Introduction to Programming and its Mathematical Foundations

Lesson 9: Functions with multiple arguments

Note: File `script09.gs`, to be loaded into the interpreter, contains the functions being discussed below.

Summary: How to provide n> 1 arguments to a function, one at a time

Terminology: When the number of arguments of a function is `1, 2, 3, ..., n`, the function is called a unary, binary, ternary, `...`, `n`-ary function. Examples: the area of a rectangle, with its length and breadth as arguments, or the volume of a cylinder, given its radius and height, are binary functions.

How to pass multiple arguments to a function? This question arises because in the lambda calculus, the mathematics underlying functional programming, all functions are supposed to be unary.

In the Gofer expression `f.x`, the dot "`.`" is a binary operator whose left operand `f` is a function and right operand `x` is the argument to which it is applied. (Therefore it appears that `f` is a unary function.) In Gofer notation, we say in general that if `f: a->b`, then `x: a` and `f.x: b`. In other words, the type of `x` has to be the same as the domain type of `f`, otherwise there is a type mismatch. Likewise, the resulting type of `f.x` is the range type of `f`.

One way to pass multiple arguments to a unary function is to put the arguments into a tuple by placing parentheses around them, and make this tuple the single argument to the function, like `x` in `f.x` above, e.g.,

```
area: Num.a => (a, a) -> a
-- area of a rectangle
-- a can be any numerical type
area . (len, br) = len * br
```

```
? area . (2, 3)
6 : Int
? area . (2.5, 2.0)
5.0 : Float
```

€: Write a binary function that computes the volume of a cylinder: `vol.(rad, ht) =pi*rad^2*ht` .

Q: What is the type of `vol`? Hint: Since `pi: Float`, the radius and height of the cylinder also have to be `Float`. (Recall: In Gofer you can perform arithmetic operations on `Int`, or on `Float`, but not on a combination.)

Rem: You can imagine the general case, when 3 or more arguments can be enclosed into a single tuple.

Using type `Num.a => (a,a)-> a` for function `area` can lead to a message (which I won't try to explain) like:

```
ERROR "script08.gs" (line 5): Unresolved top-level overloading
*** Binding              : area_c
*** Inferred type        : _5 -> _5 -> _5
*** Outstanding context : Num._5
```

So I modified `area` to `area1` and `area2`, where the sides of the rectangle are `Int` and `Float`, respectively.

```
area1: (Int, Int) -> Int
area1 . (len , br) = len * br
? area1 . (5, 10)
50 : Int
```

```
? area1 . (5.0, 10.0)
ERROR: Type error in application
*** expression : area1.(5.0,10.0)
*** term             : (5.0,10.0)
*** type             : (Float,Float)
*** does not match : (Int,Int)
```

```
area2: (Float, Float) -> Float
area2 . (len , br) = len * br
```

```
? area2 . (5.0, 10.0)
50.0 : Float
```

The `curry` and `uncurry` operators for binary functions

We want to be able to supply arguments to a function without having to consolidate them into a tuple.
The `curry` operator does that for binary functions, and the `uncurry` operation is the inverse of `curry`.
Once we understand `curry`, its generalization can be used for arbitrary number of arguments.
(In comparison to `curry`, the `uncurry` operator is not so important.)

Given functions like `area1` and `area2` above, note how the `curry` and `uncurry` operators are applied.

```
area1c = curry.area1            -- The "curried" version of area1
area1u = uncurry . area1c       -- The curried version is "uncurried" in turn

area2c = curry.area2            -- The "curried" version of area2
area2u = uncurry . area2c       -- The curried version is "uncurried"
```

The signature of the original `area1` is given below. Based upon it, the signatures of `area1c` (the curried `area1`) and `area1u` (the uncurried `area1c`) are as follows:

```
? :t area1
area1 : (Int,Int) -> Int
? :t area1c
area1c : Int -> Int -> Int
? area1c     -- "curried" area1
curry.area1 : Int -> Int -> Int
```

```
? :t area1u
area1u : (Int,Int) -> Int
? area1u -- "uncurried" area1c
uncurry.area1c : (Int,Int) -> Int
? area1u . (8, 4)
32 : Int
```

€: Functions `area2`, `area2c` and `area2u` are defined in file `script09.gs`. Find their types.

We can easily make sense of the types of functions `area1` and `area1u`, as they both have the type `(Int, Int)->Int`. That is, if their argument is a pair of integers, they will return an integer result.

€: But what to make of the type of the curried function `area1c`, namely, `Int -> Int -> Int`?
Note that 1. there are two arrows in the type signature, rather than the usual one arrow, and further, that 2. are no parentheses, i.e., a pair of integers are not to be provided.

Experiment 1: Give a single integer as an argument to `area1c`, and observe what happens.

```
? area1c . 10
curry.area1.10 : Int -> Int
```

Outcome: If `area1c` is given a single integer as an argument, it returns a function of type `Int -> Int`.

Experiment 2: Give function `area1c.10` another integer as an argument and see what happens.

```
? area1c . 10 . 20
200 : Int
```

Outcome: `area1c` computes the area of the rectangle with sides `10` and `20`, just as the original `area1` does.

The difference is that function `area1` has to be given the two arguments together, in the pair `(10, 20)` but the curried function `area1c` can take the arguments one at a time. Phrasing it in different words, we can delay giving the second argument of `area1c`.

€: In file `script08.gs`, the function `len10` has been defined as `len10 = area1c.10`. Find the type of `len10`, give it an appropriate argument, and find out what the resulting value is.

```
? :t len10
len10 : Int -> Int
? len10
curry.area1.10 : Int -> Int
```

```
? len10 . 20   -- breadth 20
200 : Int
? len10 . 56
560 : Int
```

Rem: In an expression `f.x.y`, the dot associates to the left, i.e., `f.x.y = (f.x).y`, i.e., the dot on the left is applied first—resulting in a unary function—and only then the dot on the right is applied next. While `area1` is of type `Int->Int->Int`, the most general type of `f` is `a->b->c`, with `x:a` and `y:b` .

You would be able to argue to yourself that <u>if dot associates to the left, then arrow must associate to the right.</u>

Just in case you cannot convince yourself about the arrow associating to the right, define two functions called `left1` and `right1`. These functions can do something trivial, like returning the sum of the two parameters.

Place parentheses in the type definition of function `right` to force the arrow to associate to the right, thus: `right: Int -> (Int -> Int)`. Similarly, place parentheses in function `left` to force the arrow to associate to the left, thus `left: (Int -> Int) -> Int`. We want to see which function is type correct.

```
left: (Int -> Int) -> Int
left . x . y = x+y
```

```
right: Int -> (Int -> Int)
right . x . y = x+y
```

We cannot even load the script into the interpreter, as there is a type error in the definition of function `left`.

```
ERROR "script08.gs" (line 14): Type error in function binding
*** term           : left
*** type           : a -> a -> a
*** does not match : (Int -> Int) -> Int
```

In other words, forcing left associativity of the arrow in function `left` leads to type incorrectness. SoI commented out the definition of `left1` so that the script can be loaded, and asked for the type of `right`.

```
? :t right
right : Int->Int->Int
? right . 10 . 20
30 : Int
```

Observe: Something interesting has happened. In the type definition for function `right`, we had put parentheses around the right arrow, thus `right1: Int -> (Int -> Int)`. But when we asked for the type of `right` the interpreter removed the parentheses. This has happened because in the interpreter's view the parentheses are superfluous.

Conclusion: Forcing left associativity of the arrow by putting parentheses in `left` caused a type error, but forcing right associativity of the arrow in `right` caused the interpreter to ignore the parentheses. We have discovered an important fact: <u>When dot associates to the left, arrow associates to the right</u>.

<u>Problem</u> (Example of a binary function in uncurried and curried form): We want to raise an `Int` or a `Float` to an integer power using the exponentiation operator "^". Below, and in file script09.gs, the curried form of the function is called `power`, whereas the uncurried form carries an extra letter as a suffix, thus `poweru`.

```
power: Num.a => a -> Int -> a
power . base . n = base ^ n
```

```
poweru: Num.a => (a, Int) -> a
poweru(base, n) = base ^ n
```

```
? :t power
power : Num.a => a -> Int -> a
```

```
? :t poweru
poweru : Num.a => (a,Int) -> a
```

```
? :t power . 2
power.2 : Int -> Int – type variable a takes value Int
? :t power . 2.0
power.2.0 : Int -> Float -- a takes value Float
? power . 2 . 5
32 : Int
? power . 2.0 . 5
32.0 : Float
```

Background: The builtin function `exp` is of type `Float->Float`. The value of `exp.x` is `e` (the base of natural logarithms, value `2.71828`) raised to power `x`.

Functions `poweru` and `power` are generalizations of `exp`, in the sense that the base is not `e` but any `Int` or `Float` value, and and they are variations of `exp`, in the sense that the base is raised to an integer power. The values returned by `poweru` or `powerc` are of type `Num.a => a`, i.e., `Int` or `Float`, depending on whether `base` is `Int` or `Float`.

€: Verify that function `expo`, defined in script08.gs, produces the same values as the builtin function `exp`.

Problem set (under construction):

Define `mult.m.n`, the multiplication of two integers `m` and `n`, without using the multiplication operator. Hint: use a recursive function to implement multiplication as repeated addition.

♋ ♦ ♋