Haskell Types

COMP360

“Should array indices start at 0 or 1? My compromise of 0.5 was rejected without, I thought, proper consideration.”

Stan Kelly-Bootle

British author,
singer-songwriter
and computer scientist

Haskell To Do

- Read the Haskell tutorial at learnyouahaskell.com
- The first Haskell programming assignment is due at noon on Tuesday, April 4, 2017

Usual Simple Data Types

- Int – whole numbers
- Integer – whole numbers that can be very large
- Double – floating point numbers
- Bool – true or false
- Char – character

- Datatype names all start with a CAPITAL letter

Some Haskell Typeclasses

- **Eq** is used for types that support equality testing
- **Ord** is for types that have an ordering.
- **Show** can be presented as strings
- **Read** takes a value out of a string
- **Num** are numbers
- **Integral** are whole numbers

Defining Function Types with Typeclasses

- You can specifically define the classtype of a parameter

```
sumsqr :: Num a => [a] -> a
sumsqr lst = if length lst == 0 then etc.
```

- The typeclass specified to the left of the => is called a *class constraint*
- The type definition states that the function takes a list of numerical values and returns a number

Practice

- Create a Haskell function to sum a list of list of numbers

sumList [[4,5,6] , [1,2]] returns 18

sumList :: Num a => [[a]] -> a

- You may wish to use the **sum** method to sum a list

Possible Solution

```
-- create a list of the sum of lists
sumList :: Num a => [[a]] -> a
sumList dog = if null dog then 0
                          else sum (head dog) + sumList (tail dog)
```

or

```
sumList dog = sum [ sum cat | cat <- dog]
```

Haskell Patterns

- You can write multiple definitions of a function
- Haskell will consider them in order and execute the first on that fits
- You can put constants as the parameters in the definition. If you call the function with that value, it will execute that function definition

Fibonacci Pattern Example

- You might have written the Fibonacci function as

```
fib n = if n <= 2 then 1 else fib (n-1) + fib (n-2)
```

- It can also be written as

```
fib 1 = 1
fib 2 = 1
fib n = fib (n-1) + fib (n-2)
```

Empty List Pattern

- You can specify an empty list as a parameter. The definition will be use if it is executed with an empty list

```
mySum :: Num a => [a] -> a
mySum [] = 0
mySum str = head str + mySum (tail str )
```

Practice

- Rewrite your Haskell function to sum a list of list of numbers using a pattern

sumList [[4,5,6] , [1,2]] returns 18

```
sumList :: Num a => [[a]] -> a
```

Possible Solution

```
-- create a list of the sum of lists
sumList :: Num a => [[a]] -> a
sumList [] = 0
sumList dog = sum (head dog) + sumList (tail dog)
```

Splitting Complex Values

- The parameters to a function can be specified by their parts
- A list parameter can be specified as (cat:dog)
- The ":" operator is normally used to add a new element to the beginning of a list
- Here it specifies the input list is composed of a head named cat and the tail named dog

Using Split Parameters

- Consider a function that uses the head and tail of a list
- You can describe the list as a head:tail

```
sumsqr :: Num a => [a] -> a
sumsqr [] = 0
sumsqr (cat:dog) = cat * cat + sumsqr dog
```

And More

- You can split an input parameter into more than two parts

myFunc (ant : bird : cat) = bird

- This returns the second element of a list
- (ant : bird : cat) can be read as a list where ant is the head of a list where bird is the head and cat the tail

Don't Care Variables

- Sometimes you are not going to use a part of the complex type, but need to specify that is there to completely describe the type
- The variable name _ or underscore represents a variable that will not be used

```
myFunc ( _ : bird : _ ) = bird
```

Practice

- Rewrite your Haskell function to sum a list of list of numbers using a pattern

sumList [[4,5,6] , [1,2]] returns 18

```
sumList :: Num a => [[a]] -> a
```

Possible Solution

```
-- create a list of the sum of lists
sumList :: Num a => [[a]] -> a
sumList [] = 0
sumList (cat:dog) = sum cat + sumList dog
```

Guards

- You can put conditions on the parameters of a function

```
grade score
    | score >= 90 = "A"
    | score >= 80 = "B"
    | score >= 70 = "C"
    | otherwise   = "You flunk"
```

Haskell Reading

- Read the Haskell tutorial at learnyouahaskell.com
- Learn Haskell up to and including Higher order functions in chapter 6
- This is as far as we will progress in Haskell