1 C-Like Language Interpreter

This program is an interpreter for a C-like imperative programming language with **sequence**, **selection**, **iteration** and **functions**. Several example programs have been provided as `.txt` files.

When run with `stack run`, the user is prompted for the name of a file to interpret.

Variables may be integers or floating-point numbers, with most of the common mathematical and Boolean operators supported. Explicit declarations and type specifiers do not exist. The language is technically dynamically typed.

The weirdest feature is that the result of a bare expression or function call that isn't assigned, returned or used in a loop is printed. See `hello.txt` for an example.

I have used the BNF grammar file from the CS325 coursework, which describes the syntax of a language called 'Mini-C', as inspiration for my syntax.

There are only so many ways to skin a cat and Haskell reduces that number to one quite often. Some code here was lifted directly from Mark Karpov's tutorial, frankly for want of an alternative. Any instances of this have been noted.

I have occasionally capitalised nouns corresponding to data types. E.g. This evaluates an Expression.

2 Files

File	Description
Types.hs	Most of the type declarations for the project.
SymbolTable.hs	The symbol table data type, and functions to query and update it.
Lex.hs	Basic lexicographical parser functions, mostly copied from Mark Karpov's tutorial.
Statement.hs	Parsers for code statements.
Function.hs	The pFunction parser for functions.
Expr.hs	Parsers for expressions (mathematical and Boolean). (A function call is also an expression.)
Eval.hs and Eval.hs-boot	Functions for evaluating the AST of an expression. Most notably eval.
Exec.hs	Functions for executing Statements and lists thereof.
Main.hs	Where the function table is loaded and main() is called.

3 Maths and Boolean Expressions

I first created a mathematical expression parser and evaluator. Even it appeared to technically meet the requirements for 40%.

At this point, I encountered slow-down in compile time due to all my code being in `Main.hs`. I tried to mitigate this by splitting the project into many files, but the benefit was limited since the files form a dependency tree, with every descendant needing to be recompiled if a file changes.

It was straightforward to add Boolean expressions by using the convention that zero means false, and non-zero means true. These are encoded in the `truth` and `boolVal` functions in `Eval.hs`. All binary operators strictly evaluate both their operands every time.

See `maths.txt` for an example program using maths expressions and Boolean expressions.

4 Variables, Assignment and Sequence

Variables were surprisingly easy to add to the expression evaluator, after spending 3 hours deciding how to do it. Since they are used as `String` expressions that represent `Values`, it seemed natural to implement this with a `Map` (`Eval.hs`: `Value` case in `eval`).

Having a sequence of instructions is only meaningful if the result of one affects the next, by mutating state. The most basic mutation is assignment, which is also needed for variables to be useful. The obvious implementation for this is a `State` transformer that evaluates a `Statement` (`Exec.hs`: `exec`), such as an assignment.

There's no sensible behaviour to continue execution if a variable is not found, so crashing with `error` seems reasonable, in addition to being much easier than wrapping everything in `Maybe`. I could have alternatively implemented a complex chain of passed `Either` error values back up to `main`, which could then display an error message and cancel execution.

5 Scoping

I want variables to only exist within the {scope} where they are defined, since this allows programs to be more comprehensible. My approach for implementing this is to have a stack of variable value maps, with a new one added on top for each nested scope.

Variable names cannot be 'shadowed', due to the lack of explicit declarations; I consider the practice harmful, so I am indifferent. There is no 'global scope', so all variables must be defined within functions. The program `sequence.txt` demonstrates scoping.

6 Selection & Iteration

Selection (`if`) is performed by evaluating an expression, then executing the corresponding `Statement`. This statement may be a {scoped} block, allowing multiple actions to be performed in one branch. In an (`if-else`) structure, the alternative `Statement` is executed if the expression evaluates to false.

Iteration is implemented near-identically to selection, with the addition of recursion.

See `branching.txt` for an example.

7 Functions and `return`

I wanted to implement functions as lambdas, which would themselves be `Expr` expressions. I tried this and found a dependency cycle, where Expressions could contain Statements, which could contain Expressions, making it unreasonably difficult to implement into my existing architecture.

Instead, I used the much more C-like form of the top level of a program file consisting of several function definitions, including `main()`, which is used as the entry-point to start evaluation. I literally do this by evaluating an Expression to call the `main()` function (see `Main.hs`).

Distressingly, the dependency cycle remained, and I had to use `Eval.hs-boot` to break it. If I'd known how to do this from the start, I could have implemented lambdas and first-class functions, which would have been much cooler.

As there is no global scope, and the language does not support references, each function (call) can have its own isolated symbol table for variables. These are stored in the familiar function call stack, which Haskell can implicitly create for me through recursion.

Returning works by adding a value for the key "return" to the top level of the symbol table. Since `pIdentifier` prevents user-input assignments to this symbol from parsing, it will only be created once a `return` Statement has run. The (function) interpreter may then simply evaluate the Statements in the body of a function in turn until the `return` symbol has a Value, then pass that Value out to the expression it was called in. When a value is returned inside a {`Block`}, it is necessary to 'cascade' it out to the symbol table below before it goes out of scope, as seen in `Exec.hs`.

An interesting consequence of using `Map.fromList` to construct the table of functions is that duplicate definitions are not considered a problem, and the most recent one is used. See `duplicate.txt`.

I struggled to implement the `pArgs` parser until I discovered `sepBy`, which just does what I want.

8 Technology & Resources

8.1 Code

- Megaparsec, a monadic parsing library.
- Megaparsec Tutorial, by Mark Karpov.
- Learn You a Haskell for Great Good!, by Miran Lipovača.
- Hoogle.
- The CS325 coursework grammar file.

8.2 Report

- Emacs Org Mode
- `\usepackage[margin=2cm]{geometry}`