More Haskell — Miscellaneous topics

- Simple I/O, do blocks, actions
- Modules
- `data` vs `type` declaration
- Instances of class types
- User-defined operators
- Sections
- The `$` operator

Next time: Regular Expressions (Not Haskell!)

Simple I/O

- I/O doesn't preserve referential transparency.
- If `getLine` returns the string read in, then `getLine ≠ getLine`
- Need to sequester I/O off to the side so that the referentially transparent part remains state-free
  - You can run code that does I/O, but it can't return results to the rest of your program.
  - Your I/O code can access the rest of your program, just not the other way

How to model printing/reading

- `putStrLn "some string"` prints and returns a value of type `IO ()`
  - By the way, `putStrLn "string"` is like `putStr ("string" ++ "\n")`
- Printing modifies the printer - captured by the `IO ()` value
  - Also has side effect of making printer print.
- There are routines that take `IO t` data and create other `IO t'` data
  - But no way to take `IO t` and return (e.g.) an integer
  - (Unless you ignore the `IO t` data.)
- Reading via `getLine` produces a value of type `IO String`
  - There's a string hidden in there, but you can only access it certain ways
  - To pull string out involves fairly complicated code that uses operator `>>=` ("bind") and lambda functions
  - Haskell supports simpler syntactic sugar that maps to that code.
do block

- Use `variable <- IO String` expression to set `var = the string`
- A do block consists of a sequence of "actions" reading, printing, setting `x <- getline` etc.
- Haskell translates do block into the more complicated format

```
Prelude> :{
    putStrLn "data? 
    str <- getline
    putStrLn str

Prelude> :{
    putStrLn (show (f 5))
```

- If you want a plain calculation like `y = str ++ str`, you use a `let`.

```
Prelude> :{
    let y = str ++ str
    putStrLn y

Prelude> :{
    let f x = x + 3
    putStrLn (show (f 5))
```

- You can define a function using `let`:

```
Prelude> :{
    do
    let f x = x + 3
    putStrLn (show (f 5))
```

- Note: Last part of a do block needs to be an action (not a `let` or an expression)

- A `let` or expression would allow you to define a do block that returns a non-IO value, which would break referential transparency.

```
Prelude> :{
    do s <- getline
    s ++ " " ++ s -- an expression!
```

<interactive>:115:4: error:
• Couldn't match type `[]` with `IO`
  Expected type: IO Char
  Actual type: [Char]
• In a stmt of a 'do' block: s ++ " " ++ s
  In the expression:
    do s <- getLine
       s ++ " " ++ s
  In an equation for 'it':
    it
      = do s <- getLine
          s ++ " " ++ s

• You can use if-else with actions; if a branch has > 1 action you need an inner do block for it

  Prelude> :{
  Prelude| do s <- getLine
  Prelude|    if s == ""
  Prelude|    then
  Prelude|       putStrLn "empty"
  Prelude|    else do
  Prelude|       putStrLn "String wasn't empty, it was "
  Prelude|       putStrLn s
  Prelude|  :}
  hi
  String wasn't empty, it was hi
  Prelude>

• You can use read to turn a string into a number

  Prelude> :{
  Prelude| do s <- getLine
  Prelude|    let n = read s :: Int
  Prelude|    let dbl = n + n
  Prelude|    putStrLn ("Double " ++ show n ++ " is " ++ show dbl)
  Prelude|  :}
  1234
  Double 1234 is 2468
  Prelude>

Modules
• A module is a named collection of various definitions
  • Prelude is the library module loaded by default
  • In code get access to module using import modulename
import Data.Char

- ghci commands to import and unimport :m +Data.Char, :m -Data.Char

  Prelude> :m +Data.Char
  Prelude Data.Char>

  - Note after importing Data.Char, it appears in the prompt as a loaded module.

- ghci command :browse Data.Char

  - Prints out all the names associated with the module.

    Prelude Data.Char> :browse Data.Char
    Data.Char.digitToInt :: Char -> Int
    Data.Char.isLetter :: Char -> Bool
    toUpper :: Char -> Char
    ...
    GHC.Unicode.toUpper :: Char -> Char

    Prelude Data.Char> map toUpper "abcDE"

    "ABCDE"

  - If a module isn't loaded, you can use a qualified name

    Prelude Data.Char> :m -Data.Char
    Prelude> map Data.Char.toUpper "afsfs97423wruefsd"
    "AFSFS97423WRUEFSD"
    Prelude>

**data declaration vs type declaration**

- We've seen data declarations - they get you enumerations, algebraic types

- A type declaration just gives you another name for a type

  - Structural equivalence, not name equivalence

    Prelude> :{
    Prelude| type P1 = (Int, Int)
    Prelude| type P2 = (Int, Int)
    Prelude| p1 = (3, 5) :: P1 -- P1 just stands for (Int, Int)
    Prelude| p2 = (3, 5) :: P2
    Prelude| }
    Prelude> p1 == p2
    True

  - Datatypes use name equivalence -- even if two datatypes hold data the same way, they're of different types.

    data D1 = D1con Int
    data D2 = D2con Int
d1 = D1con 17

d2 = D2con 17

d1 == d2 -- type error

• (For even better example, I tried to define D1 and D2 to both use Dcon as a constructor, but that causes an error)

    data D1 = Dcon Int
    d1 = Dcon 17
    data D2 = Dcon Int -- error- renaming constructor Dcon
    d2 = Dcon 17 -- can't do
    d1 == d2     -- can't do

• (Let's do this: Define D1 twice. Note below, Haskell uses a different name for the old version of D1, and data
from the two versions isn't comparable, even though they are built the same way.)

    Prelude> :{
    Prelude| data D1 = Dcon Int
    Prelude| d1 = Dcon 17 -- uses first version of Dcon
    Prelude| :}
    Prelude> :{
    Prelude| data D1 = Dcon Int
    Prelude| d2 = Dcon 17 -- uses first version of Dcon
    Prelude| :}
    Prelude> :t d1
d1 :: Ghci19.D1
    Prelude> :t d2
d2 :: D1
    Prelude> d1 == d2 -- error

• Because type declarations are just synonyms, this example works on type D1 = Int:

    Prelude> :{
    Prelude| type D1 = Int
    Prelude| d1 = 17 :: D1
    Prelude| :}
    Prelude> :{
    Prelude| type D1 = Int
    Prelude| d2 = 17 :: D1
    Prelude| :}
    Prelude> :t d1
d1 :: Ghci47.D1
    Prelude> :t d2
d2 :: D1
    Prelude> d1 == d2 -- is NOT a type error
    True
    Prelude>
Instances of class types

- **A type class instance** is a type that belongs to the class
  - E.g., `Eq Int, Eq Char`, etc. But not `Eq (Int -> Int)`
- You can create your own type and make it an instance
  - The definitions of various functions or operators on the type go **inside the instance declaration**, not with the datatype definition.
    ```haskell
data Fraction = Frac Int Int deriving (Eq, Show, Read)
-- numerator over denominator

instance Eq Fraction where  -- the code for == goes here
    (Frac n1 d1) == (Frac n2 d2)
    = n1 * d2 == d1 * n2

instance Num Fraction where
    (Frac n1 d1) * (Frac n2 d2)
    = ... omitted ...
```

- You can declare the functions with the data declaration by defining synonyms for the operations and using those in the instance declaration. E.g.,
  ```haskell
data Fraction = Frac Int Int deriving (Eq, Show, Read)
eqfrac (Frac n1 d1) (Frac n2 d2) = n1 * d2 == d1 * n2

instance Eq Fraction where
    (==) = eqfrac
```

User-defined operators

- Use sequence of special symbols (certain chars not useable)
- By default, user-defined operators are infix and left associative (you can change this; not shown)
  ```haskell
  (//) :: Float -> Float -> Float
  x // y = if y == 0 then x else x / y
  -- If you prefer prefix notation, you can use that instead:
  (//) x y = if y == 0 ....
  ```
- **Danger**: Don't give existing operators definitions outside of their typeclass instance declaration
  ```haskell
data Fraction = ....
(Frac n1 d1) + (Frac n2 d2) = ....
  ```
- When `+` is defined this way, outside the instance declaration for `Num`, this `+` overrides the old definition of `+`. E.g., 2+2 fails. (Very socially unacceptable !)
- Again, you can define a synonym function for `(+)` and refer to it in the instance declaration
  ```haskell
data Fraction = ....
fracadd (Frac n1 d1) (Frac n2 d2) = ...
```
instance Num Fraction where
   (+) == fracadd -- as in earlier example
   -- definitions of other operators omitted

Sections

• A **section** is a partially-applied operator: 
  \[(expr \ optr) = \ y \rightarrow expr \ optr \ y\]

  - It works as a unary function.

  ```haskell
  Prelude> f = (1 +) -- adds one to something
  Prelude> :t f
  f :: Num a => a -> a
  Prelude> f 2
  3
  ```

  - Can be useful for mapping across lists:

    ```haskell
    Prelude> map (1 -) [2..10]
    [-1,-2,-3,-4,-5,-6,-7,-8,-9]
    ```

  - There's also \[(optr \ expr) = \ y \rightarrow y \ optr \ expr\], where you've specified the right-hand operand.

    ```haskell
    Prelude> map (/ 2) [2..10]
    [1.0,1.5,2.0,2.5,3.0,3.5,4.0,4.5,5.0]
    ```

  - You can use sections on a named function via backquotes

    ```haskell
    Prelude> f x y = 2 * x + y
    Prelude> (5 `f`) 3 == f 5 3 -- section with first argument
    True
    Prelude> (`f` 5) 3 == f 3 5 -- section with second argument
    True
    ```

• The \(-\) operator is different because it has unary and binary interpretations: \((-1)\) is just \(-1\), not a section.

  ```haskell
  Prelude> :t (+ 3)
  (+ 3) :: Num a => a -> a
  Prelude> :t (- 3)
  (- 3) :: Num a => a
  ```

• You can use \((expr -)\) as a section, however.

  ```haskell
  Prelude> :t (1 -)
  (1 -) :: Num a => a -> a
  Prelude> map (1 -) [2..10]
  [-1,-2,-3,-4,-5,-6,-7,-8,-9]
  ```

• You can parenthesize \(-\) to use it as a prefix function

  ```haskell
  Prelude> map (\(-\) 1) [2..10]
  [-1,-2,-3,-4,-5,-6,-7,-8,-9]
  ```

• You can also define a synonym for binary \(-\) and use that
Prelude> sub = (-)
Prelude> map (`sub` 1) [2..10]
[1,2,3,4,5,6,7,8,9]
Prelude> map (1 `sub`) [2..10] -- works like (1 -)
[-1,-2,-3,-4,-5,-6,-7,-8,-9]

The Function Application Operator $$
•$$ The $ operator is defined as \( f x = f x \)
• Why bother? $ has very low precedence, normal function app has very high precedence
  • Using $ can let you omit some parens when you have complicated function expressions
  • You can write \((expr_1) (expr_2)\) as \(expr_1 \ expr_2\) (without the parentheses)
  • $ can also be useful as a section

Prelude> map ($ 256) [sqrt, sqrt . sqrt, sqrt.sqrt.sqrt]
[16.0,4.0,2.0]

Okay, that's it for Haskell for now — Next time: Regular Expressions