Basic Haskell Cheatsheet

Syntax, keywords, and core types

Susan Potter

February 18, 2019

Contents

Basic Syntax

Comments

- single line comments

```
-- Write your single line comment in Haskell source code here
```

- multi-line comments

```
{-
  Here is a multi-line comment in Haskell source code
  where I can continue to write on the next line without
  prefixing each line of the comment block with --.
-}
```

Documenting

```haskell
  -- This will not be rendered in Haddock generated document artifacts
  -- because I did not start the comment with ‘-- |‘

  -- | Represents a stage of development
  data Stage
    = Dev     -- ^ in development phase
    | Test    -- ^ in testing phase
    | Staging -- ^ in staging/qa phase
    | Prod    -- ^ released to production
    deriving (Eq, Show, Enum)

  -- | Produces a @Maybe Region@ for the @Stage@
  toRegion
    :: Stage        -- ^ given @Stage@
    -> Maybe Region -- ^ result
  toRegion Dev     = Nothing
  toRegion Test    = Nothing
  toRegion Staging = Just UsEast2
  toRegion Prod    = Just UsWest2

-- | ** This is a title
-- @code block here@ or we can block it out with a > prefix on a new comment line:
-- > literal code block here
-- We can also do REPL examples:
-- >>> 5 + 9
-- 14
-- ** Formatting
-- And using @doctest@ we can turn them into executable unit test cases.
-- I can /emphasis like so/ and __bold__.
-- I can even specify properties about my code like this:
-- prop> identity a == a
-- I can even make raw links <https://github.com>
-- Or [a named link](https://www.dailykos.com)
-- Or even embed an image with alt text ![pug puppy](https://i.imgur.com/MqMUU6Z.jpg)
-- I can even make a list of my favorite dried peppers:
```

```
-- - ancho
-- - chipotle
-- - chiles de arbol
identity :: a -> a
identity a = a
```

- Haddock is a tool in the Haskell ecosystem that can produce documentation artifacts from inline source documentation.
- `cabal new-haddock` will generate documentation from your configured source code.

Reserved words

- `case`, `of`
- `class`, `instance` (for type classes/interfaces and their implementations)
- `data`
- `deriving`
- `do`
- `if`, `then`, `else` (e.g. `if true then "woot" else error "END OF THE WORLD!"`)
- `import`, `qualified`
- `infix`, `infixl`, `infixr`
- `let`, `in` (e.g. `let a = 43 in a*a`)
- `module`
- `newtype`
- `type`
- `where`

Strings

- The builtin (to the base/Prelude) `String` type is problematic in Haskell for high- throughput processing of strings.
- It is represented as a singly linked list of characters for historic reasons.
- Thankfully the ecosystem solved this problem with `ByteString` (available via the `bytestring` package) for lower-level communications code and `Text` (available via `text` package) for higher-level encoding centric logic. Both of these have strict and lazy representations which are good for different use cases. Most of the time you will never need to know which you are using until - abruptly - you do. By this point you should have a lot more knowledge and understanding in your arsenal to tackle it handily.
- A `Char` (from module `Data.Char` in the base package) is one character.
- Remember a `String` is a list of `Char` elements, naively: `type String = [Char]`.

Numbers

- `1` - Integer or floating point number. In the REPL it is automatically inferred to `Integer`
- `1.0, 1e10` - Floating point value.
- `0o1, 0O1` - Octal value.
- `0x1, 0X1` - Hexadecimal value.
- `-1` - negative number. Cannot have a space between the minus sign and first digit.

Enumerating values

- `[1..10]` - range of integers from 1 through 10 inclusive.
- `[100..]` - infinite list of integers starting at 100. Don't ask why Haskell represents infinite codata as "data". :(
- `[10, 9 ..  1]` - list from 10 through 1 inclusive.
- `[10 ..  1]` - empty list!!!! Just to keep you on your toes. :)
- `[1, 3, ..  9]` - all odd numbers from 1 through 9 inclusive.
- `['a' ..  'm']` - all lower case characters from `'a'` to `'m'`.

It isn't just numbers we can do this with:

```
module Effpee.Suit
  ( Suit (..)
  ) where

import GHC.Enum
import GHC.Show

-- ** Card Suits
-- Represents suits in a deck of cards
-- >>> [Diamonds .. Hearts]
-- [Diamonds,Clubs,Hearts]
-- it :: [Suit]
data Suit
  = Diamonds
  | Clubs
  | Hearts
  | Spades
  deriving (Enum, Show)
```

Prefix and infix operator notation

In arithmetic we often use infix operator notation which looks something like this:

```
addExample  = 5 + 7
multExample = 6 * 9
divExample  = 6 / 2
subtExample = 9 - 5
```

In each of the above examples we use the binary operators (+, *, /, -) in between its two arguments. This is called infix operator notation.

We can use it like we use most functions in Haskell in prefix notation too, like so:

```
addExample2  = (+) 5 7
multExample2 = (*) 6 9
divExample2  = (/) 6 2
subtExample2 = (-) 9 5
```

These are exactly equivalent forms of the same expression in differen notation. In some cases infix notation scans better and thus the flexibility of us to be able to use infix style in Haskell allows us to build embedded domain specific languages more easily.

In the next section on 'Builtin Lists' you will see how infix data constructors are used.

PS data constructors are just a special case of a function.

```
effpee> :l src/Effpee/ADT.hs
[1 of 1] Compiling Effpee.ADT       ( src/Effpee/ADT.hs, interpreted )
Ok, one module loaded.
effpee> :info Many
data Many a = Empty | a :. (Many a)
        -- Defined at src/Effpee/ADT.hs:146:1
instance [safe] Eq a => Eq (Many a)
```

```
  -- Defined at src/Effpee/ADT.hs:149:13
effpee> :type Empty
Empty :: Many a
effpee> :type (:.)
(:.) :: a -> Many a -> Many a
```

Builtin Lists

- Note: We will be building our own list data types (called `Many` and `ManyReversed`) for our learning exercises.
- `[]` - empty list
- `[1, 2, 3]` - list of 1, 2, and 3 in that order
- `1 : 2 : 3 : []` - equivalent to above and actually how we need to think about lists as we build recursive functions on lists.
- Naively the data type definition looks something like this following without optimizations added:

```
data [] a = [] | a : [a]
-- setting this to 5 means that this operator is right-associative and more "sticky"
-- on the right most occurrence in a string of these, e.g. =1 : 2 : 3 : 4 : []=.
infixr 5 :

-- Written out with textual type name and data constructors:
data List a = EmptyList | Cons a (List a)
```

Functions & pattern matching

```
data Shape = Square Int | Rectangle Int Int | Circle Int

perimeter :: Shape -> Float
-- ^^^ a type annotation for the function’s type.
-- vvv below is how we pattern match on the LHS of a function definition
perimeter (Square n)      = 4*n
perimeter (Rectangle m n) = 2*(n+m)
perimeter (Circle r)      = 2*pi*r

area :: Shape -> Float
-- ^^^ another type annotation for the function, usually can be inferred
area (Square n)       = n*n
area (Rectangle m n)   = m*n
area (Circle r)       = pi*r*r
```

Guards

```haskell
isEven :: Int -> Bool
isEven 0 = True
isEven n | n ‘div‘ 2 == 0 = True
         | otherwise      = False

-- However, in this case it would be better to do:
isEven2 :: Int -> Bool
isEven2 n = n ‘rem‘ 2 == 0
```

Currying

```haskell
defaulter :: a -> Maybe a -> a
defaulter a Nothing  = a
defaulter _ (Just a) = a

-- Note here we can partially apply the @a@ argument of 30 and automatically get back
-- rest of the function which is of type @Maybe a -> a@. Haskell automatically do
-- this "currying" for you. The user partially applies arguments to a function
-- and Haskell automatically "curries" that argument to produce the new function
-- for you. In Scala, life would be miserable here unless you designed your API
-- to be partially applied, but Scala does zero currying automatically so it will
-- involve a lot more boilerplate. Have I mentioned lately how glad I am to not
-- be reading or writing Scala now?
usingDefault :: Maybe a -> a
usingDefault = defaulter 30
```

Anonymous functions & higher-order functions

Sometimes we need to pass a function as an argument or produce it as the result from a function. We can do that by passing an already named function, but sometime we just want to pass an anonymous function. This is how we might do that.

```haskell
-- >>> :type filter
-- filter :: (a -> Bool) -> [a] -> [a]
-- Notice the parens around the first argument signifying it is a function and not
-- a value of an @a@ followed by a value of a @Bool@.
-- >>> filterOnlyOne [1, 2, 3, 4 - 1]
-- [1,1]
filterOnlyOne :: [Int] -> [Int]
filterOnlyOne xs = filter (\x -> x == 1) xs

-- The parens around a returned function are less important due to auto-currying
-- that Haskell does, but as documentation it can help you reset your mind as to
-- what the purpose of your plight is when defining functions.
filterOnN :: Int -> ([Int] -> [Int])
filterOnN n = filter (\x -> x == n)
```

Algebraic Data Types (ADT)

Data types in Haskell have algebraic properties which we will introduce as the course proceeds (for example, ADTs in Haskell have derivatives that can be computed which provide a view of a data structure at a certain point much like derivatives of a function in mathematics gives us the gradient at a certain point).

Note: Whenever I refer to ADTs in this course I am referring to algebraic data types and not Abstract Data Types though there might be overlap sometimes.

These are the four basic categories (*I am hand waving now*) of ADTs:

- products (sometimes called records in other languages)
- coproducts (sometimes called sum types, or enum types, or union types)
- recursive types (which can also be products or coproducts)
- some hybrid of the above, e.g. coproducts or products possibly with recursive structure.

Let's go to `src/Effpee/ADT.hs` now and look at already defined ADTs and define the ones marked with TODO.

Deriving (e.g. `deriving (Eq, Enum)`)

Ignore these statements in the ADTs for now. We will return to them because this is a powerful facility (as seen in the `Enum` example with card suits above) in Haskell and deserves more than just a moment's coverage.