An Operators Manual for Algebra

How to use algebra

2023-08-02

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 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 divide $ax^2 + bx + c = 0$ into solutions that are real or complex based on $b^2 - 4ac \geq 0$. This is an infinite family of equations captured by 2 strategies, one when we accept complex numbers.

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

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

These equations concern entirely unrelated contexts, for example 0 on the right is a whole number, whereas 0 on the left is the function $0(x, y) = 0.0$ Yet, the similarities as equations shine through. This is because 0, $+$, and powers of 2 are 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, which happens everywhere in 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, we wont find a solution as no square decimal is -1.

The power of variable everything is that we are not stuck with the real numbers. Lets 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.

Why stop here? Try quaternions, or (2×2)-matrices. These each have addition, 0 and squares. Now we learn there can be infinitely many solutions to that equation. If complex solutions were not right perhaps one of these infinitely many quaternions or matrices will do. This is the method of algebra: find all the values that can be substituted into an equation to see what makes solutions possible and how they behave. 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

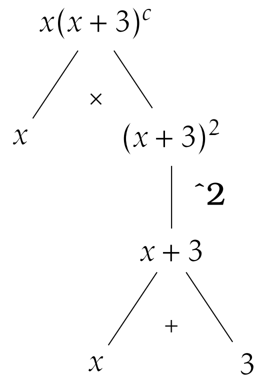
$$x^{541}+x+1\equiv 0 \pmod 2.$$

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

Every symbol in an equation is variable, meaning it can be substituted. Substitution seems well-understood: see the variable you want to replace, replace it; leave the other symbols alone. So you will make quick work of a couple examples. How do you substitute for $x := 7$ in $x(x+3)^2$. (Using $:=$ indicates an assignment, not an equality variable.) Whether obvious or not, we start by parsing the grammar. We can make this visual with a *parse tree*.



Starting at the top, apply individualized rules for each step. Substitute in a product $MN[x := 7]$ you compute $M[x := 7]N[x := 7]$, sometimes denoted by a "leads to" relation $\rightsquigarrow$. So, with $M+N$ we can write,

$$(M+N)[x := 7] \rightsquigarrow M[x := 7] + N[x := 7]$$

Finally $3[x := 7] \rightsquigarrow 3$ and $x[x := 7] \rightsquigarrow 7$. If we use $x := 3$ or $x := t$ we use the same procedure. Substitution is proceeding by induction, an induction tailored to algebra. The base cases are how we assign constants and variables and the inductive steps are defined for each operation.

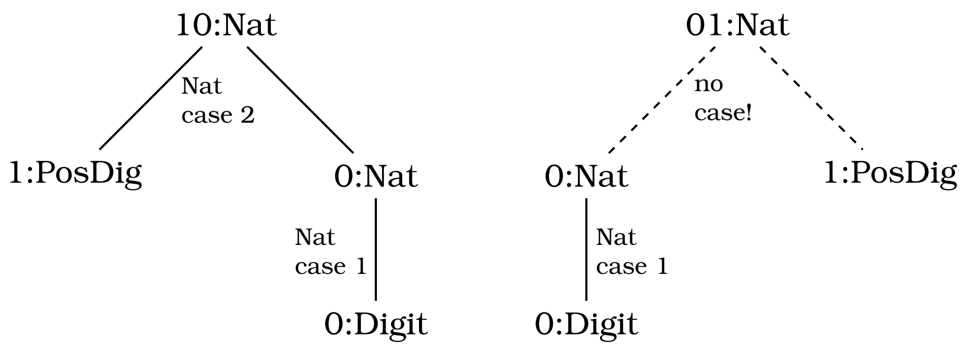
Grammar. This worked because we turned the formula into a tree. It is the same thing you do when you diagram a sentence in grammar school, only with English you can sometimes get cycles. That we got a tree is owed to the fact that the grammars for mathematics are basic and gentle, what Chompsky *Syntactic Structure* calls *context-free* grammars.[1] How we worked through the tree is known as *traversing* the tree, it is a type of induction.

To write down a context-free grammar we specify the atoms of our language. For natural numbers we can used digits `0`,`1`,...,`9`. Then we write down patterns of how to use them separated cases by the auxiliary symbol | (which reads as "or"). Each rule is given an name called a *token* (or *tag*) and denoted `<Name>`. Since := is our assignment of variables, we use ::= as assignment of token rules.

Listing 2.0.1 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.



The strings accepted by are a grammar are known as the *language* for that grammar.

[1]If an algebraist starts a talk with 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.

Operators. The rules in the grammar are often described as *operators*. For example, `<PosDig><Nat>` operates to take a positive digit and a natural number and form a new natural number. We call it a *binary* operator 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 operator *homogeneous*.) Some operators take in only one input, so called *unary*, such as the first case of `<Nat>` which takes in a digit and is said to *promote* the term to a natural number.[2] Operators that require no preexisting data, such as the digits $0,\dots,9$, are called *nullary*, or simply *atomic*.

Remark 2.1. The symbols `::=`, `<Token>` and `|` are Backus-Naur Form (BNF) notation, which is popular in computer science. The idea under different notation was written down in 600 BC by Panini's rules of Sanskrit grammar. Good ideas get rediscovered; great ideas get stolen.

Multi-sorted grammars. To add depth to this we add new sorts of symbols. The first sort are constants. The next sort are variables, e.g. $x, y, x_0, x_1, \dots$. A third sort can be operators, e.g. $+, -, \times, \div$. To make things short lets drop down to the language of Boolean (true/false) algebra instead.

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. For example, $\top \vee (\neg(x_2 \wedge y) \vee x_1) : Bool$ but $Bool$ rejects $\bot\neg$.

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.

[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.

Listing 2.0.2 A Boolean algebra grammar.

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

$$\begin{tikzcd}
\neg(x_2 \wedge y) \vee x_1 : Bool \arrow[d, dash, "\neg"] \\
(x_2 \wedge y) \vee x_1 : Bool
\end{tikzcd}
\qquad
\begin{tikzcd}
& \neg(x_2 \wedge y) \vee x_1 : Bool \arrow[dl, dash] \arrow[dr, dash] \arrow[d, phantom, "\vee"] & \\
\neg(x_2 \wedge y) : Bool & & x_1 : Bool
\end{tikzcd}$$

This is where *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 in $\neg(x_2 \wedge y) \vee x_1$ the LeMOM $\neg$ is the symbol we need to parse, which means it will "parse" once we succeed in forming a $Bool$. So we move ahead 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 $x_2 : Bool$, then $\wedge$ matches with $x_2 : Bool \wedge y : Bool$ so that we have $x_2 \wedge y : Bool$. Then we hit ')' which matches our earlier '(' thus giving us $(x_2 \vee y) : Bool$ which finally matches with $\neg$`<Bool>` and we parse $\neg(x_2 \wedge y) : Bool$. 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). But parsing a grammar with such orders of operations is slower, often requiring us to read in everything before making even a single choice. Take for example $(3x+9)^5$, it is the outermost exponent 5 that comes at the end which we will consider as the first step in parsing, i.e. the top of the tree.

> **Grammar plus order of operations are a language to express a desired induction.**

Exercises

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

How to operate 3

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:

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

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.

$$\begin{aligned} || + |||| &= |\left(| + ||||\right) \\ &= |\left(|\left(\underline{\quad} + ||||\right)\right) \\ &= |\left(|\ ||||\right) \\ &= |\ \cancel{||||}. \end{aligned}$$

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 4

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? 5

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 6

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.