An Informal Introduction to DTM

1 *Introduction*

As a language, Do The Math (DTM) serves as a tool to solve math problems. Given the way expressions are formatted, DTM reinforces the order of operation rules and avoids problems that are open to interpretation in how they should be solved. This is done with the use of parentheses around all possible operations between two numbers. The language is built using the F# programming language and is built using the .NET Framework from Microsoft. Although it was written intended to take simple expressions from the command line, DTM also supports running files with an extension of ".dtm", as well as having a functioning repl that can be used in the command line. The language implements being able to evaluate math problems in prefix and infix notation. For more information giving a much more detailed description of the language, its syntax and its semantics, be sure to read *A Look into DTM (Do the Math)*, which is included in the language's source code.

2 *Using DTM as a Calculator*

Like most other languages, DTM works as a calculator that can be used to compute math problems. However, in the case of DTM, being a calculator serves as its main purpose. The operators included in DTM are +, -, *, /, and ^. They can be used in either infix form, prefix form, or a mix of both. Below are a couple of examples of how programs could be run with DTM from the command line (represented with the notation of ->):

```
-> dotnet run "(1 + 5)"
6.0
-> dotnet run "(/ (11 - 2) 3)"
3.0
-> dotnet run "example-1.dtm"
2.25
-> dotnet run "(+ 1 2 3 4 5 6 7 8)"
36.0
```

As can be seen, there are many ways to get a problem solved with DTM and using a particular way just depends on someone's preference regarding math. In addition to running commands from the command line, DTM also includes a REPL, which gives a more interactive experience for users to test values they would like to calculate, similar to the experience provided by the Python Interpreter or Node.js. The REPL is represented using ||>.

```
Welcome to DTM!
Enter an expression to be evaluated, else type "quit" to exit.
Note: If you're attempting to run a single program or forgot to add input,
Usage: dotnet run <program>
Example: dotnet run "(+ 1 2 3)"

||> ((1 + 2) + 3)
6.0
||> quit
Exiting...
->
```

Access to a REPL allows for a quicker interaction with the program and allows a user to make quicker adjustments to a program they are trying to test if they are met with any type of error.

3 *Numbers*

Similar to many other general programming languages, the numbers accepted as input in DTM range from rational numbers that can be represented in the form of a decimal. All of the numbers are returned using a float, regardless of

whether or not the value has a remainder value. This allows consistency in values and not having to worry about not being able to get an exact value when doing division, or other float operations. An example of the representation of numbers in DTM is shown below through the REPL, along with what an error would look like if there was a mistake in input.

```
||> 1
1.0
||> -1
-1.0
||> -3.3.
Invalid program.
Cannot parse input at pos 4 in rule 'peof':

-3.3.
   ^
||> -3.3
-3.3
||> 4.5
4.5
```

As mentioned previously, DTM features 5 operations that can be done between numbers, regardless of whether it is prefix notation or infix notation. The examples below show the operations in use:

```
||> (+ 1 2)
3.0
||> (7 - 4)
3.0
||> (2 * 9)
18.0
||> (5 / 0)
ERROR: Cannot divide by 0
||> (^ 4 2)
16.0
```

The example includes a problem in which division by 0 is attempted. It is meant to serve as a reminder that division by 0 is not allowed and show what errors in the REPL would look like. But in general, most operations are handled correctly, whether it is done in prefix notation or infix notation. The example includes a problem in which division by 0 is attempted. It is meant to serve as a reminder that division by 0 is not allowed.

4 *File Input*

Along with taking in direct input from the command line and the repl, DTM supports reading in files written in the DTM language. These files can be run with the extension of "$.dtm$" at the end of it. At this moment, the files do not support the ability to write comments within the code. Along with that, when the file is read in, it only handles one complete expression included. Thus, a file that contains "(+1 2) (1 + 2)" would not be valid, because although each of those are valid expressions for DTM to handle in its own right, the parser, as of now, would try to interpret it as one whole expression to be interpreted to return one result. But that expression would just be seen as incomplete as it isn't bounded by parentheses and there is no operation joining the two sub-expressions.

As a result, a valid DTM file would be made as follows:

1. Name the file with the $.dtm$ extension at the end of it
2. Contain the file to being 1 complete expression, which in most cases would fit in one line of text
3. Save the file, and run it using "dotnet run "nameOfFile.dtm"

Along with that, running a valid DTM would be done as follow:

```
-> more example-1.dtm
(/ (* (+ 1 2) 3) 4)
-> dotnet run "example-1.dtm"
2.25
```

Along with that, file input can only be contained to the command line. The repl currently doesn't support being able to take in a file and evaluate it as it would be done when doing it from the command line.

5 *Infix vs. Prefix*

As mentioned in the introduction, DTM supports the ability to write an expression in terms of prefix notation and infix notation. Infix notation is what humans are most used to when doing math problems, as it is written in a form where two numbers are being handled, and the case in which they are being handled is determined by the operation that lies in between them. Prefix notation is usually handled more so by computers, as they are easier to interpret in computer algorithms. An example of this is the language, LISP, in which all mathematical expressions are written in prefix notation. It is written in the form where two or more numbers are being handled, and the case in which they are being handled is determined by the operation that lies at the front of the list. Here are a couple of examples of expressions written in both, prefix notation and infix notation.

```
||> (+ 1 2)
3.0
||> (1 + 2)
3.0
||> (3 ^ (4 / 2))
9.0
||> (^ 3 (/ 4 2))
9.0
||> (1 + (2 + (3 + (4 + 5))))
15.0
||> (+ 1 2 3 4 5)
15.0
```

Although the two notations are handled differently by the parsers in the language, they can both be used within the same expression as long as the syntax remains consistent with the notation. There are small benefits that come along with using either of the notations. Infix provides the benefit of being easier to read, as it is in line with how most people are used to reading math problems. However, prefix notation allows for a large amount of numbers to be dealt with in a single expression, as the numbers can be listed out without repetition of the operation, as can be seen in the last pair of examples above.