Code (Wednesday Week 1)

Haskell

```haskell
-- Quicksort

qsort1 :: Ord a => [a] -> [a]
qsort1 [] = []
qsort1 (x:xs) =
    qsort1 smaller ++ [x] ++ qsort1 larger
    where smaller = filter (\a-> a <= x) xs
          larger  = filter (\b-> b > x) xs

qsort2 :: Ord a => [a] -> [a]
qsort2 [] = []
qsort2 (x:xs) =
    let smaller = filter (\a-> a <= x) xs
        larger  = filter (\b-> b > x) xs
    in qsort2 smaller ++ [x] ++ qsort2 larger

qsort3 :: Ord a => [a] -> [a]
qsort3 [] = []
qsort3 (x:xs) =
    let smaller = filter (<= x) xs
        larger  = filter (> x) xs
    in qsort3 smaller ++ [x] ++ qsort3 larger

-- List operations:
--  sum, concat, map, filter, foldr, foldl

sum' :: [Int] -> Int
sum' [] = 0
sum' (x:xs) = x + sum xs

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

map' :: (a -> b) -> [a] -> [b]
map' f []     = []
map' f (x:xs) = f x:map f xs

filter' :: (a -> Bool) -> [a] -> [a]
filter' p []     = []
filter' p (x:xs) = if p x then x:filter p xs
                  else filter p xs

-- foldr (@) e [w,x,y,z] = w @ (x @ (y @ (z @ e)))
```

```
-- foldright :: (a-> b-> b) -> b -> [a] -> b
foldright :: (a-> b-> b) -> b -> [a] -> b
foldright f e [] = e
foldright f e (x:xs) =  x `f` (foldright f e xs)
-- foldl (@) e [w,x,y,z] = (((e @ w) @ x) @ y) @ z
-- foldleft :: (b -> a -> b) -> b -> [a] -> b
foldleft :: (b -> a -> b) -> b -> [a] -> b
foldleft f e [] = e
foldleft f e (x:xs) = foldleft f (f e x) xs
--  sum, concat, filter, map using foldr
sum'' = foldright (+) 0
concat'' = foldright (++) []
filter'' p =
    foldright (\x xs -> if p x then x:xs else xs) []
map'' f = foldright ((:).f)[]
-- reverse
reverse1 :: [a] -> [a]
reverse1 [] = []
reverse1 (x:xs) = reverse1 xs ++ [x]
-- reverse using foldr
reverse2 :: [a] -> [a]
reverse2 = foldright (\x xs -> xs ++ [x]) []
-- more efficient reverse
reverse3 :: [a] -> [a]
reverse3 l =  rev l []
  where
    rev []     ys = ys
    rev (x:xs) ys = rev xs (x:ys)
-- more efficient reverse using foldl
reverse4 :: [a] -> [a]
reverse4 = foldleft (\xs y -> y : xs) []

--------
units :: [String]
units =
    ["zero", "one", "two", "three", "four", "five",
     "six", "seven", "eight", "nine", "ten"]

convert1 :: Int -> String
convert1 n = units !!n

teens :: [String]
teens =
    ["ten", "eleven", "twelve", "thirteen", "fourteen",
     "fifteen", "sixteen", "seventeen", "eighteen",
     "nineteen"]

tens :: [String]
tens =
```

```
    ["twenty", "thirty", "fourty", "fifty", "sixty",
     "seventy", "eighty", "ninety"]
digits2 :: Int -> (Int, Int)
digits2 n = (div n 10, mod n 10)
-- -- digits2 n = (n `div` 10, n `mod` 10)

-- combine2 :: (Int, Int) -> String
-- combine2 (t, u)
--     | t == 0            = convert1 u
--     | t == 1            = teens !! u
--     | t > 1 && u == 0   = tens !! (t-2)
--     | t > 1 && u /= 0   = tens !! (t-2) ++ "-" ++ convert1 u

combine2 :: (Int, Int) -> String
combine2 (t,u)
  | t == 0     = convert1 u
  | t == 1     = teens !! u
  | u == 0     = tens !! (t-2)
  | otherwise = tens !! (t-2) ++ "-" ++ convert1 u

convert2' :: Int -> String
convert2' = combine2 . digits2

convert2 :: Int -> String
convert2 n
    |  t == 0     = convert1 u
    |  t == 1     = teens !! u
    |  u == 0     = tens !! (t-2)
    |  otherwise  = tens !! (t-2)
                    ++ "-"
                    ++ convert1 u
    where (t, u) = (n `div` 10, n `mod` 10)

convert3 :: Int -> String
convert3 n
    |  h == 0     = convert2 n
    |  t == 0     = convert1 h ++ "hundred"
    |  otherwise  = convert1 h ++ " hundred and " ++ convert2 t
    where (h, t) = (n `div` 100, n `mod` 100)

convert6 :: Int -> String
convert6 n
    |  m == 0     = convert3 h
    |  h == 0     = convert3 m ++ " thousand"
    |  otherwise  = convert3 m ++ " thousand" ++ link h ++ convert3 h
    where (m, h) = (n `div` 1000, n `mod` 1000)

link :: Int -> String
link h = if (h<100) then " and " else " "
```

```
link' :: Int -> String
link' h
    |  h<100 = " and "
    |  otherwise  = " "

convert :: Int -> String
convert = convert6
```