Exercises

1 Quickstart

Install GHC the Glasgow Haskell Compiler.

To run the interpreter, type `ghci` in a terminal/command window.
To get help in the interpreter try `:h`
To see the type of something write `:t` before it
To quit type `:q` and press enter.

1.1 More commands

To load in a program from a file `someprogram.hs`, type `:l someprogram` if the interpreter was started in that directory.

To see a list of the functions you loaded, type `:browse Main`

2 Exercises

There are many exercises and you should complete as necessary to understand each category, so if something is too easy, try skipping ahead to the last few exercises in the category and see if you can solve those. You can always come back later if you don't understand an exercise.

Sometimes I may write some formula that displays more nicely in the pdf file, you will notice the $'s, also sometimes you will see ~'s which just means the following should be read as code. Hopefully the document is still understandable, though. I apologize in advance.

2.1 Basics, interpreter, simple functions and lists

2.1.1 Arithmetic

In the interpreter try some of the following things, make sure you think about what you expect the result to be before trying it in the interpreter, feel free to experiment with more examples.

- Adding numbers
- Subtracting numbers
- Multiplying numbers

- Multiplying negative numbers
- Dividing some floating point numbers
- Dividing some integer numbers that divide evenly, like `10/2`
- Adding together the result of some multiplications

2.1.2 Boolean logic

In the interpreter try some of the following things, make sure you think about what you expect the result to be before trying it in the interpreter, feel free to experiment with more examples.

- Seeing what the value of `True` and `True` is
- Seeing what the value of `False` and `True` is
- Seeing what the value of `False` or `True` is
- Seeing what the value of the negation of `False` or `False` is
- Seeing if `True` is equal to `True`
- Seeing if `True` is different from `True`
- Seeing if `True` is different from `False`
- Seeing what the value of other expressions are e.g.
  - What is the value of `2+3 == 5` ?
  - What is the value of `4 <= 2` ?
  - What is the value of `"hi" /= "hello"` ?
  - What is the value of `pi == 3.141592653589793` ?

2.1.3 Dangerous territory

What happens if you try to add a number and a string?

What happens if you try to add two strings? (Look at the list section to see how to actually join strings)

What happens if you try to compare things of different types? e.g. `3 < "4"` or `"True"` == `True`

What happens if you try to add an integer and a floating point number?

2.1.4 Simple functions

In the interpreter try some of the following things, make sure you think about what you expect the result to be before trying it in the interpreter, feel free to experiment with more examples.

- Using `succ` on an integer
- Using `min` or `max` on two numbers
- Using `succ`, `min` and `max`, in a large expression, both with and without parentheses

2.1.5 Your own function

Write a function that doubles a number, in a small file, e.g. `funcs.hs`. Then load it in the interpreter using `:l funcs`, and test that it works properly. If it doesn't work make sure your interpreter is working from the same directory your file is located. You can check the current working directory in the interpreter by writing `:! pwd` if you're on Linux(/Mac?) or `:! cd` if you're on Windows, and use `:cd some-directory` to move around.

Write a function that adds the doubles of two numbers, you are welcome to reuse your function from before.

Write a function that doubles a number only if it is a smaller than 100, otherwise it doesn't change it.

Try changing the function by putting the entire if-then-else statement in parentheses and add 1 to that.

For many exercises from now on, I recommend writing the program in a file and loading it in. But if you want to try something out, always feel free to try it in the interpreter first. One thing to remember is to write `let` before definitions when doing things in the interpreter. E.g. `let x = 42`. You may not write `let` in a program in a file.

2.1.6 Lists

1. Creating and combining
    - Create some lists with numbers
    - Create some lists with strings
    - Create a list of single characters, using singlequotes e.g. `'x'`
    - Try creating new lists by concatenating lists, using `++`
        - What happens if you try to concatenate the list of characters with a string?
    - Try adding a single element to the beginning of a list, using `:`
2. Indexing

- What is the syntax for getting the fourth element in a list?
- What is the result of `"Hello World"!!8`?

3. Small list functions Try using the following functions on some of your lists. They're quite important to have a feel for.

   - `head` (there's also a less used counterpart `last`)
   - `tail` (there's also a less used counterpart `init`)
   - `length`
   - `null`
   - `reverse`
   - `take` (there's also a counterpart `drop`)
   - `maximum` and `minimum`
   - `sum` (there's also `product`)
   - `elem`, can be used *infix* by `` `elem` ``, e.g. `42 `elem` [2,12,22,32,42,52]`

4. Ranges and infinite lists

   - Use ranges to make a list of numbers for example from 23 to 42
   - Use ranges to make a list of characters for example from 'N' to 'P'
   - Use ranges to make a list with a different step, e.g. every multiple of 4 or all even numbers from 42 down to 20.
   - Make an infinite list with ranges and display the first few elements
   - Make an infinite list with `repeat` and display the first few elements
   - Make an infinite list with `cycle` and display the first few elements

2.2 Better functions

These exercises are all about improving your skills in writing recursive functions and using pattern matching.

2.2.1 First pattern matching

Following the book, try making a function that behaves differently depending on the input, e.g. if it receives a `1` it simply returns it, but a `2` returns `5`, and everything else is doubled.

You don't have to write the first line shown in the book, e.g. `lucky :: (Integral a) => a -> String` it is a type declaration, but the interpreter can make reasonable guesses if you make reasonable functions that has consistent input/output types.

Try making a function that behaves differently depending on whether it receives an empty list, a list with 1 element, or something else.

2.2.2 More pattern matching

Write the following functions, they should behave similar to the functions they mimic, but you should write your own version. Ask if you don't know how they should be computed. You can ignore bad cases or decide to return something that may or may not make sense.

You may have to define auxilliary (extra/helper) functions.

- `myhead`
- `mytail`
- `mynull`

1. Recursive functions
   - `mylast`
   - `mylength`
   - `mytake`
   - `mymaximum`
   - `mysum`
   - `myreverse`
   - `myelem`

2.3 Lists, revisited (list comprehensions)

Use list comprehensions to solve the following exercises, e.g. `[x | x <- [1..10], x /= 5]` is a list comprehension that creates a list of the numbers from 1 through 10, except the number 5.

Make a list of numbers to use in some the following.

List comprehensions make new lists; the exercises do not build on top of each other.

- Add your favorite number to all the numbers in your list.
- Double the numbers as long as they don't exceed some threshold.
- Let each element become its own list (inside the list).
- Remove all vowels (or some other set of letters you dislike) from a string.

Make a list of strings to use in the following.

- Get the length of each string.
- Get the strings with at least fx. 4 characters.

2.4 Something old, something new (Tuples)

2.4.1 First tuples

- Make a list of names.
- Make a list of ages.
- Use `zip` to make a list of persons (name, age tuple)
- Try using list comprehensions and/or functions to filter the persons, e.g. people starting with `'A'`, or people with more than 5 letters in their name, or people older than 50.
- Make a function with different patterns, which given a person (either the tuple or the two values), says something about that person, e.g. given a person starting with `'A'` , the function may respond with the string `"You're in the A-club"`.

2.4.2 More tuples

This is an exercise where each step builds on top of the previous.

1. Make a function that takes a point (an `(x,y)`-tuple) and returns the distance between the point and the origin `(0,0)`. (Hint: the distance formula for two points `(x1,y1)` and `(x2,y2)` is $\sqrt{(x1-x2)^2+(y1-y2)^2}$, you can simplify it for now if you want. Square root is `sqrt` and exponentiation can be done with `**`.)
2. Make a list of x-coordinates and a list of y-coordinates and use `zip` to make a list of points.
3. Make a list of numbers and use a single list comprehension on it to make a list of points. (Hint: You can reference the same list multiple times in a list comprehension.) Can be skipped if you have trouble.
4. Make a list of the distances to the origin.
5. Make a new distance function that takes two points and returns the distance between them.
6. Make a list of pairs of points, using list comprehensions.
7. Make a list of their distances.

2.5 Challenges

At this point you have acquired many tools, but getting used to how to use them takes practice. Start this challenge section by reading about quicksort in LYH, and really trying to understand how the solution is built. Think of solutions generally rather than how to implement them, this helps breaking tasks into

logical subtasks. Also sometimes, thinking about the base cases first helps get you started on the problem. Thinking about quicksort what we need to do is partition a list and sort the partitions reqursively. To partition we need a pivot element, but that can simply be the head element. The base case can simply be sorting the empty list is the empty list.

Here are two more suggestions for sorting algorithms to try and make. There are hints to how to think about the solution, as syntax is probably still a significant obstacle, but try to notice how to think about problems still.

- Mergesort. (Hint: We need a merge procedure that takes to sorted lists and creates one combined sorted list. Base cases are either list being empty. For the algorithm that solves the problem, it should `take` the first half and sort recursively and to take the second half, we can `drop` the first half. Finally we merge the two recursively sorted halves, for our solution.)
  - Rather than `take` and `drop`, you could have made your own function `mysplit` that given a list, returns a tuple with the two halves of the list.
- Insertionsort. (Hint: What we do in general is, we have an unsorted list and a sorted list (initially empty), and one step at a time we take an element from the unsorted list and **inserts** it into the sorted list. So actually, we need to supply the sorted list as an input alongside the unsorted part. We probably need a helper function as well that inserts an element into a sorted list. Note that there are better methods of doing insertionsort, but this is for practicing the thinking.)