Introduction to Programming

Lesson 4: WRITING AND LOADING A GOFER SCRIPT

~~(R~~: Some of the remarks, terminology and definitions below were already stated in earlier Lessons. The motive in repeating them is to reinforce unfamiliar concepts, so that students remember them.)

~~R~~: The Gofer interpreter operates *interactively*, i.e., the user enters an expression at the interpreter prompt "`?`" and terminates it by pressing the Enter key. The interpreter responds by evaluating the expression and printing its value on the next line. Then it repeats the interpreter prompt.

~~R~~: The interactive behaviour of the interpreter has a name: *REPL*, short for "read-evaluate-print loop." (If interpreters for other languages accept commands which are to be executed, instead of expressions that are to be evaluated, REPL is understood as "read-execute-print loop.")

~~R~~: Even if an expression is very long (and occupies multiple lines on the monitor), it is still a single expression. We cannot enter anything more than a single expression into the interpreter, but that expression can be more complicated than the ones we have seen so far: we can locally define and use variables within expressions.

NAMES, VALUES AND BINDINGS

~~R~~: When we speak of a *variable*, we have two ideas in mind: the *name* of the variable, and its *value*, e.g., the name π or `pi`, and its value `3.14159...` . (In this instance, the value is a constant, so the term "variable" is a misnomer.) We can have a name called `height`, and it might or might not have a value.

Definition (*binding*, *bound variable*): A name is bound to a value with the equality symbol "=", e.g., `height = 178` . Variable definitions have the following syntax: *name*, followed by the equality sign, followed by an *expression*, e.g., `w = 100` or `x = [1...4]` or `z = y+2`, where `y` will itself have to be defined in the script (unless it is predefined in the prelude).

(We could say that a name is bound to a value, or alternatively, that the value is bound to the name. In my view, it is one and the same thing—there is a name, there is a value; they get bound to one another.)

Important: These definitions are *not at all* the same as the assignment statements of languages like C. In languages like Gofer, name-value bindings are *immutable*, i.e., once made, they are unchangeable. In contrast, in C-like languages, a value can be assigned to a name, and later, another can be assigned. We could say that a binding in Gofer is a fact, whereas an assignment in C is an action—a verb.

Equations or definitions like this one are placed in a plaintext file known as a Gofer script (file extension `.gs`). The script is loaded into the interpreter for the name to have a value. Then if we type in the name into the interpreter, it will respond with the value.

The value of the expression containing a variable will depend on its value. If a variable has not been bound to a value using the "=" symbol, and we attempt to evaluate an expression containing the variable, the interpreter respond with an error message: "`undefined variable`."

R: We can define names locally and bind them to values, with the keywords `where` and `let` . These bindings are limited to the expression being entered, and are lost after the expression has been evaluated.

~~€~~: Evaluate expressions into the interpreter using the `let` and `where` keywords (see the examples below).

```
? x+2 where x = 10
12 : Int

? let x = 10 in x+2
12 : Int
```

```
? [n ... n+5] where n = 3
[3, 4, 5, 6, 7, 8] : [Int]

? let n = 1 in [n ... n+5]
[1, 2, 3, 4, 5, 6] : [Int]
```

€: Evaluate tuples like `(x+1, x-1)`, after binding `x` to an integer with the `let` and `where` keywords.

R: You might think that these examples are trivial, so the keywords are unnecessary. Their use comes in handy when expressions get long and complicated, and we want single names for long sub-expressions.

R: In Gofer's interactive mode, we can only use a single "=" symbol to create a binding. So you can bind two or more names to values by putting the names as components of a list or tuple, thus:

```
? x+y where (x, y) = (10, 20)
30 : Int

? [x ... y] where (x,y) = (10,15)
[10, 11, 12, 13, 14, 15] : [Int]
```

```
? let (x, y) = (10, 20) in x+y
30 : Int

? let (x, y) = (10,15) in [x ... y]
[10, 11, 12, 13, 14, 15] : [Int]
```

~~R~~: We can have variables whose bindings to values last beyond a single expression. For this purpose, we need to write Gofer scripts.

GOFER SCRIPTS

Definition (repeat) : A Gofer script is a plaintext file with extension `.gs` . It consists of a set of definitions that bind names to values. One important use of scripts is to define functions (which we have not learnt so far). A script file can be saved in non-volatile memory (like disk) and can be used repeatedly whenever needed.

~~R~~: The definitions (see examples below) are communicated to the interpreter by *loading* the script into the interpreter. At that time, the expression on the right hand side of each = symbol is evaluated, and that becomes the value of the name on the left hand side of the =, i.e., the name is ***bound*** to that value. After the script is loaded into the interpreter, we can construct expressions containing these names and load them into the interpreter, and it will evaluate these expressions in the manner we expect.

~~€~~: Open a plaintext editor, like `Notepad` on MS-Windows or `gedit` on Linux. (Notepad comes bundled with MS-Windows, and is found at `Start > All Programs > Accessories`). Enter some equations, one to a line (see below), save it as `trial04.gs` and load it in the interpreter.

~~W~~: Under MS-Windows (but not under Linux), you might get an error message like this:

```
? :l trial04.gs
ERROR "trial04.gs": Unable to open file
```

Explanation: Notepad has saved this file with name `trial04.gs` and extenstion `.txt`. If you put double quotes around `trial04.gs`, i.e., if you type in `"trial04.gs"` and only then save the file, then the file will have the name `trial04` and extenstion `.gs`. Verify this in the directory listing.

Along with this Lesson, file lsn04.gs is supplied. It contains the following bindings:

```
p = 100
q = 10.0
r = [1...4]
s = (2, '2', 2<2)y = 2
z = y+3
t = (hd, tl) where
   hd = 100
   tl = [0 ... 5]
--You have not seen the :: operator so far
--Find out the values of t and u
u = hd :: tl where
   hd = 100
   tl = [0 ... 5]
```

€: Suppose that lsn04.gs had not been supplied, and you had to make it yourself.

Be sure that *the starting characters of all 3 lines are aligned vertically, i.e., they start at the same column*. Initially, let all lines start from column 1. Save this file in the same folder as the Gofer interpreter. Repeat: While saving the file, give its name in quotes and with `gs` as the extension, like this: `"lsn04.gs"`. Repeat: If you don't put the name in double quotes, then Notepad in MS-Windows automatically gives the extension `.txt`.

€: (Repeat) Load the file `lsn04.gs` In the interpreter, with the command `:load lsn04.gs` . Note: It is sufficient to use just the first letter of the command, like `:l lsn04.gs` . The interpreter will start a new Gofer session, after loading the definitions in files `pustd.pre` and `lsn04.gs` .

€: Make simple expressions using variables defined in `lsn04.gs`, and enter them into the interpreter. Verify that the interpreter evaluates the expressions and displays their values in a manner that you expect.

~~R~~: Sometimes you need to make corrections or additions to a file you had earlier loaded. For instance, there may be typographical errors, syntax errors or logical errors in your script, or you may want to add some definitions or modify old ones . After you have made these changes, save the file. Then simply enter `:reload` or `:r` , without supplying the file name. The `reload` command loads the file that was last loaded with the `load` command. *Don't forget to save the file!* Otherwise the changes you have made won't get loaded into the interpreter. (You can get the same effect as the `reload` command by giving the `load` command and specifying the name of the file that was saved.)

€: After loading or reloading file `lsn04.gs`, enter at the interpreter prompt any of the variables you have defined, e.g., `p`, and it will print the value of `p` that you specified in the script. Optionally it will print the type of `p` also, provided the type information has been enabled (by default, or after entering the command `:s +t`). Recall that `:t p`, causes the interpreter to explicitly tell you the type of `p` without giving its value. You can do the same with the other variables you defined. You can verify that the interpreter has evaluated `z` correctly, as `y+3`, since `y` is `2`. Enter expressions involving these names.

~~R~~: The exponentiation operator in Gofer is the caret `"^"`. Make expressions like `p+y`, `p*y`, `p^2`, `p^3` (i.e., `p` square and cube), or `q^2`, `q^3` and verify that the interpreter evaluates these expressions correctly. Note especially that the type of the result can be `Int` or `Float`.

€: Change the values on the right hand side of the equations in the script, save the file (as discussed earlier), reload the changed file into the interpreter, and verify that the values of the names and of expressions containing the names are correctly evaluated.

€: Interchange the bindings of `y` and z in file `lsn04.gs`, `z=y+3` comes first, and `y=2` comes later. That is, z is first defined in terms of `y`, and next, y is defined to be `2`. Then save the file and reload it into the interpreter. Verify that this makes no difference to the interpreter.

~~R~~: In functional programming, the equations are treated as a *set* of definitions (in which the order of their occurrence is not significant). On the other hand, in the languages with which most persons are familiar, these lines are regarded as a *sequence* of assignments (the order in which the assignments are made is important). This issue is considered in greater depth in the next Lesson, entitled "The Term-rewriting Model of Computation."

€: Now let us observe what happens if all the lines of the script are not aligned. Given all the lines in file lsn04.gs starting at column 1 (as suggested above), place a blank space before the start of any one line, say line 2. The interpreter will generate a seemingly confusing error:

```
ERROR "lsn04.gs" (line 2): Juxtaposition has no meaning. Use .
```

In earlier lessons, we saw that this arose when the function application operator "`.`" is absent. We should consider that this error message is misleading, and the real issue is the lack of alignment of line 2. It is the blank space at the start of the line that caused this error.

Moving a line to the right by adding blank spaces before its start is called *indentation*, which has meaning only in multi-line statements (i.e., single statements that span multiple lines), and is an error otherwise. Remove the indentation that you had made, and the error will go away. Indent two or more lines, and the interpreter flags only the first indented line.

~~€~~: Now place a blank space before the start of line 1. The error message you will see is

```
ERROR "lsn04.gs" (line 2): Syntax error in input (unexpected symbol)
```

This error is misleading too. The indentation in line 1 caused the interpreter to expect that the subsequent line would be indented too, so that it would be vertically aligned with the first one.

Enter the same number of spaces, say 2 spaces, before each line, and save and load the file into the interpreter. The file gets loaded because the blank spaces are not treated as indentation, i.e., no line(s) are indented inwards or outwards with respect to other lines.

Note an important point: An indentation error gets caught in the process of loading the file into the interpreter, and on catching it, the loading process stops.

~~₹~~: You can give many commands to the Gofer interpreter. The commands that we have already used, `:l`, `:r`, `:t` and `:s` are among them.

~~€~~: Enter `:?` to get a list of commands that you can give to the interpreter. Go through the list before reading the description below.

All interpreter commands except one start with a colon, symbol : . The commands that we are going to use most often are `:l`, `:a`, `:r`, `:e`, `:t`, `:s`, `:n`. (These commands are short for `:load`, `:also`, `:reload`, `:edit`, `:type`, `:set` and `:names`.) We have already used many of these commands. A few words about some of them are repeated. Whereas the `:l` command loads a file after erasing all other files that have been previously loaded, the `:a` command loads additional files without removing files that were loaded earlier. The `:e` command calls the editor from within the interpreter—no need to go explicitly to the editor. (Notice: It appears that in the present version of the Gofer intrpreter wor MS-Windows, this command has not been implemented, or does not work.) The `:n` commands lists all the names that are known to the interpreter. These include those in the prelude file and those that have been loaded by you.

The only command that does not start with a colon is described as `<expr>`. The angle brackets "<" and ">" that surround the term called "`expr`" are to be found as part of other commands also, e.g., `:load <filenames>`.

~~₹~~: The angle brackets < and > serve as a placeholder for the term that is enclosed in them, e.g., `<expr>` stands for any expression, while `<filenames>` stands for any blank-separated sequence of file names. When we do enter a concrete expression like `2*3+4`, we do not put angle brackets around it.

Thus, in the list of interpreter commands `<expr>` is a way to say, any expression can be entered at the interpreter prompt, without a prefix of `:`. The explanation to the right of `<expr>` indicates that the action of the interpreter when an expression is input is to evaluate the expression.

Typical examples of <expr> are 2*3+4, 2.0+2.1, 2<3, 3<2, 'a', "abcdef". We have already experienced entering such expressions into the interpreter, which computes their values and displays them before producing yet another interpreter prompt. (The types of these various expressions are different—note that currently we are not discussing types, but instances of <expr>.)

Date 06-01-2020 Time 06:15
rev 22-12-02-2020 Time 07:48
rev 08-01-2020 Time 08:25