Introduction to functional programming, Lesson 3:
DATA AND DATATYPES IN GOFER: COMPOUND TYPE FUNCTION

Compared to tuples and lists, the third compound data type, function, initially appears to be counter-intuitive as a data type, i.e., as a type of data. But functions are so important, so central, that they give rise to the term "functional programing," the style of programming and problem-solving that we are learning. First we learn how to use already built-in functions, and at the end we note how the user can define quite-simple functions.

Lesson 3 Summary: What is a function, type of a function vs value of a function, how to denote the type of a function. Applying a function to its argument (i.e., evaluating a function for a given argument), implicit vs explicit (i.e., invisible vs visible) function application operator. Built-in functions for tuples and lists. What is a first class type, functions as first class types. Standard and non-standard notation in Gofer.

Preliminary: *What is a function?*

The basic concepts are explained from first principles, to clarify them for learners who do not know them.

Definition: A *function* `f: A -> B` is a mapping from set A to set B such that for each element `x` in `A`, there is a unique element `y` in `B` to which `x` is mapped (i.e., with which `x` is associated).

Example: `square: Int->Int` , `square(x) = x*x`. (We will not be using this notation, however.)

(~~R~~: The term "function" was introduced by Leibniz (1646-1716), and the idea was further formalized by Euler (1707-1783), who introduced the commonly used notation for a function, `y = f(x).`)

Notation is important, e.g., the differential calculus was independently invented, or discovered, about 350 years ago, a few years before 1700 CE, by Leibniz and Newton (1643-1727). They used different notations to express the same basic concepts, and as a result their work proceeded in quite different ways.

Our Gofer interpreter uses a non-standard notation, to make learning easy for beginners.

In mathematics notation, we write, e.g., `f: A -> B`, and read it as "a function named `f`, from domain `A` to range `B`." In functional programming, we think in terms of *domain type* and *range type*.

It is postulated (i.e., assumed for purposes of reasoning) that functions are a compound data type, and that their type is fully determined by their domain type and range type. Note how a function's type is written.

Given that `cos` or cosine is a built-in function of Gofer, we can query about its type with the command `:t`

```
? :t cos
cos : Float -> Float
```

Read the words `cos : Float -> Float` as "the function `cos` is of type `Float` to `Float`."

If a function `f` is from any type to any other type, we use type variables `a` and `b`, so write `f: a -> b`

The mathematical concept of a function emerged in the 17th century in connection with the development of the differential calculus; e.g., the slope of a graph at a point was regarded as a function of the x-coordinate of the point. Mathematicians of the 18th century typically regarded a function as being defined by an analytic expression ("formula"). Theoretical developments in the 19th century led to the much more general modern concept of a function as a single-valued mapping from one set to another.

(Remark: It appears that Indian mathematicians had discovered these basic concepts much earlier—but that story is for another day. Just now our focus is on the conventional approach and notation.)

~~R~~: When I enter `2+2` into the interpreter, its response is to give me its value, `4`. If the expression is the list `[0 ... 10]`, the interpreter prints all the list elements. If the expression is the infinite list `[0 ...]` the interpreter attempts to print all the list elements. But even in the best case, it would not be possible to

similarly give a complete description of a function if the domain type is `Float`. The convention with Gofer is that when asked for the value of a function, the interpreter merely prints out its name and type.

```
? cos
cos : Float -> Float
```

Repeat: The type of a function is fully determined by its domain type and range type. Thus all functions of type `Float -> Float` (e.g., `sin`, `cos` and `tan`), and in general of type `a -> b`, are of the same type.

Question: How to evaluate a function for a given argument value, e.g., `cos` at π = `3.14159`?
It is important to take note of a distinguishing feature of the notation used in our version of Gofer.

```
? pi
3.14159 : Float
? cos pi
ERROR: Juxtaposition has no meaning. Use .
? cos(pi)
ERROR: Juxtaposition has no meaning. Use .
```

Juxtaposition means being side by side. The function `cos` and its argument `pi` are next to each other. In our version of Gofer, that is not permitted. In the error message above, the first "`.`" is a full stop, while the second "`.`", as seen in "`Use .`" stands for something else: *the function application operator*.

Remark: Sometimes we will pretend that Gofer doesn't like tokens to be juxtaposed, i.e, to be side by side, or next to each other. (By *tokens* I mean function names, argument names or values, etc.) Instead, Gofer wants these things to be nicely separated from one another by the "`.`" symbol. For example, if I want to select the first 10 elements out of a longer list like `[0 ... 100]` (it could even be an infinite list), then I will write `take . 10 . [0 ... 100]`, and if I want to drop the first 10 elements and take the next 5, then I will write `take . 5 . (drop . 10 . [0 ... 100])`. The dot symbol, the explicit separator between the various tokens, does not change the apparent meaning. We can try this out in the interpreter.

€: Enter or cut-paste the two expressions above into the Gofer interpreter, and see what their values are.

℟: The "`.`" symbol is not just a convenient separator, but a fundamental operator in functional programming. I cannot overemphasise the importance of this operator—it is on the same level as the equality symbol "=".

Nomenclature: In functional programming, a function is "*applied to its argument*," e.g., `cos` is applied to `pi`.

In conventional functional programming, `cos pi` or `cos(pi)` is syntactically correct, but not in our version of Gofer. We have an *explicit function application operator*, denoted by the symbol "`.`" or dot. Conventionally, the application operator is implicit or invisible, but in our notation it is visible.

℟: An important aspect of our learning is that when we read functional programming books and programs, the invisible application operator has to "leap out of the page" and become visible to you, the reader. Your seniors will tell you that yes, this has actually happened to them—the invisible operator becomes visible.

Below, there are some examples of builtin functions and their applications to appropriate arguments.

```
? not
not : Bool -> Bool
? (not . True, not . False)
(False,True) : (Bool,Bool)
? even
even : Int -> Bool
? (even . 3, even . 4)
(False,True) : (Bool,Bool)
```

Recall: We can explicitly query the type of an expression, using the `:type` interpreter command (short form `:t`). Then, the value of the expression is not printed, but just the expression and its type.

```
? :t odd
odd : Int -> Bool
? (odd . 3, odd . 4)
(True,False) : (Bool,Bool)
? :t (odd . 3, odd . 4)
(odd.3,odd.4) : (Bool,Bool)
```

One of the most commonly used built-in function is the `length` function. Given a list of any component type, the `length` function gives its length. (Recall that the type variable `a` can match any valid Gofer type, so `[a]` is a list with any component type.)

```
? :t length -- length of any list
length : [a] -> Int
```

```
? length . ['a' ... 'z']
26 : Int
? length . [0 ... 100]
101 : Int
? (length . ['a' ... 'z'], length . [0 ... 100])
(26,101) : (Int,Int)
length . [0, 2 ... 1000]
501 : Int
? ['a', 'c' ... 'z']
acegikmoqsuwy
? length . ['a', 'c' ... 'z']
13 : Int
```

~~R~~: When we enter builtin function names, sometimes we see their internal representations.

```
? length
foldl'.v64.0 : [a] -> Int
? odd
even ; not : Int -> Bool
```

~~R~~: You know that odd means "not even", but you may not understand the use above of semi-colon `;` , and likewise, of `foldl'`. If it is function type you are interested in, then a query with `:t` suffices.

```
? :t length
length : [a] -> Int
? :t odd
odd : Int -> Bool
```

Fact: A function, since it is a valid Gofer type, can be the component type of a list or a tuple, e.g., `

```
? [sin, cos, tan]
[sin, cos, tan] : [Float -> Float]
? ((1, even), (1.0, not))
((1,even),(1.0,not)) : ((Int, Int -> Bool),(Float, Bool -> Bool))
```

The Gofer Prelude contains all the builtin functions

The standard prelude `pustd.pre` that we use contains the definitions of all builtin functions. These functions are loaded into the Gofer interpreter whenever it starts executing. There are Gofer preludes other than the standard one, e.g., the simple prelude `pustd.simple` .

Builtin functions for compound data type tuple

There are builtin functions to select individual components of pairs and triples:

```
? fst
fst : (a,b) -> a
? snd
snd : (a,b) -> b
? fst . (1, 10.0)
1 : Int
? snd . (1, 10.0)
10.0 : Float
? (snd.(1.0, 10), fst.(1.0, 10))
(10,1.0) : (Int,Float)
```

```
? fst3
fst3 : (a,b,c) -> a
? snd3
snd3 : (a,b,c) -> b
? thd3
thd3 : (a,b,c) -> c
? fst3 . (1, 10.0, 'A')
1 : Int
? snd3 . (1, 10.0, 'A')
10.0 : Float
? thd3 . (1, 10.0, 'A')
'A' : Char
```

Builtin functions for compound data type list

Functions `head` and `tail`, along with function length, are the most relevant functions for lists. Function `head` gives the first element of a non-empty list, and function `tail` gives the remaining elements.

```
? head
head : [a] -> a
? head . [0...5]
0 : Int
? tail . [0...5]
[1, 2, 3, 4, 5] : [Int]
? head . (tail . [0...5])
1 : Int
```

```
? tail . (tail . [0...5])
[2, 3, 4, 5] : [Int]
? tail . (tail . (tail . [0...5]))
[3, 4, 5] : [Int]
? head . [0]
0 : Int
? tail . [0]
[] : [Int]
```

~~R~~: Asking for the `head` or `tail` of a non-empty list results in an error.

```
? head . []
Program error: {head.[]}
```

```
? tail . []
Program error: {tail.[]}
```

~~R~~: Comparable to the `head` and `tail` functions are the `init` and `last` functions:

```
? init
init : [a] -> [a]
? last
last : [a] -> a
? init . [0 ... 5]
[0, 1, 2, 3, 4] : [Int]
? last . [0 ... 5]
5 : Int
? init . (init . [0 ... 5])
[0, 1, 2, 3] : [Int]
? last . (init . [0 ... 5])
4 : Int
```

Functions as a first class type

Definition: A *first class type* is one that can be used anywhere that other types are used, especially, if the type can be passed as an argument to a function or returned as the result of applying a function to its argument, i.e., if the type can be the domain type or range type of a function.

℞: This terminology is derived directly from the notion of a first class citizen, i.e., a citizen that has all the rights, privileges and duties that other citizens have.

℞: Functions are a first class type. In general, the type of a function is `a -> b` (where `a` and `b` are type variables). Now the domain type and range type can themselves be function types, i.e., `a` and `b` can be bound to function types, e.g., `(Float -> Float) -> (Int->Bool)` can be a valid function type. (What such a function achieves, is not the issue just now. We will see many such functions in later Lessons.)

FUNCTION DEFINITION

In this Lesson, we spent much time on function application, i.e., function evaluation. However, we have not spent any time on the dual operation, namely, function definition. Here, we make a brief mention only, because we need more preparation on the student's part before we take it up in detail.

A function definition consists of two parts: the type definition (or signature) and the function body.

Function definitions are made within a Gofer script (file extension `.gs`), and the script is saved for later use. When the functions are to be used (applied to arguments, etc.), the script is loaded into the Gofer interpreter using the `:load` or `:l` command.

```
succ: Int-> Int       dbl:  Int-> Int       sq: Int-> Int
succ . n = n+1        dbl . n = n+n         sq . n = n*n
```

℞: Once the functions are loaded into the interpreter, they can be used just like built-in functions.

Terminology: Functions are applied to their arguments. To distinguish between function definitions and function application, the usage is "parameter of the definition" and "argument of the call." Thus, in the three definitions, `n` is the parameter, while in the three sets of applications, the integers are the arguments. The first and foremost step of a function application is parameter-argument matching.

Caution (repeat): When you store a script with a new or changed name, say `newscr.gs`, take two precautions under MS-Windows: (1) put double quotes around the name, like "`newscr.gs`", otherwise `newscr.gs` will be the name and `.txt`, the file extension, and so the full file name will be `newscr.gs.txt`, and (2) store the file in the same directory as the Gofer interpreter. If you don't take these two precautions, you will get an error message like `ERROR "newscr.gs": Unable to open file`.

℞: Function composition means, apply functions, one after the other, to an original argument. There is a function composition operator in Gofer, symbol semicolon or "`;`", see below on this page.

Interpreter interactions after the functions `succ`, `dbl` and `sq` are loaded into it:

```
? succ.10
11 : Int
? dbl.10
20 : Int
? sq.10
100 : Int
? dbl . (sq . 10)
200 : Int
? succ. (dbl . (sq.10))
201 : Int
```

Note the the function composition operator "`;`"

`(f ; g).a` means

First apply `f` to `a`, thus `f.a`

Then apply `g` to the result, thus `g.(f.a)`

```
? (sq ; dbl).10
200 : Int
? (sq ; dbl ; succ) . 10
201 : Int
```

Date and time: 03-01-2020 11:10, modified 12-06-2020, 15-07-2020 and 4-1-2021

Introduction to functional programming, Lesson 3, Part 2:
DATA AND DATATYPES IN GOFER, STANDARD AND NON-STANDARD NOTATION

The programming style known as functional programming is based on the lambda calculus, a formal system of rules for defining and evaluating functions. This calculus has two operators, namely, *abstraction* (for defining functions) and *application* (for evaluating functions, given their arguments). We has seen examples of function application above. We will consider function abstraction in later lessons.

Functional programming almost always employs a typed lambda calculus, i.e., each value has type. A consequence is that for type correctness, when a function is applied to an argument, the argument type has to match the function's domain type.

The Gofer interpreter evaluates well-formed expressions (lambda expressions, to be exact). We can override this functionality by issuing "interpreter commands." This is done by entering a line starting with a colon "`:`" (that treats the subsequent string not as an expression but an interpreter command). Following the colon "`:`" is the command name, e.g., `load`, `reload`, `type`, `set`. For succinctness, it suffices to enter the first character of the command, thus `:l`, `:r`, `:t`, `:s`. Above, we saw the use of `:type` and `:set -t`. Here we investigate the use of :set and its arguments t and S (uppercase-s).

Note: Every value has a type. Type information, given by the interpreter after a value is displayed, is optional. The command :set –t causes the type information to be not displayed, while :set +t (the default setting) causes type information to be displayed. See the interpreter responses below.

```
? 2+2
4 : Int
? [0...4]
[0, 1, 2, 3, 4] : [Int]
```

```
? :set -t
? 2+2
4
? [0...4]
[0, 1, 2, 3, 4]
```

```
? :s +t
? 2+2
4 : Int
? [0...4]
[0, 1, 2, 3, 4] : [Int]
```

~~R~~: With our settings of the Gofer interpreter, by default the type of an expression is displayed along with its value. In some circumstances you may not want to see the type. In that case, use `:s -t`.

Significance of the explicit function application operator

As mentioned above, function application, along with function abstraction (i.e., definition) are the basic operations of the lambda calculus. Indeed, the lambda calculus gets its name because the Greek letter lambda λ was used by the discoverers/inventors of this calculus to denote function abstraction.
But function application has no visible operator, so if we juxtapose two syntactically correct expressions, i.e., write them next to each other, say `expr1 expr2`, the interpretation in the calculus is that expr1 is a function, expr2 is its argument, and `expr1` is applied to `expr2`.

In hindsight, it does not appear to be a good idea that blank space (or several spaces) denotes function application. Our method is to write `expr1 . expr2`. For beginners especially, the presence of an

explicit, i.e., visible, operator is helpful—seeing the dot "." indicates which function-valued expression is being applied to which argument. (For starters, it is essential that the argument be type-correct, i.e., its type should match the domain type of the function.) There are some advantages of an invisible operator, but we are not in a position just now to understand and appreciate them. The use of the dot operator for function application is non-standard notation (explained below in Part 2 of this Lesson.)

Note that functions are "first class" types in functional programming, by which we mean that that they can be used wherever other data types, simple or compound, can be used. (First class types in functional programming are like first class citizens in civic society: a first class citizen has all the rights and privileges that other citizens have. Not all programming languages have functions as first class types, e.g., C language does not.)

Standard and non-standard notation with Haskell-style programming:

We have already seen above the function application operator, dot or ".". It is important to note that this is nonstandard notation in Haskell-like languages. There are some other changes in notation also:.

In conventional functional programming, as in the lambda calculus, the application operator is implicit (or invisible). Repeat: if there are two well-formed lambda expressions `A` and `B` placed side by side, it is assumed that `A` is a function-valued expression, expression `B` is its type-correct argument, and the task of the interpreter is to apply `A` to `B` and display the result. This is the standard notation for Gofer too.

The command `:set +S` or `:s +S` causes non-standard notation to be used for function application, while `:s -S` causes standard notation. Below we mention three important differences between the two.

1. In the interpreter response, the symbol `":"` stands for "is of type" in the non-standard notation, whereas in standard notation, the interpreter uses the double colon `"::"` symbol. See below.

2. In the nonstandard notation, three dots `...` indicate a range of values in a list, e.g., `[0...4]` but in standard notation, only two dots are used, thus `[0 .. 4]`.

3. As already stated, we use the explicit function application operator `"."`, whereas the standard notation has an implicit or invisible operator (i.e., blank space). Accommodating both notations, our version of the Gofer interpreter allows both the explicit and the implicit application operators.

Non-standard notation (single colon indicates the type, three dots indicate a sequence of list elements):

```
? 2+2
4 : Int
? [0 ... 4]
[0, 1, 2, 3, 4] : [Int]
? cos
cos : Float -> Float
```

Standard notation (double colon indicates the type, two dots indicate a sequence of list elements):

```
? :set -S
? 2+2
4 :: Int
? [0 ... 4]
ERROR: Undefined variable "..."
? [0 .. 4] - the interpreter requires two dots rather than three
[0, 1, 2, 3, 4] :: [Int]
? cos
cos :: Float -> Float
```

```
? cos pi — juxtaposition works for function application
-1.0 :: Float
? cos . pi  — the explicit application operator also works in our Gofer
-1.0 :: Float
```

In standard notation, explicit function application operators cannot be used.

```
? :set +S
? cos . pi  -- using the explicit function application operator
-1.0 : Float
? cos pi    -- implicit operator, juxtaposition in other words
ERROR: Juxtaposition has no meaning. Use .
? cos . pi
-1.0 : Float
? :s -S
? cos pi
-1.0 :: Float
```

♋ ♦ ♋

These exercises are to be moved to a later lesson, after function definitions have been covered.

€: Write functions that will extract the four individual components of a 4-tuple.

€: Write three functions, `sel12`, `sel23` and `sel13` which will respectively select the first two, the second two, and the first-and-last components of a triple.

€: Write two functions, `rev2` and `rev3`, that will reverse the order of components in a pair and a triple.

Date and time 03-01-2020 18:55, modified 12-06-2020 08:15 and 15-07-2020 18:00