Haskell Cheatsheet & Study Guide (Chapters 1-3)

This document is a comprehensive study guide for chapters 1-3. It starts with a quick-reference cheatsheet and is followed by a detailed, commented study guide that merges official course concepts with your personal notes and Rust comparisons.

Part 1: Cheatsheet

Chapter 1: Basics, Functions, and Types

Syntax / Concept	Definition	Example
`let...in...`	Defines local variables for an expression. The entire `let` block is an expression.	`let x = 5 in x * 2` → `10`
`where`	Defines local variables for a function, scoped across all guards.	`area r = pi * r^2 where pi = 3.14`
Guards (`|`)	A series of conditional expressions, often cleaner than nested `if/then/else`.	`sign x | x > 0 = 1 | otherwise = 0`
Currying	Functions take one argument and return new functions until all arguments are supplied.	`let add5 = (+) 5`
`(.)` **Composition**	`(g . f) x` is `g(f(x))`. Chains functions right-to-left.	`let odd = not . even`
`($)` **Application**	Low-precedence function application to avoid parentheses.	`sum $ map (*2) [1..10]`

Chapter 2: Lists

List Construction & Basic Properties

Function	Type	Definition	Example
`:` **(Cons)**	`a -> [a] -> [a]`	Prepends an element. O(1).	`1 : [2,3]` → `[1,2,3]`

Function	Type	Definition	Example
`(++)` **(Append)**	`[a] -> [a] -> [a]`	Concatenates two lists. O(n).	`[1,2] ++ [3,4]` → `[1,2,3,4]`
`head`	`[a] -> a`	Returns the first element. **Error on `[]`.**	`head [1,2]` → 1
`tail`	`[a] -> [a]`	Returns all but the first element. **Error on `[]`.**	`tail [1,2]` → `[2]`
`last`	`[a] -> a`	Returns the last element. **Error on `[]`.**	`last [1,2]` → 2
`init`	`[a] -> [a]`	Returns all but the last element. **Error on `[]`.**	`init [1,2]` → `[1]`
`(!!)`	`[a] -> Int -> a`	Returns element at index. **Error on invalid index.**	`[1,2,3] !! 1` → 2
`length`	`[a] -> Int`	Returns the number of elements.	`length [1,2]` → 2
`null`	`[a] -> Bool`	Checks if a list is empty.	`null []` → `True`

Higher-Order Functions

Function	Type	Definition	Example
`map`	`(a -> b) -> [a] -> [b]`	Applies a function to every element.	`map (*2) [1,2]` → `[2,4]`
`filter`	`(a -> Bool) -> [a] -> [a]`	Keeps elements that satisfy a predicate.	`filter even [1,2]` → `[2]`
`zip`	`[a] -> [b] -> [(a,b)]`	Pairs elements from two lists.	`zip [1,2] "ab"` → `[(1,'a'),(2,'b')]`
`concat`	`[[a]] -> [a]`	Flattens a list of lists.	`concat [[1],[2]]` → `[1,2]`
`takeWhile`	`(a -> Bool) -> [a] -> [a]`	Takes elements while a predicate is true.	`takeWhile (<3) [1,2,3,1]` → `[1,2]`

Function	Type	Definition	Example
`dropWhile`	`(a -> Bool) -> [a] -> [a]`	Drops elements while a predicate is true.	`dropWhile (<3) [1,2,3,1]` → `[3,1]`

List Comprehensions

Syntax	Definition	Example
`[out | v <- list, guard]`	A descriptive way to create lists.	`[x*x \\| x <- [1..5], odd x]` → `[1,9,25]`

Chapter 3: Induction & Theory

Concept	Definition
Structural Induction on Lists	Prove for `[]` (Base Case), then prove for `(x:xs)` assuming it holds for `xs` (Inductive Step).
Structural Induction on Trees	Prove for `Null` (Base Case), then prove for `(Node x l r)` assuming it holds for `l` and `r` (Inductive Step).
`T` **(Bottom)**	Represents a non-terminating computation or an error (`undefined`).
Strict Function	A function `f` where `f T = T`. It cannot recover from a broken input.
Non-Strict Function	A function `f` where `f T` can produce a value (e.g., `const 1 T` → `1`).

Part 2: Detailed Study Guide

Chapter 1: Foundations of Haskell

1. Basic Function Definition

In Haskell, a function definition typically consists of a type signature (optional but highly recommended) and an implementation.

- **Type Signature**: `functionName :: ArgumentType -> ReturnType`. The `::` is read as "has type of".

- **Implementation**: `functionName argument = expression`.

```haskell
-- | `myeven` is a function that takes an Int and returns a Bool.
-- | This is conceptually similar to Rust's `fn my_even(n: i32) -> bool`.
myeven :: Int -> Bool

-- | The function body is an expression. Haskell automatically returns the result of the exp
-- | This is similar to the final expression in a Rust block that doesn't end with a semico
myeven n = mod n 2 == 0
```

2. Local Definitions: `where` vs. `let`

Both are used to define local variables, but have different scopes and syntax.

- **`where`**: Scoped over the entire function definition, including all guards. It's written at the end, which can improve readability by keeping helper details out of the main logic.

  ```haskell
  -- | Using `where` to define helper variables.
  -- | There is no direct equivalent in Rust, but it can be thought of as helper variable
  -- | defined at the end of a function for clarity.
  paymentWithWhere :: Int -> Int
  paymentWithWhere weeks = salary
    where
      days = weeks * 5
      hours = days * 8
      salary = hours * 130
  ```

- **`let...in...`**: Is itself an expression. The variables are only in scope between the `let` and the `in`. This allows for more localized definitions.

  ```haskell
  -- | `let...in...` is an expression, similar to Rust's block expressions:
  -- | `let result = { let x = 5; x + 1 };`
  paymentWithLet :: Int -> Int
  paymentWithLet weeks =
    let
      days = weeks * 5
      hours = days * 8
      salary = hours * 130
    in
      salary
  ```

3. Guards

Guards provide an elegant way to perform multi-branch conditional logic, often preferred over `if-then-else`.

```haskell
-- | The `|` is very similar to a guard `if` in a Rust `match` arm.
-- | `match weeks { w if w > 19 => ..., _ => ... }`
```

```haskell
-- | `otherwise` is a catch-all, equivalent to Rust's `_`.
paymentWithGuards :: Int -> Int
paymentWithGuards weeks
  | weeks > 19 = round (fromIntegral baseSalary * 1.06)
  | otherwise  = baseSalary
  where
    baseSalary = weeks * 5 * 8 * 130
    -- `fromIntegral` is for numeric type conversion, like Rust's `as` keyword.
```

4. Currying and Partial Application

This is a core difference from Rust. All Haskell functions are curried, meaning they can be partially applied.

```haskell
-- | The type `Int -> Int -> Int` is read as `Int -> (Int -> Int)`.
-- | It's a function that takes an Int and returns a NEW function of type `Int -> Int`.
smaller :: Int -> Int -> Int
smaller x y = if x <= y then x else y

-- | We can partially apply `smaller` by giving it only one argument.
-- | `st3` is now a new function that finds the smaller of a number and 3.
st3 :: Int -> Int
st3 = smaller 3
```

5. Higher-Order Functions & Composition

Functions that take other functions as arguments or return them.

- **Function Composition `(.)`**: `(g . f) x` is `g(f(x))`. It chains functions together, executing from right to left. Rust does not have a built-in operator for this.

  ```haskell
  square :: Int -> Int
  square x = x * x

  -- `twice_composed` takes a function `f` and returns a new function `f . f`.
  twice_composed :: (a -> a) -> (a -> a)
  twice_composed f = f . f

  -- This point-free style is very common in Haskell.
  quad_v2 :: Int -> Int
  quad_v2 = twice_composed square
  ```

- **`lift2` Example**: A more general higher-order function.

  ```haskell
  -- | `lift2` "lifts" a binary function `h` to operate on the results of two unary funct
  lift2 :: (b -> c -> d) -> (a -> b) -> (a -> c) -> a -> d
  lift2 h f g x = h (f x) (g x)
  ```

```haskell
-- | To compute `(x+1) * (x+2)`
polyCompute :: Int -> Int
polyCompute = lift2 (*) (+1) (+2)
```

Chapter 2: Mastering the List

1. The Nature of Lists

A list is a recursive data structure.

- **Definition**: `data [a] = [] | a : [a]`. A list is either empty (`[]`) or an element `a` prepended (`:`) to another list.
- **Rust Comparison**: This is very similar to a manually implemented `enum List<T> { Nil, Cons(T, Box<List<T>>) }`.
- **Syntax Sugar**: `[1, 2, 3]` is just a cleaner way to write `1 : (2 : (3 : []))`.

2. List Generation

- **Ranges**: `[1..10]`, `[2, 4 .. 10]`, `['a'..'f']`
- **List Comprehensions**: A descriptive syntax for creating lists, combining `map` and `filter` concepts.

```haskell
-- | Find the squares of all odd numbers from 1 to 10.
-- | `x <- [1..10]` is the generator.
-- | `odd x` is the guard (filter).
-- | `x*x` is the output expression (map).
comprehensionExample :: [Int]
comprehensionExample = [x*x | x <- [1..10], odd x]
```

3. Core List Operations

This section covers the most important list functions, many of which have direct parallels to Rust's `Iterator` methods.

- **Basic Operations**: `head`, `tail`, `last`, `init`, `length`, `null`, `(!!)`.
- **Higher-Order Functions**: `map`, `filter`, `zip`, `concat`, `takeWhile`, `dropWhile`.

4. Lazy Evaluation and Infinite Lists

- **Lazy Evaluation**: Expressions are not evaluated until their results are needed.
- **Infinite Lists**: This makes infinite lists a practical tool in Haskell.

```haskell
-- | The infinite list of natural numbers.
naturals :: [Int]
```

```haskell
naturals = [1..]

-- | Because of laziness, this computation is fine. `take` only requests 5 elements,
-- | so only the first 5 elements of `naturals` are ever generated.
fiveNaturals :: [Int]
fiveNaturals = take 5 naturals -- -> [1,2,3,4,5]
```

5. Application: Caesar Cipher (Algorithm)

The logic to `crack` the cipher without a key is a great example of combining these tools.

Goal: Find the key `k` that makes the decoded text look most like English.

Algorithm:

```
function crack(ciphertext):
  // 1. Generate all 26 possible decoded texts
  possible_texts = []
  for k from 0 to 25:
    decoded = encode(-k, ciphertext)
    add decoded to possible_texts

  // 2. Score each possible text
  scores = []
  for each text in possible_texts:
    // The score measures how close the letter frequency is to standard English.
    // Lower score is better.
    s = calculate_chisqr_score(text)
    add s to scores

  // 3. Find the best score and its corresponding key
  best_score = minimum value in scores
  best_key = index of best_score in scores

  return best_key
```

This algorithm is implemented in Haskell by mapping over a list of keys `[0..25]` instead of using loops.

Chapter 3: Definition and Proof by Induction

1. The Core Idea: Proving is Programming

The structure of a recursive function on a data type mirrors the structure of an inductive proof on that same data type.

2. Structural Induction on Lists

To prove a property `P` for all lists `xs`, you must prove: 1. **Base Case**: `P []` holds. 2. **Inductive Step**: Assuming `P xs` holds (the **Inductive Hypothesis**), prove that `P (x:xs)` also holds.

This exactly matches the pattern of a recursive list function:

```
f [] = ...       -- Base Case
f (x:xs) = ... -- Inductive Step, often with a recursive call on `xs`
```

3. Example Proof on Lists: `length (xs ++ ys) = length xs + length ys`

- **Base Case (`xs := []`)**: `length ([] ++ ys) = length ys` (by def of `++`) The right side is `length [] + length ys = 0 + length ys = length ys`. They match.
- **Inductive Step (`xs := x:xs'`)**:
  - **IH**: `length (xs' ++ ys) = length xs' + length ys`
  - **Goal**: `length ((x:xs') ++ ys) = length (x:xs') + length ys`
  - **Proof (LHS)**: `length ((x:xs') ++ ys) = length (x : (xs' ++ ys))` (by def of `++`) `= 1 + length (xs' ++ ys)` (by def of `length`) `= 1 + (length xs' + length ys)` (by **IH**) This matches the RHS, since `length (x:xs')` is `1 + length xs'`.

4. Example Proof on Trees: `minT (mapT (n+) t) = n + minT t`

This proof demonstrates induction on a custom `Tree` data type.

- **Data Types**: `haskell data Tree a = Null | Node a (Tree a) (Tree a) mapT f (Node x l r) = Node (f x) (mapT f l) (mapT f r) minT (Node x l r) = x `min` (minT l) `min` (minT r)`
- **Base Case (`t := Null`)**:
  - LHS: `minT (mapT (n+) Null)` → `minT Null` → `maxBound`
  - RHS: `n + minT Null` → `n + maxBound`
  - `maxBound` is equivalent to `n + maxBound` (as infinity). The base case holds.
- **Inductive Step (`t := Node x left right`)**:
  - **IH**: Assume it holds for `left` and `right`.
    - `minT (mapT (n+) left) = n + minT left`
    - `minT (mapT (n+) right) = n + minT right`
  - **Proof (LHS)**: `minT (mapT (n+) (Node x left right)) = {def of mapT} minT (Node (n+x) (mapT (n+) left) (mapT (n+) right)) = {def of minT} (n+x)`min`minT (mapT (n+) left)`min`minT (mapT (n+) right) = {apply IH to left and`

`right subtrees}` `(n+x)`min`(n + minT left)`min`(n + minT right)` `= {property: (n+a)`min`(n+b) = n + (a`min`b)}` `n + (x`min`minT left`min`minT right)` `= {def of minT}` `n + minT (Node x left right)`

– The result matches the RHS. The proof is complete.