DragginMath: About Writing Math on a Computer

DragginMath was designed to help you understand algebra on your iPad. We hope you are doing algebra on paper at the same time. Most of what you write in DragginMath looks the same as what you write on paper, except perhaps that it is neater. A few things look a little different, and you will have to make minor mental adjustments as you move between your iPad and your math books. We can wish they looked exactly the same in all ways, but some wishes just won't work the way we want. This is one of those.

Why can't they all look the same? Because you can make any marks you like with pencil and paper, and you can arrange them any way you like. The traditional notation of mathematics makes good use of that. But when people started writing math on computers in the 1950s, the only way to write *anything* on a computer involved machines you have probably never seen: the teletype and keypunch. These *can't* make any marks you like, and they arrange marks in only *one* way: in a straight line. People had to find ways to write mathematics with only 95 characters, all of the same size and all written in a line. Here are those 95 characters:

```
  ! " # $ % & ' ( ) * + , - . /
0 1 2 3 4 5 6 7 8 9 : ; < = > ?
@ A B C D E F G H I J K L M N O
P Q R S T U V W X Y Z [ \ ] ^ _
` a b c d e f g h i j k l m n o
p q r s t u v w x y z { | } ~
```

As computer designs matured, it became possible to write

more than just those 95 χαρακτερς, to have $_{\text{subscripts}}$ and $^{\text{superscripts}}$ and `different` ***fonts***, to write math like one reads math in textbooks. Writing standard typeset math notation on a computer is now *possible* for the average user. But telling your computer to make these marks on the screen is not the same as *doing the math*. Simpler notations, developed in the early days of computer programming, are still preferred when *interacting* with computer programs. That includes the computer program you are using now: DragginMath.

Differences from standard textbook notation were not undertaken lightly when designing this app. Such differences have to accomplish these things:

- They have to make more sense than the alternatives.
- They have to be easy to draw, if you need to draw them.
- They have to increase the reach of what DragginMath can do.

We hope you agree: the differences are small and worth the effort on your part.

Variables

You can think of a variable as a bucket or a box with a name painted on the outside and a number hidden inside. In an equation, a variable behaves like a number, even if we don't know which number it is. In DragginMath, as in most of traditional mathematics, variable names are *always* single lowercase letters. The usual example is **x**.

Multiplication ∗

Multiplication is written several ways in traditional math books. For example:

ab a×b a·b

These all mean the same thing: *a times b*. But there are problems. That *implicit multiplication*, **ab**, is convenient when you multiply a variable by a variable, or a number by a variable. But it doesn't work when you multiply a number by a number. For example, **23** always has to mean *twenty-three*, not *two times three*. And if you are writing on paper and your handwriting is less than excellent, a reader (including yourself) may find it hard to know if you meant to write **a×b**, or **axb**, which means something entirely different. If you write a dot to multiply **2·3**, it may be hard to know if that is what you meant, or if it was supposed to be a decimal point instead.

When computer programming languages were first developed in the 1950s, designers often chose the *asterisk* ∗ as the multiplication symbol. This is convenient because it is one of the characters on a standard mechanical keyboard. The asterisk became the multiplication symbol in most computer software, and DragginMath does this, too: **a∗b**. Even some school books have caught up with the times and teach this now.

People often say *star* when they mean *asterisk*. It is easier to pronounce.

In DragginMath, you can write implicit multiplication, **ab**, in most situations where implicit multiplication traditionally works. There are some situations where *explicit* multiplication *must* be written, for example: **2∗3**, **a∗↑b**, **a∗√b**, **a∗↓b**. The last

three cases are because the raise↑, root√, and log↓ operators have both binary and unary forms, so **a**√**b** and **a**∗√**b** don't mean the same thing in DragginMath. If you don't know what all those words or symbols mean, you will soon.

You will *never* see implicit multiplication in DragginMath *diagrams*, where it would serve no purpose. In a diagram, multiplication is always drawn with ∗ regardless of how you typed it in.

Division ÷

Division and fractions mean the same thing. When writing algebra on paper, always write division like this:

$$\frac{1-a^2bc^3}{4de^5+6}$$

This way of writing fractions is helpful when doing algebra on paper. But when writing math as we do on a computer, all on one line, this has to change. DragginMath was designed to show algebra in a particular way on an iPad screen, and *this isn't it*. Always remember: *how algebra looks* and *what algebra means* are not the same thing. It can be written many different ways, but it always *means* the same thing. It is OK to write it one way on a computer and another way on paper as long as you understand what it means.

Most computer software uses the *slash* / to represent division. DragginMath uses the *obelus* ÷ instead. It is easier to read in the diagrams, and this is what you will find on the screen keyboard. If you have a hardware keyboard attached to your iPad, you may enter / for division, which converts to ÷

automatically.

Note that / is one of the 95 characters, but ÷ is not.

Raise ↑

Traditional mathematical notation represents *raise* (exponentiation) as x^n. Other arithmetic operators + – ∗ ÷ have explicit symbols, but raise is traditionally represented, not with a symbol, but with this *typographic convention*: x^n. You are expected to *figure out* which operation is there because of *how the characters are written*, where the larger character on the left is the *base*, and the smaller character up and to the right is the *exponent*.

Today's computers can make characters that look like that. But typing it on a keyboard is still cumbersome, so most computer software represents raise differently. DragginMath represents x^n as x↑n. You can find the ↑ character on DragginMath's screen keyboard. If you have a hardware keyboard attached to your iPad, you may enter a *caret* ^ for raise, which converts to ↑ automatically. The use of the caret for raise is not as common as the use of the asterisk for multiplication, but you are still likely to encounter it elsewhere: in most spreadsheets, some programming languages, and Google's calculator feature. If you find yourself using unfamiliar computer software and you don't know its symbol for raise, the caret is a good bet.

Note that ^ is one of the 95 characters, but ↑ is not.

Root √

Some operations are traditionally represented with special symbols *and* typographic conventions working together. For example, √ represents the square root (2nd root), ∛ represents the cube root (3rd root), ∜ represents the 4th root, etc. For the most commonly used roots, this is fairly easy to type these days. But if you need the (3x+4)th root, that may be hard to type even if you know what you're doing. DragginMath solves this problem by thinking of √ as an *operator* instead of as a traditional *function* that *just happens* to be written with the special character √ (called a *radical*). As a binary operator, **n√x** means the nth root of x. ***This is different from traditional notation***, in which **n√x** means *n times the square root of x*. If that is what you want in DragginMath, then write **n∗2√x** or **n∗√x** (be sure to read about Unary Operators, below). We couldn't get around this without crippling DragginMath.

Note that √ is not one of the 95 characters. Until recently, roots had to be written on computers by spelling them out, usually abbreviated as **`sqrt(x)`** or **`cbrt(x)`** or something like that. In most computer software, they still are.

Log ↓

As with roots, traditional notation represents logarithms with symbols *and* typographic conventions: $\log_{\mathbf{n}}\mathbf{x}$. DragginMath represents logarithms as an operator using ↓. Why? Because log↓ is the inverse of raise↑. As a binary operator, **n↓x** means $\log_{\mathbf{n}}\mathbf{x}$ (log *base n* of x). So if you want $\log_{\mathbf{10}}\mathbf{x}$ (the *common log*), write **10↓x**. As a unary operator, **↓x** means $\ln \mathbf{x}$ (the *natural* log), equivalent to $\log_{\boldsymbol{e}}\mathbf{x}$.

Note that ↓ is not one of the 95 characters. In most computer software, logarithms are written by spelling them out, usually abbreviated as **`log(n,x)`** or **`ln(x)`**.

Factorial !

Factorial is traditionally written as a *postfix* unary operator: **n!** The definition is **n! = n∗(n-1)∗(n-2)∗…∗3∗2∗1**. On a current iPad or iPhone, **20!** is the largest factorial that can conveniently be evaluated. Attempts to evaluate larger values will remain in symbolic form. **0!** is 1. Factorials of negative integers are all **?** (*NaN*, or *Not-a-Number*, see below).

Factorial has no widely accepted inverse. But DragginMath needs one, so it defines the postfix unary operator **Antifactorial ¡** (this has actually been suggested elsewhere). Also, it defines **Permute** Ⓟ and **Combine** Ⓒ as binary operators. If you are already familiar with basic combinatorics, this will come easily to you.

Absolute Value ‖

Absolute Value is traditionally written with *vertical bar notation*, where the vertical |bars| act something like parentheses, as in **|x+3|**. DragginMath uses ‖ as a unary operator instead. We tried to do it the traditional way, but making that work is incompatible with other important DragginMath behaviors. Also, one can notice that no programming language has ever implemented vertical bar notation. There is a reason for that. Make a list of the parts of traditional notation that should never have happened, and this is at the top of the list. That

doesn't mean we can discard the *idea* of absolute value, but we can legitimately want a different way to write it. DragginMath chooses ‖. You can find the ‖ character on DragginMath's screen keyboard. If you have a hardware keyboard attached to your iPad, you may enter a single *vertical bar* | for absolute value, which converts to ‖ automatically.

Note that | is one of the 95 characters, but ‖ is not. In most computer software, absolute values are written by spelling them out, usually abbreviated as **`abs(x)`**.

Plus-and-Minus ±

Plus-and-Minus ± is closely related to the idea of absolute value, being something like its inverse. If you ask DragginMath to solve an equation containing ‖, it gives you an equation containing ±, and vice versa.

DragginMath implements ± in both binary and unary forms. Most operators create only one result, but ± creates two. For example, **±3** means 3 *and also* ¯3, and **7±2** means 9 *and also* 5. How do you see these dual results? Double-tap the ± to obtain two expressions: one for the + case, and one for the − case.

Note that ± is not one of the 95 characters. It is often seen in math, science, or engineering books. But even now, it is hard to find computer software that deals with this idea *at all*.

Unary Operators

DragginMath diagrams are based on the concept of *operator trees*. There are two operator categories here: *binary* and *unary*.

Addition is an example of a binary operator. In traditional notation, we write this as **a+b**, where the + symbol represents the *operator*. Also, there are two *operands*, one on each side, where an operand can be a number, a variable, or the result of another operator. It is the fact that there are *two* operands that makes + a *binary* operator.

For *unary* operators, there is an operator symbol and only *one* operand. Depending on which operator it is, the operator symbol might come before the operand (a *prefix* operator) or after it (a *postfix* operator). DragginMath follows traditional notation here in those cases for which a tradition exists. Where no tradition exists, DragginMath has to be inventive.

Binary and unary operators often come in pairs. An example is – (dash), which can mean either *subtraction* (the binary operator, as in **a–b**) or *negation* (the unary operator, as in **-b**, which is properly read as *negate b* or *the opposite of b*). Some people (and DragginMath) write those as two different kinds of dashes, but most of us just figure it out from context. Most unary operators look exactly the same as their binary counterparts, while some are easily seen as different. You can often think of a unary operator as being the same as its binary counterpart but with an *implied operand*. An example is once again –, where **-b** means the same as **0–b**. Each unary operator has its own implied operand. If it isn't what you want in a given situation, you have to write the binary form.

Binary	Unary	Same As	Unary Name
a–b	-b	0–b	Negate
a±b	±b	0±b	Plus-and-Minus
a÷b	⅟b	1÷b	Reciprocate
a√b	√b	2√b	Square (2nd) Root

a↑b	↑b	***e***↑b	Exponential
a↓b	↓b	***e***↓b	Natural Log

Note that ***e*** in this table is the special number ***e*** (approximately 2.71828…), not the variable **e**. Yes, there is a *very* good reason for this. Expressions like e^x are found all over mathematics, physics, chemistry, biology, etc. And some say the *natural log*, or log *base* ***e*** (traditionally written $\ln x$ or $\log_e x$) is the only log that matters, which is almost true. The number ***e*** is so important, it has its own key on DragginMath's screen keyboard, right next to other special numbers like **π** (approximately 3.14159...).

DragginMath can always tell the difference between unary and binary operators, even when they have the same symbol. So can you. Because of this, they do not have separate keys on the screen keyboard. For example, if you enter ÷ when a unary operator is allowed, you will get ⅟ instead.

If you ever need to convert a unary operator to its binary counterpart in a diagram, flick down on it with your fingertip. To convert a binary operator to its unary counterpart, drag the implied operand you want to go away onto its operator.

Not all unary operators have binary partner operators. Absolute Value ‖ and Factorial **!** are examples.

Operator Precedence

For beginners, one of the most annoying things about traditional mathematical notation is *operator precedence*, which means that some operators *stick together* more tightly than

others. Whether this was ever a good idea or not can be argued, but it is now a fixture in the way we write algebra, and in most computer programming languages, too. After getting used to it, most people decide that it might even be a good thing. Meanwhile, there it is, and it is about to become a part of your life.

Here is an example: what is **2+3∗4**? Don't look... Figure it out... Do you have an answer now? Is it 20, or is it 14? Actually, it is 14, because multiplication has *higher precedence* than addition. That means multiplication *sticks together* more tightly than addition, so you have to do the multiplication before the addition. Why? *Because...*

If you insist on having an actual reason, try this: If you are doing arithmetic because your teacher told you to practice doing arithmetic, operator precedence doesn't make much sense. But if you are doing arithmetic because you are working with useful numbers outside of a math class, operator precedence almost always describes how things group together naturally. So operator precedence makes *most* practical math *easier to think* and *easier to write*.

With some practice in reading, operator precedence makes the structure of most equations easier to see, and the structure of equations matters a lot. *DragginMath was designed and built to show you the structure of equations*. It is the reason this app exists.

Here is a simplified precedence table:

Parentheses ( )
Negate Reciprocate (Unary only) - ⅟
Raise Root Log (Binary & Unary) ↑ √ ↓
Multiply Divide ∗ ÷
Add Subtract + –

That is the simplified table. The real table is a little more complicated and needs more explaining. Showing all the details of how operator precedence works is hard to do in writing. It is better to see it happening. DragginMath shows you how it works, step by step, every time you enter an expression.

Some people think *implicit* multiplication, **ab**, should have higher precedence than *explicit* multiplication, **a∗b**, or division, **a÷b**. For example, **a÷bc** would mean **a÷(b∗c)** instead of the standard **(a÷b)∗c**. Some books are even written this way (mostly physics books, and not even very many of those). Because this is not standard, such books make a point of telling you about it *right away*. If you encounter a book like this, you will have to be mentally flexible enough to deal with it. We experimented with making this something you could choose in DragginMath, which does *interactive* algebra. It caused problems that math doesn't have when it is just sitting there on paper. We took this feature out.

Remember: this issue of operator precedence is only about how we *traditionally* write algebra. There are other ways it *could* be written, and there can be good reasons for wanting to write it differently. But knowing how to read and write the traditional notation allows us to communicate freely across the current mathematical world, several hundred years into the past and probably at least that far into the future. Nevertheless, how we *write* algebra has little to do with what algebra *means*.

Numbers

There are many different kinds of numbers. DragginMath only knows how to work with *integers*: the positive and negative

whole numbers. This is intentional.

Arithmetic *as a computer does it* and arithmetic *as algebra does it* are not really the same thing. For example, in an algebra class, √**50** is not 7.071067..., it is **5**∗√**2**. This is a shock to some people, but holding a calculator in your hand will *never* be a substitute for understanding algebra. DragginMath tries hard to do algebra-style arithmetic, which is not something computers do any more naturally than you do. And because arithmetic is infinite and this is a finite computer, there are practical limits on what DragginMath can do. In particular, if you ask DragginMath to factor or divide very large integers or compute their roots, you may run into problems. To perform these operations efficiently, DragginMath keeps something called a Factorization Cache, which can consume a lot of computer memory. If you decide you want to find the limits of DragginMath, you will succeed, probably because you exploded the Factorization Cache. But that will not do anything for you that really needed doing, and you will have to restart DragginMath afterward to make it usable again. If you use DragginMath to work on the problems typically found in algebra textbooks or in real life, you will not have such difficulties.

It was mentioned earlier that – (dash) means two things in standard notation: subtraction and negation. Actually, it means three: subtraction, negation, and *negativity*. Subtraction and negation apply to both numbers and variables, but negativity only applies to actual numbers, for example, ¯7. It is a property of the number itself, called its *sign*.

If you encounter a number with a negative sign attached to it, you can convert the sign to the equivalent unary operator by flicking down on the number. To collapse unary operators back

into a number, drag and drop the number up over those operators. Any redundant operators are automatically and correctly removed.

Some families have secrets they would rather not talk about, and the family of Numbers has those, too. For example, what is **1÷0**? A typical answer from a typical math teacher is that **1÷0** is *undefined*, at which point a typical math student is expected to go away and not ask more questions like that. But computer programmers actually need an answer to that question. Without it, computers halt, important machinery misbehaves, money is lost, and people get hurt or die. Really. And then there is a related question: what is **0÷0**? That is *undefined*, too, but it is a different kind of *undefined*. Things like this will become increasingly visible to you in your life, because the computers you use every day deal with these things now, even if traditional mathematics (with good reason) doesn't like to and never will.

In DragginMath, for any x > 0, **x÷0** = ∞. That sideways-8 symbol represents *Infinity*, otherwise known as Really Really Really Large. In some ways, ∞ behaves like a number. In other ways, it doesn't. For example, ∞+**1** = ∞, ∞–**1** = ∞, ∞∗2 = ∞, ∞÷**2** = ∞, and other strange things. But **1÷0** is not *undefined* here. It is ∞. Some other operations with the wrong operands result in ∞, also.

In DragginMath, **0÷0 = ?** That question mark represents *Not-a-Number* (or *NaN*). This name, Not-a-Number, may not be the best name for this idea, but the people who invented this name couldn't think of a better one, and they tried. **?** is even stranger than ∞. We know that ∞ is Really Really Really Large, but we simply have no idea what **?** is. It is not large. It is not small. It is not positive or negative, real or imaginary. It may be

something, but whatever it its, it is Not-a-Number (it *sometimes* turns out to mean *any number* or *all numbers*, but don't count on it). Some other operations with the wrong operands result in **?** also. Any operator that is given **?** as an operand returns **?** as a result. Usually, once **?** appears in a computation, everything else quickly becomes **?** too.

This topic is larger than we can talk about here. But ∞ and **?** give us ways to talk about things we otherwise could not talk about at all. Do not let this make you complacent about these things. If you ever encounter ∞ or **?** in DragginMath or anywhere else, this is probably something you need to think about, and maybe even worry about. But at least your worries will not be *undefined*.