FUNCTIONAL PROGRAMMING MT2020

Sheet 1

1.1 Recall that function application binds more tightly than any other operators. Put in all the parentheses implicit in the expressions

1. `a plus f x + x times y * z`
2. `3 4 + 5 + 6`
3. `2^2^2^2`

It may help to know that `2^2^2^2` evaluates to 65536, as you can check with GHCi.

1.2 Prove that function composition is associative. (Remember that functions are equal precisely when they return the same result whenever applied to the same argument.)

1.3 Suppose that the ++ operator is defined by

```
as ++ bs = concat [as, bs]
```

(This is not the standard definition, but it defines the same function.)

Is this operator associative? Is it commutative? Does it have a unit (identity element)? Does it have a zero?

(e is a unit of $\oplus$ if $e \oplus x = x = x \oplus e$. z is a zero of $\oplus$ if $z \oplus x = z = x \oplus z$.)

You might be able to tell these things without yet being able to prove that you are right.

1.4 Suppose that

```
double :: Integer -> Integer
double x = 2 * x
```

is the function that doubles an integer. What are the values of

```
map double [3,7,4,2]
map (double.double) [3,7,4,2]
map double []
```

You might check your answers on an interpreter.

Suppose that

```
sum :: [ Integer ] -> Integer
```

is a function that adds up all of the elements of its argument. (There is such a standard function.) Which of the following are true, and why?

$$\begin{aligned} sum . map\, double &= double . sum \\ sum . map\, sum &= sum . concat \\ sum . sort &= sum \end{aligned}$$

2.1 Use the standard function $product :: [Int] \to Int$ to define *factorial*, a function that calculates the factorial of its argument.

Hence define a function *choose* that makes `n 'choose' r` be the number of ways of choosing r things from n.

Write a function *check* that when applied to n returns a *Bool* indicating whether $\sum_{r=0}^{n} \binom{n}{r} = 2^n$.

2.2 The standard function `not :: Bool -> Bool` maps each Boolean value to the other one. Without using the *not* function itself, write four definitions of functions equal to `not` using essentially different syntactic forms.

2.3 There are two things (the values `True` and `False`) of type `Bool`. How many different things are there with type

1. `Bool -> Bool`
2. `Bool -> Bool -> Bool`
3. `Bool -> Bool -> Bool -> Bool`
4. `(Bool, Bool)`
5. `(Bool, Bool) -> Bool`
6. `(Bool, Bool, Bool)`
7. `(Bool, Bool, Bool) -> Bool`
8. `(Bool -> Bool) -> Bool`
9. `(Bool -> Bool -> Bool) -> Bool`
10. `((Bool -> Bool) -> Bool) -> Bool`

Geraint Jones, 2020