Haskell

Language Features Overview

- **functional**
  - higher order functions, recursion instead of loops
- **declarative feel**
  - focus on describing logic of "what", instead of "how" to compute
- **non-strict ("lazy")**
  - Only computes any sub-expression when absolutely needed
- **pure***
  - No side-effects like re-assignment, printing, etc.
  - * - for all but one tiny corner of the language (the IO monad)
- typing:
  - **strongly statically typed**
  - **type inference with optional ascriptions**

Usage

- **Compile**
  - can compile to a main function **ghc --make -o somename YourFile.hs**
- **Interpret**
  - Can load modules and interpret functions **ghci YourFile.hs**
  - REPL ("read-eval-print loop") – you can interactively load modules, add definitions inline, explore computations
- **Compilation/Type Checking**
  - You will fight the type checker at compilation time much more, and then see far fewer runtime exceptions, than you may be used to.
  - So use the REPL to explore what changes your code needs next

Basic Datatypes

- Some basic types: Int, Float, Double, Bool, Char

- **lists**: [Int] [Double] [a] [a -> b]
  - Comma-separated values in square brackets. **[1,2,3]**
  - These are singly-linked lists. brackets/commas are just "syntactic sugar" for the underlying representation. You can represent one node with the cons operator (:)
    - **headval : tailval** **(1:[])** **(1: (2: (3:[])))**

- **strings** are literally just a list of Char: [Char]
  - Also can use double-quotes as "syntactic sugar" (pretty syntax for usability)

- **tuples** of length 2+:
  - parenthesized comma separated listing. (x,y,z)
- **Lambdas(functions)**:
  - **\ x -> expr**

More about Types

- **type variables**: any lowercase identifiers where a type is expected
    - Example: a ⟶ [a]
        - representing "forall types a, the function from one value of type a to a list of values of type a"
    - Example: (a⟶ b) ⟶ [a] ⟶ [b]
        - "forall types a and b, this accepts a function of type a-to-b, and a list of a's, to return a list of b's.
- **Type ascriptions**:
    - any expression can have a required type ascribed with (::)
    - Useful for narrowing down the type for your intent, and get better error messages
    - Examples:
        - (5::Int)
        - (\x -> x+1)::(Int ⟶ Int)

Definitions

- a file can have many top-level definitions.
- Each definition is one or more equations, with a **pattern** on the lefthand side, and an expression on the righthand side.
  - patterns are for matching your data's shape – it's not an expression you evaluate! Learning where patterns go and where expressions go is an important early step.

- *we'll revisit this more all throughout, and learn more of what happens here.*

```
inc x = x+1

length [] = 0
length (val:vals) = 1 + (length vals)
```

Lists

- focus on the fact that they are singly-linked lists
  - We process them one node at a time
  - Pattern matching lets us focus on a list either being empty [] or non-empty (using the cons(:) constructor)

```
isEmpty :: [a] ⟶ Bool
isEmpty [] = True
isEmpty (x:xs) = False

length :: [a] ⟶ Int
length [] = 0                              -- [] means empty list
length (x:xs) = 1 + length (xs)            -- (x:xs) is a pattern; x is the head value,
                                           --  xs is the rest of the list
```

Expressions

- **Branching**
  - If-expressions
    - if expr then expr else expr
  - Case statements
    - case expr of
      pattern -> expr
      pattern -> expr
      . . .

- **Iteration**
  - Recursion only (no loops!)

- **functions**
  - anonymous: lambdas
    - \x -> expr
  - named: let-expressions
    - let f x = expr in expr
  - higher-order functions
    - arg. or return type is a function
  - partial applications are common
    - Feed some but not all args.

Abstract Data Types

- Build-your-own datatypes.
- Some basic types are defined this way:
  - data Bool = True | False
  - data Maybe a = Just a | Nothing
  - data Color = Green| Blue | RGB Int Int Int
  - data Either a b = Left a |Right b
- Give a name, perhaps some type variables, and then different constructors that can each take some number of arguments.
- may feel much like our lambda calculus extensions

General file contents

- At the top, a module statement

```
module Homework4 where
```

- Next, maybe some import statements

```
import Data.List
import Prelude hiding (zipWith, any)
```

- The rest of the file is just "top-level definitions".

```
add x y = x+y

isEmpty [ ] = True
isEmpty (x:xs) = False
```

Pattern Matching

```
map f [] = []
map f (x:xs) = (f x) : map f xs

justs :: [Maybe a] -> [a]
justs (Nothing:xs)  =   justs xs
justs ((Just x)  :xs)  = x:justs xs
justs [ ]              = []
```

- We see a focus on the ***shape*** of our data
  - We know what type we have, but which constructor was used?
  - We need:
    - a constructor, with sub-patterns for each of its arguments.
    - simple variable names work as guaranteed-match patterns
    - concrete values, e.g. [ ], 5, True, etc. (some are truly just more constructors)

- We define functions as a series of patterns-to-expressions
  - In the order presented, if the pattern matches, simplify to that expression (take that path)
  - Variables and concrete values allowed in patterns
  - Expressions don't occur inside patterns (common beginner's syntactic mistake!)
  - the "don't-care" wildcard underscore _ is useful.
    - Matches one thing that you won't be using. Example: middle (_,v,_) = v

Top-Level Definitions

- Functions
  - Can have multiple equation lines, each with a different pattern of arguments
  - A function with no arguments? These variables are like "zero-argument" functions

- datatype definitions
  - data Bool = True | False
  - data Color = Green | Gold | RGB Int Int Int
  - data Optional a = Present a | Empty
  - data MyList a = Cons a (MyList a) | Nil

- Type synonyms
  - Only benefit is ease of use, e.g.:
    - type Name = String
    - type State = [(String,Int)]

Some corner cases, catalogued

- Patterns (functions or case expressions)
  - non-exhaustive pattern match error:
    - the function was called with data that didn't match any of our provided patterns.
  - overlapping patterns warning:
    - we wrote a pattern that can't ever be used, because an earlier more general pattern wins

- Numbers
  - Type classes and the provided numeric types
    - Many functions are more general than we're expecting, and we'll choose to use type ascriptions to keep life simple
    - converting between number types is weird…

  - Negative numbers, precedence of (-)
    - Partial application makes negative numbers a bit cumbersome. We'll often need to parenthesize them.