# Introduction to Programming: Quiz 1 Solutions

*Solutions by Ryan Hota, from* `github.com/Ryan-Hota/Haskell2026FrontOfHouse` (`quiz1Soln.hs`).

1. Write a recursive function `dropOdds :: [a] → [a]` that drops the elements at odd indices of a list (indexing from 0). For example:

```
dropOdds [1,2,3,4,5] = [1,3,5]
```

**Solution.** Keep the first element, skip the second, and recurse on the rest. The empty list and the one-element list are the base cases.

```
dropOdds :: [a] → [a]
dropOdds []       = []
dropOdds [x]      = [x]
dropOdds (x:y:ys) = x : dropOdds ys
```

2. Write a function `sums` that takes a list of lists of numbers `xs` and returns the list of sums of the non-empty lists in `xs`. For example:

```
sums [[1..5], [], [2,6]] = [15,8]
```

**Solution.** In a list comprehension, the pattern `l@(_:_)` matches only non-empty lists, and elements that do not match the pattern are skipped, so the empty lists are dropped.

```
sums :: Num a ⇒ [[a]] → [a]
sums ls = [ sum l | l@(_:_) ← ls ]
```

3. Consider the function

```
f xs = foldr (/=) False xs
```

Find the type of `f`, and compute `f` on two example inputs.

**Solution.** Since `(/=)` is applied to `False`, the elements and the result are `Bool`. GHC reports the type

```
f :: Foldable t ⇒ t Bool → Bool
```

which, for lists, is `f :: [Bool] → Bool`. Two examples:

```
f [False, False, False, False] = False
f [True, False, False, False]  = True
```

For the second, `foldr` gives `True /= (False /= (False /= (False /= False)))`. The innermost comparisons give `False` each time, and `True /= False` is `True`. In general, `f xs` is `True` exactly when `xs` contains an odd number of `True`s.

4. Given the decimal expansions of two numbers (as lists of digits of equal length, most significant digit first), compute the decimal expansion of their sum. For example:

```
add [0,8,5,3,3] [0,2,8,6,8] = [1,1,4,0,1]
```

**Solution.** The helper `addWithCarry` adds the lists from the least significant end. It returns the digits of the sum together with the carry out of the leftmost position. Each step adds the two current digits and the carry from the digits to its right, then uses `divMod` by 10 to split this into a new carry and a new digit.

```
addWithCarry :: [Int] → [Int] → ([Int], Int)
addWithCarry [] [] = ([], 0)
addWithCarry (x:xs) (y:ys) = let
    (prevSum, prevCarry) = addWithCarry xs ys
    (newCarry, newDigit) = (x+y+prevCarry) `divMod` 10
    in (newDigit : prevSum, newCarry)

add :: [Int] → [Int] → [Int]
add xs ys = fst $ addWithCarry xs ys
```

The final carry is discarded, so the inputs need a leading 0 when the sum could have one more digit, as in the example.