Design

For this milestone we followed a roughly test driven development model. We started by having one team member write tests, specified as some IBTL code paired with expected output after being run through gforth. Another team member worked on a test suite that would compile the IBTL code, run the resulting gforth, compare the output, and show any errors. The remaining two team members started in on the translator itself.

After the tests and test suite were reasonably complete, we started testing very often, focusing our development effort on either making tests to highlight problems in the code, or writing code that made the tests pass.

Our parser outputs S expressions, which made translating code into gforth straightforward. For each expression, we assumed the first element was a function name, which was mapped to a function in our translator. The translator would gather as many arguments as needed from the input stream. An incorrect number of arguments supplied to the function would cause a compiler error, and abort. Each argument gathered from standard input is analyzed; if it is an atom, it is passed to the function unmodified, and if it is an expression, the translator recurses to compile the sub-expression.

The output from each function is a new type of lexical token, `OutputToken`, which contains the text of the gforth code that was compiled, and the expected type that would be on the stack after that code ran. The type tag was needed so that functions like println could probably handle different types, and so that operators like plus could choose the right kind of gforth operation to perform.

We were then able to use this type information to allow for auto-coercion of types under certain operations. For example, the add could handle code such as `(+ 1 2.5)`, which would promote 1, and integer to 1.0, a float before adding it to 2.5, resulting in the intuitive result of 3.5. Most other math operators had similar logic.

After all this, the gforth code from all expressions is concatenated together, and written to `stdout`. Our test suite then took those and ran them through gforth to analyze their correctness.

Specification

This milestone is to teach us how to make a translator, and so that we form a form a stronger definition of the semantics of our language (such as parameter counts and function names).

Processing

We solved the problem by getting together and assigning tasks. We then all worked on our tasks together, collaborating with git and sharing code very often, so that our development was never too far out of sync with the rest of the team. We all worked together, and often shared and discussed tasks.

Testing

We tested the code extensively by writing a test suite that tested every function in IBTL. We verified that the compiler compiled correct code, and threw errors on incorrect code. We also verified that the output of running the generated code matched the epected output, thus verifying that the compiler produced code that is likely correct. We chose test cases that would be unlikely to produce false positives.

Retrospective

I learned why Lisp was written in S expressions: they are extremely easy to reason about and translate into other kinds of code. The base of our translator surprised me with it's brevity. Most of the body of the program is the code that defines what IBTL function do, which is straight forward to write in most cases. I also learned more Ruby, which is good because we chose to write our code in Ruby to force ourselves to learn the language.