evincarofautumn.blogspot.com

Why Concatenative Programming Matters

24-31 minutes

Introduction

There doesn't seem to be a good tutorial out there for concatenative programming, so I figured I'd write one, inspired by the classic "Why Functional Programming Matters" by John Hughes. With any luck it will get more people interested in the topic, and give me a URL to hand people when they ask what the heck I'm so excited about.

Foremost, there seems to be some disagreement over what the term "concatenative" actually means. This Stack Overflow answer by Norman Ramsey even goes so far as to say:

> "…it is not useful to use the word 'concatenative' to describe programming languages. This area appears to be a private playground for Manfred von Thun. There is no real definition of what constitutes a concatenative language, and there is no mature theory underlying the idea of a concatenative language."

This is rather harsh, and, well, wrong. Not entirely wrong, mind, just rather wrong. But it's not surprising that this kind of misinformation gets around, because concatenative programming isn't all that well known. (I aim to change that.)

Concatenative programming is so called because **it uses function *composition* instead of function *application***—a non-concatenative language is thus called *applicative*. That's the defining difference, and it's as "real" a definition as I need, because, well, it's the one that people are using. Bang. Descriptivism.

One of the problems with functional programming is that the oft-touted advantages—immutability, referential transparency, mathematical purity, &c.—don't *immediately* seem to apply in the real world. The reason "Why Functional Programming Matters" was necessary in the first place was that functional programming had been mischaracterised as a paradigm of negatives—no mutation, no side-effects—so everybody knew what you *couldn't* do, but few people grasped what you *could*.

There is a similar problem with concatenative programming. Just look at the Wikipedia introduction:

> A concatenative programming language is a point-free programming language in which all expressions denote functions and the juxtaposition of expressions denotes function composition.

> The combination of a compositional semantics with a syntax that mirrors such a semantics makes concatenative languages highly amenable to algebraic manipulation.

This is all true—and all irrelevant to your immediate problem of *why you should care*. So in the next sections, I will show you:

- How concatenative programming works
- How typing in a concatenative language works
- How concatenative languages are efficient
- What concatenative languages are *not* good for
- Where to go for more information

Now let's get started!

♦ ♦ ♦

The Basics

In an applicative language, functions are applied to values to get other values. λ-calculus, the basis of functional languages, formalises application as "β-reduction", which just says that if you have a function ($f\ x = x + 1$) then you can substitute a call to that function ($f\ y$) with its result ($y + 1$). In λ-calculus, simple juxtaposition ($f\ x$) denotes application, but function composition must be handled with an explicit composition function:

> compose := $\lambda f.\ \lambda g.\ \lambda x.\ f\ (g\ x)$

This definition says “the composition (compose) of two functions (*f* and *g*) is the result of applying one (*f*) to the result of the other (*g x*)”, which is pretty much a literal definition. Note that this function can only be used to compose functions of a single argument—more on this later.

In concatenative languages, composition is implicit: “*f g*” is the composition of *f* and *g*. However, this does not mean that function application becomes explicit—it actually becomes *unnecessary*. And as it turns out, this peculiar fact makes these languages a whole lot simpler to build, use, and reason about.

The untyped λ-calculus is a relatively simple system. However, it and the many systems derived from it still need three kinds of term—variables, lambdas, and applications—as well as a number of rules about how to correctly replace variable names when applying functions. You have to deal with name binding, closures, and scope. For a supposedly low-level language, it has quite a bit of inherent complexity.

Concatenative languages have a much simpler basis—**there are *only* functions and compositions**, and evaluation is just the *simplification* of functions. It is never necessary to deal with named state—there are no variables. In a sense, concatenative languages are “more functional” than traditional functional languages! And yet, as we will see, they are also easy to

implement efficiently.

♦ ♦ ♦

Composition

Let’s say we want to multiply a couple of numbers. In a typical concatenative language, this looks like:

```
2 3 ×
```

There are two weird things about this.

First, we use postfix notation (2 3 ×) rather than the prefix notation (× 2 3) that is common in the functional world, or the infix notation (2 × 3) that most languages provide for convenience. There is nothing *stopping* a concatenative language from having infix operators, but for the sake of consistency, most stick to postfix notation: “*f g*” means (*g* ∘ *f*), i.e., the *reverse* of function composition. This is actually rather nice notation, because it means that **data flows in the order the functions are written in.**

Second, we said all terms denote functions—so what the heck are 2 and 3 doing in there? They sure look like values to me! But if you tilt your head a little, you can see them as functions too: **values take no arguments and return themselves**. If we were to write down the inputs and outputs of these functions, it would look like this:

```
2 :: () → (int)
```

$3 :: () \to (\text{int})$

As you may guess, “$x :: T$” means “x is of type T”, and “$T_1 \to T_2$” means “a function from type T_1 to type T_2”. So these functions take no input, and return one integer. We also know the type of the multiplication function, which takes two integers and returns just one:

$\times :: (\text{int}, \text{int}) \to (\text{int})$

Now how do you compose all of these functions together? Remember we said “$f\ g$” means the (reverse) composition of f and g, but how can we compose 2 with 3 when their inputs and outputs don’t match up? You can’t pass an integer to a function that takes no arguments.

The solution lies in something called *stack polymorphism*. Basically, we can give a generic, *polymorphic* type to these functions that says they’ll take *any* input, followed by what they actually need. They return the arguments they don’t use, followed by an actual return value:

$2 :: \forall A.\ (A) \to (A, \text{int})$

$3 :: \forall A.\ (A) \to (A, \text{int})$

$\times :: \forall A.\ (A, \text{int}, \text{int}) \to (A, \text{int})$

“$\forall A.$” means “For all A”—in these examples, even if A has commas in it. So now the meaning of the expression “2 3” is clear:

it is a function that takes no input and returns *both* 2 and 3. This works because when we compose two functions, we match up the output of one with the input of the other, so we start with the following definitions:

$$2 :: \forall A.\ (A) \to (\boldsymbol{A}, \textbf{int})$$
$$3 :: \forall B.\ (\boldsymbol{B}) \to (B, \text{int})$$

We match up the types:

$$(\boldsymbol{A}, \textbf{int}) = (\boldsymbol{B})$$

By substituting, we get a new polymorphic type for 3 within the expression:

$$3 :: \forall C.\ (\boldsymbol{C}, \textbf{int}) \to (\boldsymbol{C}, \textbf{int}, \text{int})$$

This matches the non-polymorphic type:

$$3 :: \forall A.\ (\boldsymbol{A}, \textbf{int}) \to (\boldsymbol{A}, \textbf{int}, \text{int}) = \forall B.\ \boldsymbol{B} \to (\boldsymbol{B}, \text{int})$$

So the final type of the expression becomes:

$$2\ 3 :: \forall A.\ (A) \to (A, \text{int}, \text{int})$$

Going through the same process for multiplication, we get:

$$2\ 3 :: \forall A.\ (A) \to (A, \text{int}, \text{int})$$
$$\times :: \forall B.\ (B, \text{int}, \text{int}) \to (B, \text{int})$$
$$\text{A} = \text{B}$$
$$2\ 3\ \times :: \forall A.\ (A) \to (A, \text{int})$$

This is correct: the expression “2 3 ×” takes no input and produces

one integer. Whew! As a sanity check, note also that the equivalent function “6” has the same type as “2 3 ×”:

$$6 :: \forall A.\ (A) \to (A, \text{int})$$

So already, concatenative languages give us something applicative functional languages generally can’t: we can ***actually* return multiple values from a function, not just tuples.** And thanks to stack polymorphism, we have a uniform way to compose functions of different types, so the flow of data in our programs doesn’t get obscured, as it were, by the plumbing.

♦ ♦ ♦

Cool Stuff

In the above example, we worked from left to right, but because composition is associative, you can actually do it in any order. In math terms, $(f \circ g) \circ h = f \circ (g \circ h)$. Just as “2 3 ×” contains “2 3”, a function returning two integers, it also contains “3 ×”, a function that returns thrice its argument:

$$3\ \times :: (\text{int}) \to (\text{int})$$

(From now on I’ll omit the ∀ bit from type signatures to make them easier to read.)

So we can already trivially represent partial function application. But this is actually a huge win in another way. Applicative languages need to have a defined associativity for function

application (almost always from left to right), but here we're free from this restriction. A compiler for a statically typed concatenative language could literally:

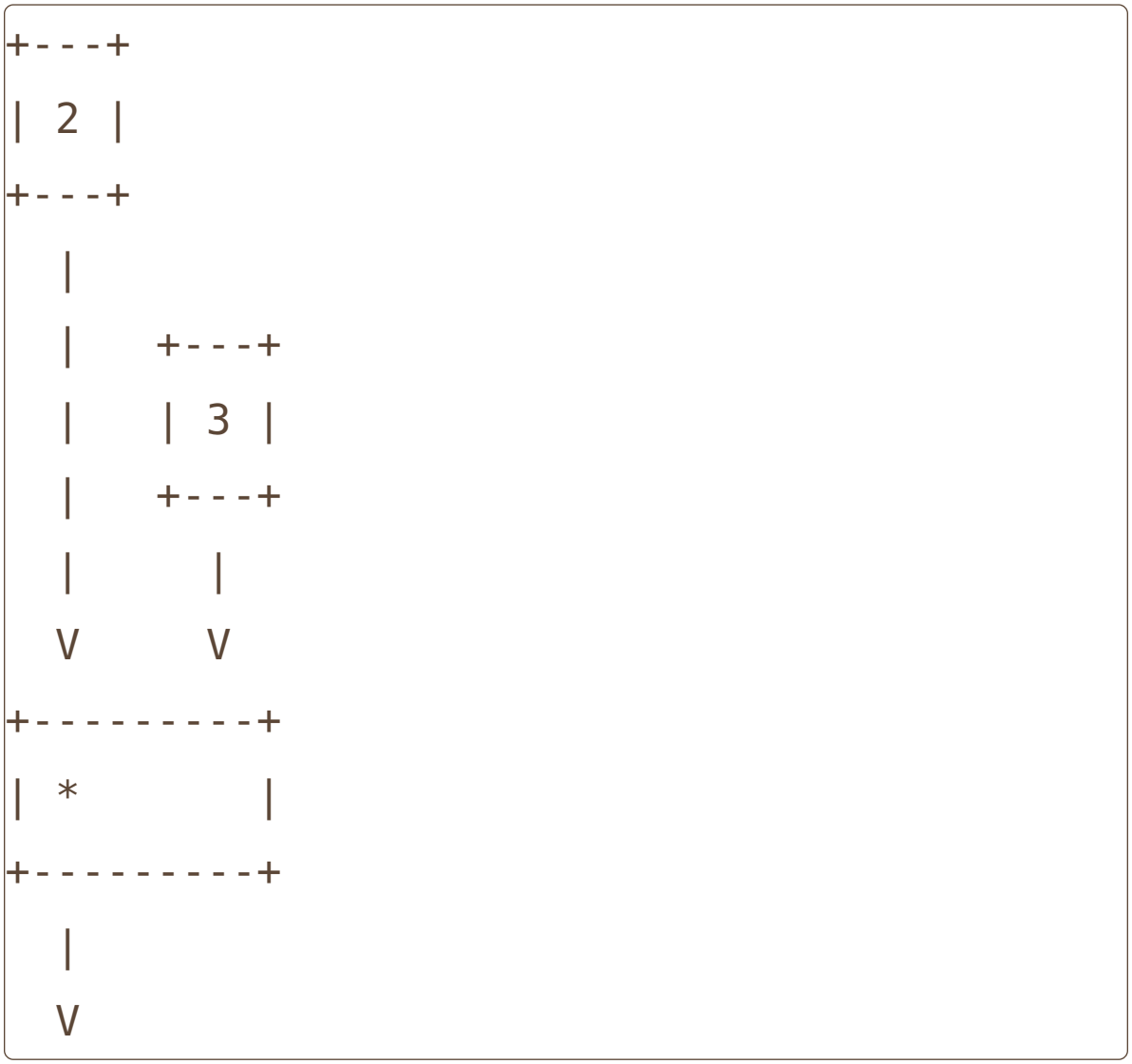
1. Divide the program into arbitrary segments
2. Compile every segment *in parallel*
3. Compose all the segments at the end

This is impossible to do with any other type of language. **With concatenative programming, a parallel compiler is a plain old map-reduce!**

Because all we have are functions and composition, a concatenative program is a *single function*—typically an impure one with side effects, but that's by no means a requirement. (You can conceive of a pure, lazy concatenative language with side-effect management along the lines of Haskell.) Because a program is just a function, you can think about composing programs in the same way.

This is the basic reason Unix pipes are so powerful: they form a rudimentary string-based concatenative programming language. You can send the output of one program to another (`|`); send, receive, and redirect multiple I/O streams (`n<`, `2&>1`); and more. At the end of the day, a concatenative program is all about the flow of data from start to finish. And that again is why concatenative

composition is written in “reverse” order—because it’s *actually forward*:



♦ ♦ ♦

Implementation

So far, I have deliberately stuck to high-level terms that pertain to all concatenative languages, without any details about how they’re actually implemented. One of the very cool things about concatenative languages is that while they are inherently quite functional, they also have a very straightforward and efficient

imperative implementation. In fact, concatenative languages are the basis of many things you use every day:

- The **Java Virtual Machine** on your PC and mobile phone
- The **CPython** bytecode interpreter that powers BitTorrent, Dropbox, and YouTube
- The **PostScript** page description language that runs many of the world's printers
- The **Forth** language that started it all, which still enjoys popularity on embedded systems

The type of a concatenative function is formulated so that it takes any number of inputs, uses only the topmost of these, and returns the unused input followed by actual output. These functions are essentially operating on a list-like data structure, one that allows removal and insertion only at one end. And any programmer worth his salt can tell you what that structure is called.

It's a stack. Yep. Just a stack.

Consider the information flowing between functions in the expression "2 3 × 4 5 × +", which, if you're not up on your postfix, is equal to (2 × 3 + 4 × 5):

Function	Output
	()

2	(2)
3	(2, 3)
×	(6)
4	(6, 4)
5	(6, 4, 5)
×	(6, 20)
+	(26)

Moving from left to right in the expression, whenever we encounter a “value” (remember: a nullary self-returning function), we push its result to the stack. Whenever we encounter an “operator” (a non-nullary function), we pop its arguments, perform the computation, and push its result. Another name for postfix is reverse Polish notation, which achieved great success in the calculator market on every HP calculator sold between 1968 and 1977—and many thereafter.

So a concatenative language is a functional language that is not only easy, but *trivial* to run efficiently, so much so that **most language VMs are essentially concatenative.** x86 relies heavily on a stack for local state, so even C programs have a little bit of concatenativity in ’em, even though x86 machines are register-based.

Furthermore, it's straightforward to make some very clever optimisations, which are ultimately based on simple pattern matching and replacement. The Factor compiler uses these principles to produce very efficient code. The JVM and CPython VMs, being stack-based, are also in the business of executing and optimising concatenative languages, so the paradigm is *far* from unresearched. In fact, a sizable portion of all the compiler optimisation research that has ever been done has involved virtual stack machines.

♦ ♦ ♦

Point-free Expressions

It is considered good functional programming style to write functions in *point-free* form, omitting the unnecessary mention of variables (*points*) on which the function operates. For example, "$f\ x\ y = x + y$" can be written as "$f = (+)$". It is clearer and less repetitious to say "*f* is the addition function" than "*f* returns the sum of its two arguments".

More importantly, it's more meaningful to write what a function *is* versus what it *does*, and point-free functions are more succinct than so-called "pointful" ones. For all these reasons, point-free style is generally considered a Good Thing™.

However, if functional programmers *really* believe that point-free

style is ideal, they shouldn't be using applicative languages! Let's say you want to write a function that tells you the number of elements in a list that satisfy a predicate. In Haskell, for instance:

```
countWhere :: (a -> Bool) -> [a] -> Int
countWhere predicate list = length (filter
predicate list)
```

It's pretty simple, even if you're not so familiar with Haskell. `countWhere` returns the `length` of the list you get when you `filter` out elements of a `list` that don't satisfy a `predicate`. Now we can use it like so:

```
countWhere (>2) [1, 2, 3, 4, 5] == 3
```

We can write this a couple of ways in point-free style, omitting `predicate` and `list`:

```
countWhere = (length .) . filter
countWhere = (.) (.) (.) length filter
```

But the meaning of the weird repeated self-application of the composition operator `(.)` isn't necessarily obvious. The expression `(.) (.) (.)`—equivalently `(.) . (.)` using infix syntax—represents a function that composes a *unary* function (`length`) with a *binary* function (`filter`). This type of composition is occasionally written `.:`, with the type you might expect:

```
(.:) :: (c -> d) -> (a -> b -> c) -> a -> b
```

```
-> d
(.:) = (.) . (.)


countWhere = length .: filter
```

But what are we *really* doing here? In an applicative language, we have to jump through some hoops in order to get the basic concatenative operators we want, and get them to typecheck. When implementing composition in terms of application, **we must explicitly implement every type of composition.**

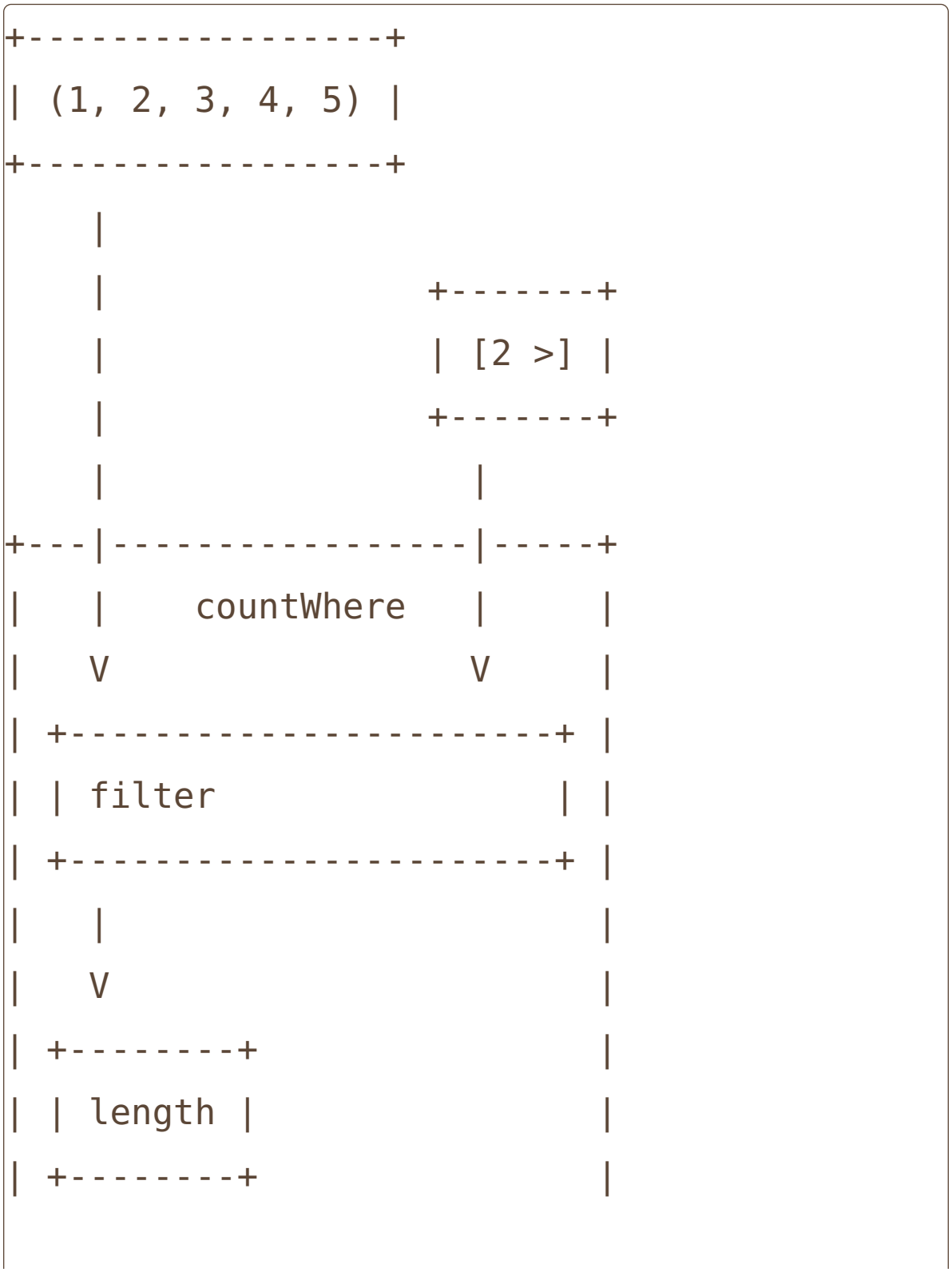
In particular, there is no uniform way to compose functions of different numbers of arguments or results. To even get close to that in Haskell, you have to use the `curry` and `uncurry` functions to explicitly wrap things up into tuples. No matter what, you need different combinators to compose functions of different types, because Haskell doesn't have stack polymorphism for function arguments, and it *inherently can't.*

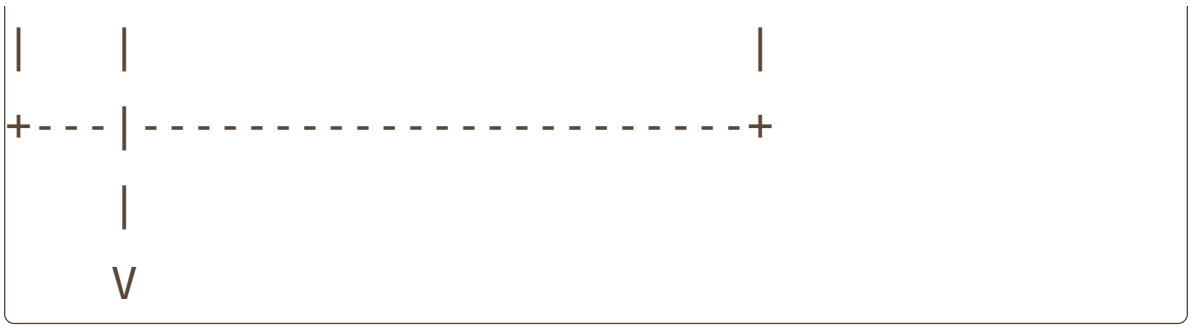
Writing point-free expressions demands concatenative semantics, which just aren't natural in an applicative language. Now contrast a concatenative example:

```
define countWhere [filter length]
(1, 2, 3, 4, 5) [2 >] countWhere
```

It's almost painfully literal: "to count the number of elements in a

list that match a predicate is to filter it and take the length”. In a concatenative language, there is no need at all for variables, because all you’re doing is building a machine for data to flow through:





When you're building a diagram like this, just follow a few simple rules:

1. Each block is one function
2. A block takes up as many columns as needed by its inputs or outputs, whichever are more
3. When adding a block, put it in the rightmost column possible:
   - If it takes no inputs, add a column
   - If there aren't enough arrows to match the block, the program is ill-typed

♦ ♦ ♦

Quotations

Notice the use of brackets for the predicate [2 >] in the preceding example? In addition to composition, the feature that completes a concatenative language is *quotation*, which allows *deferred* composition of functions. For example, "2 >" is a function that returns whether its argument is greater than 2, and [2 >] is a function that returns "2 >".

It's at this point that we go meta. While just *composition* lets us build descriptions of dataflow machines, *quotation* lets us build machines that *operate on descriptions of other machines*. Quotation eliminates the distinction between code and data, in a simple, type-safe manner.

The "filter" machine mentioned earlier takes in the *blueprints* for a machine that accepts list values and returns Booleans, and filters a list according to the instructions in those blueprints. Here's the type signature for it:

$$\text{filter} :: (\text{list } T, T \to \text{bool}) \to (\text{list } T)$$

There are all kinds of things you can do with this. You can write a function that applies a quotation to some arguments, without knowing what those arguments are:

$$\text{apply} :: \forall A\ B.\ (A, A \to B) \to (B)$$

You can write a function to compose two quotations into a new one:

$$\text{compose} :: \forall A\ B\ C.\ (A \to B, B \to C) \to (A \to C)$$

And you can write one to convert a function to a quotation that returns it:

$$\text{quote} :: (T) \to (() \to T)$$

♦ ♦ ♦

The Dark Side

Say you want to convert a math expression into concatenative form:

$$f\ x\ y\ z = y^2 + x^2 - |y|$$

This has a bit of everything: it mentions one of its inputs multiple times, the order of the variables in the expression doesn't match the order of the inputs, and one of the inputs is ignored. So we need a function that gives us an extra copy of a value:

$$\text{dup} :: (T) \to (T, T)$$

A function that lets us reorder our arguments:

$$\text{swap} :: (T_1, T_2) \to (T_2, T_1)$$

And a function that lets us ignore an argument:

$$\text{drop} :: (T) \to ()$$

From the basic functions we've defined so far, we can make some other useful functions, such as one that joins two values into a quotation:

$$\text{join}_2 :: (T_1, T_2) \to (() \to (T_1, T_2))$$

$$\text{join}_2 = \text{quote swap quote swap compose}$$

Or a function that rotates its three arguments leftward:

$$\text{rot}_3 :: (T_1, T_2, T_3) \to (T_2, T_3, T_1)$$

> $\text{rot}_3 = \text{join}_2$ swap quote compose apply

And hey, thanks to quotations, it's also easy to declare your own control structures:

```
define true  [[drop apply]]      # Apply the
first of two arguments.
define false [[swap drop apply]] # Apply the
second of two arguments.
define if    [apply]             # Apply a
Boolean to two quotations.
```

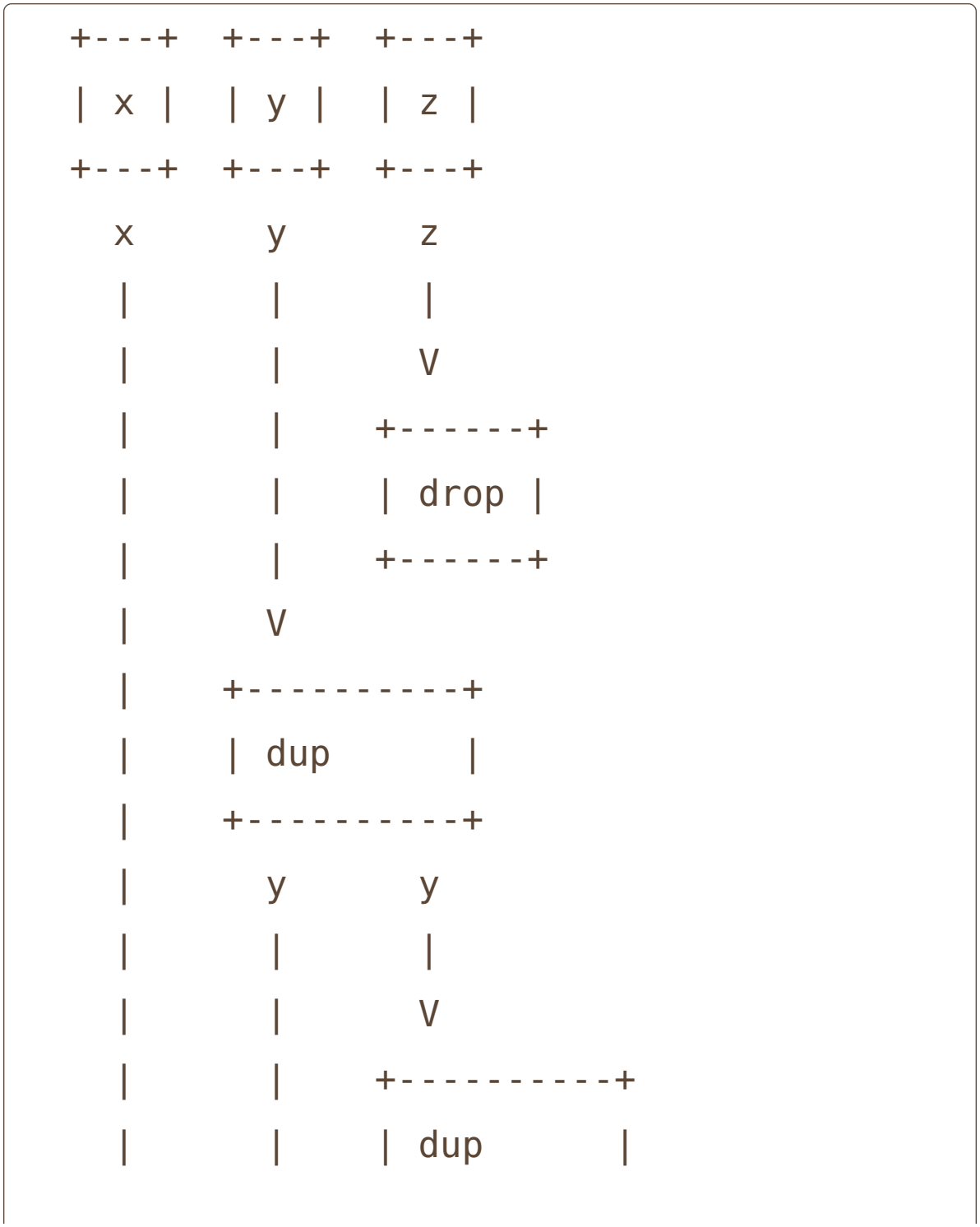
And using them is like—actually, just *is*—using a regular function:

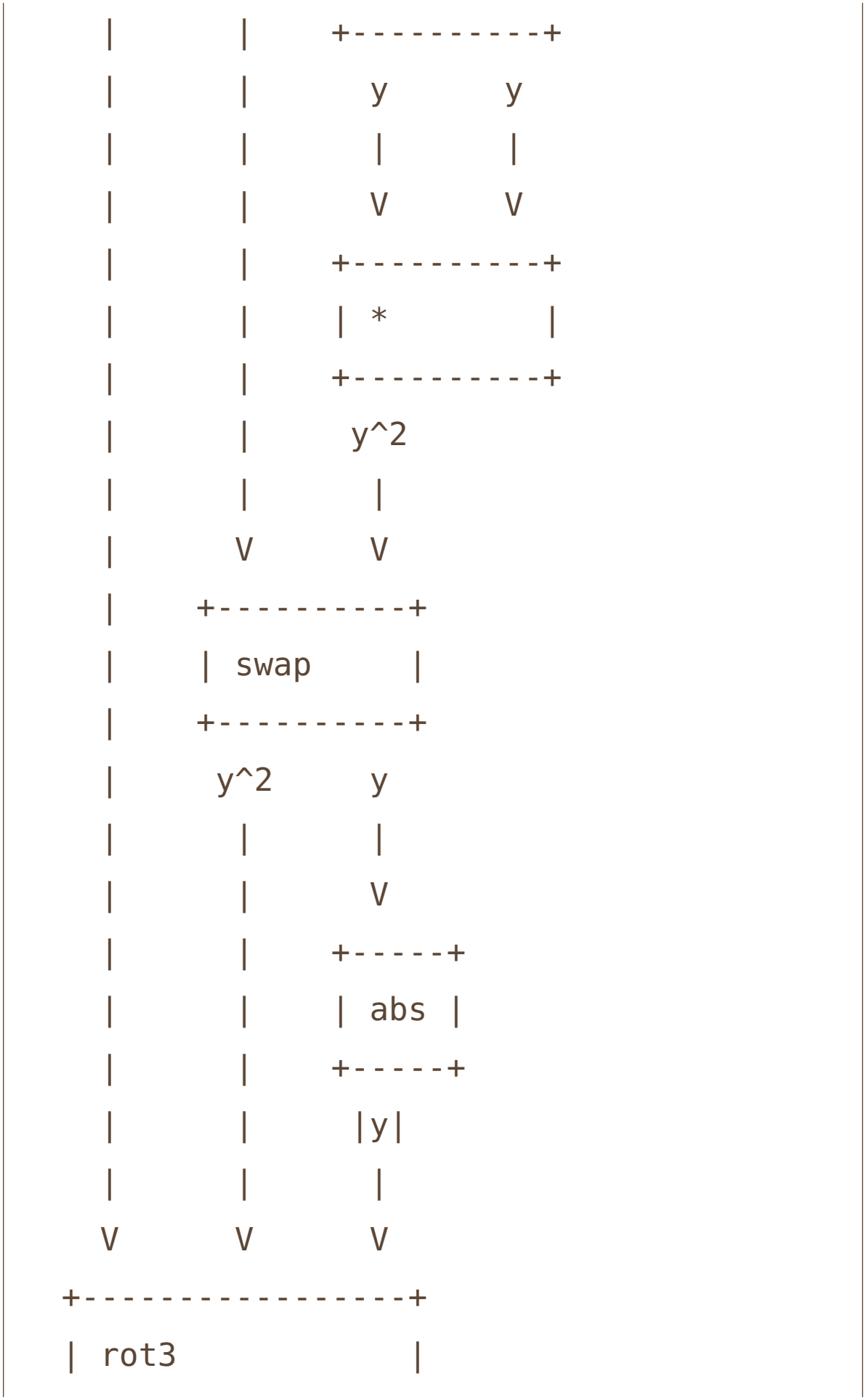
```
["2 is still less than 3."]    # "Then"
branch.
["Oops, we must be in space."] # "Else"
branch.
2 3 <                          # Condition.
if print                       # Print the
resulting string.
```

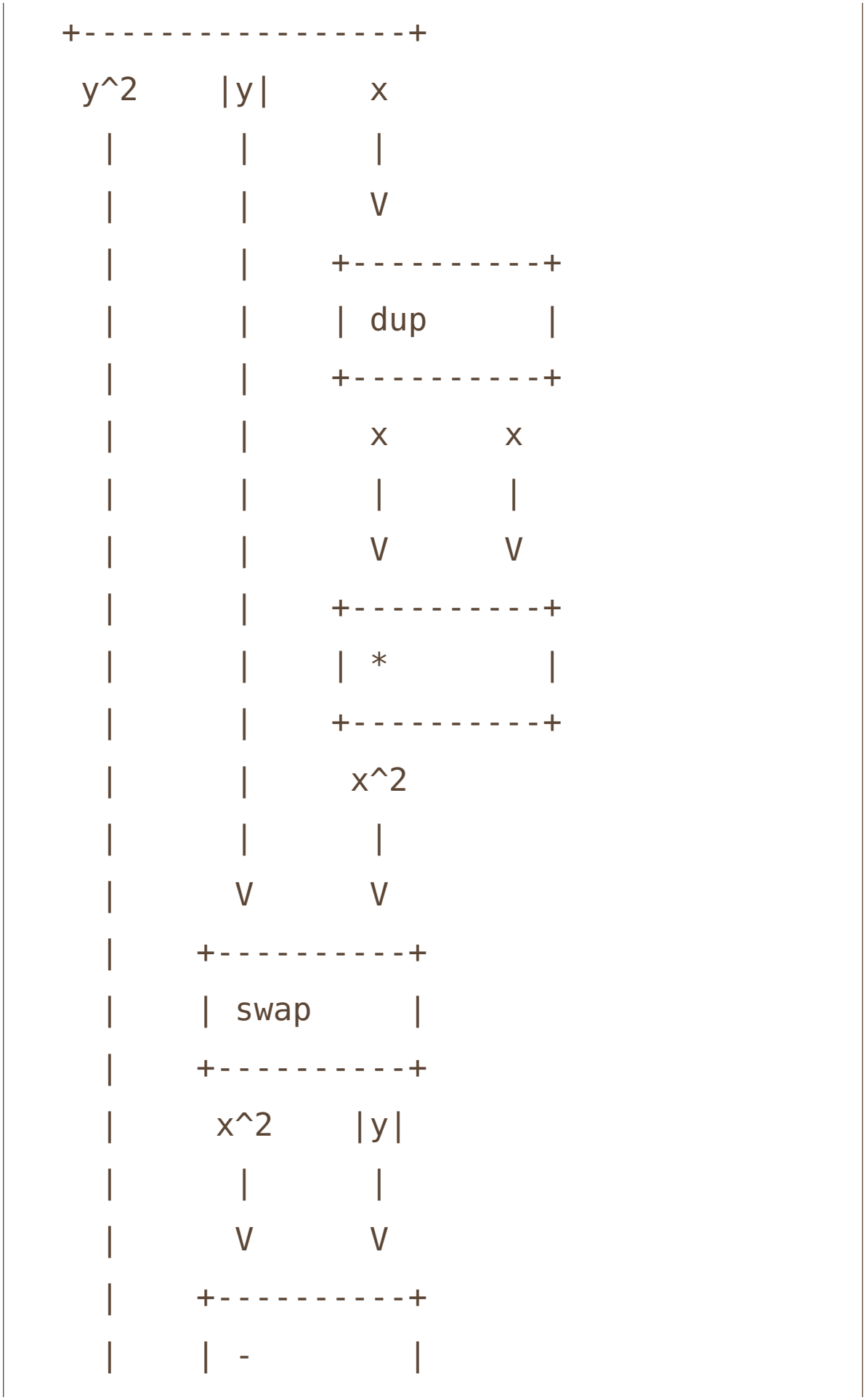
Those particular definitions for `true` and `false` will be familiar to anyone who's used Booleans in the λ-calculus. A Boolean is a quotation, so it behaves like an ordinary value, but it contains a binary function that chooses one branch and discards another. "If-then-else" is merely the application of that quotation to the

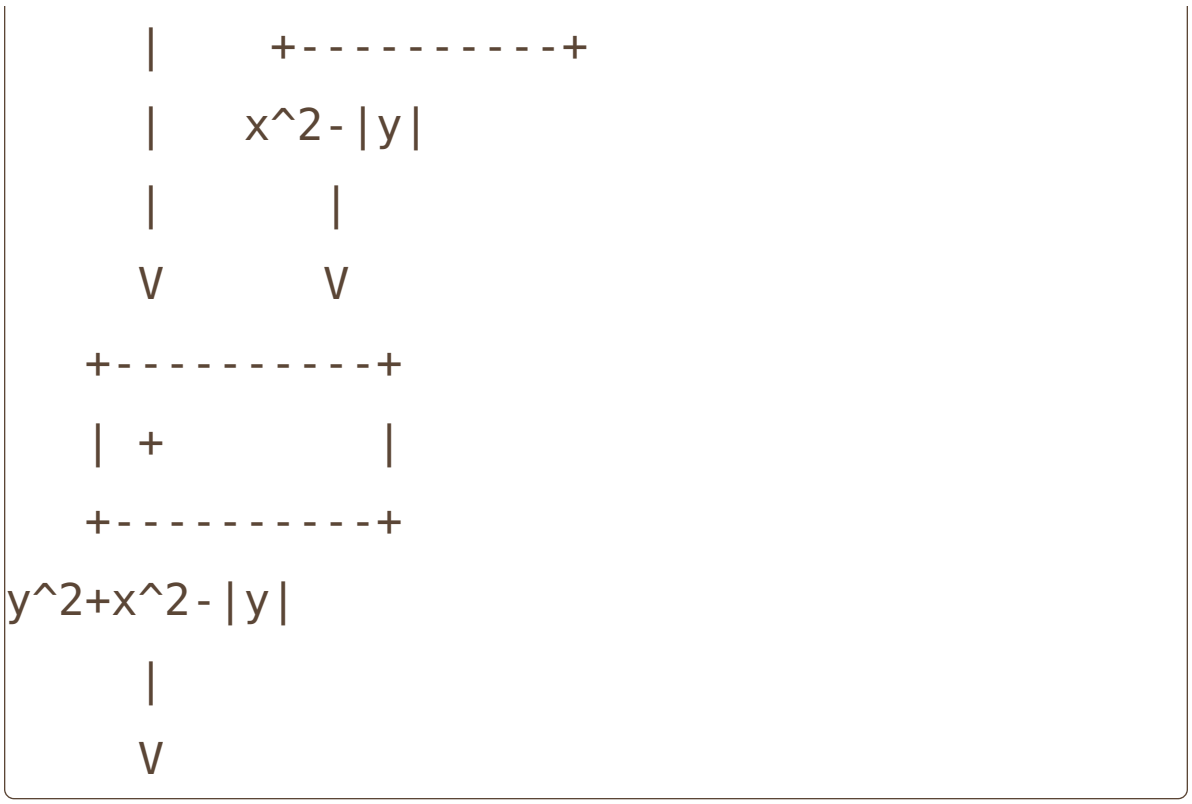
particular branches.

Anyway, getting back to the math, we already know the type of our function ((int, int, int) → (int)), we just need to deduce how to get there. If we build a diagram of how the data flows through the expression, we might get this:




y
y
*
y^2
swap
y^2
y
abs
|y|
rot3


y^2
|y|
x
dup
x
x
*
x^2
swap
x^2
|y|
-



So our final function can be written:

$$f = \text{drop dup dup} \times \text{swap abs rot}_3 \text{ dup} \times \text{swap} - +$$

Well…that sucked.

♦ ♦ ♦

A Lighter Note

You've just seen one of the major problems with concatenative programming—hey, every kind of language has its strengths and weaknesses, but most language designers will lie to you about the latter. **Writing seemingly simple math expressions can be difficult and unintuitive,** especially using just the functions we've seen so far. To do so exposes all of the underlying complexity of the expression that we're accustomed to deferring to

a compiler.

Factor gets around this by introducing a facility for lexically scoped local variables. Some things are simply more natural to write with named state rather than a bunch of stack-shuffling. However, the vast majority of programs are not predominated by mathematical expressions, so in practice this feature is not used very much:

> “Out of 38,088 word and method definitions in the source code of Factor and its development environment at the time of this writing, 310 were defined with named parameters.”—Factor: A Dynamic Stack-based Programming Language

One of the great strengths of concatenative languages, however, is their ability to refactor complex expressions. Because every sequence of terms is just a function, you can *directly* pull out commonly used code into its own function, without even needing to rewrite anything. There are generally no variables to rename, nor state to manage.

So in practice, a lot of expressions can be refactored using a wide array of handy functions, of the sort that commonly end up in a standard library. With some refactoring, that math expression might look like this:

```
square = dup ×
```

```
f = drop [square] [abs] bi − [square] dip +
```

Which doesn't look so bad, and actually reads pretty well: the difference between the square and absolute value of the second argument, plus the square of the first. But even that description shows that our mathematical language has evolved as inherently applicative. It's better sometimes just to stick to tradition.

♦ ♦ ♦

Whither Hence

So you've got the gist, and it only took a few dozen mentions of the word "concatenative" to get there. I hope you're not suffering from semantic satiation.

You've seen that concatenative programming is a paradigm like any other, with a *real definition* and its own pros and cons:

- Concatenative languages are simple and consistent ("everything is a")
- They are amenable to dataflow programming
- Stack languages are well studied and have good performance
- You can easily roll your own control structures
- Everything is written in point-free style (for better or worse)

If you're interested in trying out a mature, practical concatenative language, check out Factor and the official blog of the creator,

Slava Pestov. Also see Cat for more information on static typing in concatenative languages.

I've been idly working on a little concatenative language called Kitten off and on for a while now. It's dynamically typed and compiles to C, so you can run it just about anywhere. I wanted a language I could use for a site on a shared host where installing compilers was irritating. That shows you the extent of my language geekery—I'd rather spend hours writing a language than twenty minutes figuring out how to install GHC on Bluehost.

Anyway, the implementation is *just barely* complete enough to play around with. Feel free to browse the source, try it out, and offer feedback. You'll need GHC and GCC to build it, and I imagine it works best on Linux or Unix, but there shouldn't be any *particularly* horrible incompatibilities.

This would also be a good time to mention that I'm working on a more serious language called Magnet, which I mentioned in my last article about how programming is borked. It's principally concatenative, but also has applicative syntax for convenience, and relies heavily on the twin magics of pattern matching and generalised algebraic data types. Hell, half the reason I wrote this article was to provide background for Magnet. So expect more articles about that.

Edit (20 April 2013) The above information is no longer accurate.

Kitten is currently being rewritten; the work that was done on Magnet has been incorporated. Kitten is now statically typed and, until the compiler is complete, interpreted.

And that about does it. As always, feel free to email me at evincarofautumn@gmail.com to talk at length about anything.

Happy coding!