Recursion and higher order functions

1 Advice when writing recursive functions

1. Define the type
2. Enumerate the cases
3. Define simple or base cases
4. Define the reduction of other cases to simpler ones
5. (optional) generalise and simplify

2 Example: drop

1. Define the type
   Drop the first n elements from a list:

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

2. Enumerate the cases
   Two cases each for the integer and the list argument

   ```
   drop 0 [] =
   drop 0 (x:xs) =
   drop n [] =
   drop n (x:xs) =
   ```

3. Define simple or base cases
   Zero and the empty list are fixed points

   ```
   drop 0 [] = []
   drop 0 (x:xs) = x:xs
   drop n [] = []
   drop n (x:xs) =
   ```

4. Define the reduction of other cases to simpler ones
   Apply drop to the tail

   ```
   drop 0 [] = []
   drop 0 (x:xs) = x:xs
   drop n [] = []
   drop n (x:xs) = drop (n-1) xs
   ```

5. (optional) generalise and simplify
   Compress cases

   ```
   drop :: Int -> [a] -> [a]
   drop 0 xs = xs
   drop _ [] = []
   drop n (x:xs) = drop (n-1) xs
   ```

3 Equivalence of recursion and iteration

- Both purely iterative and purely recursive programming languages are Turing complete
- Hence, it is always possible to transform from one representation to the other
- Which is convenient depends on the algorithm, and the programming languages

Recursion ⇒ Iteration

- Write looping constructs, manually manage function call stack

Iteration ⇒ recursion

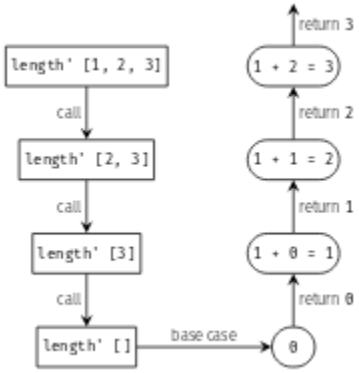
- Turn loop variables into additional function arguments
- And write tail recursive function (see later)

4 How are function calls managed

Usually a stack is used to manage nested function calls

```
length' :: [a] -> Int
length' [] =0
length' (x:xs) = 1 + length' xs
```

Calling length' on [1,2,3] does:



- Each entry on the stack uses memory
- Too many entries cases errors: the dreaded stack overflow
- How big this stack is depends on the language
- Typically "small" in imperative languages and "big" in functional ones

5 Typically don't have to worry about stack overflows

- In traditional imperative languages, we often try and avoid recursion
- Function calls are more expensive than just looping
- Deep recursion can result in stack overflow
- In contrast, Haskell is fine with much deeper recursion
- Unsurprising, given the programming model
- Still prefer to avoid recursion trees that are too deep

6 Classifying recursive functions

- Since it is natural to write recursive functions, it makes sense to think about classifying the different types we can encounter
- Classifying the type of recursion is useful to allow us to think about better/cheaper implementations

Linear recursion - only contains a single self reference

```
length' [] = []
length' (_:xs) = 1 + length' xs
```

Multiple recursion - The recursive call contains multiple self references

```haskell
fib 0 = 0
fib 1 = 1
fib n = fib (n-1) + fib (n-2)
```

Direct recursion - The function calls itself recursively

```haskell
product' [] = []
product' (x:xs) = x * product' xs
```

Mutual/indirect recursion - multiple functions call each other recursively

```haskell
even' :: Integral a => a -> Bool
even' 0 = True
even' n = odd' (n-1)

odd' Integral a => a -> Bool
odd' 0 = False
odd' n = even' (n-1)
```

A function is 1 of the 1st two and 1 of the 2nd two

7 Tail recursion: a special case

Definition: Tail recursion

A function is tail recursive if the last result of a recursive call is the result of the function itself.
Loosely, the last thing a tail recursive function does is call itself with new arguments, or return a value

- Such functions are useful because they have a trivial translation into loops
- Some languages (e.g. Scheme) guarantee that a tail recursive call will be transformed into a "loop-like" implementation using a technique called tail call elimination
- Complexity remains unchanged, but implementation is more efficient
- In haskell implementations, while nice, this is not so important

8 Iteration ⇔ tail recursion

Loops are convenient:

```python
def factorial (n):
    res = 1
    for i in range(n,1,-1)
        res *= i
    return res
```

Tail recursive implementation

- We can't write this directly, since we're not allowed to mutate things
- We can write it with a helper recursive function where all loop variables become arguments to the function

```haskell
factorial n = loop n 1
    where loop n res | n<0       = undefined
                     | n>1       = loop (n-1)(res*n)
                     | otherwise = res
```

9 What about complexity?

- Linear recursion often appears in list traversals. Typically make $O(n)$ recursive calls on data of size n
- Multiple recursion often appears in tree or graph traversals, as well as "divide and conquer" algorithms. Number of recursive calls more problem dependent

10 Maps and folds

10.1 Higher order functions

We've seen many functions that are naturally recursive
We'll now look at higher order functions in the standard library that capture many of these patterns

Definition: Higher order function

A function that does at least one of:

- Take one or more functions as arguments
- Returns a function as its result

Due to currying, every function of more than one argument is higher order in haskell

10.1.1 Why are they useful?

- Common programming idioms can be written as functions in the language
- Domain specific languages can be defined with appropriate collections of higher order functions
- We can use the algebraic properties of higher order functions to reason about programs ⇒ provably correct program transformations

⇒ Useful for domain specific compilers and automated program generation

10.1.2 Higher order functions on lists

- Many linear recursive functions on lists can be written using higher order library functions.
- map: apply a function to a list

```haskell
map :: (a -> b) -> [a] -> [b]
map _ [] = []
map f xs [f x | x <- xs]
```

- filter: remove entries from a list

```haskell
filter :: (a -> Bool) -> [a] -> [a]
filter _ [] = []
filter p xs = [x | x <- xs, p x]
```

10.2 Function composition

- Often tedious to write brackets and explicit variable names
- Can use function composition to simplify this

$$(f \circ g)(x) = f(g(x))$$

- Haskell uses the (.) operator

```haskell
(.) :: (b -> c) -> (a -> b) -> (a -> c)
f . g = \backslash x -> f (g (x))
-- example
odd a = not (even a)
odd   = not . even -- no need for the variable
```

- Useful for writing compositions of functions to be passed to other higher order functions
- Removes need to write λ-expressions

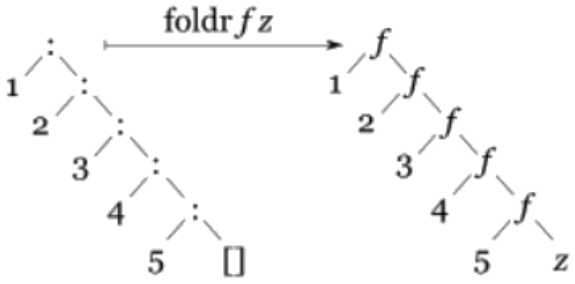
10.3 Folds

- Folds process a data structure in some order and build a return value
- Haskell provides a number of these in the standard prelude, with more available in the `Data.List` module

10.3.1 foldr: right associative fold

Process list from the front

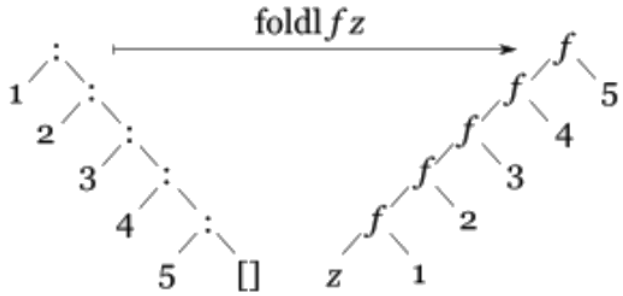
```
foldr :: (a -> b -> b) -> b -> [a] -> b
foldr f z []     = z
foldr f z (x:xs) = x `f` (foldr f z xs)
```



10.3.2 foldl: left associative fold

Processes list from the back (implicitly in reverse)

```
foldl :: (b -> a -> b) -> b -> [a] -> b
foldl f z []     = z
foldl f z (x:xs) = foldl f (z `f` x) xs -- tail recursive
```



10.3.3 Why would I use them?

- Capture many linear recursive patterns in a clean way
- Can have efficient library implementation ⇒ can apply program optimisations
- Actually apply to all **`Foldable`** types, not just lists
- e.g. `foldr`'s type is actually

  ```
  foldr :: Foldable t => (a -> b -> b) -> b -> t a -> b
  ```

- So we can write code for lists and (say) trees identically

Folds are general:

- Many library functions on lists are written using folds

  ```
  product = foldr (*) 1
  sum = foldr (+) 0
  maximum = foldr1 max
  ```

10.3.4 Which to choose?

foldr:

- Generally the right choice
- Works even for infinite lists
- Note `foldr (:) [] = id`
- Can terminate early

foldl:

- Usually best to use strict versions

```
import Data.List
foldl' -- note trailing
```

- Doesn't work on infinite lists (needs start at the end)
- Can't terminate early