A Look into DTM (Do the Math)

1 *Introduction*

The *Do the Math* (DTM) programming language is a language that can be used to solve math problems. Solvable problems are limited to those including the following operations: addition, subtraction, multiplication, division, or exponentiation. While this may not necessarily need its own programming language, DTM serves as a way to reinforce the order of operation rules and avoid problems that are open to interpretation in how they should be solved. An example of this is a popular math problem that makes its way around social media occasionally, causing people to second guess the way they learned how to do math: $6 \div 2(1 + 2)$. Possible solutions to this problem include 1 or 9, depending on which operations are done first. However, with DTM, there would be no room left for interpretation because the language requires specific grouping of numbers in order to solve a problem. Thus, DTM clears any confusion there may be when solving a math problem and serves as a way to help people learn math.

2 *Design Principles*

As mentioned, a key part of DTM is leaving no room left for interpretation in what order a math problem should be solved in. This is done by having all possible operations between two numbers be enclosed in parentheses. An example for this is as follows: suppose you would like to divide some i, j, and k, which in its simplest form would be $i \div j \div k$. Following the order of operations, in which division is done from left to right, the problem expressed in terms of operations between two numbers would be $((i \div j) \div k)$, which makes it clear that i is divided by j, and their result is divided by k. Where the rules get confusing is in the involvement of parentheses taking the place for multiplication, where $i(j) = i \cdot j$. As can be seen in the example problem of the introduction, it causes issues because parentheses are supposed to take precedence of all other operations. However, since it can sometimes imply multiplication when in the form $i(j)$, in an example such as $6 \div 2(1 + 2)$, after evaluating $(1 + 2)$ as 3, it isn't clear if $6 \div 2$ or if $2(3)$ should be done first.

Another way to combat the confusion of order of operations is with the inclusion of prefix notation in DTM. The examples shown previously were all in infix notation, which takes the form of ¡number¿ ¡operation¿ ¡number¿. However, as the name suggests, prefix notation evaluates a math problem with the operation being done preceding the values, following the form ¡operation¿ ¡number¿ ¡number¿. This helps prevent confusion as there is no need to know the order of operation rules and how they coexist with another. The operation is just completed on all of the numbers that follow it and in the case of multiple operations, they are done from right to left, as the left-most operation relies on the values to the right of it to be done. Prefix notation also includes the advantage of being able to provide a list of numbers with a length greater than 2 after the operation, and have it all be evaluated in the same way without repetition of the operation. In other words, the infix expression $((((((a + b) + c) + d) + e) + f) + g)$ would just be $(+\ a\ b\ c\ d\ e\ f\ g)$ in prefix.

3 *Example Programs*

Here is a list of example problems with their expected output. Each of the programs are included in a directory titled, "examples" in the source code and can be tested using the command `dotnet run "../examples/example-<number>.dtm"`, when in the `lang` directory of the source code.

1. `(/ (* (+ 1 2) 3) 4)` $= 2.25$
2. `(((1 + 2) * 3) / 4)` $= 2.25$
3. `(- 65 (* 40 (^ 3 2)))` $= -295.0$
4. `(((5 * (76 ^ 2)) - 40) + 9)` $= 28849.0$
5. `(1004 - ((29 * 85) / 3))` $= 182.\overline{3}$

6. `(+ (^ 6353 -3) (* (/ 73637 7373) 2))` $= 19.97477282$

In addition to the examples provided in the folder, the DTM language can be tested with unique input, as long as it fits into the form of infix or postfix notation. It can be tested when in the `lang` directory, by typing the following into the command line: `dotnet run "<valid-expression>"`.

4 *Language Concepts*

In terms of primitives, there are only 2 to really be aware of, a number (¡num¿) and a decimal (¡deci¿). Both are represented as floats, which allows outcomes that may result in a float to be returned. With these two primitives, the rest of the available expressions can be formed. There are two type of expressions to be aware of, infix expressions and prefix expressions. An infix expression is in the form of numbers (or decimals) that focuses on evaluating a number and an expression together (refer to syntax for clarification). A prefix expression is in the form of numbers (or decimals) that focuses on evaluating a list of numbers together (refer to syntax for clarification). Both infix and prefix expressions rely on 5 operators, +, -, *, /, and ^. Another key part of the language is that each possible expression is separated by a white-space, which in most cases is a space. Lastly, all possible expressions being joined by operators must be in a set of parentheses, as this prevents any sort of confusion that may occur from order of operation rules.

5 *Syntax*

```
<expr>      ::= (<num> <ws> <op> <ws> <expr>)
              | (<op> <ws> <expr> <ws> <expr>+)
              | <num>
<num>       ::= <d><num>
              | <d>
<deci>      ::= <num><dot><num>
<d>         ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
<op>        ::= + | - | * | / | ^
<dot>       ::= .
<ws>        ::= ␣
```

6 *Semantics*

Syntax	**Abstract Syntax**	**Type**	**Prec./Assoc.**	**Meaning**
n	Num of float	$float$	n/a	n is a primitive. We represent integers using the F# float data type.
d	Deci of float	$float$	n/a	d is a primitive. We represent decimals using the F# float data type.
$(p_1 + p_2)$	In_Plus of Expr * Expr	$float \to float \to float$	3/left	In_Plus evaluates in infix form p_1 and p_2, adding their results, yielding a float.
$(s_1 - s_2)$	In_Sub of Expr * Expr	$float \to float \to float$	3/left	In_Sub evaluates in infix form s_1 and s_2, subtracting their results, yielding a float.
$(m_1 * m_2)$	In_Mult of Expr * Expr	$float \to float \to float$	2/left	In_Mult evaluates in infix form m_1 and m_2, multiplying their results, yielding a float.
(d_1/d_2)	In_Div of Expr * Expr	$float \to float \to float$	2/left	In_Div evaluates in infix form d_1 and d_2, dividing their results, yielding a float.
$(e_1 \hat{} e_2)$	In_Expo of Expr * Expr	$float \to float \to float$	1/left	In_Expo evaluates in infix form e_1 and e_2, raising e_1 to the power of e_2, yielding a float.
$(+\ p_1\ p_2 \dots)$	Pre_Plus of Expr list	$float \to float \to float$	3/left	In_Plus evaluates in prefix form a list of p's, adding their results, yielding a float.
$(-\ s_1\ s_2 \dots)$	Pre_Sub of Expr list	$float \to float \to float$	3/left	In_Sub evaluates in prefix form a list of s's, subtracting their results, yielding a float.
$(*\ m_1\ m_2 \dots)$	Pre_Mult of Expr list	$float \to float \to float$	2/left	In_Mult evaluates in prefix form a list of m's, multiplying their results, yielding a float.
$(/\ d_1\ d_2 \dots)$	Pre_Div of Expr list	$float \to float \to float$	2/left	In_Div evaluates in prefix form a list of d's, dividing their results, yielding a float.
$(\hat{}\ e_1\ e_2 \dots)$	Pre_Expo of Expr list	$float \to float \to float$	1/left	In_Expo evaluates in prefix form a list of e's, raising e_1 to the power of e_2 and so on, yielding a float.

7 *Remaining Work*