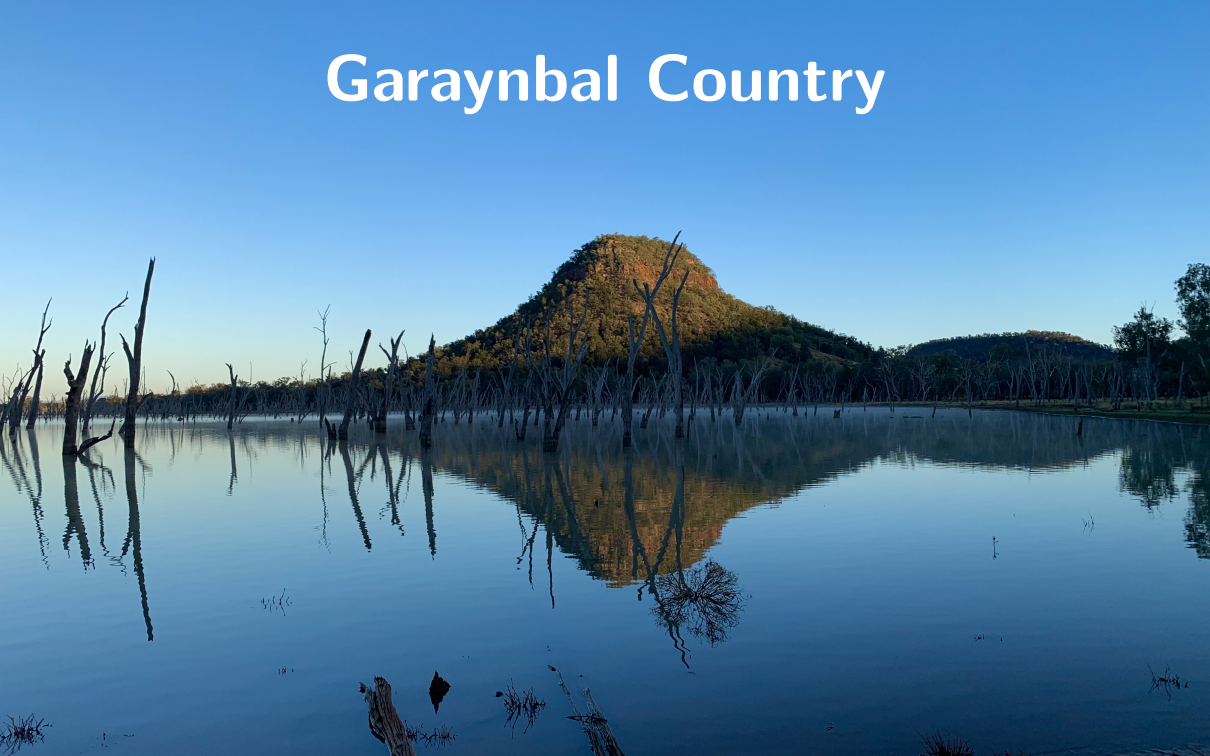
Garaynbal Country



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

Code quality

Contents

What makes “good code”?

- Correct
- Fast
- Readable

How can we ensure correctness?

- Reading your code
- Testing your code
- Mathematical proof about your code

Testing

"Testing shows the presence,
not the absence of bugs"

Edsger W. Dijkstra (1930-2002)

Black box vs white box testing

(THOMPSON: §4.8)

Black box testing involves writing tests without looking at your code

- Motivated by the specification
- Can be written before you write a line of code

White box testing involves writing tests motivated by your code

- Be your own enemy - how can I break this code?

Good black box testing

(Thompson: §4.8)

Identify **testing groups** of inputs

- Input from the same group behaves similar
- Input from different groups behaves differently
- Groups should cover all possibilities

Test a "typical" representative of each testing group

Remark Pay attention to **special cases**

- For example boundaries between groups, or extreme input
- These should be tested individually

Good black box testing

(Thompson: §4.8)

Example The function

```
1   maxThree :: Int -> Int -> Int -> Int
```

takes three input values and returns the largest one.

We find the following **testing groups**

- The group of inputs where the first is greatest
- The group where the second is greatest
- The group where the third is greatest
- Boundary cases: situations where some inputs are equal

Good white box testing

(Thompson: §4.8)

Identify points where your program **makes a choice** on input:

- case expressions
- guards
- base case vs step case of recursion

Test a "typical" input for each possible choice

Remark Pay particular attention to:

- Inputs on the boundary between choices
- Overlapping situations
- The use of `_` and `otherwise`, which can sweep up many different situations

Good white box testing

(Thompson: §4.8)

Example

```
1  maxThree :: Int -> Int -> Int -> Int
2  maxThree x y z
3    | x > y && x > z = x
4    | y > x && y > z = y
5    | otherwise      = z
```

The boundary of the first two tests suggests testing e.g. the input `2 2 1`

Recording your tests

Tests in `ghci` is a good start, but not a robust approach

- You will forget what you have tested so far (and miss possible errors)
- If you change your code you need to redo all tests

Better approach: write **unit tests** in your code (including expected result)

- You can run the same tests repeatedly as your code changes

Doctest

Doctest checks all unit tests in your code, if they are written in the right format

```haskell
-- | Compute Fibonacci numbers
-- >>> fib 10
-- 55
-- >>> fib 5
-- 5

fib :: Int -> Int
```

Then call `doctest MyFileName.hs` from **outside** ghci

Doctest

Doctest checks all unit tests in your code, if they are written in the right format

```
1  -- Compute Fibonacci numbers
2  -- >>> fib 10
3  -- 55
4  -- >>> fib 5
5  -- 5
6
7  fib :: Int -> Int
```

Then call `doctest MyFileName.hs` from **outside** ghci

- The `|` is essential

Doctest

Doctest checks all unit tests in your code, if they are written in the right format

```
1  -- | Compute Fibonacci numbers
2  -- >>> fib 10
3  --  55                                  ×
4  -- >>> fib 5
5  -- 5
6
7  fib :: Int -> Int
```

Then call `doctest MyFileName.hs` from **outside** ghci

- The `|` is essential
- Indenting and spacing has to be perfect

Randomised testing

Randomised testing allows for many tests with little effort

Can be done using Haskell's **QuickCheck** library for property-based testing

- Not explained in lectures or labs
- If interested, read up in the textbook or online

Special cases still need to be dealt with separately

Style

What do we mean by **style**?

- By **style** we mean writing code that is **easy to read**
- It is worth marks in Assignments 2 and 3, and the finale exam
- More info in the style guide online (taken from UPenn)

Style: Names and comments

- Use **comments** to clarify your code
  - Syntax:

```
-- this comments out one line
{- syntax for a bigger comment -}
```

  - Say **what** your code does, not **how** it works
- **Type declarations** for (top-level) functions give vital information
  - Always declare, even when Haskell can infer them
  - The `type` keyword can help clarify type declarations
- **Unit tests** can help explain what a function is for
- Use **descriptive names** for variables, constructors functions, modules, ...
  - This helps keep your code readable
  - Use the `camelCase` naming convention

Style: Using the language appropriately

- Don't use cases if you can use guards

Example

```
1  myAbs :: Int -> Int
2
3  myAbs x = case x >= 0 of
4    True  -> x
5    False -> -x                    ✗
6
7  myAbs x
8    | x >= 0     = x
9    | otherwise = -x                ✓
```

Style: Using the language appropriately

- Don't use guards if you can do without

Example

```
1  isPositive :: Int -> Bool
2
3  isPositive x
4    | x > 0      = True
5    | otherwise = False          ✗
6
7  isPositive x = x > 0           ✓
8
9  isPositive = (> 0)             ✓
```

Style: other tips

- Make sure you have no warnings (e.g. use `_`)
- Delete dead code (code that is not used anywhere)
- Minimise repeated code
- Give structure to big definitions, e.g. with helper functions
- Avoid unnecessary code, e.g. use Prelude functions
- Use consistent indentation (spaces, not tabs)
- Use lines of 80 characters wide or less