Mathematical Structure for Programming

Conor McBride

July 18, 2024

Chapter 1

spuds and a lemon

Let's make some numbers! You'll need a bag of lemons, a sack of potatoes, and a load of sticks (e.g. cocktail sticks) which are pointy at both ends. (If you have suitable electrodes and wires, you can make batteries as well as number.)

There are two ways to get a number in your hand:

1. grab a lemon—it's a number all by itself;
2. if you already have a number in your hand, stab the thing in your hand with a stick (don't stab your hand), then press a potato onto the other end of the stick and hold onto that potato instead

So that means we can make (with your hand on the left)

lemon
spud—lemon
spud—spud—lemon
spud—spud—spud—lemon
⋮

(Incidentally, a crucial "let's pretend" in this game is that you can tell potatoes apart from lemons, but that blind to any difference between two potatoes or between two lemons.)

Now, if you only make numbers this way, starting only with a lemon and holding onto only the most recently joined potato, your numbers will all have an important property. **If you follow the sticks down the number, you will eventually get to the lemon.** No sneaky moves, like making a circular model from potatoes and sticks, or always adding on the next potato just in time.

If you're thinking "Those aren't numbers: that's just sculpture with groceries!", then I have two things to tell you.

1. Our silly models with potatoes and lemons held together by sticks are as valid a representation of numbers as 5 or MMXXIV.

2. Count the spuds!

But how do you know you *can* count the spuds? How do you know that you won't just keep counting for ever? It's because you eventually get to the lemon.

1.1 addition as substitution

Adding these numbers is easy. If you're holding one of them in each hand, work your way along the left number until you reach the left lemon, then detach it, and stab the right number with the exposed stick, then work your way back out to the leftmost spud. In other words, *substitute* the right *number* for the left *lemon*

```
spud-spud-spud-spud-spud-lemon
                         +
                         spud-spud-spud-spud-spud-spud-lemon
                         =
spud-spud-spud-spud-spud-spud-spud-spud-spud-spud-spud-lemon
```

Note that the effectiveness of this procedure relies on the fact that you are guaranteed to find the left lemon.

It's also worth asking these three things:

1. What if the number in your left hand is a lemon? (You replace the lemon in your left hand with the number in your right, and the answer is exactly the right number.)

2. What if the number in your right hand is a lemon? (You replace the left lemon by the lemon in your right hand and the answer is indistinguishable from the left number you started with.)

3. What if you add three things? Does it matter whether you add left to middle first, or middle to right? (It doesn't matter. Both ways round, you end up with all the spuds you started with, in the same order (not that you can tell), and the rightmost lemon on the end.)

1.2 synchronized spuds make a race to the lemon

Suppose you are holding two numbers (left and right, say), each by their most recent potato. What will happen if you work your way towards the two lemons in tandem, one potato per step in each? Remember that for both numbers, you are sure that stepping in this way means you will eventually get to the lemon? It's a race to the lemon! There are three possibilities:

- you reach the left lemon while the right number still has a potato. That means the original left number was strictly smaller than the original right number, and now that the left number is a lemon, the current right number is the *difference*. As it were, you have found out that the original numbers were x and $x +$ spud-y for some x and y;

- you reach the left lemon and the right lemon at the same time — the dead heat means the numbers were both some x;

- you reach the right lemon while the left number still has a potato. So the original left number was greater and the current left number is by how much. The original numbers were $x +$ spud-y and x, for some x and y.

You should notice that, in all three cases, the number which won the lemon race, called x above, is the largest number from which you can reach *both* numbers by *adding*. You either add lemon and spud$y$, lemon and lemon, or spud$y$ and lemon, respectively. You have figured out how close you can get to both numbers by adding, together with what the differences are. In other words, we're *undoing* adding by as little as possible. Here, at least one of the differences is nothing, and when the inputs are equal they both are.

It's funny: this lemon race tells you 'maximum' and 'minimum', 'less than, equal, or greater than', and 'difference' all at once. The usual mathematical or programming presentation makes all of these tests and operations distinct, but they are all incomplete facets of the same underlying computation. (The same is true of computations in hardware using binary numbers.)

1.3 Euclid's lemon racing game

Here's a game that the Greek mathematician Euclid used to play with potatoes and lemons.[1] It's a game which computes one number from two.

You start with two numbers, one in each hand. If one hand holds a lemon, stop: the answer is the number in your other hand! Otherwise, run a lemon race. Now, at least one hand is holding a lemon: put that hand back to the start of its number; keep the other hand where it is and detach any potatoes you passed by to get there. That's to say, replace the larger number by the difference. Go back to the start.

Why does this game reach an end? Because you don't run the lemon race unless you start with potatoes in both hands, so each step will result i one or other hand moving strictly nearer its lemon, and both sides promise that you eventually get to the lemon.

But, perhaps more importantly, what on earth is the meaning of the output of this game?

(Some people have been known to tell a version of this story in which Euclid repeatedly breaks the largest square possible off from a rectangular piece of chocolate[2] until there is no chocolate left. I find the spuds and lemons version more savoury, because nobody wants a rectangle of chocolate with width zero.)

[1]Consider a pinch of historical salt.
[2]Likewise.

1.4 puzzles

Puzzle 1.1 *An 'add-respects-or' test is a computable way to decide a true-or-false property of spuds-and-lemon numbers such that*

- *the test gives false for a lemon;*
- *whenever you can split up any number x as $y + z$ in any way, the test gives true for x if and only if the test gives true either for y or for z or for both.*

A 'boring' test is one which always gives the same answer, no matter what number you test.

1. *Find one boring add-respects-or test.*
2. *Either find a different boring add-respects-or test, or explain why no such test exists.*
3. *Find an add-respects-or test which is* not *boring.*
4. *Either find a different non-boring add-respects-or test, or explain why no such test exists.*

(To show that two tests are different, you must be able to explain why there is some number where they disagree.)

Puzzle 1.2 *Let us say that a number is 'munky' if it is exactly divisible by three, 'minky' if division by three leaves a remainder of one, and 'manky' if division by three leaves a remainder of two. Explain how to test these properties of spuds-and-lemon numbers. You should be able to remove one spud at a time and test what is left. After all, you will eventually get to the lemon! Hint: it is easier to solve all three parts of this puzzle together, rather than taking the one at a time.*

Puzzle 1.3 *Our method of addition replaced the left lemon by the right number. You could also consider computations which replace each spud⊣in a number by the same something (which had better have a stick on the end). What can you compute in this way?*

Puzzle 1.4 *A 'multiply-respects-or' test is a computable way to decide a true-or-false property of spuds-and-lemon numbers such that*

- *the test gives false for spud⊣lemon;*
- *whenever you can split up any number x as $y \times z$ in any way, the test gives true for x if and only if the test gives true either for y or for z or for both.*

1. *Are any of your 'minky', 'manky', 'munky' tests multiply-respects-or? If so, which and why?*

2. *In a black box, I have a multiply-respects-or test which is not boring. What answer does it give for the lemon? No, you may not look in the box!*
3. *Is divisibility by* four *a multiply-respects-or test? Why?*
4. *What are the non-boring multiply-respects-or tests? (Note: it's terribly unfair for me to ask this question at this stage, but we'll make sense of it eventually.)*

Chapter 2

seeing the trees

Computer programs are written in formal languages. You can't just say any old thing and expect a computer to make sense of it, even if these days they're quite willing to bullshit you that they understand. Computer programs tend to be written textually, as a sequence of *lines*, each of which is a sequence of *characters* (i.e., letters, digits, spaces, punctuation, emoji, etc), and that's not particularly helpful if you're trying to understand them—it's merely a convenient fit with our ancient methods for writing stuff down. Text doesn't have much obvious structure, but computer programs do. We need to learn to see, and to talk about how to see.

We have a language (the language of *grammars*) for talking about languages. Let me start with an example

$$\begin{array}{rcl}\langle \mathit{Number}\rangle & ::= & \text{lemon} \\ & \mid & \text{spud-}\langle \mathit{Number}\rangle\end{array}$$

How to read this? Where to begin? Which of these things is nearest to hand? You can't be expected to know until someone establishes the conventions.

The place to start is with ::=, which you can pronounce 'is defined to be'. It's a variation on the theme of =, but with a particular spin. You're not being asked to agree with this equation, and there's no way you can check it. You're not supposed to think (or pretend) 'Oh I knew that!'. When I write this equation, I'm telling you something *new*, and asking you to put up with it for the time being.

The thing to the left of ::= is the name of a new *sort of thing*—by convention, the names of sorts of things are written in $\langle\cdot\rangle$ to distinguish them from *actual* things. The $\langle\cdot\rangle$ does the job of 'a' or 'an' in English—the indefinite article—we're talking about a generic class of things by imagining one, but not a special one. What I'm saying is 'I'm inventing a new sort of thing, namely a *Number*, and I'm telling you of what a *Number* may consist.'. Everything after ::= is the explanation of what I allow these *Number*s to be.

The next thing to pay attention to is the $|$ which you can read as 'or'. I'm giving you a choice of ways to make *Number*s.

The first choice I offer is 'lemon'. See? In English prose I write it in quotation marks, because I mean the actual word 'lemon'. It's not a *variable* which could stand abstractly for a variety of things. It's concretely what it says.

The second choice I offer is the concrete symbol 'spud-' which again (because no $\langle\cdot\rangle$) means only what it says, followed by $\langle \mathit{Number} \rangle$. The latter indicates that any *Number* you have made already can go in that place.

I didn't offer any other choices. Implicit in this definition is that it is *exhaustive*. Spuds and lemons, that's your lot! I'm saying "I'm inventing a new sort of thing, namely a *Number*, which consists either of 'lemon' or of 'spud-' followed by another *Number*." and I'm also saying "You can't possibly have *finished* making a *Number* unless you eventually get to a 'lemon'.".

So what are these *Number*s?

```
                  lemon
             spud-lemon
        spud-spud-lemon
   spud-spud-spud-lemon
                      ⋮
```

But these examples and the 'and so on' indicated by '$\vdots$', don't tell us precisely what *Number*s are, in general, whereas the grammar does.

The grammar does another useful thing, as well. It tells us how to perceive a *Number* as a structure. We can draw a picture of how to see spud-spud-lemon as a *Number*:

$$\begin{tikzcd}
 & & \text{lemon} \\
 & \text{spud-} & \langle \mathit{Number} \rangle \arrow[u] \\
\text{spud-} & \langle \mathit{Number} \rangle \arrow[u] & \\
\langle \mathit{Number} \rangle \arrow[u] & &
\end{tikzcd}$$

I've drawn this picture, which is called a 'parse tree', with the bottom nearest to hand (they're sometimes drawn the other way up), so you should read it bottom-to-top, like a detective, even though time went top-to-bottom[1]. It records how the choices offered by the grammar can be employed to make sense of the text.

Old computer scientists *see* these trees when we *look* at text. We don't even notice that we're doing it, but we experience distress when it doesn't just happen automatically and we look for someone else to blame. You are not born with this mode of perception, and it may take some practice to acquire it.

[1]In the beginning was the lemon. . .

2.1 what's in a grammar (and what isn't)?

Let's have another grammar with a little more to it. It's a little language of formulae involving one variable X.

$$
\begin{array}{rcl}
\langle \textit{Formula} \rangle & ::= & \mathsf{X} \\
& | & \langle \textit{Number} \rangle \\
& | & \langle \textit{Formula} \rangle * \langle \textit{Formula} \rangle \\
& | & \langle \textit{Formula} \rangle + \langle \textit{Formula} \rangle \\
& | & \mathsf{SumBelow}\ \langle \textit{Formula} \rangle \\
& | & \mathsf{Shift}\ \langle \textit{Formula} \rangle \\
& | & (\langle \textit{Formula} \rangle)
\end{array}
$$

What do we notice? Firstly, we have two ways to make a formula with no subformulae—the variable, X and any number you like. We could insist that numbers be written in spud–lemon form, but let's allow ourselves the brevity of Arabic numerals. That means we're in a position to make an 'eventually reaching the lemon' promise about formulae: if you keep tearing formulae apart, you eventually reach X or a number (in which you eventually reach the lemon).

Secondly, we have choices with *two* subformulae. That means our parse trees can spread out. E.g., $(\mathsf{X} + 2) * \mathsf{X}$ has parse tree

$$\begin{tikzcd}
 & & & 2 & & & \\
 & \mathsf{X} & & \langle \textit{Number} \rangle \arrow[u] & & & \\
 & \langle \textit{Formula} \rangle \arrow[u] & + & \langle \textit{Formula} \rangle \arrow[u] & & & \\
(& \langle \textit{Formula} \rangle \arrow[u] & & &) & & \mathsf{X} \\
\langle \textit{Formula} \rangle \arrow[u] & & & & & * & \langle \textit{Formula} \rangle \arrow[u]
\end{tikzcd}$$

If we're exploring the parse tree, we have a choice of paths we can take. Our 'eventually lemon' promise has to get stronger: you eventually reach X or a number, *whatever path you choose.*

Thirdly, there are weird things in the grammar that I haven't explained yet—$\mathsf{SumBelow}\ \cdot$ and $\mathsf{Shift}\ \cdot$. You don't need to know what they *mean* to be able to check whether a formula has a parse tree. You can see that $\mathsf{SumBelow}\ (\mathsf{X} * \mathsf{X})$ is fine, but $\mathsf{Shift}\ +$ is not (because $+$ is not a formula). And you might *think* you know what $+$ and $*$ mean, because they look like standard symbols for addition and multiplication, but I haven't confirmed that that's what *I* mean. Grammars don't say what things mean—they filter and structure texts which are *plausible*, discarding texts which are obvious *gibberish.* That's to say they *ask* helpful questions about the meanings of some things in particular, but they don't offer any *answers.*

Fourthly, the grammar is *ambiguous.* It is not immediately obvious whether

$\mathsf{Shift}\ \mathsf{X} * \mathsf{X}$ says

$$\begin{tikzcd}
 & \mathsf{X} & & \mathsf{X} \\
 & \langle \mathit{Formula} \rangle \arrow[u] & * & \langle \mathit{Formula} \rangle \arrow[u] \\
\mathsf{Shift} & \langle \mathit{Formula} \rangle \arrow[u] & & \\
\langle \mathit{Formula} \rangle \arrow[u] & & &
\end{tikzcd}$$

or

$$\begin{tikzcd}
 & \mathsf{X} & & \\
\mathsf{Shift} & \langle \mathit{Formula} \rangle \arrow[u] & & \mathsf{X} \\
\langle \mathit{Formula} \rangle \arrow[u] & & * & \langle \mathit{Formula} \rangle \arrow[u] \\
\langle \mathit{Formula} \rangle \arrow[u] & & &
\end{tikzcd}$$

Worse, we do not automatically know whether $\mathsf{X} + \mathsf{X} + \mathsf{X}$ groups to the left or to the right, and we might hope not to care, but for now we must worry about it. The simplest strategy to deal with ambiguity is to reject ambiguous texts, given that our grammar lets us use parentheses to resolve them: at least, that's what we might *hope* the parentheses are for. At the moment, we have no reason to believe X, (X) and $((\mathsf{X}))$ will all mean the same thing, and there are computer languages where wrapping extra sets of parentheses round stuff really does change its meaning.

As a side remark, one awkward consequence of mathematics being taught at school to some fairly standardised curriculum is a misplaced entitlement to notation working in all sorts of contexts the way it did in that context. One symptom of this malaise is the presumption that parentheses $(\cdot)$, brackets $[\cdot]$ and braces $\{\cdot\}$ are interchangeable and serve only to indicate grouping. This is very seldom, if ever, the case in computer programming languages, or in the notations we use to write about them. You should never expect $[x]$ to mean the same as x.

For *this* particular language of formulae, I'm willing to promise you that parentheses $(\cdot)$ are used only for grouping, i.e. to choose between possible parse trees. The meaning of a formula with enclosing parentheses will always be the same as that formula without parentheses. We may thus omit parentheses from parse trees—by the time we have *one* parse tree, their work is done!

The notion of grammar we have here is often called 'context-free'. The rules for what constitutes a formula don't change, depending on where the formula is. There are more complex aspects to meaningfulness which context-free grammars can't capture. For example, if we wanted to write the grammar of grammars themselves, we could not enforce the property that every sort of thing mentioned to the right of ::= has a unique definition, i.e., occurring to the left of exactly one ::= somewhere in the same grammar. That's to say the *scope* of names is exactly the sort of context which context-free grammars can't handle. So, don't expect grammars to encode complicated requirements for coordination between separate regions of a text.

2.2 a note on recursion

Both of the grammars we've seen so far have been *recursive*. We've said how to make bigger numbers from smaller numbers, and how to make bigger formulae from smaller formulae. Recursion often makes beginners nervous—circularity is suspicious—and some teachers make the mistake of playing down that nervousness, in an 'Oh, you'll get used to it!' sort of way. I'm here to tell you that it's quite reasonable to be nervous about recursion, and to help you ask the extra questions you will need to reassure yourself about recursion you may encounter in the wild.

For a start, it helps to make a distinction between two ways in which recursion can make sense. Some recursions are trying to achieve some functionality which are *eventually finished*: these rely on each recursive sub-thing being simpler, with some guarantee that things can't keep getting simpler forever—you eventually stop because you eventually reach a 'base case' with no subproblems. Other recursions are trying to achieve some functionality which is *always ready*: it might stop, it might keep going, but it will always let you ask 'what's next?'. The latter is how you expect computer games to behave: you might win, you might lose, but you might just keep on going, and you expect the game to keep responding to you.

The bad recursions never finish but can become unresponsive, and that can happen. Those recursions are worth being afraid of. If you see something defined by recursion, don't run away, but don't assume it makes sense. See if you can find out why it's eventually finished or always ready.

Now, in our world of grammars, we expect our parse trees to have the property that following any path through them eventually stops. What can we say about Theresa May's famous grammar?

$$\langle \mathit{Brexit} \rangle \quad ::= \quad \langle \mathit{Brexit} \rangle$$

Is it a bad grammar? No, merely *pointless*! It is well constructed. There is one choice of how to make a Brexit: you make it from a Brexit. So there are no parse trees for this grammar with finite paths, and no text can possibly make sense as a Brexit.

2.3 abstract syntax trees

Chapter 3

squish a bunch of stuff

If you tap your foot five times, and then you tap your foot six times, you'll have tapped your foot eleven times. There's some kind of relationship between adding up numbers and repetitive action sequences.

If you have one bag of five potatoes and another of six, combining both bags into one will give you a bag of eleven potatoes. Bags of spuds don't behave this way because they were taught arithmetic at school. Arithmetic behaves this way because it helps us think about bags of spuds.

If you multiply two by five, that's ten; and two times six is twelve. Adding ten and twelve gives twenty-two, which is twice eleven. There's some kind of relationship between adding up numbers and adding up their doubles. (You may have heard this called "The Distributive Law", as if there were only one.)

If you raise two to the power five, that's thirty-two; and two to the six is sixty-four. I don't expect you to multiply numbers as large as those in your head, but you might add five and six to get eleven, then happen to know that two to the eleven is two thousand and forty-eight. (Having a head full of powers of two is an occupational hazard of programming computers.) Why does that trick work?

Exactly one of five and six is an odd number, and so is eleven. That's odd! It may seem trivial, but at least one of five and six is positive (as opposed to zero), and eleven is positive too. (The only way the sum of two counts can be zero is if they were both separately zero.)

My point is that

$$5 + 6 \;=\; 11$$

is not a random isolated truth. It has a relationship with a whole bunch of other truths about other senses of "combining stuff", which all go

Combining the fivey thing with the sixy thing gives you the eleveny thing.

and of course, five, six and eleven are overly concrete examples—what matters is that they are related by a truth about adding numbers, a truth which somehow survives the translation to a truth about combining ny things, be they toe-taps, bags of spuds, or tests for oddness. Moreover, it wasn't only the things which mattered—it was also how they were combined. When we doubled we got a truth about adding, but when we two-to-the-powered we got a truth about multiplying, and when we tested for positivity we got a truth about "either or both".

3.1 pictures of combining

We can draw pictures of strategies for combining things like this

$$\underbrace{5\ 6}_{11}$$

Data flows from top to bottom. This component

$$\underbrace{\downarrow\ \downarrow}_{\downarrow}$$

has two inputs and one output. We can wire these components together to combine more things.

$$\underbrace{\begin{array}{c}\underbrace{\downarrow\ \downarrow}\\ |\end{array}\quad \downarrow}_{\downarrow}$$

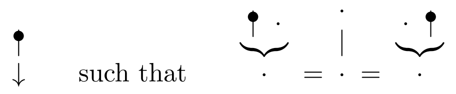
If we choose what sort of things we're combining (numbers, say) and how (adding, say), then by labelling the wires with data, we can make assertions about the results of combining things. I usually just write the data instead of the wire, so

$$\underbrace{\underbrace{5\ 6}_{11}\quad 7}_{18} \qquad \text{asserts} \qquad 5+6 = 11 \quad \text{and} \quad 11+7 = 18$$

but it's allowed to label the same wire more than once as long as you label it with the same thing. Signals don't change value in the middle of a wire. The same calculation is indicated by

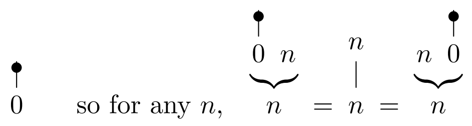
$$\begin{array}{c}\underbrace{\underbrace{5\ 6}_{11}\quad \begin{array}{c}7\\ |\\ 7\end{array}}_{18}\\ |\\ 18\end{array}$$

Now, some notions of combining things are equipped with "the sensible way to do nothing". That's given by a component with no inputs and one output

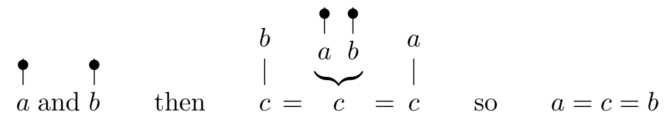


In other words, combining with nothing turns into wire: that the output must be the same as the input is what "doing nothing" means.

For adding numbers, 0 is the right way to do nothing,

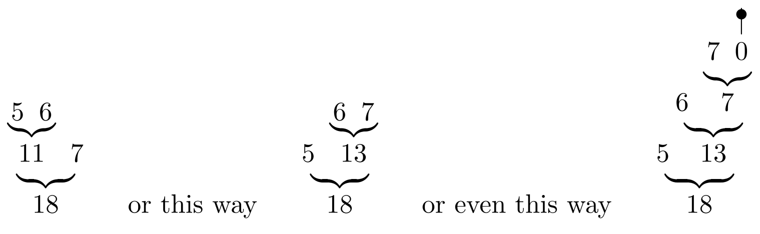


It's very useful to have a sensible way to do nothing. That way, we can look out of an aeroplane window and count the fish, or say how many spuds we have left once we've eaten them all. Our rules tell us that there's only one sensible way to do nothing for any given way of combining. Imagine we had

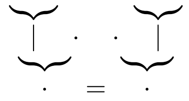


That is, on the left, we do the left nothing to the right nothing, and on the right, we do the right nothing to the left nothing: our rules tell us we must end up with the same nothing. So if you see a "way of combining", *look* for "the sensible way to do nothing"—you won't find two, you might find one, and if there's no sensible way to do nothing, that's interesting, too.

When we're combining a bunch of stuff, we are sometimes in the lucky position that it's the sequence of *the stuff* which determines the result of combining them, not the structure of pairwise combinations we choose. If we add up the list $[5, 6, 7]$, we get 18, whether we calculate it this way



In general, we want



so that we can regroup the calculation without reordering the inputs and be sure of getting the same output (even though the intermediate data change). That's why we don't bother writing brackets in $5+6+7$. You can combine data onto a "running total" one at a time, or you can give half the data to a friend and combine your outputs afterwards. Knowing what doesn't matter makes it far easier to gain confidence about what does!

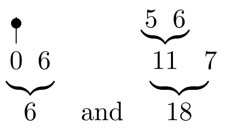
Of course, I haven't really demonstrated that only the sequence of the inputs determines the outputs, not the structure of the combination tree. Let me sketch one way to see it. Let me say that one of our diagrams is *listy* if it is always overbalanced as far to the right as possible, and has a nothing in the top right corner, i.e.

- inputs occur only on the left of $\underbrace{\cdot\ \ \cdot}_{\cdot}$
- only inputs occur on the left of $\underbrace{\cdot\ \ \cdot}_{\cdot}$

Think carefully about the difference in meaning made by the difference in word order. The first condition rules out

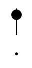
$$\begin{array}{c}6\\|\\6\end{array} \quad \text{and} \quad \underbrace{5\ \ 6}_{11}$$

in both cases because 6 is somewhere it is not permitted. The second condition rules out

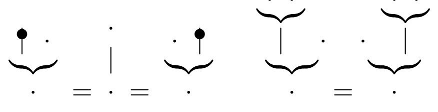


because some left things are not inputs.

Each of our sequences of inputs can be put into a listy diagram because

- if you have no inputs, only  is listy;
- if you have a first input x, you must make $\underbrace{x\ \ L}_{t}$ where L is the listy diagram made with all but the first input.

But that's not enough, because we need to know that *any* diagram can be transformed into its listy counterpart by using the three rules we gave ourselves



Here's how:

- ⫯ is already listy;
- $\overset{\overset{\cdot}{|}}{\cdot}$ can be make listy like this $\underbrace{\cdot\ ⫯}_{\cdot}$;
- if you have a diagram $\underbrace{D_0\ D_1}_{t}$, you can make it listy by first making D_0 into listy L_0 and then making listy D_1 into listy L_1, so you have $\underbrace{L_0\ L_1}_{t}$. Now *rotate* the whole thing into being listy like this:
  - if you have $\underbrace{⫯\ L_1}_{t}$, turn it into L_1 (whose 'total' must already be t);
  - if you have $\underbrace{\underbrace{x\ L_0'}_{|}\ L_1}_{t}$ rotate it to $\underbrace{x\ \underbrace{L_0'\ L_1}_{|}}_{t}$, then keep rotating on the right.

So we've used all and only the rules we asked for. The rotation process effectively pastes L_1 over the ⫯ in the top right corner of L_0.

There are lots of things to say about this "explanation", some good, some bad. Does it make sense to you? How would you check it?

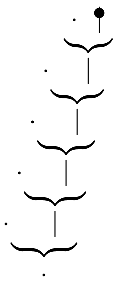
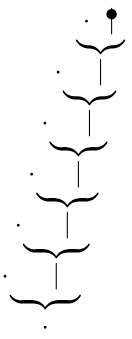
How do you know, for example, that "then keep rotating on the right" actually achieves something other than endless rotation? There's a sense that rotation of $\underbrace{L_0\ L_1}_{t}$ works by looking at L_0, and if $L_0 = \underbrace{x\ L_0'}_{|}$, then we keep rotating with $\underbrace{L_0'\ L_1}_{|}$, where the new diagram on the left, L_0' is smaller than the old diagram on the left L_0. As long as we can't keep cutting smaller diagrams out of bigger ones forever, we'll be fine—sooner or later, we'll hit ⫯. But is that explanation of the explanation just more waffle, or is it something we can codify?

Another potential grumble is that the whole thing depends on inventing this notion of which diagrams are "listy", and it looks like I just plucked that definition out of my arse. Where did it really come from?

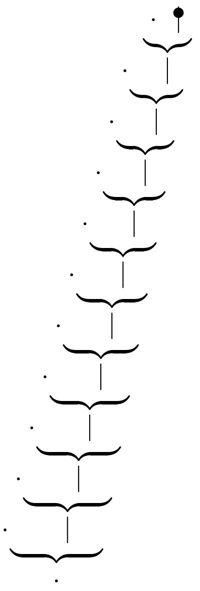
As for the positives, for one thing, no numbers got added in the course of this narrative. We said which three rules we needed, and we deduced an "only the sequence of inputs determines the output" result for *any* notion of combining things which obeys those rules. If we want the same result for multiplication (where 1 is "the right way of doing nothing"), we know what we need to check.

Moreover, our story about pictures of combining things involved inventing a way to combine *listy pictures* of combining *things*. To combine L_0 and L_1, paste L_1 over the ⫯ in the top right corner of L_0, or in other words, combine the sequences by concatenating them—joining them up end-of-one-to-start-of-the-next in space. Here ⫯ is the "right way of doing nothing", as it represents the empty sequence.

For example, here are the listy pictures for combining five things and six things, respectively:

and if you combine them, you get



which is, of course, the listy picture for combining eleven things, so perhaps numbers *did* get added, after all.

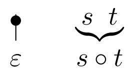
3.2 it's called a "monoid"

While I'm trying to reduce the number of words by drawing more pictures, we are going to need a name for these "ways of combining a bunch of stuff". When I do introduce terminology, I'll try to make it the standard terminology. The standard name is "monoid", which is Greek for "one-ish", as in "you can take any sequence of things and combine them into one thing". Let's review the definition.

A **monoid** is given by a type T, a value ε in T, and an operator $\circ$ which takes two inputs from T and yields one output in T, satisfying the laws

$$\varepsilon \circ t = t \qquad t \circ \varepsilon = t \qquad (r \circ s) \circ t = r \circ (s \circ t)$$

We've been drawing


ε
s t
$s \circ t$