Haskell Sequences

COMP40009 – Computing Practical 1

7th – 8th October 2021

Aims

- To gain some familiarity with the Linux operating system, and the basics of the programming language **Haskell**.
- To learn the basics of the version control system **git** to access and submit your work.
- To become familiar with the lab and the **CATe** submission tracking processes.

Submit by 19:00 on Friday, 8th October

The process

Each week you will be given a laboratory exercise that will be marked and fed back to you at your weekly PPT meeting. The process of completing a laboratory exercise consists of three main steps:

1. Getting the skeleton files for the exercise with git.
2. Implementing and testing your solution to the exercise, including *checkpointing* your work with git as you go along, and committing your work for *autotesting* when you think you have it right.
3. Submitting from the LabTS system to the online coursework administration system (CATe) so that your work, and the marks awarded for it, are correctly archived.

In this first exercise, you will work through the three sections above in detail using a few very simple functions as examples. After this, there are some additional questions that will give you the opportunity to go through the submission cycle again so that you get the hang of it.

Before you proceed, you must first log into the GitLab server at:

`https://gitlab.doc.ic.ac.uk`

using your normal college username and password.

1. Getting the skeleton files

The skeleton files for this exercise can be acquired using *git*, which is a powerful, industry-standard, tool for managing version control. However, we will cover all the basic commands you will need for this exercise as we go along.

A git repository for this exercise has been set up for you on the departmental GitLab[1] server. You will need to *clone* this to your DoC home directory in order to work on it. Later, you will need to push your changes back to the GitLab server.

On a DoC Linux system[2], log in and open a terminal. You can get your skeleton repository by issuing the following command at the terminal prompt (usually a `>` symbol):

```
git clone https://gitlab.doc.ic.ac.uk/lab2122_autumn/haskellsequences_username.git
```

This will create a directory `haskellsequences_`*`username`* in your home directory where *`username`* is your user name.

You can list the contents of this directory using the Linux `ls` (list files) command, *viz.* `ls haskellsequences_username`. You can switch into the directory using the Linux `cd` (change directory) command and then list the files by typing `ls`, or `ls -A`, which lists the contents of the current directory (the '`-A`' is an option that means list *all*):

```
> cd haskellsequences_username
> ls -A
IC Sequences.hs Tests.hs .git .gitignore
```

The files contained in the directory are as follows:

- `Sequences.hs` - this is the source file you should edit to define the functions required for this exercise.
- `Tests.hs` - this file contains a small test suite that you can use to help test your implementations. You may edit this file to add further test cases if you wish.
- `IC` - this directory contains some supporting code for the test suite. You should not need to make changes to the code in here.
- `.git` - this directory contains information for git to be able to track your changes locally. You should not normally need to edit the files in here.
- `.gitignore` - this file tells git what files it should ignore. We have set it up with some sensible defaults, but you are free to edit it if you need to.

[1] `https://gitlab.doc.ic.ac.uk`

[2] Note that if you wish to work from home (or some combination of home and college), the given git commands should work from outside college too. However, to continue working from a different location after making further changes, you may need to investigate the commands `git fetch` and `git merge`. If you need additional support, ask a lab helper and look out for an upcoming programming tools lecture on git to find out more.

It is really, really, really bad practice to try to send lab work to/from home using sharing systems such as Dropbox or (much worse!) e-mail. Git allows you to keep track of your changes and can scale from single to multi-user projects – something which will happen next term. It is worth spending the time now to get used to it.

2. Implementing and testing your solution

In order to define the required functions, you will need to open the file `Sequences.hs` in a text editor. You can use a graphical one that comes with Ubuntu (e.g. gedit), or a terminal-based one, like vim. In order to help you get started with Linux, DoCSoc have prepared a "Guide To Linux And The Command Line", which you can find under the "Notes" section of COMP40009 on Materials.

In a terminal you should also load the `Sequences` module into `ghci` (do not forget to `cd haskellsequences_username` first, if you open a new terminal):

```
> ghci Sequences.hs
GHCi, version 8.6.5: http://www.haskell.org/ghc/  :? for help
[1 of 1] Compiling Sequences        ( Sequences.hs, interpreted )
Ok, modules loaded: Sequences.
*Sequences>
```

In your text editor, you should first complete the definition of `maxOf2` so that it returns the biggest of two integers. Note that this has similar behaviour to the more general built-in function `max` (try it!), but it is instructive at this point to define your own version. Thus, change the given definition to the following:

```
maxOf2 :: Int -> Int -> Int
-- Returns the first argument if it is larger than the second,
-- the second argument otherwise
maxOf2 x y
  | x > y     = x
  | otherwise = y
```

Note that lines beginning with `--` are comments. You do not have to enter those exact comments, but **you should now get into the habit of commenting all your work**. Larger programs, in particular, should be liberally commented to show what they are supposed to do and to help yourself and others to understand and maintain them.

Make sure you have saved these definitions in the text editor, and then ask `ghci` to reload the file by issuing it the command `:r` (short for `:reload`):

```
*Sequences> :r
[1 of 1] Compiling Sequences     ( Sequences.hs, interpreted )
Ok, modules loaded: Sequences.
*Sequences>
```

If there is anything wrong with the Haskell script, `ghci` will print out a description of the error and its location. You will need to fix these errors in the text editor and then reload the file, before you can proceed.

The script `Sequences.hs` now contains the declaration and definition of the function `maxOf2`. Evaluate this function with a number of different values. Try running the following in ghci:

```
maxOf2 1 2
maxOf2 2 1
```

```
maxOf2 2 2
```

If you look at `Tests.hs`, you will notice that these examples correspond to the given test cases for `maxOf2`. If you open another terminal, you can run this test suite using `runghc`.

```
> cd haskellsequences_username
> runghc Tests.hs
```

You should see a lot of test case failures for the functions you have yet to write, but hopefully if you look at the top you will see the tests for `maxOf2` have passed.

Do not forget to test your function in `ghci` to make sure it compiles and works. Also, you can add any test cases you come up with to test this function to the `sequencesTestCases` part of `Tests.hs`, so it is quick and easy to rerun your tests in the future.

Now, using *only* `maxOf2`, define the function `maxOf3` that returns the biggest of three integers:

```
maxOf3  :: Int -> Int -> Int -> Int
```

As an exercise try to define these functions using nested conditionals (`if ... then ... else ...`), e.g. using the operators >= and &&. Notice how much easier it is to solve the problem in terms of higher-level functions like `maxOf2` rather than >= and &&. This is your first lesson in abstraction; remember this always!

What test cases could you use to check that `maxOf3` is correctly defined?

In addition to `Int`, `Float` and `Bool`, Haskell provides the built-in type `Char` for processing characters. Characters are represented using **unicode** numerical values, which is a superset of the ASCII encoding standard. Hence there is a distinction between the character `'7'` and the number `7`; they have different types so, for example, `'7' + 6` will give a type error.

The ASCII numbers have been designed so that the encoding of consecutive lower-case letters `'a'..'z'`, consecutive upper-case letters `'A'..'Z'`, and consecutive digits `'0'..'9'` are contiguous, e.g. `ord 'a'` = 97, `ord 'b'` = 98 etc. However, if you look before, after and in-between these sets, you will find other character symbols living there. All characters are ordered based on their numerical encoding, so in Haskell you can write something like `'Z' > 'A'`, which gives the result `True`.

Haskell's `Data.Char` module includes two functions, `ord` and `chr`, for converting characters to numbers and vice versa, so that `chr (ord 'a')` gives the result `'a'`. You can import modules into a program using the `import` keyword, hence the line `import Data.Char (ord, chr)` at the beginning of your script which loads *just* the `ord` and `chr` functions from the module `Data.Char`. If you omit the bracketed term then *all* the functions from `Data.Char` will be imported.

As an exercise, you are now going to provide your own definitions of some of the other functions defined in `Data.Char`. Note that if you import all of the functions of `Data.Char`, then GHCi will complain that your definitions clash with ones it can already see. That is why you should import just `ord` and `chr`.

Define the following functions:

```
isDigit :: Char -> Bool
-- Returns True if the character represents a digit '0'..'9';
```

```
-- False otherwise

isAlpha :: Char -> Bool
-- Returns True if the character represents an alphabetic character
-- either in the range 'a'..'z' or in the range 'A'..'Z';

digitToInt :: Char -> Int
-- Pre: the character is one of '0'..'9'
-- Returns the integer [0..9] corresponding to the given character.
-- Note: this is a simpler version of digitToInt in module Data.Char,
-- which does not assume the precondition.

toUpper :: Char -> Char
-- Returns the upper case character corresponding to the input.
-- Uses guards by way of variety.
```

To define the above functions you will need to use `ord` and `chr` for type conversions. Hint: `ord '7' - ord '0'` gives the result `7`.

Checkpointing your work with `git`

As you develop your code you are strongly encouraged to add and commit changes frequently, in case you need to "undo" something and revert back to an earlier version. We will not attempt to document all of git's version control features here – you will learn much more about them later on. You will soon become an expert! For the time being, here is how to commit a change...

In a terminal that is in the `haskellsequences_username` directory, use `git status` to see what has changed from the skeleton:

```
> git status
On branch master
# Changes not staged for commit:
#   (use "git add <file>..." to update what will be committed)
#   (use "git checkout -- <file>..." to discard changes in working directory)
#
#       modified:   Sequences.hs
#       modified:   Tests.hs
```

If you have not modified `Tests.hs` then it will not appear in the output as a modified file. You can tell git you care about changes to these files using `git add`:

```
> git add Sequences.hs Tests.hs

> git status
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
```

```
#
#       modified:   Sequences.hs
#       modified:   Tests.hs
```

To store these changes in git, you now need to *commit* them with a descriptive message; for example

```
> git commit -m "Implementation of utility functions"
[master 6b2906c] Implementation of utility functions
 Committer: Your Name <username@imperial.ac.uk>
Your name and e-mail address were configured automatically, based
on your username and hostname. Please check that they are accurate.
You can suppress this message by setting them explicitly:

    git config --global user.name "Your Name"
    git config --global user.email you@example.com

After doing this, you may fix the identity used for this commit with:

    git commit --amend --reset-author

 2 files changed, 14 insertions(+)
```

Alternatively, you can type `git commit` without any argument and a text editor window will pop up to let you write a short, but meaningful, description of the changes you have made.

Autotesting

The laboratory has an automated testing service system called LabTS. In order to invoke the autotester you need to send your modified files back to the GitLab server. Doing this also means that they are backed up independently and are available to access from home. To do this, issue the command `git push`:

```
> git push
Username for 'https://gitlab.doc.ic.ac.uk': <enter your username>
Password for 'https://<username>@gitlab.doc.ic.ac.uk': <enter your password>
Counting objects: 7, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (4/4), done.
Writing objects: 100% (4/4), 4.00 KiB, done.
Total 4 (delta 2), reused 0 (delta 0)
Unpacking objects: 100% (4/4), done.
To https://gitlab.doc.ic.ac.uk/lab2122_autumn/haskellsequences_username.git
   50db2c6..97cd58f  master -> master
```

You can check that the push succeeded by looking at the state of your repository using the GitLab webpages.

To autotest your submission log into `https://teaching.doc.ic.ac.uk/labts` and look for

your **Haskell Sequences** exercise. On its repository page, you will see the commits you have pushed and buttons for requesting an autotest, as well as a link to submit your work to CATe (see below). You should request an autotest for your final work – it should take about 5 minutes to give you back a result. When it gets tested, **check that the resulting "preview" pdf file is what you want to be marked**.

It is your responsibility to ensure your work compiles and runs on this system before the deadline. If there are any problems with this please seek out a lab helper or the laboratory co-ordinator **before the deadline**. While we will be forgiving during the First Year laboratories as you are getting used to the system, in other courses – and especially in later years – **work that fails to compile or run on the autotester, where we have not been contacted beforehand, will score 0**.

3. Submitting to CATe

When you are happy with your submission you need to submit it to the Department's Continuous Assessment Tracking engine (CATe). This is done by clicking the "Submit to CATe" button on your `labts` repository page (see above).

This will redirect you to the declaration page. This page allows you to declare that your submission was your unaided work and to acknowledge anyone who has helped through original discussions. If you have done the lab in the usual way and you have not had any significant help then you can click on **"Submit declaration"** and return a blank declaration. Otherwise you should fill it in and submit it.

After you have submitted the declaration you will see a new page where your commit version ID has been uploaded and submitted.

Note: you can submit again if you want (so long as it is still in by the due date), for example if you realise you need to correct something. A later submission will overwrite and supersede any earlier ones. You can practice this when you complete the mini-exercise that follows below.

Important: Please take extra care to submit the correct revision ID for your exercises.

You can check which version of your work was submitted to CATe on your LabTS repository page. The corresponding commit submission button will have been replaced by a green confirmation label.

Please note that the "preview" PDF shows the tests and your code in exactly the same format as your UTA will see.

You can check that you really have committed everything by doing a `git status`:

```
> git status
# On branch master
nothing to commit (working directory clean)
```

You can also check the state of your git repository using e.g. `git log`, or a repository browser such as `gitk` or `gitg` (try running these commands in a terminal that has `cd`'d into

the `haskellsequences_username` directory). You can also check the state of your `GitLab` repository using the GitLab webpages:

`https://gitlab.doc.ic.ac.uk`

Your submitted labwork and its automated test results will be made available to your PPT UTA who will mark it and return it to you in your next PPT session.

Test your program thoroughly before submission as some marks are awarded on the basis of automatic test runs of your submitted labwork. You should test your program on a wide range of inputs to make sure it works as specified and should **not** assume that the autotest suite is complete, or the same as that used by your UTA. Your PPT will deduct marks from programs with errors, especially if those errors could have been detected easily by simple tests.

Also, it is critically important you produce code that is clear, well laid out, and uses sensible variable names. Failing to do this will make it hard to get a grade above a B (see below) for your laboratory exercises.

That's it! Normally at this point you will be done. However, to give you some more practice with submissions, we have given you an additional 'mini' exercise below. Implement the following by expanding the code you have already written and re-submit the updated version to your git and LabTS/CATe, as you did above.

Mini exercise: Sequences

You will now implement the explicit formula of some well known sequences and series.

- An arithmetic sequence is a sequence $(u_n)_{n\in\mathbb{N}}$ recursively defined by $u_0 = a$ for a given *initial term* $a \in \mathbb{R}$ and for all $n \geq 1$, $u_n = u_{n-1} + d$ for a given *common difference* $d \in \mathbb{R}$. For example, the first terms of the arithmetic sequence of initial term 0 and common difference 3 are $0, 3, 6, 9, 12, 15, \cdots$. You can display those terms in Haskell using the special syntax for sequences:

  ```
  *Sequences> [0.0,3.0..15.0]
  [0.0,3.0,6.0,9.0,12.0,15.0]
  ```

  It can be shown that the explicit formula for the n^{th} term of such a sequence can be written:

  $$u_n = a + n\,d$$

  Based on that formula[3], define the function `arithmeticSeq :: Double -> Double -> Int -> Double` that takes as arguments two `Doubles` a and d defining a sequence and an `Int` n and returns the n^{th} term of the sequence. Note that the 0^{th} term is a.

  Once you have implemented the formula, you can check in ghci that the following gives you the same result as above:

[3] You can convert an `Int` into a `Double` using the built-in function `fromIntegral`.

```
*Sequences> [arithmeticSeq 0 3 n | n <- [0..5]]
[0.0,3.0,6.0,9.0,12.0,15.0]
```

- Similarly, a geometric sequence is a sequence $(u_n)_{n\in\mathbb{N}}$ recursively defined by the initial term $u_0 = a$ for a given $a \in \mathbb{R}$ and for all $n \geq 1$, $u_n = r\,u_{n-1}$ for a given common ratio $r \in \mathbb{R}$. It can be shown that the explicit formula for such a sequence is, for $n \in \mathbb{N}$:

$$u_n = a\,r^n$$

Thus, define a function `geometricSeq :: Double -> Double -> Int -> Double` that returns the n^{th} term of such a sequence. The first 5 terms of the sequence with initial term 1 and common ratio 2 should give the following result:

```
*Sequences> [geometricSeq 1 2 n | n <- [0..4]]
[1.0,2.0,4.0,8.0,16.0]
```

- An arithmetic series with initial term a and common difference d is a sequence $(v_n)_{n\in\mathbb{N}}$ defined for all $n \in \mathbb{N}$ by $v_n = \sum_{k=0}^{n} u_k$, where u_k is the k^{th} term of an arithmetic sequence with same initial term and common difference. Its explicit *closed form* for $n \in \mathbb{N}$ is:

$$v_n = (n+1)\,(a + \frac{d\,n}{2})$$

Thus, define a function `arithmeticSeries :: Double -> Double -> Int -> Double` that will compute v_n, given a, d and n. You can check your implementation using `arithmeticSeq` above and the built-in function `sum`:

```
*Sequences> arithmeticSeries 0 3 5
45.0
*Sequences> sum [arithmeticSeq 0 3 n | n <- [0..5]]
45.0
```

- Similarly, a geometric series with initial term a and common ratio r is a sequence $(v_n)_{n\in\mathbb{N}}$ defined for all $n \in \mathbb{N}$ by $v_n = \sum_{k=0}^{n} u_k$, where u_k is the k^{th} term of an geometric sequence with same initial term and common ratio. Its explicit formula for arbitrary $n \in \mathbb{N}$ is given by:

$$v_n = \begin{cases} a(n+1), & \text{if } r = 1 \\ a\frac{1-r^{n+1}}{1-r}, & \text{otherwise} \end{cases}$$

Thus, define a function `geometricSeries :: Double -> Double -> Int -> Double` that will compute v_n, given a, r and n. Again, you can test your implementation similarly to the above:

```
*Sequences> sum [geometricSeq 1 2 n | n <- [0..4]]
31.0
*Sequences> geometricSeries 1 2 4
31.0
```

Final note: logging out

If you are using one of our lab machines, please remember to **log out** at the end. If you have any problems then please **ask the lab helpers** to help you.

Assessment

In general, the assessment for laboratory exercises uses the following scheme:

```
F - E: Very little to no attempt made.
       Submissions that fail to compile cannot score above an E.

D - C: Implementations of most functions attempted;
       solutions may not be correct, or may not have a good style.

B:     Implementations of all functions attempted, and solutions
       are mostly correct. Code style is generally good.

A:     There are no obvious deficiencies in the solution or
       the student's coding style. In addition, there is
       evidence of productive testing.

A*:    As for an A -- plus the student has done additional work
       beyond the basic spec, e.g. by considering (and clearly
       commenting) interesting variations or extensions to the
       given functions; e.g. based on their own research.
```