Practical Work of Languages, Technologies, and Paradigms of Programming

2018-19

Part II Functional Programming

UNIVERSITAT
POLITÈCNICA
DE VALÈNCIA

Practice 4

Índice

1. Objectives of this practical exercise

The aim of this practical exercise is to introduce the Haskell programming language and to present the basic functionalities of the GHCi environment, an interactive version of the "The Glasgow Haskell Compiler"(GHC).

We will use GHCi in this first practice and we will combine its use with that of GHC in the following practices devoted to Haskell.

2. How to use GHCi

The system is installed in the practice labs and the application is accessible from any directory. The interpreter responds by typing the command `ghci` in a Linux shell:

```
$ ghci
GHCi, version 7.8.3: http://www.haskell.org/ghc/  :? for help
Loading package ghc-prim ... linking ... done.
Loading package integer-gmp ... linking ... done.
Loading package base ... linking ... done.
Prelude>
```

When starting up the environment, the interpreter is waiting for us to insert a command or an expression to evaluate. It is possible the autocompletion of expressions and commands, by means of the tab key. Most of the available commands en GHCi start with a ':' followed by one or more characters. It might be important to remember the following frequently used commands:

`? :q` Quit GHCi
`? :?` Show the list of available commands.

Some of the commands are shown below. Any of these commands can be abbreviated with the first character, that is `:l` in stead of `:load`.

```
:load <modulename>   - loads the specified module
:reload              - reloads the current module
:type <expression>   - print type of the expression
:info <function>     - display information about the given names
:quit                - exit the interpreter
:help                - shows available commands
```

To interrupt a running process, you can use the usual `^C`.

Normally we will have simultaneously open the GHCi interpreter and a text editor to write the programs that will be loaded in GHCi. We recommend

using an editor with syntax highlighting of Haskell code. In the Linux environment there are several editors with syntax highlighting which recognize as Haskell code the files with the `.hs` extension. Among them, we can mention EMACS (as it is configured in the laboratories), GEDIT or GEANY.[1]

2.1. Evaluating expressions with GHCi

We can type Haskell expressions directly into the input line of commands. For example, to evaluate

```
2+3*8
```

it suffices to write it and typing intro afterwards. GHCi will evaluate the expression and will show the result. Then it will show again the *prompt* which, by default, unless we have imported other modules, will be `Prelude>`.[2] The result on your terminal should look like this:

```
Prelude> 2+3*8
26
Prelude>
```

Arithmetic functions like `+` and `*` are predefined and are written with infix notation. Other arithmetic functions are written with prefix notation, like `div` or `mod` that compute, respectively, the quotient and the remainder of the integer division:

```
Prelude> div 15 2
7
Prelude> mod 12 5
2
```

However, infix functions can be turned into prefix by putting parenthesis around them:

```
Prelude> (+) 2 3
5
```

Similarly, prefix can be turned into infix by outting it in between ‘. For example:

```
Prelude> 15 ‘div‘ 2
7
Prelude> 12 ‘mod‘ 5
2
```

[1] Although the practices are done in Linux, we can mention that there are text editors with Haskell syntax highlighting available for Windows: CONTEXT as well as the versions of GEANY and EMACS.

[2] `Prelude` refers to the default module that is initially loaded by the system and whose definitions can be used in the session.

It is possible to query the type of an expression (`:type` command, that can be abbreviated `:t`). Compare, por instance, the result of the following expressions:

```
Prelude> True && False
False
Prelude> :t True && False
True && False :: Bool
```

In the second case, GHCi tells us that the expression is evaluated to the logical type (`Bool`). Notice that the symbol `::` is used to indicate that the expression on the left is of the type that appears at the right.

2.2. Editing and loading programs

A Haskell program is a set definitions of data types, functions and so on. This is organized as a collection of modules. From a syntactic point of view, a modules starts with the keyword `module` followed by the module name (which must start in **uppercase**) and the word `where` which opens a block, as in the following example:

```
module Signum where
  -- definition of signum' function:
  signum' x = if x < 0 then -1 else
              if x == 0 then 0 else 1
```

One of the ways to create comments in Haskell is to use the characters `--` to start a comment until the end of the line, as shown in the previous example.[3] Observe also that in Haskell each `if` must be accompanied by its corresponding `else`.

Haskell makes use of indentation to mark block endings as in other languages such as Python and unlike others like Java, which makes use of curly braces (`{` and `}`) to this end.

For very simple examples, we can write Haskell code in a file without defining a module. In that case, this is like defining a module called `Main`.

In order to be able to load a program in GHC or in GHCi, the filename must have the `.hs` extension. Although the name of a module does not have to match the name of the file where it was written, it is highly recommended to do so. In this way, edit and save the previous file as `Signum.hs`.

You can load the program in GHCi as follows:

```
Prelude> :load Signum.hs
```

[3] We can also create comments scoping several lines by using the symbol `{-` to open the comment and `-}` to close it.

```
[1 of 1] Compiling Signum           ( Signum.hs, interpreted )
Ok, modules loaded: Signum.
```

In that case, we assumed that the program `Signum.hs` was in the same folder where GHCi is being executed.

As explained before, GHCi is able to perform automatic completion by using the TAB key, as is the case of many shells. For instance, if you write `:l S` and press the TAB key, GHCi completes the command with the complete filename. The same can be observed when evaluating expressions, since GHCi can complete the name of a function.

Observe that, once the program is loaded, the *prompt* changes and now, instead of `Prelude>`, we can read `*Signum>`. Now, let's try some simple calls to the `signum'` function:

```
*Signum> signum' 0
0
*Signum> signum' 100
1
*Signum> signum' (-100)
-1
```

It is worthwhile remembering that every `:load` command initializes the database of the interpreter. This means we can only evaluate expressions that contain predefined functions or functions that have been defined in the current loaded module (in the above example, the `signum'` function), or functions from modules that have been loaded by the current module. Nevertheless, `Prelude` is always imported by default.

2.3. Warning and error messages

GHCi reports possible syntax and type errors during the loading of a file. Open the editor and write the following program:

```
module Hello where
  hello n = concat (replicate n 'hello ')
```

Save the file with the following name: `Hello.hs`. Then load the file in GHCi using the `:load` command. This will produce the following error message:

```
Prelude> :l Hello.hs
[1 of 1] Compiling Hello            ( Hello.hs, interpreted )

Hello.hs:2:41: parse error on input `)'
Failed, modules loaded: none.
```

Note that it is shown the exact position where the error can be found (row 2, column 41). Correct the error by replacing the simple quotes in `'hello '`

with double quotes, i.e. `"hello"`. Try to load the program again using the command `:r`, which is the abbreviation of `:reload` which reloads the last module.

Now, although the compilation does not produce any error, it is a good practice to explicitly write the profile of the functions that are defined in a module.

The profile of the functions indicates its type. If it is not written, the compiler tries to deduce or infer it from the definition. For example, you can check the type automatically inferred by the interpreter for the function `hello` as follows:

```
*Hello> :t hello
hello :: Int -> [Char]
*Hello>
```

which indicates that the `hello` function receives an `Int` value and returns a list of `Char`. Adding this profile to the function definition is advisable and may help to detect errors.

Note: Lists of elements are defined in Haskell using square brackets, e.g. `[Char]` for a list of Characters, `[Int]` for a list of Integers, etc. Moreover, the type String in Haskell is managed as a list of Characters. In this way, the expression `"hello"` is internally represented as the list `['h','e','l','l','o']`. Lists will be studied into more detail in the following practice.

After loading the program and checking that it does not have any error, the interpreter can evaluate expressions containing functions defined in the program. For instance, we can evaluate the expression `Hello 10`, as follows:

```
*Hello> hello 10
"hello hello hello hello hello hello hello hello hello hello "
```

2.4. Types

Haskell is a strongly typed language. Type checking is done at compile time. GHCi is not only capable of detecting type errors, but can also suggest possible ways in which certain conflicts can be solved. To illustrate this, write the following program in the editor and save it with the name `Typeerrors.hs`.

```
module Typeerrors where
  convert :: (Char, Int) -> String
  convert (c,i) = [c] ++ show i

  main = convert (0,'a')
```

Note: you can define in Haskell tuples of any arity (amount of elements) by using parenthesis and commas. Two examples of tuples are the date

type `(Char, Int)` and the pattern `(c,i)` that you can see in the function definition above.

Note: The `show` function is used to convert into string (`String` type or also `[char]`) many different data types (specifically, the ones of the *type class* `Show`, this will be studied in practice 6).

Now, go back to the interpreter and load the file. You will see that the interpreter shows us the following error:

```
Typeerrors.hs:6:20:
    Couldn't match expected type 'Int' against inferred type 'Char'
    In the expression: 'a'
    In the first argument of 'convert', namely '(0, 'a')'
    In the expression: convert (0, 'a')
Failed, modules loaded: none.
```

This error indicates that the expression `(0, 'a')` is of type `(Int , Char)` while an expression of type `(Char , Int)` was expected according to the type defined in the program for the function `convert`. Moreover, the error message suggests a possible way to solve the problem: re-order elements `0` and `'a'`.

Note: it is possible to define functions directly in the GHCi console by starting the definition with the reserved word `let`, whose general usage in Haskell is described in the prior-reading material:

```
Prelude> let convert (c,i) = [c] ++ show i
Prelude> convert ('c',3)
"c3"
Prelude> :t convert
convert :: Show a => (Char, a) -> [Char]
```

3. Exercises

3.1. Already solved exercises

1. Write a function `nextchar` that takes as an argument a character and returns the character that follows it in the alphabet.

   ```
   module Nextchar where
     import Data.Char
     nextchar :: Char -> Char
     nextchar c = chr ((ord c) + 1)
   ```

   and its execution

   ```
   *Nextchar> nextchar 'a'
   ```

```
'b'
```

2. Define a function `fact` that computes the factorial of a given non-negative integer number.

```
module Factorial where
  fact :: Int -> Int
  fact 0 = 1
  fact n = n * fact (n - 1)
```

and its execution

```
*Factorial> fact 3
6
```

3.2. Exercises you have to solve

1. Write a function `numCbetw2` that returns the amount of characters that are between two given characters in the alphabet (without including them). For instance:

```
> numCbetw2 'a' 'c'
1
> numCbetw2 'e' 'a'
3
> numCbetw2 'a' 'b'
0
> numCbetw2 'x' 'x'
0
```

2. Write a recursive function that returns the sum from one given integer to another one (both of them are included in the sum).

```
> addRange 5 5
5
> addRange 5 10
45
> addRange 10 5
45
```

3. Define a binary (receives two arguments) function `max'` that returns the maximum of two given integer arguments. Example:

```
> max' 5 50
50
> max' 10 1
10
```

4. Write a function `leapyear` that determines if a given year is a leap year or not. A year is a leap year if it is evenly divisible by 4. An exception to this rule is that years that are also evenly divisible by

100, are leap years if and only if they are also evenly divisible by 400. Example:

```
> leapyear 1992
True
> leapyear 1900
False
```

5. Write a function `daysAMonth` that computes the number of days in a given month and year. Take into account the leap years for February. Example:

```
> daysAmonth 2 1800
28
> daysAmonth 2 2000
29
> daysAmonth 10 2015
31
```

6. Write a function `remainder` that returns the remainder of the division of two integers *using subtraction*, which means that you cannot use `div`, `mod`, `rem`,... Example:

```
> remainder 20 7
6
```

7. Using the previous functions, define a function `sumFacts` that computes the sum of the factorials till a given number n. Otherwise stated, `sumFacts` n `= fact 1 +` $\cdots$ `+ fact` n. Example:

```
> sumFacts 5
153
```