6️⃣

Week 6: Lists, Tuples, and Types

Beginning

Statically Typed

Haskell is a statically typed language which means that data types are defined during compile time and cannot change during runtime.

This enables static type checking.

Example File

```
-- fileName: baby.hs

doubleNumber x = x * 2
tripleNumber x = x * 3
```

Compiling

To use the interpreter in the terminal for Haskell, type:

```
ghci
```

Then, to compile the file (and the functions inside), type:

```
:l **fileName** -- In this case :l baby.hs
```

To use the functions, type:

```
Prelude> let x = 30
Prelude> doubleNumber x
```

```
60
Prelude> tripleNumber x
90
```

Lists

Defining Lists

```
let list = [5, 10, 15, 20]
```

Concatenation (++)

Concatenations with the **++** sign, append items to end of lists.

```
Prelude> [1,2,3,4] ++ [7,8,9]
[1,2,3,4,7,8,9]

Prelude> "Hello" ++ " " ++ "World"
"Hello World"

Prelude> ['H','e','l','l','o'] ++ [' ', 'W', 'o', 'r', 'l', 'd']
"Hello World"

-- Strings are lists of characters and we can use list functions on strings
```

Cons (:)

Cons add with the **:** sign, prepend items to the head of lists.

```
Prelude> 'A' : " BIG MESS"
"A BIG MESS"

Prelude> 54 : [45,69,27]
[54,45,69,27]
```

Accessing List Elements (!!)

By using the **!!** sign, you can get a list's member by index number. The index starts from 0.

```
"Claude Shannon" !! 10
'n'

[3.4,7.89,9.4,12.0] !! 3
12.0
```

Lists Inside Lists

```
Prelude> let z = [[1,2,3,4],[5,3,3,3],[1,2,2,2,3,4], [1,2,3]]

Prelude> z ++ [[99]]
[[1,2,3,4],[5,3,3,3],[1,2,2,2,3,4], [1,2,3], [99]]

Prelude> [10, 11] : z
[[10, 11], [1,2,3,4],[5,3,3,3],[1,2,2,2,3,4], [1,2,3]]

Prelude> z !! 3
[1,2,3]

-- Using funtions doesn't change the actual lists
```

List Comparison

Comparison is possible with **<, >, <=, >=, == s**igns.

```
Prelude> [5,3,7] > [4,0,0]
True

Prelude> [7,90,45,5] > [7,67,5,6]
True
```

```
Prelude> [7,90,45,5] > [7,90,45,6]
False
```

Using Ranges

Singular patterned lists can easily be created.

```
Prelude> [1..10]
[1,2,3,4,5,6,7,8,9,10]

Prelude> [‘m’..’q’]
“mnopq”

Prelude> [‘A’..’J’]
“ABCDEFGHIJ”
```

Different patterned lists can also be created.

```
Prelude> [2,4..10]
[2,4,6,8,10]

Prelude> [3,6..30]
[3,6,9,12,15,18,21,24,27,30]

Prelude> [20..1]
[ ]

Prelude> [20,19..1]
[20,19,18,17,16,15,14,13,12,11,10,9,8,7,6,5,4,3,2,1]
```

Cycles, repeats, and replicates are also possible to create

```
Prelude> take 11 [13,26..]
[13,26,39,52,65,78,91,104,117,130,143]

-- Cycle
Prelude> take 5 (cycle [4,3,2])
[4,3,2,4,3]
```

```
-- Repeat
Prelude> take 6 (repeat 7)
[7,7,7,7,7,7]

-- Replicate
Prelude> replicate 5 'a'
"aaaaa"
```

❗ caveat emptor: floating point numbers only have finite precision

```
Prelude> [0.1, 0.3..1]
[0.1,0.3,0.5,0.7,0.899999999999,1.0999999999]
```

List Comprehension

Similar to set comprehension in mathematics, e.g. { 2*x | x > 0 and x in Z } (set of all even whole numbers) build lists out of other lists: filter, transform and combine lists.

```
Prelude> [ x*2 | x <- [1..10]]
[2,4,6,8,10,12,14,16,18,20]

Prelude> [ x*2 | x <- [1..10], x*2 >= 12]
[12,14,16,18,20]

Prelude> [ x | x <- [50..100], x `mod` 7 == 5]
[54,61,68,75,82,89,96]

-- The sign /= is used to exclude a number
Prelude> [ x | x <- [10..30], x /= 13, x /= 23, odd x ]
[11,15,17,19,21,25,27,29]

Prelude> [ x + y | x <- [17..20], y <- [10,100,0]]
[27,117,17,28,118,18,29,119,19,30,120,20]
```

```
Prelude> [ x*y | x ← [2,5,10], y ← [8,10,11]]
[16,20,22,40,50,55,80,100,110]

Prelude> [ x*y | x ← [2,5,10], y ← [8,10,11], x*y > 50]
[55,80,100,110]

-- Nested List Comprahension
Prelude> xxs = [[1,3,5,7,8,7,6,2],[2,3,4,5,6,1],[12,6,7,8,9,4,6,77]]
Prelude> [ [x | x ← xs, even x] | xs ← xxs ]
[[8,6,2],[2,4,6],[12,6,8,4,6]]
```

Function Examples

Example 1: Checking Length

```
length' xs = sum [ 1 | _ ← xs]
-------------------------------
Prelude> length' "David"
5
```

Example 2: Keep Lower Case Letters

```
keepLowerCase st = [ c | c ← st, c 'elem' ['a'..'z']]
---------------------------------------------------
Prelude> keepLowerCase "David"
"avid"
Prelude> keepLowerCase "David + 12345"
"avid"
```

Tuples

Lists vs. Tuples

Lists	Tuples
Lists only have homogeneous elements	Tuples can have heterogeneous elements

Lists have flexible size: grow, shrink	Tuples have a fixed size

Defining Tuples

```
Prelude> (1,10)
(1,10)
Prelude> (“Hi!”, ‘a’, 72, 1.01)
(“Hi!”, ’a’, 72, 1.01)
```

❗ Use of tuples **enforces a type discipline**: try entering [(1,2),(1,2,3),(5,6)] at the command line. The type discipline extends to within the tuples, so [(1,2),(1,’a’)] is a **problem** too.

Zip

```
Prelude> zip [1,3,5,7,9] [‘a’,’b’,’c’,’j’,’k’]
[(1,’a’),(3,’b’),(5,’c’),(7,’j’),(9,’k’)]

Prelude> zip [5,3,4,2,6,7,8,9,0,1,2,3] [5,6,7]
[(5,5),(3,6),(4,7)]

Prelude> zip [10..] [5,6,7]
[(10,5),(11,6),(12,7)]
```

Example: Creating Triangles

```
Prelude> let triples = [ (a,b,c) | c ← [1..10], a ← [1..10], b ← [1..10] ]
Prelude> let rightTriangles = [ (a,b,c) | c ← [1..10], a ← [1..c], b ← [1..a], a^2
+ b^2 == c^2 ]
Prelude> let rightTriangles’ = [ (a,b,c) | c ← [1..10], a ← [1..c], b ← [1..a], a^2
+ b^2 == c^2, a+b+c == 24 ]
Prelude> rightTriangles’
[(6,8,10)]
```

Types

- Bool
- Char
- String
- Int - Fixed-precision integers
- Integer - Arbitrary-precision integers

Type Inference

Every expression mush have a valid type in Haskell, which is calculated prior to evaluating the expression. This process is called **Type Inference.**

Haskell programs are type safe because type errors can never occur during evaluation.

Type inference detects a very large class of programming errors, and one of the most powerful and useful features of Haskell.

Example

```
e :: T
False :: Bool
not :: Bool → Bool
not False :: Bool
True && False :: Bool
```

List Types

A list is sequence of values of the same type.

> ℹ️ [T] is the type of lists with elements of type T.

```
[False,True,False] :: [Bool]
['a','b','c','d'] :: [Char]
```

The type of the elements is unrestricted. For example, we can have lists of lists:

```
[['a'],['b','c']] :: [[Char]]
```

Tuple Types

A tuple is a sequence of values of the different types.

> ℹ️ (T1,T2,...,Tn) is the type of n-tuples whose with components have type Ti for any i in 1...n.

```
(False,'a',True) :: (Bool,Char,Bool)
```

Function Types

A function is a mapping from values of one type to values of another type.

> ℹ️ T1 → T2 is the type of functions that map arguments of type T1 to results of type T2.

```
not :: Bool → Bool
isDigit :: Char → Bool
```

Curried Functions

Functions with multiple arguments are also possible by returning functions as results.

```
add :: (Int,Int) → Int -- Tuple input
add' :: Int → (Int → Int) -- Spaced normal input

add' :: Int → (Int → Int)
add' x y = x+y
-- add and add' produce the same final result,
```

```
-- but add takes its two arguments at the same time,
-- whereas add' takes them one at a time.
```

Curry Conventions

To avoid excess parentheses when using curried functions, two simple conventions are adopted. The → arrow associates to the right.

```
Int → (Int → (Int → Int))
-- is equal to
Int → Int → Int → Int
```

For example,

```
mult :: Int → (Int → (Int → Int)) -- This means ((mult x)y)z
mult x y z = x*y*z
```

More examples,

```
curry :: ((a, b) → c) → (a → b → c)
curry g x y = g (x, y)

uncurry :: (a → b → c) → ((a, b) → c)
uncurry f (x, y) = f x y


multiply :: Int → Int → Int
multiply x y = x * y

multiplyUC :: (Int, Int) → Int -- neater, permits partial application
multiplyUC (x, y) = x * y
```

Polymorphic Functions

The function length calculates the length of any list, irrespective of the type of its elements.

```
> length [1,3,5,7]
4
```

```
> length ["Yes","No"]
2
> length [isDigit,isLower,isUpper]
3
```

The method length can take any variable because of the way it is written.

```
length :: [a] → Int -- a shows the inclusion of a type variable
-- For any type a, length takes a list of values of type a and returns an integ
er
```

Many of the functions defined in the standard prelude are **polymorphic**.

```
fst :: (a,b) → a
head :: [a] → a
take :: Int → [a] → [a]
zip :: [a] → [b] → [(a,b)]
```

Overloaded Types

A type with constraints is called overloaded.

The arithmetic operator + calculates the sum of any two numbers of the same numeric type.

```
(+) :: Num a → a → a → a
```

Classes

A class is a collection of types that support certain operations, called the methods of the class.

```
(==) :: a → a → Bool
(/=) :: a → a → Bool
```

Type Classes

Haskell has basic classes as:

- **Eq** - Equality Types (Must define == and /=)
- **Ord** - Ordered Types (Must define <, <=, >, >=)
- **Show** - Showable Types (Allows converting to a human-readable string)
- **Read** - Readable Types (Allows converting a String back into another type)
- **Num** - Numeric Types (Providing +, -, *, /, etc.)
- **Integral** - (Subclass of num for Integer)
- **Fractional** - (subclass of num for Float/Double)
- **Enum** - (Sequentially ordered types that can be enumerated)
- **Bounded** - (Has an upper and lower bound)

```
(==) :: Eq a -> a -> a -> Bool
(<) :: Ord a -> a -> a -> Bool
show :: Show a -> a -> String
read :: Read a -> String -> a
(*) :: Num a -> a -> a -> a

bigzip :: Ord a -> [a] -> [a] -> [(a, a)]
```