M21274 – MATHFUN
Discrete Mathematics and Functional Programming

Worksheet 3: Pattern Matching and Recursion

Introduction

In this worksheet you will practise using pattern matching and recursion. Remember to use Hoogle to lookup functions and get support from us via the Discord channel.

First save `Week3.hs` into the folder you have for this module. The code reimplements the `||` operator (or). Since this operator comes with Prelude (Haskell's standard library), we need to hide it with:

```
import Prelude hiding ((||))
```

Because we want to use `||` with its normal precedence level, the file also contains the following line (read the comments in the file for more info):

```
infixr 2 ||
```

The `Week3.hs` file also defines functions `fact`, `mult` and `divide` from the lecture. Load this script into GHCi and experiment by evaluating some expressions.

Next, work your way through the worked examples before attempting the the programming exercises at the end of this document. Write your solutions in the same `Week3.hs` file and test them using the provided tables (and your own test data) before asking staff for a sign-off.

Worked example 1

Question: Using pattern matching, define a NOR gate function with the following signature:

```
nor :: Bool -> Bool -> Bool
```

Note that `nor` is a function and not an operator. This is because `nor` has a name (a valid identifier) and is not a symbol. This table shows how `nor` should behave:

Test	Output
`nor False False`	`True`
`nor False True`	`False`
`nor True False`	`False`
`nor True True`	`False`

Solution: Let us start with a four-line, pattern matching implementation of `nor` where each line corresponds to a row in the table. Add the following to the bottom of your `Week3.hs` file:

```
nor :: Bool -> Bool -> Bool
nor False False = True
nor False True = False
nor True False = False
nor True True = False
```

Before we start to improve our solution, let's test it by evaluating expressions. Run the following and see if it returns the output shown in the table:

```
nor True False
```

Run a few other tests on other values.

Notice that the first argument of the first two patterns are both `False`; also, when the second argument of these patterns is `False`, the output is `True` and when the second argument is `True`, the output is `False`. So, when the first argument is `False`, the output is the opposite of the second argument. We can therefore use the `not` function and a variable `x` to combine the first two patterns:

```
nor :: Bool -> Bool -> Bool
nor False x = not x
nor True False = False
nor True True = False
```

There is a similarity in the last two patterns: the first arguments are `True` and the outputs are always `False`. So we can use a **wildcard**, "_" (underscore), to demonstrate that the second input does not matter when the first one is `True`:

```
nor :: Bool -> Bool -> Bool
nor False x = not x
nor True _ = False
```

Worked example 2

Question: The Fibonacci numbers (0, 1, 1, 2, 3, 5, 8 . . .) form a sequence of numbers where every value (except the first two) is the sum of the previous two numbers.

Write a `fibonacci` function that takes an index (`Int`) and returns the Fibonacci number at that index. Assume that the indices are non-negative and test your function with the data in this table:

Test	Output
`fibonacci 0`	0
`fibonacci 1`	1
`fibonacci 6`	8

Solution: Note that the solution will give is actually a naïve, very inefficient function, since it results in many duplicate recursive calls at run-time. We will ignore this issue but will come back to it towards the end of the module.

Let's start with a function that uses guards rather than pattern matching. As suggested by the definition above, the **base cases** of the recursion should be given by the first two rows of the table: "`fibonacci 0 = 0`" and "`fibonacci 1 = 1`". Here we have covered the base cases in our definition (add it to the end of `Week3.hs`):

```haskell
fibonacci :: Int -> Int
fibonacci n
  | n == 0 = 0
  | n == 1 = 1
```

We can cover every other case using the `otherwise` keyword. But what does `fibonacci n` evaluate to for any number `n` where `n` ≥ 2? The discussion above says that `fibonacci n` is the sum of the Fibonacci numbers at the indices `n-1` and `n-2`. To calculate the Fibonacci number for these indices, we can simply call `fibonacci` again. The fact that `fibonacci` calls itself is what makes it a recursive function. Update your code as shown below:

```haskell
fibonacci :: Int -> Int
fibonacci n
  | n == 0 = 0
  | n == 1 = 1
  | otherwise = fibonacci (n - 1) + fibonacci (n - 2)
```

Let us test if `fibonacci` is implemented correctly by evaluating an expression:

```
fibonacci 4
```

Now, comment out your current definition as shown below (using "--" that comment out a single line) and add the following pattern matching definition:

```haskell
fibonacci :: Int -> Int
-- fibonacci n
-- | n == 0 = 0
-- | n == 1 = 1
-- | otherwise = fibonacci (n - 1) + fibonacci (n - 2)
fibonacci 0 = 0
fibonacci 1 = 1
fibonacci n = fibonacci (n - 1) + fibonacci (n - 2)
```

Save this and again test if `fibonacci` is working correctly.

Programming exercises

Pattern matching

1. Write three definitions of the `&&` operator (and) using pattern matching. See the re-implementations of the `||` operator (or) for an example.

   Hide the default definition of `&&` by updating your import statement as shown below:

   ```haskell
   import Prelude hiding ((||), (&&))
   ```

   Also include the fixity declaration for this function by adding the following line which forces complex Boolean expressions to be correctly interpreted:

   ```haskell
   infixr 3 &&
   ```

   Make sure you have three definitions for `||` are similar to the definitions given for `&&`. See the "Pattern matching" lecture notes for more details:

   (i) on slide 2

   (ii) at the top of slide 3

   (iii) at the bottom of slide 3

   After writing each definition, test it using the table below. Once you have tested a definition, comment it out so you can write another one.

Test	Output
`False && False`	`False`
`False && True`	`False`
`True && False`	`False`
`True && True`	`True`

2. Using pattern matching, define the `exOr` function (XOR gate). Verify it using the truth table provided.

```
exOr :: Bool -> Bool -> Bool
```

Test	Output
`exOr False False`	`False`
`exOr False True`	`True`
`exOr True False`	`True`
`exOr True True`	`False`

Note: `exOr` does not come with Prelude, so you do not need to hide anything. Also note that `exOr` is a function (not an operator), so you do not need to include a fixity declaration.

3. Using pattern matching, define an `ifThenElse` function that takes three arguments. If the first argument evaluates to `True`, `ifThenElse` should return the second argument, otherwise it should return the third (see the table).

```
ifThenElse :: Bool -> Int -> Int -> Int
```

Test	Output
`ifThenElse True 0 1`	`0`
``ifThenElse (5 `mod` 2 == 0) 1 2``	`2`

4. Use pattern matching to write a function `daysInMonth` that takes an `Int` (assumed to be between 1 and 12) and returns the number of days in the corresponding month. Your goal is to make your solution as short as possible, but do not use concepts that are not covered yet (e.g., lists).

```
daysInMonth :: Int -> Int
```

Test	Output
`daysInMonth 1`	`31`
`daysInMonth 2`	`28`
`daysInMonth 4`	`30`

Once you have written and tested your `daysInMonth` function, write a new (simpler) version of `validDate` from the previous worksheet (guards or patterns should not be needed).

Recursion

5. Write a recursive function `sumNumbers` that takes a non-negative number `n` (as `Int`) and returns the sum of all integers from 1 to `n`.

```
sumNumbers :: Int -> Int
```

Test	Output
`sumNumbers 0`	0
`sumNumbers 3`	6
`sumNumbers 50`	1275

6. Write a recursive function `sumSquares` that takes a number `n` and returns the sum of the squared numbers from 1 to `n` (i.e., $1^2 + 2^2 + \ldots + n^2$).

```
sumSquares :: Int -> Int
```

Test	Output
`sumSquares 3`	14
`sumSquares 20`	2870

7. Using multiplication, write a recursive `power` function that raises the first argument to the power of the second. You cannot use the power operator (`^`) for your solution.

```
power :: Int -> Int -> Int
```

Test	Output
`power 5 2`	25
`power 2 5`	32

Hint: Think first about the base case. What if the power is 0?

8. Write a recursive function `sumFromTo` that given two non-negative `Int`s, returns the sum of all numbers between them. The second argument should be no smaller than the first, otherwise `sumFromTo` should return 0 (see the table).

```
sumFromTo :: Int -> Int -> Int
```

Test	Output
`sumFromTo 5 5`	5
`sumFromTo 5 6`	11
`sumFromTo 7 5`	0

9. The greatest common divisor (GCD) of two non-negative can be defined as:

   - **If they are equal**: their common value
   - **If they are not equal**: the GCD of the following two values:
     - the positive difference between the two numbers
     - the smaller number

   Write a `gcd` function that takes two non-negative integers and returns their GCD; your function should **use the above definition of GCD.**

```
gcd :: Int -> Int -> Int
```

Test	Output
`gcd 8 12`	4
`gcd 120 75`	15

   Note: there is a `gcd` function defined in the Prelude. Hide this default definition by updating your import statement as shown below:

```
import Prelude hiding ((||), (&&), gcd)
```

10. The integer square root of a positive integer n is the largest integer whose square is less than or equal to n.

    Add the following definition to your `Week3.hs` file and save it:

```
intSquareRoot :: Int -> Int
intSquareRoot n = findRoot n n
```

    Now define a recursive function `findRoot` that given two arguments `n` and `s`, returns the square root of `n` starting from the guess `s`.

```
findRoot :: Int -> Int -> Int
```

    Once you are done, test your `intSquareRoot` using the following table:

Test	Output
`intSquareRoot 9`	3
`intSquareRoot 7`	2
`intSquareRoot 125`	11

    Note: Your solution cannot use power (`power`, `^`) or `sqrt`.

11. If many of your answers to questions 5 to 10 involve guards, write versions of three of them that use pattern matching and no guards. Similarly, if many already use pattern matching, write versions of two of them that use guards.

    Note: Comment out one version before you write a new one below it.