Basic Haskell Cheatsheet

Syntax, keywords, and core types

Susan Potter

March 6, 2019

Basic Syntax

Comments

- single line comments

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

- multi-line comments

```haskell
{-
  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)
```

Documenting even more

```haskell
-- | 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
```

More haddock inline markup examples

```
-- | ** 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
plus ::Int -> Int -> Int
plus a b = a + b
```

And a little bit more haddock with lists and links

```haskell
-- ** Heading here
-- doctest can turn them into runnable unit test cases.
-- I can /emphasis like so/ and __bold__.
-- I can 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.twitter.com)
-- ![An embedded image](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
```

Inline source documentation

- 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`, `newtype`, `type`
- `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`
- `where`

Strings

- The builtin (to the base/Prelude) `String` type is problematic in Haskell for high- throughput processing of strings.
- 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]`.
- Use `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.

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'`.

Custom enumerations

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

```
module Effpee.Suit (Suit (..)) where
import GHC.Enum
import GHC.Show

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

Prefix 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.

Infix operator notation

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

```haskell
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.

Data constructors are special cases of functions

```
effpee> :l src/Effpee/ADT.hs
[1 of 1] Compiling Effpee.ADT       ( src/Effpee/ADT.hs
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:

Builtin lists: code examples

```
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

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

perimeter :: Shape -> Float
-- ^^^ a type annotation for the function’s type.
perimeter (Square n)      = 4*n
perimeter (Rectangle m n) = 2*(n+m)
perimeter (Circle r)      = 2*pi*r

area :: Shape -> Float -- typically 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
```

You might see this syntax but probably best not to use it.

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".
usingDefault :: Maybe a -> a
usingDefault = defaulter 30
```

Anonymous 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
```

Higher-order functions (HOFs)

```haskell
-- 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)

-- We can accept a function as an argument.
-- Only one thing we can do with a given function:
-- apply it.
filter :: (a -> Bool) -> [a] -> [a]
filter = error "todo: this is an exercise later."
```

Algebraic Data Types (ADT)

Algebraic Data Types (ADTs) have alegraic properties (discussed later). There are four basic primitives to use when definint ADTs:

- products (sometimes called records or structs in other languages)
- coproducts (sometimes called sum types, or enum types, or union types)
- recursive reference

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

ADT: Product constructor syntax

```
ConstructorName Field1Type Field2Type Field3Type
```

To add functions `attrName1`, `attrName2`, and `attrName3` to easily pull out specific fields from the record:

```
ConstructorName { attrName1 :: Type1
                , attrName2 :: Type2
                , attrName3 :: Type3 }
```

ADT: Coproduct construtor syntax |

```
data Bool = True | False

data USCoin
  = Cent
  | Nickel
  | Dime
  | Quarter
  | Dollar
  | TwoDollar
```

ADT: Recursive and mixing products and coproduct forms together

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

-- add accessors
data BinTree’ a
  = Leaf’ a
  | Branch’ { leftBranch :: BinTree’ a
            , rightBranch :: BinTree’ a }
```

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.