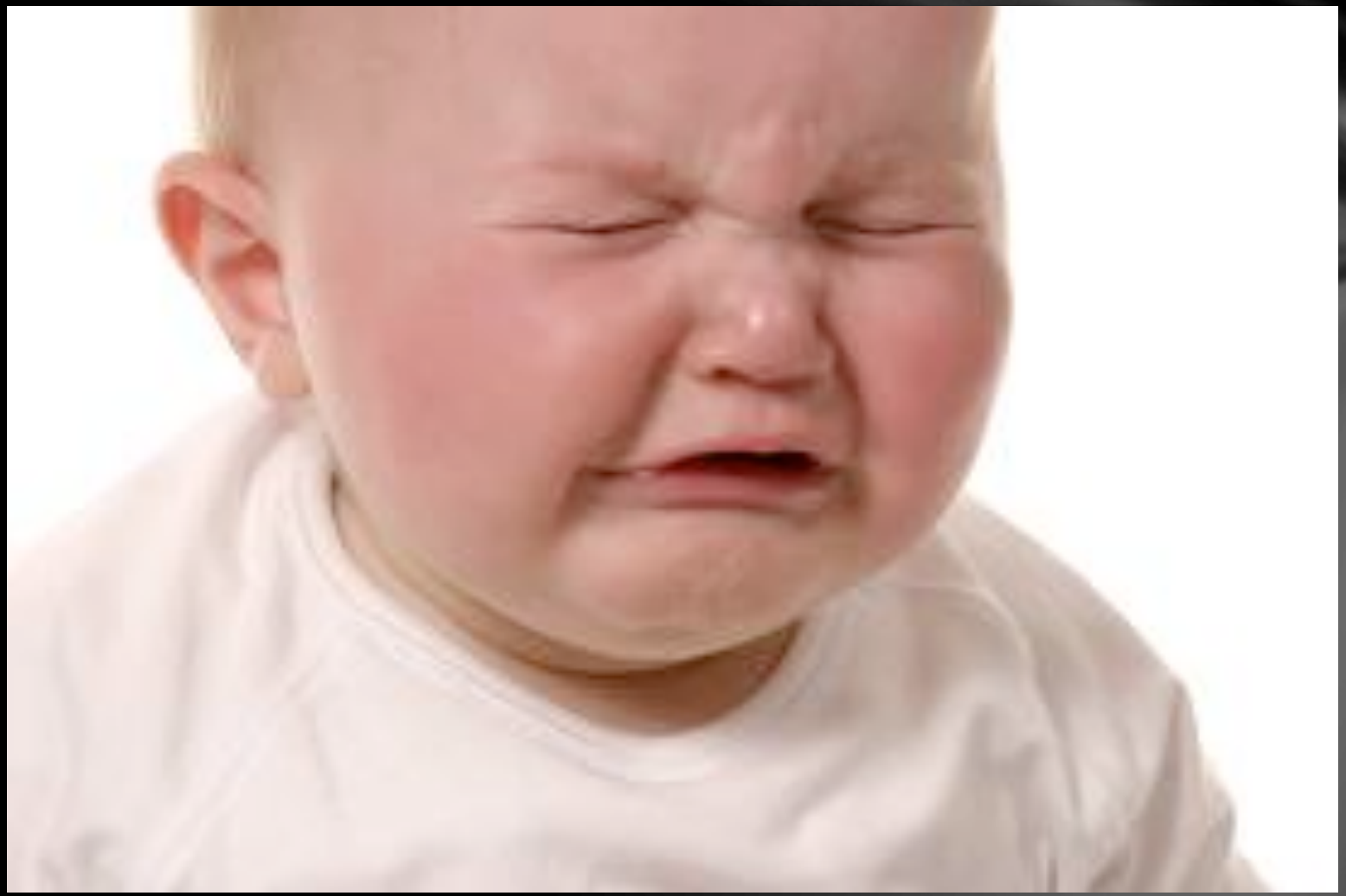
```
newtype Evaluator a = Ev (Either String a)
```

- Bind in infix operator notation

```
instance Monad Evaluator where
    (Ev eitherValue) >>= continueWith =
        case eitherValue of
        Left msg     -> Ev (Left msg)
        Right result -> continueWith result
    return x = Ev (Right x)
```

- return turns any value into enriched value

What? Monads?



Do Notation

- This is explicit use of bind and return

```
evaluate symTab (SumNode op left right) =
    evaluate symTab left >>= \(lft, symTab') ->
        evaluate symTab' right >>= \(rgt, symTab'') ->
            case op of
            Plus  -> return (lft + rgt, symTab'')
            Minus -> return (lft - rgt, symTab'')
```

- This is the same code using do

```
evaluate symTab (SumNode op left right) = do
    (lft, symTab')  <- evaluate symTab left
    (rgt, symTab'') <- evaluate symTab' right
    case op of
        Plus  -> return (lft + rgt, symTab'')
        Minus -> return (lft - rgt, symTab'')
```

Fail

- Optional redefinition of fail

```
instance Monad Evaluator where
    …
    fail err = Ev (Left err)
```

- Use of fail

```
lookUp :: SymTab -> String -> Evaluator (Double, SymTab)
lookUp symTab str =
    case M.lookup str symTab of
      Just v  -> return (v, symTab)
      Nothing -> fail ("Undefined variable " ++ str)
```

- Exercise: Maybe monad

State Monad

Threading mutable state

Threading state

- Error prone code

```
evaluate symTab (SumNode op left right) =
    let (lft, symTab') = evaluate symTab left
        (rgt, symTab'') = evaluate symTab' right
    in
        case op of
          Plus  -> (lft + rgt, symTab'')
          Minus -> (lft - rgt, symTab'')
```

```
evaluate :: SymTab -> Tree -> (Double, SymTab)
```

- Postpone the symbol table manipulation. Return functions

```
evaluate :: Tree -> (SymTab -> (Double, SymTab))
```

- Create actions that accept and return symbol table

```
let act = evaluate tree
(val, symTab') = act symTab
```

Creating Actions

```
evaluate :: Tree -> (SymTab -> (Double, SymTab))
evaluate (UnaryNode op tree) =
    \symTab ->
        let act = evaluate tree
            (x, symTab') = act symTab
        in case op of
            Plus  -> (x, symTab')
            Minus -> (-x, symTab')
```

```
addSymbol :: String -> Double -> (SymTab -> (Double, SymTab))
addSymbol str val =
    \symTab ->
        let symTab' = M.insert str val symTab
        in (val, symTab')
```

- Action that will add symbol to symbol table
- str and val are captured by the lambda -- closure

Enriched Values

- Instead or returning a, return a function that takes SymTab and returns (a, SymTab)

```
newtype Evaluator a = Ev (SymTab -> (a, SymTab))
```

- Construction and pattern matching noise

```
evaluator :: Tree -> Evaluator Double
evaluate (UnaryNode op tree) =
    Ev (\symTab ->
        let (Ev act) = evaluate tree
            (x, symTab') = act symTab
        in case op of
            Plus  -> (x, symTab')
            Minus -> (-x, symTab'))
```

- Inner lambda take SymTab and returns (Double, SymTab)

State Monad

- Figure out types first

```
(Ev act) >>= k
act    :: SymTab -> (a, SymTab)
k      :: a -> Ev (SymTab -> (b, SymTab))
result :: Ev (SymTab -> (b, SymTab))
```

```
instance Monad Evaluator where
    (Ev act) >>= k = Ev $
        \symTab ->
            let (x, symTab') = act symTab
            in
                let (Ev act') = k x
                in act' symTab'
    return x = Ev (\symTab -> (x, symTab))
```

- Execute action, unpack result, pass it to continuation, unpack result, execute result

Do Notation

```
evaluate :: Tree -> Evaluator Double
evaluate (SumNode op left right) = do
    lft <- evaluate left
    rgt <- evaluate right
    case op of
       Plus  -> return $ lft + rgt
       Minus -> return $ lft - rgt
```

```
evaluate (AssignNode str tree) = do
    v <- evaluate tree
    addSymbol str v
```

```
addSymbol :: String -> Double -> Evaluator Double
addSymbol str val = Ev $ \symTab ->
    let symTab' = M.insert str val symTab
    in  (val, symTab')
```

- Exercise: Monadic parser

Parsing Combinators

Composability

Failure as a Good Thing

```
term = do
   facTree <- factor
   tok <- lookAhead
   case tok of
          (TokOp op) | elem op [Times, Div] -> do
             accept
             termTree <- term
             return $ ProdNode op facTree termTree
          _ -> return facTree
```

- Two parsers combined
  - Look ahead for Times or Div, parse term, return ProdNode, or
  - `return facTree` – a trivial parser

```
term = do
   facTree <- factor
   multiplicative <|> return facTree
```

Mini Parsers

```
multiplicative :: Parser Double
multiplicative = do
    op <- tokProd
    termTree <- term
    return ProdNode op facTree termTree
```

- Define tokProd
- Capture facTree from inside term

```
tokProd :: Parser Operator
tokProd = do
    tok <- lookAhead
    case tok of
    TokOp op | elem op [Times, Div] -> do
        accept
        return op
    _ -> fail ""
```

The OR Combinator

```
infixr 2 <|>  -- infix, right associative, precedence 2

(<|>) :: Parser a -> Parser a -> Parser a
(P act) <|> (P act') =
   P (\toks ->
      case act toks of          -- act may "consume" tokens
         Left _ -> act' toks    -- but here we "roll back"
         Right (x, toks') -> Right (x, toks'))
```

- Try first parser
- If it fails, continue with the second parser (ignore error)
- If it succeeds, return the result

Matching a single token

```
token :: Token -> Parser Token
token tok = do
   t <- lookAhead
   if t == tok
   then do
       accept
       return t
   else
       fail ""
```

- When token recognized, accept, otherwise fail without accepting
- Exercise: Implement tokProd using token

Conclusion

- Functions are first class; pure, lazy, curriable, generic
- IO monad for side effects
- Data is immutable; pattern matching, persistent structures
- Flow of control and recursion abstracted into higher order functions
- Monad: a very general pattern, used in I/O, error propagation, state threading, and many more
- Abstraction, reusability, refactoring, composability
- Things we didn't talk about
  - Type classes
  - Functor, Applicative, Monoid
  - Concurrency, Software Transactional Memory, parallelism
  - Functional Reactive Programming