An Operators Manual for Algebra

How to use algebra

2023-08-10

James B. Wilson
Department of Mathematics
Colorado State University

James.Wilson@ColoState.Edu

James.Wilson@ColoState.Edu

Contents

What is algebra? 1

Algebra is the study of equations, for the most part equations involving variables. That is because applications have unknowns and if the shape of the equation can tell us anything about the options to solve it, we shall want to take advantage of this. Witness how we solve $ax^2+bx+c=0$ in one strategy despite it covering an infinite number of equations.

You probably already know every color, shape, and pattern of equation you will ever need. Compare these two equations

$$x^2+y^2 \equiv 0 \pmod{541} \qquad \frac{\partial^2 f}{\partial x^2}+\frac{\partial^2 f}{\partial y^2}=0.$$

Can you stop yourself from seeing them as related? These equations concern entirely different things. On the left 0 can equal 541. On the right 0 is functions on the xy-plane. Yet, the similarities as equations shine through. Why? Is it because 0, $+$, and powers of 2 are general concepts, "abstractions". This doesn't connect them to polarizing art movements nor render the concept inapplicable. Abstract here, and everywhere, means to study by limited attributes. That's how we do all math and science. So when we abstract the equation on the left we forget about mod 541 and the precise meaning of these numbers. On the right we forget about functions and the notions of derivatives. We are left with just 0, $+$, and squares and where they sit. We abstract both to a common equation

$$x^2+y^2=0.$$

In fact, even the equality was an abstraction which could vary from context to context. With such flexibility a small number of symbols and their grammar are enough to capture the huge variety of equations we encounter in real life.

Now since every symbol in an equation is a variable we have new powers to solve equations. Look to the humble

$$x^2+1=0.$$

By our own view, right now this is nothing but variables, so it means nothing to solve this. But, drop this into a context such as decimal numbers $\mathbb{R}$ and the understanding is to replace $1 := 1.0$, $0 := 0.0$, $+$ is substituted for addition of decimals, and square is by multiplying decimals. Equality now means two equal decimals, or in practice two decimal numbers that are close enough to be considered equal. The only remaining unknown is x, but as everyone knows, $0 \leq x$ or $x < 0$ so in both cases $-1 < 0 \leq x^2$.

The power of variable everything is that we are not stuck with the real numbers. Let us replace everything with complex numbers $\mathbb{C}$. Substitute $0 := 0+0i$, $1 = 1+0i$, $(a+bi)+(c+di) := (a+b)+(c+d)i$, and $(a+bi)^2 = (a^2-b^2)+2abi$. Now we find $\pm i$ are the solutions. Solutions do exist! Since they exist we can return to a problem and ask if the solutions we found will do the job. Maybe not, or maybe we should revisit our models and see these solutions as predicting the presence of previously unknown realities.

Why stop here? Quaternions have $\hat{\imath}^2 = \hat{\jmath}^2 = \hat{k}^2 = -1$; so, at least 6 solutions to $x^2+1 = 0$. Even more. Try (2×2)-matrices, I bet you can find infinitely many matrices M where $M^2 = -I_2$. This is the method of algebra: dream up new numbers that might be used to solve equations. Alter their properties, e.g. drop the order of real numbers and you can get complex solutions. Drop commutative multiplication of complex numbers and you might get infinite solutions. Learn enough by this process and we can begin to predict if solutions are to be expected and fathom algorithms to find them when they do exist. When the solutions become infinite we find ways to parameterize them with smaller data such as a basis.

Don't forget equality is variable too! Suppose we wanted to solve $x^{541}+x+1 = 0$ using only integers. Replace equality with $\equiv$ modulo 2 and ask for a whole number x that solves

$$x^{541}+x+1 \equiv 0 \pmod 2.$$

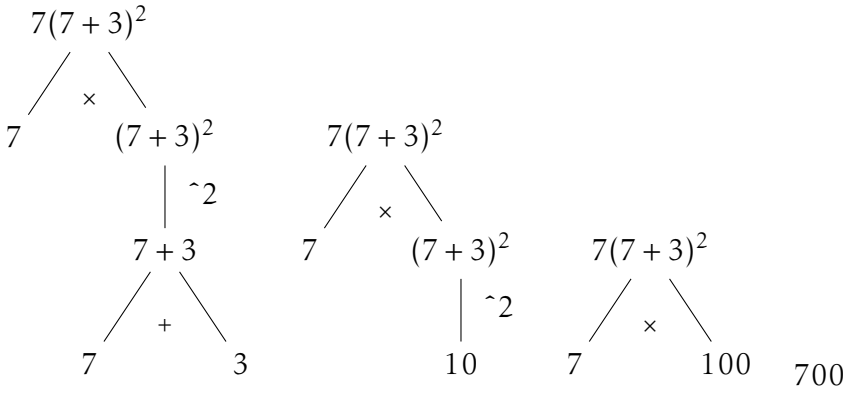
All integers in this equality become either 0 or 1, but neither will solve this equation. By varying equality we confirm there are no solutions.

The power of algebra is that every symbol in an equation is a variable, especially the equals sign.

Grammar and Induction 2

> "... small number of symbols and their grammar are enough to capture the huge variety of equations..."

Ask yourself how you know what to do when asked to calculate $7(7+3)^2$. Parenthesis, Exponents, Multiplication, Division, Addition, Subtraction (PEMDAS) correct? Whether obvious or not, this is parsing grammar. We can make this visual with a *parse tree*.



We can read the tree like step-by-step instructions. Start at the leaves and join them by whatever operation is displayed on adjacent branches. We start at the bottom with $7, 3$, and join them as $7+3$ (computing 10), then the next step is to square (now 100), then multiply by 7, we reach 700.

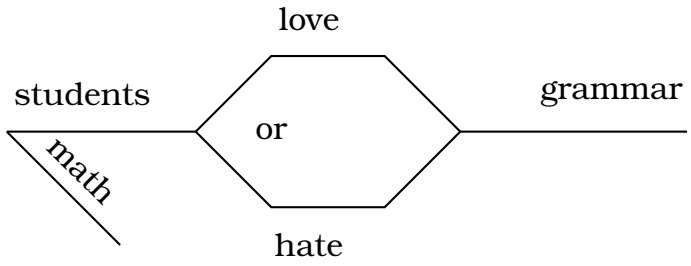
You may have been taught induction through stories of falling dominos. Good. But what if induction was more like climbing, and the domino illustration was bottling up the experience of climbing stairs? Surely its more fun to climb trees and mountains. Climbing can go up (induct) or down (recurse), but of course I drew the tree upside-down; so, the metaphor must be rotated. My goal is to turn all of your views on induction on their head. There are many inductions, one for each free algebra, but that's the forest and we should look first at some more trees.

2.1 Grammar

Climbing could find many routes, even go in cycles, but evaluating a formula was a precise algorithm without ambiguity. The reason was that we had a tree. Trees have unique paths between any to vertices. So if we start at the bottom we have a unique direction to reach the top. Recursion on trees is so special it has a special name: *traversing* the tree.

Parsing grammars in natural language is not always that uniform. English grammar may have cycles.

"Math students love or hate grammar."



In mathematics two paths to the same place are said to be a relation, they are related. So systems that have no cycles, i.e. trees, are free of relations, or simply *free*. What is free in this case is the grammar we used for arithmetic formulas. That is not be confused with saying that we couldn't some how rewrite a formula to mean the same thing. For example $7(7+3)(7+3)$ is a different formula with the same result. But if we diagrammed that formula as parse tree it would quite different, and it would not have a cycle.

That we got a tree in math formulas means we have a rather boring grammar, what Chompsky's *Syntactic Structure* calls *context-free* grammars.[1] Don't be too disheartened. Virtually every programming language has a context-free grammar and programs can communicate a lot of hefty ideas.

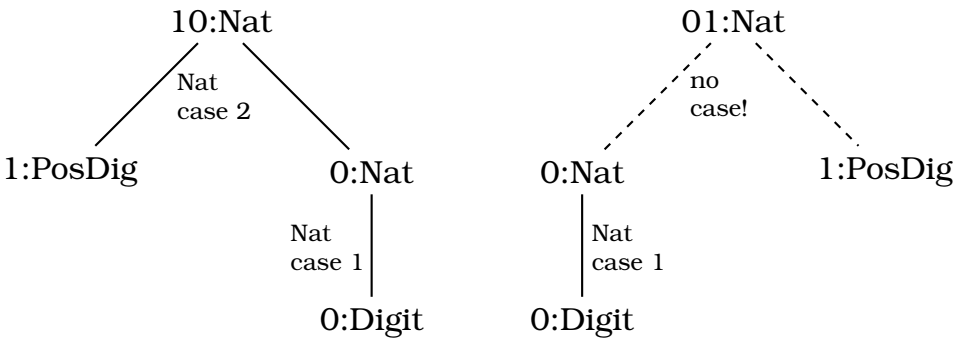
[1] If an algebraist starts a talk with a story that "...It was thought all natural languages were context-free until some obscure dialect in the alps or Africa was found...", then tune out until they return to equations. Linguist never had such illusions. Even english is not context-free, read James Higginbotham.

To specify a context-free grammar we specify the atoms of our language. For natural numbers we can used digits $0, 1, \ldots, 9$. Then we write down patterns of how to combine digits. The separate cases are designated by the auxiliary symbol '|' (which reads as "or"). Each rule is given a name called a *token* (or *tag*) and denoted `<Name>`. Since the Walrus := is our assignment of variables, we use the "astonished Walrus" ::= as assignment of token rules. Listing 2.1.1 shows the grammar for natural numbers as digits.

Listing 2.1.1 The grammar for natural numbers.

```
<PosDig> ::= 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
 <Digit> ::= 0 | <PosDig>
  <Nat>  ::= <Digit> | <PosDig><Nat>
```

The natural number grammar is said to *accept* 0 as a digit, written "0:Digit" and also as a Nat, "0:Nat". It also accepts 541:Nat and 10:Nat but it rejects "01:Nat" as that case does not exist as a rule for any token in this grammar. Deciding to accept or reject is recursion, work down until you can decide. Induction can only ever deliver a sentence accepted by the grammar. The strings accepted by our grammar are known as the *language* for that grammar. They are the source of one of the most important algebras out there, the free algebras.



Remark 2.1. The symbols `::=`, `<Token>` and `|` are Backus-Naur Form (BNF) notation, which is popular in computer science. It is actually subject to its own gammar, admittedly basic and fixed in length. But you can be forgiven for wondering if this is all circular reasoning. When this occurs, mathematicians like to attach the word *meta*, which means literally "self-referential". So BNF would be called a *meta-language*. Sometimes self-referential can be turned into paradoxes (Russell's paradox, Gödel's Incompleteness, Turing's Halting problem). So you may worry. But I suppose if you didn't believe in language, why would you be reading?

Back to Bakus and Naur, these ideas under different notation were written down in 600 BC by Panini's rules of Sanskrit grammar. Good ideas get rediscovered; great ideas get stolen.

Vocabulary.

The rules in the grammar are often described as *production rules*. For example, `<Nat> ::= <PosDig><Nat>` produces a natural number by taking in a positive digit and a natural number. We call it a *binary* production because it takes two inputs. We also call it *heterogeneous* because it takes in data of different types. (If all the inputs and outputs are of the same type we call the production *homogeneous*.) Some productions take in only one input, so-called *unary*, such as squaring a number, or the first case of `<Nat>` which takes in a digit and is said to *promote* the term to a natural number.[2] Productions that require no preexisting data, such as the digits $0,\ldots,9$, are called *nullary*, *atomic*, or *terminal*.

2.2 Multi-sorted grammars.

To add depth to language we sort the allowed productions. In natural languages this may be verbs and nouns. In mathematics the typical first sort are constants, `<Nat>` for instance. The next

[2]Promote improves over the alternative *coerce*, which in turn replaced the use of "caste"—an all too casual allusions to a discriminator societal system.

sort are variables, e.g. $x, y, \ldots$. These may depend on constants, such as numbering variables $x_1, x_2, \ldots$.

```
<Var> ::= x | y | x_<Nat> | y_<Nat>
```

A third sort might be operators, e.g. $+, -, \times, \div$. Symbols like parenthesis, braces and other groupings are usually considered as part of the meta-language and used to clarify how to read formulae, but not specifically part of any formula. Like "set", "sort" now has this technical meaning: one of these designations in that language. So a sort consists of types.

Example. For Boolean (true/false) algebra we have true '$\top$', false '$\bot$', and '$\wedge$', or '$\vee$', and not '$\neg$' with the following grammar which includes parenthesis. See Listing 2.2.1. For example, $\top \vee (\neg(x_2 \wedge y) \vee x_1) : \mathit{Bool}$ but Bool rejects $\bot\neg$.

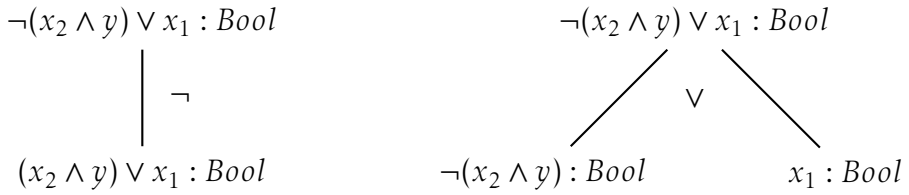
Listing 2.2.1 A Boolean algebra grammar.

```
<Var>  ::= x | y | x_<Nat> | y_<Nat>
<Bool> ::= ⊤
         | ⊥
         | <Var>
         | ¬ <Bool>
         | <Bool> ∨ <Bool>
         | <Bool> ∧ <Bool>
         | (<Bool>)
```

What we have built is very nearly the *free boolean algebra of countable rank*, or sometimes known as 1st order logic. The missing ingredient (relations) will come later. What this demonstrates is how close to language algebra really is. You start with some desired operators, some constants, and some variables and you build it formulas by induction. If you insist on repeating the use of common symbols like $+, \times, -, \div$ and etc. you will end up with formulas that look like polynomials from high school. It is really one language with different dialects.

2.3 Order of operations.

There is something still missing. Try reading $\neg(x_2 \wedge y) \vee x_1$. Is this meant to negate $(x_2 \wedge y) \vee x_1$ or is it to negate $(x_2 \wedge y)$ and then or that with x_1? This grammar offers two associated parse trees.



Order of operations steps in to resolve the ambiguity. One option is *left-most outer-most (LeMOM)*, which—similar to English language, reads from left to right and assumes we start to match the pattern from the outer most symbols. So $\neg(x_2 \wedge y) \vee x_1$ in LeMOM means $\neg$ is the symbol we need to parse first. That requires us to build a `<Bool>`. So we move left and read $(x_2 \wedge y) \vee x_1$. Here the LeMOM is '(' which will need to match with `<Bool>)`. Reading further $x_2 : Var$ which promotes to $x_2 : Bool$, but this is not followed by ')' so it must continue left-ward. Next is $\wedge$ which matches with $x_2 : Bool \wedge y : Bool$ so that we have $x_2 \wedge y : Bool$. Then we hit ')' which matches our earlier '('. Now we finally get $(x_2 \vee y) : Bool$ which finally matches with $\neg$`<Bool>`. We have parsed $\neg(x_2 \wedge y) : Bool$. At this point we have parsed $\neg$ and so we continue with $\vee$ to match $\neg(x_2 \wedge y) \vee x_1 : Bool$ with the right-hand-side parse tree.

LeMOM parsing is favored in mathematics and computer science because we can decide what things mean the moment we encounter them in the string. There are other orders of operations, for example right-most inner-most (RiMIM) or the high-school PEMDAS (Parenthesis, Exponential, Multiplication, Division, Addition, Subtraction). These allow for short formulas to be more expressive (jargon to mean they communicate more than a similarly long string in a less "expressive" language).

Grammar plus order of operations are a language to express an induction.

Exercises

1. Write the grammar for decimal numbers.
2. Write the grammar for a 4 function calculator with decimal numbers and $+, -, \times, \div$.

2.4 Paradox of the Trapped Variable.

Think back to substitution in $x(x+3)^c$. Suppose we use the rules to substitute $c := 2$, we get $x(x+3)^2$, $c := \pi$ gives use $x(x+3)^\pi$. For $c := d$, we want $x(x+3)^d$. Now try $c := x$. Would you really convert this to the function $x(x+3)^x$? Maybe, but it is more likely that we see x and c as distinct, that is, c is constant to x. Something about substitution does not follow induction as cleanly as we have described so far. If you don't see this as a problem yet lets look at something more basic.

Lets slow down to see what happened. Set:

$$I(x) = x \qquad\qquad K_c(x) = c.$$

You may call I the identity function and the K constant functions. Try some substitutions, I tried $I(3) = 3$, $I(\clubsuit) = \clubsuit$. I found $K_3(2) = 3$ and $K_3(\clubsuit) = 3$ as well. I even tried $K_\clubsuit(2)$ and got $\clubsuit$. I changed x for y, $I(y) = y$, and d for c, $K_d(x) = d$. Next I substituted x for c and got

$$K_x(x) = x = I(x).$$

Now we have a true problem: a constant function should not equal an identity function.[3] This is the paradox of the trapped variable.

As my philosophy colleague Professor Dustin Tucker says, "A system studied long enough reveals its paradoxes." Paradoxes (para = distinct + dox = opinion) are inconsistencies that you can avoid by revisiting the scope of your definitions. Just withhold some options and you wont end up with two different options. It

[3]Functions in this sense are so primitive they have no domains and codomains. You can put anything into these functions.

is not a philosophically satisfying resolution, which is why most paradox hacks lead to schisms.

So what is the root cause of our paradox of the trapped variable? The answer are bound (local) verses free variables.

To avoid trapped variables I use the rules set out by Curry-Feys. First you need a grammar. Keeping to context free and learning by example

Why did something so basic fail? It has to do with variables coming in two forms: free and bound. If you program you might think of a global verses local variable. First things first: variables? First sort out the data. One sort will be constants, atoms like an alphabet, maybe digits, or a word or special symbol. A second sort will be called variables. That its, variables are symbols from special alphabet we call variables. This means that a variable can never equal a constant, statements like $x = 2$ are in strict sense nonsense. But hold off on that journey for a moment. Now having all these alphabets we can form strings using the various letters. We could define arithmetic using digits $0,\ldots,9$, $+,-,\times,\div$ and some variables but lets simplify things to true/false which is long enough to explore the idea. We need to specify a grammar of how allowed formulas can be made. We basically separate the options by | (reads as "or") and use patterns to explain structures that are built up recursively. So for Boolean (true/false) algebra we have true $\top$, false $\bot$, and $\wedge$, or $\vee$, and not $\neg$ language might be defined like this.

```
<var>  ::= x | x_<int>
<Bool> ::= ⊤
         | ⊥
         | <var>
         | ¬ <Bool>
         | <Bool> ∨ <Bool>
         | <Bool> ∧ <Bool>.
```

To get started lets return to our use of a grammar. The diagram we had was a tree, what is known as a *parse tree*. It is the same thing you do when you diagram a sentence in grammar school, only with English you can sometimes get cycles. That we got a tree is owed to the fact that the grammars for mathematics are basic and gentle, what Chompsky calls *context-free* grammars.

This situation comes about because of two flavors of variables: free and bound, also called local. A variable can be bound in many ways, for example $\forall x$ binds x to $\forall$, same with $\exists x$. The binding tells us that even if we are using x somewhere else, form this point till the end of the block we are simply recycling the name x, but its meaning is now controlled by the start of the binding. The binding in the substitution examples above is hidden by notation but it is third form known as λ-binding, such as $x \mapsto x + 2$ (historically $\lambda x.(x + 2)$ which is where the name comes from). This says that x's role is to serve as the variable in describing a function. In the constant function $c \mapsto K_c$ or rather $c \mapsto (x \mapsto c)$. Likewise $\sqrt{n}u$ means $n \mapsto (u \mapsto \sqrt[n]{u})$ Now the point is that a local variable is just reusing a symbol it has no visibility outside.

Why did something so basic fail? It has to do with variables coming in two forms: free and bound. If you program you might think of a global verses local variable. First things first: variables? First sort out the data. One sort will be constants, atoms like an alphabet, maybe digits, or a word or special symbol. A second sort will be called variables. That its, variables are symbols from special alphabet we call variables. This means that a variable can never equal a constant, statements like $x = 2$ are in strict sense nonsense. But hold off on that journey for a moment. Now having all these alphabets we can form strings using the various

letters. We could define arithmetic using digits $0,\ldots,9$, $+,-,\times,\div$ and some variables but lets simplify things to true/false which is long enough to explore the idea. We need to specify a grammar of how allowed formulas can be made. We basically separate the options by | (reads as "or") and use patterns to explain structures that are built up recursively. So for Boolean (true/false) algebra we have true $\top$, false $\bot$, and $\wedge$, or $\vee$, and not $\neg$ language might be defined like this.

```
<var>  ::= x | x_<int>
<Bool> ::= ⊤
         | ⊥
         | <var>
         | ¬ <Bool>
         | <Bool> ∨ <Bool>
         | <Bool> ∧ <Bool>.
```

Before leaving a word on *sorts*. Refine this as you like. For instance, one sort $a, b, c, \ldots, m, n, \ldots, x, y, z, x_1, x_2, \ldots$ for numbers, a sort $+, -, \times, [\ldots]_\ell, \ldots$ for operators, $\cong, \equiv, \ldots$ for equality. You decide, it is a made up language.

Formulas are strings over these alphabets. Each of these has a a grammar and a handy notation is to separate options by | which reads as *or*. For example, a true $\top$, false $\bot$, and $\wedge$, or $\vee$, and not $\neg$ language might be defined like this.

```
<var>  ::= x | x_<int>
<Bool> ::= ⊤
         | ⊥
         | <var>
         | ¬ <Bool>
         | <Bool> ∨ <Bool>
         | <Bool> ∧ <Bool>.
```

So x is a boolean, as is $\top$ and $\neg x \wedge x_3$. Math languages assume also the symbols for parenthesis.

of variables for numbers will be $m, n, \ldots, x, y, \ldots$ whereas a separate sort of variable is used for operations, such as $+, -, \times, \ldots$, and still another $A, B, C, \ldots$ for sets and so forth. Sorts used in this way have formal meaning in logic and I mention that because you will come across it in examples of formal methods—the growing field blending the idea of proving theorems with proving programs, what we will need to make safe self-driving cars and video games that you can't cheat.

Starting with a string M with symbols of various sorts, the task is to substitution variable x in by another string N. If there is only one variable around you may think of this as $M(N)$. Since M may have many variables let us be specific:

$$M[x := N]$$

To see how to do this lets break the down th process of making M. For example we might be in context of a simple calculator. So our constants are the digits $0, 1, \ldots, 9$, and we have also $+, -, \times, \div$. Variables we call x, y, z and if we need more x_n where is a sequence of digits will do.

We can start out small with two sorts of data. One sort are atomic symbols, digits, an alphabet or some other meaning of a constant. The second sort are variables. Then we formula M is either an atom a, a variable x, or the concatenation of two formulas K and L. There is a popular notation for this is to separate each case by the stroke | which reads as "or".

```
F[X] = 0 | 1 | ... | 9 | + | -
```

```
        | x
        | <F[X]> cat <F[X]>
```

So if our atoms are ♣,♡,♠,♢ and

```
F[X] = a:A
     | x:X
     | K:F[X] cat L:F[X]


M[x:=N] =
    match M with
            a:A ⇒ a
            y:X ⇒ if x=y then N else y
        K cat L ⇒ K[x:=N] cat L[x:=N]
         y ↦ L ⇒ if x=y then y ↦ L
                                    else if x free
                                       in L
```

```
F[X] = a:A
     | x:X
     | M:F[X] cat N:F[X]
     | x:X ↦ M:F[X]

M[x:=N] =
    match M with
            a:A ⇒ a
            y:X ⇒ if x=y then N else y
        K cat L ⇒ K[x:=N] cat L[x:=N]
         y ↦ L ⇒ if x=y then y ↦ L
                                  else if x free in L
```

Free algebra 3

How to operate 4

One day + means to add natural numbers, the next day polynomials, later matrices. You can even add colors “Yellow=Blue+Green”. When you program you learn to add strings

```
In [1]  print "Algebra " + " is " + " computation"

Out [1] Algebra is computation
```

The + is in fact a variable stand in for what we call a *binary operator* or *bi-valent operator*. It takes in a pair, according to the grammar $\square + \square$. The valence and the grammar of an operator comprise its *signature*.

Unlike these notes, addition should only be used when it is grammatically correct e.g. $2+3$ rather than $+23$. Think of this like any other language where there could be a dialect that evolves the operator's grammar and lexicon. A program to add two lists could get away with the following linguistic drift:

```
In [2]  cat [3,1,4] [1,5,9]
        [3,1,4] + [1,5,9]

Out [2] [3,1,4,1,5,9]
        [4,6,13]
```

Using `cat` avoided confusion with the later + concept and was the better choice. Challenge yourself to see both as addition and you will find addition everywhere.

Since we are evolving we my as well permit multiplication as a binary operator symbol, changing the signature to $\square \cdot \square$, i.e. $2 \cdot 4$; or $\square\square$, e.g. xy. Avoid $\square \times \square$, we need that symbol elsewhere. Addition is held to high standards in algebra (that it will evolve into linear algebra). So when you are considering a binary operation with few if any good properties, use a multiplication inspired notation instead.

Valence 1, *unary*, operators include the negative sign $-\square$ to create -2. Programming languages add several others such as

`++i, --i` which are said to *increment* or *decrement* the counter i (change it by ± 1). Programs also exploit a ternary (valence 3) operator:

```
if (...)then (...)else (...)
```

You may find that with shorter syntax like `n!=0 ? m/n : error`. If your interested look into *variadic* operators to see how far this idea goes.

Operators that generate

Incrementing is more a definition than it is a case of adding 1. A natural number is either 0, or a successor $S(k)$ to another natural number k. Often this is expressed as a definition where | stands for separating cases, for example:

$$\mathbb{N} := 0 \mid S(k)$$

So 0 is "zero", and 1 is just a symbol representing $S(k)$, $2 = S(S(0))$ and so on. Replace S's with tally marks (no relation to or at this point) we recover childhood counting:

$$0 := \_\_, 1 := |, 2 := ||, 3 := |||, 4 := ||||, 5 := \text{𝍸}, \ldots$$

The point is, the successors are not so much a function moving around the numbers we have, it actually is a producer of numbers.

Perhaps because it is so primitive, this is an idea we can imitate to create more meaningful values, like a string of characters in an alphabet `['a','b',...,'z']`.

```
data String = Empty | Prepend( head, tail)
```

Some readers might relate to a different dialect of programming such as the following

```
class String
    case Empty extends List
    case Prepend( head, tail) extends List
```

The head here caries around what we put in the list and the tail is what comes next in the list. Observe the similarities:

$$2 := S(S(0)) \qquad (\mathbb{N})$$

$$\texttt{"me"} := \texttt{Prepend('m',Prepend('e',Empty))} \qquad (\text{String})$$

The left-hand sides are merely notation for what the data really is on the right. Both the successor and the `Prepend` are operators that generate new values. So part of algebra is to generate new data; so, it is no wonder that it closely connections to computation.

Generated operators

We can take this the idea of generating further, for example, using the unary operator, successor, prepend, etc., and have them generate binary operators.

$$m + n := \begin{cases} n & m = 0 \\ S(n+k) & m = S(k) \end{cases}$$

So does $2 + 4 = 6$? We can test this out.

$$\begin{aligned} || + |||| &= | \left(| + |||| \right) \\ &= | \left(| \left(\_\_ + |||| \right) \right) \\ &= | \left(| \; |||| \right) \\ &= | \; \text{卌}. \end{aligned}$$

Try this for strings

$$s + t := \begin{cases} s & t = \texttt{Empty} \\ \texttt{Prepend}(x, s + tail) & t = \texttt{Prepend}(x, tail) \end{cases}$$

What is `"awe"+"some"`?

While no one will seriously add integers as a tally, knowing that it can be done establishes a pattern which can be exported to other context with meaningful new structure. Just notice $3+4 = 4+3$ but not so with adding strings. These siblings have their own

personalities, and it this might even help us recognize that while $3+4$ does equal $4+3$ it might be for somewhat subtle reasons.

What about multiplication? Isn't is just this:

$$m \cdot n := \overbrace{n + \cdots + n}^{m}.$$

That is nice, but seems to leave us to figure out missing parenthesis or set aside time to prove they don't matter. I am in the mood to continue making things rather than study them. Lets repeat what we have done.

$$m \cdot n := \begin{cases} 0 & m = 0 \\ k \cdot n + n & m = S(k). \end{cases}$$

It works, but check it out for yourself. And for strings what might we get?

$$s \cdot t := \begin{cases} 0 & s = \texttt{Empty} \\ \texttt{Prepend}(x, k \cdot n) + n & s = \texttt{Prepend}(x, tail) \end{cases}$$

What is "Mua"·"Ha"? You can carry on to make exponents and more. If you do you may find Knuth arrow notation helpful on your journey, look it up.

Operators measuring defects

Algebraist spend a lot of time worried about misbehaving operators causing them to generate new operators that spot the flaws. If we can add, subtract, and multiply then we can make the following operators as well.

$$[a, b] = ab - ba \qquad \text{(Commutator)}$$
$$(a, b, c) = a(bc) - (ab)c \qquad \text{(Associator)}$$

Commutative algebra requires $[a, b] = 0$ while associative algebra needs $(a, b, c) = 0$. For example matrices fail to be commutative algebra but are associative. Replace the role of multiplication of matrices with $[a, b]$ and ask for it's associate, i.e.

$$(a, b, c)_{[,]} = [a, [b, c]] - [[a, b], c]$$

and we no longer get associative nor commutative algebra. These are structures known as Lie algebras. While not associative, because they are based originally on matrix products that are associative we can stumble eventually upon a graceful alternative

$$0 = [a, a] \qquad \text{(Alternating)}$$
$$0 = [a, [b, c]] + [b, [c, a]] + [c, [a, b]]. \qquad \text{(Jacobi)}$$

So the problem is not getting worse, at least we wont be needing to look into some valence 4 operators as defects. Sabinin algebra studies how defects in operators pile up or die off.

Keep in mind requiring that $[a, b] = 0$ or $(a, b, c) = 0$ is an equation. Like any equation it has limited solutions. By that reasoning, most of algebra wont be behave nicely.

Types of algebra 5

You can add music, well at least you can play two songs at the same time. Is that again music? Probably it is safer to say we can add sound, and some sound is music. This is thinking like an algebraist. This is clarifying that addition has a context. Operations are carried out without leaving that context. We say that:

> "Sound is closed under addition."

You needn't be too strict about the context. When you add a list of length 3 to a list of length 3 it gets to a list of length 6, not 3. Just like music is to sound, a list of length 3 is to a list of any length. We recover closure by adding lists of all sizes.

Broadening our context may not be enough. When we divide we avoid division by 0. When we subtract natural numbers $m - n$ we need $m \geq n$. When we compose functions we need the domain of the second to contain the image of the first. You might be in a place to fix this, for example add $\pm\infty$ to define as $x/0$ or negative numbers to account for $3 - 5$. Functions f and g that cannot be compose can be composed as relations:

$$(g \circledcirc f)(x) := \{z \mid \exists y, y = f(x), z = g(y)\}.$$

When doing algebra with such operators we nearly always spend our time explore independent cases, most of which are roughly speaking "errors". The better idea is to acknowledge we don't really want to divide by 0, produce negatives we are not using, or composing non-composable functions. It is time for types.

What to do isAny system A paradox is not a contradiction

Evidently $I(2) = 2$ and $K_3(2) = 3$. These definitions are so simplicistic they work without a domain or co The function on the left is an identity function, changing nothing to the given inputs. The functions on the left are constant functions, ignore the input and outputing a fixed value. Neither concept is in need of a domain or domain, which is convenient when explaining functions before there are sets, such as when we program, or when sets aren't big enough, such as trying to make a set of all sets.

Using what we know about substitution we can choose the constant c however we want, maybe 2, maybe ♣, or another variable y. So why not substitute x for c?

$$K_c(x)|_{c:=x} = x = I(x).$$

A constant function is suddenly the identity function, which cannot be right. This is a mistake in substitution known as the *paradox of the trapped variable* and what it tells us that substitution is not so naive after all.

We call the first an *identity* function and the second is a *constant* function. This notation is misleading, technically I is a symbol which denotes some unknown process such that when applied to data ♣, its output, often denoted $I(♣)$, is given input data unchanged. So $I(♣) = ♣$ is a judgement we can make about an identity function not a program.

There is a bit of implicit information in our use of parenthesis and what they mean. Traditionally I and K are the functions. An input ♣ passed to a function I yields an output denoted $I(♣)$. In the case of the identity function I does nothing to modify the input so we can judge $I(♣) = ♣$. Since this applies for any input we can abstract over the inputs by replacing their role with a variable x and write $I(x) = x$ It is not a definition of I so much as a consequence. Of course we could use the outcome rule to conjure a perspective algorithm to perform the function. When we do this we often insert some extra language like "define I(x)=x" or use the Walrus notation $I(x) := x$. In programs words like define are abbreviated. For example, the following two programs satisfy the identity function rule without following the same process.

```
def doNothing(x) = x
def doNothingUseful(x) = compute 200! then return x
```

Much of the feasibility of algebra comes down to appreciating different processes that achieve the same overall calculations.

Suppose we have two types of data, one we call A and the other B.

Operators forced into service

Sometimes you have an operator and it doesn't work. to be clever, even lucky, to guess at an operator. In famous history, Heisenberg escaped to an island from Copenhagen to avoid hay-fever, or an over zealous host Niels Bohr. Able to think clearly he summarized what he knew: a particle ψ could take possibly many states, maybe spin up $\uparrow$ or spin down $\downarrow$. so it record this as linear combination it as a linear combination $|\psi\rangle = \alpha|\uparrow\rangle + \beta|\downarrow\rangle$ where in vector space with $\{\uparrow,\downarrow\}$ as a basis.

The Copenhagen interpretation suggested that the particles was in a mixture of states that when observed would spontaneously collapse to one of $\uparrow,\downarrow$. Others think everything happens just in other universes but I don't see why the only normal universe would be randomly the one I am in so I think that notion is equaly philosophical and meaningless. Perhaps quantum Bayssian interpretations make sense. There is some unknown distribution of the particle, , The α and β explained the probability that when we measure the state of ψ

somewhat famously was having difficulty with hay-fever in Copenhagen and was stuck with quantum puzzles who could not solve. Clearing his head on an island he conjured the view that particles ψ could be viewed as linear combinations of all the states the could o $\langle\psi| = \alpha\langle$ consisting of the probabilities of each state. recognized matrices capture all the states of a quantum system and to compute the next event involv

Further operators

These examples lean on some addition pre-existing somewhere.

Other operators

To enforce a philosophy such as that some algebra is closed to operators Such a philosophy is one

To get the details rolling in earnest we must know the application. The world around us partly interacts with computers so

lets assume this is a uniform assumption: any problem in algebra that I want to state should be expressible to a computer. That computer might not be able to handle it but its hard to imagine an equation we need to consider where the equation itself cannot be stored in a computer.

That is one philosophy, a philosophy where addition is an abstraction. That word makes some bristle. It reminds them of polarizing art installations or refrains from the bar room rants between pure and applied thinkers. Abstract, for for those who reason, means to limit argument to specified attributes. Thats every type of math and a good chunk of science so it shouldn't raise dismay. *Raising the level of abstraction* is then just the declaration the we're about to gather the instance we have and ma

Perhaps you feel some one owes you a foundation for addition, a place you derive what addition really means before you take to making it show up everywhere else. You can count on it, literally.

$$\mathbb{N} = 0|S\ k.$$

How do we do algebra? 6

This leaves us with the job of making those data which can be substituted into equations. This means firstly new numbers that, like the decimals, complex numbers, polynomials or matrices; can give a meaning to things like 0 and 1 and eventually the answers x we seek in solving applications. Secondly we need to find what works for the operations, the sums, squares, products, and etcetera. Last and most important in the method of algebra is to invent the possible substitutes for equality, what algebraist call *congruences*.

Over 2 centuries the methods have evolved and with it notation and emphasis to the point where the three roles just articulated may not be recognizable. This is where it becomes necessary to add in constraints: a family of problems you need solved that guide you to the algebra you need. But when we pause to see the similarities we can carry forward a far great number of algebraic lessons than we can by concentrating only on the best tools for individual examples. That will be our perspective. To further constrain the study we invite in applications that constrain the questions to important families the ones which might solve your problem or lead you to the algebra the may one day define you.

Formulas & Equations 7

Lets get precise about equations. They have a left hand side and right hand side. But the items that can occur on the left can also occur on the right so we could define any one side and be done. The equations worth study are those involving variables, more precisely variable and operations since and equation $x \cdot y = 1$ is infinitely more interesting than $x = 1$. So we start out with what are called formulas, but you would do well to think of these as generalized polynomials, only we might involve more operations than simply those found in common polynomials. As a byproduct we will re-invent induction in a form that is used everywhere in in the design of software data types.

Index