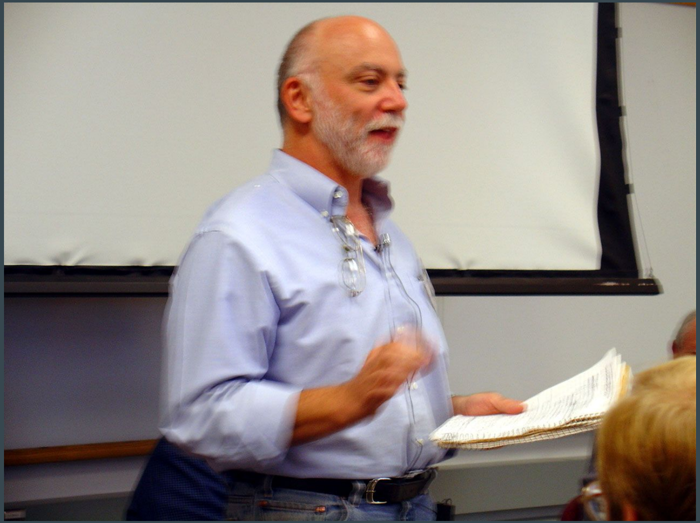


To iterate is human,
To recurse divine

L. Peter Deutsch

Today’s Superpower

No more loops!

But First...

Recursion

What is it?

Recursion

Informally:

- Something defined in terms of itself
- Thing where parts are instances of the thing

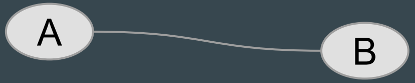
Applies to:

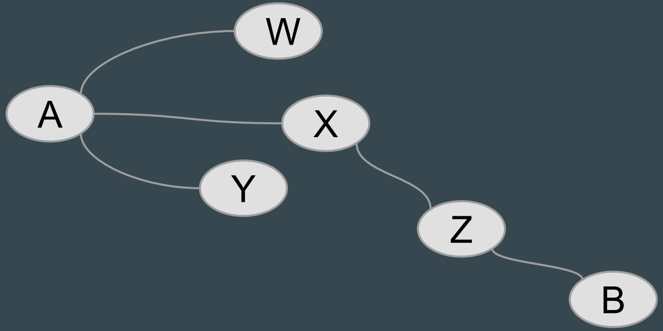
- Code
- Data
- Both at the same time

Recursion

A is connected to B if:

- There’s a wire from A to B, or
- There’s a wire from A to $\boldsymbol{x}$, and $\boldsymbol{x}$ is connected to B

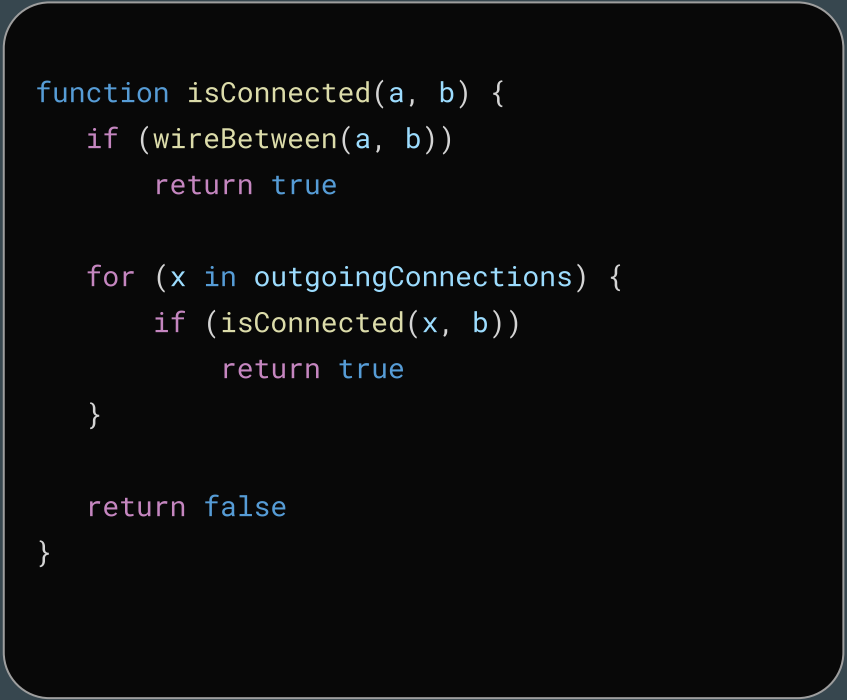




Recursion

A is connected to B if:

- There's a wire from A to B, or
- There's a wire from A to x, and x is connected to B

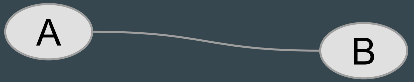


```
function isConnected(a, b) {
   if (wireBetween(a, b))
       return true

   for (x in outgoingConnections) {
       if (isConnected(x, b))
           return true
   }

   return false
}
```

A **is connected to** B if:

- There’s a wire from A to B, or
- There’s a wire from A to ***x***, and ***x*** **is connected to** B

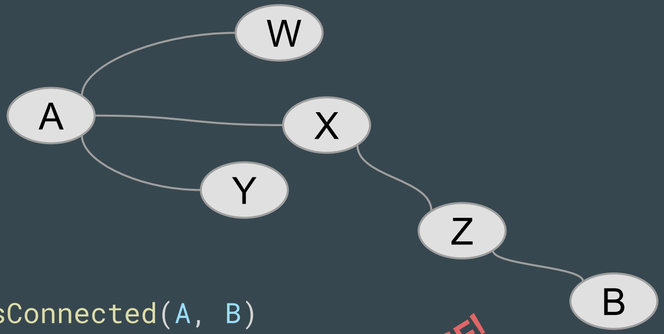
```
function isConnected(a, b) {
   if (wireBetween(a, b))
       return true

   for (x in outgoingConnections) {
       if (isConnected(x, b))
           return true
   }

   return false
}
```



```
isConnected(A, B)

     if (wireBetween(a, b))
            return true
```

TRUE!

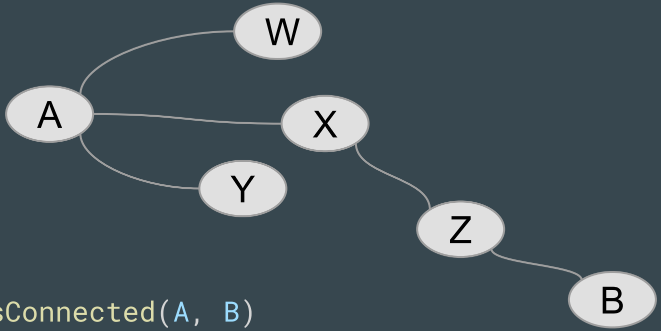
```
function isConnected(a, b) {
    if (wireBetween(a, b))
        return true

    for (x in outgoingConnections) {
        if (isConnected(x, b))
            return true
    }

    return false
}
```



```
isConnected(A, B)
      wireBetween(A, B)
```

FALSE!

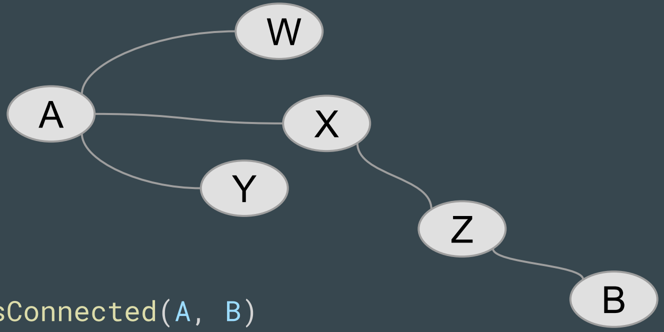
```
function isConnected(a, b) {
   if (wireBetween(a, b))
       return true

   for (x in outgoingConnections) {
       if (isConnected(x, b))
           return true
   }

   return false
}
```



```
isConnected(A, B)
      wireBetween(A, B)
      x = W
      isConnected(W, B)
            wireBetween(W, B)
```

FALSE!

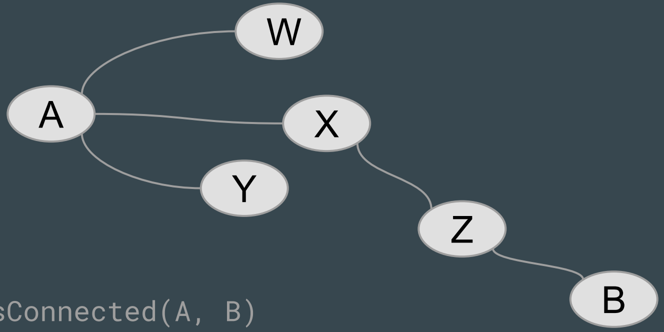
```
function isConnected(a, b) {
   if (wireBetween(a, b))
       return true

   for (x in outgoingConnections) {
       if (isConnected(x, b))
           return true
   }

   return false
}
```



```
isConnected(A, B)
      wireBetween(A, B)
      x = W
      isConnected(W, B)
            wireBetween(W, B)
            x = no outgoing connections
            return false
```

FALSE!

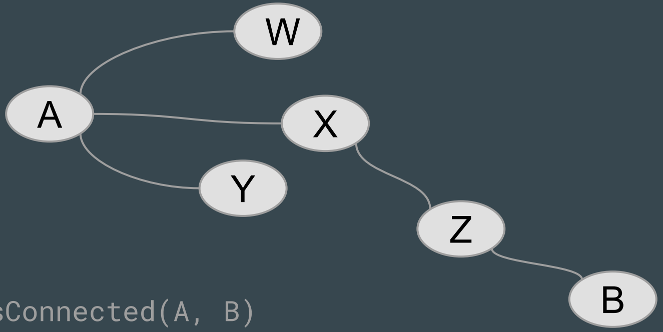
```
function isConnected(a, b) {
    if (wireBetween(a, b))
        return true

    for (x in outgoingConnections) {
        if (isConnected(x, b))
            return true
    }

    return false
}
```



```
isConnected(A, B)
    wireBetween(A, B)
    x = W
    isConnected(W, B)
        wireBetween(W, B)
        x = no outgoing connections
        return false
    x = X
    isConnected(X, B)
        wireBetween(X, B)
```

FALSE!

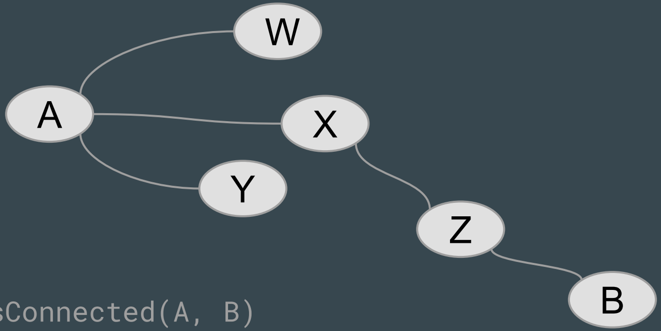
```
function isConnected(a, b) {
   if (wireBetween(a, b))
       return true

   for (x in outgoingConnections) {
       if (isConnected(x, b))
           return true
   }

   return false
}
```



```
isConnected(A, B)
      wireBetween(A, B)
      x = W
      isConnected(W, B)
            wireBetween(W, B)
            x = no outgoing connections
            return false
      x = X
      isConnected(X, B)
            wireBetween(X, B)
            x = Z
            isConnected(Z, B)
                  wireBetween(Z, B)
                  return true
```

TRUE!

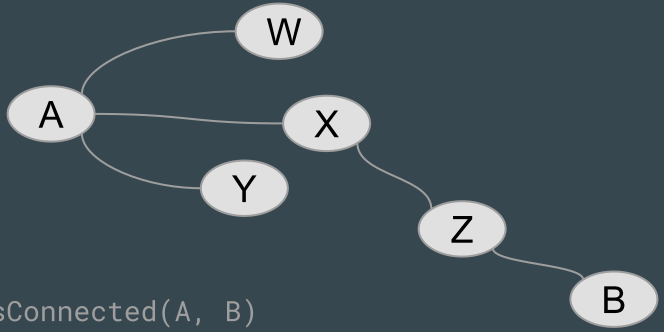
```
function isConnected(a, b) {
   if (wireBetween(a, b))
       return true

   for (x in outgoingConnections) {
       if (isConnected(x, b))
           return true
   }

   return false
}
```



```
isConnected(A, B)
      wireBetween(A, B)
      x = W
      isConnected(W, B)
            wireBetween(W, B)
            x = no outgoing connections
            return false
      x = X
      isConnected(X, B)
            wireBetween(X, B)
            x = Z
```

TRUE!

```
function isConnected(a, b) {
   if (wireBetween(a, b))
       return true

   for (x in outgoingConnections) {
       if (isConnected(x, b))
           return true
   }

   return false
}
```



isConnected(A, B)

TRUE!

Recursion

Slightly more formally, recursion has:

- A base case
  which requires no more recursion
- A recursive step

For recursion to terminate, the recursive step must take you closer to the base case.

A **is connected to** B if:

- There's a wire from A to B, or
- There's a wire from A to x, and x **is connected to** B

Recursion

Slightly more formally, recursion has:

- A base case
  which requires no more recursion
- A recursive step

For recursion to terminate, the recursive step must take you closer to the base case.

A **is connected to** B if:

- There’s a wire from A to B, or
- There’s a wire from A to x, and x **is connected to** B and we have not already visited x.

Recursion

Is everywhere:

- A base case, which requires no more recursion
- A recursive step

Natural numbers (ℕ):

Base:	0 is in ℕ
Recursive:	if x is in ℕ, then x+1 is in ℕ

Recursion

Is everywhere:

- A **base case**, which requires no more recursion
- A **recursive step**

Sum of numbers up to *n*:

Base:	sumUpTo(0) is 0
Recursive:	sumUpTo(*n*) is *n* + sumUpTo(n-1)

Recursion

Is everywhere:

- A base case, which requires no more recursion
- A recursive step

Definition of a list:

Base:	The constant [ ] is an (empty) list
Recursive:	A value concatenated to a list is a list

Recursion

Is everywhere:

- A **base case**, which requires no more recursion
- A **recursive step**

Length of a list:

Base:	The length of [ ] is 0
Recursive:	The length of a list with a first value and a trailing list is 1 + the length of the trailing list

Haskell

Sum of numbers up to *n*:

Base: sumUpTo(0) is 0

Recursive: sumUpTo(*n*) is *n* + sumUpTo(n-1)

```haskell
sumUpTo 0 = 0
sumUpTo n = n + sumUpTo (n-1)
```

Haskell

Length of a list:

Base:	The length of [ ] is 0
Recursive:	The length of a list with a first value and a trailing list is 1 + the length of the trailing list

```
len [] = 0
len (hd : rest) = 1 + len rest
```

Quick lab...

Haskell

Sum of a list:

Base:	???
Recursive:	???

Haskell

Sum of a list:

Base:	The sum of [ ] is 0
Recursive:	The sum of a list with a first value *n* and a trailing list is *n* + the sum of the trailing list

This is what “length of a list” looked like

```
len [] = 0
len (hd : rest) = 1 + len rest
```

Haskell

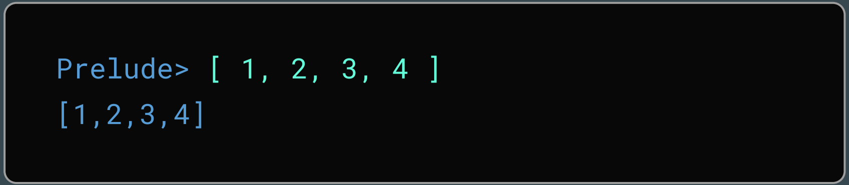
Sum of a list:

Base:	The sum of [ ] is 0
Recursive:	The sum of a list with a first value *n* and a trailing list is *n* + the sum of the trailing list

```
sum [] = 0
sum (hd : rest) = hd + sum rest
```

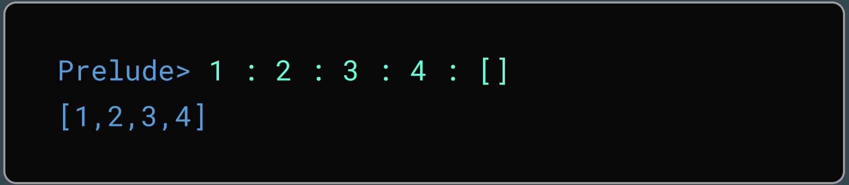
Haskell Lists

We’ve seen this style:

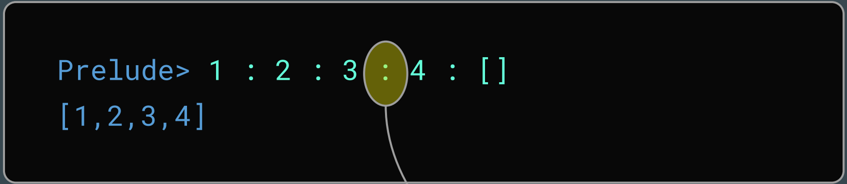


```
Prelude> [ 1, 2, 3, 4 ]
[1,2,3,4]
```

It’s a shortcut for this:



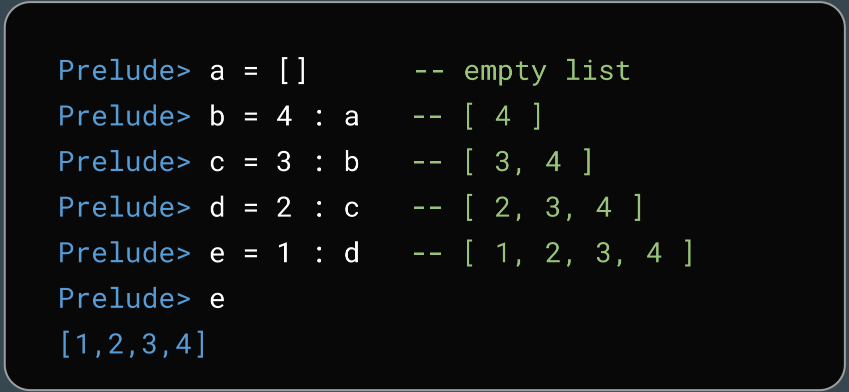
```
Prelude> 1 : 2 : 3 : 4 : []
[1,2,3,4]
```



```
Prelude> 1 : 2 : 3 : 4 : []
[1,2,3,4]
```

:

Joins an element to the start of a list



```
Prelude> a = []       -- empty list
Prelude> b = 4 : a    -- [ 4 ]
Prelude> c = 3 : b    -- [ 3, 4 ]
Prelude> d = 2 : c    -- [ 2, 3, 4 ]
Prelude> e = 1 : d    -- [ 1, 2, 3, 4 ]
Prelude> e
[1,2,3,4]
```

Pattern Matching

```
Prelude> [ 1, 2, 3 ]
```

Is the same as

```
Prelude> 1 : 2 : 3 : []
```

```
len [] = 0
len (x:xs) = 1 + len xs
```

```
Prelude> len [ 1, 2, 3 ]
Is the same as
Prelude> len 1 : 2 : 3 : []
so

len  1 : 2 : 3 : []
len (x : xs) = 1 + len xs
```

```
sumList [] = 0
sumList (x:xs) = x + sumList xs
Prelude> sumList [ 1, 2, 3 ]
```

```
sumList [ 1, 2, 3 ]

sumList 1 : 2 : 3 : []

1 + sumList 2 : 3 : []
1 + 2 + sumList 3 : []
1 + 2 + 3 + sumList []
1 + 2 + 3 + 0
6
```

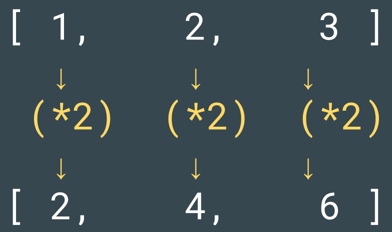
It's as if the function is crawling down the list, leaving a result in its wake... Remember this for next week.

```
double [] = []
double (x:xs) = 2*x : double fn xs

Prelude> double [ 1, 2, 3 ]
```

```
double [ 1, 2, 3 ]

double 1 : 2 : 3 : []

2 : double (2 : 3 : [])
2 : 4 :  double (3 : [])
2 : 4 : 6 : double []
2 : 4 : 6 : []
```

[2, 4, 6]

*It's as if the function is applying (*2) to each element to create a new list.*



```
[ 1,    2,    3 ]
  ↓     ↓     ↓
 (*2)  (*2)  (*2)
  ↓     ↓     ↓
[ 2,    4,    6 ]
```

Ligatures

ff → ﬀ

fi → ﬁ

fl → ﬂ

…

diﬃcult to

ﬁnd ﬂuid

You can pattern match on multiple elements at the start of a list

```haskell
convertToLigatures:: String -> String

convertToLigatures [] = []
convertToLigatures ('f' : 'f' : 'i' : rest) = 'ﬃ'   : convertToLigatures rest
convertToLigatures ('f' : 'f' : 'l' : rest) = 'ﬄ'   : convertToLigatures rest
convertToLigatures ('f' : 'f' : rest)       = 'ﬀ'   : convertToLigatures rest
convertToLigatures ('f' : 'i' : rest)       = 'ﬁ'   : convertToLigatures rest
convertToLigatures ('f' : 'l' : rest)       = 'ﬂ'   : convertToLigatures rest
convertToLigatures ('i' : 'i' : rest)       = 'ĳ'   : convertToLigatures rest
convertToLigatures (other : rest)           = other : convertToLigatures rest

main = do
   putStr $ convertToLigatures "difficult to find fluid wiffle"
```

diﬃcult to ﬁnd ﬂuid wiﬄe

To Recurse is Divine

- A lot of data can be recursively defined
- Recursive code makes to easy to handle this data
- It also eliminates many of the problems with loops for other tasks

But…

- But there’s a lot of boilerplate code.
- Next week: factor that out; even more functional.

Homework...

Due Tuesday, May 4

Fork and clone the repo https://github.com/Eastside-FP/dayfour You'll be working on code the the assignment/ directory, and submitting using a pull request.

Fork and clone the repo https://github.com/Eastside-FP/dayfour. You'll be working on code the the assignment/ directory, and submitting using a pull request.

In that directory, you'll find the file assignment.hs.

It contains placeholders for code that you'll write, along with a set of tests.

Update the placeholders to make them do what the description says, and so that the tests pass.