Algebra Outside

How algebra is used

2023-08-18

James B. Wilson

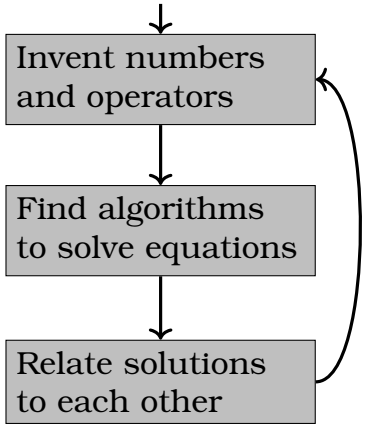
Department of Mathematics
Colorado State University

James.Wilson@ColoState.Edu

James.Wilson@ColoState.Edu

Algebra today is taught chronologically by example. First groups, then rings, then fields, then modules, then bigger groups, then crazier rings, then fusing groups and rings with outside influences like topology, analysis, and geometry. Today there are few general-practice algebraist. There are instead geometric group theorist, algebraic geometers, commutative ring theorist, representation theorist, non-associative algebraist, computational algebraist (that's me), and even those topics are too general for any one theorist to master.

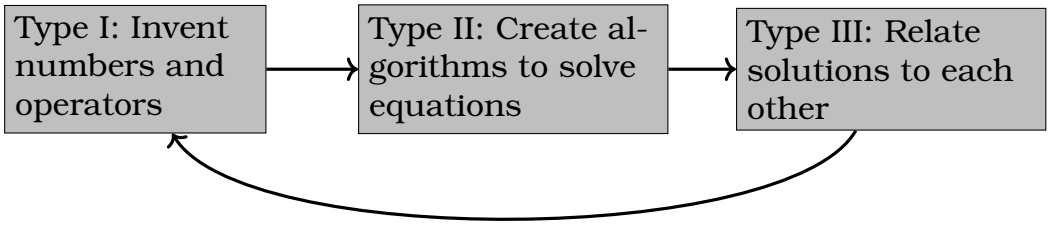
It bucks tradition (set out by Noether) to change this approach to algebra. Chances are high that the books that brought you here already laid out the subject in this order. But I speculate that there is a place in the sciences for a view of algebra as a whole. What is it that algebra does? Like many who work in math a likely response has a does of calculated propaganda to impress your parents, funding agency, or quite the passenger next to you. This text offers some clues about making that response a bit more honest in intention. I divide all of algebra into a 3 phase flow chart.



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. Basic though it may seem that solution illustrates the general arc of an algebra investigation.



Introducing numbers.

Early history (early childhood) is type I algebra. Count pebbles or beads and give the patterns names

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

"0" might have been chosen as the shape left by removing the last pebble from a sand table leaving behind zero pebbles. It struck Giuseppe Peano in the early 1900's that tallies would be easier to get right. With the following two rules Peano introduced the natural numbers to the formalism of math.

> N_0 *vale "numero", et es nomen commune de 0,1,2, etc.*
> $0 \rightarrow$ *"zero"*
> $+ \rightarrow$ *"plus". Si* a *es numero,* $a+$ *indica "numero sequente* a*".*

(See G. Peano *Formulaire de mathematiques. I-V*, p.27.) Replacing $+$ with $|$ (and equating $\text{𝍸} := ||||\ |$), Peano's model of numbers is simply the grammar of tallies.

The notation has evolved. These numbers are now almost always designated as *natural numbers* and denoted $\mathbb{N}$. Instead of $a+$ we now often write $S(k)$ of the "successor" to the natural number k. Other authors use $a{+}{+}$ or ${+}{+}a$ to mimic trends in programs. We start at 0 not because 0 is the first thing we count, but because counting is S! So from 0 our first count is $1 := S(0)$. So it transpires that the natural numbers do include 0, and at the same time we begin counting at 1. It seems in this case math and computation learn from each other, with neither the victor of their childish debate.

We mentioned this was merely a grammar. Today we write grammars with list of rules, called *production rules*. Some rules are to specify what makes up the alphabet of symbols in our grammar. For us 0 and S are the complete alphabet. So we may write either 0 on its own, or we may pre-pend the symbol S to any existing natural number. Each rule is given a name called a *token* (or *tag*) and denoted `<Name>`. Since the Walrus $:=$ is our assignment of variables (more on this later), we use the "astonished Walrus" $::=$ as assignment of token rules. Letting `<Nat>` stand for any string that we consider a natural number in Peano's language then we arrive at the following primitive recursive rule.

```
<Nat> ::= 0 | S <Nat>
```

Here the `|` serves to separate cases, it is often read aloud as "or". The word "primitive" here means that the recursion can only depend backwards in history. That is you must follow S by an already existing natural number, so it is not a circular description. So we can claim that this grammar accepts 0 as a natural number, denoted `0:Nat` (i.e. $0 \in \mathbb{N}$) but also `SS0:Nat` (i.e. $2 \in \mathbb{N}$).

Believe it or not we have just created our first algebra. It has two operators. The first 0 depends on nothing, it is a constant or *nullary* operator. The second, successor $S\square$, is *unary* operator in that it depends on a single input k.

We do well to acknowledge that computers understand this much. Listing 1.0.1 shows two programs you could run today that implement Peano's idea. There are of course many differences. Visibly, the left-hand side favors mathematically minded

Listing 1.0.1 Peano's natural numbers programmed in two different languages.

```
data Nat = Z
         | S k

zero = Z
two = S (S zero)
```

```
class Nat
    case Zero() extends Nat
    case Next(k:Nat) extends Nat
sealed  // no more cases
zero = new Zero()
two = new Next(new Next(zero))
```

symbolic notation and economizes even on parentheses in the spirit of "$\sin x$" notation. Meanwhile the right-hand side favors a verbose imitation of natural language and prefers the $\sin(x)$ notation. Set the differences aside.

Even without a deep understanding of these programs, one can make out the contours of Paeno's definitions. Both use a mix of keywords (in blue) to tell our system to prepare a new type (or class) of data that will be called `Nat`. Then they instruct the system to accept exactly two ways to make such data. It may be some initial state, `Z`, respectively `Zero`, that depends on nothing; otherwise, we must give data `k` of type `Nat`, denoted `k:Nat`, which will then produce new data `S k`, respectively `Next(k)`.

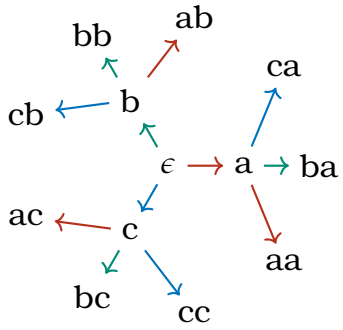
Introducing strings

The point of basic patterns is to use them in more complex systems with an understanding of the resulting behavior. Imagine a string of characters in an alphabet a, b, and c. The grammar evolved to have more constants

```
<abc> ::=
        | a <abc>
        | b <abc>
        | c <abc>
```

So this grammar accepts *aabaabcac* but would reject *adabb* since

'd' is not in the list of productions rules. This grammar also accepts an empty string, often denoted ϵ. We can draw the accepted words as a graph with each vertex being an already accepted word and an arrow indicated which production rule advanced it to another accepted rule. This builds out an infinite 3-regular tree, of which we show just a snippet.



This is another algebra, with one nullary operator ϵ and three unary operators a□, b□, and c□ being the production rules, that is the tree colors of arrows.

`Char:=['a','b',...,'z']`.

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

Writing `head:Char` or `tail:String` indicates that head must come from the alphabet we chose and tail must be some already produced string, possibly empty. Some readers might relate to a different dialect of programming such as the following

```
class String
    case Empty extends String
    case Prepend( head:Char, tail:String) extends String
sealed
```

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.

Exercise

1. Mimic the String data type to make a list of integers (that is switch from the alphabet to integers).

2. Mimic the String data type to make a list of fixed by unknown data of type A, call it `List[A]`.[1]

Historically `Empty` for lists is called `Nil` and `Prepend` is called `Cons`.

Eliminating numbers.

Now that we have created numbers we shall want to use them. An obvious use of numbers in counting is to make the process modular and parallel. For example we can have two separate counts later combined by addition.

"add"					

+	0	1	2	3	4	5
0	0	1	2	3	4	5
1	1	2	3	4	5	6
2	2	3	4	5	6	7
3	3	4	5	6	7	8
4	4	5	6	7	8	9
5	5	6	7	8	9	10

Beyond a table we need some formulas. If we have two groups m and n and each is either nothing or a successor to something else, then that leaves us with just four cases to consider.

+	0	$S(k)$
0	0	$S(k)$
$S(\ell)$	$S(\ell)$	$S(S(\ell+k))$

[1] Alternatives include `List a` and `List<A>`. Search for *generics* in your programming language to learn more.

As the first column does not change m we can simplify this down to 2 cases.

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

Addition is different from S because addition is not making new numbers, it is actually just refashioning numbers into others.

approach was more mathematical than digits and he wrote down

Whatever you understand about mathematics or programs one thing should stand out about the nature of 0, `Z`, `Zero`, +, `S`, and `Next`. These are not functions. That is because each of these is in the service of creating numbers, not merely moving them around like a function would. Zooming in on one of the programs makes this exceptionally noticeable.

A closer look

Perhaps the most convincing way to see that we are creating data is to run this process all the way down to what a machine must do and witness that somewhere the computer must somewhere as for storage space. We will do this in stages as the bottom is a mighty long fall.

A first step might be to convert the cases into a sequence of two functions which both make appropriate changes to some internal value. This might look like this:

```
class NaturalNumber {
    int value // Internal computer storage
    Zero() {value=0} // zero out the memory
    Next(k:NaturalNumber){value=k+1}
}
z = new Zero()  // make a 0
two = new Next(new Next(z)) // make a 2
```

Grouping this under a single data type 'NaturalNumber' is mostly just for us not to get confused. When we zoom in closer we find that the computer splits this apart, making the storage space separate from the functions that use it. Here is a lower level take and a realization that indeed both introductions (constructors) need to ask for some storage, they do not really know how much, the computer calculates that for them.

```
struct Nat { int length, uint8[] digits }
Nat NaturalNumber_Zero() {
    memory = malloc(Nat) // allocate memory
    memory.length = 1
```

```
    memory.value[0] = 0
    return memory
}
Nat NaturalNumber_Next(k:Nat) {
    memory = malloc(Nat) // allocate memory
    if k.digits[k.length-1] != 0 then
        memory.length = length +1

    memory.digits = k.digits +1
    return memory
}
```

If we zoom in even further we see this 'malloc' command eventually translate into asking for actual memory in the computer. Here is a simplified take that assumes the length is fixed at 9 bytes, that is the maximum value reached is 2^{128}, it would take all the computers in the world to tally from 0 to 2^{128} in your lifetime so this is a safe limit.

```
%macro NaturalNumber_Zero 0
    mov param1, 0x0009      ; move 4 into first parameter
    call malloc            ; storage of size param1 (=9)
    mov store, 0x0000      ; mov 0 into the storage
    ret

%macro NaturalNumber_Next 1
    int  temp
    mov  temp, param1     ; make a copy of first input
    mov param1, 0x0009
    call malloc          ; storage of size 9
    mov  store, temp     ; move input to storage
    inc  store           ; increment storage
    ret
```

As this snippet reveals this program comes to a physical limit once we exceed the internal storage capabilities of integers in the system. We can fix this but the spirit of the programs above is to separate the need to understand the computer deeply and see the overall behavior.

If we hold on only to these ideas then we are narrowing what we need to know about natural numbers to just some fixed rules, so far the rules on how to create them. So let us say that the symbol $\mathbb{N}$ represents all the rules that apply to natural numbers and that writing $0 : \mathbb{N}$ or $n : \mathbb{N}$ is a way of saying that the symbols 0 and n obey those rules. Using $\vdash$ to mean "yields", In other words

Zero $\vdash 0 : \mathbb{N}$, nothing is need to yield 0 as a natural number.

Successor $k : \mathbb{N} \vdash S(k) : \mathbb{N}$, every natural number k yields another $S(k)$.

We can get back to childhood counting by giving names to these Peano numerals

$$0 := 0,\ 1 := S(0),\ 2 := S(S(0)),\ 3 := S(S(S(0))),\ldots$$

The point is, the successors are not so much a function moving around the numbers that we find laying around, it *makes* numbers. If it were not it would not continue forever and what would '…' and 'etc.' mean in your mind?

Computational significance.

Just as we switched Peano's $a+$ to $S(k)$; it would be no serious issue to have used $a++$, $++a$, or `next k`. Programmers today use a number of alternatives to encode natural numbers. Some languages hold close to mathematical notation preferring symbols to have meaning, like the following formula:

```
data Nat = Z:Nat | S (k:Nat)
```

Whatever "data" does we can assume it make the computer see what comes next as describing a new type of data, 'Nat'. And the symbols Z and Sk are distributed into cases separated by '—'.

Other languages face business and industry and package data verbosely to help communicate the meaning, for example:

```
class NaturalNumber
    case Zero extends NaturalNumber
    case Next(k:NaturalNumber) extends NaturalNumber
sealed  // no more cases
```

Words like “data”, “class”, “case” and “sealed” are called keywords and usually show up in different colors just to make sure we notice they have special roles to play in the language. A successful programming language (PL) will choose keywords that have some reasonable “obvious meaning”. This means that you can learn to read code in most languages by scanning for patterns and using keywords as clues to tie the meaning together. Take a moment to see that you see Peano’s main idea in both of the above code fragments.

Definition 1.1. In logic rules that explain what produces a symbol are called *introductions*, e.g. $\vdash 0 : \mathbb{N}$ and $k : \mathbb{N} \vdash S(k) : \mathbb{N}$ are introduction rules.

In programming the word *constructor* is often preferred as it explains how to construct data of a specified type.

Reaching new uses.

The point of basic patterns is to use them in more complex systems with an understanding of the resulting behavior. Imagine a string of characters in an alphabet `Char:=['a','b',...,'z']`.

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

Writing `head:Char` or `tail:String` indicates that head must come from the alphabet we chose and tail must be some already produced string, possibly empty. Some readers might relate to a different dialect of programming such as the following

```
class String
    case Empty extends String
    case Prepend( head:Char, tail:String) extends String
sealed
```

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.

Exercise

1. Mimic the String data type to make a list of integers (that is switch from the alphabet to integers).

2. Mimic the String data type to make a list of fixed by unknown data of type A, call it `List[A]`.[2]

Historically `Empty` for lists is called `Nil` and `Prepend` is called `Cons`.

, find distances of length $\sqrt{2}$, learn about imaginary numbers and π. The shift to π signals the shift to type II because π is such a fiddly number that it makes sense to leave it alone moving it round solely by rules of algebra until the end when we might replace with with 3.14 or a better approximation. For practical purposes π is the first quantity used as a variable.

High school algebra is nearly all type II, and mostly following al Khwarizmi's *al Jabr*—the method of balancing parts, which today is called Algebra. Move over Euclid, al Khwarizmi's method made people money, settled land disputes, calculated grain totals and arguably did more to make math a necessity for average people than any other source. His name has passed down to modern life as the word *algorithm*. At this stage students are told that polynomials of degree d have d complex roots. Perhaps because some non-algebraist call this fact "the fundamental theorem of algebra"

[2] Alternatives include `List a` and `List<A>`. Search for *generics* in your programming language to learn more.

(it's not) this seems a good place to leave algebra for calculus or leave math for good. Most people never reach a type 3 problem.

You are one of the few who have heard of type 3 algebra. You have seen the solving linear equations gives you back often infinitely many answers and you don't have time nor the interest to write down infinitely many answers. You know that this can be done with a basis. The next example is even more greedy. Why write down d roots to a d degree polynomial?! Instead, you want to write down one solution and sprinkle in some pattern to explain how you might find all the others. Think back to $x = \frac{-b \pm \sqrt{b^2-4ac}}{2a}$ solving $ax^2 + bx + c = 0$ (with $a \neq 0$). That '$\pm$' trick saved you thinking of two roots. Ever solved $x^6 - 1 = 0$ by using $x = e^{(2\pi i/6)t}$ thinking of rotating around by 60 degrees? These relationships turned out to have a name (groups) and were the first type 3 algebra.

This is where the heavy weights got involved: Lagrange made a formula for quartic polynomials from a primitive concept of groups but he got distracted by a fluke concerning symmetric polynomials and missed out on the eventual full solution by Galois. Gauss showed that construction by ruler and compass produces only numbers that are iterated applications of square-roots by observing what in today's language would say that the group of relations is nilpotent. Abel found a specific obstacle with $x^5 - x + 1 = 0$ by showing that if its relations were to be solvable they would create contradictions with facts from calculus, the first proof that not all polynomials are solvable by radicals. Galois characterizes all such polynomials by describing what their type 3 algebra would look like. Both men died tragically and without their work being understood until after their young deaths.

Here I should settle a confusion in the subject. Abel and Galois did not show that the quintic has no solutions! That would fly in the face of Gauss' "fundamental theorem of algebra" (a 5th degree polynomial has 5 complex roots). What Abel and Galois proved is that roots of these polynomials where not of the form hinted at by the solutions for 1,2,3 and 4th degree polynomials. Take for example al Khwarizmi's quadratic formula gives roots of the form $\frac{-b}{2a} \pm \frac{1}{2a}\sqrt{b^2 - 4ac}$, where a, b, c are numbers we already created. Cardano's cubic formula goes a step further. Given numbers $a, \ldots, g$

already created, then solutions to the cubic can be made in the following form.

$$a + b\sqrt{c} + d\sqrt[3]{e + f\sqrt{g}}.$$

Lagrange's quartic formula gives solutions now involving rational linear combinations of fourth roots. When a number is a linear combination of n-th roots of iteratively created numbers then we say that number is *radical*. That the quintic is not solvable by radicals just means that to factor quintics and higher students need to go back to a type 1 problem: invent new constructions of numbers. In fact, history had already done this for degrees 2,3. For example, if $b^2 - 4ac < 0$ then the solutions may be written as

$$x = \sqrt{\frac{c}{a}}\cos\theta \pm \sqrt{\frac{c}{a}}\sin\theta \qquad\qquad \theta = \cos^{-1}\frac{-b}{2\sqrt{ac}}$$

The cubic equations can likewise be solved with trigonometry. So Abel and Galois point the way to revisiting how we build numbers and finding new methods. That is, the outcome of type 3 algebra was a need to build new type 1 algebra.

This feedback look has been hard at work for more than a century. These new type 1 problems lead to rings and modules. That lead to algorithms and representation theory to solve type 2 problems with those numbers. Those solutions generated new relations, more groups (type 3), but also new type 1 algebras (Clifford, Lie, Hecke, and Hopf to name a few). The iteration continues.

As a final remark, we should settle one last potential confusion. The fundamental theorem of algebra already told us the shape of roots, i.e. complex numbers. Why not stop here? Answer this yourself: what are the roots of $x^2 = 2$? If you thought $x = \pm\sqrt{2}$ and not $x \approx \pm 1.41$ then you appreciate the difference. By prescribing $x = \pm\sqrt{2}$ we both solve nothing (its just notation) and everything (this solution truly works). This gets a a philosophical position that numbers are made up to mean what they can do. Real numbers are made up to mean that finite patterns that stay close together (Cauchy sequences) can in fact be considered as numbers (they converge to themselves, which means nothing but lets there be real numbers).

Radical numbers like $\sqrt{2}$ are another form of invention, a number that when squared is 2. It is freeing to think this way because we can make such numbers precisely with ease. For example $\begin{bmatrix} 0 & 2 \\ 1 & 0 \end{bmatrix}$ squares to $\begin{bmatrix} 2 & 0 \\ 0 & 2 \end{bmatrix}$ which gives precise calculations. But for some situations precision wont be important and we can use the approximation $1.41^2 \approx 2$. Algebra therefore sees the matrix and the decimal examples of $\sqrt{2}$ as interchangeable. To algebra, $\sqrt{2}$ is a property, not so much a number. Because of this we can imagine many more creative algorithms to work with these numbers.

For the purposes of solving it is important to have both but exact and approximate solutions. To expose this difference look no further than finding the eigenvalues of a matrix.

$$\begin{bmatrix} -9 & 7 & -12 & 6 \\ -3 & 5 & -4 & 6 \\ 7 & -5 & 8 & -2 \\ 2 & -2 & 4 & -4 \end{bmatrix}$$

With 64 bit floating points in Julia, my computer printed out

```
julia> eigvals([-9 7 -12 6; -3 5 -4 6; 7 -5 8 -2; 2
   -2 4 -4])
4-element Vector{ComplexF64}:
 -0.0005295477729425071 - 0.00052951901343231 46im
 -0.0005295477729425071 + 0.00052951901343231 46im
  0.0005295477729438736 - 0.00052957650470254 66im
  0.0005295477729438736 + 0.00052957650470254 66im
```

But actually this matrix has a single eigenvector 0 and is not diagonalizable but instead it is conjugate to a nilpotent matrix:

$$\begin{bmatrix} -9 & 7 & -12 & 6 \\ -3 & 5 & -4 & 6 \\ 7 & -5 & 8 & -2 \\ 2 & -2 & 4 & -4 \end{bmatrix} = \begin{bmatrix} 1 & 2 & 1 & 2 \\ -1 & 2 & -1 & 2 \\ 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 1 \end{bmatrix}^{-1} \begin{bmatrix} 0 & 4 & 0 & 0 \\ 0 & 0 & 4 & 0 \\ 0 & 0 & 0 & 4 \\ 0 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} 1 & 2 & 1 & 2 \\ -1 & 2 & -1 & 2 \\ 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 1 \end{bmatrix}$$

Solutions to equations are not stable under approximations.

1.1 Everything is variable

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.

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

1.2 Inventing numbers

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.

This is why so much of algebra today is concerned with making new numbers and searching out which equalities (congruences) can be used on new numbers. It is the front line of algebra, the first question that must be concurred. And it part of why so much

research on algebra doesn't even feature solving equations because the goal is to have numbers ready for whatever equations show up.

> **The first method of algebra is to invent numbers: their constants, their operations, their rules. We call these "algebras".**

1.3 Computation

Algebra is the computational wing of mathematics.

Look to calculus. We obviously work with limits and derivatives to recover Rolle's theorem and the significance of $f'(x) = 0$. But when we want to find such x we inevitably are left to solve equations, like $x^2 - x - 3 = 0$, which we do by algebra.

Switch to topology. Roll up a stretchy sheet of plastic along the following Van Kampen patterns viewed in Figure 1.1 to make a torus $\mathbb{T}^2$ and a Klein bottle $\mathbb{K}$. These are different, right? Why? Of course we use topology to prove the interesting bit, like the Seifert-van Kampen theorem: the loops $\pi_1(M^2)$ on a surfaces M^2 can each be described as a sequence of blue loops b and red loops r. Loops that contract to nothing can be read along the boundary. Hence,

$$\pi_1(\mathbb{T}^2) = \text{ words over } \{b, r\} \text{ rewritten by } br = rb;$$
$$\pi_1(\mathbb{K}^2) = \text{ words over } \{b, r\} \text{ rewritten by } r = brb.$$

We use algebra to compute that those numbers are not the same so neither are the two spaces.

Other fields? How many times in applied math, number theory, or combinatorics do we end up solving something even if just a linear equation, in order to get a final answer? You don't have to be convinced but the evidence is there.

Yet most of us do not want to be computers. So what does an algebraist do when the questions come down to computing? We put our energy into crafting algorithms to solve these problems. In fact the word "algorithm" comes from al Khwarizmi whose influential book *al Jabr* (the method of parts) gave use the first detailed

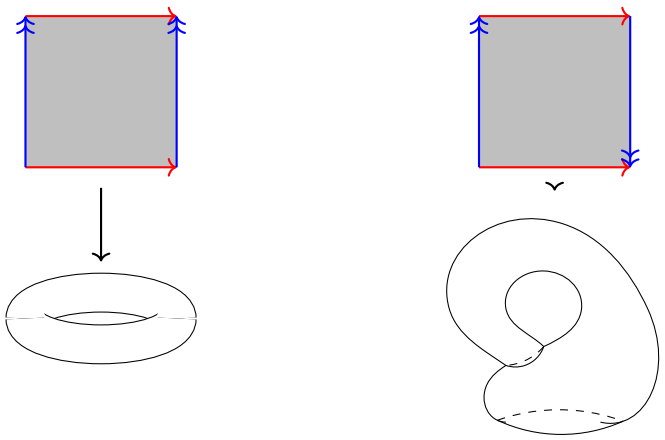


Figure 1.1: A torus and Klein bottle, with thanks to https://tex.stackexchange.com/q/77606/86.

explanations of algebra. Half of that book is a list of story problems solved by various algorithms in algebra. If you are concerned that this now trends to computer science, well know that computer science cares more about the data structures than about any algorithms. Just like counting get faster with an abacus, algebraic algorithms get better with data structures. So these two worlds interact but care deeply about different parts of the problem.

How to operate 2

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 (the number of inputs) and the grammar of an operator comprise its *signature*.

Unlike these notes, addition should only be used when it is grammatically correct e.g. *infix* $2+3$ rather than *prefix* $+,2,3$ or *postfix* $2,3,+$. Think of this like any other language where there could be a dialect that evolves the operator's grammar and lexicon. Famously HP calculators were postfix for some time to match engineering requirements. It turns out humans will adapt to technology easier than technology adapting to humans. 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` reminded us to concatenate and avoided confusion with the later + concept. It was the better choice. Challenge yourself to see both as addition and you will find addition everywhere.

Since we are evolving, we may 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. These days composition $\square\circ\square$ is written as multiplication; so, you can use that symbol however you like. 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. The transpose of a matrix is a unary operator. Programming languages add several others such as `++i`, `--i` which are said to *increment* or *decrement* the counter i (change it by ± 1).

We can combine binary and unary operators to make useful (and useless) ternary operators. One of my favorite (but generally useless) ternary products multiplies (2×3)-matrices, that is right, rectangles not squares. The product goes like this:

$$[A,B,C] = AB^\dagger C$$

where $B^\dagger$ is the transpose. A more serious product comes up in symmetric matrices where we need what is called the *Jordan Triple product*

$$\{A,B,C\} = \frac{1}{6}(ABC + CBA)$$

This is part of an whole family of Jordan products including

$$A \bullet B = \frac{1}{2}(AB + BA)$$
$$\langle A_1,\ldots,A_\ell\rangle = \frac{1}{\ell!}(A_1\cdots A_\ell + A_\ell\cdots A_1).$$

Notice in all these case if $A_i = A_i^\dagger$ then $\langle A_1,\ldots,A_\ell\rangle = \langle A_1,\ldots,A_\ell\rangle^\dagger$. So these are products that explain how symmetric matrices behave, and if you know about quantum mechanics that means these are the products you can use on observable quantum events.[1]

Stranger ternary products showup in places where we wish we had easier binary products to explain things. For example in geometry at small dimensions (dimension 2) we do not always have coordinates to explain what makes some set of points into a line. So Marshall Hall decided why not make a line ℓ be described by a ternary operator

$$-\otimes-\oplus-$$

[1] Pascual Jordan was a physicist inventing math for quantum mechanics, not be be confused with Camile Jordan of Jordan-Holder and Jordan Normal form fame.

This is just notation but put in a formula like $y = m \otimes x \oplus b$ and you start to imagine the result as an algebraic line. With this ternary product he was able to associated every projective plane to coordinates in some ternary ring. Once you have such a ternary ring you can go to work to see if it might actually decompose into two binary operations of multiplicaton and addition, e.g. by locating a "one" and a "zero" where $x = 1 \otimes x \oplus 0$. Then you reverse the process and define $m \cdot x := m \otimes x \oplus 0$ and $x + b := 1 \otimes x \oplus b$ to get a more familiar ring-like structure.

Programs also exploit a ternary (valence 3) operator:

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

The words, while helpful, are unimportant. Some programming languages replace them with symbols emphasizing their "operatorness"

```
_?_:_
```

Here for example is division with remainder of positive integers

```
div(m,n)=(m>=n)?(div(m-n,n)+(1,0)):(0,m)
```

Programs go further making operators with huge valence. If you find that interesting look into *variadic* operators to see how far this idea goes.

A number of subtle problems are mounting. For example, we probably want a Boolean (true/false) to go in the the left-most spot of an if-then-else-, and we cannot compose just any two functions and get expected results. We scale a vector on one side by not the other. Grammar must be more than just $\square \cdot \square$.

Lets consider an operator

Another operator takes a pair A and B of sets or more general types of data and creates ordered pairs or disjoint unions.

$$\frac{A,B:Type}{A\times B:Type} \quad (\mathsf{F}_\times)$$

$$\frac{a:A \quad b:B}{(a,b):A\times B} \quad (\mathsf{I}_\times)$$

$$\frac{x:A\times B}{\pi_A(x):A} \qquad \frac{x:A\times B}{\pi_B(x):B} \quad (\mathsf{E}_\times)$$

$$\frac{a:A \quad b:B}{\pi_A(a,b) := a} \qquad \frac{a:A \quad b:B}{\pi_B(a,b) := b} \quad (\mathsf{C}_\times)$$

In code you may encounter this as `Pair[A,B]` or as `(A,B)`.

```
Either A B = Left (a:A) | Right (b:B)

apply (f:A->C, g:B->C, x:Either[A,B]) : C =
  match x with
    Left a => f(a)
    Right b => g(b)
```

A special case of the disjoint union operator is to extend a type by one term:

$$A^? := A \sqcup \{*\}.$$

It is said that $x : A^?$ is "maybe an A", or "optionally A". In code you will find these written out this way.

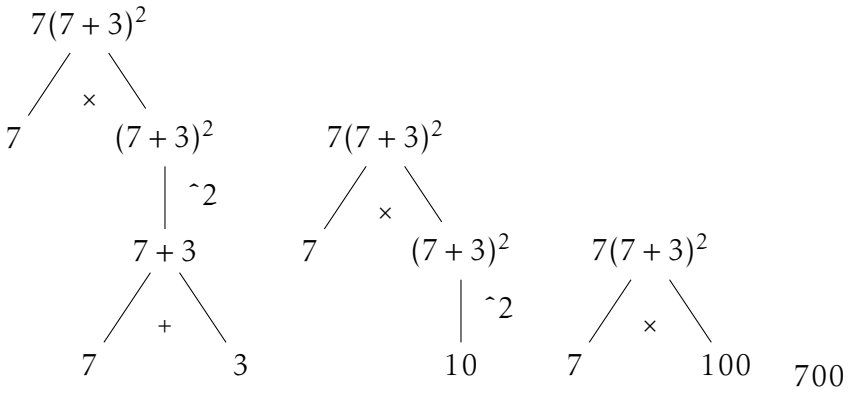
```
Maybe A = Just (a:A) | None

class Option[A]
    case Some(a:A) extends Option[A]
    case None extends Options[All]
sealed  // You cannot add to these cases
```

2.1 Grammar school

> "... 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.

Complex inductions can be specified by grammar.

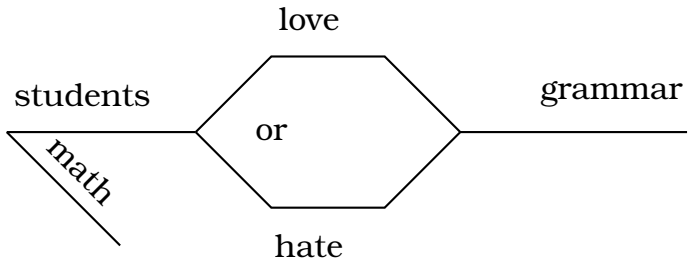
2.2 Grammar

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 two

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.[2] Don't be too disheartened. Virtually every programming language has a context-free grammar and programs can communicate a lot of hefty ideas.

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,

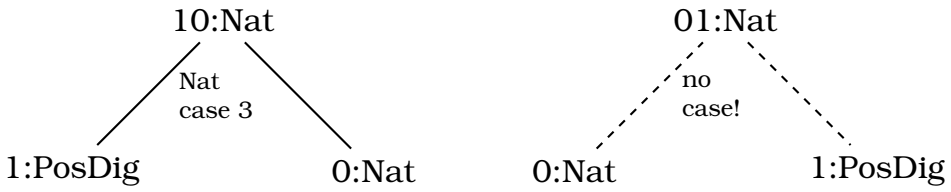
[2] If an algebraist starts a talk with a story that "...It was thought all natural languages were context-free until some obscure dialect in the alps or Africa was found...", then tune out until they return to equations. Linguist never had such illusions. Even english is not context-free, read James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

we use the "astonished Walrus" ::= as assignment of token rules. Listing 2.2.1 shows the grammar for natural numbers as digits.

Listing 2.2.1 The grammar for natural numbers.

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

The natural number grammar is said to *accept* `0` as a natural number, written "0:Nat". It also accepts 541:Nat and 10:Nat but it rejects "01:Nat" as that case does not exist as a production rule for any token in this grammar.



Induction is to apply production rules, so it 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.

Deciding to accept or reject is recursion, and so it may start with some string using the alphabet but with the wrong grammar and reach a point of failure, or it might get stuck trying to decide. Context-free grammars fortunately never get stuck, they do not even need a full strength computer to run (a push-down automata will do, like the chip you might need to run a microwave). Yet Moore's law makes actual computers abundant so lets skip the microwave in favor of a general purpose program to parse.

Listing 2.2.2 Parsing

```
posDigitize( token:Char):Option[PosDig] =
    match token with
    '1' => Some 1:PosDig;   '2' => Some 2:PosDig;
    '3' => Some 3:PosDig;   '4' => Some 4:PosDig;
    '5' => Some 5:PosDig;   '6' => Some 6:PosDig;
    '7' => Some 7:PosDig;   '8' => Some 8:PosDig;
    '9' => Some 9:PosDig;
    _  => None

digitize( stream:String ):Option[Digit] =
    match stream with
        nil => None
        head::tail =>
            match head with
                '0' => match tail with
                            nil => Some <0,0>:Nat
                             _  => None
                 _ => match posDigitize(head) with
                        Some (x:PosDig) =>
                            match digitize(tail) with
                                Some (y:Nat) =>
                                   <(10^(y.digits)*x+y),1+y.digtis>:N
                                None => None
                        None => Some (<x,1>:Nat)
```

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?

All these ideas appear under different notation in 600 BC by Panini's rules of Sanskrit grammar. Good ideas get rediscovered; great ideas get reappropriated.

Context-free grammars parse into trees leading complex inductions that are still follow a unique path.

Index