# Introduction to functional programming, Lesson 1:
# DATA AND DATATYPES IN GOFER, SIMPLE TYPES

K M I T
R&D Lab
28-12-2021

When you start the Gofer interpreter, the message that you see is like this:

```
Copyright (c) Mark P Jones 1991-1994
Copyright (c) Rusi Mody 1995-2012
Gofer session for:
pustd.pre
?
```

The Gofer interpreter is an expression evaluator: think of it as a kind of calculator.

Mark Jones created the original language, Gofer, and its interpreter. That was 25 or 30 years ago.

Rusi Mody made changes to notation of the original language to make it suited for beginners.

Gofer stands for "good for equational reasoning". You already know equational reasoning from middle school mathematics, so the basic concept will not be new to you. We will soon review this notion, and you will discover the computational power of what you already know. Before we get there, it will help to get some practice with Gofer and its interpreter.

The current language standard for Gofer-style "purely functional programming" is Haskell. As Mody put it, "Haskell is real; Gofer is ideal." Decades have passed by since Gofer was first designed, but the ideas it contains are up to date, or we could say that the ideas are timeless.

The file `pustd.pre` is the "standard prelude" which contains quite a few predefined functions. Think of the predefined functions as "buttons on the calculator" that you can use to perform calculations. Soon, you will be able to write and use your own functions too.

The "?" is called the ***interpreter prompt***. If an expression that is consistent with Gofer syntax is entered after the prompt, the interpreter will evaluate its value, display it on the next line, and show the prompt on the following line. This activity is known as the "read-execute-print loop", or REPL for short. If a syntactically incorrect expression is entered at the prompt, the interpreter attempts to find the syntax error, and print it out for the user to read and repair.

Note: In these lessons, € is the symbol for an exercise that the student to perform, ₨ is a remark that the student is to understand, and ₩ is the indication of a warning that the student is supposed to heed.

€: Enter the following expressions at the interpreter prompt, one at a time. Notice that the interpreter does indeed evaluate these expressions. (Note that `True` and `False` start with a captial letter.)

```
2            2.1          False        'a'

2+2          2.1 + 3.2    2<3          '2'

2*3 + 4      True         3<2          ' '
```

Notice that after the value of each expression, thie interpreter prints out its type, e.g., "`2 : Int`". The colon symbol " `:` " is to be read as "of type", thus the interpreter response "`True : Bool`" is read as "`True` is of type `Bool`." (`Bool` is short for Boolean, a data type contains just two values, `True` or `False`). The expressions we have entered so far are of the four "simple" types: `Int` (integer), `Float` (floating-point, explained in a paragraph below), `Char` (character) and `Bool`.

Definitions: A simple data type is one that cannot be expressed as a composition of simpler types, while a compound type is made out of simpler (simple or compound) types.  Informally, a simple type is like an atom, whereas a compound type is a molecule made out of several atoms.

~~R~~:  You are familiar with numbers written using a decimal point, e.g., 3.14159. So why do programmers call this type floating point? Consider the following interpreter interactions:

```
? 1.23                      ? pi
1.23 : Float                3.14159
? 123000.0                  0.000123
123000.0 : Float            0.000123 : Float
? 1230000.0                 ? 0.0000123
1.23e+006 : Float           1.23e-005 : Float
? 123000000000.0            ? 0.0000000123
1.23e+011 : Float           1.23e-008 : Float
```

Notation: When we want to write numbers having an exponent and a mantissa, e.g., $1.23 * 10^{6}$, without superscripts, we write 1.23e+06.  Similarly, we write $1.23 * 10^{-6}$ as 1.23e-06

~~R~~: Mathematicians would normally call type Float as real, but that usage is not correct, e.g., we are denoting the value of pi correct to only 6 digits, whereas in the exact value of pi the number of digits is unbounded. Such a representation of real numbers is called “finite precision.”  (The number of digits is about 6 for 32-bit computer architectures, and is about 16 for 64-bit computer architectures.)

~~R~~ (Floating point): Type Float is short for floating-point because, when the magintude of a number is sufficiently large (>= $10^{6}$, approx) or sufficiently small (<= $10^{-5}$, approx), the decimal point “floats” to a position immediately after the most significant non-zero digit, see examples above.

€: Verify that the types of different expressions are printed out with a capital or upper-case letter. Thus Int, Float, Char, Bool.

~~W~~: While typing the Boolean constants  True  or  False, be sure that the first letter is upper-case.

€: You can see for yourself what happens if you use lower-case letters and enter  true  or  false.

(Remark: Terms starting with a lower-case letter have a different significance: they are the names of variables, e.g., pi. When the name pi is entered at the prompt, its value is returned (correct to 6 places )

€: Enter expressions like  2^5 and 2.0^5, and verify that ^ is the exponentiation (power of) operator.

€: Enter expressions like 2^5.0 and 2.0^5.0, and verify that while the mantissa can be Int or Float, the exponent has to be Int only.

€: Enter the expressions 2+2 == 4,  2+2 == 5, 2 /= 3,  2 /= 4,  True == False,  True /= False.

Conclude that  ==  and  /=  are Boolean-valued equality and inequality operators that return the values True and False depending on the values of their operands.

€: Enter any expression like 2+2 = 4, or even something as simple as  2=2, and see what happens.

~~R~~: Recall that it made sense in school mathematics it made sense to write, e.g.,  2+2 = 4.  You will soon find that in functional programming, the "="  symbol means “equality by definition”, e.g., pi is given its value with the equation  pi = 3.14159.  (We will postpone the details for a few more Lessons.)

€: Enter the expressions `False&&False`, `False&&True`, `False||True`, `True||True`, etc., and observe their values. Conclude that `&&` and `||` are the OR and AND operators for Boolean-valued operands.

Ɍ: Defnition (*Literal*): In natural language, to be literal is to understand words in their usual or most basic sense . In programming, a *literal* is a value written exactly as it is meant to be interpreted, e.g., `100`, `1.00`, `'a'` or `True`. These are literals of type `Int`, `Float`, `Char` and `Bool`, respectively.

Ɍ: Literals are the simplest expressions, i.e., you cannot get any simpler expressions, but expressions composed out of more than one literal are not themselves literals , e.g., 2+2 is not a literal.

Ɍ: In mathematical logic, the words *value* and *meaning* are synonymous. Now we understand well that the value of 2+2 is 4. We need to understand likewise that "the meaning of 2+2" is 4.

₩: If we enter character sequences that do not amount to "syntactically well-formed expressions", the interpreter will give an error message. Initially, some error messages may sometimes appear confusing, and not indicate explicitly and clearly what exactly the error is. With practice you will learn to decipher these messages.

€: Try entering expressions that are grammatically not correct, like `2 * 3 *` , `.1.` and `'a` and see the error messages for yourself. If you enter a sequence of letters like `xyz`, the interpreter will state "`ERROR: Undefined variable`". If the sequence starts with an upper-case letter, the interpreter will respond "`ERROR: Undefined constructor function`". (We will study constructor functions in later lessons.)

€ (Note carefully): Observe the error message when you type two sequences of letters with a blank space in between, e.g., `abc def`. An explanation of this error message requires a few basic concepts, so it is postponed. It is enough to note that the word "juxtaposed" means being next to each other: the interpreter is stating that the letter sequences `abc` and `def` are juxtaposed (which is, apparently, an error).

€: For the interpreter to indicate the type of an expression, enter `:type` followed by the expression. (Instead of `:type` you may enter just the first letter, i.e., `:t`, followed by the expression.) The interpreter will first print out the entire expression (not its value) and then indicate its type.

€: Enter `:t` followed by some of the expressions above, and see how the interpreter responds.

€: Enter some characters in double quotes, like `"hello, world!"` , the interpreter will print back these characters minus the enclosing quotes, but it will not say what the type is.

€: To explicitly ask for the type of a character sequence in double quotes, enter `:t "hello"`. The interpreter will print out the characters along with the quotes, and indicate that the type is `String`, i.e., a string of characters.

₩: Take note that a string of characters is not a simple type. A single character, denoted in single quotes, e.g., `'a'` or `','` (comma) or `' '` (blank space) is indeed a simple type.

€: Enter a pair of single quotes (with nothing in between), i.e., `''`, and note the error.

€: Enter a pair of double quotes with nothing in between, i.e., `""`, a string of length 0. What happens?

Ɍ: One reason that the interpreter normally does not print out the type of strings (in general, string-valued expressions) is that in the course of its work, e.g., while printing error messages, the interpreter has to print out strings. It is not a good practice to label such messages with their type, because doing so will only distract the user.

€: Enter the string `error. "Invalid argument"` and see how the interpreter responds. It states `Program error: Invalid argument`. An error message is being printed, indicating the kind of error. (The message is a string, but we don't want its type mentioned, because that would be confusing.)

€: Enter a string with arbitary characters within double quotes, e.g., `error. " ... "` and see how the interpreter responds.

(It appears that these two strings above are syntactically correct expressions. A further explanation, e.g., why does the interpreter respond in this way, is postponed till you write some programs.)

~~R~~: So far, whenever the interpreter displayed the value of an expression, its type was also invariably displayed. (Type `String` is the exception.) This is the default style of displaying expression values, but it is possible to avoid displaying the type for all expressions.

€: Enter the interpreter command `:set -t`, or simply use the first letter, thus `:s -t`. Then enter some expressions, and verify that their correct value is displayed, but their type is not. (To view the type of an expression, or its calculated value, we have already seen the use of the interpreter command `:type` or `:t`.) Enter the command `:set +t`, or `s +t`, and verify that the type of an expression is displayed with its value.

€: Assume that with `:s -t`, type information is not being displayed along with values. Now enter these expressions, one at a time: `2:Int`, `2.1:Float`, `'2':Char`, `2+2: Int`, `2.5^2: Float`. Verify that the interpreter accepts the type along with the input expression, but does not display it with its value.

€: Enter an incorrect type with an input expression, e.g., `2+2:Char`, and observe the interpreter response.

~~R~~: The expression `2.1 + 2` makes perfect sense to us.
€: Try entering it in the interpreter. You will get an error message of a few lines.

```
ERROR: Type error in application
*** expression     : 2.1 + 2
*** term           : 2.1
*** type           : Float
*** does not match : Int
```

Let us parse the multiline error message, for this kind of message is going to recur again and again. The message indicates that `2.1` is of type `Float`, `2` is of type `Int`, and the two types do not match. The conclusion is: we cannot add two numbers, of type `Int` and `Float`, respectively. Either they have to be both `Float` or both `Int`.

€: Enter `2 + 2.1` and notice how the error message is different from the one above.

€: Enter `2.0 + 2.1` and see the result for yourself. Also verify that the same rule applies for other arithmetic operators, like `–`, `*` and `/`.

€: Enter the expressions `2 + 3 + 4 + 5.0` and `2.0 + 3.0 + 4.0 + 5`, and note the error messages.

€: Now enter the expression `2 * 3 / 4` and see the error. The expression appears ambiguous to the interpreter. Next enter two expressions, `(2*3)/4` and `2 * (3/4)`. The first causes the `*` to happen first, and the other that causes the division to happen first.

~~R~~: Verify that `6/4` has the value `1` and `3/4` is `0`. We conclude that when the operands are integers, the interpreter performs integer division, i.e., only the quotient is retained as the value of the division, and the remainder is lost. There is no such ambiguity when the operands are floating point numbers, e.g., `2.0 * 3.0 / 4.0`. But we still get the same warning, that there is ambiguous use of the `/`

operator. Just be sure to put parentheses in the right place when we use the division operator along with the multiplication operator.

Definition (Conditional expression): A conditional expression is one whose value depends on a Boolean-valued expression. Its syntax is as follows:

```
if <expression: Bool> then <expression1> else <expression 2>
```

Its value depends on the value of `<expression:Bool>`. If it is `True`, then the value of the expression is `<expression1>`, otherwise it is `<expression2>`.

The tokens `then <expression 1>` are called the *then clause* of the conditional expression, and `else <expression 2>` is the else clause.

€: Try out these examples on your interpreter:

```
? if 2+2 == 4 then 'T' else 'F'
'T' : Char

? if x<100 then 10 else 20 where x = 0
10 : Int
```

€: Verify the workings of conditional expressions by evaluating these expressions (and make your own):

```
if True then 100 else 200      if False then 100 else 200
if 1<2 then 3 else 4           if 2+2==5 then 3 else 4.
```

Definition (Conditional expression): A conditional expression is one that takes one of two values, depending on the value of a “condition”, i.e., a Boolean-valued expression. Its syntax is as follows:

```
if <expression: Bool> then <expression1> else <expression 2>
```

Its value depends on the value of `<expression:Bool>`. If it is `True`, then the value of the expression is `<expression1>`, otherwise it is `<expression2>`.

The tokens `then <expression 1>` are called the *then clause* of the conditional expression, and `else <expression 2>` is the else clause.

€: Try out these examples on your interpreter:

```
? if 2+2 == 4 then 'T' else 'F'
'T' : Char

? if x<100 then 10 else 20 where x = 0
10 : Int
```

€: Verify the workings of conditional expressions by evaluating these expressions (and make your own):

```
if True then 100 else 200      if False then 100 else 200
if 1<2 then 3 else 4           if 2+2==5 then 3 else 4.
```

R: It is mandatory that the if- and else-clauses of a conditional expression return values of the same type.

€: What happens if the following conditional expressions are entered into the interpreter?

```
if True then 100 else 100.0    if 3<2 then 100 else 'a'
```

<See the next page for a summary of this Lesson>

What we learnt in this Lesson: 1. The gofer interpreter is an expression evaluator. 2. Expressions always have type. The types we have seen so far are: `Int`, `Float`, `Bool`, `Char`, `String`. 3. The type information provided with the value of an expression, or not provided, depending on whether we give the interpreter command `:s +t` or `:s -t`. In our system, the former is the default. 4. The type of expressions of type `String` is not indicated. 5. The type of an expression can be explicitly asked, using the command `:t` followed by the expression. 6. `++` is the string concatenation operator. 7. Binary arithmetic operators like `+` and `*` require both operators to be either `Int` or `Float`. We cannot have one `Int` operand and another `Float` operand. 8. Use of `/` along with `*` results in a perceived ambiguity. Parentheses are required to resolve it. 9. Conditional expressions take one of two values, depending on the value of a condition (i.e., Boolean-valued expression).

♋ ♦ ♋

IPMF Lesson 1: date 06-06-2020, revised 1-07-2020, 15-12-2020, 25-02-2021