Haskell Lists: The Ultimate Guide

Work In Progress

This guide is "Work in progress"

Haskell Lists: Two big Caveats

There are ***two major differences*** in Haskell lists, compared to other languages, especially dynamically typed languages, like Python, Ruby, PHP, and Javascript.

1. First, lists in Haskell are ***homogenous***. This means that a Haskell list can only hold elements of the ***same type***
2. Second, lists in Haskell are (internally) implemented as ***linked lists***. This is different from many other languages, where the word "list" and "array" is used interchangably. Linked lists and arrays have very different performance characterstics when operating on large amounts of data. Keep this in mind for the future. If you're dealing with very large lists, where you want to randomly access the `i`-th element multiple times, use a `Vector` instead.

Constructing lists in Haskell

There are ***five different*** ways to construct lists in Haskell:

1. **Square-bracket syntax:** This is the simplest and most recognisable way.

```haskell
-- A list of numbers
let a = [1, 5, 7, 12, 56]

-- A list of booleans
let b = [True, False, False, True]
```

2. **Colon operator:** This is very similar to the `cons` function from Lisp-like languages. It adds a ***single element*** to the ***beginning*** of a ***list*** (and returns a new list).

```haskell
-- A list containing a single element
-- This is the same as saying `[99]`
let a = 99:[]
```

```haskell
-- Keep adding single elements to the beginning of the list
-- to progressively get a larger list
let a = 1:5:6:12:[]

-- A list of booleans
let b = True:False:False:True:[]
```

3. **Using ranges:** This is short-hand for defining a list where the elements **TODO**

4. **List comprehension:** If you are starting out with Haskell, I would **strongly recommend against** using list comprehensions to construct lists. They seem like cool feature, but I find them very opaque and unmaintable. Similar to complex regular expressions - write once, read never! Nevertheless, there is a section dedicated to list comprehensions in Haskell for the sake of completeness.

5. **Monoid interface:** The most "complicated", but often used way of defining a list is via its *Monoid* interface. There is a section dedicated to the Monoid interface of lists if you'd like to know more.

```haskell
mempty :: [Int] == []
```

Pattern-matching of Haskell lists

There are ***two ways*** to pattern-match over a list in Haskell, and there's a subtle difference between them.

1. Using the `:` constructor:

```haskell
-- Return the first element of a list, taking care of the edge-case where
-- the list may be empty. Which is why the result is a (Maybe a)
firstElem :: [a] -> Maybe a
firstElem xs = case xs of
  [] -> Nothing
  -- Remember to put parantheses around this pattern-match else
  -- the compiler will throw a parse-error
  (x:_) -> Just x
```

```haskell
-- Return the second element of a list
secondElem :: [a] -> Maybe a
secondElem xs = case xs of
  -- Remember the parantheses
  (_:y:_) -> Just y
  (_:[]) -> Nothing
  [] -> Nothing
```

2. Using the `[]` constructor:

```
-- Return the first element of a list
firstElem :: [a] -> Maybe a
firstElem xs = case xs of
  -- Remember to put parantheses around this pattern-match else
  -- the compiler will throw a parse-error
  (x:_) -> Just x
  [] -> Nothing
```

Subtle difference between `:` and `[]` when pattern-matching

With `:` you can pattern-match a list with any number of elements. This is because the last `:` matches the *remainder of the list.* Whereas, with `[]`, you can only pattern match a list with an exact number of elements.

For example, to pattern-match a list into (a) first element, (b) second element, and (c) everything else, you can use the `:` operator as demonstrated below...

```
-- in the following expression:
-- x = 1
-- y = 5
-- z = [7, 12, 45]
let (x:y:z) = [1, 5, 7, 12, 45]
```

... however, there is no way to write a similar expression using `[]`. If you try, you'll get an error:

```
-- the following will always throw an error...
let [x, y, z] = [1, 5, 7, 12, 45]
```

```
-- ...while the following will work
let [x, y, z] = [1, 5, 7]
```

If you need to, you can also use `:` to match a list with an exact number of elements. In fact, in the secondElem example above, we've used it to match a list with exactly *one* element. Here's an example of how to use it to pattern-match on a list with exactly *two* elements:

```
let x:y:[] = [1, 2]
```

Be careful how you use this. The following will always throw an error because you are forcing the *last* `:` to match with a `[]` (empty list), but instead it gets a `[3]` (list with single element `3`). If you want this to work, you'll have to go back to the first example in this section.

```
-- the following will always throw an error...
let x:y:[] = [1, 2, 3]
```

Here's a complex example using both kinds of pattern matching. This converts a given list into a English phrase, such as "x, y, and z". For the four special cases (where the length has three, or fewer, elements) we use `[]`, whereas for the most general case, we use `:`

```
-- Complex example using multiple list-related functions
let x = [1, 5, 20, 77, 45, 67]
in case x of
  [] -> "(none)"
  [a] -> show a
  [a, b] -> show a ++ " and " ++ show b
  [a, b, c] -> show a ++ ", " ++ show b ++ ", and " ++ show c
  (a:b:c) -> show a ++ ", " ++ show b ++ ", and (" <> (show $ length c) <> ") more"
```

Iterating over a Haskell list

If you're starting out, you'd be surprised to know that there is ***no way to "iterate" over a list in Haskell***, in a way that you might already be familiar with. To be specific, there's no way to do the following in Haskell:

```
// Familiar for-loops are NOT possible in Haskell!

var lst = [1, 5, 7, 2, 8, 10];
var sum = 0;
for(i = 0; i < lst.length; i++) {
  sum = sum + lst[i];
}
```

If your thought-process requires you to iterate over a list, step back and think about *why* you need to it. Merely iterating over a list is not interesting; what you *do* in each iteration is the interesting part. And the `Data.List` module has a rich set of functions which help you visit and *do something* with each element in a list, without having to write a `for(i=0; i<l; i++)` boilerplate each time.

Keep this in mind when you're reading about the various operations you can do with lists. Haskell ***almost forces*** you to express your solution using a higher-level API, instead of dropping down to a `for`-loop every time. Get familiar with the `Data.List` API - you will be using it a **lot** when writing real-world Haskell code.

> The closest that you can get to a `for`-loop in Haskell, is the `foldl` (or `foldr`) function. Almost every other function in `Data.List` can be written using this function. Or, you always have the option of implementing any iteration as a recursion - that's really the "lowest level" of getting this done - but it is not the idiomatic way of doing simple data transformations in Haskell.

Appending / Joining / Growing Haskell lists

There are ***four*** ways to join / concatentate / append / grow Haskell lists:

`(++) :: list1 -> list2 -> joined-list`

When you have a few known lists that you want to join, you can use the `++` operator:

```
[True, False] ++ [False, False]
```

```
[1, 2, 3] ++ [4, 5, 6] ++ [10, 20, 30, 50, 90]
```

You can also use the `++` operator in it "prefixed function" form. This is useful short-cut when you want to pass it to another function, such as a `foldl`, and don't want to write the verbose `(\x y -> x ++ y)`

```
-- you need to put parantheses around the operator otherwise Haskell
-- will throw a parse-error
(++) [1, 2, 3] [4, 5, 6]
```

`concat :: list-of-lists -> joined-list`

While `++` is useful to join a fixed/known number of lists, sometimes you're dealing with an unknown/varying number of lists. In all probability you will represent them as a "list of lists". To join them together, use the `concat` function:

```
concat [[1, 2, 3, 4, 5], [10, 20, 30, 40], [100, 200, 300]]
```

`(:) :: element -> list -> consed-list`

The `:` operator is also known as a the `cons` operation, is actually a constructor of the `[]` type (it's a subtle fact that you don't need to bother with for most use-cases). Use it when you want to ***add a single element to the beginning of a list***

```
1:[3,4,5]
```

Be careful, that the single element comes first, and the list comes next.

```
let x = [10, 20, 30]
    y = 99
in y:x
```

You can also cons on top of an empty list. The example given below is the same as saying `[999]`

```
999:[]
```

`intercalate :: delimeter -> list -> joined-list`

This function is *typically* used with a list of `String`s where you want to join them together with a comma, or some other delimiter. Remember that a String is a type-synonym for [Char], so when `intercalate` is used with strings the type-signature specializes to: `[Char] -> [[Char]] -> [Char]`, which is the same thing as `String -> [String] -> String`

```
intercalate ", " ["Haskell", "Tutorials", "Are", "Awesome"]
```

Do not confuse `intercalate` with the similarly named `intersperse`. The latter **does not** join lists.

Determining the length of a Haskell list

TODO

Finding a single element in a Haskell list

There are four commonly used ways to find a single element in a list, which vary slightly.

`find :: condition -> list -> Maybe element`

The most general function for finding an element in a list that matches a given condition. Two things to note about this function:

1. It returns a `Maybe a`, because it is possible that no element matches the given condition.
2. It return the **first matching element**

```
-- Find the first element greater than 10
find (\x -> x > 10) [5, 8, 7, 12, 11, 10, 99]
```

The following example is the same as the previous one, just written in a point free syntax.

```
-- Find the first element greater than 10
find (> 10) [5, 8, 7, 12, 11, 10, 99]
```

A slightly more complex example where we do something on the basis of whether an element exists in a list, or not (remember, the result is not a `Bool`, but a `Maybe a`):

```
-- Find the first user that has an incorrect age (you can possibly
-- use this to build some sort of validation in an API)
let users = [("Saurabh", 35), ("John", 45), ("Doe", -5)]
```

```haskell
in case (find (\(_, age) -> age < 1 || age > 100) users) of
  Nothing -> Right users
  Just (name, age) -> Left $ name <> " seems to have an incorrect age: " <> show age
```

`elem :: element -> list -> Bool`

Use `elem` if you want to check whether a ***given element*** exists within a list. Two important differences with `find`:

1. The first argument to `elem` is the actual element you're searching for, whereas for `find` it is a *condition* to be applied to each element. `elem` **implicitly uses the `==` operator** to check for equality when searching for the given element, which is why there is an `Eq a` type-class constraint in its type-signature.
2. In the case of `elem` the result is a `Bool`, since you've already specified the exact element you're searching for. Whereas in the case of `find` the result is a `Maybe a` because you don't know beforehand which element will match the given condition.

```haskell
elem 5 [1, 2, 5, 10]
```

Usually, `elem` is used in its infix form, because it is easier to *verbalize* mentally.

```haskell
5 `elem` [1, 2, 5, 10]
```

```haskell
("Saurabh", 35) `elem` [("Saurabh", 35), ("John", 45), ("Doe", -5)]
```

`notElem :: element -> list -> Bool`

`notElem` is the negation of elem

`any :: condition -> list -> Bool`

`any` lies in the "middle" of `find` and `elem`. It allows you to specify your own condition (like `find`), but simply returns a `True`/`False` (like `elem`) depending upon whether a match was found, or not.

```haskell
let users = [("Saurabh", 35), ("John", 45), ("Doe", -5)]
in if (any (\(_, age) -> age < 1 || age > 100) users)
  then Left "Some user has an incorrect age. Please fix the input data"
  else Right users
```

Filtering / Rejecting / Selecting multiple elements from a Haskell list

There are three general ways to filter / reject / select multiple elements from a Haskell list:

- You want to go through the *entire* list and decide whether the element should be present in the resultant list, or not. You'd want to use filter for this.
- You want to extract the first (or last) `N` elements of a list (and `N` is independent of the contents of the list). You'd want to use `take` or `takeEnd` in this case.
- You want to stop selecting elements (basically terminate the iteration) as soon as a condition is met. For example:
  - Select all elements from the ***beginning*** of a list of `Char` till you reach a delimiter, say `,`. This could be the core of a home-grown CSV parser (although you shouldn't have to write it -- there's a library for that!)
  - You'd use one of `takeWhile`, `dropWhile`, or `dropWhileEnd` in this case.

> Why is there no `takeWhileEnd`?

`filter :: condition -> list -> filtered-list`

The `filter` function ***selects*** *all* elements from a list which satisfy a given condition (predicate).

> This function is unfortunately named, because *filter* could mean either the act of *selecting*, or the act of *removing* elements based on a condition. I *still* get confused about which it is!

```
-- select all even elements from a list
filter (\x -> x `mod` 2 == 0) [1..20]
```

```
-- A more complex example that uses `filter` as well as `null`
let users = [("Saurabh", 35), ("John", 45), ("Trump", 105), ("Biden", 88), ("Doe", -5)]
    incorrectAge = filter (\(_, age) -> age < 1 || age > 100) users
in if (null incorrectAge)
   then Right users
   else Left $ "Multiple users seem to have an incorrect age: " <> show incorrectAge
```

`take :: number-of-elements-to-take -> list -> shorter-list`

`take` is used to take the first `N` elements from the beginning of a list. If `N` is greater than the list's length, this function will NOT throw an error. It will simply return the entire list.

```
-- Simple example
let x = [1, 5, 20, 77, 45, 67]
in take 3 x
```

```
-- `N` is greater than the list length
let x = [1, 5, 20, 77, 45, 67]
in take 10 x
```

> If you'd like to look at just the first element of the list, use one of the following methods instead:

- Pattern match on the first element
- `null` - if you just want to check if the list is non-empty.
- `uncons` - if you want to split the list into it's "head" (first element) and "tail" (remaining elements)
- `head` - **Caution:** Use this if, and only if, you are 100% sure that the list is non-empty. This function is infamously unsafe, and will throw an error on empty lists.

`drop :: number-of-elements-to-drop -> list -> shorter-list`

`drop` removes the first `N` elements from a given list. If `N` is greater that the list's length, an empty list will be returned.

`takeWhile :: condition -> list -> shorter-list`

Keep taking (selecting) elements from the beginning of a list as long as the given condition holds true.

Here's how you can keep selecting `Char`s till you encounter a `,`:

```
-- keep selecting elements from a [Char] till we encounter a comma
takeWhile (\x -> x /= ',') ['H', 'e', 'l', 'l', 'o', ',', 'W', 'o', 'r', 'l', 'd']
```

Same example, but using the familar syntax of writing a `String`, which is a type-synonm for `[Char]`

```
-- keep selecting elements from a [Char] till we encounter a comma
takeWhile (\x -> x /= ',') "Hello,World"
```

Same example, but in *point-free* syntax:

```
-- keep selecting elements from a [Char] till we encounter a comma
takeWhile (/= ',') "Hello,World"
```

`dropWhile :: condition -> list -> shorter-list`

`dropWhile` is similar to `takeWhile`, but instead of *selecting* elements based on the given condition, it *removes* them from the *beginning* of the list instead.

`dropWhileEnd :: condition -> list -> shorter-list`

`dropWhileEnd` is similar to `dropWhile`, but instead of removing elements from the *beginning* of the list, it removes them from the *end* instead.