Practical 2 CS2PLC Programming Language Concepts
Assessed: **Yes** (At start of Practical 3 in week3)

Practical 2

Theme

- Introduction to Haskell (syntax and usage)
- Introduction to functional programming in and Haskell
- Comparison of functional and imperative programming
- Passing functions as arguments in Haskell, Python3 and Javascript

Key Concepts: Haskell and JavaScript syntax, functional programming, imperative programming

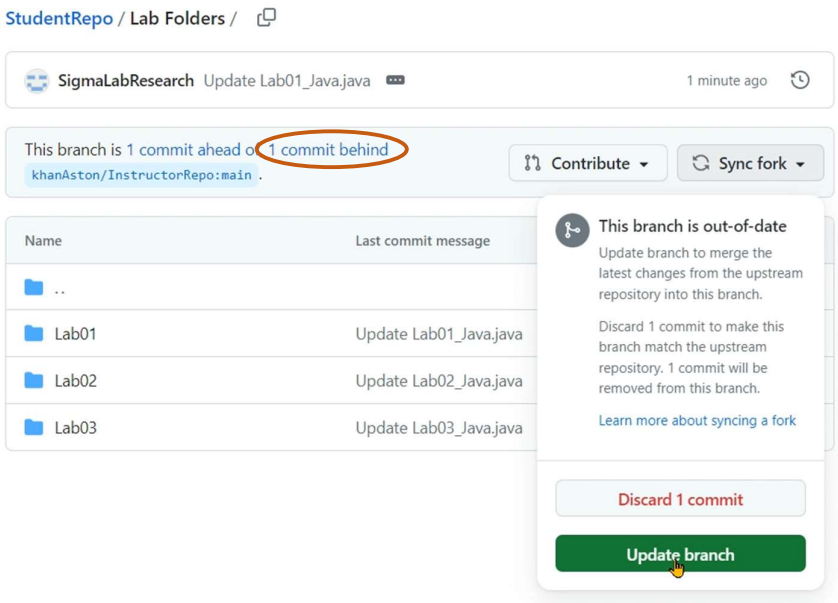
---

1) Start up and essential configuration

a) *Open PLC Labs GitHub repo in Gitpod*

If you plan to save changes to your work, then sign-in to your GitHub account and open your (forked) version of the PLC2025 Git repo in GitPod by appending 'gitpod.io/#' to start of your repo's URL and pressing enter.

If your repo is some commits behind the instructor repo then click on 'Sync Repo -> Updated Branch' to retrieve any new changes made to the instructor repo.



If you do not plan to save changes to your work on GitHub, then go directly to the instructor's Git repo (khanAston/PLC2025) and open it in GitPod by appending 'gitpod.io/#' to start of your repo's URL and pressing enter.

b) *Open Lab02*

Locate the folder `lab2- functionalprimer`, in the PLC2025 GitHub repository. Have the PLC_Cheatsheet handy for reference (File Location: '`Lab01 Part02 Executing Demo Code for Different PLs/PLC_Cheatsheet.pdf`').

2) Basics of Haskell

All practice code in this section can be found in the intro files (e.g `haskell/intro1.hs`)

Practical 2 CS2PLC Programming Language Concepts
Assessed: **Yes** (At start of Practical 3 in week3)

Practice

a) Start by running an existing program **`'intro1.hs'`** by entering

```
runhaskell intro1.hs
```

in the terminal window. **NOTE:** Before running, make sure you're at the correct path in the terminal: `lab2-functionalPrimer/haskell`
If you don't get an error after pressing enter then CONGRATULATIONS! You just successfully executed your first Haskell program.

b) The program ran but nothing happened. To interact with the functions and variables contained in `intro1.hs` you will need to load it using the `ghci`[1]. which can be loaded from `terminal` as below

```
ghci
```

```
:l intro1.hs
```

c) Enter names of different constants, functions and lists to observe their values in the `terminal`. For example, entering `r1` in the terminal prints `1`.

d) Type signatures of different entities can be examined by using `:t` command in `ghci` as below

```
:t r1
```

The above command should output the text below

```
r1 :: Integer
```

e) Feel free to play around with `intro1.hs` and make small modifications. Try resolving any errors you observe by using posts on code forums and online haskell documentation (I recommend spending 5-7 mins on this activity)

f) Try looking up documentation on the `map()` function in haskell and try to understand what it does (I recommend spending 5 mins on this activity). Does this remind of something similar in other languages like Python?

g) You can quit the ghci by entering `:q` in the terminal.

h) Now move to **`'intro2.hs'`** load and run this file and play around with the code.

i) Read the comments in this file and try to understand how it works. If something is not clear, try searching for answers on code forums and online documentation (Recommended time: 20 mins).

j) Interact with `toString, greet1` and other functions written below the `main` function in the `ghci` and try to understand how they work (5 mins).

k) Functions (methods) in Haskell are written slightly differently to how write methods in Java; however, Haskell's way of writing functions is similar to how mathematical functions are written. For example, the function `inc` that increments a number input to it is defined as below:

```
inc x = x + 1
```

You write:

- The name of the function
- The argument
- Definition/ body of the function

[1] The "Interactive" Glasgow Haskell Compiler

Before defining a function, you should also preferably include its **'type signature'**; however, this is not a mandatory requirement and Haskell will infer it based on the function definition if you do not include it. The type signature for the `inc` function is as follows:

```
inc :: Int -> Int
```

The preceding type signature tells anyone else reading your code that the `inc` function maps an integer (input) to an integer (output).

Tasks

1. *Writing Haskell **functions***
   Open the file `task1.hs` and uncomment the statements on lines 6-7 and 10-11. Running this file will throw an error. Read the section related to Haskell functions at the following link http://learn.hfm.io/first_steps.html and see if you can resolve the syntax errors in the `task1.hs` file. Answer **Review Quiz Q1** related to the above task.

2. *The **do** notation*
   The do notation gives us a convenient way of putting actions together. Take a look at the following code

```
1  main = do
2      putStrLn "Welcome to the programme. Please enter your name"
3      name <- getLine
4      putStrLn("Hello " ++ name ++ ", hope you like Haskell.")
```

   The `do` notation allows you to build programs with structure similar to imperative languages, with functions being called from the main. See `task2.hs` for code.

   Write a simple function called `onePlusone` on top of the main. This function should print the string "`1 + 1 = 2`" in the terminal. Call the onePlusone from `main`. Answer **Review Quiz Q2** related to this task.

3. ***Conditionals** in Haskell*

   The `sgn` function below maps all negative values to -1, and all positive values to +1, 0 remains 0. Check the output of this function for the following arguments: 10, 5, 0, -5.

```
1  sgn x = if x < 0 then -1 else if x == 0 then 0 else 1
2
3  main = do
4      putStrLn "Please enter a number"
5      input <- getLine
6      let x = (read input :: Int)
7      putStrLn( show (sgn (x)) )
```

   Writing conditionals like above, in Haskell can be cumbersome. Haskell also provides **Guards** as alternatives which are more readable compared to conditionals. Read up more on guards in the "Branches in the Control Flow" section of the tutorial at the following link: http://learn.hfm.io/fundamentals.html (NOTE: You do not need to read the entire tutorial

from top to bottom to solve the upcoming task. However, feel free to read it all if it interests you).

Open `task3.hs` and rewrite the `sgn` function using **guards**. Answer **Review Quiz Q3** related to this task.

4. ***Looping*** *using Recursion*

Haskell incorporates loops using recursion i.e., a function repeatedly calls itself until a terminating condition is met. This can take some getting used to, `task4.hs` contains a programme that uses recursion to implement something similar to a while loop.

```
1  ask :: String -> IO ()
2  ask prompt =
3    do
4    putStrLn prompt
5    line <- getLine
6    if line == ""
7      then ask prompt
8      else putStrLn ("you said: " ++ reverse line)
9
10 main :: IO ()
11 main =
12   do
13   let prompt = "please say something"
14   ask prompt
```

i) First, compile and execute the program using the command:
`runhaskell tasks.hs`
(Reminder: Make sure you `cd` to `haskell` folder first)
You will be prompted to "say something" and if you do, it is echoed. If you press `Enter` without typing anything, the prompt will repeat.

ii) Alter the above code so that when someone inputs "quit" and then presses `Enter`, the system will exit after displaying "quitting. . . ". HINT: Look at line-1 of the programme under the heading '3. Conditionals in Haskell' for a reference on how to write conditionals in Haskell.
Answer **Review Quiz Q4** related to this task.

iii) Alter the code so that when someone presses enter without typing anything, the repeated prompt will have an extra "!" at the end. With each further repeat, there should be an additional "!". A sample output is given below

```
> please say something

> please say something!

> please say something!!
```

Answer **Review Quiz Q5** related to this task. HINT: Think about the `prompt` argument being passed to the `ask` function during recursive calls.

3) Functional Features Haskell, Python3 and JavaScript

Over time, a number of imperative PLs have also incorporated features from functional PLs like Haskell. For example, Python3 and JavaScript do allow you to (somewhat easily) pass functions

as arguments to other functions. To illustrate this, we start off with the sum and average example covered during the lecture01.

```haskell
--create inpFunc
inpFunc = [1..5]

--Define applicatorFunc
applicatorFunc inpFunc s = if s=='s' then sum inpFunc else (sum inpFunc)/5

main = do
    let result = applicatorFunc inpFunc 'a' --Call applicatorFunc with inpFunc and 'a' as args
    putStrLn("sum = " ++ show(result))
```

The equivalent of the preceding function in Python3 is given below:

```python
def listFunc():
    #Create list of ints from 1 to 5, Haskell equivalent [1..5]
    return [i for i in range(1, 6)]

def applicatorFunc(inpFunc, s):
    if s=='s':
        return sum(inpFunc())
    else:
        return sum(inpFunc())/5

print(applicatorFunc(listFunc, 's'))
```

The Javascript equivalent is given below:

```javascript
//Create list of ints from 1 to 5, Haskell equivalent [1..5]
function arrFunc(){
    let arr = [];
    for (let i = 1; i<=5; i++) {
        arr.push(i);
    }
    return arr;
}

function applicatorFunc(inpFunc, s){
    if(s=='s'){
        const arr = inpFunc();
        let sum = arr.reduce((accumulator, currentValue) => accumulator + currentValue, 0);
        return sum;
    }
    else{
        const arr = inpFunc();
        let sum = arr.reduce((accumulator, currentValue) => accumulator + currentValue, 0);
        return sum/5;
    }
}

let x = applicatorFunc(arrFunc, 's');
console.log(x);
```

Tasks

1. *Passing functions as arguments* ***in Haskell***

   Open the file `func_as_arg.hs` in the Haskell folder.

   i) Write a more generalised version `func_as_arg.hs` which can compute the sum/average for any set of integers instead of just 1 and 5. HINT: Replace `[1..5]` with

`[a..b]` where, a and b represent any integers a and b. Answer **Review Quiz Q6** related to this task.

ii) Rewrite the generalised version of `func_as_arg.hs` using guards. Answer **Review Quiz Q7** related to this task.

2. *Passing functions as arguments **in Python3***

   Open the file `func_as_arg.py` in the python3 folder.

   Write a more generalised version `func_as_arg.py` which can compute the sum/average for any set of integers instead of just 1 and 5. HINT: Replace `[1..5]` with `[a..b]` where, a and b represent any integers a and b. Answer **Review Quiz Q8** related to this task

3. *Passing functions as arguments **in Javascript***

   Open the file `func_as_arg.js` in the javascript folder.

   Write a more generalised version `func_as_arg.js` which can compute the sum/average for any set of integers instead of just 1 and 5. Answer **Review Quiz Q9** related to this task

End of Lab02