Introduction to functional programming, Lesson 2:
DATA AND DATATYPES IN GOFER, COMPOUND TYPES TUPLE AND LIST

In Lesson 1, the very first introduction to functional programming with Gofer, we were introduced to four simple types: `Int`, `Float`, `Char` and `Bool`. Simple types are atomic, i.e., they cannot be broken down into constituent data or types.

In Lesson 2 we consider compound data and their types. A compound datatype is made out of constituent data (*constituent* means "part of a whole") so it may be likened to a molecule that contains several atoms.

There are three compound types, denoted by the use of 1. parentheses `(` and `)`, 2. brackets `[` and `]`, and 3. arrow `->`. These datatypes are called tuple, list and function, respectively. As of now, we have no information about the internal structure of these types, but we will be able to recognize values as being of these types because of their distinguishing symbols `(` and `)`, `[` and `]` and the arrow `->`. The present Lesson, Lesson 2, is on tuples and lists, while the next Lesson will address functions.

Summary of Lesson 2: Internal structure of the datatypes tuple and list; type errors in list (not possible in tuples); empty list and empty tuple; type variables and the values they can take; tuples having tuple components and list components; lists having tuple components or list components; the type of a tuple consisting only of empty lists; the type of a list consisting only of empty lists; list expressions made by using the ellipsis symbol `" ... "`, infinite lists, joining lists together; strings are simply lists of Char.

~~R~~ (Revision): Recall that the notation "`<val> : <type>`" is read as "value `<val>` is of type `<type>`", e.g., "`100 : Int`" is read as "value `100` is of type `Int`", i.e., symbol `":"` is to be read as "is of type".

~~R~~ (Revision): Recall that after the interpreter prompt "`?`", we are supposed to enter syntactically correct Gofer expressions, so that the interpreter calculates and displays their values. Instead of saying "Enter `<expr>` after the interpreter prompt," our shorthand is to simply say, "`? <expr>`" See the example below:

TUPLES

~~€~~: Find out the significance of a sequence of values enclosed within parentheses "`(`" and "`)`". Example:

```
?(2+2, 2.0, '2', 2<2)
```

From the interpreter response, observe the resulting value and its type.

```
(4, 2.0, '2', False) : (Int,Float,Char,Bool)
```

Repeat: symbol `":"` is to be read as "is of type.", i.e., the colon `":"` lies between the tuple and its type.

Definition: A *tuple* is a sequence of values enclosed in parentheses `(` and `)`, with fixed length and heterogeneous component types. (The term ***heterogeneous*** means diverse or different.) Thus, the number of components in a tuple is fixed, and each component type can be any legal Gofer type.

Notation (repeat): Values of type tuple, as well as the type itself, are denoted using parentheses `(` and `)`. From the response, note the manner in which a tuple value, and the tuple type itself, are written.

In other words, take any comma-separated sequence of Gofer expressions and enclose them in parentheses, and you have a tuple. Replace the expressions by their types, and you have the tuple type.

~~€~~: Enter the following tuples into the interpreter, one tuple at a time, and note their values and types.

```
(1,2.0,3,4.0)  ('1', 2, 3.0)      (1, 2.0,3, 4.0)      ('a ', 'A', 2)
(True, False)  (2<2, 2==2, 2>2)   ((1, 2),(1.0,2.0))   ((1,2,(1,2, '1')))
```

~~R~~: Use your imagination and make up your own tuples. To save yourself the effort of typing the tuples above, character-by-character, copy-paste each tuple into the interpreter. You can copy a string with ctrl-C, and paste it

in the interpreter with `right-click top border > Edit > Paste`.
(This cut-paste operation has proved to be "tricky" for some students. Tell me if there are problems.)

~~R~~: The components of a tuple need not have only simple types.. They can themselves be compound types. The only compound type we know as of now is tuple. Below, there are a few tuples some of whose components are themselves tuples. You can make up your own tuples-within-tuples too.

~~€~~: Find out the values and types of the following tuples by entering them into the interpreter:

```
(1,2,(3,4),5,6),    (1,2.0, (3,(4,5.0)), '6'),   (1,(2,(3,(4, '5'))))
```

LISTS

~~€~~: Find out the significance of a sequence of values enclosed within brackets "`[`" and "`]`". Example:

```
?[1, 10, 10*10]
```

results in the response

```
[1, 10, 100] : [Int]
```

~~R~~: Read `[Int]` as *list*-`Int`, meaning "list of integers".

Definition: A *list* is a sequence of values enclosed in brackets `[` and `]`, with zero or more components of *homogeneous* type, (*Homogeneous* means "all of the same type") . A list with zero components is also known as the empty list, written `[]`. Because the type of such a list is unclear, we postpone discussing it.

Please note: Interpreter experiments are the way to learn a whole variety of facts about lists (and other datatypes).

~~€~~: Find out the value and type of:

```
?[ 'A', '1', '?']
```

~~€~~: Find out the value and type of this list—it is just one list (the caret "^" is the exponentiation operator).

```
?[(10^1, 1.0, 'a' ), (10^2, 2.0, 'b'), (10^3, 4.0, 'c')]
```

~~R~~: Note that values of type list, as well as the list type itself, are denoted using brackets `[` and `]`.
Repeat: Observe the manner in which values of type list, and the list type itself, are written using `[` and `]`.
Note that since all components of a list are of the same type, the list type contains just a single type name, e.g., if a list consists of integers, the list type is `[Int]`, and its component type is `Int`.

~~€~~: Observe what happens when not all components in a list are of the same type, e.g., if we enter

```
?[1, 2, 3, 4, 5.0]
```

the interpreter response is

```
ERROR: Type error in list
*** expression     : [1,2,3,4,5.0]
*** term           : 5.0
*** type           : Float
*** does not match : Int
```

~~R~~: Since tuple components can be of any legal Gofer type, this kind of error cannot happen with tuples.

~~€~~: Enter the following lists into the interpreter and note their values and types.
Cut-paste the values below into the interpreter, one at a time, or make up your own lists.

```
['a','A','1']     [(1, 2.0),(3, 4.0)]     [[1,2], [3,4,5,6], [7,8] ]
['?',' ','=']  [('T', True),('F',False)]   [[[1,2]],[[3,4]], [[5,6]] ]
[False, False&&True, False||True, True]
```

~~R~~ (New operator): To join two lists together, use the *concatenation* or *append* operator, written as "++"

~~€~~: Enter the following expressions into the interpreter, and next, make up new expressions of your own.

```
[0 ... 4] ++ [5, 4 ... 0]   "abcd" ++ "1234"    ['a' ... 'z'] ++ ['A' ... 'Z']

[0.0 ... 4.0] ++ [5.0, 4.0 ... 0.0]      [0 ... 4] ++ []      [] ++ [0 ... 4]

[0 ... 3] ++ [4 ... 6] ++ [7, 6 ... 0]     [0,10,100] ++ ['a', 'b', 'c']
```

Question: What is the type of the empty list `[ ]`? Enter `[]` into the interpreter and note its response

```
? []
[] : [a]
```

~~R~~: *Important remark*: The component type above, `a`, is not a specific or fixed type, but a ***type variable***. A type variable can take on any type as its value. In any context, `a`, `b`, `c`, `d`, `...` will be the names of the type variables, e.g., if a context calls for three type variables, they will always be `a`, `b`, `c`.

Important note: For Gofer-like languages (dialects of Haskell, to be more precise), the "world of values" and "the world of types" are very different. In the world of values, an `Int` variable n can be bound to any valid integer value, say -10, `0`, `1000`, etc., whereas in the world of types, a type variable like `a` can be bound to any valid type like `Int`, `Char`, `(Int,Float)`, `[[Int]]`, `[(Int,Float)]`, etc.

(It is possible that the notion of type variables will not be clear to you. Please wait till it becomes clear.)

~~€~~: Verify that in the ex pression `[[1, 2],[]]` the type of the empty list is `[Int]`, while in `[['1', '2'],[]]` the empty list is of type `[Char]`. In `[[[1, 2]], [[3, 4]] []]`, the list is of type `[[[Int]]]`, so all its components, including the empty list, are of type `[[Int]]`.

Question: What is the type of a list of many empty lists, e.g., `[[], [], [], []]`?
(Recall that the components of a list have to be all of the same type.)

```
? [ [],[],[],[] ]
[[], [], [], []] : [[a]]
```

~~R~~: The type of the first component is `[a]`, i.e., list of any type. but in a list, the type of all components must be the same. Hence the type of all the remaining components is `[a]` , and therefore the type of the entire list is list-list-`a` or `[[a]]`.

Question: What is the type of a tuple of 2 empty lists, or a tuple of 3 empty lists?

Nomenclature: A tuple with 2 components is a pair, and one with 3 components is a triple.

```
? ([], [])
([],[]) : ([a],[b])
? ([], [],[])
([],[],[]) : ([a],[b],[c])
```

Explanation: Recall that unlike the components of a list, the components of a tuple need not be the same. For example, the first and the second components of the pair `([], [])` are each lists of any type, and moreover, their respective component types need not be the same. Therefore, if the first compnent type is, by convention, the type variable `a`, the second component type can, in general, be

another type variable, call it `b`. The same logic applies to a triple of empty lists: their component types are the type variables `a`, `b` and `c`.

~~R~~: If we can have an empty list `[]`, how about the empty tuple `()`? What is the type of the empty tuple?

```
? ()
() : ()
? :t ()
() : ()
```

In Gofer `()` is the 0-tuple. The value and its type are both called *unit*, and are written `()` in Gofer or Haskell. The type `()` has exactly one value, namely, `()`. It signifies "nothing" or "no information." In advanced classes, we might have use for `()`, but not any time soon. (This information is given just for your curiosity :)

~~R~~: We can have a singleton list, but not a singleton tuple, because of the usual semantics of parentheses:

```
? [100]
[100] : [Int]
? (100)
100 : Int
```

~~R~~: (This remark is a very general idea) In mathematical logic, value and meaning are synonyms, i.e., they mean one and the same thing. That is, the *meaning* of `2+2` is the same as the *value* of `2+2`.

~~R~~: If the value (or meaning) of the expression `2+2 is 4`, the meaning of the list expression `[0...4]` is

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

~~€~~: Find out the values of the expressions `[4...0]` and `[4,3...0]`. Try them out on your computer.

```
? [4 ... 0]
[] : [Int]
```

~~R~~: Here the upper bound of the list, i.e., `0`, is lower than the lower bound, `4`. Only an empty list can satisfy this constraint. If we want the integers 4 to 0 in decreasing order, then the way to write such a list is:

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

~~€~~: Find the values of the expressions

```
[0,2 ... 10]          [10,8 ... 0]          [0.0 ... 5.0]
[0.0, 10.0 ... 50.0]  [0.0, 0.5 ... 5.0]    [0.0, 0.1 ... 1.0]
```

~~€~~: What do you think are the values of the expressions

```
[0 ...]        [0,10 ...]        [0.0 ... ]        [0.0, 0.1 ... ]
```

~~R~~: This is how you write infinite lists!

~~W~~: You can stop the interpreter with the keyboard interrupt ctrl-C. This works with Linux, but! On Windows, be prepared for the machine to crash—you can restart Gofer (I think it is a software bug).

~~R~~: The computer performs not decimal arithmetic but binary arithmetic. When list elements increase in steps of `0.1`, you might see a numerical error creeping in, e.g., `38.7999` in place of `38.8`.

~~R~~: In the previous Lesson we noted an exception to the explicit display of type information when values of expressions are displayed. This exception is for strings (enclosed in double quotes) are entered into the interpreter. The double quotes are not displayed either. Q: What is its type?

```
? "Here is a string enclosed in double quotes"
Here is a string enclosed in double quotes
```

~~R~~: Observe that no quotes have been placed around the string.

```
? :t "Here is a string."
"Here is a string." : String
```

~~R~~: At the very end of this Lesson, you can learn an important fact about Gofer.

Important fact: A string is simply a list of characters, so the type `String` is a synonym for `[Char]`.

```
? "abcd" == ['a','b','c','d']
True : Bool
```

```
? ['a' ... 'z']
abcdefghijklmnopqrstuvwxyz
```

♋ ♦ ♋