COMP304 Tutorial 1
15/07/2016

1 Getting Started

When you download and install Haskell it will come with two programs: `ghc` and `ghci`. `ghc` is a compiler that produces executables from Haskell source files (like `gcc` for C). `GHCi` is an interpreter; as you enter lines of Haskell, `GHCi` will spit out the result (like the Python interpreter).

When you run `GHCi` it automatically loads the standard library (which is called Prelude) and prompt you for some input. If you enter in some arithmetic or boolean expressions it will give you the answer.

```
Prelude> -3
-3
Prelude> 2 - 3
-1
Prelude> 1 /= 1
False
Prelude> 4 == 4
True
```

Note that Haskell uses /= for "not equals", which sort of resembles the inequality symbol $\neq$ from mathematics. A good mental model is to think of the built-in boolean and arithmetic operators as being like functions. With that in mind, what's wrong with the following?

```
Prelude> 1 + -2

<interactive>:10:1:
    Precedence parsing error
        cannot mix ‘+’ [infixl 6] and prefix ‘-’ [infixl 6] in the same infix expression
```

Some operators are written infix i.e. between their arguments. In this case, `GHCi` is confused about how to interpret this because we've mixed an infix operator "+" with a prefix operator "-". The fix is to add brackets to disambiguate:

```
Prelude> 1 + (-2)
-1
```

When you see -2, think of it as the operator "-" (negation) applied to the value 2. The operator "-" is overloaded: it can either mean the operator which negates numbers, or the operator which subtracts one number from another. Functions in Haskell are frequently overloaded by the argument's type and even value (by pattern-matching, as we'll eventually see).

You can exit GHCi like so:

```
Prelude> :quit
Leaving GHCi.
```

2 Loading From Files

Haskell source files usually end with the file extension .hs. There are also literate Haskell files, which end in .lhs. We'll talk about those more next week. For now we can write some Haskell code into the file fact.hs and load it into `GHCi` like so:

```
fact n = if n == 0 then 1 else n * fact (n-1)
```

```
Prelude> :load fact
[1 of 1] Compiling Main ( fact.hs, interpreted )
Ok, modules loaded: Main.
*Main> fact 3
6
```

The prompt has changed from Prelude to Main. This is `GHCi` keeping track of which modules are currently loaded. In fact.hs we haven't specified a module, so our fact function defaults to residing in the Main module.

Here's an expression which recurses forever:

```
*Main> fact (-2)
```

In some ways Haskell functions are like mathematical functions because they're referentially transparent: on the same input, they'll give the same output, regardless of any ambient external state. Unlike mathematical functions, Haskell functions don't have to be total (defined on every input), as with `fact`(-2).

3 Conditionals

Let's modify the `fact` function so it only has one branch and then load it into `GHCi` again.

```
fact n = if n = 0 then 1
```

```
:reload
[1 of 1] Compiling Main ( fact.hs, interpreted )

fact.hs:3:1:
    parse error (possibly incorrect indentation or mismatched brackets)
Failed, modules loaded: none.
```

Unlike conditionals in Java or C, Haskell's conditional is an expression; when executed it reduces to a value. Therefore it must be defined on both branches (it's like the ternary expression if you've seen that).

4 First-Class Functions

In Haskell, functions are "first-class". This means they can be treated as variables, return-values, etc. like other values in your program. Here's the `twice` function:

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

```
*Main> twice fact 3
720
```

Note: `twice fact 2` is NOT the same as `twice (fact 2)`. `twice` is being applied first (function application takes highest precedence, left-to-right). `twice` takes two arguments, so its first argument is bound to the function `fact` and its second argument is bound to the number 2.

To understand how we got the value 720 we can use substitution. Because Haskell is referentially transparent, when you see a function being applied you may always substitute it for its body. Here's what that looks like:

```
twice fact 3
fact (fact 3)
fact (if 3 = 0 then 1 else 3 * fact (3-1))
fact (3 * fact (3-1))
fact (3 * fact 2))
fact (3 * (if 2 = 0 then 1 else 2 * fact (2-1)))
fact (3 * (2 * fact (2-1)))
fact (3 * (2 * fact (1)))
...
720
```

With repeated substitution we'll eventually get to 720. Useful when debugging! Use this technique when you see a complicated Haskell expression you don't understand:

```
twice (2-) 2
2- (2- 2)
2- 0
2
```

5 Lazy Evaluation

Though we can use substitution to correctly reason about programs, Haskell really does something else under the hood. Expressions are only evaluated when they're needed. Later we'll see that we can do cool things with lazy evaluation (the poster-boy being infinite lists). For now think about this program:

```
recurseForever n = recurseForever n
tricky n = 69
```

```
*Main> tricky (recurseForever 5)
69
```

In a strictly-evaluated language we would evaluate `recurseForever 5`, bind the result to `n`, then apply `tricky` to the result.

In a lazily-evaluated language we put the expression `recurseForever 5` in a "thunk" and continue. The thunk is evaluated if it gets used later on. Since the body of `tricky` never uses `n`, we never need to know the value of `recurseForever 5`, so never enter into the endless recursion.

6 Types

Haskell has a strong, inferred type-system. This means you often don't have to provide explicit type signatures for functions; Haskell will figure that out using an algorithm called Hindley-Milner type inference. Explicitly writing out the type of a function is good practice though since it serves as documentation to other programmers.

In GHCi we can check the type of a function like this:

```
*Main> :type fact
fact :: (Eq a, Num a) => a -> a
```

Haskell is telling us that the type of the "fact" function is (`Eq a`, `Num a`) ⇒ `a` → `a`. `a` is a type parameter; a bit like Java's generics. The stuff left of ⇒ specifies that whatever the type of `a` is, you must be able to test it for equality (it must be `Eq a`) and you must be able to do arithmetic on it (it must be `Num a`).

The meaning of `a` → `a` is that `fact` takes in something of type a and outputs something of type `a`, where `a` is any type contrained by the type-classes above.

Since `fact` only uses equality and arithmetic, type inference has ascribed it as operating on any type for which equality and arithmetic is defined. This means we can immediately use it on doubles.

Is `Bool` a vaild type that matches the signature? What happens if we try to apply `fact` to `Bool`?

```
*Main> fact True

<interactive>:6:1:
    No instance for (Num Bool) arising from a use of ‘fact’
    Possible fix: add an instance declaration for (Num Bool)
    In the expression: fact True
    In an equation for ‘it’: it = fact True
```

Though you can check if two `Bools` are equal, you can't multiply them so our `fact` function isn't going to work. This is why GHCi is complaining about `True` not having a type that matches `Num Bool`.

`Eq` and `Num` are type-classes; they are sets of types upon which certain functions are defined. `Eq` is the set of types where == is defined. `Num` is the set of types that you can perform arithmetic on (`Double`, `Int`, etc). Type-classes are a little bit like interfaces in Java (but don't be fooled by the word "class"; typeclasses are different to object-oriented programming classes).

Note: “it” is a special variable in GHCi which stores the value of the last expression. If you enter “it” into the prompt you can see what that is. This is the meaning of the last line in the error message.

Since the mathematical factorial function is only defined on whole numbers, let’s put an explicit type on it.

```
fact :: Int -> Int
fact n = if n = 0 then 1 else n * fact (n-1)
```

Now if we reload the file and inspect the type of fact:

```
*Main> :type fact
fact :: Int -> Int
```

When we omit type information, Hindley-Milner type inference will figure out the most general type for which the function typechecks. Here’s the type of the built-in max function:

```
*Main> :type fact
max :: Ord a => a -> a -> a
```

The `max` function operates on two values of any type `a` in the `Ord` typeclass (meaning its values can be compared/ordered). It returns something of type `a`. Note that the arguments have to be of the same type.

As you might expect, you can apply `max` to two numbers. You can also apply it to two `Chars`, because `Chars` can be compared (alphabetically). However, you can’t feed a `Char` and an `Int` to `max` because they have different types.

```
*Main> max ‘c’ ‘d’
‘d’
*Main> max 3 ‘c’

<interactive>:14:5:
    No instance for (Num Char) arising from the literal ‘3’
    Possible fix: add an instance declaration for (Num Char)
    In the first argument of ‘max’, namely ‘3’
    In the expression: max 3 ’c’
    In an equation for ‘it’: it = max 3 ’c’
```

The error message is trying to consider ‘c’ and 3 as the same type hence the error message about `NumChar` (it’s trying to consider ‘c’ as a number).

For comparison, the signature for this Java function:

```
int myMethod(double d, char c)
```

Would look like this in Haskell:

```
myMethod :: Double -> Char -> Int
```

Later on we’ll see the reason for the strange arrow notation. For now, just remember the type that the function returns is the last type in the sequence of arrows. Lastly, here’s the type for `twice`:

```
twice :: (t -> t) -> t -> t
```

The first grouped `t` → `t` is a function from `t` to `t`. We may read this as: `twice` is a function which takes a function from `t` → `t`, and an input of type `t`, and produces an output of type `t`. There are no type-classes in this signature, so `t` has no constraints; it can be any type.

→ is right-associative, so the brackets are sometimes important. `t` → `t` → `t` is not the same thing as A: (`t` → `t`) → `t`, but it is the same thing as B: `t` → (`t` → `t`). A is the type of a function which takes a `t` → `t` as input and outputs a `t`; B is the type of a function which takes a `t` as input and outputs a `t` → `t`.

7 Extra

Most commands in GHCi can be abbreviated by typing in the first letter they start with. So the following are the same:

- :`load fact` and :`l fact`
- :`reload` and :`r`
- :`type 3` and :`t 3`
- :`quit` and :`q`