Programming In Haskell Chapter 5

CS 1JC3

Overview

As we have seen, many functions can naturally be defined in terms of other functions

```haskell
factorial :: Int -> Int
factorial n = product [1..n]
```

Overview

As we have seen, many functions can naturally be defined in terms of other functions

```
factorial :: Int -> Int
factorial n = product [1..n]
```

factorial maps any integer n to the product of the integers between 1 and n

Overview

Expressions are evaluated by a stepwise process of applying functions to their arguments.
For Example:

```
    factorial 4
=
  product [1..4]
=
  product [1,2,3,4]
=
  1*2*3*4
=
  24
```

Recursive Functions

- In Haskell, functions can also be defined in terms of themselves.
- Such functions are called recursive

Recursive Functions

- In Haskell, functions can also be defined in terms of themselves.
- Such functions are called recursive
- For Example:

```
factorial 0       = 1
factorial (n+1) = (n+1) * factorial n
```

Recursive Functions

For Example:

```
    factorial 3
=
    3 * factorial 2
=
    3 * (2 * factorial 1)
=
    3 * (2 * (1 * factorial 0))
=
    3 * (2 * (1 * 1))
=
    3 * (2 * 1)
=
    3 * 2
=
    6
```

Why is Recursion Useful

- Some functions, such as factorial, are simpler to define in terms of other functions
- As we shall see, however, many functions can naturally be defined in terms of themselves

Why is Recursion Useful

- Some functions, such as factorial, are simpler to define in terms of other functions
- As we shall see, however, many functions can naturally be defined in terms of themselves
- Properties of functions defined using recursion can be proved using the powerful mathematical technique of induction

Set Comprehensions

In Mathmatics, the comprehension notation can be used to construct new sets from old sets

$$\{x^2 | x \varepsilon \{1..5\}\}$$

Set Comprehensions

In Mathmatics, the comprehension notation can be used to construct new sets from old sets

$$\{x^2 | x \varepsilon \{1..5\}\}$$

The set $\{1, 4, 9, 16, 25\}$ of all numbers x^2 is an element of the set $\{1..5\}$

List Comprehension

In Haskell, a similar comprehension notation can be used to construct new lists from old lists

```
[x^2 | x <- [1..5]]
```

List Comprehension

In Haskell, a similar comprehension notation can be used to construct new lists from old lists

```haskell
[x^2 | x <- [1..5]]
```

The list $[1, 4, 9, 16, 25]$ of all numbers x^2 such that x is an element of the list [1..5]

Note:

- The expression $x < -[1..5]$ is called a generator, as it states how to generate values for x.
- Comprehensions can have multiple generators, seperated by commas. For example:

```
[(x,y) | x <- [1,2,3], y <-[4,5]]
    [(1,4),(1,5),(2,4),(2,5),(3,4),(3,5)]
```

Note:

- The expression $x < -[1..5]$ is called a generator, as it states how to generate values for x.
- Comprehensions can have multiple generators, seperated by commas. For example:

```
[(x,y) | x <- [1,2,3], y <-[4,5]]
   [(1,4),(1,5),(2,4),(2,5),(3,4),(3,5)]
```

- Try switching the position of the generators around, what happens?

Guards

- List comprehensions can use guards to restrict the values produced by earlier generators.

```
[x | x <- [1..10], even x]
    [2,4,6,8,10]
```

Guards

- List comprehensions can use guards to restrict the values produced by earlier generators.

```
[x | x <- [1..10], even x]
    [2,4,6,8,10]
```

- Using a guard we can define a function that maps a positive integer to its list of factors
(remember to use backwords quotes around mod)

```
factors     :: Int -> [Int]
factors n = [x | x <- [1..n], n `mod` x == 0]
```

Type Declarations

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

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

Note: String is a synonym for the type [Char]

Type Synonym Declarations

- Type declarations can be used to make other types easier to read
- For example, given

```
type Pos = (Int,Int)
```

Type Synonym Declarations

- Type declarations can be used to make other types easier to read
- For example, given

```haskell
type Pos = (Int,Int)
```

- we can define:

```haskell
origin     :: Pos
origin     = (0,0)

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

Type Parameters

- Like function definitions, type declarations can also have parameters
- For example, given

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

Type Parameters

- Like function definitions, type declarations can also have parameters
- For example, given

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

- we can define:

```haskell
mult          :: Pair Int -> Int
mult (m,n)    = m * n

divide        :: Pair Float -> Float
divide (m,n) = m / n

copy          :: a -> Pair a
copy x        = (x,x)
```

Nested Types

- Type declarations can be nested

```
type Pos   = (Int,Int)

type Trans = Pos -> Pos
```

Nested Types

- Type declarations can be nested

```
type Pos   = (Int,Int)

type Trans = Pos -> Pos
```

- However, they cannot be recursive

```
type Tree = (Int,[Tree])    -- Error
```

Data declarations

A completely new type can be defined by specifying its values using a data declaration. For example, pretend Bool is not already a type and theres no such values as True or False. It could be defined as

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

Bool is a new type, with new values False and True

Data declarations

Note:

- The two values False and True are called the constructors for the type Bool
- Type and constructor names must begin with an upper-case letter.
- Data declarations are similar to context free grammers. The former specifies the values of a type, the latter the sentence of a language.

Data declarations

- Say we want to develop a type representing the possible answers to a given yes or no question
- How would we declare such a type?

Data declarations

- Say we want to develop a type representing the possible answers to a given yes or no question
- How would we declare such a type?

```haskell
data Answer = Yes | No | Unknown
```

- How would we write functions containing a list of all possible answers and another function that gives the opposite of a given answer, and just returns unknown for unknown?

Data declarations

- Say we want to develop a type representing the possible answers to a given yes or no question
- How would we declare such a type?

```haskell
data Answer = Yes | No | Unknown
```

- How would we write functions containing a list of all possible answers and another function that gives the opposite of a given answer, and just returns unknown for unknown?

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

flip          :: Answer -> Answer
flip Yes      = No
flip No       = Yes
flip Unknown  = Unknown
```

Data Parameters

- The constructors in a data declaration can also have parameters. For example, given

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

- We can define a function the returns a square, and functions that give the area of a Shape

Data Parameters

- The constructors in a data declaration can also have parameters. For example, given

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

- We can define a function the returns a square, and functions that give the area of a Shape

```
square       :: Float -> Shape
square n     = Rect n n

area         :: Shape -> Float
area (Circle r) = pi * r^2
area (Rect x y) = x * y
```

Data Parameters

Note:

- Shape has values of the form Circle r where r is a Float, and Rect x y where x and y are Floats
- Circle and Rect can be viewed as functions that construct values of type Shape:

```
Circle :: Float -> Shape
Rect   :: Float -> Float -> Shape
```

Data Parameters

- Data declarations themselves can also have parameters
- For example, given

```haskell
data Maybe a = Nothing | Just a
```

- We can define a function safediv that doesn't return an error when you divide by zero.

Data Parameters

- Data declarations themselves can also have parameters
- For example, given

```haskell
data Maybe a = Nothing | Just a
```

- We can define a function safediv that doesn't return an error when you divide by zero.

```haskell
safediv     :: Int -> Int -> Maybe Int
safediv _ 0 = Nothing
safediv m n = Just (m `div` n)
```

Recursive Types

In Haskell, data declarations can be used to declare new types in terms of themselves. This is, types can be recursive. For example

```
data Nat = Zero | Succ Nat
```

Recursive Types

In Haskell, data declarations can be used to declare new types in terms of themselves. This is, types can be recursive. For example

```
data Nat = Zero | Succ Nat
```

Nat is a new type, with constructors

```
Zero :: Nat
```

and

```
Succ :: Nat -> Nat
```

Recursive Types

Note:

- A value of type Nat is either Zero, or of the form Succ n where n :: Nat
- That is, Nat contains the following infinite sequence of values:

```
Zero                0
Succ Zero           1
Succ (Succ Zero)    2
...
...
```

Recursive Types

- We can think of values of type Nat as natural numbers, where Zero represents 0, and Succ represents the successor function 1+
- For example, the value

```
Succ (Succ (Succ Zero))
```

- represents the natural number

Recursive Types

- We can think of values of type Nat as natural numbers, where Zero represents 0, and Succ represents the successor function $1+$
- For example, the value

```
Succ (Succ (Succ Zero))
```

- represents the natural number

```
1 + (1 + (1 + 0)) = 3
```

Using Recursive Types

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+1)    = Succ (int2nat n)
```

Using Recursive Types

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

Using Recursive Types

- 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 can be defined without the need for conversions:

Using Recursive Types

- 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 can be defined without the need for conversions:

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

For example:

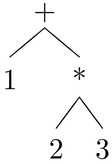
```
    add (Succ (Succ Zero)) (Succ Zero)
  =
    Succ (add (Succ Zero) (Succ Zero))
  =
    Succ (Succ (add Zero (Succ Zero)))
  =
    Succ (Succ (Succ Zero))
```

Note: The recursive definition for add corresponds to the laws $0 + n = n$ and $(1 + m) + n = 1 + (m + n)$

Arithmetic Expressions

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

$$1 + 2 * 3$$



Arithmetic Expressions

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

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

- For example, the expression on the previous slide would be represented as follows:

Arithmetic Expressions

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

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

- For example, the expression on the previous slide would be represented as follows:

```
Add (Val 1) (Mult (Val 2) (Val 3))
```

Arithmetic Expressions

Using recursion, it is now easy to define functions that process expressions.

Arithmetic Expressions

Using recursion, it is now easy to define functions that process expressions.

```
eval               :: Expr -> Int
eval (Val n)       = n
eval (Add x y)     = eval x + eval y
eval (Mult x y)    = eval x * eval y
```

Arithmetic Expressions

Using recursion, it is now easy to define functions that process expressions.

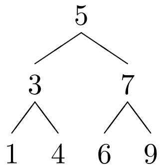
```
eval                :: Expr -> Int
eval (Val n)        = n
eval (Add x y)      = eval x + eval y
eval (Mult x y)     = eval x * eval y
```

Note: The three constructors have types

```
Val  :: Int  -> Expr
Add  :: Expr -> Expr -> Expr
Mult :: Expr -> Expr -> Expr
```

Binary Trees

In computing, it is often useful to store data in a two-way branching structure or binary tree



Binary Tree Data Types

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

Binary Tree Data Types

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

```
data Tree = Leaf Int
          | Node Tree Int Tree
```

- For example, the tree on the previous slide would be represented as follows:

```
Node (Node (Leaf 1) 3 (Leaf 4))
     5
     (Node (Leaf 6) 7 (Leaf 9))
```

Using Binary Trees

We can now define a function that decides if a given integer occurs in a binary tree:

Using Binary Trees

We can now define a function that decides if a given integer occurs in a binary tree:

```
occurs                :: Int -> Tree -> Bool
occurs m (Leaf n) = m == n
occurs m (Node l n r) = m == n
                        || occurs m l
                        || occurs m r
```

Using Binary Trees

Now consider a function flatten that returns the list of all the integers contained in a tree:

Using Binary Trees

Now consider a function flatten that returns the list of all the integers contained in a tree:

```
flatten              :: Tree -> [Int]
flatten (Leaf n) = [n]
flatten (Node l n r) = flatten l
                       ++ [n]
                       ++ flatten r
```

Note: A tree is called a search tree if it flattens to a list that is ordered. Our example tree is a search tree, as it flattens to the ordered list [1,2,3,4,5,6,7,9]

Search Trees

Search trees have the important property that when trying to find a value in a tree we can always decide which of the two sub-trees it may occur in:

```
occurs m (Leaf n)                   = m == n
occurs m (Node l n r) | m == n = True
                      | m <  n = occurs m l
                      | m >  n = occurs m r
```

This new definition is more effiecient, because it only traverses one path down the tree

Exercise 1

Define a recursive function

```
fib :: (Integral a) => a -> a
```

That returns the n^{th} fibonacci number

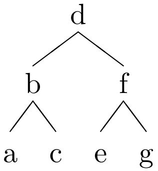
see https://en.wikipedia.org/wiki/Fibonacci_number

Solution 1

```haskell
fib :: (Integral a) => a -> a
fib 0 = 0
fib 1 = 1
fib n = fib (n-1) + fib (n-2)
```

Exercise 2

- Give the data type Tree a parameter so it can represent trees of all types, not just of Int
- Create the following tree using your new data type



- Is this a Search Tree? Use flatten to find out.

Solution 2

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

tree1 :: Tree Char
tree1 = Node (Node (Leaf 'a') 'b' (Leaf 'c'))
             'd'
             (Node (Leaf 'e') 'f' (Leaf 'g'))
```

Note: Yes, this is a search tree

Exercise 3

Using recursion and the function add, define a function that multiplies two Nat.

Remember:

```haskell
data Nat = Zero | Succ Nat
    deriving Show

add :: Nat -> Nat -> Nat
add Zero     n = n
add (Succ m) n = Succ (add m n)
```

Solution 3

```haskell
data Nat = Zero | Succ Nat
    deriving Show

add  :: Nat -> Nat -> Nat
add Zero     n = n
add (Succ m) n = Succ (add m n)

mult :: Nat -> Nat -> Nat
mult Zero     n = Zero
mult (Succ m) n = add n (mult m n)
```

Exercise 4

Create a function that returns a Bool specifying whether or not a given tree is a search tree. (Hint: create a function that decides whether or not a list is sorted).

Solution 4

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

flatten :: Tree a -> [a]
flatten (Leaf n) = [n]
flatten (Node l n r) = flatten l
                       ++ [n]
                       ++ flatten r

sorted :: (Ord a) => [a] -> Bool
sorted xs = and [x <= y | (x,y) <- zip xs (tail xs)]

search :: (Ord a) => Tree a -> Bool
search tree = sorted (flatten tree)
```

Exercise 5

The Prelude defines a data type Maybe that allows you to return Just a value or Nothing. Define a type Possibly that allows you to return Only a value or Notta. Define a function safehead that returns Only the first value or Notta if given an empty list.

Solution 5

```
data Possibly a = Only a | Notta
   deriving Show

safehead  :: [a] -> Possibly a
safehead []      = Notta
safehead (x:xs) = Only x
```

Exercise 6

Consider a graph to be a list of edges. Consider an edge to be a tuple containing two nodes (represented by strings) and an Integer weighting.

Develop appropriate types to represent this, then create a function that takes two nodes and a graph and returns the wieghting of the edge those nodes create (return 0 if the edge does not exist).

Solution 6

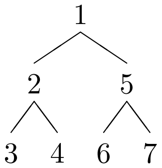
```haskell
type Node  = String
type Edge  = (Node,Node,Int)
type Graph = [Edge]

distance :: Node -> Node -> Graph -> Int
distance _  _  [] = 0
distance n1 n2 ((n3,n4,w):gs)
     | n1 == n3 && n2 == n4 = w
     | n1 == n4 && n2 == n3 = w
     | otherwise            = distance n1 n2 gs
```

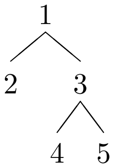
Exercise 7

A binary tree is complete if the two sub trees of every node are of equal size. Define a function that decides if a binary tree is complete

Complete



Not Complete



Solution 7

```
complete :: Tree a -> Bool
complete (Leaf n)     = True
complete (Node l _ r) = (size l == size r)
   where
        size (Leaf _)     = 1
        size (Node l _ r) = size l + size r
```

Exercise 8

- Create a function factor that takes an Int and returns a list of all its factors

```
factors 6 = [2,3]
```

- Create a function lesser that takes a list and returns all elements less than a certain value

```
lesser 6 [2,3,4,5,6,7,8]
   = [2,3,4,5]
```

- Create a function pivot that takes a pivot point, and pushes all the elements in a list less than it to the left, and all the elements greater than it to the right

```
pivot 3 [5,6,4,2,1,3]
   = [2,1,3,5,6,4]
```

Solution 8

```haskell
factors   :: (Integral a) => a -> [a]
factors n = [x | x <- [2..n-1], n `mod` x == 0]

lesser    :: (Ord a) => a -> [a] -> [a]
lesser n xs = [x | x <- xs, x < n]

pivot     :: (Ord a) => a -> [a] -> [a]
pivot n xs = [x | x <- xs, x <= n] ++ [x | x <- xs, x > n]
```

Exercise 9

A triple (x,y,z) of positive integers is called pythagorean if

$$x^2 + y^2 = z^2$$

Using a list comprehension, define a function

```
pyths :: Int -> [(Int,Int,Int)]
```

that takes an Int n, and generates a list of all such triples with x, y and z in the range [1..n]. For example

```
pyths 5
  [(3,4,5),(4,3,5)]
```

Solution 9

```
pyths     :: Int -> [(Int,Int,Int)]
pyths n = [(x,y,z) | x <- [1..n],
                     y <- [1..n],
                     z <- [1..n],
                     x^2 + y^2 == z^2]
```

Exercise 10

A positive integer is called perfect if it equals the sum of all of its factors, excluding the number itself. Using a list comprehension, define a function

```
perfects :: Int -> [Int]
```

that returns the list of all perfect numbers up to a given limit. For example:

```
perfects 500
    [6,28,496]
```

Solution 10

```
perfects n = [p | p <- [1..n], sum (factors p) == p]
    where
      factors x = [f | f <- [1..x-1], x `mod` f == 0]
```