

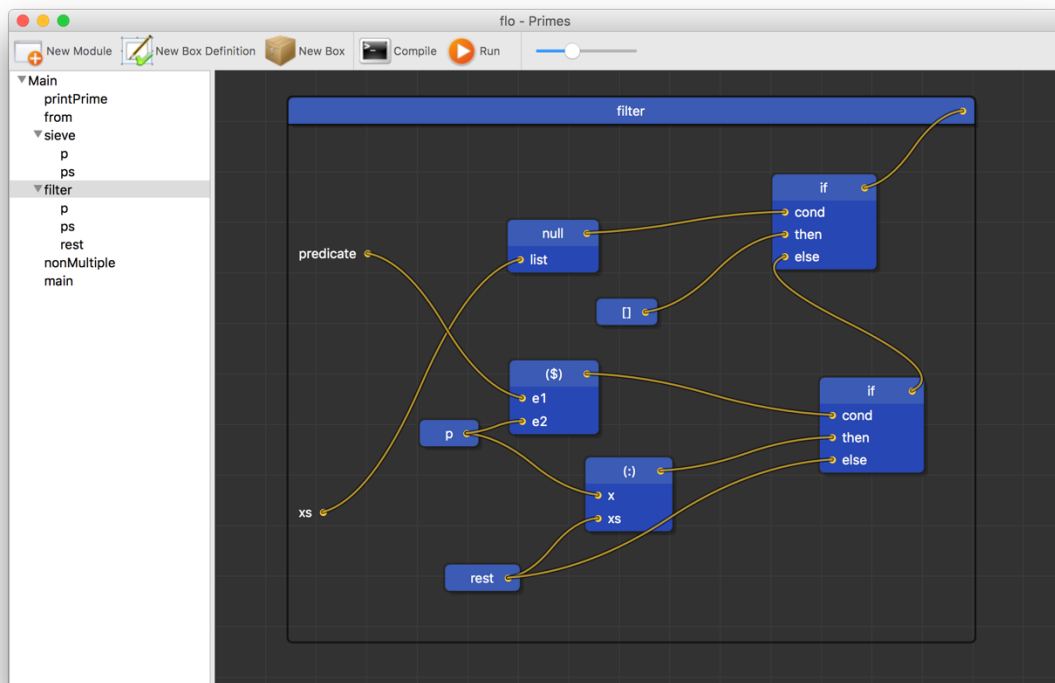
flo

A visual, purely functional language

Overview

Purely functional programming languages have the special property that every function is a function in the mathematical sense. This means they have no side effects and exhibit referential transparency, which can be extremely helpful for programmers and compilers when trying to reason about and optimize code. The benefits of functional programming have been championed in academia for years, but one of its drawbacks, in my opinion, is that it can be difficult to learn. My goal with this project was to create a language that exhibits these beneficial features but is much easier to learn.

With that, I introduce you to flo, a visual, purely functional language. The syntax and semantics are heavily inspired by Haskell, whereas the visual editor is modeled after Apple's Quartz Composer. Instead of writing out lines of code, programmers create flow charts that model the execution of their program. Flo features strong static typing, non-strict semantics, parametric polymorphism, algebraic data types, and other features commonly shared by purely functional languages, presented in a visual manner that is intuitive and extremely simple to learn.



The Editor

The visual editor is written in Java and consists of a tree, a toolbar, and a grid. The tree lists the contents of the file that's currently open, which will consist of various modules and function definitions (called box definitions in flo). The toolbar has buttons for adding a new module, adding a new box definition, adding a new box (more about that later), compiling and running the program, and a slider for changing the current zoom level of the grid. The grid allows the programmer to edit the visual component of their programs. The visual syntax is described below.

Boxes and Cables

Flo's syntax is extremely simple and is comprised entirely of boxes and cables. Boxes primarily represent functions, but they are also used for literals and constructors. Boxes may have a list of inputs, or they may have none if they are literals or constant applicative forms. Each box has an output, which is the expression the box returns.

Cables connect outputs to inputs. They represent a flow of values from one box to another. To apply a function to a set of values, simply connect the outputs of the values to the corresponding inputs on the function's box. Boxes can also be partially applied if not all inputs have cables connected to them. In particular, if you need to pass a function as an input to another function, attach a cable to the box's output, and don't apply any inputs to it.

In the picture shown above, the box named "if" represents the if function. It takes a condition and two expressions, returning the first expression if the condition is true and the second expression if the condition is false. The cable coming out of the right side of the "if" box represents the expression that is returned.

Inputs

Box inputs have names which can provide helpful annotations. However, these names are essentially meaningless and are stripped away by the compiler. The only thing that really matters is the order of inputs.

Literals

Literals are represented as boxes with no inputs. The name of the box determines which kind of literal it is. Integers and floating point numbers are boxes with numbers for their names. Strings are surrounded by double quotes, while characters are surrounded by single quotes.

Box Definitions

When using boxes in expressions, they appear as black boxes. Inputs go in and an output comes out, but the internals are not visible. A box definition, however, defines the internals of a box. To use the box's inputs, hook them up to cables. The value of the box is equal to the expression that is connected to the box's output, just as you'd expect.

Box definitions can be arbitrarily nested, creating the equivalent of a `let` expression in Haskell. When a box definition occurs nested inside another box definition, the scope of the local definition is only for the box definition it is contained in.

The picture above shows the box definition for the “filter” function. “predicate” and “xs” are the function's inputs, and the cable coming out of the the top “if” box represents the function's output.

Modules

Just as in other languages, a program can have multiple modules. The purpose of modules is to limit the scope of box definitions and support code reusability.

Types

Flo is strongly, statically typed, but type inference is performed so that type annotations are unnecessary. An important exception to this is when defining the types of fields in data constructors (described below). To specify that an expression has a certain type, use the built-in function `idMono`, which is the monomorphic identity function. It takes two parameters, a type and an expression. It returns the second parameter unchanged, just like the regular identity function, but it forces it to have the type of the first parameter.

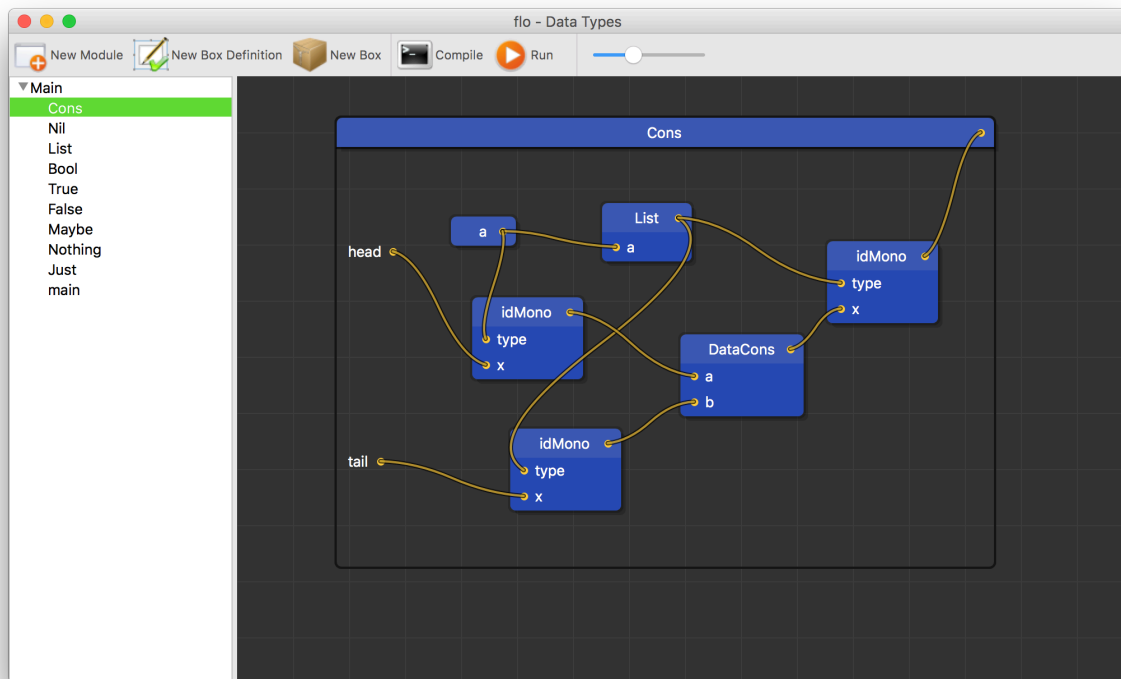
Type Constructors and Data Constructors

Type constructors are defined just like normal functions, but they are essentially just a wrapper around `TypeCons`, which is a built-in type constructor. It can take zero or more arguments, and it returns the component types packaged up into a new type which has the same name as the surrounding box definition.

Similarly, data constructors are also defined like normal functions that are simply a wrapper around `DataCons`, a built-in data constructor. It also takes zero or more arguments and packages up its components into a new data type.

The picture below shows the definition for the data constructor `Cons`, which creates a linked list. It takes two parameters, the head and tail of the list, and packages them up with `DataCons`

into a new list. As you can see, `idMono` is used to annotate that if the head has type `a`, then the tail must be a list of `a`'s.

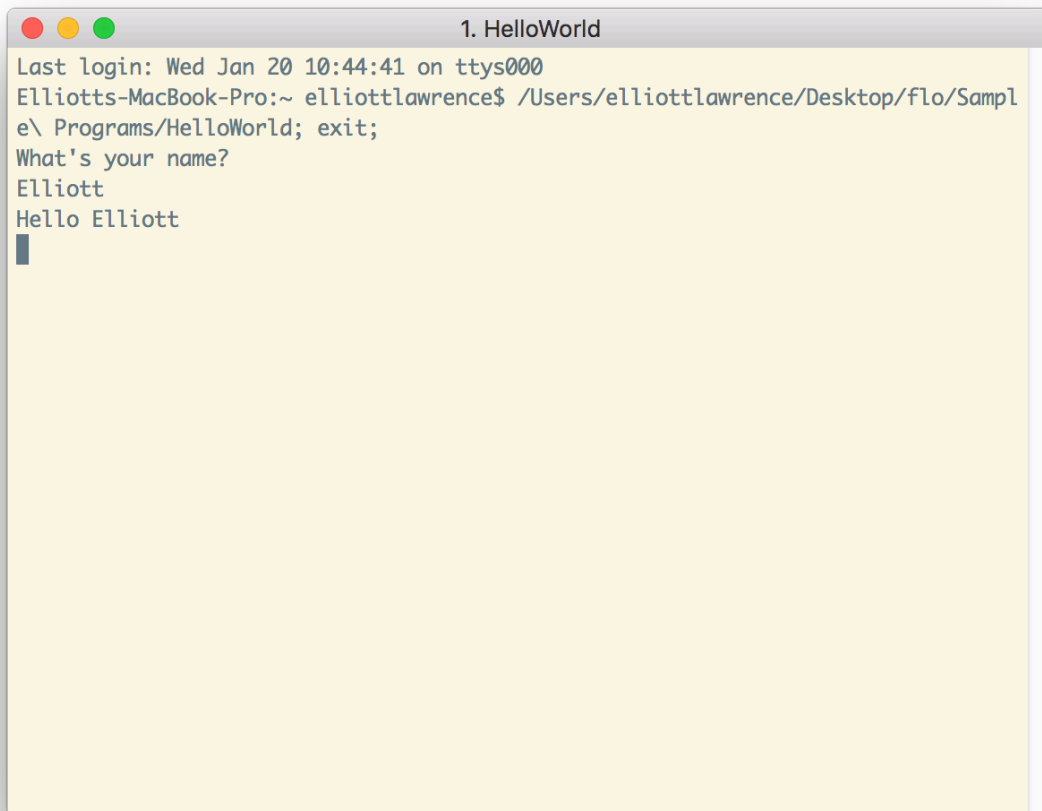


The Compilation Process

When the programmer clicks the Compile button, the following steps take place:

1. First, the file is saved as a `.flo` file, which is essentially just a JSON file that encodes all of the necessary program data.
2. The editor then runs the compiler, which is a separate program written in Haskell.
3. The Haskell program loads and parses the JSON format into the `FloGraph` data type. This data type represents the graphical structure of a program.
4. A `FloGraph` is converted into a `FloProgram` data type. This is the actual abstract syntax tree of the program.
5. In future versions, the compiler will continue compiling this AST in the normal way, but for now it converts a `FloProgram` into a `HaskellProgram` data type. This data type represents actual Haskell code and is then printed out to a file.
6. The Haskell compiler is run on the generated Haskell code, outputting the final executable.

After this happens the user can click the Run button to run the program. Here is a screenshot of a sample Hello World program being run:



```
1. HelloWorld
Last login: Wed Jan 20 10:44:41 on ttys000
Elliotts-MacBook-Pro:~ elliottlawrence$ /Users/elliottlawrence/Desktop/flo/Sample\ Programs/HelloWorld; exit;
What's your name?
Elliott
Hello Elliott
█
```

Although this compilation process is much simpler than it will be in the future, what's nice about this method is that it allows the user to see the actual Haskell code that is generated.

Conclusion

My intention in designing flo was to create a general-purpose functional language that is easy to learn and fun to use. Programming with visual elements offers a refreshing break from the monotony of writing lines upon lines of code, not to mention the fact that it offers a unique perspective on functional programs that I think is very illuminating. Although visual programming languages are rarely used in the industry, I think it is quite possible that flo could one day be used as an educational tool for teaching functional languages, and perhaps it will find its footing in other niche markets as well.