COMP105 Lecture 5

Lists

Lists

A **list** contains items that **all have the same type**

Examples:

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

```
['a', 'b', 'c', 'd', 'e']
```

```
["Quite", "a", "lot", "of", "words"]
```

The following will give an **error**:

```
[1, "two", 3, "four"]
```

Lists

A list can have **any** number of elements, including zero

```
[]
```

```
[1]
```

```
[1, 2]
```

You can **join** lists with the ++ operator

```
ghci> [1,2,3] ++ [4,5,6]
[1,2,3,4,5,6]
```

Strings

In Haskell, a **string** is just a list of characters

```
ghci> ['a', 'b', 'c']
"abc"
```

Any operation that you can do on a list can also be done on a string

```
ghci> "Hello" ++ " World!"
"Hello World!"
```

List indexing

The `!!` operator gets a specified element from the list

- Lists are zero-indexed

```
ghci> [1, 2, 3, 4, 5] !! 1
2
```

Note that Haskell lists are **linked lists**

- This means that random access is expensive
- Internally Haskell will walk the entire list to get the last element

Processing Lists

Since Haskell uses **linked lists** we usually process lists **from the front**

The **head** of a list is its first element

```
ghci> head [1,2,3,4,5]
1
```

The **tail** of a list is everything but the first element

```
ghci> tail [1,2,3,4,5]
[2,3,4,5]
```

Processing Lists

The `:` operator glues a new head onto an existing list

```
ghci> 1 : [2,3,4,5]
[1,2,3,4,5]
```

In fact we can build up lists using nothing but `:` and `[]`

```
ghci> 'a' : ('b' : ('c' : []))
"abc"
```

Processing Lists

We can also process lists **from the back**

The function **last** gives the last element of the list

```
ghci> last [1,2,3,4,5]
5
```

The function **init** gives everything but the last element of the list

```
ghci> init [1,2,3,4,5]
[1,2,3,4]
```

These are like `head` and `tail` for the back of the list

Writing our own list functions

Lists can be passed as parameters to functions

```
double_head list = 2 * head list
```

```
ghci> double_head [1,2,3]
2
```

We can also make use of **pattern matching** to get the head of the list

```
triple_head (x:xs) = 3 * x
```

```
ghci> triple_head [3,2,3]
9
```

Pattern Matching

```
triple_head (x:xs) = 3 * x
```

When you pass a list to the function, Haskell will match it using `x:xs`

- `x` will be bound to the head of the list
- `xs` will be bound to the tail

If the pattern cannot be matched, then you will get an error

```
ghci> triple_head [1]
3
ghci> triple_head []
*** Exception
```

Pattern matching

Pattern matching is quite **flexible**

```
mult_first_two (x:y:xs) = x * y
```

Here `x` is bound to the first element and `y` is bound to the second element

If you don't care about an argument you can use the **wildcard** pattern `_`

```
double_second (_:y:_) = 2 * y
```

Useful list functions

```
ghci> length [1,2,3]
3
```

```
ghci> reverse [5,4,3,2,1]
[1,2,3,4,5]
```

```
ghci> sum [5,2,1,6,3,2,5,7]
31
```

```
ghci> product [6,2,1,2]
24
```

Useful list functions

take returns the first x elements of a list

```
ghci> take 3 [5,4,3,2,1]
[5,4,3]
```

drop returns all but the first x elements of a list

```
ghci> drop 3 [5,4,3,2,1]
[2,1]
```

Useful list functions

elem returns True if the specified element is in the list

```
ghci> elem 4 [3,4,5,6]
True
ghci> elem 10 [3,4,5,6]
False
```

It is perhaps more naturally used as an infix operator

```
4 `elem` [3,4,5,6]

10 `elem` [3,4,5,6]
```

Exercises

1. Write a function `thricesum` that takes a list of numbers and returns three times its sum

2. Use pattern matching to write a function `thirdelement` that takes a list and returns the third element of the list (do not use `!!`)

3. Write a function `exclaim` that takes a string and returns a copy of that string with the `'!'` character at the start and end. So `exlaim "hi"` will return `"!hi!"`