Installing Haskell

In the first part of this tutorial we will see how to work with Haskell on campus and at home. We only need the bare basics:

- **GHCi**, the Glasgow Haskell Compiler (interactive), an interpreter that reads your Haskell files and executes Haskell code at the prompt.
- A **standard text editor** of your choice to write your code. Haskell files are standard text files with the extension `.hs`

If you prefer a fancier installation at home with a full IDE, that's fine, but it's not necessary.

On campus

Haskell is installed on campus machines. You can run it as follows:

- Open the start menu
- In the search bar, search for `command prompt` and open the app
- If you want to run Haskell on a file, navigate to that directory
- Type `GHCi` , or `GHCi <filename>` to open a file

For a text editor, I will use:

- `Start menu` $\rightarrow$ `Notepad++`

You may use any text editor of your choice.

Online

You can work online if you make an account here:

- replit.com

Warning: everything you program on this site is public. You should **not** use this for the coursework. I will try to set up a common account with private repositories that we can use for the coursework, and publish it on Moodle.

At home

Installing Haskell at home starts here:

- www.haskell.org/get-started/

This will give you two options:

- A direct install with GHCup: www.haskell.org/ghcup/
- An installation of VSCode, which includes Haskell: code.visualstudio.com/

Running things

Once you have GHCi running, you can do the obligatory "Hello world":

```
Prelude> putStrLn "Hello world!"
Hello world!
```

The interpreter can evaluate any Haskell expression, which is quite a lot. Let's start with some arithmetic. Simply type in some sums like the following ones, to get a feel for things.

```
Prelude> 2 + 3 * 4 ^ 5
3074
Prelude> ((2 + 3) * 4) ^ 5
3200000
Prelude> 2 - 3 / log 4
-0.1640425613334453
Prelude> sin (0.5 * pi)
1.0
```

You can set variables:

```
Prelude> x = 6*7
Prelude> x
42
Prelude> 2 * x
84
```

And those variables can be functions:

```
Prelude> square x = x * x
Prelude> square 123
15129
Prelude> hyp x y = sqrt (square x + square y)
Prelude> hyp 3 4
5.0
```

Note how in Haskell a function `fun` with two arguments is written `fun x y`.

Loading files

Let's put those functions in a file. Create a standard text file as `new.hs`, and make sure that GHCi was invoked in the same directory as your file (restart it if necessary). Then in the interpreter, use `:load` or `:l` to load the file.

```
Prelude> :l new.hs
Ok, one module loaded.
*Main>
```

The file is empty, but Haskell is fine with that. (It has changed the prompt, which turns out to be the name of the **module** that you've loaded, and if your module has no name, for instance because your file is empty, it settles on `Main`.) Open the file in your favourite editor to add the following lines, and don't forget to save.

```
square x = x * x
hyp x y = sqrt (square x + square y)
```

Use `:r` to reload the edited file. If you didn't forget to save, then you can use the functions `square` and `hyp` at the prompt:

```
Main> :r
Ok, one module loaded.
Main> square 2345702938
5502322273341831844
Main> hyp 6 8
10.0
```

And that's it! This will be our workflow for the coming weeks.

Welcome to Haskell!

In this part of the tutorial we will look at some basic concepts in Haskell: functions, types, recursion, and lists. You can use the lecture slides, available on Moodle, as a function reference.

Load the file `1-Haskell_Basics.hs` into `GHCi`, and open it in the text editor.

Error & undefined

In the file you should see the code:

```
square :: Int -> Int
square x = undefined
```

The expression `undefined` is a placeholder, and not working code. An attempt to evaluate it will result in an error. Internally it is defined as follows:

```
undefined = error "Prelude.undefined"
```

We will use `undefined` to present you with partial code (in particular because Haskell does not accept a type signature without a matching function declaration). With the function `error` you can define your own exceptions.

Exercise 1:

a) Try using the function `square`. Look up the types of `undefined` and `error`.

b) Complete the function `square`, replacing `undefined` with the required code.

c) Use `square` to write a function `pythagoras` that, for positive integers a, b, c, determines if they form a Pythagorean triple, $a^2+b^2 = c^2$. First, give a type signature.

```
*Main> square 4
16
*Main> pythagoras 6 8 10
True
*Main> pythagoras 1 2 3
False
```

Guards

You should see the code:

```
factorial :: Int -> Int
factorial n
    | n <= 1    = undefined
    | otherwise = undefined
```

The vertical bars, called **guards**, create a **conditional**, a choice depending on a boolean. Operationally, each guard is evaluated in turn, and the first to evaluate to `True` gives the return value for the function. The suggestively named expression `otherwise` is defined as `True`. Indentation is significant: Haskell uses it to decide where a new function starts, and which guards belong together.

Exercise 2:

a) Complete the function `factorial`.

b) Experiment with the indentation and placement of the guards to see what works and what doesn't. Can you put the first one on the same line as the function? Can you put everything on one line?

c) Write a function `power` to compute a^b given a and b. Use the following algorithm: $a^0 = 1$ and $a^{n+1} = a \times a^n$.

d) Add another guard to the `power` function to throw an exception when b is negative.

e) A more efficient algorithm for `power` (which reduces the number of multiplications needed) is the **exponentiation–by–squaring** method. For a^b, if b is even it takes a^2 and raises that to the power $\frac{1}{2}b$; if b is odd, it raises a^2 to the power $\frac{1}{2}(b-1)$ and multiplies that by a. Adjust your `power` function to use this method.

f) The Euclidean algorithm for the greatest common divisor (GCD) of two natural numbers is this: for input x and y, if x and y are equal, that is also their GCD; otherwise, take the GCD of the smaller one of x and y and the difference between x and y. Implement this as the function `euclid`. Raise an error when one of the inputs is zero or negative.

```
*Main> factorial 20
2432902008176640000
*Main> power 6 7
279936
*Main> euclid it 42075
9
```

If–then–else

In addition to guards, Haskell has another conditional:

```
if b then x else y
```

Here, `b` is of type `Bool`, and is evaluated first; if it returns `True`, the expression evaluates `x`; if `False`, it evaluates `y`. We can use it to write our `factorial` function as follows.

```
factorial n = if n <= 1 then 1 else n * factorial (n-1)
```

One can see guards as a convenient notation ("syntactic sugar") for repeated if–then–else statements. The next exercise asks you to explore how this works.

Exercise 3:

a) Rewrite the three versions of the `power` function of the previous exercise as `pow`, this time using if–then–else instead of guards.

b) Experiment with the layout and indentation of your function. Can you break it by changing the layout? Can you make it easily readable?

Extra

For additional practice, here are a few more traditional recursive functions that you can implement.

The Collatz conjecture

A nice open problem in number theory is the Collatz conjecture, which supposes that for any number n greater than 1, the sequence of the following two steps eventually reaches one:

$$\begin{aligned} n &\mapsto \tfrac{1}{2}n && \text{if } n \text{ is even} \\ n &\mapsto 3n+1 && \text{if } n \text{ is odd} \end{aligned}$$

We can build this as a function in Haskell, but we don't know if it will terminate!

Exercise 4:

a) Implement the function `collatz :: Int -> Int` to test the Collatz conjecture. It should return `1` when it terminates.

b) Adapt the Collatz function as `collatzCount :: Int -> Int -> Int` so that it counts the number of steps taken. Use the extra input to count steps, and change the output to return the number of steps instead of always `1`.

c) Implement the function `collatzMax :: Int -> (Int,Int) -> (Int,Int)` so that `collatzMax n (0,0)` finds the number between 1 and n that requires the most steps to reach `1`. Use the second argument in `collatzMax n (m,s)` to remember the current number m with the longest sequence of s steps.

```
*Main> collatz 97
1
*Main> collatzCount 97 0
118
*Main> collatzMax 100 (0,0)
(97,118)
```

The Ackermann–Péter function

The Ackermann–Péter function (often just called the Ackermann function) was designed as an example of a computable function that is not primitive recursive. It grows very quickly!

$$\begin{aligned} A(0,n) &= n+1 \\ A(m+1,0) &= A(m,1) \\ A(m+1,n+1) &= A(m,A(m+1,n)) \end{aligned}$$

Exercise 5:

a) Implement the Ackermann function as `ackermann :: Int -> Int -> Int`. Try it out for very small values.

```
*Main> ackermann 2 2
7
*Main> ackermann 3 7
1021
```