Lecture Notes
CS141

Basics 1

1. **Church-Turing Thesis**: Any algorithm that can be described using a Turing machine can also be expressed in λ-calculus and vise-versa.
2. **Programming paradigm** is collective name for programming languages based on style, evaluation and the toolset.**Imperative** languages uses step-by-step computation, while **functional** languages behave recursively and mathematically.
3. **Pure function**: given same inputs, will produce same output. **Lazy** functions only evaluate values of computations when needed (called later).
4. Haskell is a strongly typed, lazy, purely functional programming language.

Modules

```haskell
-- Module: file with Haskell source code.
module ThisModule where
import SomeOtherModule
import SubFolder.SomeModule
```

Lambda and Function application

```
double = \x -> x * 2 -- is same as:
double x = x * 2
-- partial function application
-- (providing some of the inputs turns functions into ones that accept remaining arguments)
min x y = ...      min x = \y -> ...      min = \x -> \y -> ... -- are all equvalent
plusFive = (+5) -- will take another argument, and return their sum

-- Infix operators come between arguments, making the following two equvalent:
max a b      a `max` b
```

Pattern Matching

```
fac x = if x == 0              fac x = case x of      fac 0 = 1
    then 1                       0 -> 1       fac n = n * fac (n - 1)
    else x * fac (x-1)           n -> n * fac (n-1)
```

Type Class

```haskell
class Eq a where -- Type Class called Eq
  (==) :: a -> a -> Bool

instance Eq Bool where -- Instance of the Eq typeclass on Booleans
  -- (==) :: a -> a -> Bool
  True  == True  = True
  False == False = True
  _     == _     = False

instance (Show a, Show b) => Show (a,b) where -- subtype polymorphic instance of a Show
   typeclass. a,b have to already be instances of Show
  show (x,y) = "(" ++ show x ++ "," ++ show y ++ ")"

class Foldable t where
  foldr :: (a -> b -> b) -> b -> t a -> b

isNotEmpty :: Foldable t => t a -> Bool
isNotEmpty xs = foldr f z xs -- can do this because xs is foldable

class Functor f where
  fmap :: (a -> b) -> f a -> f b -- SAME AS <$>

class Semigroup a where
  (<>) :: a -> a -> a -- has to be associative

instance Semigroup [a] where
  x <> y = x ++ y

class Semigroup a => Monoid a where
  mempty :: a -- Monoid is a semigroup with identity ( x <> mempty === mempty <> x === x)

instance Num a => Semigroup (Sum a) where
  Sum x <> Sum y = Sum (x + y)
instance Num a => Monoid (Sum a) where mempty = 0

($)   :: (a -> b) -> a -> b
(<$>) :: Functor f => (a -> b) -> f a -> f b
(<*>) :: Applicative f => f (a -> b) -> f a -> f b
(<>) :: Semigroup f => f (a -> b) -> f a -> f b
(=<<) :: Monad f => (a -> f b) -> f a -> f b

instance Applicative ((->) e) where -- have to define pure, <*>
  pure a = \x -> a
  f <*> g = \x -> (f x) (g x)
```

Parametric polymorphism will work for any type passed in, and always uses the same implementation.
Ad-hoc polymorphism only works on types where we have expicitly given a type-specific implementation for the polymorphic function to use.
Subtype polymorphism is able to use functions that are inherited from a given type, like ad-hoc with prerequisites

Data Types

```haskell
data Bool = False | True -- Boolean data type

data RPSThrow = Rock | Paper | Scissors -- custom data type
  deriving (Show, Eq) -- inherit Show and Eq type class behaviour

data Date = BC Int Int Int | AD Int Int Int -- data types can take inputs

data Maybe a = Nothing | Just a -- polymorphic data type

-- newtype - giving a custom name to an existing type to treat it in a special way.
newtype Mark = Mark Int                          -- a wrapper type around Int (bc 1 value)

type MMBool = Maybe (Maybe Bool) -- type synonym (type alias)
type String = [Char] -- String is just a list of Chars

data BinTree a = Empty | Node a (BinTree a) (BinTree a) -- Tree ADT definition

data Point a = Point { x :: a, y :: a } -- polymorphic Record Syntax
```

Precedence Fixity and Folding

```haskell
infix[l/r] [precedence] [operator]

infixr 0 $ -- parenthesies have lowest RIGHT precedence
($) :: (a -> b) -> a -> b
f $ x = f x -- so double $ double $ 5 === double(double 5)

infixr 9 . -- function composition has highest right precedence
(.) :: (b -> c) -> (a -> b) -> a -> c
g . f = \x -> g (f x) -- so double

sameList list = foldr (\x xs -> x:xs) [] list -- return to the same list using folding

foldl f z (x:xs) = foldl f (f z x) xs -- fold each new value directly into base case
```

Monads and IO

```haskell
class Monad m where
  (>>=) = m a -> (a -> m b) -> m b
pure x >>= f === f x

instance Monad (Either e) where
  Left x  >>= f = Left x
  Right x >>= f = f x

newtype State s a = St { runState :: s -> (a, s) }

pure lifts a single value into an applicative data type

addTwoNums :: Maybe Int -> Maybe Int -> Maybe Int
addTwoNums mx my =  do
  x <- mx
  y <- my
  return (x+y)

main :: IO ()
main = forever $ do -- forever keeps it going forever.
  foo <- getLine
  case parse five "" foo of
    Left err -> putStrLn $ errorBundlePretty err
    Right v -> print v
```

Parsing

```haskell
type Parser = Parsec Void String
parseTen :: Parser Int
parseTen = do
  _ <- char '5'
  _ <- char '+'
  _ <- char '5'
  pure 10

parseInteger :: Parser Int
parseInteger = do
  digits <- takeWhile1P "integer" isDigit
  pure (read digits)

(<|>) :: Alternative f => f a -> f a -> f a -- choice between items upon fail
```

Separation of Concerns: A section of code should do as few different jobs as possible. Moreover, code should interact only with that which it needs to do its job.
Principle of Least Surprise: Code (and any programs built from that code) should not be "surprising". Things should work how they would be expected to work, and be arranged how a common practitioner would expect them to be arranged.
Make use of libraries where possible, rather than reimplementing nontrivial amounts of logic. Don't import a whole library just to make use of a single function you could implement yourself. Use explicit export lists to present an interface which ensures your code is used as intended. Use explicit import lists to import only the functionality that you need from modules that you are importing.
Do not write partial functions.