FIT2102
Programming Paradigms
Lecture 11

Case studies:

- Efficiency of pure data structures
- Solving problems
- Parser combinators

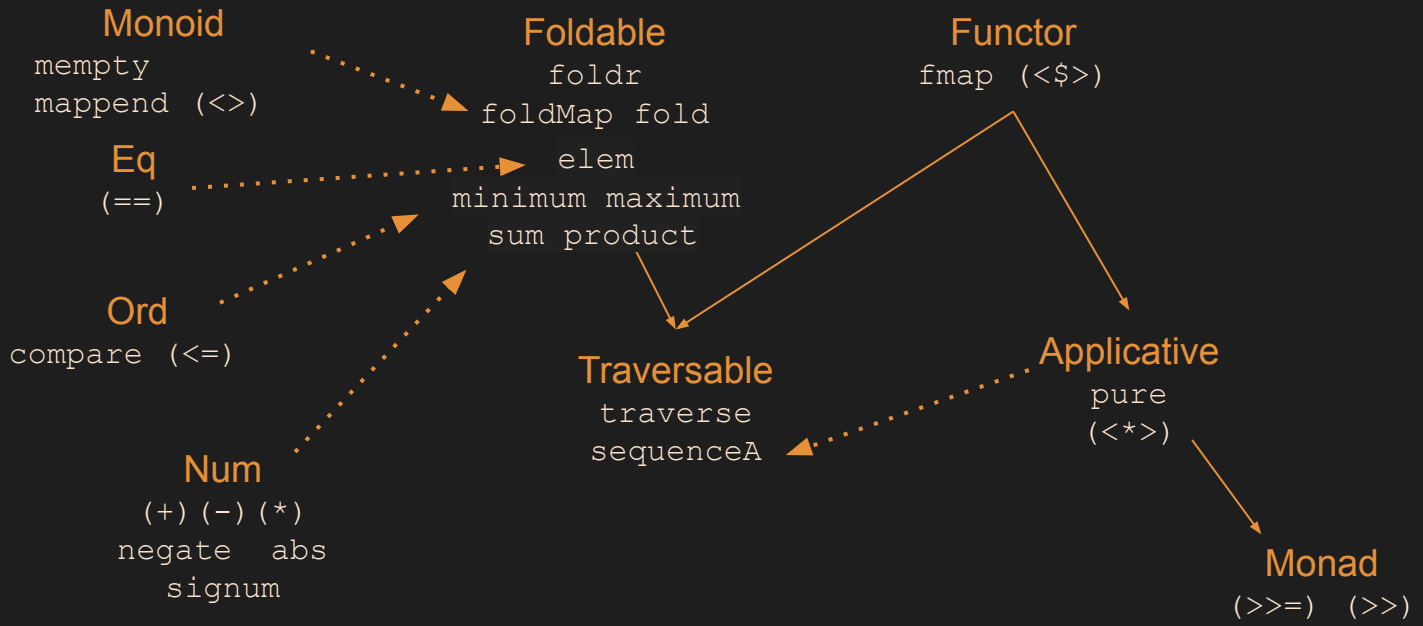
Faculty of Information Technology

MONASH University

Learning Outcomes

- Describe how pure Haskell functions are able to efficiently modify data-structures such as trees without mutating variables
- Apply what we have learned about Haskell typeclasses and other functional programming concepts to create solutions to real-world problems
- In particular, we learn to use Parser combinators and see how they are put together

Typeclasses for:
maps, folds, traversals and joining contexts



Parser Typeclass Instances

```haskell
instance Functor ParseResult where
  fmap f (Result i a) = Result i (f a)
  fmap _ (Error e)    = Error e

instance Functor Parser where
 fmap f (P p) = P (\i -> f <$> p i)          -- e.g. ("03"++) <$> basicNumber

instance Applicative Parser where
 pure x = P (`Result` x)                     -- creates a Parser that always succeeds with the given input
 (<*>) p q = p >>= (\f -> q >>= (pure . f)) -- e.g. parse ((,) <$> fourDigits <*> fourDigits) "12345687"

instance Monad Parser where
 (>>=) (P p) f = P (                         -- creates a parser that
   \i -> case p i of                         -- runs the first parser on the given input
     Result rest x -> parse (f x) rest       -- gives the result to f, and runs the resulting parser on rest
     Error e -> Error e)                     -- or fails
```

Monad recap

```
> :i Monad
class Applicative m => Monad (m :: * -> *) where
   (>>=) :: m a -> (a -> m b) -> m b
   (>>) :: m a -> m b -> m b
   return :: a -> m a
   ...
   {-# MINIMAL (>>=) #-}
       -- Defined in `GHC.Base'
instance Monad (Either e) -- Defined in `Data.Either'
instance Monad [] -- Defined in `GHC.Base'
instance Monad Maybe -- Defined in `GHC.Base'
instance Monad IO -- Defined in `GHC.Base'
instance Monad ((->) r) -- Defined in `GHC.Base'
instance Monoid a => Monad ((,) a) -- Defined in `GHC.Base'
```

Parser Combinators

```haskell
-- | Parse a phone number given precisely as: 9583 1762
basicNumber = do
   first <- thisMany 4 digit
   is ' '
   second <- thisMany 4 digit
   pure $ first ++ second
```

Parser Combinators

What about area codes? (03) 9583 1762
What about arbitrary spaces between groups? 9583 1762

Grammar (BNF):

```
<areaCode> ::= '(' <digit> <digit> ')'

<fourDigits> ::= <digit> <digit> <digit> <digit>

<basicNumber> ::= spaces <fourDigits> spaces <fourDigits>

<fullNumber> ::= areaCode basicNumber

<phoneNumber> ::= fullNumber | basicNumber
```

Our Parser (from Week 11 Tute)

```haskell
-- | Return a parser that produces a character between '0' and '9' but fails if
--   * the input is empty; or
--   * the produced character is not a digit.
digit :: Parser Char

-- | Return a parser that produces the given number of values off the given
-- parser.  This parser fails if the given parser fails in the attempt to
-- produce the given number of values.
--
-- /Hint/: Use 'sequenceParser' and 'replicate'.
--
-- >>> parse (thisMany 4 upper) "ABCDef"
-- Result >ef< "ABCD"
--
-- >>> isErrorResult (parse (thisMany 4 upper) "ABcDef")
-- True
thisMany :: Int -> Parser a -> Parser [a]
```

Our Parser (from Week 11 Tute)

```haskell
-- | Write a function that applies the given parser, then parses 0 or more
-- spaces, then produces the result of the original parser.
--
-- >>> parse (tok (is 'a')) "a bc"
-- Result >bc< 'a'
-- >>> parse (tok (is 'a')) "abc"
-- Result >bc< 'a'
tok :: Parser a -> Parser a

-- | Write a function that parses the given char followed by 0 or more spaces.
-- >>> parse (charTok 'a') "abc"
-- Result >bc< 'a'
-- >>> isErrorResult (parse (charTok 'a') "dabc")
-- True
charTok :: Char -> Parser Char
```

Our Parser (from Week 11 Tute)

```haskell
-- | Write a function that applies the first parser, runs the third parser
-- keeping the result, then runs the second parser and produces the obtained
-- result.
-- >>> parse (between (is '[') (is ']') character) "[a]"
-- Result >< 'a'
between :: Parser o -> Parser c -> Parser a -> Parser a
```

Our Parser (from Week 11 Tute)

```haskell
-- | Write a function that applies the given parser in between the two given characters.
--
-- >>> parse (betweenCharTok '[' ']' character) "[a]"
-- Result >< 'a'
betweenCharTok :: Char -> Char -> Parser a -> Parser a
betweenCharTok o c = between (charTok o) (charTok c)
```

Our Parser (from Week 11 Tute)

```
-- | Return a parser that tries the first parser for a successful value, then:
--   * if the first parser succeeds then use this parser; or
--   * if the first parser fails, try the second parser.
--
-- >>> parse (character ||| pure 'v') ""
-- Result >< 'v'
--
-- >>> parse (failed UnexpectedEof ||| pure 'v') ""
-- Result >< 'v'
--
-- >>> parse (character ||| pure 'v') "abc"
-- Result >bc< 'a'
--
-- >>> parse (failed UnexpectedEof ||| pure 'v') "abc"
-- Result >abc< 'v'
(|||) :: Parser a -> Parser a -> Parser a
```

```
areaCode = betweenCharTok '(' ')' (thisMany 2 digit)

fourDigits = thisMany 4 digit

basicNumber = do
   spaces
   first <- fourDigits
   spaces
   second <- fourDigits
   pure (first ++ second)

fullNumber = do
   ac <- areaCode
   n <- basicNumber
   return (ac ++ n)

phoneNumber = fullNumber ||| ("03"++) <$> basicNumber
```

```
GHCi> parse phoneNumber "(02)9583 1762"
Result >< "0295831762"

GHCi> parse phoneNumber "9583  1762"
Result >< "0395831762"
```

The Parser Type

```haskell
data ParseError =
   UnexpectedEof
 | ExpectedEof Input
 | UnexpectedChar Char
 | UnexpectedString String
 deriving (Eq, Show)

data ParseResult a =
   Error ParseError
 | Result Input a
 deriving Eq

type Input = String
newtype Parser a = P { parse :: Input -> ParseResult a}
```

Calculator

```
e1 = " 12 + 21* 3 "
e2 = "6 * 4 + 333 + 8 * 24"
e3 = "6 * 4 + 333 - 8 * 24"

data Expr = Plus Expr Expr | Minus Expr Expr | Times Expr Expr | Number Int
 deriving Show

t = Plus (Number 12) (Times (Number 21) (Number 3))

calc (Plus x y) = calc x + calc y
calc (Minus x y) = calc x - calc y
calc (Times x y) = calc x * calc y
calc (Number x) = x
```

Calculator Parser

```
-- grammar:
-- <expr> ::= <term> { <add> <term> }
-- <term> ::= <number> { "*" <number> }
-- <add> ::= "+" | "-"

add :: Parser (Expr -> Expr -> Expr)
add = (op '+' >> pure Plus) ||| (op '-' >> pure Minus)

times :: Parser (Expr -> Expr -> Expr)
times = op '*' >> pure Times

expr :: Parser Expr
expr = term `chain` add

term :: Parser Expr
term = number `chain` times

parseCalc :: String -> ParseResult Expr
parseCalc = parse expr
```

```
> calc <$> parseCalc "6 * 4 + 333 + 8 * 24"
Result >< 549
```

Standard Parser Bits

```
op :: Char -> Parser Char -- parse a single char operator
op c = do
   spaces
   charTok c

number :: Parser Expr     -- parse a Number
number = Number <$> read <$> list1 digit

chain :: Parser a -> Parser (a->a->a) -> Parser a -- chain p op parses 1 or more instances of p
chain p op = p >>= rest                           -- separated by op
   where                                          -- (see chainl1 from Text.Parsec)
   rest a = (do
               f <- op
               b <- p
               rest (f a b)
            ) ||| pure a
```

Parser Combinators

- Not going to be on the exam, but this exercise:
  - Gives you a last chance to explore the typeclasses we have studied: Functor, Applicative, Monad
  - Demonstrates Haskell's potential as a platform for creating Domain Specific Languages (DSLs)
  - Closes a nice loop opened by the JSON pretty printer we build in week 4 (we're going to build a simple JSON parser)
  - Parsing is another classical topic when considering Programming Language Paradigms… the nice thing about the parser combinators we are going to be building is that they allow us to build parsers using code that very much looks like the standard conventions for formally defining grammars (BNF)

Pure Functions and Dynamic Data Structures

We’ve constructed immutable data structures such as linked (cons) lists and trees.

We saw that you build these up by calling constructor functions that take a data value and one or more other instances of the type.

But once constructed, how do you modify an element in such a data structure?

A problem with pure linked lists…

A Haskell list is composed by consing elements together:

```
l = 1 : 2 : 3 : 4 : []
```

Changing the head element is easy (and requires O(1) operations)

```
> 9 : tail l
```

```
[9,2,3,4]
```

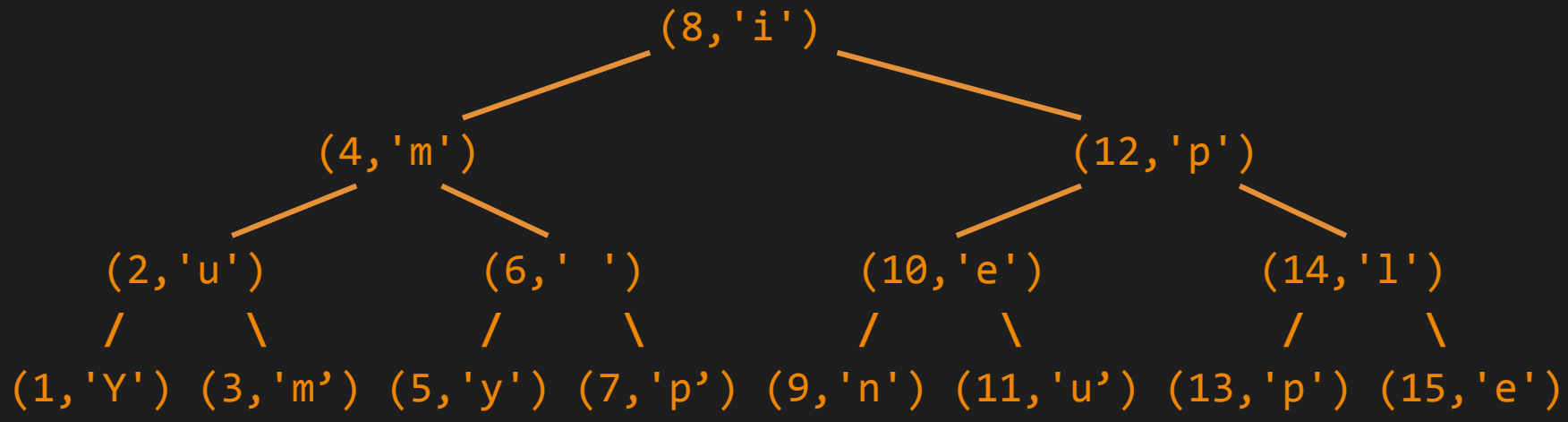
How many times have we called cons so far?

But how do we change the last element?

Exercise 1

To be announced...

Consider an Ordered, Balanced Binary Tree




```
data BinTree a = Nil | Node a (BinTree a) (BinTree a)
type LookupTree a = BinTree (Integer, a)
```

```
t :: LookupTree Char
t = Node (8,'i') (
       Node (4,'m') (
           Node (2,'u')
               (Node (1,'Y') Nil Nil)
               (Node (3,'m') Nil Nil)
       ) (
           Node (6,' ')
               (Node (5,'y') Nil Nil)
               (Node (7,'p') Nil Nil)
       )
   ) (
       Node (12,'p') (
           Node (10,'e')
               (Node (9,'n') Nil Nil)
               (Node (11,'u') Nil Nil)
       ) (
           Node (14,'l')
               (Node (13,'p') Nil Nil)
               (Node (15,'e') Nil Nil)
       )
   )
```

Walking the tree

```haskell
data BinTree a = Nil | Node a (BinTree a) (BinTree a)
type LookupTree a = BinTree (Integer, a)

fromBinTree :: BinTree a -> [a]
fromBinTree Nil = []
fromBinTree (Node v l r) = fromBinTree l ++ [v] ++ fromBinTree r
```

```
ghci> fromBinTree t
[(1,'Y'),(2,'u'),(3,'m'),(4,'m'),(5,'y'),(6,'_'),(7,'p'),(8,'i'),(9
,'n'),(10,'e'),(11,'u'),(12,'p'),(13,'p'),(14,'l'),(15,'e')]
```

Walking the tree

```
data BinTree a = Nil | Node a (BinTree a) (BinTree a)
type LookupTree a = BinTree (Integer, a)

fromBinTree :: BinTree a -> [a]
fromBinTree Nil = []
fromBinTree (Node v l r) = fromBinTree l ++ [v] ++ fromBinTree r


ghci> map (\(key,value)->value) $ fromBinTree t
"Yummy pineupple"
```

Walking the tree

```
data BinTree a = Nil | Node a (BinTree a) (BinTree a)
type LookupTree a = BinTree (Integer, a)

fromBinTree :: BinTree a -> [a]
fromBinTree Nil = []
fromBinTree (Node v l r) = fromBinTree l ++ [v] ++ fromBinTree r


ghci> map snd $ fromBinTree t
"Yummy pineupple"
```

Walking the tree with Foldable

```
data BinTree a = Nil | Node a (BinTree a) (BinTree a)
type LookupTree a = BinTree (Integer, a)

instance Foldable BinTree where
   foldMap _ Nil = mempty
   foldMap f (Node v l r) = mconcat [foldMap f l, f v, foldMap f r]

ghci> foldMap (\(_,c)->[c]) $ fromBinTree t
"Yummy pineupple"
```

Walking the tree with Foldable

```
data BinTree a = Nil | Node a (BinTree a) (BinTree a)
type LookupTree a = BinTree (Integer, a)

instance Foldable BinTree where
   foldMap _ Nil = mempty
   foldMap f (Node v l r) = mconcat [foldMap f l, f v, foldMap f r]

ghci> foldMap (\(_,c)->[c]) t
"Yummy pineupple"
```

Walking the tree with Foldable

```
data BinTree a = Nil | Node a (BinTree a) (BinTree a)
type LookupTree a = BinTree (Integer, a)

instance Foldable BinTree where
   foldMap _ Nil = mempty
   foldMap f (Node v l r) = mconcat [foldMap f l, f v, foldMap f r]

ghci> foldMap (pure.snd) t :: String
"Yummy pineupple"
```

Walking the tree with Foldable

```haskell
data BinTree a = Nil | Node a (BinTree a) (BinTree a)

-- implementing foldMap for the Foldable instance
-- gives us foldr for free
count :: BinTree a -> Integer
count = foldr (\i a->(1+a)) 0

-- | @count@
-- >>> count t
-- 15
```

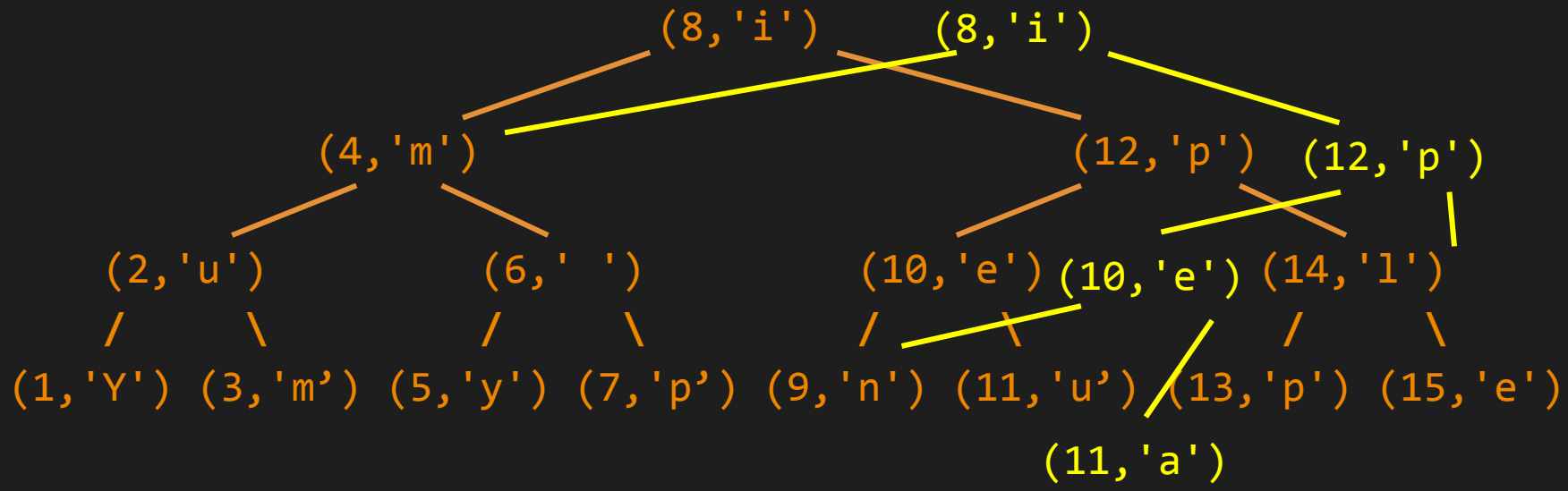
Walking the tree with Foldable

```haskell
data BinTree a = Nil | Node a (BinTree a) (BinTree a)

-- implementing foldMap for the Foldable instance
-- gives us foldr for free
count :: BinTree a -> Integer
count = foldr (const (1+)) 0     -- const :: a -> b -> a

-- | @count@
-- >>> count t
-- 15
```

Mutating the tree

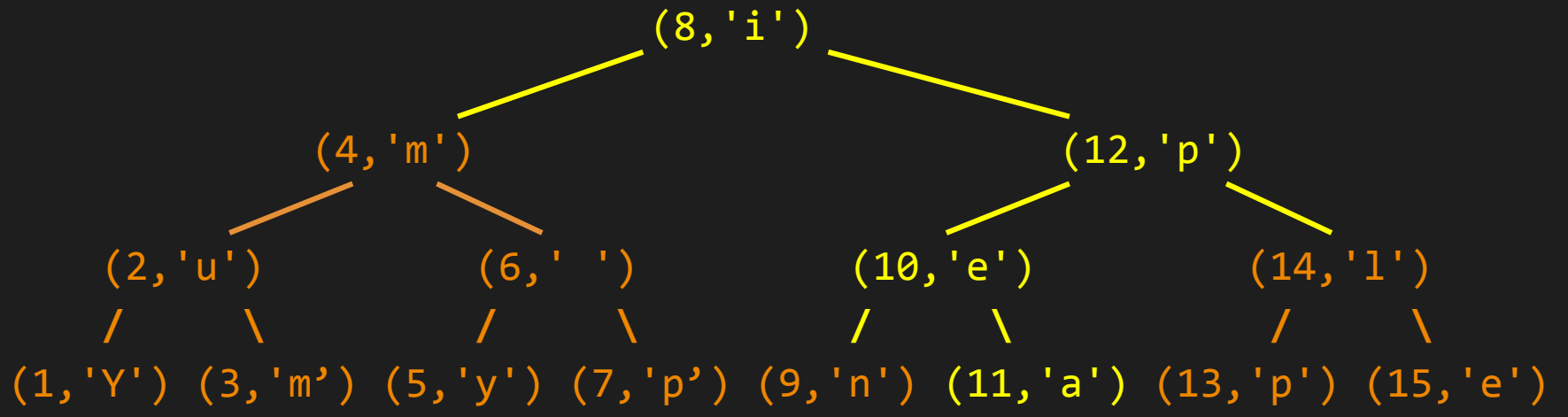
```
replace :: LookupTree a -> Integer -> a -> LookupTree a
replace Nil _ _ = Nil
replace (Node l (k,v) r) q c
 | q < k = Node (replace l q c) (k,v) r
 | q > k = Node l (k,v) (replace r q c)
 | otherwise = Node l (q,c) r

ghci> display $ replace t 11 'a'
Just "Yummy pineapple"

ghci> display $ replace t 8 'o'
Just "Yummy poneupple"
```

Replace only replaces a minimal path



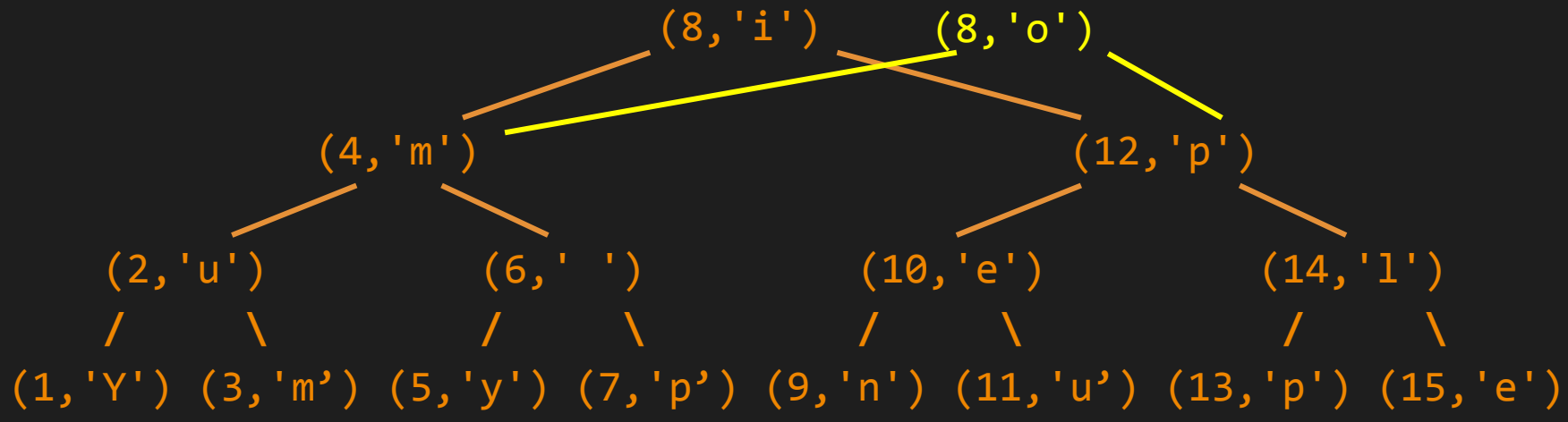

```
replace t 11 'a'
```

Replace only replaces a minimal path



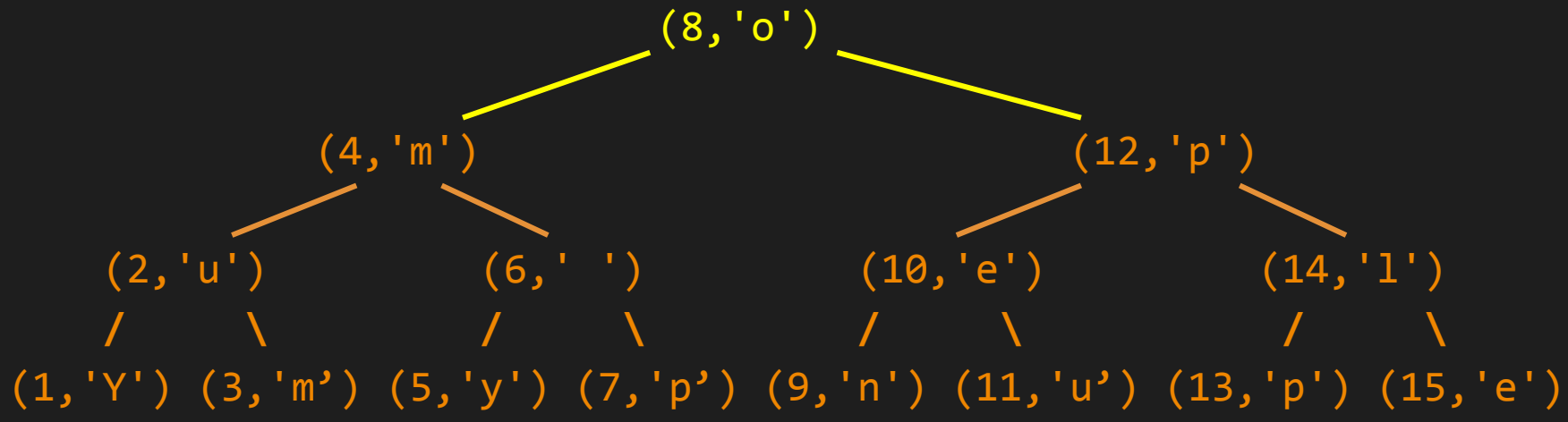

```
replace t 11 'a'
```

Replace only replaces a minimal path




```
replace t 8 'o'
```

Replace only replaces a minimal path




```
replace t 8 'o'
```

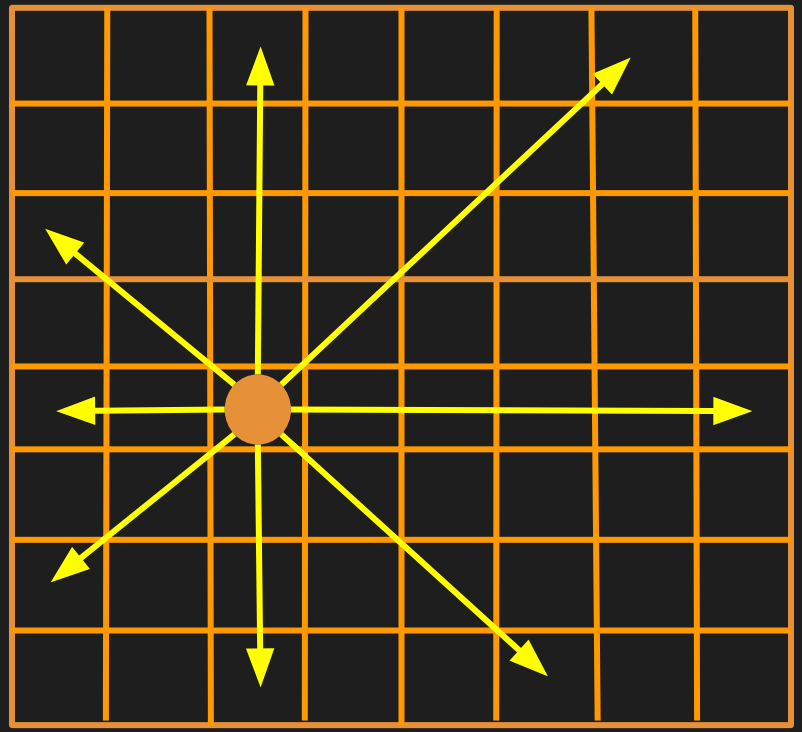
Using Applicative to work with a list of Maybe

```
-- |
-- >>> display t
-- Just "Yummy pineupple"
display :: LookupTree Char -> Maybe String
display t = sequenceA $ map (find t) [1 .. (count t)]
```

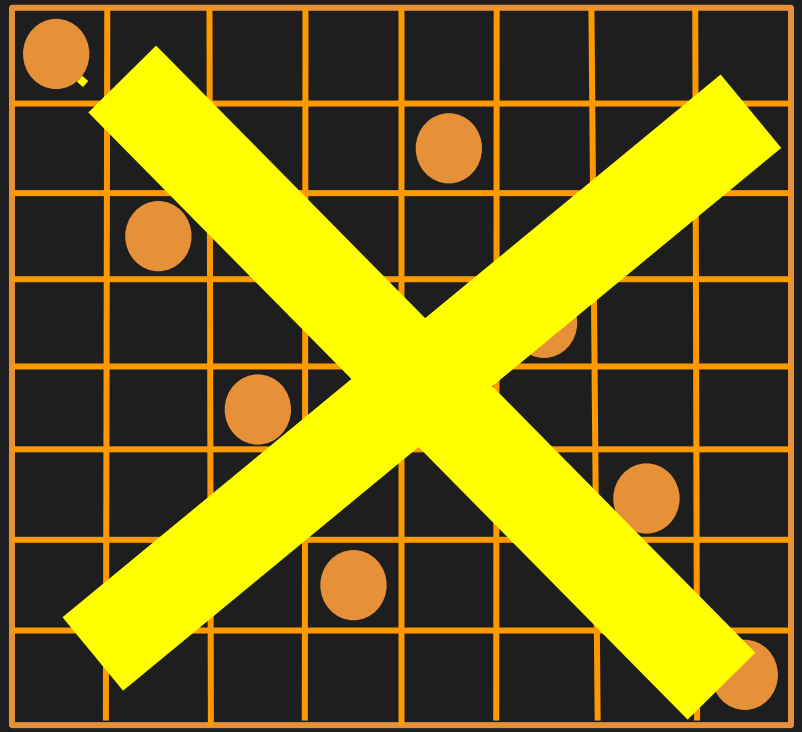
Traverse a function with an effect

```
display :: LookupTree Char -> Maybe String
display t = traverse (find t) [1 .. (count t)]
-- |
-- >>> display t
-- Just "Yummy pineupple"
```

Modelling problems in Haskell: n-queens



Modelling problems in Haskell: n-queens



Modelling n-queens

```
type Board = [Int]
```

```
0 -- . . . . . Q . .
1 -- . . Q . . . . .
2 -- Q . . . . . . .
3 -- . . . . . . Q .
4 -- . . . . Q . . .
5 -- . . . . . . . Q
6 -- . Q . . . . . .
7 -- . . . Q . . . .
```

```
[2,6,1,7,4,0,3,5]
```

Modelling n-queens

```
-- Q . . . Q . . .
-- . . . . . . . Q
-- . Q . . . . . .
-- . . . . . Q . .
-- . . Q . . . . .
-- . . . . . . Q .
-- . . . Q . . . .
-- . . . . . . . .
  [0,2,4,6,0,3,5,1]
```

Modelling n-queens

```
-- . . . . . Q . .
-- . . Q . . . . .
-- Q . . . . . . .
-- . . . . . . Q .
-- . . . . Q . . .
-- . . . . . . . Q
-- . Q . . . . . .
-- . . . Q . . . .
  [2,6,1,7,4,0,3,5]
```

Exercise 3

To be announced...

Recap: list comprehensions

Source for `instance Applicative []`:

```
pure x    = [x]
fs <*> xs = [f x | f <- fs, x <- xs] -- list comprehension
```

English: “the set (Haskell list) of all functions in `fs` applied to all values in `xs`”

Mathematical set builder notation: $\{f(x) \mid f \in fs \wedge x \in xs\}$

You can also put in conditions to filter by:

```
GHCi> [ (i,j) | i <- [1..5], j <- [1..5], i < j ]

[(1,2),(1,3),(1,4),(1,5),(2,3),(2,4),(2,5),(3,4),(3,5),(4,5)]
```

```haskell
nQueens n =
    let
        -- for a given set of starting columns,
        -- prepend all the possible first columns
        addColumns :: Board -> [Board]
        addColumns cols =
            filter check [r:cols | r <- [1..n]]

        -- generate k columns of feasible queens
        queens :: Int -> [Board] -> [Board]
        queens 0 s = s
        queens k s = queens (k-1) $ concatMap addColumns s
    in
        queens n [[]]
```

```haskell
-- list monad version
nqueens :: Int -> [Board]
nqueens n = extend n n []
extend :: Int -> Int -> Board -> [Board]
extend n 0 partial = return partial
extend n remaining partial = do
  -- "choose" a value.
  q <- [1..n]
  -- Glue it on the front.
  let extended = q:partial
  -- Make sure it's legal.
  guard $ check extended
  -- Recurse.
  extend n (remaining-1) extended
```

Conclusions

- Haskell is a very powerful and modern programming language that is still very much state of the art
- It has been a steep learning curve, but in many ways that is because Haskell is so expressive and concise that it makes higher-order programming a first class citizen
- You think in terms of powerful abstractions from the outset, rather than programming minute details
- The abstractions you have learned are applicable in any language (but still most elegant in a modern functional language)
- Although you have used them for relatively simple problems, the abstractions you have learned are applicable, and remain elegant in very complex situations