Week 9

S

University of the ley

Dr. Russell Campbell

Russell.Campbell@ufv.ca

COMP 481: Functional and Logic Programming

Overview

- Intro to Monads
- Tightrope Walking Simulation (Pierre)
- Banana on a Wire
- `do` Notation
- Pierre Returns
- Pattern Matching and Failure
- Th
- `M
- A Knights Quest
- Monad Laws
  - Left Identity
  - Right Identity
  - Associativity
- More Simplifications

Bind `>>=`

We have worked with implementing applicatives for various types so that:

- `Maybe a` values represent computations that may end up in failure

- `[a computational results (n

- `IO a` values represe ns with side effects

These can be facilitated with the special characters `>>=` as a binary operation between Monad values. This function is called bind.

Monad

Monads are a type class with similar behaviour as `Functors` and `Applicatives` to make functions work in context:

```
(>>=) :: (Monad m) => m a -> (a -> m b) -> m b
```


This

- to take an input va ontext `m a`
- a function that expects no input context `a ->`
- but the function returns a result `m b` with context when applied on the input `m a`

Context of `Maybe`

Recall how we mapped with functors:


```
ghci> fmap (++"!") (Just "wisdom")
Just
ghci
Nothing
```


- a value of `Nothing` as a result of such a mapping can be interpreted as a failure for some calculation

Context with Applicative

Applicative functors have the added context to the function as well:

```
ghci> Just (+3) <*> Just 3
Just 6
```



```
ghci                                       "
```



```
Noth
```



```
ghci> Just (ord) <*> Nothing
Nothing
```

- if either of the operands is `Nothing` it is propagated to the result

Applicative <$> and <*>

There was also the applicative style:


```
ghci                                        6
Just
```



```
ghci>  max <$> Just                    ng
Nothing
```

Monad Implementation

Now the implementation of the `Monad` type class:

```
class Applicative m => Monad m where

  (>>=) :: m a -> m b

  (>>) :: m a -> m b -> m b
  x >> y = x >>= \_ -> y
```

return

- the `return` function is the same as `pure` as we saw it in the `Applicative` type class
  - recently Haskell developers decided it would be a requirement to make any `Monad` also be a subclass of `Applicative`

  - not like in other programming s a value within the context of `m`

- the `>>` operation ha lementation that is rarely changed

- there also used to be a `fail` function, but that is no longer required to implement an instance of `Monad`

Maybe as a Monad

The `Maybe` type as an instance of `Monad`:

```
instance Monad Maybe where
    return x = Just x
    Nothing >>= f = Nothing
```


- if there is `Nothing` in -hand side of `>>=`, the expression evalua hing`

- otherwise, there is a nested value within `Just` and we can apply the function `f` to it
  - note that the result of `f` is in a context with a nested value that at least has the same type as `x`

Example
`Maybe` Monad

Now we give `Maybe` a try as a monad:

```
ghci> return "WHAT" :: Maybe String
Just "WHAT"
```



```
ghci                                        (x*10)
Just 90
```

```
ghci> Nothing >>= \x -> return (x*10)
Nothing
```

— Tig tion —

Using Monads

We will demonstrate one of the advantages of monads:

- change the behaviour of calculations in the way we desire
- co e in the computation
- we cannot do this wit alone, since they only lift computations into the nested context

Tightrope Walking

Suppose we have a man Pierre
that tightrope walks with a pole:

- birds land on either side of the

- if more than a difference of 3 b
either side, Pierre falls…

(to a safety net, of course)

Simulation of Birds

We will first implement a few types to help us keep track of the number of birds:

```
type Birds = Int
type Pole = (Birds, Birds)
```


Nex ulate birds
land

one:

```
landLeft :: Birds - le
```



```
landLeft n (left, right) = (left + n, right)

landRight :: Birds -> Pole -> Pole
landRight n (left, right) = (left, right + n)
```

Without Monads

We try out our functions without monads:

```
ghci> landLeft 2 (0, 0)
(2, 0)
ghci
(1,3
ghci> landRight (-1
(1,1)
```

- just use a negative number to simulate birds flying away

Order of Operations

Chain simulated birds landing by nesting operations:

```
ghci> landLeft 2 (landRight 1 (landLeft 1 (0, 0)))
(3,1)
```


- we can create a utility function to help us write more concisely:


```
x -: f = f x
```


- then we can write the parameter before the function, and rewrite our previous expression:

```
ghci> (0, 0) -: landLeft 1 -: landRight 1 -: landLeft 2
(3,1)
```

Maybe to Manage Failure (1)

So far, this does not check our condition for if Pierre will fall

- if irds th d pair

- instead, we would lik Pierre's failure, so we use `Maybe`

Maybe to Manage Failure (2)

Implement new versions of our bird-landing simulation functions:


```
landLeft :: Birds -> Pole -> Maybe Pole
landLeft n (left, right)
  | a                                   = Just (left + n, right)
  | T                                   = Nothing

landRight :: Birds -             be Pole
landRight n (left, right)
  | abs (left - right - n) < 4   = Just (left, right + n)
  | True                         = Nothing
```

Simulating Imbalance

- Our implementation will maintain a difference of three for the number of birds on either side of the pole

- if > 3, the result of any `landLeft` or `landRight` will be `Nothing` to indicate imbalance and represent falling

- since these versions of our functions:

  - but a `Maybe Pole`

  - we will need to ma
  - to apply successive operations together

```
ghci> return (0, 0) >>= landLeft 1 >>= landRight 1 >>= landLeft 2
Just (3,1)
```

Using >>=

```
return (0, 0) >>= landLeft 1 >>= landRight 1 >>= landLeft 2
```

- note that we had to begin the calculation with the context of a monad, so we used `return`
- the `return` function can be used no matter the specific application of a monad context for a sequence of calculations

Fina


```
ghci> return (0, 0) >>= landLeft 1 >>= landRight 4
        >>= landLeft (-1) >>= landRight (-2)
Nothing
```

- can you tell at which point the pole became imbalanced?

Know the Monad

Try to make sure you do not conflate the context of the monad with the functions used.

- unctions `landLeft` and `landRight`

- the functions have ited to take advantage of ` ad

Banana Slip

We implement more functions that can combine with the other computations we have designed for simulation:

- suppose a banana on the wire could slip Pierre while walking
- this automatically forces Pierre to fall


```
bana
```



```
banana _ = Nothing
```


It is fairly clear what will happen when we use this function:

```
ghci> return (0, 0) >>= landLeft 1 >>= banana >>= landRight 1
Nothing
```

Changing Default >>

To ignore a monadic value on the *right* and *return the left* value, we can adjust the `>>` operation from its default:

```
(>>) :: (Monad m) => m a -> m b -> m a
n >> m = m >>= \_ -> n
```


Otherwise, the default is to ignore the *left* value and *retu* ent to the `do` block:


```
do
  n
  m
```

for monad values `n` and `m`, the above returns `m`.

Carrying Monads Forward

```
ghci> Nothing >> Just 3
Nothing

ghci> Just 3 >> Just 4
Just 4
```



```
ghci
```



```
Nothing
```


(keep in mind the above demonstrates the default implementation!)

Thus, we can omit having to write a `banana` function,
and just use `>> Nothing` to the same effect.

>>= with Lambdas

We can use monad-style expressions with lambdas:

```
ghci> Just 3 >>= (\x -> Just (show x ++ "!"))
Just "3!"
```

- monadic value `Just 3` has its nested `3` passed as input into the lambda on the right side

- a "3!"`

The above expressi written as two nested `>>=` operations:


```
ghci> Just 3 >>= (\x -> Just "!" >>= (\y -> Just (show x ++ y)))
Just "3!"
```

Notice >>= "binds" an unwrapped value to the parameter.

Binding and Nesting

The expression can be rewritten as two nested `>>=`:

```
ghci> Just 3 >>= (\x -> Just "!" >>= (\y -> Just (show x ++ y)))
Just "3!"
```


Noti ped value to the parameter.

- thi

```
let
```



```
    x = 3;
    y = "!"
in show x ++ y
```

Helpful But Less Readable

The advantage of the more elaborate version:

- we get monads to help manage context
- at each part of the calculation
- without needing to explicitly write code at each stage to deal with it

```
ghci> Nothing >>= (\x -> Just "!" >>= (\y -> Just (show x ++ y)))
Noth

ghci                                >>= (\y -> Just (show x ++ y)))
Nothing

ghci> Just 3 >>= (\x -> Just "!" >>= (\y -> Just Nothing))
Just Nothing
```


- at each point, the value could instead be `Nothing`, and the result is dealt with appropriately without error

Organized as a Function

We move toward a nicer syntax available, first, in the form of a function:


```
:set +m

let
foo :: Maybe String
foo = Just 3
    >>= (\x -> Just "!"
    >>= (\y -> Just (show x ++ y)
    ))
```

Maybe Context (1)

- there is an alternative cleaner syntax available with the `do` block


```
foo

foo
    x <- Just 3
    y <- Just "!"
    Just (show x ++ y)
```

Maybe Context (2)

```
foo :: Maybe String
foo = do
    x <- Just 3
    y <- Just "!"
    Just (show x ++ y)
```

- the `do` block allows a different way to chain monadic calculations into one monadic calculation

- if a Nothing` then th ill be `Nothing`

- lines that are not mo e to be in a `let` expression

- we use `<-` assignment to obtain a nested value (bind)
  - if we have a Just "!" monadic value, the nested value is "!" as a `String` type
  - if we have a `Just 3` monadic value, the nested value is `3` as a numeric type
- the last line of a `do` block cannot use `<-`, since this would not make sense as the result returned for a monadic expression

Typical Do Block

The typical design:

- to compute and assign nested values

- and return them ined expression

- within the mona

Equivalent Examples

One more small example:

```
ghci> Just 9 >>= (\x -> Just (x > 8))
Just True

let
mary
marySue = do
    x <- Just 9
    Just (x > 8)

ghci> marySue
Just True
```

Review

(Simranjit Singh)

```
-- various types of addition

-- infix (any func that's a special symbol is automatically infix)
1 + 2

-- prefix
(+) 1 2
```



```
-- f
fmap (+1) [1,2,3]
(+1) <$> [1,2,3]

-- applicative functor
[(+1)] <*> [1,2,3]
[(+)] <*> [1] <*> [1,2,3]
pure (+) <*> [1] <*> [1,2,3]
(+) <$> [1] <*> [1,2,3]
```

Examples

(Simranjit Singh)

```
-- monads
[1] >>= \x -> return (x+1)
[1,2,3] >>= \x -> return (x+1)

-- nested >>= operations
[1,2                        )   >>= \y -> return (y+1)

-- alternative do b
do
    x <- [1,2]
    y <- [3,4,5]
    return $ x + y + 1
```

Simulation with Do Block

We can rewrite our previous example of Pierre's tightrope walking with a simulation for birds landing on a pole.

We now design it in a `do` block:

```
routine :: Maybe Pole
routine = do
    start <- return (0, 0)
```



```
    landLeft 1 second
```

```
ghci> routine
Just (3,2)
```

- each line of a `do` block depends on the success of the previous one

Nested Cases

Without monads, this issue can be seen differently where computation would have to be *nested*:

```
routine :: Maybe Pole
routine = case Just (0, 0) of
    Nothing -> Nothing
                                    ft 2 start of

        Just first                    Right 2 first of
            Nothing
            Just second -> landLeft 1 second
```


- the ghci session will issue a warning with the above code, but you should still be able to issue `routine`

Nothing Overwrites Results

Then if we want to throw in a banana peel like we did before:

```
routine :: Maybe Pole
routine = do
    start <- return (0, 0)
    first <- landLeft 2 start
    second <- landR
    landLeft 1 seco
```


- the line with `Nothing` does not use `<-`, much like our use of `>>` to ignore a previous monadic value
  - this is nicer than needing to write equivalently `_ <- Nothing`

>>= vs Do

It is up to you whether you want to use `>>=` versus `do` blocks, but in general:

- to `>>=`
- to results, use `do` blocks

Neither is exclusively needed to accomplish the above…

— Pa ilure —

Bind with Pattern Matching

Pattern matching can be used on a binding:

```
justH :: Maybe Char
justH = do
```


- the above grabs the first letter of the string "hello"
- the `justH` function evaluates to `Just h`
  - remember, the left value of a `:` operation is a `Char`, not a singleton

Failing a Pattern Match

When a pattern match fails within a function:

- the next pattern is attempted
- if matching falls past all patterns, the function throws an error

With `let` expressio occurs on failure of matching because there is no falling mechanism for matching further patterns.

Implementing `fail` Function

When matches fail within a `do` block:

- the context of the monad often implements a `fail` function
- to deal with the issue in its context
- as we have seen with the `Maybe` type

- this used to be implemented as part of a default `Monad` function
- it is now dealt with as an instance of the `Monad` type with a custom implementation of `fail` per each type

For [illegible] `Monad`, we can implement:


```
fail :: Maybe a ->
fail _ = Nothing
```


- but `fail` is a default function to throw an error with String message
- then when all patterns fall through unmatched within a `do` block, the function expression will evaluate to `Nothing` *instead of crashing*

Example of `fail`

```
wopwop :: Maybe Char
wopwop = do
    (x:xs) <- Just ""
    return x
```



```
ghci
```



```
Nothing
```


- there is only a failure mitigated within the context of monad `Maybe`
- there is no program-wide failure

Lists as Monads

Recall that we can do nondeterministic calculations with lists using the applicative style:

```
ghci> (*) <$> [1,2,3] <*> [10, 100, 1000]
[10,100,1000,20,200,2000,30,300,3000]
```


Let tation of `Monad` for lists:


```
inst
    return x = [x]
    xs >>= f = concat (map f xs)
    fail _ = []
```


- `return` just puts the input value within minimal list contex, i.e.: a singleton `[x]`

List Context

The function `concat` might seem not to fit the context, but we want to implement nondeterminism.

```
ghci> [3,4,5] >>= \x -> [x,-x]
[3,-3,4,-4,5,-5]
```


- as sults of `[3,4,5]` fe wn as one conjoined list

The `>>=` operation `[]`:

```
ghci> [] >>= \x -> ["bad", "sad"]
[]
ghci> [1,2,3] >>= \x -> []
[]
```

Chaining

`>>=`

It is possible to chain `>>=` operations to propagate the nondeterminism:

```
ghci> [1,2] >>= \n -> ['a','b'] >>= \ch -> return (n, ch)
[(1,'a'),(1,'b'),(2,'a'),(2,'b')]
```


- no shows up as part of the final expression after the ation

  - remember, each next `>>=` operation is nested as part of the previous one
- `return` places each pair within a singleton context
- all the pairs are concatenated together into one flat list

Using Chaining

Describing the propagation of nondeterministic operations:

- "for all" elements in `[1,2]` should be paired
- with every element of `['a','b']`.

The previous expression could be written in a `do` block, but [illegible] ng syntax easier to read:


```
:{
[1,2]
    >>= \n -> ['a','b']
    >>= \ch -> return (n, ch)
:}
```

```
[(1,'a'),(1,'b'),(2,'a'),(2,'b')]
```

Chaining in a `do` Block

Otherwise, in a module, I would use a `do` block:

```
listOfTuples :: [(Int, Char)]
listOfTuples = do
    n <- [1,2]
    ch <- ['a','b']
    return (n, ch)
```



```
ghci> listOfTuples
[(1,'a'),(1,'b'),(2 [illegible] )]
```


- these syntax make the nondeterminism clearer to keep track of
  - `n` takes on every value of `[1,2]`
  - `ch` takes on every value of `['a','b']`

Similar to List Comprehension

Lastly, we had originally learned list comprehension to do essentially the same thing as above:

```
ghci> [ (n, ch) | n <- [1,2], ch <- ['a','b'] ]
[(1,'a'),(1,'b'),(2,'a'),(2,'b')]
```


- the `<-` notation wor the same, to handle the nondeterministic com
- we did not need to use the `return` function because list comprehension takes care of that for us
- documentation typically calls alternatives such as this syntactic sugar for the more formally written expressions

— `Monad `Function —

Monad Filtering

List comprehension can apply filtering with a conditional expression:

```
ghci> [ x | x <- [1..50], '7' `elem` show x ]
[7,17,27,37,47]
```


The r implementing filtering.

- t as monoids


```
class Monad m => Mo ere
    mzero :: m a
    mplus :: m a -> m a -> m a
```

- `mzero` is synonymous with `mempty` from `Monoid`
- `mplus` corresponds to `mappend`

MonadPlus

We know lists are both monads as well as monoids, so:

```
instance MonadPlus [] where
```


- a failed computation for lists is an empty list
- `mplus` concatenates two nondeterministic computational results

Filtering (1)

There is also a `guard` function that helps perform filters:

```haskell
import Control.Monad


guard :: (MonadPlus m) => Bool -> m ()
guard True = pure ()
guard False = mzero
```


- a Boolean expression [illegible]`uard` as the test to either create a dummy value or nothing (`mzero`)
- empty tuple `pure ()` is used as a dummy and used to then filter
  - input into `>>` operations on the left side, it will either keep or throw away the right-hand side values

Filtering (2)

```
ghci> import Control.Monad

ghci> guard (5 > 2) >> return "cool" :: [String]
["co

ghci> guard (1 > 2)              cool" :: [String]
[]
```

Using guard

There are two ways we can write the use of `guard` in order to filter as in the list comprehension:

- the first is with nested `>>=` expressions
- the second is within a `do` block

```
ghci> [1..50] >>= (\x -> guard ('7' `elem` show x) >> return x)
[7,17,27,37,47]
```



```
let
sevensOnly :: [Int]
sevensOnly = do
    x <- [1..50]
    guard ('7' `elem` show x)
    return x

ghci> sevensOnly
[7,17,27,37,47]
```

Examples

(David Semke)

```haskell
import Control.Monad

-- Using list1, create all possible pairs (x, y)
--   such that x is always greater than y

list1 = [1, 2, 3, 4, 5]

listCompPairs =
  [(x                                    > y]
```



```haskell
nestedMethodPairs =
  list1 >>= (\x -> list1              d (x > y) >> return (x, y)))

doMethodPairs = do
    x <- list1
    y <- list1
    guard (x > y)
    return (x, y)
```

Simulating Knights in Chess

We would like to simulate on a chess board, a knight which has a restricted `L` move each turn.

Are they able to reach a square within three turns?

- the image below shows the positions in one turn where a knight piece could choose to move
  - it should be symmetrical, and there are two spots missing behind the first row, but the pieces cannot move off the board

Pairs for Positions

We use a pair to keep track of the row and the column

- the first number gives the row
- the second number gives the column

type

So, if the knight starts a 2)`, can they move to `(6, 1)`?

- we might wonder which is the best move to choose toward the goal
- instead, we just let nondeterminism try all of the moves

moveKnight

```haskell
moveKnight :: KnightPos -> [KnightPos]
moveKnight (c, r) = do
    (c', r') <- [
                (c+2, r-1), (c+2, r+1), (c-2, r-1), (c-2, r+1),
                (c+1, r-2), (c+1, r+2), (c-1, r-2), (c-1, r+2)
            ]
    guard (c' `elem` [1..8] && r' `elem` [1..8])
```



```
ghci> moveKnight (6, 2
[(8,1),(8,3),(4,1),(4, [illegible] )]
ghci> moveKnight (8, 1)
[(6,2),(7,3)]
```

- we can filter the new positions with use of `guard`

`in3` Possibilities

Next, we can use this to write a concise function to move three times:

```
in3 :: KnightPos -> [KnightPos]
in3 start = do
    first <- moveKnight start
                                      t
```


Passing in `(6, 2)` generates a fairly long list:

```
ghci> in3 (6, 2)
```

(results omitted for space)

Using `in3` Output

We could rewrite the `in3` function using `>>=` notation:

```
in3 start =
 return start >>= moveKnight >>= moveKnight >>= moveKnight
```


- the `return` puts `start` within the co m)

Finally, we can test for)` is an element in the result:


```
ghci> (6, 2) `elem` in3 (6, 1)
True
ghci> (6, 2) `elem` in3 (7, 3)
False
```

Extending Chess Simulation

We could write the previous movement testing as a function and pass in the start and end positions.

- (the next chapter shows how to modify the above as a function th le moves to take)

- we could also specify ves in general as input, not just three moves,

Monad Laws

Each rule expects two equivalent expressions:

— Left Identity —

- `return x >>= f`
- `f x`

dentity —

- `m >>= return`

- `m`

— Associativity —

- `(m >>= f) >>= g`
- `m >>= (\x -> f x >>= g)`

Left Identity

- `return x >>= f`
- `f x`

Remember, that in the situation of monads, the function `f` will result in a value with context.

- note that `return` wraps with that context, and `>>=` removes context to pass the nested value to `f`


```
f ::
f x
```



```
ghci> return 3 >>=
Just 100003

ghci> f 3
Just 100003
```

Right Identity

- `m >>= return`
- `m`

Consider right side of first expression:

- function `return` takes a value and wraps it in a minimal context
- for `Maybe` type, minimal context "does not introduce any failure"
- for ot introduce extra nondeterminism"

With lists, say if we]` into `return` with `>>=`:

- first, every element of the list gets wrapped, to get `[[1],[2],[3]]`
- the elements concatenate with `(++)` applied to result in `[1,2,3]`

Associativity

- `(m >>= f) >>= g`
- `m >>= (\x -> f x >>= g)`

The order that operations executed in a sequence should not matter.

The functions `f` and `g` rst

- but the notation for lambda expression is the resu d function,

- instead of something a bit more concise as in mathematics as with $(g \circ f)$ (yes, the order is correct with the above monad law)
- but notice Haskell syntax makes sense for order of execution when we are writing our code

Chaining and Associativity

Recall that we had simulated tightrope walking, and chained `>>=` expressions as with the law of associativity:

```
pure (0, 0) >>= landRight 2 >>= landLeft 2 >>= landRight 2
Just (2,4)
```


- th ith `>>=` before

- it r ` at the start of a block of code

The law of associativity allows us to drop parentheses, but with parentheses, we have:

```
((pure (0, 0) >>= landRight 2) >>= landLeft 2) >>= LandRight 2
Just (2,4)
```

Multiline

But we can also write the expression as:

```
:{
pure (0, 0)
 >>= (\x -> landRight 2 x
 >>=
 >>=
 )))
:}
Just (2,4)
```

- each successive function is further nested in parentheses

Flipping with `<=<`

At least the law of associativity allows us to be very concise and avoid excessive use of parentheses.

The following operation flips use of `>>=` for nesting functions that work with monads together.

```
(<=<) :: (Monad m) => (b -> m c) -> (a -> m b) -> (a -> m c)
f <=< g = (\x -> g x >>= f)
```


in `Control.Monad`

- it helps establish ws for Monads
  - the function `f` takes `b -> m c`
  - the function `g` takes `a -> m b`
- the problem is `g` outputs values of type the same as input for `f`, but monadic
- so `<=<` helps manage their composition

Associativity of `<=<`

Recall that the following are equivalent (associativity we should implement for `>>=`):

- `(m >>= f) >>= g`
- `m >>= (\x -> f x >>= g)`

The e following are equivalent:

- `(f <=< g) <=< h`
- `f <=< (g <=< h)`

Then we can also omit parentheses with chaining `<=<`.

But, you have to implement `>>=` properly!

More Simplifications

Translating left identity laws:

- `return x >>= f`
- `f x`

For y the following:

- `f <=< return` is th `f`

So, for right identity, `return <=< f` is also the same as `f`.

Helpful Resources

For knowing exactly what thing you are working with and its corresponding documentation (like, which package?):

`:info <name_of_thing>`

A gr r handling:

Gab ogger)

https://www.haskellforall.co t-to-avoid-deeply-nested-error.html

Working with json style object initialization from files:

- grab the `json.zip` file from Blackboard
    - you will need to install the yaml package, but there are notes to help with the edits from the article

Names for Binary Operations

$	(none, just as " " [whitespace])	
->	to	a -> b: a to b
.	pipe to	a . b: "b pipe-to a"
<$>	(f)map	
<*>	ap(ply)	(as it is the same as Control.Monad.ap)
>>=	bind	
<-	bind	(as it desugars to >>=)
>>	then	
!!	index	
[]	empty list	
:	cons	
\	lambda	
@	as	go ll@(l:ls): go ll as l cons ls
::	of type / as	f x :: Int: f x of type Int

(others we have not covered)

*>	then (evaluates to right hand functor, unless left mempty)	
<$	map-replace by (e.g.: 3 <$ [2] evaluates to [3])	0 <$ f: "f map-replace by 0"
<|>	or / alternative (import Control.Applicative)	expr <|> term: "expr or term"
	or irrefutable pattern) //en.wikibooks.org/wiki/Haskell/Laziness#Lazy_pattern_matching)	
!	(use in signatures) (causes pattern matching errors even for _)	

https://stackoverflow.com/questions/7746894/are-there-pronounceable-names-for-common-haskell-operators

Why Did We Learn Haskell?

https://crypto.[illegible]haskell/why.html

Thank You!

Questions?