

Forum:[Hack Forums](#)

Thread: [Haskell - Part 2 - Recursion, comments, lists and tuples \[LEGENDS\]](#)

SIM-2015410 - September 5, 2015, 12:34 am

[img]http://s9.postimg.org/57pm7cwyr/LEGENDS_2_0_HEADER.png[/img]

Haskell - Tutorial 2

Recursion, comments, lists and tuples

In this tutorial, I will go over recursion in Haskell, show how comments are written, and give an introduction to another major component of the language: lists.

[img]http://s9.postimg.org/jzubyp2v3/LEGENDS_2_0_DELIMITER.png[/img]

Recursion

In the previous tutorial, I mentioned that you cannot change the value of something once you have defined it. Those of you who come from OOP or imperative language backgrounds may think of this as a roadblock at first, but I will demonstrate the fact that this isn't necessary in Haskell.

Let's begin with an example. Here is a basic factorial function in Haskell:

Code factorial :: Int -> Int

factorial 0 = 1

factorial x = x * factorial (x-1)

I will point out a few new things: We have defined 3 separate function bodies. This is called pattern matching, and acts as an 'edge case' for recursion. If we call factorial with 0 as a parameter, we get 1 returned to us but if the parameter doesn't match 0, it will drop through to the final function.

This function is a recursive function, because it calls itself each iteration until it gets to 0

To see how this works, we shall work through an example of how this program would run:

Code factorial 3 --not 0, not 1 so we use the third function body.

= 3 * factorial (3-1)

and factorial 2 = 2 * factorial 1

and factorial 1 = 1 * factorial 0

But now, factorial 0 = 1 matches so we end up with:

factorial 3 = 3 * (2 * (1 * (1))) = 6 which is exactly what we wanted.

[img]http://s9.postimg.org/jzubyp2v3/LEGENDS_2_0_DELIMITER.png[/img]

Comments

Single line comments in Haskell look like this:

Code -- this is a comment

multiline comments in Haskell look like this:

Code {- this

is a multiline

comment -}

[img]http://s9.postimg.org/jzubyp2v3/LEGENDS_2_0_DELIMITER.png[/img]

Lists

Lists are a very important part of Haskell programs, so I will cover them here before we get on to list comprehensions in the next tutorial. A list is similar to what is called an 'array' in Java, all elements in the list have to be the same type (e.g. Int, Char, Bool, or even lists of lists, lists of functions - the functions must all have the same type, etc.). This is what a list looks like in Haskell:

```
Code coolList = [1,2,5,73,2]
```

This is an empty list:

```
Code emptyList = []
```

To prepend elements to a list, we use the `:` operator like this:

```
Code 1:[2,3,4] == [1,2,3,4]
```

```
1:2:3:[] == [1,2,3]
```

notice the second example, it shows hows lists are truly defined in Haskell

To combine two lists into one, we use the `++` operator like this:

```
Code [1,2,3] ++ [4,5,6] == [1,2,3,4,5,6]
```

--This also works for strings, i.e. a list of characters:

```
"hel" ++ "lo" == "hello"
```

An example of a list of functions, all of which have type `(a -> a)`:

```
Code anotherList = [(+ 3), addTwo, (addTogether 5)]
```

notice the partial application of the `addTogether` function we defined in the last tutorial, this means it has type `(a -> a)`

A string is just a list of characters, so these two declarations are the same:

```
Code listOne = ['h','e','l','l','o']
```

```
listTwo = "hello"
```

these lists both have type `[Char]`

A list of list of lists:

```
Code [ [ [1,2], [3] ], [ [4,5,6,7], [8,9], [10,11] ] ]
```

this list has type `[[[Int]]]`

We can retrieve the `nth` element of a list like this:

```
Code aList = [2,5,2,1,3,3,2,5]
```

```
fifthElement = aList !! 5
```



Tuples

Tuples are groups of values, similar to lists, but the values inside them can be of different types. Here is an example of a 2-tuple, better known as a pair:

```
Code aTuple = ("Hello", 45)
```

This tuple has the type `([Char], Int)`

We can also have longer tuples, here is an example of a 3-tuple or a triple:

```
Code aTriple = (1,True, 700)
```

This tuple has type `(Int, Bool, Int)`

We can also have lists of tuples, for example:

```
Code [(1,"Apples"), (2, "Oranges"), (3, "Milk")]
```

This list has type `[(Int, [Char])]`



This tutorial was written by MeshCollider ?

Nutar - September 5, 2015, 12:39 am

Great tutorial you've got there bud. Keep it up. You really got me into Haskell.

g4143 - September 5, 2015, 1:07 am

Important note for list and recursion.

List are recursive by their nature.

Remix - September 5, 2015, 1:28 am

Nice tutorial

Keep going ?

csrk - September 5, 2015, 2:09 am

(Sep 5, 2015 12:34 AM)?eshCollider Wrote: [\[CITE\]](#) Lists are a very important part of Haskell programs, so I will cover them here before we get on to list comprehensions in the next tutorial. A list is similar to what is called an 'array' in Java I'd like to expand your list explanation a bit!

Lists are actually not that similar to arrays in java, they are more similar to LinkedLists in java. Lists in Haskell are defined something like this

Code -- Nil = empty node/list, Cons a (List a) = element a followed by the rest of the list.

data List a = Nil | Cons a (List a)

In java you can index an array and retrieve any value in $O(1)$ time (Big-O notation), meaning that it's a constant time. In Haskell, when you are asking for an item in a list, you will have $O(n)$ worst time complexity (where n is the number of items in the list). You have to first check if the first item is the one you want, and if it's not, check the rest of the list recursively. In java a LinkedList is defined as a head followed by the next item, so you have to loop through them in a similar fashion.

This definition has both advantages and disadvantages, the most obvious disadvantage being the one I mentioned above. One advantage is that it's very easy to map functions to every element in the list.

the function map is defined like this. Code `map :: (a -> b) -> [a] -> [b]` It takes as argument a function and a list, and performs that function on every item in the list and returns the new list.

Code `map (+2) [2,3,4,5]`

-- returns [4,5,6,7] Notice how Haskell allows for function declarations to contain another function's declaration. This allows for a much stronger typecheck.

Nex+ - September 5, 2015, 8:01 am

Very nice tutorial you got here Mesh