G6021: Comparative Programming

Exercise Sheet 1

1 Starting Haskell

The aim is to make a start with using the Haskell system. We will use GHCi, which is an interactive environment (interpreter), and is part of a larger toolset known as the Glasgow Haskell Compiler. Haskell expressions can be interactively evaluated and programs can be interpreted. Follow the examples below then try the questions. In the labs you will occasionally find things that have not yet been covered in the lectures—in these cases, try to work out what to do (guess, ask, or alternatively come back to them later).

- Locate and run the Haskell interpreter: GHCi.

  There is also an online system you can use: `https://replit.com/languages/haskell` in case you have problems running the local version.

  You can also install Haskell on your own computer: `https://www.haskell.org/ghc/`.

- When you run the system, a window should open, and the Haskell system should initialise, leaving a prompt:

  ```
  Prelude>
  ```

  Think of this as a sophisticated calculator where you can type expressions that you want evaluated. A number of functions are built-in, and we will make use of some of these.

- Enter some expressions, including some with syntax errors. Start with these for example:

  ```
  > 6
  > 3+4*5
  > it
  > 'c'
  > True␣&&␣False
  > "this␣is␣a␣string"
  > (3,4)
  > fst␣(3,4)
  > snd␣(3,4)
  > (1,2,3,4,5,6,7,8,9,10,11,12,13,14,15)
  > [1,2,3]
  > [1+2,2+3,3+4]
  > head␣[1,2,3]
  > tail␣[1,2,3]
  > 4␣+
  > 4␣+␣True
  > fst␣(1,2,3)
  ```

```
> (1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16)
```

The last four give an error: can you understand the nature of the errors? Make a few other errors to see the kind of error messages that are given.

Answer:

- `+` is an infix operator needing two arguments, which it hasn't in `4␣+`
  note: `(4␣+)` does parse (it's a *section*) but denotes a function so can't be shown
- `+` expects its arguments to be numbers; `True` is not in `4␣+␣True`
- `fst` expects it argument to be a pair, not a triple
- showing tuples is defined (arbitrarily) by default only for tuples up to length 15

When developing a Haskell script, it is useful to keep two windows open, one running an editor for your program, and the other running the interpreter. There are a number of ways to do this: find a way that you like best.

- Start an editor (of your choice), type in the following function, and save the script as `test.hs`

```
double x = x + x
```

- At the Haskell prompt, enter the command `:load test` (`:l test` is an abbreviation). If you made any errors, correct these. You can now use these functions. Type some expressions to test your new function: `double 10`, etc. You will need to make sure that Haskell is looking in the correct folder for this to work correctly.
- In your editor window, add another function so that you have:

```
double x = x + x
quad x = double (double x)
```

Depending on how you are using the Haskell system, GHCi may not automatically detect that the script has been changed, so a reload command must be executed before the new definitions can be used. You can do this with the command `:reload` (`:r` is an abbreviation). You can now test your new functions: `quad 10`, etc.

You now have all that you need to be able to do the following exercises.

1. Write a function `square` that will take one argument (an integer) x and compute the square x^2 (try both with and without using the "^" operator).

   Answer:

   ```
   square x = x*x
   square x = x^2
   ```

2. Write a function `i` that just returns the argument (this is called the identity function). `i 4` should give 4. What is the value of `i "hello there!"` ?

   Answer:

   ```
   i x = x

   >i "hello there!"
   "hello there!"
   ```

3. The previous functions all had one parameter, and we can use them by giving an argument. For example, `square 4`, etc. What happens if you do not give an argument, or you give too many? Try the following: `square`, `square 4 5`, `square "hello there"`. Try to understand what Haskell is telling you in the error messages. We can write functions with several parameters, or none as we see below.

4. Write a function `firstone` that will take two arguments and return the first only: `firstone 3 4` should give 3. Next write `secondone` that gives the second argument only. What happens if you do not give two arguments, for example: `firstone 3`?

   Answer:

   ```
   firstone x y = x
   secondone x y = y
   ```

5. Define two functions:

   - `fortytwo x = 42`
   - `infinity = infinity + 1`

   Note: Control-C might be needed to stop the computation if you test `infinity` alone. What is the value of `fortytwo infinity`? What does this tell you about the evaluation strategy of Haskell?

   Answer: 42
   Therefore the reduction is not call-by-value.

6. Write a function `apply` that will take two arguments, one at a time, and return the first argument applied to the second. `apply fortytwo 3` should give 42. `apply square 3` should give 9, etc.

   Answer:

   ```
   apply f x = f x
   ```

7. Write a function `twice` that will take two arguments, one at a time, and return the first argument applied to the second twice. `twice square 2` should give 16, `twice square 3` should give 81, etc.

   Answer:

   ```
   twice f x = f(f x)
   ```

8. Write a recursive function `fibonacci` that will compute the following mathematical function:

$$\begin{array}{lcl} f(0) & = & 1 \\ f(1) & = & 1 \\ f(n) & = & f(n-1) + f(n-2) \text{ (if } n > 1) \end{array}$$

   Give several versions of this function: one using pattern matching, one using if-then-else, and another using conditional equations (guarded equations). [Hint: look ahead in the notes to see what this means.]

   Answer:

```
fib 0 = 1
fib 1 = 1
fib n = fib (n-1)+ fib(n-2)

fib1 n = if n<2 then 1 else fib1(n-1)+fib1(n-2)

fib2 n
 | n<2 = 1
 | otherwise = fib2(n-1)+fib2(n-2)
```

If you have more time:

- Type `:?` and look at the output.
- Look at some of the examples on `www.haskell.org`, and try some of them.

2 Optional (for fun): comparison of languages

Below are three programs, written in three different languages. Examine these, then answer the questions below.

Program 1:

```
insertion_sort(Xs, Ys) :- insertion_sort_1(Xs, [], Ys).
insertion_sort_1([], Ys, Ys).
insertion_sort_1([X|Xs], Ys0, Ys) :- insert(Ys0, X, Ys1), insertion_sort_1(Xs, Ys1, Ys).
insert([Y|Ys], X, [Y|Zs]):- Y < X, !, insert(Ys, X, Zs).
insert(Ys, X, [X|Ys]).
```

Program 2:

```
insert e [] = [e]
insert e (x:xs)
  | e < x     = e : (x:xs)
  | otherwise = x : (insert e xs)

isort [] = []
isort (x:xs) = insert x (isort xs)
```

Program 3:

```
void iSort(int a[], int len){
 int i, j, v;
 for(i = 1; i < len; i++) {
  v = a[i];
  for(j=i-1; j>=0 && a[j]>v; j--)
   a[j + 1] = a[j];
  a[j + 1] = v;
 }
}
```

1. Identify the programming language used for each program.
2. Try to find a compiler or interpreter (either in the lab or on your own laptop/desktop, etc.) that can execute these programs, so that you can test them. You might have to write some additional code to complete the programs so that it will compile and run.
3. Think about comparing them, and then decide which one you think is the best way to represent this algorithm.
4. Send an email to `vvo@sussex.ac.uk` listing the three programming languages, the compilers that you managed to use, and your opinion about which is the best language of these for this algorithm (only 1-2 sentences needed). The best answer will win a prize.