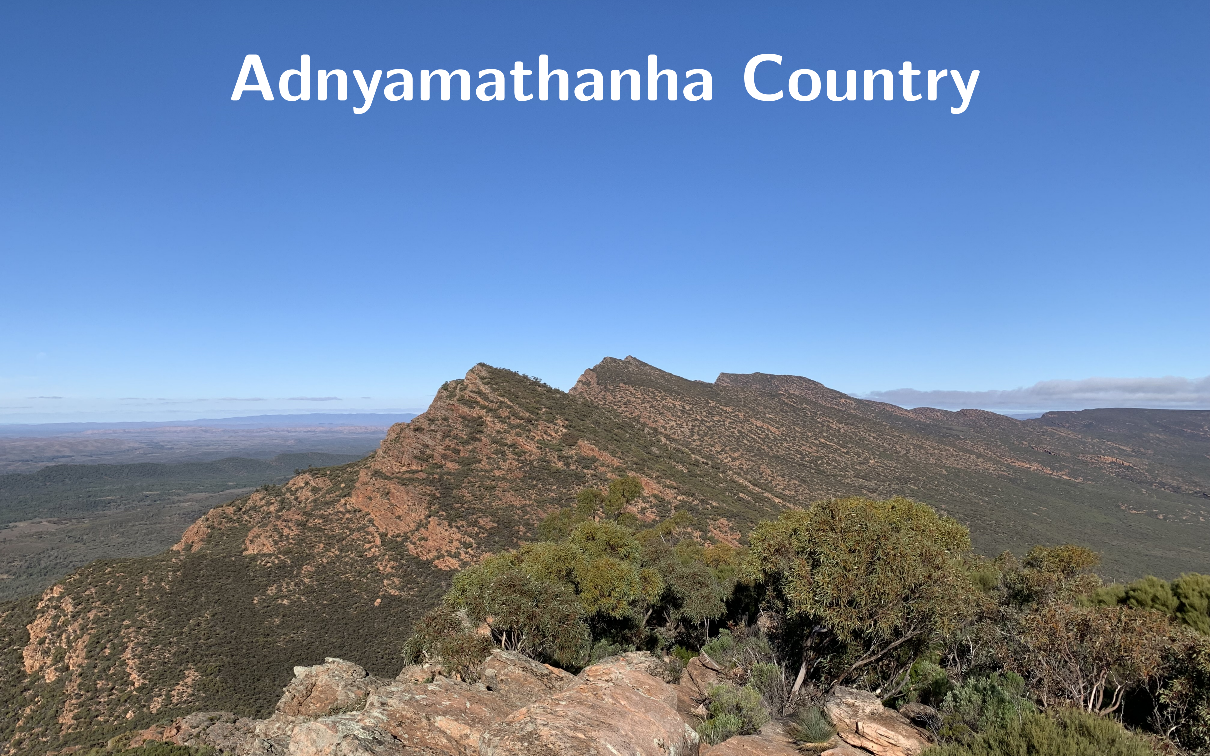
Adnyamathanha Country



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

In the photo: Adnyamathanha Country in South Australia

Find out more about Canberra's Aboriginal history here

Case expressions

Contents

Sum types

(Thompson: §5.3, §14.1)

Sum types (constructors can have zero arguments)

```haskell
data IntPlusBool = First Int | Second Bool  deriving Show
```

keyword

name of type

tag or constructor

argument

- Functions often act different on different constructors
- The expression `case` lets us say how each constructor should be handled
- If constructors have arguments, `case` lets us give variable names to each argument (so we can refer to them)

Case expressions

The expression `case` allows us to **pattern match** to say how different constructors should be dealt with

Example

```
data Bool = True | False

myNot :: Bool -> Bool
myNot b = case b of
  True  -> False
  False -> True
```

Remark Cases are read from top to bottom

Custom && function

The expression `case` allows us to **pattern match** to say how different constructors should be dealt with

Example

```
data Bool = True | False

myAnd :: Bool -> Bool -> Bool
myAnd b c = case (b, c) of
  (True, True)   -> True
  (True, False)  -> False
  (False, True)  -> False
  (False, False) -> False
```

Custom && function

(Thompson: §7.2)

The expression `case` allows us to **pattern match** to say how different constructors should be dealt with

Example

```haskell
data Bool = True | False

myAnd' :: Bool -> Bool -> Bool
myAnd' b c = case (b, c) of
  (True, True)    -> True
  _               -> False
```

Remark

- The notation `_` captures **anything**
- Cases are read from top to bottom, so `(True, True)` will not trigger `_`

Custom absolute value function

The expression `case` allows us to **pattern match** to say how different constructors should be dealt with

Example

```
data Bool = True | False

myAbs :: Int -> Int
myAbs n = case n >= 0 of
  True  -> n
  False -> -n
```

Animal farm

(Thompson: §7.2)

```haskell
data Animal = Cat | Duck | Worm


legs :: Animal -> Int
legs x = case x of
  Cat        -> 4
  Duck       -> 2
  Worm       -> 0
```

Animal farm

(Thompson: §7.2)

```
data Animal = Cat | Duck | Worm | Millipede Int
-- the argument to Millipede is the number of body segments

legs :: Animal -> Int
legs x = case x of
  Cat         -> 4
  Duck        -> 2
  Worm        -> 0
  Millipede y -> 2 * y
```

Naming variables

```haskell
data Animal = Cat | Duck | Worm | Millipede Int
-- the argument to Millipede is the number of body segments

legs :: Animal -> Int
legs animal = case animal of
  Cat                -> 4
  Duck               -> 2
  Worm               -> 0
  Millipede segments -> 2 * segments
```

Remark

- The upper-case `Animal` is a **type**
- The lower-case `animal` is a **variable**

Error messages

(Thompson: §4.4, §7.5)

```haskell
data Animal = Cat | Duck | Worm | Millipede Int | Octopus
-- the argument to Millipede is the number of body segments

legs :: Animal -> Int
legs animal = case animal of
  Cat                -> 4
  Duck               -> 2
  Worm               -> 0
  Millipede segments -> 2 * segments
  Octopus            -> error "Are tentacles legs?"
```

Guarded expressions

Guarded expressions or **guards** can be used when case gives a Boolean value

Example

```haskell
myAbs :: Int -> Int                 -- using case expression
myAbs x = case x >= 0 of
  True  -> x
  False -> -x

myAbs' :: Int -> Int                -- using guarded expressions
myAbs' x
  | x >= 0     = x
  | otherwise = -x
```

Guarded expressions

(Thompson: §3.4, §3.7)

Example How would you write the following function using `case`?

```
approxSize :: Int -> String
approxSize x
    | abs(x) > 1000000000000000000 = "quintillions"
    | abs(x) > 1000000000000000 = "quadrillions"
    | abs(x) > 1000000000000 = "trillions"
    | abs(x) > 1000000000 = "billions"
    | abs(x) > 1000000 = "millions"
    | abs(x) > 1000 = "thousands"
    | otherwise = "small"
```

Remarks

- Guards are **syntactic sugar**: they are **never necessary**, but are better style
- Guards are evaluated from top to bottom
- Guards should **always** end with `otherwise`

Piecewise definitions (Thompson: §3.1, §5.2, §5.3)

Piecewise definitions can be used instead of `case x of` when `x` ranges over the the input of the function

Example

```
data Animal = Cat | Duck | Worm

legs :: Animal -> Int
legs Cat  = 4
legs Duck = 2
legs Worm = 0
```

Remark This notation will not be used in the lectures, but you may encounter it elsewhere

Summary

Case expressions

- The expression `case` allows us to **pattern match**
- In sum types, it lets us say how different constructors are dealt with
- If constructors have arguments, we can give variable names to each argument and refer to them on the right side of the function definition
- We can also put more complicated expressions inside `case`

Guarded expressions

- Can be used if the outcome of the `case` expression is Boolean

Piecewise definitions (not used in the lectures)

- Can be used if `case` ranges over the input of a function

Remark Each of these is evaluated from top to bottom

Example The next three functions do the same

```
myNeg1 :: Bool -> Bool          -- using case expressions
myNeg1 b = case b of
  True  -> False
  False -> True

myNeg2 :: Bool -> Bool          -- using guarded expressions
myNeg2 b
  | b == True = False          -- case for b == True
  | otherwise = True           -- other case is b == False

myNeg3 :: Bool -> Bool          -- using piecewise definition
myNeg3 True  = False
myNeg3 False = True
```

Next

Lab week 3

- Lots of practice with **case expressions** and **guarded expressions**

Next lectures

- Wednesday: **lists**
- Thursday: **recursion**