COMP3258
Functional Programming
Tutorial Session 3: Recursive Functions and Sorting

The 1st Assignment Is Out!

Recipe of the Recursion

- Recursion on Numbers
  - base case: 0
  - inductive case: n
- Recursion on Lists
  - base case: []
  - inductive case: (x:xs)
- Recursion on Multiple arguments (e.g., two lists, list and numbers)
  - base case
  - inductive case

Review Functions (via Hoogle)

- `product`, `length`, `reverse`, `zip`, `drop`, `(++)`

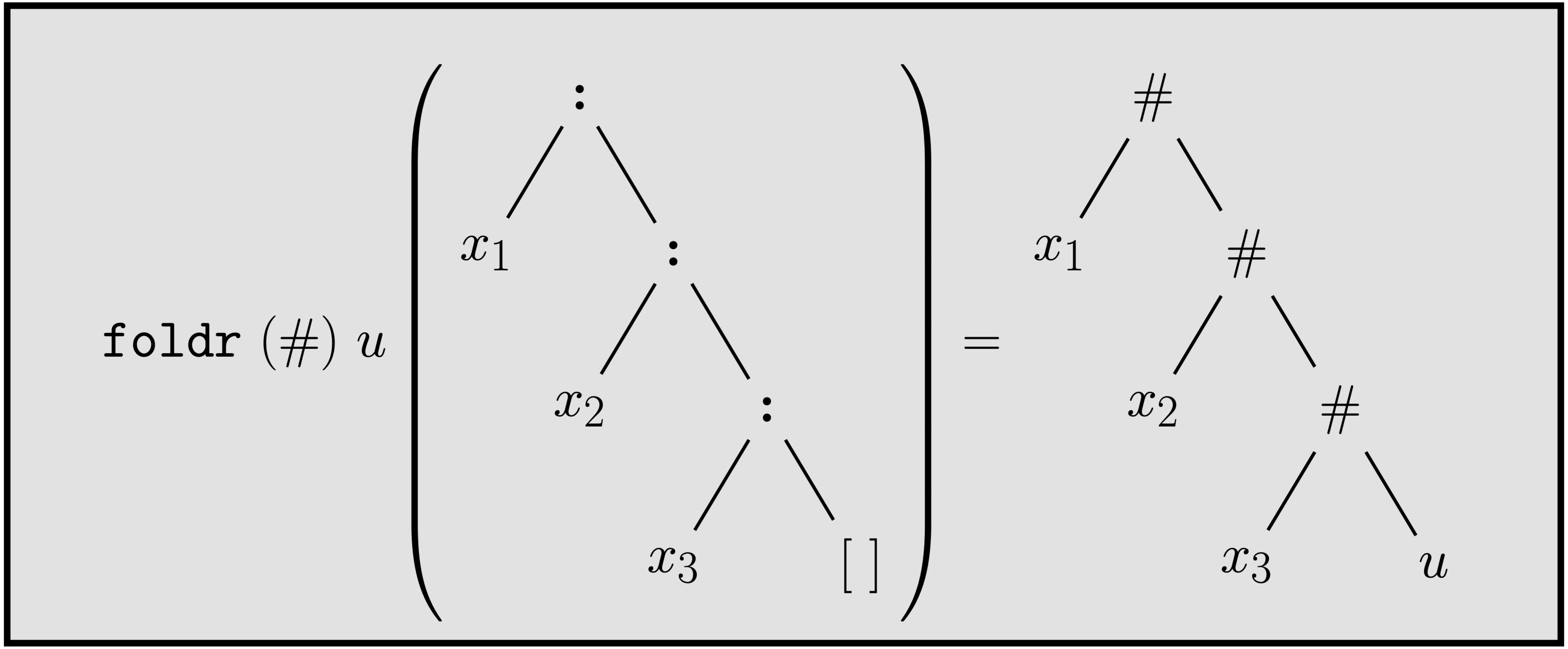
product

```haskell
-- | The 'product' function computes the product of the numbers of a
-- structure.
--
-- ==== __Examples__
--
-- Basic usage:
--
-- >>> product []
-- 1
--
-- >>> product [42]
-- 42
--
-- >>> product [1..10]
-- 3628800
--
-- >>> product [4.1, 2.0, 1.7]
-- 13.939999999999998
--
-- >>> product [1..]
-- * Hangs forever *
--
-- @since 4.8.0.0
product :: Num a => t a -> a
product = getProduct #. foldMap' Product
{-# INLINEABLE product #-}
```

length

```
--
-- ==== __Examples__
--
-- Basic usage:
--
-- >>> length []
-- 0
--
-- >>> length ['a', 'b', 'c']
-- 3
-- >>> length [1..]
-- * Hangs forever *
--
-- @since 4.8.0.0
length :: t a -> Int
length = foldl' (\c _ -> c+1) 0
```

A Taste of Folding



Credit: https://www.cantab.net/users/antoni.diller/haskell/units/unit06.html

reverse

```haskell
-- | 'reverse' @xs@ returns the elements of @xs@ in reverse order.
-- @xs@ must be finite.
--
-- >>> reverse []
-- []
-- >>> reverse [42]
-- [42]
-- >>> reverse [2,5,7]
-- [7,5,2]
-- >>> reverse [1..]
-- * Hangs forever *
reverse                 :: [a] -> [a]
#if defined(USE_REPORT_PRELUDE)
reverse                 =  foldl (flip (:)) []
#else
reverse l =  rev l []
  where
    rev []     a = a
    rev (x:xs) a = rev xs (x:a)
#endif
```

Tail Call Optimisation

zip

```
----------------------------------------------
-- | \(\mathcal{O}(\min(m,n))\). 'zip' takes two lists and returns a list of
-- corresponding pairs.
--
-- >>> zip [1, 2] ['a', 'b']
-- [(1,'a'),(2,'b')]
--
-- If one input list is shorter than the other, excess elements of the longer
-- list are discarded, even if one of the lists is infinite:
--
-- >>> zip [1] ['a', 'b']
-- [(1,'a')]
-- >>> zip [1, 2] ['a']
-- [(1,'a')]
-- >>> zip [] [1..]
-- []
-- >>> zip [1..] []
-- []
--
-- 'zip' is right-lazy:
--
-- >>> zip [] undefined
-- []
-- >>> zip undefined []
-- *** Exception: Prelude.undefined
-- ...
--
-- 'zip' is capable of list fusion, but it is restricted to its
-- first list argument and its resulting list.
{-# NOINLINE [1] zip #-}  -- See Note [Fusion for zipN/zipWithN]
zip :: [a] -> [b] -> [(a,b)]
zip []     _bs    = []
zip _as    []     = []
zip (a:as) (b:bs) = (a,b) : zip as bs
```

drop

```haskell
-- | 'drop' @n xs@ returns the suffix of @xs@
-- after the first @n@ elements, or @[]@ if @n >= 'length' xs@.
--
-- >>> drop 6 "Hello World!"
-- "World!"
-- >>> drop 3 [1,2,3,4,5]
-- [4,5]
-- >>> drop 3 [1,2]
-- []
-- >>> drop 3 []
-- []
-- >>> drop (-1) [1,2]
-- [1,2]
-- >>> drop 0 [1,2]
-- [1,2]
--
-- It is an instance of the more general 'Data.List.genericDrop',
-- in which @n@ may be of any integral type.
drop                   :: Int -> [a] -> [a]
#if defined(USE_REPORT_PRELUDE)
drop n xs      | n <= 0 =  xs
drop _ []               =  []
drop n (_:xs)           =  drop (n-1) xs
#else /* hack away */
{-# INLINE drop #-}
drop n ls
  | n <= 0     = ls
  | otherwise  = unsafeDrop n ls
  where
    -- A version of drop that drops the whole list if given an argument
    -- less than 1
    unsafeDrop :: Int -> [a] -> [a]
    unsafeDrop !_ []     = []
    unsafeDrop 1  (_:xs) = xs
    unsafeDrop m  (_:xs) = unsafeDrop (m - 1) xs
#endif
```

(++)

```
-----------------------------------------------
--                append
-----------------------------------------------

-- | Append two lists, i.e.,
--
-- > [x1, ..., xm] ++ [y1, ..., yn] == [x1, ..., xm, y1, ..., yn]
-- > [x1, ..., xm] ++ [y1, ...] == [x1, ..., xm, y1, ...]
--
-- If the first list is not finite, the result is the first list.
--
-- WARNING: This function takes linear time in the number of elements of the
-- first list.

(++) :: [a] -> [a] -> [a]
{-# NOINLINE [2] (++) #-}
  -- Give time for the RULEs for (++) to fire in InitialPhase
  -- It's recursive, so won't inline anyway,
  -- but saying so is more explicit
(++) []     ys = ys
(++) (x:xs) ys = x : xs ++ ys
```

Question 1

- Add the following line at the top of your file to avoid name clashes:

```
import Prelude hiding (concat, and, (!!), replicate, elem)
```

- Then define those library functions using recursion:

```
and       :: [Bool] → Bool
concat    :: [[a]] → [a]
replicate :: Int → a → [a]
(!!)      :: [a] → Int → a
elem      :: Eq a ⇒ a → [a] → Bool
```

Question 1

```haskell
and :: [Bool] -> Bool
and [] = True
and (x:xs) = x && and xs

concat :: [[a]] -> [a]
concat [] = []
concat (x:xs) = x ++ concat xs

replicate :: Int -> a -> [a]
replicate 0 x = []
replicate n x = x : (replicate (n - 1) x)
```

```haskell
(!!) :: [a] -> Int -> a
(!!) [] _ = error "index too large"
(!!) (x:xs) 0 = x
(!!) (x:xs) n = (!!) xs (n - 1)

elem :: Eq a => a -> [a] -> Bool
elem a [] = False
elem a (x:xs) = if a == x then True else elem a xs
```

Question 2

- Implement a recursive function doubleList which doubles all the elements and returns the list.
- Implement the function using map and lambda functions.

```haskell
double :: Int → Int
double x = x + x

doubleListRec :: [Int] → [Int]
doubleListRec [] = []
doubleListRec (x:xs) = double x : doubleListRec xs

doubleList = map double

doubleList' = map (\x → x + x)
```

Question 3

- Implement a recursive function zipSum, which takes two lists and returns the list of corresponding sums.
- Implement the function using the library function zipWith and lambda functions.

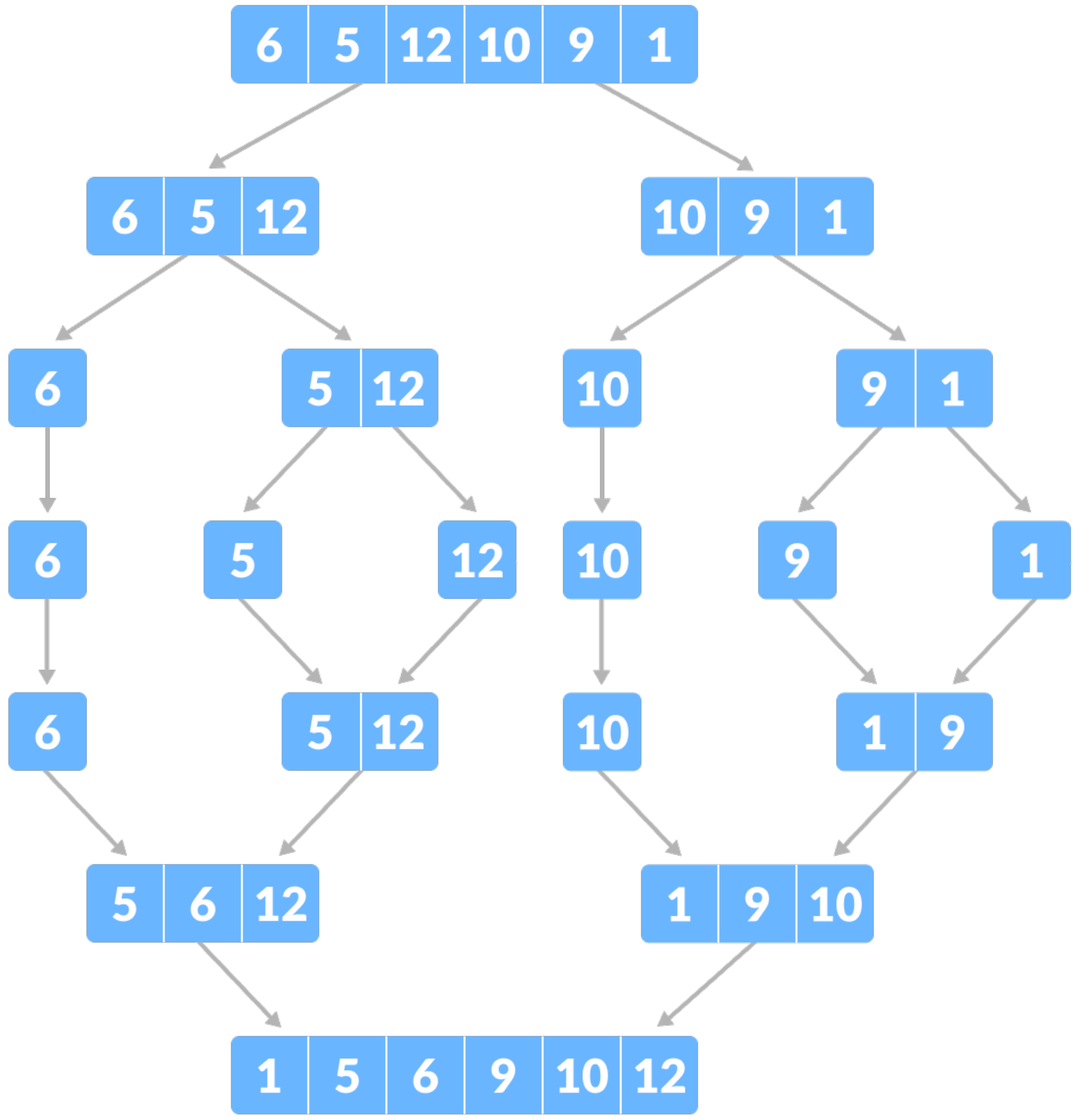
```haskell
zipSum :: [Int] → [Int] → [Int]
zipSum [] _ = []
zipSum _ [] = []
zipSum (x:xs) (y:ys) = (x + y) : zipSum xs ys

zipSum' = zipWith (+)
```

Merge Sort

1. Divide the unsorted list into n sublists, each containing one element (a list of one element is considered sorted).

2. Repeatedly merge sublists to produce new sorted sublists until there is only one sublist remaining. This will be the sorted list.



Credit: https://www.programiz.com/dsa/merge-sort

Step 1: merge function

- Define a recursive function merge that merges two sorted lists of integers to give a single sorted list.

```
merge :: [Int] → [Int] → [Int]
merge xs ys = undefined

-- >>> merge [5] [12]
-- [5,12]
-- >>> merge [6] [5, 12]
-- [5,6,12]
-- >>> merge [5, 6, 12] [1, 9, 10]
-- [1,5,6,9,10,12]
```

Step 2: msort function

- Define a recursive function msort that implements merge sort.
- Hint:
  - Lists whose length <= 1 are already sorted;
  - The other lists can be sorted by recursively sorting the two halves and merging the results;
  - There is a library function `splitAt :: Int → [a] → ([a], [a])`.

Merge Sort In Haskell

```haskell
merge :: [Int] → [Int] → [Int]
merge [] l = l
merge l [] = l
merge (x:xs) (y:ys) = if x < y then x : merge xs (y:ys) else y : merge (x:xs) ys

msort :: [Int] → [Int]
msort [] = []
msort [x] = [x]
msort xs = merge (msort l) (msort r)
  where
    (l, r) = splitAt p xs
    p = length xs `div` 2
```

($) Operator

- It's called function application operator.
- Why it matters?
  - f x (normal function application) has high precedence
  - ($) has the lowest precedence

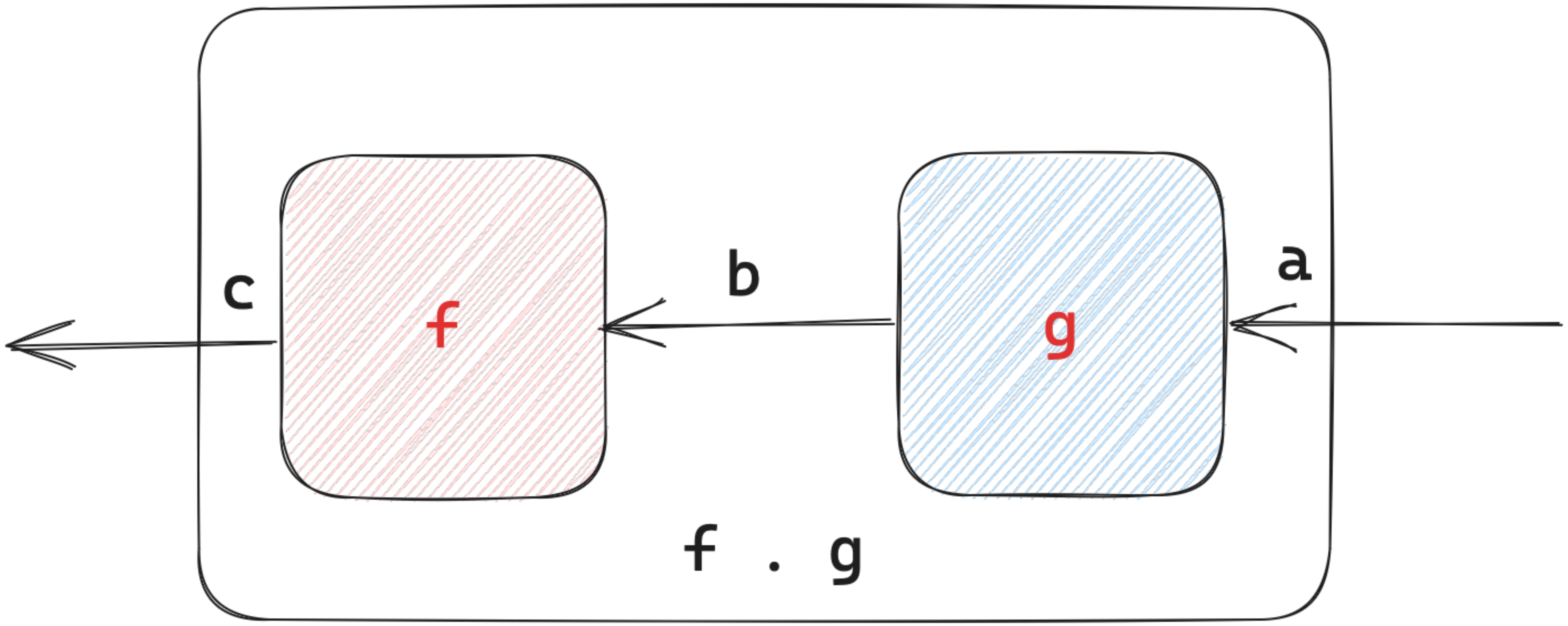
```
($) :: (a → b) → a → b
f $ x = f x

expr1 = sqrt (3 + 4 + 9)
expr2 = sqrt $ 3 + 4 + 9
```

(.) Operator

- function composition operator

```
(.) :: (b → c) → (a → b) → a → c
(f . g) x = f (g x)
```



Practice

- Remove parens from the following expression using operators we just learnt.

```
foo = reverse (take 6 [1..10])
```