

Forum:[Hack Forums](#)

Thread: [Haskell - Part 1 - Functions, If statements and values \[LEGENDS\]](#)

SIM-2015410 - September 3, 2015, 2:20 pm

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

Haskell - Tutorial 1

Functions and If statments

I think it appropriate to begin this tutorial series with a tutorial on functions - the basis for functional languages.

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

How functions are defined

In Java or C# style syntax, we would define a function like this:

```
Code returnType functionName (parameters) {  
  //function body  
  return someValue;  
}
```

In Haskell, we define a function like this:

Code functionName parameters = someValueWe pretty much define the function to be something rather than do something, and if you call the same function twice with the same parameters, you MUST get the same result. Otherwise, the function is not 'referentially transparent' and has 'side effects'. This is one of the big ways Haskell is so good at parallel processing. All 'pure' functions do not change the state of things outside the function so they do not affect each other - and therefore it doesn't matter in what order they are run.

Here is a really basic example program to start with, it takes one parameter (x) and 'returns' (x + 2)

Code addTwo x = x + 2

It is usually a good idea to write explicit types for all your functions to make sure they work as expected, so here is how we do that:

Code addTwo :: a -> a

addTwo x = x + 2

This now says that we have a function which takes a parameter of type a and returns a value of the same type. But a could be anything - a character, an integer, a boolean, even an IO action. Our function obviously only works with numbers because it doesn't make sense to add 2 to false. So we can add in a Type Constraint, to say that a has to be a member of the typeclass Num (all numeric types are a part of the Num typeclass).

Code addTwo :: (Num a) => a -> a

addTwo x = x + 2

Now, let's define a function which takes 2 parameters. This will introduce a new concept, called 'currying', but don't worry if you don't fully understand it yet. We will define a function which takes 2 variables and adds them together:

Code addTogether :: (Num a) => a -> a -> a

```
addTogether x y = x + y
```

That seems pretty straightforward, except for the type definition. Why so many arrows? For now you can think about this as just taking a parameter, then taking another parameter, then returning a value. Because, if you want to get into the picky details, all Haskell functions only take one parameter! In reality, this `addTogether` function takes a parameter, and returns a new function which takes a parameter, and returns a value. This is known as currying. Let's work through an example. Let's say we want to add 5 and 6 together using this function. We would do that like this:

Code Prelude> `addTogether 5 6`
`addTogether` first takes the 5 and returns a new function which looks like this:

```
Code addTogether :: (Num a) => a -> a
```

`addTogether y = 5 + y`The function has been 'partially applied'. To show this in another context, let's redefine the `addTwo` function we saw before to use our new function:

```
Code addTwo' :: (Num a) => a -> a
```

```
addTwo x = addTogether 2 x
```

[img]http://s9.postimg.org/jzubyp2v3/LEGENDS_2_0_DELIMETER.png[/img]

If statements

Let's look now at If statements, a vital part of programming logic.

```
Code eatHowMany :: Int -> Int
```

```
eatHowMany hunger = if (hunger > 5)
```

```
    then 2
```

```
    else 1
```

Pretty self explanatory, it is just a simple if-then-else statement. The function takes a 'hunger' amount (which is just an integer), and if it is more than 5 we return 2 otherwise we return 1. A couple of things to note: We have used `Int` instead of `a` in the type definition of the function, because we only want to accept `Int` values

Indentation is important in Haskell, and it improves readability if the then and else clauses line up. Unfortunately, HackForums code boxes aren't the best for indentation and formatting.

We can also put the entire if statement on one line like this:

Code `eatHowMany hunger = if (hunger > 5) then 2 else 1` Personally, I think this is fine for short statements but with longer statements, it is a lot better to use multiple lines to make it more readable and easier to debug.

[img]http://s9.postimg.org/jzubyp2v3/LEGENDS_2_0_DELIMETER.png[/img]

Constants

In a functional languages, constants are just functions which don't take any parameters. So they are defined in the same way. We usually don't bother with a type definition, but it is perfectly fine if you do:

```
Code numberTwo :: Int
```

```
numberTwo = 2
```

There is no such thing as a 'variable' in Haskell, because once you declare something with a certain value, you cannot change it. If you could change the value of a variable inside a function, that function would then have side effects and wouldn't be pure. You might see this as a limitation at the moment if you come from OOP or imperative languages, but this isn't a limitation - it is just an example of the differences between the 2 paradigms. We will see later how this plays out in functional languages when we come to recursion.

[img]http://s9.postimg.org/jzubyp2v3/LEGENDS_2_0_DELIMETER.png[/img]

This tutorial was written by MeshCollider ?

Unknown - September 3, 2015, 2:35 pm

Great thread! I've always wanted to learn Haskell

scam. - September 3, 2015, 9:55 pm

Nice one mate. I'd like to learn haskell just as a learning experience. Looks really interesting but seems a little confusing.

g4143 - September 3, 2015, 10:01 pm

One important thing to note.

In Haskell the 'if' is an expression which returns a value and that has to be considered when you use it in your programs.

SIM-2015410 - September 3, 2015, 10:08 pm

(Sep 3, 2015 10:01 PM)g4143 Wrote: [\[CITE\]](#) One important thing to note.

In Haskell the 'if' is an expression which returns a value and that has to be considered when you use it in your programs.

You are correct, thanks for the clarification :)

Tyranus. - September 3, 2015, 10:10 pm

Very HQ thread, i'm glad I stumbled upon this because i'm actually learning Haskell right now. Bookmarking for later use.

g4143 - September 3, 2015, 10:16 pm

(Sep 3, 2015 10:08 PM)?eshCollider Wrote: [\[CITE\]](#) You are correct, thanks for the clarification :)
How does Haskell handle the 'if' problem is this pseudo code?

```
Code let ans = if false then 5
```

?iven - September 3, 2015, 10:20 pm

Is Nex the number one student?

SIM-2015410 - September 3, 2015, 10:29 pm

(Sep 3, 2015 10:20 PM)?iven Wrote: [\[CITE\]](#) Is Nex the number one student?

I don't believe Nex intends on learning Haskell sorry. Hope that answers your question.

(Sep 3, 2015 10:16 PM)g4143 Wrote: [\[CITE\]](#) How does Haskell handle the 'if' problem is this pseudo code?

```
Code let ans = if false then 5
```

You have to have an else statement, there is only if-then-else in haskell, for the reason you have above. if is an expression which has to be converted to a value

g4143 - September 3, 2015, 10:40 pm

(Sep 3, 2015 10:29 PM)?eshCollider Wrote: [\[CITE\]](#) I don't believe Nex intends on learning Haskell sorry. Hope that answers your question.

You have to have an else statement, there is only if-then-else in haskell

OK. I come from a bit of OCaml where they allow an 'if - then' but only if the 'then' branch returns a unit type because the implied else will return a unit type.

These are the little things which turn off the traditional programmer from a more functional environment. Its always the simple things...

SIM-2015410 - September 3, 2015, 10:46 pm

(Sep 3, 2015 10:40 PM)g4143 Wrote: [\[CITE\]](#) OK. I come from a bit of OCaml where they allow an 'if - then' but only if the 'then' branch returns a unit type because the implied else will return a unit type.

These are the little things which turn off the traditional programmer from a more functional environment. Its always the simple things...

There are some useful functions in Control.Monad package which you might want to take a look at :)

g4143 - September 3, 2015, 10:53 pm

(Sep 3, 2015 10:46 PM)?eshCollider Wrote: [\[CITE\]](#) There are some useful functions in Control.Monad package which you might want to take a look at :)

No thanks. Haskell is too over the top for me. I'll stick with my brand of hipster languages - OCaml and Clojure.

SIM-2015410 - September 3, 2015, 10:59 pm

(Sep 3, 2015 10:53 PM)g4143 Wrote: [\[CITE\]](#) No thanks. Haskell is too over the top for me. I'll stick with my brand of hipster languages - OCaml and Clojure.

At least you are well rounded in different paradigms, that is all I ask ?

g4143 - September 4, 2015, 12:57 am

(Sep 3, 2015 10:59 PM)?eshCollider Wrote: [\[CITE\]](#) At least you are well rounded in different paradigms, that is all I ask ?

I think everyone should expose themselves to a Lisp or ML style language. They'll end up a better programmer for it in the end.

scam. - September 4, 2015, 6:56 am

(Sep 4, 2015 12:57 AM)g4143 Wrote: [\[CITE\]](#) I think everyone should expose themselves to a Lisp or ML style language. They'll end up a better programmer for it in the end.

I am going to have to learn lisp because Richard Stallman says it is good. ?

g4143 - September 4, 2015, 7:33 am

(Sep 4, 2015 06:56 AM)Mael Wrote: [\[CITE\]](#) I am going to have to learn lisp because Richard Stallman says it is good. ?

I would try Clojure then.

Some things you'll need to make your Clojure experience nicer.

1. Go here and use this and make your Clojure experience great -> <http://leiningen.org/>
2. Go here and read -> <https://aphyr.com/tags/Clojure-from-the-ground-up>
3. Or go here and read -> <http://www.braveclojure.com/>
4. Light Table IDE -> <http://lighttable.com/>

oramck - September 7, 2015, 8:51 am

Great thread! tks for sharing!!