CS141 # Bonus Lab: A Helpful Haskell Compiler! January 2025

Hi there! Thank you for agreeing to take part in user testing of a new compiler/debugger I've been building for my 3rd year project. The aim of the project is to: 1) Improve / simplify the errors given for bugs in code over GHC (Haskell's compiler) and 2) Add a 'tracing' feature which allows the user to see exactly how their program was evaluated. Both of these should make the life of a programmer who is new to Haskell much easier and aid in the self-learning process.

This lab sheet will guide you through a few different tasks and show off different features of the debugger, as well as asking you questions along the way to allow you to give feedback - feel free to be as critical as you like, suggest changes you might want to see and comment on what features are the most helpful. If you have any questions feel free to ask them (I should be in the lab in person at 5pm on Monday 20th), and once you are done (or have gone as far as you can in the time you have), please email the file `myCode` to me at 'Alex.Hobbs.1@warwick.ac.uk'. I hope you enjoy and find it useful!
-Alex

Ex. 1. To start, run:

```
git clone https://cs141.fun/helpful-haskell-compiler
```

in a terminal. A folder called `helpful-haskell-compiler` will be created which contains the project files.

Navigate into that folder with '*cd helpful-haskell-compiler*' and open the project in VSCode with '*code .*' , or any editor of your choice.

Ex. 2. If you are on a DCS machine, run:

```
chmod +x ./compile
```

in the terminal while in the project folder. This will grant you the permissions to run the compiler.

Look at the file `myCode`. The function *main* is where the evaluation starts, and other functions can be written as they normally would in Haskell. Type signatures (e.g. main :: ...) can be written but are not necessary, as you will see later.

Ex. 3. While in the project folder, run:

```
./compile myCode --trace=1 --verb=1
```

and it should show the output '10.0', because this compiler only works in doubles (floating-point numbers) rather than integers.

Now try changing 'trace=1' to 2 and then 3, and see what difference it makes.

> After that, with trace on 3 try changing 'verb=1' to 2 and then 3.

The *trace* argument (for tracing) determines the level to which you can see how your code has been evaluated. The *verb* argument (for verbosity) determines how much explanation the tracer gives for what it is doing at each step, where 1 is none, 2 is concise and 3 is full explanation. I recommend you keep both of these at 2 or 3 for the rest of the lab but feel free to change them as you see fit.

> **Ex. 4.** Now time to write some code! In the `myCode` file, write a recursive factorial function called *fac*. As a reminder, the factorial function multiplies an integer by all of the positive integers less than it (0! = 1), and recursive just means that it does so by calling itself (rather than using a for loop, for example). To do this you can use top-level pattern matching or a case statement, both of which you will have seen in last week's lab (**if statements and guards are not currently supported**).
>
> To run this function change the *main* function to 'main = fac 4' (or any number you like), save it and run the command like last time.

Question 5. Do you find the tracing feature useful (being able to see each step of evaluation)? Why or why not? Do you prefer using verbosity level 2 or 3 and why?

> **Ex. 6.** In the `myCode` file I've defined a function called *addTwo*. Try it out by changing the *main* function to 'main = addTwo 3'.
>
> Now I'm sure your code will have been perfectly written so far, so you won't have seen any error messages, but now I want you to deliberately break the code! Perhaps try 'main = addTwo "hello!"', 'main = addTwo' and 'main = addTwo 3 4', as well as any other way you feel like! Depending on the level of verbosity you have set when you run the code, the error messages will look slightly different and will be more or less informative.

Question 7. Do you find the error messages helpful? Why or why not? If you find an unhelpful error message how might you change it or what else would you like to see added to make debugging easier?

This kind or error checking is known as "dynamic type checking", as it evaluates until an issue (such as a number being added to a letter) stops it from going any further. Haskell's regular compiler GHC uses "static type checking" which infers the type of every function and checks that the program is error-free before running.

> **Ex. 8.** Going back to 'main = addTwo 3' or some other working program, save it and run:
>
> ```
> ./compile myCode --trace=3 --verb=3 --dump=output.txt --stc
> ```
>
> in the terminal. The flag *stc* turns on static type checking for the program, and the argument *dump=output.txt* will output a de-sugared version of your code to the file `output.txt`.
>
> Have a look at that file and see how different your code looks, and that type signatures have now been added!

> Now you can again try breaking your code, and notice that it reports the error before running.

Question 9. Do you prefer dynamic or static type checking when working in Haskell, and why? Is the `output.txt` file readable to you, and can you see how your code has been transformed into that equivalent code?

From this point onwards, you can use any level of tracing and verbosity you like and can choose to keep static type checking enabled with the *stc* flag or remove it to return to dynamic type checking.

> **Ex. 10. THE FINAL CHALLENGE:** In regular Haskell, Prelude provides a series of standard functions to help when you code - but this version of Haskell doesn't have them (except for 'head', 'tail', 'fst' and 'snd' for handling lists and tuples). Write a function *len* which takes as an input a list of any type, and returns the number of items in that list. Test your function by changing the *main* function to 'main = len [1,2,3,4]'. *Hint: This will be another recursive function using either top-level pattern matching or a case statement.*

Question 11. How do you find the ease of development using this new compiler/debugger vs. when using GHC? Is this compiler easy to use, and is there anything that can be done to make it more appealing or improve your quality of life while using it?

> **Ex. 12. Bonus (totally optional) exercise:** This new compiler is still in development and so lacks features and inevitably has bugs - it's only 3 months old after all! A brief summary of what it does have is as follows: lambda functions and sugaring, basic types (Double, Char, String/List[Char], Lists (including ranges) and Tuples), 'head', 'tail', 'fst' and 'snd' functions, type inference, top-level pattern matching and case statements, tracing and simple error messages.
>
> With that being said, I'm happy for you to try and break it and would love to see you stretch it as far as it will go, to make interesting and creative programs with such limited tools. Alex has informed me that you're a very talented year group so I look forward to seeing what you come up with!

Question 13. Is there anything else you'd like to say about about this project? It can be critical review, your favourite / most useful aspects, observations about this compiler or GHC, or anything else.

Thank you so much for taking part in user testing for my compiler and debugger project. Could you please now email the file `myCode` to me at 'Alex.Hobbs.1@warwick.ac.uk'. Parts of your answers to the questions asked may be used anonymously in my final report (either as direct quotes or general sentiment) which will be submitted in April and which I'd be happy to share after. If you have any questions about this user test or about the project as a whole, feel free to email them to me and I'll try to respond as soon as possible!