# Week 4

## Administrative Stuff

- Drop in sessions have begun, Fridays at 2pm at Skaidrite Darius N109
- **Oliver Walsh** has been appointed as a class rep for this course.
    - Still looking for one more if possible!

## Prelude

Combining our knowledge from the previous topics:

- recursion (on both lists and integers)
- polymorphism
- common datatypes (lists, tuples, maybes)

We will now look at Haskell's default built-in library, called the "Prelude". The types and implementation details of built-in functions can be more involved than what we have studied so far, but the basic ideas are the same.

**Syntax for infix operators**

Some functions in Haskell are infix operators, like the `++` operator we have seen which appends two lists together. These operators are just normal functions, just with a symbolic name rather than an alphabetical one. To define a function with a symbolic name, we must wrap the name in parentheses like this: `(++)`. Thus, we can write a definition of `(++)` as follows:

```
(++) :: [a] -> [a] -> [a]
xs ++ ys = case xs of
  [] -> ys
  (x:rest) -> x : (rest ++ ys)
```

Similarly, a function with an alphabetical name, like `div` or `mod`, can be used as an infix operator by surrounding it in `` `backticks` ``. For example, `div 10 2` is the same as writing `` 10 `div` 2 ``

We have already seen many functions in the Prelude, including:

```
length :: [a] -> Int
replicate :: Int -> a -> [a]
head :: ... => [a] -> a
(++) :: [a] -> [a] -> [a]
```

For more details about Prelude's functions, check the documentation pages for it on Hackage. Inside GHCi, use the commands `:type` (or `:t`) or `:doc`.

## Some Important Prelude Functions

- The `zip` function takes two lists, and returns a list of pairs, where the left hand side of each pair comes from the first list and the right hand side of each pair comes from the second list. For exampl, `zip "ABCDE" [1..5]` evaluates to `[('A',1),('B',2),('C',3),('D',4)('E',5)]`.
  - **Exercise:** implement our own zip function.
- The `unzip` function takes a list of pairs and produces a pair of lists, effectively undoing the

operation of `zip`. So, for example, `unzip [('A',1),('B',2),('C',3),('D',4)('E',5)]` evaluates to `("ABCDE",[1,2,3,4,5])`
  - **Exercise:** implement our own unzip function.
- The `take` function takes an integer `n` and a list `lst` and returns a list containing just the first `n` elements of `lst`. Similarly, `drop n lst` returns a list containing everything in `lst` except the first `n` elements.
  - Implementing these is a lab exercise.
- The `(!!)` function takes a list and an integer `n` and returns the element at index `n` in the list.
  - **Exercise:** implement this function
- The `reverse` function reverses a list.
- The `concat` function, given a list of lists, appends them all together into one list. For example, `concat ["hello"," world"] == "hello world"` .

## Details to Ignore

Many functions in Prelude involve things that are not covered in this course. For example:

```
ghci> :type length
length :: Foldable t => t a -> Int
```

The actual type-signature `(t a -> Int)` is on the right of the double arrow (`=>`). The left of the double arrow says that `t` has to be “foldable”. Lists are foldable, so `t a` can be substituted with `[a]`. Hence, `length` has type `[a] -> Int`.

Similarly,

```
ghci> :type head
head :: GHC.Stack.Types.HasCallStack => [a] -> a
```

The preamble ‘`HasCallStack`’ stuff just means that this function may produce an error. Overall this function still has the type `[a] -> a`.

## Beyond the Prelude

Sometimes we will need to `import` functions from libraries other than the Prelude. You have already seen this with some examples from `Data.Char`. There are also several useful functions in `Data.List` and `Data.Maybe`. As an example, take `catMaybes` from `Data.Maybe`. Given a list of `Maybe` values, say `[Nothing, Just 3, Nothing, Nothing, Just 4, Just 5]`, it will return the list `[3,4,5]` -- every non-`Nothing` value in the input list.

- **Exercise:** implement our own catMaybes

# Style and Testing

## Overview

- **Important Program Qualities**
  1. **Correctness** — via testing and verification
  2. **Maintainability** — through good coding style and clear reports
  3. **Efficiency** — addressed later in the course

These are in order of priority.

## 3. Testing vs. Verification

- **Testing**
  - Involves checking a program against a set of test cases (manually crafted or auto-generated)
  - Reveals the presence of bugs—but not their absence
- **Verification**
  - Uses mathematical reasoning to prove the absence of bugs
  - Highly labor-intensive and beyond the course scope

## Black-Box vs. White-Box Testing

### Black-Box Testing

- Focuses on behavior defined by the specification, **independent of implementation details**
- Test cases can be crafted before code is written
- Design strategy:
  - Group inputs by behavior
  - Select representative cases per group
  - Pay careful attention to edge cases (e.g., boundaries, zero)
  - Examples of input groups:
    - `Int`: zero, positives, negatives, extremes
    - `List`: empty, singleton, longer lists

### White-Box Testing

- Based on the actual implementation
- Identify where the code takes branches (e.g., case expressions, guards, recursion)
- Focus on:
  - Boundary values
  - Overlapping or catch-all (`otherwise`) cases

  - Example:

    ```
    maxThree :: Int -> Int -> Int -> Int
    maxThree x y z
      | x > y && x > z = x
      | y > x && y > z = y
      | otherwise      = z
    ```

    A good case to test: `maxThree 2 2 1` — it probes behavior in overlapping conditions

## 5. Summary of Testing Principles

- No test suite can cover every input.
- Aim to **maximize bug detection** by testing boundary and representative cases.
- Use testing groups to ensure broad coverage of potential input types

## 6. Documenting Test Cases & Tools

- Manual testing (e.g., in GHCI) is useful but not scalable or maintainable.
- Better practice: document tests using **doctest**:

  ```
  -- | Compute the n'th Fibonacci number (counting starts at 1)
  -- >>> fib 10
  -- 55
  -- >>> fib 5
  -- 5
  fib :: Int -> Int
  ```

  Then run: `doctest MyFileName.hs`

## 7. Other Testing Approaches

- **Randomised Testing** can run many test cases automatically but may miss edge cases
- Tools like **QuickCheck** support such strategies, though they are beyond this course's scope

## 8. Code Style & Readability

- Code should be written not just to run, but to be **read, understood, maintained, and extended** by others and your future self
- Code style is assessed in assignments and the final exam

### Commenting

- Explain **what** the code does—not how it works
- Use:
  - `--` for single-line comments
  - `{- ... -}` for multi-line comments

- Example of better commenting:

  ```
  -- Returns the length of the list.
  ```

  *versus*:

  ```
  -- If the list is empty, returns 0.
  -- Otherwise, return 1 plus the length of the tail.
  ```

## Type Declarations

- Type signatures clarify a function's intent and improve readability
- Even though GHCi can infer types, explicit declarations are good style
  Example: `altTake :: [a] -> Int -> [a]`

## Naming & Formatting

- Use meaningful, descriptive names
- Conventions:
  - **lowerCamelCase** for functions and variables
  - **UpperCamelCase** for types and constructors
    Examples to avoid: `f`, `x:xs` when more descriptive names exist

## Case Expressions vs Guards

- Prefer guards for clarity and conciseness:

  ```
  myAbs :: Int -> Int
  myAbs x
    | 0 <= x    = x
    | otherwise = -x
  ```

- Simplify trivial guard functions:

  ```
  isPositive x = 0 < x
  ```

## Additional Style Tips

- Remove dead or unused code (and resolve related warnings)
- Abstract repeated logic into helper functions
- Use Prelude functions when possible
- Keep lines ≤ 80 characters
- Use spaces—not tabs—for indentation—and remain consistent