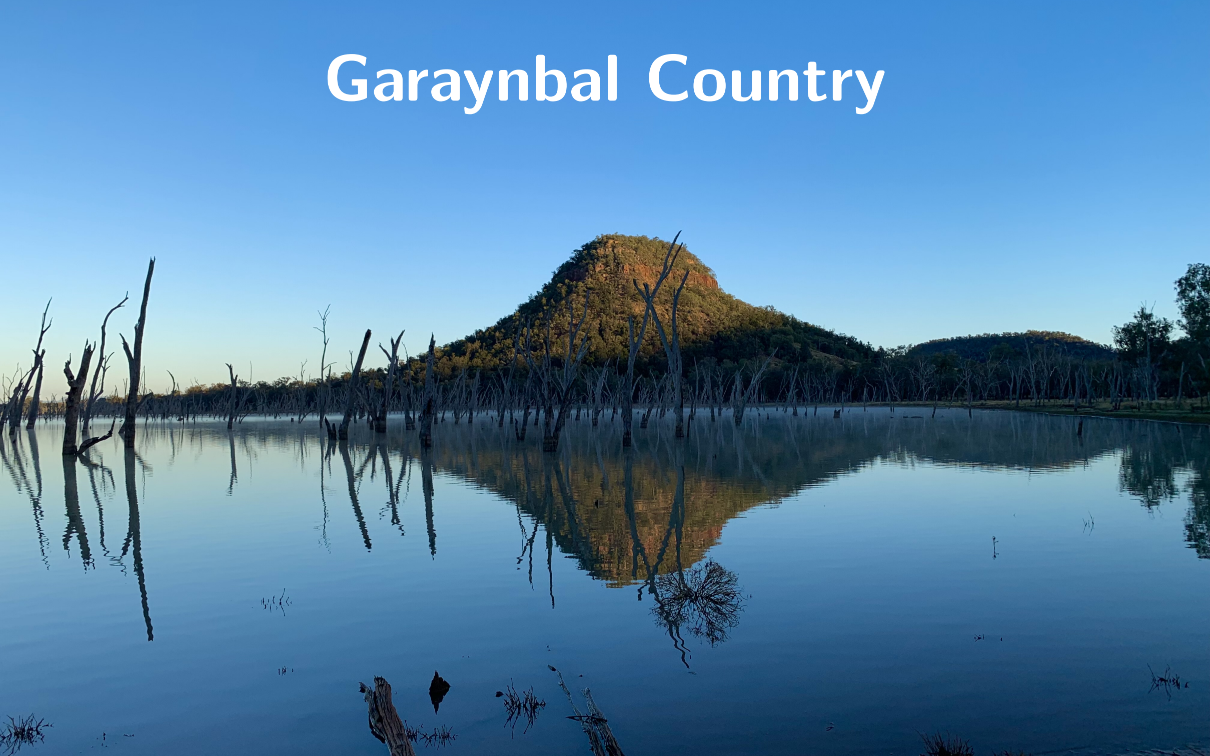
Garaynbal Country



We acknowledge, celebrate and pay our respects to the Ngunnawal and Ngambri people of the Canberra region and to all First Nations Australians on whose traditional lands we meet and work, and whose cultures are among the oldest continuing cultures in human history.

Learn more about Acknowledgement of Country here

Find out more about Canberra's Aboriginal history here

Recursion with lists

Contents

Announcements

Labs week 4

- Practice codeworld

Assignment 1 is out

- Part A: due Thursday 24 August (Week 5)
- Part B: due Friday 8 September
- Next week: presentation about report writing

Midsemester quiz

- Monday 28 August (Week 6)
- Next week: example solutions of previous questions

Lists

(Thompson: §5.4 and §5.5)

Remember lists

```
data [Int] = [] | Int : [Int]
data [Char] = [] | Char : [Char]
data [Int -> Bool] = [] | (Int -> Bool) : [Int -> Bool]
```

In general

```
data [a] = [] | a : [a]
```

This is a **recursive type**, because `[a]` is defined by referring to itself

List ranges

(Thompson: §5.4)

We can define lists of numbers using **ranges**

```
ghci> [0,4..20]
[0,4,8,12,16,20]

ghci> [2,7..24]
[2,7,12,17,22]
```

Remark This can also be defined by recursion on numbers (Lab 5)

Other uses

- `[0,4..20]` will output the list `[0,4,8,12,16,20]`
- `[1..]` will output the infinite list of all positive integers
- `[0.1,0.3..1.1]` might not behave as you would expect

Recursion

(THOMPSON: §4.4 and §4.5)

Solving a problem with recursion

1. **Base case:** Identify the smallest version(s) of the problem
   - Solve the problem on a very small piece of data (like the empty list)
   - Write code that solves the problem **without recursion**
   - This avoids infinite repetition
2. **Step case:** Solve the problem using a smaller version of the problem
   - This involves a **recursive call** to a smaller version of the problem
   - Here "smaller" means closer to a base case
3. **Distinguish:** between the base and step cases using `case` or guards

Remarks

- We saw this work with integers
- Works well with recursive data types

Recursion on lists

(THOMPSON: §7.3)

Lists are defined by

```
data [a] = [] | a : [a]
```

base case   step case

Observations

- Recursive functions on lists often use this base case and step case (but not always!)
- Recursive functions on recursive types often follow the structure of the type

Length

There is a Prelude (built-in) function `length` on lists

If a list contains elements of type `Char` then `length` has type

```
length :: [Char] -> Int
```

Length

There is a Prelude (built-in) function `length` on lists

If a list contains elements of type `Double` then `length` has type

```
length :: [Double] -> Int
```

Length

(Thompson: §7.3)

There is a Prelude (built-in) function `length` on lists

If a list contains elements of type `a` then `length` has type

```
length :: [a] -> Int
```

Remark The actual type of `length` is

```
length :: Foldable t => t a -> Int
```

- We have not learned what that means yet
- For now, we think of `t a` as `[a]`

Length

(Thompson: §7.3)

How can we define `length` ourselves?

```haskell
myLength :: [Char] -> Int
```

Base case: the empty list has length 0

Step case: a non-empty list `x:xs` is 1 longer than the length of `xs`

`x :: Char`   `xs :: [Char]`

```haskell
myLength list = case list of
  []   -> 0                   -- base case
  x:xs -> 1 + myLength xs     -- step case
```

Length

(Thompson: §7.3)

```haskell
myLength :: [Char] -> Int
myLength list = case list of
  []    -> 0                             -- base case
  x:xs -> 1 + myLength xs               -- step case
```

Running Haskell with `-W` gives an **error**

```
Lists.hs:4:3: warning: [-Wunused-matches]
    Defined but not used: 'x'
  |
4 |   x:xs -> 1 + myLength xs
  |   ^
```

Length

(Thompson: §7.3)

```
myLength :: [Char] -> Int
myLength list = case list of
  []     -> 0                            -- base case
  _:xs -> 1 + myLength xs               -- step case
```

Example Suppose `list = []`, then we get `myLength [] = 0`

Example We have

```
  myLength ['H','a','s','k']
= 1 + myLength ['a','s','k']
= 1 + 1 + myLength ['s','k']
= 1 + 1 + 1 + myLength ['k']
= 1 + 1 + 1 + 1 + myLength []
= 1 + 1 + 1 + 1 + 0           = 4
```

A common pattern

(Thompson: §7.3)

Many recursions on lists have a similar shape to `length`

1. Choose the behaviour for the empty list
2. Define how to combine the head of a list with a recursive call on the tail

Square all

(Thompson: §7.3)

Suppose we want to define a function which squares each integer in a list

```
ghci> squareAll [1, 2, 3, 4]
[1, 4, 9, 16]
```

How can we define `squareAll`?

```
squareAll :: [Int] -> [Int]
squareAll list = case list of
  []    -> []
  x:xs -> (x^2) : squareAll xs
```

Square all

(THOMPSON: §7.3)

```haskell
squareAll :: [Int] -> [Int]
squareAll list = case list of
  []     -> []
  x:xs -> (x^2) : squareAll xs
```

Example

```haskell
  squareAll [4,6,3]                          -- [4,6,3] = 4:[6,3]
= (4^2) : squareAll [6,3]
=  16   : squareAll [6,3]                      -- [6,3] = 6:[3]
=  16   : (6^2) : squareAll [3]
=  16   :  36   : squareAll [3]                  -- [3] = 3:[]
=  16   :  36   : (3^2) : squareAll []
=  16   :  36   :  9    : squareAll []
=  16   :  36   :  9    : []
```

Minimum

Not all recursive functions follow the same pattern

```
minimum :: [Double] -> Double
```

This finds the smallest element of a list, and gives an error on the empty list

Question What are the base cases?

- `[]`
- `[x]`

Remark We can use the built-in function `min :: Double -> Double -> Double` that takes the minimum of *two* numbers

Minimum

Not all recursive functions follow the same pattern

```
minimum :: [Double] -> Double
```

This finds the smallest element of a list, and gives an error on the empty list

Question What are the base cases?

```
myMinimum :: [Double] -> Double
myMinimum list = case list of
  []    -> error "empty list has no minimum"
  [x]   -> x
  y:ys  -> min y (myMinimum ys)
```

Remark Here `min` is the built-in function that takes the minimum of two numbers

Minimum

(THOMPSON: §7)

```haskell
myMinimum :: [Double] -> Double
myMinimum list = case list of
  []    -> error "empty list has no minimum"
  [x]   -> x
  y:ys  -> min y (myMinimum ys)
```

Example

```haskell
  myMinimum [2,5,1,3]
= min 2 (myMinimum [5,1,3])
= min 2 (min 5 (myMinimum [1,3]))
= min 2 (min 5 (min 1 (myMinimum [3])))
```

Minimum

(Thompson: §7)

```
myMinimum :: [Double] -> Double
myMinimum list = case list of
  []    -> error "empty list has no minimum"
  [x]   -> x
  y:ys  -> min y (myMinimum ys)
```

Example

```
  myMinimum [2,5,1,3]
= min 2 (myMinimum [5,1,3])
= min 2 (min 5 (myMinimum [1,3]))
= min 2 (min 5 (min 1 (myMinimum [3])))
= min 2 (min 5 (min 1 3))
```

Minimum

(Thompson: §7)

```haskell
myMinimum :: [Double] -> Double
myMinimum list = case list of
  []    -> error "empty list has no minimum"
  [x]   -> x
  y:ys  -> min y (myMinimum ys)
```

Example

```haskell
  myMinimum [2,5,1,3]
= min 2 (myMinimum [5,1,3])
= min 2 (min 5 (myMinimum [1,3]))
= min 2 (min 5 (min 1 (myMinimum [3])))
= min 2 (min 5 (min 1 3))
= min 2 (min 5 1)
```

Minimum

(Thompson: §7)

```
myMinimum :: [Double] -> Double
myMinimum list = case list of
  []     -> error "empty list has no minimum"
  [x]    -> x
  y:ys   -> min y (myMinimum ys)
```

Example

```
  myMinimum [2,5,1,3]
= min 2 (myMinimum [5,1,3])
= min 2 (min 5 (myMinimum [1,3]))
= min 2 (min 5 (min 1 (myMinimum [3])))
= min 2 (min 5 (min 1 3))
= min 2 (min 5 1)
= min 2 1
```

Minimum

(THOMPSON: §7)

```haskell
myMinimum :: [Double] -> Double
myMinimum list = case list of
  []    -> error "empty list has no minimum"
  [x]   -> x
  y:ys -> min y (myMinimum ys)
```

Example

```haskell
  myMinimum [2,5,1,3]
= min 2 (myMinimum [5,1,3])
= min 2 (min 5 (myMinimum [1,3]))
= min 2 (min 5 (min 1 (myMinimum [3])))
= min 2 (min 5 (min 1 3))
= min 2 (min 5 1)
= min 2 1 = 1
```

Product from the right

In the examples so far, we did computations **from the right** of the list: first the empty list, then the right most element, and so on

Example

```
myProductR :: [Double] -> Double
myProductR list = case list of
  []    -> 1.0
  x:xs -> x * product xs
```

```
myProductR (1.5 : 2.0 : [])
= 1.5 * myProductR (2.0 : [])
= 1.5 * (2.0 * myProductR [])
```

Product from the right

In the examples so far, we did computations **from the right** of the list: first the empty list, then the right most element, and so on

Example

```
myProductR :: [Double] -> Double
myProductR list = case list of
  []    -> 1.0
  x:xs  -> x * product xs
```

```
myProductR (1.5 : 2.0 : [])
= 1.5 * myProductR (2.0 : [])
= 1.5 * (2.0 * myProductR [])        -- compute empty list
= 1.5 * (2.0 * 1.0)
```

Product from the right

In the examples so far, we did computations **from the right** of the list: first the empty list, then the right most element, and so on

Example

```
myProductR :: [Double] -> Double
myProductR list = case list of
  []   -> 1.0
  x:xs -> x * product xs
```

```
myProductR (1.5 : 2.0 : [])
= 1.5 * myProductR (2.0 : [])
= 1.5 * (2.0 * myProductR [])
= 1.5 * (2.0 * 1.0)
= 1.5 *  2.0
```

Product from the right

In the examples so far, we did computations **from the right** of the list: first the empty list, then the right most element, and so on

Example

```
myProductR :: [Double] -> Double
myProductR list = case list of
  []    -> 1.0
  x:xs -> x * product xs
```

```
myProductR (1.5 : 2.0 : [])
= 1.5 * myProductR (2.0 : [])
= 1.5 * (2.0 * myProductR [])
= 1.5 * (2.0 * 1.0)
= 1.5 *  2.0
= 3.0
```

Product from the left

We can also write a recursion that computes **from the left** by introducing an extra value of the result "so far"

Example

```
myProductSoFar :: Double -> [Double] -> Double
myProductSoFar soFar list = case list of
  []   -> soFar
  y:ys -> myProductSoFar (soFar * y) ys
```

```
  myProductSoFar 1.0 (1.5 : 2.0 : [])
= myProductSoFar (1.0 * 1.5) (2.0 : [])
= myProductSoFar ((1.0 * 1.5) * 2.0) []
= (1.0 * 1.5) * 2.0
= 1.5 * 2.0
= 3.0
```

Product from the left

We can also write a recursion that computes **from the left** by introducing an extra value of the result “so far”

Example

```
myProductSoFar :: Double -> [Double] -> Double
myProductSoFar soFar list = case list of
  []    -> soFar
  y:ys -> myProductSoFar (soFar * y) ys

myProductL :: [Double] -> Double
myProductL list = myProductSoFar 1.0 list
```

Remark The `soFar` is usually called an **accumulator**, and denoted by `acc`

Left vs right

Question Which one is better?

Observations

- For products, the result is the same because `*` is **associative**

```
1.5 * (2.0 * 1.0) == (1.5 * 2.0) * 1.0
```

- If the operation is not associative, then the result can be different
- Sometimes one is faster than the other
  - For example, `fibonacci` with or without accumulator (see Lab 5, Exercise 5 and Extension 2)

Acronym

The **acronym** of a list of words is the first letter of each word

```haskell
-- acronym computer from the right
acronymR :: [String] -> String
acronymR list = case list of
  []    -> ""
  x:xs -> (head x) : (acronymR xs)

-- acronym with an accumulator
acronymL :: [String] -> String
acronymL = acronymAcc ""

acronymAcc :: String -> [String] -> String
acronymAcc    soFar     list = case list of
  []    -> soFar
  x:xs -> acronymAcc (soFar ++ [head x]) xs
```

Acronym evaluation from the Right

Example of evaluation of `acronymR`

```
acronymR ["This", "is", "an", "example"]
= 'T' : (acronymR ["is", "an", "example"])
= 'T' : 'i' : (acronymR ["an", "example"])
= 'T' : 'i' : 'a' : (acronymR ["example"])
= 'T' : 'i' : 'a' : 'e' : (acronymR [])
= 'T' : 'i' : 'a' : 'e' : ""
= 'T' : 'i' : 'a' : "e"
= 'T' : 'i' : "ae"
= 'T' : "iae"
= "Tiae"
```

Acronym evaluation from the Left

Example of evaluation of `acronymL`

```
acronymL ["This", "is", "an", "example"]
= acronymAcc "" ["This", "is", "an", "example"]
= acronymAcc ("" ++ ['T']) ["is", "an", "example"]
= acronymAcc (("" ++ ['T']) ++ ['i']) ["an", "example"]
= acronymAcc ((("" ++ ['T']) ++ ['i']) ++ ['a']) ["example"]
= acronymAcc (((("" ++ ['T']) ++ ['i']) ++ ['a']) ++ ['e']) []
= ((("" ++ ['T']) ++ ['i']) ++ ['a']) ++ ['e']
= (("T" ++ ['i']) ++ ['a']) ++ ['e']
= ("Ti" ++ ['a']) ++ ['e']
= "Tia" ++ ['e']
= "Tiae"
```

All

The function `all :: [Bool] -> Bool` returns `True` if all values in a list are `True`, and `False` otherwise

```haskell
-- all computer from the right
allR :: [Bool] -> Bool
allR list = case list of
  []    -> True
  x:xs -> x && (allR xs)

-- acronym with an accumulator
allL :: [Bool] -> Bool
allL = allAcc True

allAcc :: Bool -> [Bool] -> Bool
allAcc acc list = case list of
  []    -> acc
  x:xs -> allAcc (acc && x) xs
```

All evaluation from the right

Example of evaluation of `allR`

```
allR [True, False, True, True]
= True && (allR [False, True, True])
= True && (False && (allR [True, True]))
= True && (False && (True && (allR [True])))
= True && (False && (True && (True && (allR []))))
= True && (False && (True && (True && True)))
= True && (False && (True &&  True))
= True && (False &&  True)
= True &&  False
= False
```

All evaluation from the left

Example of evaluation of `allL`

```
allR [True, False, True, True]
= allAcc True [True, False, True, True]
= allAcc (True && True) [False, True, True]
= allAcc ((True && True) && False) [True, True]
= allAcc (((True && True) && False) && True) [True]
= allAcc ((((True && True) && False) && True) && True) []
= (((True && True) && False) && True) && True
= ((True && False) && True) && True
= (False && True) && True
= False && True
= False
```

Recursion on two lists

This function runs through two lists and pairs up the elements

```
zip :: [Int] -> [Char] -> [(Int, Char)]
```

Example

```
ghci> zip [1,5] ['c','d']
[(1,'c'),(5,'d')]
ghci> zip [1,5] ['c','d','e']
[(1,'c'),(5,'d')]
ghci> zip [1,5] []
[]
ghci> zip [] ['c','d']
[]
```

Recursion on two lists

(Thompson: §7.5)

This function runs through two lists and pairs up the elements

```
myZip :: [Int] -> [Char] -> [(Int, Char)]
myZip list1 list2 = case (list1, list2) of
  ([],_)         -> []                           -- base case
  (_,[])         -> []                           -- base case
  (x:xs, y:ys) -> (x,y) : (myZip xs ys)          -- step case
```

Remark

- In the code above, we could also swap `Int` and `Char`
- In fact, we can replace `Int` and `Char` for any other types!
- We can use **type variables** to indicate this

Recursion on two lists

This function runs through two lists and pairs up the elements

```
myZip :: [a] -> [b] -> [(a, b)]
myZip list1 list2 = case (list1, list2) of
  ([],_)         -> []                              -- base case
  (_,[])         -> []                              -- base case
  (x:xs, y:ys) -> (x,y) : (myZip xs ys)           -- step case
```

Remark

- `a` and `b` are **type variables**
- They can be replaced for any type
- `myZip` is called a **parametric polymorphism**

Summary

Recursion

- Recursive functions on lists often use:
  - Base case: `[]` (empty list)
  - Step case: `x:xs`
- Sometimes different base and step cases! (we have seen many examples)

Left vs Right

- We can go through a list from the left or from the right
- The results may differ
- Computing from the left often requires an **accumulator** and a helper function

Next weeks

Labs

- Week 5: recursion and lists
- Week 6: lists, parametric polymorphism, recursive data types

Lectures

- More on parametric polymorphisms
- Presentation: how to write a report
- Example solutions of some old midsem exercises