
    Declarative Programming

Functional Programming
- Based on <equational reasoning>

Basically, *if two expresions have equal values*, then *one can be replaced* *by the other*.

You use <equational reasoning> to <rewrite a complex expression> to be <simpler>
Until it is "as simple as possible".

e.g. reducing x = 2, y = 4, x + (3 * y) to 14

    Basics of Haskell

!Lists

- [] means empty list[] // lol rip notes syntax, will use [ ] for []
- x:xs means a non-empty list whose head (first element) is x and xs is tail (everything else in the list)

The notation ["a", "b"] is syntactic sugar for "a":"b":[ ]

How this works is basicaly it'll evaluate the following:
- Put "a" in front of "b"
- Put "a","b" into an empty list
- We now have an array of ["a", "b"]

Some operators to know:

: (double colon) - <appends an element> of a list <to a list>
    - Note that this only works if the element is of the same type as whatever's in the list.
    - This also only works in one direction (as in you can only add to the top)

++ - appends list to a list
    - Does not work if you try to add single elements to the list
    - To get around this you can use [ ] around your element like this:
      fish ++ [jellyfish] // assuming jellyfish is an element of type fish

!Functions

A function definition consists of <equations>, each of which <establishes an
<equality between the left and right hand sides> of the equal sign.

Example:
[haskell]
```
-- Most equations look something like this
len []      = 0              -- Base case
len (x:xs)  = 1 + len xs     -- Recursion (there's no loops in haskell)
```
[end]

Each equation <expects input to conform to a given pattern>. The empty list [ ] and (x:xs) are two patterns.

The set of patterns *must be exhaustive*, as in it covers <all possible calls>.

It is <good programming style> to ensure that the <set of patterns is also >
<exclusive>, which means that <at one most pattern should apply> for any possible call.

If the set of patterns is both <exhaustive and exclusive>, then <exactly one> pattern will apply <for any possible call>

If you don't make your function exhaustive, enjoy your errors.
<You can get around this by using _, which signifies any case.>

Functions look like the following:
--> f fa1 fa2 fa3

Use brackets for precedence.
Operators are also functions, including the : thing, which is why we do (x:xs)
This means technically you can do
(+) 1 2 // equivalent 1 + 2

The names of functions and variables are sequences of letters/numbers/underscore *that must start with a lower case letter* (except if you're an operator).

The Offside Rule:
Indentation works like python!
- Further right of prev line = continuation
- Same level = new statement
- Further left = either of the above (like in guards)

Recursion
Done like above, with the whole length thingo.
Usually your structure should look like the following:

<base case> // usually empty list or 0, might have more if you have more args
<the next function call with a new list or modified arguments>

Let's try it out using the len function from above:
[haskell]
```
len [] = 0
len (x:xs) = 1 + len xs

step 0: len ["a", "b"] -- ("a":("b":[]))
step 1: 1 + len ["b"] -- ("b":[])
step 2: 1 + 1 + len []
step 3: 1 + 1 + 0
step 4: 1 + 1
step 5: 2
```
[end]

Expression Evaluation:
Haskell basically runs a loop that looks for function calls and goes from top down until all of the variables are replaced with their most simplest versions.

Due to <lambda calculus> we know for functional programming languages rewriting the order of the terms does not make a difference.

!Everything is immutable in Haskell

You can assign things with "let"

[haskell]

```
let memes = 420 in ... -- expression where dots are for scope
```

[end]

    Builtin Haskell Types

Int, Bool, Char, String, Float, Double, etc.

Haskell uses :: to say that one variable is a type of another.

x :: [Char] means x is of type char.

Functions also have types:
- A function type lists the types of all the arguments and the result, all separated by arrows.

So for example,

```
:t fst
fst :: (a, b) -> a
// This returns the first element of a tuple
```

Function Types

You should declare the type of each function before stating the function.
For example,

[haskell]

```
isEmpty :: [t] -> Bool
isEmpty [] = True
isEmpty _ = False
```

[end]

If the type declaration is wrong, then it won't compile.

Number Types:
Haskell has a bunch of numeric types, including int, integer, float and double.
So what does haskell say 3 is?

It's a 3 :: (Num t) => t

Here the notation (Num t) means type t is of class <Num>
<Num> is a numerical class.

The notation 3 :: (Num t) => t means:
"If t is a numeric type, then 3 is of that type"
<Nothing else is defined for this function> if it's not a numerical type so
<return an error.>

Basically all of the letters in a type declaration denote an argument.

For example:

```
:t (+)
(+) :: (Num a) => a -> a -> a
```

So all the arguments a will be the same type (either Float, Int, Double etc.) as they are all type 'a'.

If-then-else
Work the same way as they do in other languages

Guards
Kinda like switch/case statements. Just have cases depending on the value.

[haskell]
```haskell
-- if else
fish a =
    if a == 42 then "Yes" else "No"
-- You can also do multiline if else with:
    if a == 0
    then
        "Yes"
    else
        "No"

-- guards
fishGuards a
    | a == 0    = "Zero Fish"
    | a  > 0    = fish a
    | otherwise = "Jellyfish" -- this covers all remaining cases, like default
```
[end]

Parametric Polymorphism
You guys know the drill now. A function that can change depending what type the args are.

You have to use a function type definition though like this:

[haskell]
```haskell
len :: [t] -> Int
len []      = 0
len (_:xs)  = 1 + len xs
```
[end]

This can adapt to whatever the input type is, so long as it's in a list.

    Defining Our Own Types

We can make our own types in Haskell!
The simplest types are <enumerated types>

[haskell]

```haskell
data Gender = Female | Male -- can only be either one
data Role = Staff | Student

data Suit = Club | Diamond | Heart | Spade
```

```haskell
data Rank = R2 | R3 | R4 | R5 | R6 | R7 | R8
            | R9 | R10 | Jack | Queen | King | Ace
            deriving Show

-- This is just a structured thing (constructs a card with suit and rank inside)
data Card = Card Suit Rank

-- Discriminated union types
data JokerColor = Red | Black
data JCard =
    NormalCard Suit Rank |
    JokerCard JokerColor
```

[end]

These are also data constructor - given zero arguments they construct a value
// just like in oop

There's a class called Show. It converts data types to string so we can print!

We can <specify what classes our type is in>.

Some examples are:
- Show (converts to string)
- Eq (Allows things to be compared)
- Ord (Ordered)

In haskell you can have an <optional value>

[haskell]

```haskell
data Maybe t = Nothing | Just t
```

[end]

What this does is that, for any type t, a value of type maybe t is <either >
<Nothing, or Just x>, where x is a value of type t.

// skipping section 4 and 5 because I'm a bad student and I'm running out of
// time and these sections basically just cover imperitive languages compared
// to haskell
// *Will complete this after midsems probably
// someone save me

Let and Where Clauses:
- A let clause introduces a name for a value <to be used in the main function>
- A where clause has the <same meaning> but has the definition of the name <after the main expression>
- You can only use where clauses <at the top level of a function>, where you can use a let for any expression

You can use backquotes to make an <infix operator> like +, using this:
[haskell]

```haskell
fish = 3 `mod` 420
```

[haskell]

Higher Order Functions

First order values are data.
Second order values are functions whose arguments are first order values.

Third (and higher) order values are <spicy bois> and have functions with second order values as arguments.

So basically, *Higher Order functions use other functions as arguments*.

The best example of this is <filter>, which is a very neato function that filters out a list according to a supplied boolean function.

Here's filter in action:
[haskell]

```haskell
-- Write a simple second order function
fourTwenty :: a -> Bool
fourTwenty 420 = True
fourTwenty a   = False

-- Make an array to filter
let x = [1, 2, 3, 4, 5, 420, 70, 12, 9]

-- Now to use our third order function, filter
let spicymemes = filter fourTwenty x
-- This'll leave [420]
```

[end]

Note that <filter> is <polymorphic> and works with any array.

[haskell]

```haskell
filter :: (a -> Bool) -> [a] -> [a]
filter _ [] = []
filter f (x:xs) =
    if f x then x:fxs else fxs
    where fxs = filter f xs
```

[end]

Anonymous Functions:
Sometimes you want to <use filter with a function> that has <two or more args>, or if you want to test for equality, this is where <anonymous functions> come in

Works like this:
[haskell]

```haskell
filter (\x -> x `mod` 2 == 0) [1, 2, 3, 4, 5]
-- takes x (each element of the array) and compares it to the anon function)
-- should return [2, 4]
```

[end]

Another <very neato function> is <map>, where it applies a second order function to a array.

Map and filter are basically the reason why we can get around using only recursion in Haskell.

Composite Function
You can make a higher order function by combining some second order functions.

There's a built in operator '.' which <composes two functions>.

[haskell]
```haskell
-- Using our last example:

-- Write a simple second order function
fourTwenty :: a -> Bool
fourTwenty 420 = True
fourTwenty a   = False

-- Write a simple second order function
nope :: Int -> Int
nope 420 = 420
nope 7 = 420
nope a = 0
-- This means when filter is called with (fourTwenty . nope), all 7s and 420s
-- will be kept
-- (because when checking the composite function, all 7s will be evaluated
-- as 420 and everything else as 0)
```
[end]

Note that using the "." operator means it will <evaluate from right to left!>.
You can do it with multiple functions if you're cool.

    Higher Order Programming

Higher order programming is cool?

map and filter <operate on and transform> lists, but the <other class> of higher order functions are <reduction operations>.

Folds
// grab my foldy flaps

The popular ones are *folds*.
There are three main ones:
- foldl (fold left)
- foldr (fold right)
- balanced_fold (fold balanced)

Folding is how you get things like <sum>.
Basically it allows you to <reduce an array> to a single point.

Here's the code for <foldl>:
[haskell]
```haskell

foldl :: (v -> e -> v) -> v -> [e] -> v
-- funciton that applies to arg1 and arg2 (and returns arg1 type), arg1 and
-- array of arg2
foldl _ base [] = base                    -- base case
foldl f base (x:xs) =
```

```
    let newbase = f base x in                  -- apply fold to head
    foldl f newbase xs

-- Here's some examples of how some other functions use foldl

sum :: [Integer] -> Integer
sum = foldl (+) 0

product :: [Integer] -> Integer
product = foldl (*) 1

concat :: [[a]] -> [a]
concat = foldl (++) []

const :: a -> b -> a
const a b = a

length = foldr ((+) . const 1) 0

[end]
```

folds are real powerful. Google's second most important algorithm is effectively a fold (MapReduce).

List Comprehensions

Just make lists like they do in python. These are pretty good cause the conditions means you can do lots of things with them. (like writing out a program that normally takes 20+ lines into one)

```
[haskell]
-- some examples
-- makes a list of multiples of 420 from 1 to 100
fourTwnetyList = [x*420 | x <- 1..100]

-- and you can also iterate multiple things
pairs = [(a, b) | a <- [1, 2, 3], b <- [1, 2, 3]]
[end]
```

Working with HTML:

```
[haskell]

-- This describes html documents
type HTML = [HTML_element]
data HTML_element
    = HTML_text String
    | HTML_font Font_tag HTML
    | HTML_p HTML
data Font_tag = Font_tag (Maybe Int)
                         (Maybe String)
                         (Maybe Font_color)
data Font_color
    = Colour_name String
    | Hex Int
    | RGB Int Int Int

[end]
```

```
# Frameworks:
You can work with frameworks in Haskell like the following:
[haskell]
main = framework plugin1 plugin2 plugin 3
plugin1 = ...
...
[end]

// the rest of this section covers working with font tags and frameworks,
// not sure if this is going to be properly tested since it seems like it'll
// just be given

// good luck and godspeed
// fly you fool
```