Introduction to Haskell

Haskell’s type system

- Three aspects
  - Strong types:
  - Static types
  - Automatic inference

Haskell’s type system

- Three aspects
  - Strong types:
    - Writing expressions that don’t make sense.
    - Expression that obeys a language’s type rules are **well typed**.
    - Expression that disobeys the type rules is **ill typed.**
      - Leads to type error
  - Haskell doesn’t allow automatic coersion

Haskell’s type system

- Three aspects
  - Static types:
    - Compiler knows the type of every value and expression at compile time, before any code is executed.
    - Expressions whose types don’t match are detected and rejected with an error.
    - Example: True && "false"
  - Haskell’s combination of strong and static typing makes it impossible for type errors to occur at runtime

Haskell's type system

- Three aspects
  - Type inference:
    - Haskell compiler can automatically deduce the types of almost all expressions in a program.
    - Haskell allows us to explicitly declare the type of any value.
    - But the presence of type inference means that this is almost always optional.

- It is safer than popular statically typed languages and often more expressive than dynamically typed languages

Data Types

- Char value
- A bool value
- An Int type
- An Integer value
- Value of Type Double

Data Types

- To write a type explicitly, we use the notation: *expression :: MyType*

to say that expression has the type MyType.

- If we omit the :: and the type that follows, a Haskell compiler will infer the type of the expression:
- Try
  - :type 'a'
  - 'a' :: Char
  - [1, 2, 3] :: Int

Function Application

- Write the name of the function followed by its arguments.
- Merely writing the name of the function, followed by each argument in turn, is enough.
- Grouping parameters in parentheses is not required.

Example:

Odd 3

Odd 6

Compare 2 3

Compare 10 10

Compare 2 3 == LT

Parentheses required for complex expressions with complex arguments

Compare sqrt 3 sqrt 4

Compare (sqrt 3) (sqrt 4)

Composite Data Types

- Lists:
  - Homogeneous data structure
  - Basic Head and Tail functions are provided
  - The head function returns the first element of a list.
  - >head [1, 2, 3]
  - >head[“abc”, ”def”, “ghi”]
  - >head ['a','b','c']

  - Tail, returns all but the head of a list:
  - >tail [1,2,3,4]
  - > tail [2,3,4]
  - > tail [True,False]
  - > tail "list“

  - >head []
  - > tail []

Composite Data Types

- Tuple
  - A tuple is a fixed-size collection of values, where each value can have a different type.
  - Example: Tracking two pieces of information about a book: its year of publication—a number—and its a title—a string
  - Write a tuple by enclosing its elements in parentheses and separating them with commas.

  - > :type (True, "hello") ➔ (True, "hello") :: (Bool, [Char])
  - > (4, ['a', 'm'], (16, True)) => (4,"am",(16,True))
  - (), that acts as a tuple of zero elements

  - A tuple's type represents the number, positions, and types of its elements.
  - > :type (False, 'a')
  - > :type ('a', False)

Functional Programming

Different programming Paradigms

- Imperative programming
  - Procedural programming
  - Object oriented programming

- Declarative Programming Paradigm
  - Logic programming
  - Functional programming
  - Database programming approach

Functional Programming Paradigm

- Style of building the structure and elements of computer program in form of pure functions.
- Computation is treated as an evaluation of mathematical functions.
- Changing-state and mutable data is avoided.
- Emphasis is on “what to do” instead of “how to do”
- Features:
  - Pure functions
  - Referentially transparent
  - Recursion
  - Functions are first class and higher order
  - Variables are immutable

Functional Composition

We will use the Haskell notation

```
f :: X -> Y
```

to assert that `f` is a function taking arguments of type `X` and returning results of type `Y`. For example,

```
sin     :: Float -> Float
age     :: Person -> Int
add     :: (Integer,Integer) -> Integer
logBase :: Float -> (Float -> Float)
```

Suppose `f :: Y -> Z` and `g :: X -> Y` are two given functions.

```
f . g :: X -> Z
```

```
(f . g) x = f (g x)
```

Functional Programming Paradigm

- Getting most frequent words from a book

What is wanted as output? Answer: something like

```
the: 154
of: 50
 a: 18
 and: 12
 in: 11
```

" the: 154\n of: 50\n a: 18\n and: 12\n in: 11\n“
Design a function, commonWords say, with type commonWords :: **Int -> [Char] -> [Char]**

```
words :: [Char] -> [[Char]]
```

```
type Text = [Char]
type Word = [Char]
```

Functional Programming Paradigm

words :: [Char] -> [[Char]]

type Text = [Char]
type Word = [Char]
So now we have words :: Text -> [Word]

Functional Programming Paradigm

```
commonWords :: Int -> Text -> String
commonWords n = concat . map showRun . take n .
                sortRuns . countRuns . sortWords .
                words . map toLower
```

Evaluations

```
sqr :: Integer -> Integer
sqr x = x*x
```

```
  sqr (3+4)
= sqr 7
= let x = 7 in x*x
= 7*7
= 49
```

```
  sqr (3+4)
= let x = 3+4 in x*x
= let x = 7 in x*x
= 7*7
= 49
```

The method on the left is called **innermost reduction and also eager evaluation**; t
he one on the right is called **outermost reduction or lazy evaluation.** With eager evaluation arguments are always evaluated before a function is applied. With lazy evaluation the definition of a function is installed at once and only when they are needed are the arguments to the function evaluated.

Evaluations

```
  fst (sqr 1,sqr 2)
= fst (1*1,sqr 2)
= fst (1,sqr 2)
= fst (1,2*2)
= fst (1,4)
= 1
```

```
  fst (sqr 1,sqr 2)
= let p = (sqr 1,sqr 2)
  in fst p
= sqr 1
= 1*1
= 1
```

- Under eager evaluation the value sqr 2 is computed, while
- Under lazy evaluation that value is not needed and is not computed.

Evaluations

```
infinity :: Integer
infinity = 1 + infinity

three :: Integer -> Integer
three x = 3
```

```
  three infinity
= three (1+infinity)
= three (1+(1+infinity))
= ...
```

```
  three infinity
= let x = infinity in 3
= 3
```

Evaluations

```
factorial :: Integer -> Integer
factorial n = fact (n,1)

fact :: (Integer,Integer) -> Integer
fact (x,y) = if x==0 then y else fact (x-1,x*y)
```

```
  factorial 3
= fact (3,1)
= fact (3-1,3*1)
= fact (2,3)
= fact (2-1,2*3)
= fact (1,6)
= fact (1-1,1*6)
= fact (0,6)
= 6
```

```
  factorial 3
= fact (3,1)
= fact (3-1,3*1)
= fact (2-1,2*(3*1))
= fact (1-1,1*(2*(3*1)))
= 1*(2*(3*1))
= 1*(2*3)
= 1*6
= 6
```

Types and Type classes

```
data Bool = False | True
```

This is an example of a *data declaration.*

```
to :: Bool -> Bool
to b = not (to b)
```

The prelude definition of `not` is

```
not :: Bool -> Bool
not True  = False
not False = True
```

Haskell has built-in compound types, such as

`[Int]`	a list of elements, all of type `Int`
`(Int,Char)`	a pair consisting of an `Int` and a `Char`
`(Int,Char,Bool)`	a triple
`()`	an empty tuple
`Int -> Int`	a function from `Int` to `Int`

Types and Type classes

```
data Bool = False | True
```

This is an example of a *data declaration.*

```
to :: Bool -> Bool
to b = not (to b)
```

The prelude definition of `not` is

```
not :: Bool -> Bool
not True  = False
not False = True
```

Haskell has built-in compound types, such as

`[Int]`	a list of elements, all of type `Int`
`(Int,Char)`	a pair consisting of an `Int` and a `Char`
`(Int,Char,Bool)`	a triple
`()`	an empty tuple
`Int -> Int`	a function from `Int` to `Int`

Types and Type classes

Type

- Type :
  - Kind of label that every expression has.
  - Expression True is a boolean, “hello”is a string, etc.
  - :t on an expression prints out the expression followed by :: its type.
  - :: is read as "has type of".
  - (True, 'a') has a type of (Bool, Char)
  - "HELLO!" yields a [Char] or String.

- **Try following:**
  - :t (4==5)
  - :t “This is a world”
  - :t (‘a’, ‘b’, ‘c’)
  - :t [‘a’, ‘b’, ‘c’]

Defining a New Data Type

data BookInfo = Book Int String [String] deriving (Show)

- BookInfo after the data keyword is the name of the new type.
- BookInfo is a type constructor.
- We will use its type constructor to refer to it
- The Book that follows is the name of the value constructor (sometimes called a data constructor).
- The Int, String, and [String] that follow are the components of the type.

Tuple

- We write a tuple by enclosing its elements in parentheses and separating them with commas. We use the same notation for writing its type:

```
ghci> :type (True, "hello")
(True, "hello") :: (Bool, [Char])
ghci> (4, ['a', 'm'], (16, True))
(4,"am",(16,True))

ghci> :type (False, 'a')
(False, 'a') :: (Bool, Char)
ghci> :type ('a', False)
('a', False) :: (Char, Bool)
```

- In this example, the expression (False, 'a') has the type (Bool, Char), which is distinct from the type of ('a', False).

Type

- Functions also have types.
- While writing our own functions, we can give them an explicit type declaration.
- Ex:

```
removeNonUppercase :: [Char] -> [Char]
removeNonUppercase st = [ c | c <- st, c `elem` ['A'..'Z']]
```

`removeNonUppercase` maps from a string to a string

```
addThree :: Int -> Int -> Int -> Int
addThree x y z = x + y + z
```

- The parameters are separated with ->
- There's no special distinction between the parameters and the return type.
- The return type is the last item in the declaration

Type Variable

- Consider following examples

```
take 3 [1,2,3,4,5] = [1,2,3]
take 3 "category"  = "cat"
take 3 [sin,cos]   = [sin,cos]
```

```
ghci> :t take
take :: Int -> [a] -> [a]
```

```
ghci> :t head
head :: [a] -> a
```

```
ghci> :t fst
fst :: (a, b) -> a
```

What is
a / b ???

Type Class

- A typeclass is a sort of interface that defines some behavior.
- If a type is a part of a typeclass, that means that it supports and implements the behavior the typeclass describes.
- It has a collection of named methods, such as (+), which can be defined differently for each instance of the type class (Int, Float, Integer).

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

- Type classes therefore, provide for overloaded functions, functions with the same name but different definitions.
- Another kind of polymorphism.

```
ghci> :t (==)
(==) :: (Eq a) => a -> a -> Bool
```

- New : **the => symbol.**

Type Class

```
ghci> :t (==)
(==) :: (Eq a) => a -> a -> Bool
```

- Everything before the => symbol is called a class constraint.
- It says the equality function takes any two values that are of the same type and returns a Bool.
- The type of those two values must be a member of the Eq class (this was the class constraint).

Type Class : Eq

```
ghci> :t (==)
(==) :: (Eq a) => a -> a -> Bool
```

- The Eq typeclass provides an interface for testing for equality.
- Any type where it makes sense to test for equality between two values of that type should be a member of the Eq class.
- All standard Haskell types except for IO and functions are a part of the Eq typeclass.
- The functions its members implement are == and /=.
- If there's an Eq class constraint for a type variable in a function, it uses == or /= somewhere inside its definition.
- Example: The `elem` function has a type of (Eq a) => a -> [a] -> Bool :
  - it uses == over a list to check whether some value we're looking for is in it.

Type Class : Ord

```
ghci> :t (>)
(>) :: (Ord a) => a -> a -> Bool
```

- Ord is for types that have an ordering.
- All the types we covered so far except for functions are part of Ord.
- Ord covers all the standard comparing functions such as >, = and <=.
- The compare function takes two Ord members of the same type and returns an ordering.
- Ordering is a type that can be GT, LT or EQ, meaning greater than, lesser than and equal, respectively.
- To be a member of Ord, a type must first have membership in the prestigious and exclusive Eq club.

```
ghci> 5 >= 2
True
ghci> 5 `compare` 3
GT
```

Type Class : Show

- Members of Show can be presented as strings.
- All types covered so far except functions are a part of Show.
- The most used function that deals with the Show typeclass is show.
- It takes a value whose type is a member of Show and presents it to us as a string.

```
ghci> show 3
"3"
ghci> show 5.334
"5.334"
ghci> show True
"True"
```

Type Class : Read

```
ghci> :t read
read :: (Read a) => String -> a
```

- Read is sort of the opposite typeclass of Show.
- The read function takes a string and returns a type which is a member of Read.

```
ghci> read "True" || False
True
ghci> read "8.2" + 3.8
12.0
```

- What about `ghci> read "4"` ?????

Type Class : Read

- What about `ghci> read "4"` ?????
- Error
- What GHCI is telling us here is that it doesn't know what we want in return.
- Cannot infer type as results are not used.
- Solution: Type annotations can be used

```
ghci> read "5" :: Float
5.0
ghci> (read "5" :: Float) * 4
20.0
ghci> read "[1,2,3,4]" :: [Int]
[1,2,3,4]
```

Type Class : Enum

- Enum members are sequentially ordered types— they can be enumerated.
- Advantage : We can use its types in list ranges.
- They also have defined successors and predecesors.
- We can get with the succ and pred functions.
- Types in this class: (), Bool, Char, Ordering, Int, Integer, Float and Double.

```
ghci> ['a'..'e']
"abcde"
ghci> [LT .. GT]
[LT,EQ,GT]
ghci> [3 .. 5]
[3,4,5]
ghci> succ 'B'
'C'
```

```
ghci> :t [1..7]
[1..7] :: (Num a, Enum a) => [a]
```

Type Class : Enum

Bounded members have an upper and a lower bound.

```
ghci> minBound :: Int
-2147483648
ghci> maxBound :: Char
'\1114111'
ghci> maxBound :: Bool
True
ghci> minBound :: Bool
False
```

```
ghci> :t minBound
minBound :: Bounded a => a
```

All tuples are also part of Bounded if the components are also in it.

```
ghci> maxBound :: (Bool, Int, Char)
(True,2147483647,'\1114111')
```

Type Class : Num

```
ghci> :t 20
20 :: (Num t) => t
```

- Num is a numeric typeclass.
- Its members have the property of being able to act like numbers.
- Whole numbers are also polymorphic constants.
- They can act like any type that's a member of the Num typeclass.

```
ghci> 20 :: Int
20
ghci> 20 :: Integer
20
ghci> 20 :: Float
20.0
```

```
ghci> :t (*)
(*) :: (Num a) => a -> a -> a
```

Type Class : Num

- Num is a numeric typeclass.

```
ghci> 20 :: Int
20
ghci> 20 :: Integer
20
ghci> 20 :: Float
20.0
```

```
ghci> :t (*)
(*) :: (Num a) => a -> a -> a
```

- Evaluate (5 :: Int)* (6 :: Integer)   ???????

- 5 * (6 :: Integer)   ?????

Type Class : Integral

- Integral includes only integral (whole) numbers.
- In this typeclass are Int and Integer.
- fromIntegral : A very useful function .
- Type declaration :

- **fromIntegral : : (Num b, Integral a) => a -> b.**

- It takes an integral number and turns it into a more general number.

- Evaluate **length [1,2,3,4] + 3.2** ?????

- Evaluate **fromIntegral (length [1,2,3,4]) + 3.2** ?????

Pattern Matching

- Pattern matching consists of:
  - specifying patterns to which some data should conform
  - then checking to see if it does
  - deconstructing the data according to those patterns.

- It helps to define separate function bodies for different patterns

- Pattern match can be on any data type — numbers, characters, lists, tuples, etc.

- Example : Check if a number if 7??

```
checkSeven :: (Integral a) => Int -> Bool
checkSeven 7 = True
checkSeven x = False
```

Can be done easily using If-Else statement as well.

Type Class :Pattern Matching

- Suppose we want a function that says the numbers from 1 to 5 in string format and says "Not between 1 and 5" for any other number?

```
sayMe :: (Integral a) => a -> String
sayMe 1 = "One!"
sayMe 2 = "Two!"
sayMe 3 = "Three!"
sayMe 4 = "Four!"
sayMe 5 = "Five!"
sayMe x = "Not between 1 and 5"
```

- **IF-Else will be complicated while pattern matching will be clean**

- **Last statement is catch-all one**

Type Class :Pattern Matching

```
sayMe :: (Integral a) => a -> String
sayMe 1 = "One!"
sayMe 2 = "Two!"
sayMe 3 = "Three!"
sayMe 4 = "Four!"
sayMe 5 = "Five!"
sayMe x = "Not between 1 and 5"
```

- **Can we reverse the order of statements ????**
- **Try putting the last statement in the beginning?????**

Type Class :Pattern Matching

- Rewriting Factorial using Pattern matching:

Pattern Matching

- How will this work ??

```
charName :: Char -> String
charName 'a' = "Albert"
charName 'b' = "Broseph"
charName 'c' = "Cecil"
```

- charName ‘a’ = ???
- CharName ‘h’ = ???

Pattern Matching: Tuples

- Pattern matching ca be used with tuples as well.
- Consider adding two vectors in 2-D space.
- One way:

```
addVectors :: (Num a) => (a, a) -> (a, a) -> (a, a)
addVectors a b = (fst a + fst b, snd a + snd b)
```

- Using Pattern Matching:

```
addVectors :: (Num a) => (a, a) -> (a, a) -> (a, a)
addVectors (x1, y1) (x2, y2) = (x1 + x2, y1 + y2)
```

- Easier to work on 3-tuple data and 4 – tuple data as well.

Pattern Matching: Tuples

- We can write our own functions for triples as well

```
first :: (a, b, c) -> a
first (x, _, _) = x

second :: (a, b, c) -> b
second (_, y, _) = y

third :: (a, b, c) -> c
third (_, _, z) = z
```

- '_' means that we really don't care what that part is, so we just write a _.

Pattern Matching: List

- Lists themselves can also be used in pattern matching.

- Match with the empty list [ ] or
- Any pattern that involves : and the empty list.

- Example : [1,2,3] is just syntactic sugar for 1:2:3:[].

- A pattern like x:xs will :
  - bind the head of the list to x
  - the rest of it to xs.
  - If there's only one element so xs ends up being an empty list.
  - We can bind it against more number of elements.
  - Like x:y:z:xs will work for lists with minimum three elements

Pattern Matching: List examples

- Following function tells us some of the first elements of the list

```
tell :: (Show a) => [a] -> String
tell [] = "The list is empty"
tell (x:[]) = "The list has one element: " ++ show x
tell (x:y:[]) = "The list has two elements: " ++ show x ++ " and " ++ show y
tell (x:y:_) = "This list is long. The first two elements are: " ++ show x ++ " and " ++ show y
```

- Note that (x: [ ]) and (x:y: [ ]) could be rewriten as [x] and [x,y]
- We can't rewrite (x:y:_) with square brackets because it matches any list of length 2 or more.

Pattern Matching: List examples

- **Think of rewriting length function on Lists using Pattern matching**

Pattern Matching: List examples

- **Try writing a function that can add all elements for lists**

Pattern Matching: List examples

- **What is Pattern??**
  - Those are a handy way of breaking something up according to a pattern and binding it to names whilst still keeping a reference to the whole thing.
  - Can be done by putting a name and an @ in front of a pattern.
  - Example: the pattern xs@(x:y:ys) .
  - This pattern will match exactly the same thing as x:y:ys but you can easily get the whole list via xs instead of repeating yourself by typing out x:y:ys in the function body again.

```
capital :: String -> String
capital "" = "Empty string, whoops!"
capital all@(x:xs) = "The first letter of " ++ all ++ " is " ++ [x]
ghci> capital "Dracula"
"The first letter of Dracula is D"
```

Guards

- Guards are a way of testing whether some property of a value (or several of them) are true or false.
- a simple function that berates you differently depending on your BMI (body mass index).

```
bmiTell :: (RealFloat a) => a -> String
bmiTell bmi
    | bmi <= 18.5 = "You're underweight, you emo, you!"
    | bmi <= 25.0 = "You're supposedly normal. Pffft, I bet you're ugly!"
    | bmi <= 30.0 = "You're fat! Lose some weight, fatty!"
    | otherwise   = "You're a whale, congratulations!"
```

Guards

- Guards are a way of testing whether some property of a value (or several of them) are true or false.
- a simple function that berates you differently depending on your BMI (body mass index).

```
bmiTell :: (RealFloat a) => a -> a -> String
bmiTell weight height
    | weight / height ^ 2 <= 18.5 = "You're underweight, you emo, you!"
    | weight / height ^ 2 <= 25.0 = "You're supposedly normal. Pffft, I bet you're ugly!"
    | weight / height ^ 2 <= 30.0 = "You're fat! Lose some weight, fatty!"
    | otherwise                   = "You're a whale, congratulations!"
```

Types and Type classes

```
data Bool = False | True
```

This is an example of a *data declaration.*

```
to :: Bool -> Bool
to b = not (to b)
```

The prelude definition of `not` is

```
not :: Bool -> Bool
not True  = False
not False = True
```

Haskell has built-in compound types, such as

`[Int]`	a list of elements, all of type `Int`
`(Int,Char)`	a pair consisting of an `Int` and a `Char`
`(Int,Char,Bool)`	a triple
`()`	an empty tuple
`Int -> Int`	a function from `Int` to `Int`