TUTORIAL 1

August 16, 2021 5:23 PM

Steps for Downloading Haskell

So there's no straightforward way from doing this within China because china blocks vpn's ... however almost everyone living in china has at some point figured out how to get access to one (from a friend or when they travelled or whatever) and already has one they just need to be told to use it

[4:58 p.m.] C.B. Dalves
The Haskell stack documentation actually has instructions of how to configure stack to work without a vpn here https://docs.haskellstack.org/en/stable/install_and_upgrade/#china-based-users but most students will already have a vp
<https://teams.microsoft.com/l/message/19:tvKQ_HT6cBg-Gd5iQHXDQwcnRVNNa2LWeBUaaNa9FuU1@thread.tacv2/1631307466616?tenantId=44376307-b429-42ad-8c25-28cd496f4772&amp;groupId=94deb803-2b03-4829-89fe-0ebdc4ea218e&amp;parentMessageId=1631295760269&amp;teamName=COMPSCI 1JC3 Teaching Staff&amp;channelName=General&amp;createdTime=1631307466616>

** WINDOWS
1. VISIT https://docs.haskellstack.org/en/stable/README/
2. Click Windows 64-bit installer
3. Make note of install location
4. Make sure "add to path" is checked off
5. On windows open up 'Powershell'
6. Type stack ghci (may take some time)

** MacBook
1. VISIT https://docs.haskellstack.org/en/stable/README/
2. Copy command: curl -sSL https://get.haskellstack.org/ | sh
3. Open terminal
4. First run command: xcode-select --install
5. Paste the copied command from step 2
6. Make note where stack was installed in case need to add to path later on.
7. Should arrive to haskell interpretor: prelude.
8. :quit to exit prelude

<u>Install Visual Studio Code</u> - should be self explanatory, make note of path install location

For mac might want to right-click open, if not letting you open.

<u>Install Haskell Extension:</u>

1. Haskell - first one that pops up.
2. Haskelly

<u>Terminal command lines:</u>

on

Cd - "change directory"
Ls - "list"
:re
:quit
Stack ghci src/Exercise01.hs
Stack run (automatically does stack build)
Stack build
Stack clean
Stack new
Pwd - to see path

:t to check an expressions type

Create new project:

Open terminal and cd into a folder you would want your project to be created in.

Stack new --resolver=lts-16.31 'projectname'

Cd into test projectname

You will see a bunch of auto generated files

Resolver: chooses what version of the compiler you are using.

Only certain versions of ghc are supported by the Haskell features.

[LTS 16.31 for ghc-8.8.4] ghc resolver specifies libraries and packages

https://marketplace.visualstudio.com/items?itemName=haskell.haskell to see supported ghc

Open up whole project folder

Running Module

Lib.hs automatically gets generated by stack
Open terminal
Make sure you are in root project
  stack ghci src/Lib.hs

Should get prompt specifying module name - now you are in interpretor in command line. Can execu
the functions you've written

**Make sure you save before you run ghci! If it shows white dot means not saved!

Must reload it back to ghci! :re , save and reload -- will show changes you've made

-- to add comments
{- -} to add block comments

te

If downloaded Haskell extension properly, tells you the error.

<u>Exercise 01</u>

1. Visual Studio Code , open Folder
2. Click on src/ Exercise01.hs
3. stack ghci src/Exercises01.hs

<u>Some other information:</u>

Interpretor - directly runs code, translates just one statement at a time to machine code
Compiler - turns entire source code into machine language (at once) that can be read by the computer

Tutorial 1 – Tips on Haskell
http://learnyouahaskell.com/types-and-typeclasses
Static type system, Type of every expression is known at compile time.
Haskell has type inference -> Haskell can infer the type on it's own

One of the things that makes Haskell unique is its strong, static type system.
One important benefit of static typing is that many bugs are found at compile time, rather than run time.
With a dynamically typed language, you might ship code that seems to work perfectly well. But then in production a user stumbles on an edge case where, for instance, you forget to cast something as a number. And your program crashes.

With static typing, your code will simply not compile in the first place. This also means that errors are more likely to be found where they first occur.

An object with the wrong type will raise a compile issue at its source, rather than further down the call chain where it is actually used.

However development can be slower.

Compile time is the period when the programming code (such as C#, Java, C, Python) is converted to the machine code (i.e. binary code). Runtime is the period of time when a program is running and generally occurs after compile time.

Haskell has type inference -> Haskell can infer the type on it's own

```
ghci> :t 'a'
'a' :: Char
```
:: is read as "has type of"

<u>Constructor:</u>

Constructor can mean:

Type constructor
Data constructor (or value constructor)

In a data declaration, a type constructor is the thing on the left hand side of the equals sign.
The data constructor(s) are the things on the right hand side of the equals sign.
You use type constructors where a type is expected, and you use data constructors where a value is expected.

Data constructors

```
data Colour = Red | Green | Blue
```

Here, we have three data constructors.
Colour is a type, and Green is a constructor that contains a value of type Colour.

```
data Colour = RGB Int Int Int
```

RGB is not a value – it's a function taking three Ints and returning a value!

```
RGB :: Int -> Int -> Int -> Colour
```

RGB is a data constructor that is a function taking some values as its arguments
      uses those to construct new value

```
Prelude> RGB 12 92 27
#0c5c1b. <------------------- Color Value!!!
```

We have constructed a value of type Colour by applying the data constructor.
A data constructor either contains a value like a variable would, or takes other values as its argument and creates a new value.

```
data SBTree = Leaf String
            | Branch String SBTree SBTree
```

type SBTree that contains two data constructors.
there are two functions (namely Leaf and Branch) that will construct values of the SBTree type.

Type constructors

Both SBTree and BBTree are type constructors.

```
data BTree a = Leaf a
             | Branch a (BTree a) (BTree a)
```

type variable a as a parameter to the type constructor.
In this declaration, BTree has become a function. It takes a type as its argument and it returns a new type.

      concrete type (examples include Int, [Char] and Maybe Bool) which is a type that can be assigned to a value

A value can never be of type "list", because it needs to be a "list of something".

If we pass in, say, Bool as an argument to BTree, it returns the type BTree Bool,

If you want to, you can view BTree as a function with the kind
BTree :: * -> * -- these are called "Kinds"

BTree is from a concrete type -> concrete type. Concrete type to a concrete type

Wrapping up

> A data constructor is a "function" that takes 0 or more values and gives you back a new value.
> A type constructor is a "function" that takes 0 or more types and gives you back a new type.

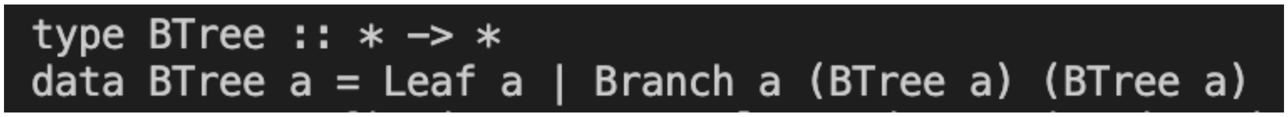
```
data Maybe a = Nothing
             | Just a
```

Here, Maybe is a type constructor that returns a concrete type.
Just is a data constructor that returns a value.
Nothing is a data constructor that contains a value.

```
Maybe :: * -> *
```

In other words, Maybe takes a concrete type and returns a concrete type.
Once again! The difference between a concrete type and a type constructor function.

In prelude: :info BTree



```
type BTree :: * -> *
data BTree a = Leaf a | Branch a (BTree a) (BTree a)
```

---TYPE VARIABLES

Head a
A is a type variable , takes a type

TYPECLASSES

Interface that defines some behaviour.

MULTIPLE CRADLE ERRORS

MULTIPLE CRADLE ERRORS

multiple cradles, is fixed by specifying the different components of your project in a hie.yaml file. There's a tool for automatically generating the file that can be installed with stack, i.e.

```
 $ stack install implicit-hie
 $ gen-hie --stack > hie.yaml # from the project root
```

Tutorial 1 – Tips on Haskell

A monad is an algebraic structure in category theory, and in Haskell it is **used to describe computations as sequences of steps**, and to handle side effects such as state and IO. Monads are abstract, and they have many useful concrete instances. Monads provide a way to structure a program.

For the specific case of the IO monad, as in your getArgs example, a rough but useful intuition can be made as follows:

- x <- action *runs* the IO action, gets its result, and binds it to x
- let x = action defines x to be equivalent to action, but does not run anything. Later on, you can use y <- x meaning y <- action.

$$S = \{2 \cdot x \mid x \in \mathbb{N},\ x \leq 10\}$$

Prelude is **a module that contains a small set of standard definitions** and is included automatically into all Haskell modules