8️⃣

Week 8: User Defined Types and Interactive Programs

User Defined Types

Type Declarations

In Haskell, a new name for an existing type can be defined using a type declaration.

```
type String = [Char]
```

```
type Pos = (Int, Int)

origin :: Pos
origin = (0,0)

left :: Pos → Pos
left (x, y) = (x-1, y)
```

Similar to function definitions, type declarations can also have parameters.

```
type Pair a = (a, a)

mult :: Pair Int → Int
mult (m, n) = m * n

copy :: a → Pair a
copy x = (x, x)
```

Data Declarations

A completely new type can be defined by specifying its values by using data declaration

```
data Bool = False | True
-- The values False and True are called the constructors for the type
```

ℹ️ Types and Constructors must always be written in uppercase letters.

Example 1

```
data Answer = Yes | No | Unknown

answers :: [Answer]
answers = [Yes, No, Unknown]

flip :: Answer → Answer
flip Yes = No
flip No = Yes
flip Unknown = Unknown
```

Example 2

The constructor in a data declaration can also have parameters.

```
data Shape = Circle Float | Rect Float Float

square :: Float → Shape
square n = Rect n n

area :: Shape → Float
area (Circle r) = π * r^2
area (Rect x, y) = x * y

-- Circle and Rect can be view as functions that construct values of type S
hape
```

```
Circle :: Float → Shape
Rect :: Float → Float → Shape
```

Example 3

Data declarations themselves can also have parameters.

```
data Maybe a = Nothing | Just a

safeDiv :: Int → Int → Maybe Int
safeDiv _ 0 = Nothing
safeDiv m n = Just (m `div` n)

safeHead :: [a] → Maybe a
safeHead [] = Nothing
safeHead xs = Just (head xs)
```

Recursive Types

In Haskell, new types can be declared in terms of themselves. That is, types can be recursive.

```
data Nat = Zero | Succ Nat
-- Nat is a new type, with constructors zero :: Nat and Succ :: Nat → Nat
```

ℹ️ Using recursion, it is easy to define functions that convert between values of type Nat and Int.

```
nat2int :: Nat → Int
nat2int Zero = 0
nat2int (Succ n) = 1 + nat2int n

int2nat :: Int → Nat
int2nat 0 = Zero
int2nat n = Succ (int2nat (n-1))
```

Two naturals can be added by converting them to integers, adding, and then converting back.

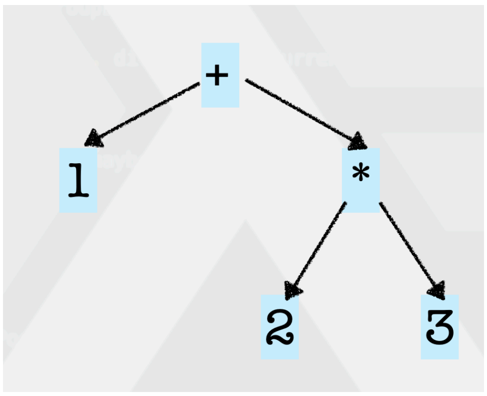
```
add :: Nat → Nat → Nat
add m n = int2nat (nat2int m + nat2int n)
```

However, using recursion the function add can be defined without the need for conversions.

```
add Zero n = n
add (Succ m) n = Succ (add m n)
```

Arithmetic Expressions

Consider a simple form of expressions built up from integers using addition and multiplication.



Using recursion, a suitable new type to represent such expressions can be declared by:

```
data Expr = Val Int
     | Add Expr Expr
     | Mul Expr Expr

-- The above graph would look like:
Add (Val 1) (Mul (Val 2) (Val 3))
```

Using recursion, it is easy to deine functions that convert between values of type Nat and Int.

```
size :: Expr → Int
size (Val n) = 1
size (Add x y) = size x + size y
size (Mul x y) = size x + size y

eval :: Expr → Int
eval (Val n) = n
eval (Add x y) = eval x + eval y
eval (Mul x y) = eval x * eval y
```

Binary Trees

Using recursion, a suitable new type to represent such binary trees can be declared by:

```
data Tree a = Leaf a | Node (Tree a) a (Tree a)
```

```
t :: Tree Int
t = Node (Node (Leaf 5) 2 (Leaf 6)) 1 (Node (Leaf 7) 3 (Leaf 8))
```

Interactive Programs

Until now, we have seen **batch programs** that take inputs at the start and write outputs at the end. **Interactive programs** read from the keyboard and write to the
screen, as they are running.

The Problem:

Haskell programs are pure mathematical functions which makes them have no side effects.
However, reading from the keyboard and writing to the screen are side effects, so interactive programs have side effects.

The Solution:

Interactive programs can be written in Haskell by using types to distinguish pure expressions from impure actions that may involve side effects.

```
IO a -- The type of actions that return a value of type a.
IO Char -- The type of action that returns a character
IO () -- The type of purely side effecting actions that return no result value.
          -- () is a value of tuple with no components.
```

Primitive Actions

The standard library provides a number of actions, including the following three primitives:

```
getChar :: IO Char
-- The action getChar reads a character from the keyboard,
-- echoes it to the screen,
-- and returns the character as its result.

putChar :: Char → IO ()
-- The action putChar c writes the character c to the screen,
-- and returns no result value.

return :: a → IO a
-- The action return v simply returns the value v,
-- without performing any interaction.
```

Sequencing Actions

A sequence of actions can be combined as a single composite action using the keyword do.

Example

```
getTwo :: IO (Char,Char)
getTwo = do x ← getChar
            y ← getChar
            return (x,y)
```

In sequence of actions...

- Each action must begin in precisely the same column. That is the **layout rule** applies.
- The value returned by intermediate actions are **discarded** by default, but if required can be named using the ← operator.
- The value returned by the **last** action is the whole value returned by the sequence as a whole.

Other Library Actions

Example 1

Reading a string from the keyboard.

```
getLine :: IO String
getLine = do x ← getChar
             if x == '\n' then return [ ]
             else do xs ← getLine
                     return (x:xs)
```

Example 2

Writing a string to the screen.

```
putStr :: String → IO ()
putStr [] = return ()
putStr (x:xs) = do putChar x
                   putStr xs
```

Example 3

Writing a string and moving to a new line.

```
putStrLn :: String → IO ()
putStrLn xs = do putStr xs
                 putChar '\n
```

Example 4

Action that prompts for a string to be entered and displays its length:

```
strlen :: IO ()
strlen = do putStr "Enter a string: "
            xs ← getLine
            putStr "The string has "
            putStr (show (length xs))
            putStrLn " characters"
```

```
> strlen
Enter a string: hello there
The string has 11 characters
>
```

❗ Evaluating an action executes its side effects, with the final result value being discarded.

Example 5

Doing the swap around.

```
swapAround = do line ← getLine
                if null line then return ()
                else do putStrLn $ reverseWords line

reverseWords :: String → String
reverseWords = unwords . map reverse . words
-- The list ist divided into words using words.
-- The words are reversed using map.
-- The words are again put into a using unwords.

unwords :: [String] → String
-- Haskell function that joins a list of words into a single string with spaces.

words :: String → [String]
-- Haskell function that splits a string into a list of words.
```

❗ `.` is for composing functions.
`$` is for applying a function to an argument with lower precedence to reduce parentheses.

Example 6

Making the hangman game. We code it with implementing a top-down manner.

```
import System.io

hangman :: IO ()
hangman = do putStrLn “Think of a word:”
             word ← sgetLine
             putStrLn “Try to guess it”
             play word
```

```
sgetLine :: IO String
sgetLine = do x ← getCh
              if x == ‘\n’
                 then do putChar x
                         return []
              else do putChar ‘-‘
                      xs ← sgetLine
                      return (x:xs)
```

```
getCh :: IO Char
getCh = do hSetEcho stdin False
           x ← getChar
           hSetEcho stdin True
           return x
```

```
play :: String → IO ()
play word = do putStr “?”
               guess ← getLine
               if guess == word
                  then putStrLn “You got it!”
```

```
                    else
                       do putStrLn (match word guess)
                        play word
```

```
match :: String → String → String
match xs ys = [ if elem x ys then x else '-' | x ← xs ]
```