Title of the Book

Author Name

Urbit

The Personal Cloud Computer

C.G. Yarvin, J.C. Burnham, M.D. Hartl

Contents

“Man is a Tool-using Animal. Weak in himself, and of small stature, he stands on a basis, at most for the flattest-soled, of some half-square foot, insecurely enough; has to straddle out his legs, lest the very wind supplant him. Feeblest of bipeds! Three quintals are a crushing load for him; the steer of the meadow tosses him aloft, like a waste rag. Nevertheless he can use Tools; can devise Tools: with these the granite mountain melts into light dust before him; he kneads glowing iron, as if it were soft paste; seas are his smooth highway, winds and fire his unwearying steeds. Nowhere do you find him without Tools; without Tools he is nothing, with Tools he is all.”

-Carlyle

Chapter 1

Preface: Urbit

> A complex system that works is invariably found to have evolved from a simple system that worked. The inverse proposition also appears to be true: A complex system designed from scratch never works and cannot be made to work. You have to start over, beginning with a working simple system.
>
> Gall's Law

Urbit is a general-purpose programming environment designed to execute a purely functional network operating system and a referentially transparent network address space.

Urbit consists of four conceptual levels:

At the lowest-level, Urbit reduces to Nock, a stateless virtual machine based on combinatory logic and defined in 340 bytes. The Nock machine is sealed - all execution is "pure." Nock's goal is extreme commoditization of computing semantics.

Nock acts as our functional equivalent of assembly language. Nock is differs from assembly language in that its interpreted, rather than directly executed by the hardware. It is like assembly language in that everything in Urbit executes as Nock and most programming in Urbit will be done using higher-level tools

that reduce to Nock.

Hoon is a high-level language which defines itself in Nock. Its self-compiling kernel, 7000 lines of code, specifies Hoon unambiguously; there is no Hoon spec. Hoon can be classified as a pure, strict higher-order static type-inferred functional language, with co/contra/bivariance and genericity. However, Hoon does not use lambda calculus, unification, or other constructs from “PL theory.”

Hoon also excels at handling and validating untyped data, a common task on the Internet. Its syntax is entirely novel and initially quite frightening a consequence of Hoon having been designed to optimize for actual usability rather than cosmetic friendliness. A frequent experience amongst Hoon programmers is writing code that works the first time it successfully compiles.

Arvo is a deterministic functional operating system and network protocol, written in 5000 lines of Hoon. Arvo features a revision control filesystem and the network is a blocking global immutable namespace. Meaning that all data on the network or off it can be treated as a constant. Additionally, Arvo preserves a log of all the events it sees, meaning that all computation on an Arvo instance is recorded and can be played back and examined.

The network blah blah blah

Chapter 2

Introduction: What is a Personal Cloud Computer?

> Tlon's three rules of platform warfare:
>
> 1. General-purpose computers beat special-purpose computers.
> 2. Personal computers beat industrial computers.
> 3. Cloud computers beat ground computers.

Urbit is an open-source software stack that runs a Personal Cloud Computer, or PC2.

A PC2 is any general-purpose machine, physical or virtual, that operates natively on the Internet (cloud), and whose operation is sufficiently simple that it can be understood and controlled by the user (personal).

The PC2 is general-purpose, in that it can run programs and applications that perform a wide range of functions. This is contrasted with special-purpose platforms, such as a pocket calculator or consumer web application, that can only perform a single function, or limited range of related functions.

The PC2 is personal, in that the ordinary user can understand its operation without requiring a high degree of specialized technical skill. This is contrasted

with industrial services, such a Linux server, whose computational semantics are rendered largely opaque by their complexity, even to the technically skilled.

The PC2 is a cloud computer, in that it intrinsically operates on a network and blurs the distinction between network and local resources. This is contrasted with non-network, or ground, computers, such as the early personal computer, which maintains a sharp distinction between data on its own disk and data on the network.

The individual user can easily setup and administer a PC2 on their own, run and write a endless variety of applications, and can not only can access their programs and data seamlessly from any client, but can serve any kind of content to all other clients on the network. Meaning that all programs running on a PC2 can communicate with programs running on different PC2's as easily as they can communicate with programs on the same PC2. The PC2 is not just a general-purpose client, it is also a general-purpose server. In other words, a network of PC2's is, by definition, a general-purpose social network.

By no means is this concept an idea original to Urbit. Many, many systems desinged to implement the PC2 have been tried and, without exception, have failed to do so. These previous attempts have been crippled by their inadequate solution two fundamental problems: software portability and network identity.

Portable software can be trivially executed with the same semantics on any general-purpose computer. One can transfer portable from one machine to any other machine, and expect it to produce the same output. If the two machines differ in their output, it is easy to see which one is correct and which has an an implementation error.

In contrast, unportable sofware must be modified for each platform it is run on without any expectation of identical semantics on different platforms and with complex chains of dependency on other software tools and libraries, also ususally different per platform.

Unportable software drastically increases complexity, making it difficult for ordinary users to understand and administer their own general-purpose com-

puter and incentivizing them to use special-purpose applications and services administered by others. A lack of portability also makes the goal of indifference to resource locality significantly harder to acheive.

[Section on Network Identity needs reworking]

Network identity can best be thought of in the context of the classic computer science problem of Zooko's Triangle, which states:

The address space of any network can be at most two of secure, human-meaningful and decentralized.

Secure means that it is impractical for any node on the network to impersonate another, i.e. when any talks to any other node can authenticate that the node is who they say they are.

Human-meaningful means that addresses on the network present a good user interface, e.g. a domain name, such as pets.com, is human-meaningful, an IP address is not.

Decentralized means that addresses and identities on the network are not stored with and verified by a central party.

Choosing only secure and human-meaningful but not decentralized effectively turns your network a large distributed server. Consumer web applications that claim to be social "networks" are, in fact, simply social servers.

Networks that are secure and decentralized, but not human-meaningful present a poor interface to the user and fail to gain adoption, because long numbers are difficult to distiguish from one another.

And sacrificing secure for human-meaningful and decentralized means that the network is impossible to defend against malicious activity such as spam.

Succinctly, the problem of network identity is: How does one authenticate another node in the network? And how does one distinguish between a new user and a returning spammer?

It would not be too broad a mischaracterization to describe Urbit as a system that implements portable software on a network with strong identity.

Urbit solves the problem of software portability with a stateless virtual machine called Nock, which can be thought of as a “Maxwell’s equations” of software. The Nock machine is defined in 340 bytes of pseudocode and executes a pure Turing-complete function. Nock is not the smallest Turing-complete computational automaton, but is easily one of the smallest practical ones. Everything implemented in Urbit is built on top of Nock.

Nock is portable to any platform that can run a Nock interpreter, which is all general-purpose platforms. Since the Nock specification is conunambigs If two Nock interpreters disagree with one another, it is trivial to determine which is wrong and which is right.

Urbit solves the problem of network identity by embedding

[Section on Network Identity needs reworking]

This book is designed for the novice programmer. . .

Chapter 3

Introductory Nock

What one fool can do, another can.

—**Ancient Simian proverb**

Think of Nock as a kind of functional assembly language. It's not like assembly language in that it's directly executed by the hardware. It is like assembly language in that (a) everything in Urbit executes as Nock; (b) you wouldn't want to program directly in Nock; and (c) learning to program directly in Nock is a great way to start understanding Urbit from the ground up.

Just as Unix runs C programs by compiling them to assembler, Urbit runs Hoon programs by compiling them to Nock. You could try to learn Hoon without learning Nock. But just as C is a thin wrapper over the physical CPU, Hoon is a thin wrapper over the Nock virtual machine. It's a tall stack made of thin layers, which is much easier to learn a layer at a time.

And unlike most fundamental theories of computing, there's really nothing smart or interesting about Nock. Of course, in a strictly formal sense, all of computing is math. But that doesn't mean it needs to feel like math. Nock is a simple mechanical device and it's meant to feel that way.

Let's get start by learning how to use Urbit's operating system, Arvo, to evaluate Nock code.

3.1 Section I: Getting Started

We'll assume that you've gone through the Urbit setup process and have an Arvo prompt that looks something like this:

```
~tomsyt-balsen/try=>
```

At your prompt, type the following exactly:

```
~tomsyt-balsen/try=> .*(42 [0 1])
```

This should return `42`. Don't worry about what this is doing yet.

It's very important that you actually go to your prompt and type in our examples. Copying and pasting is cheating. This might seem silly, but to learn Nock (or any language) it's very important that your fingers get comfortable writing it.

If you accidentally make a mistake typing in a Nock expression, you'll get a syntax error:

```
~tomsyt-balsen/try=> .*(42[0 1])
~ <syntax error at [1 6]>

~tomsyt-balsen/try=> .*( 42 [0 1])
~ <syntax error at [1 4]>
```

Easiest way to get a sytax error is to accidentally leave out a space. Or add an extra one. Fortunately, the error message tells you where the mistake is:

```
~ <syntax error at [1 4]>
```

means that there's an error at `line 1`, `column 4`. Once you know where an error is, it's much easieer to fix.

But even if your Nock expression is formatted correctly, you might get something that looks like this:

```
~tomsyt-balsen/try=> .*(42 [5 0 1])
! exit
```

This means that the expression you typed in is correct Nock, but it just doesn't produce anything. Unfortunately, we can't give you line and column numbers on this one, so the only surefire way to debug an exit message is to understand what your code is doing. Literally speaking, an exit message means you tried to do something that just doesn't make sense. Such trying to reference data that doesn't exist, or trying to increment something that's not a number, or asking if `42` is equal, without asking what it's equal to.

Enough about errors, let's practice some expressions that work:

```
~tomsyt-balsen/try=> .*(41 [0 1])
41

~tomsyt-balsen/try=> .*(40 [0 1])
40

~tomsyt-balsen/try=> .*(374 [0 1])
374
```

The perceptive reader will notice the pattern here: If a is a number, `.*(a [0 1])` always produces `a`. To test it, run the following, but replace `a` with any number you like.

```
~tomsyt-balsen/try=> .*(a [0 1])
a
```

Once you're satisfied that this is true, let's do something slightly different:

```
~tomsyt-balsen/try=> .*(374 [1 0])
0

~tomsyt-balsen/try=> .*(40 [1 0])
0

~tomsyt-balsen/try=> .*(41 [1 0]) /
0
```

This pattern is pretty easy: `*(a [1 0])` always produces `0`, no matter what a is.

Again, play around with the above yourself by choosing your own values for a:

```
~tomsyt-balsen/try=> .*(a [1 0])
0
```

One more pattern, and then we'll actually explain what these numbers and brackets represent:

```
~tomsyt-balsen/try=> .*(374 [1 374])
374

~tomsyt-balsen/try=> .*(40 [1 374])
374

~tomsyt-balsen/try=> .*(374 [1 40])
40
```

As an exercise, run the last three lines again but replace `374` and `40` with number of your own.

You've probably already guessed the pattern here: `.*(a [1 b])` always produces `b`, regardless of `a`. But feel free to test it, replacing `a` and `b` with any number.

```
~tomsyt-balsen/try=> .*(a [1 b])
b
```

Let's run the following piece code again:

```
~tomsyt-balsen/try=> .*(42 [0 1])
```

Nock is made up of two basic building blocks: atoms, which can be any non-negative whole number, and cells, which are pairs of numbers or cells (cells can go inside of other cells.) `42`, for example, is an atom. `[0 1]` is a cell. Even `[42 [0 1]]` is a cell. (It's very important to note that cells can nest inside other cells.)

Collectively, both atoms and cells are called nouns. And Nock is just a list of rules (or a set of patterns) for transforming nouns. You put one noun in, you get another noun out. It's like algebra, `f(x) = y`. `nock(noun_a) = noun_b`

The atom `42` is a noun, the cell `[0 1]` is a noun, In fact, `.*(42 [0 1])` is a noun too, except that its wrapped in syntax that tells Arvo "run this noun through Nock". `.*(42 [0 1])` is actually the way to tell Arvo (Urbit's operating system) to evaluate `Nock([42 [0 1]])`. The cellular noun `[42 [0 1]]` goes into Nock, the atomic noun `42` comes out.

We could simply write our very first example again as

```
nock([42 [0 1]]) = 42
```

except that we wouldn't be able to evaluate it in Arvo.

Actually, we almost never use the above mathematical notation. For the sake of brevity we almost always write nock(a) using the notation `*a`. Thus, nock([42 [0 1]]) `*[42 [0 1]]` mean the same thing. If we want to tell Arvo to evaluate the noun, we use `.*(42 [0 1])`. You will notice that the outermost brackets somewhat confusingly disappear when we use the .*() function in Arvo, we will explain why in [replaceme: Chapter 2. Section 1]

But in documenting Nock, we most frequenly use the `*[42 [0 1]]` style of notation:

Instead of, for example, writing

```
~tomsyt-balsen/try=> .*(a [1 b])
b
```

the standard way of writing out the rules of Nock is

```
*[a [1 b]]                  b
```

with the left hand side being the noun that matches our input and the right side being the product of that input.

We've been using `a` and `b` as variables that represent numbers (i.e. atoms), but we can and do use them more broadly to represent nouns in general.

Let's apply the rule

```
*[a [1 b]]                  b
```

which, to reiterate, means that when we run any noun of the form [a [1 b]] through Nock (using the expression `.*(a [1 b])` in Arvo) always produces `b`, regardless of `a`.

Let's run a few examples:

```
~tomsyt-balsen/try=> .*(301 [1 374])
374

~tomsyt-balsen/try=> .*([42 43] [1 312])
312

~tomsyt-balsen/try=> .*(374 [1 [44 48]])
[42 43]
```

```
~tomsyt-balsen/try=> .*([46 49] [1 [456 539]])
[456 539]

~tomsyt-balsen/try=> .*(374 [1 [[[32 34] 33]])
[[[31 32] 33]
```

To conclude this section, let's review what we've learned:

3.1.1 Summary:

Notation:

Arvo syntax:

```
~tomsyt-balsen/try=> .*(42 [0 1])
42
```

Math notation:

```
nock([42 [0 1]]) = 42
```

Nock notation:

```
*[42 [0 1]]                 42
```

Error Messages:

A syntax error occurs when an expression in Arvo is not typed in correctly.

```
~tomsyt-balsen/try=> .*( 42 [0 1])
~ <syntax error at [1 4]>
```

The cell in the syntax error gives the line and column number of the location of the error.

```
~tomsyt-balsen/try=> .*(42 [5 0 2])
! exit
```

Structures:

A noun is an atom or a cell. An atom is a natural number. A cell is an ordered pair of two nouns. I.e. two atoms, two cells, or a cell and an atom.

Nock Rules

Lower-case letters such as `a` or `b` are variables that represent nouns.

Nock rules are notated with two columns, where the left hand side indicates what pattern the noun matches, and the right hand side indicates what the noun produces.

```
*[a [0 1]]                  a
*[a [1 b]]                  b
```

___

3.1.2 Exercises:

1. Take the noun [[a b] c] and replace each of `a` `b` and `c` with a cell. Test your cell by applying, in Arvo using the correct syntax, the rule:

```
*[a [0 1]]                  a
```

2. Write a cell that contains 12 atoms. Test your cell by applying, in Arvo using the correct syntax, the rule:

```
*[a [1 b]]                  b
```

   setting `a` eqaul to `1`.

3. Produce a syntax error that reads:

```
~ <syntax error at [1 14]>
```

4. With pen and paper, write out all three possible forms of notation for each of the expressions you evaluated in exercises 1. and 2.

3.2 Section II: Noun Structure

Last section we played around with this:

```
~tomsyt-balsen/try=> .*(42 [0 1])
42
```

which corresponds to the pattern

```
*[a [0 1]]                  a
```

Now we're going to experiment with what happens when we replace `1` with different values. Remember, as you follow along, type out the examples at your own Arvo prompt.

```
~tomsyt-balsen/try=> .*([42 43] [0 1])
[42 43]

~tomsyt-balsen/try=> .*([42 43] [0 2])
42

~tomsyt-balsen/try=> .*([42 43] [0 3])
43

~tomsyt-balsen/try=> .*([42 43] [0 4])
! exit
```

We don't really have enough examples to figure out the pattern yet. Let's change `[42 43]` to `[[44 45] 43]` and try some more.

```
~tomsyt-balsen/try=> .*([[44 45] 43] [0 1])
[[44 45] 43]

~tomsyt-balsen/try=> .*([[44 45] 43] [0 2])
[44 45]

~tomsyt-balsen/try=> .*([[44 45] 43] [0 3])
43

~tomsyt-balsen/try=> .*([[44 45] 43] [0 4])
44

~tomsyt-balsen/try=> .*([[44 45] 43] [0 5])
45

~tomsyt-balsen/try=> .*([[44 45] 43] [0 6])
! exit
```

It looks like anything of the form `*[a [0 b]]` produces the sub-nouns that are inside of a. Remember that notationally, `*[a [0 b]]` is the same as `.*(a [0 b])` e.g. a is `[[44 45] 43]` and b is one of the atoms `1` through `6`.

But how does `*[a [0 b]]` know which pieces of a to choose?

Let's do one more example and then we'll explain it.

```
~tomsyt-balsen/try=> .*([42 [46 47]] [0 1])
[42 [46 47]]

~tomsyt-balsen/try=> .*([42 [46 47]] [0 2])
42

~tomsyt-balsen/try=> .*([42 [46 47]] [0 3])
[46 47]

~tomsyt-balsen/try=> .*([42 [46 47]] [0 4])
! exit

~tomsyt-balsen/try=> .*([42 [46 47]] [0 5])
! exit

~tomsyt-balsen/try=> .*([42 [46 47]] [0 6])
46

~tomsyt-balsen/try=> .*([42 [46 47]] [0 7])
47
```

Try to solve this puzzle on your own by playing with the following:

```
~tomsyt-balsen/try=> .*(a [0 b])
```

where `a` is a cell and `b` is an atom. Try to pick atoms for `b` that are small and try to pick cells for `a` that have lots of nesting.

When you're ready to have things explained to you, read on.

Think of a noun as a tree structure:

```
  [42 [46 47]]
  /       \
42      [46 47]]
          / \
        46   47
```

Every cell has two branches (the head of the cell and the tail) leading down from it. Atoms have no branches, because they can't be broken down any further (and this is exactly why they're called atoms).

Let's look at the tree of the other noun we played with, `[[44 45] 43]`:

```
 [[44 45] 43]
    /      \
[44 45]    43
 /   \
44   45
```

It should be pretty obvious that we could change the values of any of the atoms in the tree without changing the structure of the tree. That is to say, `[[44 45] 43]` and `[[24 25] 23]` have the same tree structure:

```
 [[22 25] 23]
    /      \
[22 25]    23
 /   \
22   25
```

And now, for a more complicated tree, here's the noun `[[[48 49] 45] [46 47]]`:

```
  [[[48 49] 45] [46 47]]
       /            \
 [[48 49] 45]      [46 47]
   /       \        /   \
[48 49]    45      46   47
 /   \
48   49
```

So how do the above trees relate to running `.*(a [0 b])`? Simple, every part of the tree gets mapped to an atomic address (we call it an axis). The mapping looks something like this:

```
          1
        /   \
      2       3
    /   \   /   \
  4      5 6     7
 / \    / \ / \  / \
8   9 10 11 12 13 14 15
```

Or, because the lines are kind of ugly:

```
         1
     2       3
 4     5   6   7
8 9 10 11 12 13 14 15
```

Of course, this only a very small part of the entire tree. We extend the tree by applying the rule: Every axis `/n` has a head with the axis `/2n` and a tail with the axis `/2n+1`"

We map from noun to axis by comparing the tree of the noun with the axis tree and seeing what matches. Like so, marking axes with a `/` character:

```
  /1 : [[44 45] 43]
         /      \
 /2 : [44 45]    43 : /3
      /    \
/4 : 44     45 : /5
```

Again, because it bears repeating: the head of axis `/n` is `/2n` and the tail of axis `/n` is `/2n+1`. Remember that the head is the left-hand noun and the tail the right-hand noun of a cell-pair.

Start with 1. This is your root axis. All nouns have a valid axis `/1`, even atoms. and the axis `/1` just refers to the noun itself. In the above example, axis /1 of `[[44 45] 43]` is just `[[44 45] 43]`. The head of `[[44 45] 43]` is `[44 45]` and the tail is `43`. Thus, axis `/2` of `[[44 45] 43]` is `[44 45]` and axis /3 is `43`.

Another way to think about it is that the tree map has layers:

```
layer 0             1
layer 1         2           3
layer 2      4     5     6     7
layer 3    8 9 10 11 12 13 14 15
```

which correspond to the nesting depth of the noun. If a noun is inside two cells, like `46` inside `[[[48 49] 45] [46 47]]` then its axis is at layer 2 of the tree. If its inside three cells like `48`, then its axis is at layer three.

Recall the pattern we learned in section 1:

```
*[a [0 1]]                  a
```

This, we now see, is a special case of

```
*[a [0 b]]              axis /b of a
```

The 0 in `*[a [0 b]]` is just an operator that means axis. Nock maps simple operators and functions to atoms, instead of a character like `/` for example, because atoms (and cells) are all Nock knows. This would Fortunately for us, there are only eleven atoms that are operators, atoms `0` through `10`.

3.2.1 Summary:

Noun structure:

Nouns are trees that look like this:

```
 [[44 45] 43]
    /      \
[44 45]    43
 /   \
44   45
```

The left-hand of a cell is called the head. The right hand is the tail.

Axes:

An axis is the address of a node of the noun tree.

The notation for axis n is /n.

The first part of the axis tree looks like this:

```
         1
    2          3
 4    5     6     7
8 9 10 11 12 13 14 15
```

The head of axis /n is /(2n) and the tail of axis /n is /(2n+1).

Nock Operators:

The operators in Nock are functions mapped onto the eleven atoms `0` through `10`.

Nock 0:

The Nock operator that produces a given axis of a noun:

```
*[a [0 b]]              /b of a
```

___

3.2.2 Exercises:

1. Take pen and paper and map out the axes of

```
   [[[48 49] 45] [46 47]]
       /              \
 [[48 49] 45]       [46 47]
   /       \        /   \
[48 49]    45      46   47
 /   \
48   49
```

   then to test yourself, run:

```
~tomsyt-balsen/try=> .*([[[48 49] 45] [46 47]] [0 b])
```

   for each axis `/b` of `[[[48 49] 45] [46 47]]`

2. Write a noun `a` such that the following produces something:

```
~tomsyt-balsen/try=> .*(a [0 100])
```

   In other words, find a noun that has an axis `/100`.

3. Prune your noun from the last exercis so that its as short as possible, while still having an axis `/100`.

4. If you still feel confused, replicate the previous two exercises with the axes `/7`, ‘/17’, ‘/27’, ‘/47’ and `/87`.

5. Build a noun that has every atom set to its own axis. The bigger the noun the better. I’ll get you started:

```
1
[2 3]
[[4 5] 3]
[[4 5] [6 7]
```

To test different atoms in your noun, run

```
~tomsyt-balsen/try=> .*(a [0 b])
```

___

3.3 Section III: Nock 3

In the last section we learned how to access data inside nouns. Now we're going to learn what you can do with data. After all, Nock is a computer, so you should be able to, you know, compute things.

We mentioned in the last section that Nock has eleven operators, `0` through `10`. We saw `Nock 1` in Section I:

```
*[a [1 b]]                b
```

We could describe `Nock 1` the constant operator, since it always produces `b` no matter what you put in as a.

We also have `Nock 0`, which lets you refer to the sub-noun at the `axis b` of your subject `a`.

```
*[a [0 b]]              /b of a
```

We're going to skip `Nock 2` and cover it in Section V and for now just jump straight to `Nock 3`.

Before we do, let's quickly go over three new terms that'll help us talk about operators:

Let's say you have the Nock expression

```
*[a [1 b]]
```

We know that `1` is the operator, but shouldn't we have names for what `a` and `b` are?

Let's call `a` the subject, because it's being subjected to our computation. Poor `a`.

We'll call `b` an argument, And we'll call the cell `[1 b]` a formula. Diagramming it out:

```
*[a [1 b]]
*[subject [operator argument]]
*[subject formula]
```

Good, now that we've got vocab out of the way, let's get cooking.

Let's use the big noun `[[[48 49] 45] [46 47]]` from the last section as our subject.

```
~tomsyt-balsen/try=>.*([[[48 49] 45] [46 47]] [3 0])
! exit
```

Uh-oh, Arvo is telling us we just tried to do something indecent and unnatural.

```
~tomsyt-balsen/try=>.*([[[48 49] 45] [46 47]] [3 1])
! exit

~tomsyt-balsen/try=>.*([[[48 49] 45] [46 47]] [3 2])
! exit
```

```
~tomsyt-balsen/try=>.*([[[48 49] 45] [46 47]] [3 3])
! exit

~tomsyt-balsen/try=>.*([[[48 49] 45] [46 47]] [3 4])
! exit
```

Looks like `Nock 3` doesn't like an atomic argument. Let's go cellular.

```
~tomsyt-balsen/try=>.*([[[48 49] 45] [46 47]] [3 [0 1]])
0
```

Okay, so at least that does something.

Let's try some more:

```
~tomsyt-balsen/try=>.*([[[48 49] 45] [46 47]] [3 [0 2]])
0

~tomsyt-balsen/try=>.*([[[48 49] 45] [46 47]] [3 [0 3]])
0

~tomsyt-balsen/try=>.*([[[48 49] 45] [46 47]] [3 [0 4]])
0

~tomsyt-balsen/try=>.*([[[48 49] 45] [46 47]] [3 [0 5]])
1
```

Wait, what?

Hold on, aren't `[0 1]` or `[0 5]` valid Nock formulas on their own? If `3` only takes cells, and formulas are cells, maybe it matters what the formula does on its own.

Let's try that sequence again without `Nock 3`:

```
~tomsyt-balsen/try=>.*([[[48 49] 45] [46 47]] [0 1])
[[[48 49] 45] [46 47]]

~tomsyt-balsen/try=>.*([[[48 49] 45] [46 47]] [0 2])
```

```
[[48 49] 45]

~tomsyt-balsen/try=>.*([[[48 49] 45] [46 47]] [0 3])
[46 47]

~tomsyt-balsen/try=>.*([[[48 49] 45] [46 47]] [0 4])
[48 49]

~tomsyt-balsen/try=>.*([[[48 49] 45] [46 47]] [0 5])
45
```

One of these things is not like the others. Looks like something changes whether the formula `[0 n]` refers to an atom or a cell in our subject.

Let's see what happens if we run Nock 3 with `[0 1]` as its argument and try some different subjects:

```
~tomsyt-balsen/try=>.*([42 42] [3 [0 1]])
0

~tomsyt-balsen/try=>.*(42 [3 [0 1]])
1

~tomsyt-balsen/try=>.*([1 1] [3 [0 1]])
0

~tomsyt-balsen/try=>.*(1 [3 [0 1]])
1
```

Looks like that's `0` for cells and `1` for atoms. Which means that `Nock 3` is cell/atom tester.

How's that work?

`Nock 3`'s argument is itself a formula. Let's call it `b`.

```
*[subject-a [3 formula-b]]
```

`Nock 3` produces either a `0` or a `1`, depending on what formula-b does when applied to subject-a:

```
*[subject-a formula-b]
```

Or more simply:

```
*[a b]
```

If `*[a b]` produces a cell, then `*[a [3 b]]` produces 0. If `*[a b]` produces an atom, `*[a [3 b]]` produces 1. So our pattern for Nock 3 is:

```
*[a [3 b]]          ?(*[a b])
?(cell)             0
?(atom)             1
```

`?(x)` is just a litte functional notation so that we can write out both possible branches.

We should also note that Urbit uses the atom `0` to mean "yes" and the atom `1` to mean "no." (this may seem different and annoying. it is, but it's the same thing that Unix does).

In that context, `Nock 3` is just asking the question "Is the product of these two nouns a cell?

Let's review what we've learned:

3.3.1 Summary:

Vocabulary:

The subject is the noun that gets computed.

The operator is an atom that tells us how to Nock the subject and arguments.

The arguments modify the behaviour of the operator.

A formula is a cell of `[operator arguments]`

As a chart:

```
*[a [1 b]]
*[subject [operator arguments]]
*[subject formula]
```

Yes and No

Atom 0 means “yes” Atom 1 means “no.”

Nock 3

Nock 3 is a cell tester.

```
*[a [3 b]]          ?(*[a b])
?(cell)             0
?(atom)             1
```

___

3.3.2 Exercises:

1. Run and compare

   ```
   ~tomsyt-balsen/try=>.*(a b)
   ```

   and

```
~tomsyt-balsen/try=>.*(a [3 b])
```

with different values for a and b.

2. Try finding a value for `b` that will return `1` if `a` is the atom `42`:

```
~tomsyt-balsen/try=>.*(42 [3 b])
```

Not only is this possible, but you already know the formula to do it. ___

3.4 Section IV: Nock 4

Last section we learned how to test whether a noun is an atom or a cell with `Nock 3`. Now we're going to figure out what `Nock 4` does.

Let's start playing with `Nock 4`, using `[[44 45] 46]` as our subject:

```
~tomsyt-balsen/try=>.*([[44 45] 46] [4 0])
! exit

~tomsyt-balsen/try=>.*([[44 45] 46] [4 1])
! exit

~tomsyt-balsen/try=>.*([[44 45] 46] [4 2])
! exit
```

Okay, this is starting to feel a lot like `Nock 3`. Looks like `Nock 4` doesn't like atoms either. Remember that `Nock 3` took a cell that was a formula as its argument:

```
*[subject [3 formula]]
```

And then depending on what the expression

```
*[subject formula]
```

produced, `Nock 3` would return a `0` or `1`, according to the function `?(x)` that we defined:

```
*[a [3 b]]          ?(*[a b])
?(cell)             0
?(atom)             1
```

Let's assume that Nock 4 operates in a similar way. Let's use the formula `[0 n]` to try to apply Nock 4 to different axes in our subject:

```
~tomsyt-balsen/try=>.*([[44 45] 46] [4 [0 1]])
! exit

~tomsyt-balsen/try=>.*([[44 45] 46] [4 [0 2]])
! exit

~tomsyt-balsen/try=>.*([[44 45] 46] [4 [0 3]])
47
```

Well! That's blessedly simple then. Watch:

```
~tomsyt-balsen/try=>.*([[44 45] 46] [0 3])
46

~tomsyt-balsen/try=>.*([[44 45] 46] [4 [0 3]])
47
```

Can you guess what

```
~tomsyt-balsen/try=>.*([[44 45] 46] [4 [0 4]])
```

would produce?

If you said

```
~tomsyt-balsen/try=>.*([[44 45] 46] [4 [0 4]])
45
```

then you're starting to get the hang of this. Yes, ladies and gentlemen, `Nock 4` is increment. Nock together your subject and the formula in your argument, and whatever that produces, add 1 to it.

But what if `*[subject formula]` produces a cell instead of an atom? How do we add `1` to a cell? Simple, we don't. The sun continues to rise in the east, pigs remain regretfully earthbound, Hell is still rather toasty, and incrementing a cell in Nock produces an ! exit:

```
~tomsyt-balsen/try=>.*([[44 45] 46] [4 [0 2]])
! exit
```

Assuming we understood how `Nock 3` worked, writing dow our pattern for `Nock 4` is easy:

Nock 3:

```
*[a [3 b]]          ?(*[a b])
?(cell)             0
?(atom)             1
```

Nock 4:

```
*[a [4 b]]          +(*[a b])
+(cell)             ! exit
+(atom)             1 + atom
```

`+(x)` is, again, just some notation so we can write out both branches of Nock 4.

An interesting property of Nock 4 is that we can chain it together to increment successive times.

```
~tomsyt-balsen/try=>.*(44 [4 [0 1]])
45

~tomsyt-balsen/try=>.*(44 [4 [4 [0 4]]])
46

~tomsyt-balsen/try=>.*(44 [4 [4 [4 [0 4]]]])
47

~tomsyt-balsen/try=>.*(44 [4 [4 [4 [4 [0 4]]]]])
48

~tomsyt-balsen/try=>.*(44 [4 [4 [4 [4 [4 [0 4]]]]]])
49
```

Those brackets are starting to really pile up. Which is making this whole process a lot less legible than we would like.

Fortunatley Nock has a notational rule that'll let us not have to write so many of those brackets.

```
~tomsyt-balsen/try=>.*(44 [4 [4 [4 [4 [4 [0 4]]]]]])
49

~tomsyt-balsen/try=>.*(44 [4 4 4 4 4 0 4])
49
```

Woah. That's a lot cleaner.

Concisely, Nock considers brackets to group to the right. If you'll recall from back in section 1:

```
A noun is an atom or a cell.
An atom is a natural number.
A cell is an ordered pair of two nouns. I.e. two atoms, two cells, or a cell and an atom.
```

Which means that formally, all nouns in Nock are either singletons (atoms) or cells (pairs). There are no triples, quadruples, or n-tuples:

```
[a b c]
[a b c d]
[a b c d e]
```

etc.

But because a triple like [a b c] doesn't exist in Nock, but is easier on the eyes, we can map a nested pair onto it.

```
[a b c]         [a [b c]]
```

Which means that the Nock interpreter, whenever it sees a triple (or any n-tuple, for n>2), just inserts the needed brackets.

Let's do some more examples to help you get the hang of it:

```
~tomsyt-balsen/try=>.*(44 [4 [4 [4 [4 [4 0 1]]]]])
49

~tomsyt-balsen/try=>.*(44 [4 [4 [4 [4 4 0 1]]]])
49

~tomsyt-balsen/try=>.*(44 [4 [4 [4 4 4 0 1]]])
49

~tomsyt-balsen/try=>.*(44 [4 [4 4 4 4 0 1]])
49

~tomsyt-balsen/try=>.*(44 [4 4 4 4 4 0 1])
49
```

We can't get rid of the last pair though:

```
~tomsyt-balsen/try=>.*(44 4 4 4 4 4 0 1)
~ <syntax error at [1 18]>
```

This is just an artifact of Arvo's Nock interpreter, which we directly access with the `.*` function, pronounced `dottar` (dot-star, get it?). `.*` takes two arguments, a subject and a formula:

```
.*(subject [formula])
```

And for inscrutable reasons, the formula has to be in brackets. As does the subject, if it's a cell:

```
~tomsyt-balsen/try=>.*(44 45 [4 4 4 4 4 0 2])
~ <syntax error at [1 9]>

~tomsyt-balsen/try=>.*([44 45] [4 4 4 4 4 0 2])
49
```

Let's review:

3.4.1 Summary:

Nock 4:

```
*[a [4 b]]          +(*[a b])
+(cell)             ! exit
+(atom)             1 + atom
```

Brackets:

Brackets group to the right.

```
[a b c]         [a [b c]]
```

Nock interpreter:

`.*` is pronounced "dottar"

Takes two arguments, subject and formula:

```
.*(subject [formula])
```

formula must be bracketed.

3.4.2 Exercises:

1. Chain together Nock 4 and Nock 3, so that cells produce 2 and atoms produce 3.

2. Write a formula that always returns the cell [4 0 1].

3.5 Section V: Nock 5, Nock 2 and Formula Distribution

So we've learned how to do some simple operations with Nock. Now we're going to get a little fancier.

To jog your memory, we've seen the following operators so far:

```
~tomsyt-balsen/try=> .*([42 43] [0 1])
[42 43]

~tomsyt-balsen/try=> .*([42 43] [1 0])
[42 43]

~tomsyt-balsen/try=> .*([42 43] [3 0 1])
0

~tomsyt-balsen/try=> .*([42 43] [4 0 2])
43
```

These have all had the following pattern in common:

```
*[subject [operator arguments]]
```

with the atomic operator (0, 1, 3 or 4).

What if we replaced that atom with a cell? With a formula, even? Let's try it:

```
~tomsyt-balsen/try=> .*(42 [[0 1] 0 1])
[42 42]
```

Well that's different. Let's try another subject:

```
~tomsyt-balsen/try=> .*([42 43] [[0 1] 0 1])
[[42 43] 42 43]
```

And if we change the formulas?

```
~tomsyt-balsen/try=> .*([42 43] [[0 2] 0 1])
[42 42 43]

~tomsyt-balsen/try=> .*([42 43] [[0 2] 0 2])
[42 42]

~tomsyt-balsen/try=> .*([42 43] [[0 1] 0 2])
[[42 43] 42]
```

It looks like its running both formulas and then just combining the results in a cell.

```
~tomsyt-balsen/try=> .*([42 43] [[3 0 1] 3 0 2])
[0 1]

~tomsyt-balsen/try=> .*([42 43] [[3 0 1] 4 0 3])
[0 44]
```

```
~tomsyt-balsen/try=> .*([42 43] [[1 [0 1]] 4 0 3])
[[0 1] 44]

~tomsyt-balsen/try=> .*([42 43] [[4 0 3] 1 [0 1]])
[44 [0 1]]
```

Yup, the subject is definitely running through both formulas in parallel. The last example seems to do something like this:

```
*[[42 43] [4 0 3] 1 [55 73]]         [*[[42 43] [4 0 3]] *[[42 43] 1 [55 73]]]
```

which we can evaluate using Arvo:

```
~tomsyt-balsen/try=> .*([42 43] [4 0 3])
44

~tomsyt-balsen/try=> .*([42 43] [1 [0 1]])
[0 1]
```

Or by hand, which is good practice. Open up a blank text file or grab a pen and copy along:

```
*[[42 43] [4 0 3] 1 [0 1]]  [*[[42 43] [4 0 3]] *[[42 43] 1 [0 1]]]

<<  Nock 4:       *[a [4 b]]          +(*[a b]) >>

[+(*[[42 43] 0 3]]) *[[42 43] 1 [0 1]]]

<<  Nock 0:       *[a [0 b]]          /b of a  >>

[+(43) *[[42 43] 1 [0 1]]]

<<     +():       +(atom)             1 + atom  >>

[44 *[[42 43] 1 [0 1]]]

<<  Nock 1:       *[a [1 b]]          b  >>

[44 [0 1]]
```

Thus,

```
~tomsyt-balsen/try=> .*([42 43] [[4 0 3] 1 [0 1]])
[44 [0 1]]
```

We could write the first line of the reduction more generally as

```
*[subject [formula1] formula2]     [*[subject formula1] *[subject formula2]]
```

Which is a little long for my taste, since a formula is a cell we can rewrite it as:

```
*[a [b c] d e]     [*[a b c] *[a d e]]
```

where a is the subject, [b c] is formula1 and [d e] is formula2.

But actually we can simplify this further and just say:

```
*[a [b c] d]     [*[a b c] *[a d]]
```

since d has to be a cell for `*[a d]` to evaluate. We really just want to specify what happens when you replace an atomic operator with a cell.

What's really cool about this rule is that, like the operators, it also chains:

```
~tomsyt-balsen/try=> .*([42 [46 47]] [[0 1] [3 0 1] [0 2]])
[[42 46 47] 0 42]
```

So if we wanted to produce our subject with all the atoms incremented, we could do that:

```
~tomsyt-balsen/try=> .*([42 [46 47]] [[4 0 1] [4 0 2] [4 0 3]])
[43 47 48]
```

We can make our chains as long as we like

```
~tomsyt-balsen/try=> .*([42 [46 47]] [[0 1] [3 0 1] [0 2] [3 0 2] [0 3] [3 0 3]])
[[42 46 47] 0 42 1 [46 47] 0]
```

That is, we can evaluate arbitrary numbers of formulas on the same subject in parallel.

But what if we want to run them in series?

The expression `*[[42 43] [[4 0 3] 1 [0 1]]]` is a good example of how this might work:

```
~tomsyt-balsen/try=> .*([42 43] [[4 0 3] 1 [3 0 1]])
[44 [3 0 1]]
```

Wouldn't it be interesting if we could run [44 [3 0 1]] through Nock again and end up with `*[44 [3 0 1]]` or just `1`?

We'd need a recursive operator to do that. Fortunately, we've got one, Nock 2:

```
~tomsyt-balsen/try=> .*([42 43] [2 [4 0 3] 1 [3 0 1]])
1
```

Obviously this is a toy example because we could just do the same thing functionally with:

```
~tomsyt-balsen/try=> .*([42 43] [3 4 0 3])
1
```

But Nock 2 also lets us call a formula inside our subject.

```
~tomsyt-balsen/try=> .*([[40 43] [4 0 1]] [2 [0 4] [0 3]])
41

~tomsyt-balsen/try=> .*([[40 43] [4 0 1]] [2 [0 5] [0 3]])
44
```

Or we could completely separate the operator and arguments:

```
~tomsyt-balsen/try=> .*([[40 43] [0 1 3 4]] [2 [0 2] [0 31] [0 6] [0 30]])
44
```

We did a lot of slicing and dicing of nouns with the formula distribution rule. Nock 2 lets us run those reassembled nouns as expressions. We could think of Nock 2 as being exactly like the distribution rule:

```
*[a [b c] d]          [*[a b c] *[a d]]
```

except that Nock 2 has an extra `*` meaning we run everything through Nock a second time. So if we have two formulas

```
*[subject 2 formula1 formula2]              *[*[subject formula1] *[subject formula2]]
```

which we can rewrite as:

```
*[a 2 b c]            *[*[a b] *[a c]]
```

Let's work through that last example again:

```
~tomsyt-balsen/try=> .*([[40 43] [0 1 3 4]] [2 [0 2] [0 31] [0 6] [0 30]])
44
```

So we've got our subject [[40 43] [0 1 3 4]] and a four different formulas: [0 2] [0 31] [0 6] [0 30]

Let's apply each of these to our subject separately:

```
~tomsyt-balsen/try=> .*([[40 43] [0 1 3 4]] [0 2])
[40 43]

~tomsyt-balsen/try=> .*([[40 43] [0 1 3 4]] [0 31])
4

~tomsyt-balsen/try=> .*([[40 43] [0 1 3 4]] [0 6])
0

~tomsyt-balsen/try=> .*([[40 43] [0 1 3 4]] [0 30])
3
```

If instead of Nock 2, we had just used the formula distribution rule:

```
~tomsyt-balsen/try=> .*([[40 43] [0 1 3 4]] [[0 2] [0 31] [0 6] [0 30]])
[[40 43] 4 0 3]
```

But since Nock 2 is recursive:

```
*[[[40 43] [0 1 3 4]] [2 [0 2] [0 31] [0 6] [0 30]]]
```

reduces to:

```
*[[40 43] 4 0 3]
```

which is, of course, 43 + 1, or 44.

Now that we understand how to slice up nouns in our subject, let's introduce Nock 5.

Nock 5 is exactly like Nock 3 and Nock 4 in structure, but we've saved it for last because it's easier to understand how to use it, once you know how to distribute formulas. See if you can figure it out from the following:

```
~tomsyt-balsen/try=> .*([42 42] [5 [0 2] [0 3]])
0

~tomsyt-balsen/try=> .*([42 43] [5 [0 2] [0 3]])
1

~tomsyt-balsen/try=> .*([42 44] [5 [0 2] [0 3]])
1

~tomsyt-balsen/try=> .*([[42 42] [42 42]] [5 [0 2] [0 3]])
0

~tomsyt-balsen/try=> .*([[42 43] [42 43]] [5 [0 2] [0 3]])
0

~tomsyt-balsen/try=> .*([[42 43] [42 43]] [5 [0 4] [0 3]])
1

~tomsyt-balsen/try=> .*([[42 42] [42 42]] [5 [0 2] [0 3]])
0
```

Yes, Nock 5 is an equality test:

```
*[a 5 b]
```

if the head and the tail of the cell produced by `*[a b]` are the same, then Nock 5 produces 0, if they are different, Nock 5 produces 1:

```
*[a 5 b]              =(*[a b])
=([a a])              0
=([a !a])              1
```

Where `!a` just means “not a.”

But what if `*[a b]` inside `=(*[a b])` produces an atom?

```
~tomsyt-balsen/try=> .*([[42 42] [42 42]] [5 [1 1]])
! exit
```

So we need to add

```
=(atom)              ! exit
```

to our rule.

Let’s reduce the last example from above by hand:

```
*[[[42 42] [42 42]] [5 [0 2] [0 3]]]

    *[a 5 b]                =(*[a b])

=(*[[[42 42] [42 42]] [0 2] [0 3]])

    *[a [b c] d]            [*[a b c] *[a d]]

=([*[[[42 42] [42 42]] [0 2]] *[[[42 42] [42 42]] [0 3]]])

    *[a [0 b]]               /b of a

=([[42 42] *[[[42 42] [42 42]] [0 3]]])

    *[a [0 b]]               /b of a

=([[42 42] [42 42]])

    =([a a])                 0

1
```

———

3.5.1 Summary:

Formula Distribution:

A formula with a second formula at its head instead of an operator distributes the subject over both formulas:

```
*[a [b c] d]          [*[a b c] *[a d]]
```

`[b c]` is the first formula, d is the second formula.

Nock 2

```
*[subject 2 formula-a formula]              *[*[subject formula1] *[subject formula2]]
```

which translates to

```
*[a 2 b c]            *[*[a b] *[a c]]
```

Nock 5

```
*[a 5 b]              =(*[a b])
=([a a])              0
=([a !a])              1
=(atom)               ! exit
```

___

3.5.2 Exercises:

1. Using the above rule, write a formula that reverses the order of the atoms in `[42 46 [68 69] 55]` i.e. produces `[55 [68 69] 46 42]`.

2. Put the subject `[4 3 7 2 5 1 6]` in order from least to greatest.
3. Does `*[[42 42] 5 [0 1] [0 3]]` produce a `yes` or a `no`?
4. Write a noun that contains some data (nouns you find interesting) and some code (formulas you find interesting), write an expression with that noun as the subject that produces a single data-noun and a single code cell. Then use Nock 2 to apply the formula to the data.
5. Choose a subject such that the following expression evaluates

```
`~tomsyt-balsen/try=> .*(subject [2 [0 5] [0 4] [0 3]])`
`43`
```

3.6 Section VI: Conclusion

Let's list out all the rules we've learned so far, with the explanations collapsed:

```
A noun is an atom or a cell.
An atom is a natural number.
A cell is an ordered pair of nouns.


[a b c]              [a [b c]]

?(cell)               0
?(atom)               1
+(cell)               ! exit
+(atom)               1 + atom
=([a a])              0
=([a !a])              1
=(atom)               ! exit


*[a 0 b]              /b of a
*[a 1 b]              b
*[a 2 b c]            *[*[a b] *[a c]]
*[a 3 b]              ?(*[a b])
*[a 4 b]              +(*[a b])
*[a 5 b]              =(*[a b])
```

```
*[a [b c] d]          [*[a b c] *[a d]]
```

That looks cleaner. Now let's look at an abridged version of the real Nock spec, which you should now be able to mostly read

```
1  ::    A noun is an atom or a cell.
2  ::    An atom is a natural number.
3  ::    A cell is an ordered pair of nouns.
4  ::
5  ::    nock(a)           *a
6  ::    [a b c]           [a [b c]]
7  ::
8  ::    ?[a b]            0
9  ::    ?a                1
10 ::    +[a b]            +[a b]
11 ::    +a                1 + a
12 ::    =[a a]            0
13 ::    =[a b]            1
14 ::    =a                =a
15 ::
16 ::    /[1 a]            a
17 ::    /[2 a b]          a
18 ::    /[3 a b]          b
19 ::    /[(a + a) b]      /[2 /[a b]]
20 ::    /[(a + a + 1) b]  /[3 /[a b]]
21 ::    /a                /a
22 ::
23 ::    *[a [b c] d]      [*[a b c] *[a d]]
24 ::
25 ::    *[a 0 b]          /[b a]
26 ::    *[a 1 b]          b
27 ::    *[a 2 b c]        *[*[a b] *[a c]]
28 ::    *[a 3 b]          ?*[a b]
29 ::    *[a 4 b]          +*[a b]
30 ::    *[a 5 b]          =*[a b]
31 ::
...
38 ::
39 ::    *a                *a
```

Since the Nock spec has been designed to be as concise as possible, a couple things are notationally different from the rules we've learned.

The largest difference is the block that defines what the `/` or axis operator does:

```
16 ::    /[1 a]           a
17 ::    /[2 a b]         a
18 ::    /[3 a b]         b
19 ::    /[(a + a) b]     /[2 /[a b]]
20 ::    /[(a + a + 1) b] /[3 /[a b]]
```

This block of pseudocode is functionally equivalent to saying:

The head of axis /n is /(2n) and the tail of axis /n is /(2n+1).

If you read and understood Chapter 1, Section II, you understand what this is doing, even if you can't parse its recursive structure.

The next most obvious difference is that between

```
8  ::    ?[a b]           0
9  ::    ?a               1
10 ::    +[a b]           +[a b]
11 ::    +a               1 + a
12 ::    =[a a]           0
13 ::    =[a b]           1
14 ::    =a               =a
```

and

```
?(cell)              0
?(atom)              1
+(cell)              ! exit
+(atom)              1 + atom
=([a a])             0
=([a !a])            1
=(atom)              ! exit
```

We'll work through the evolution of this block:

First thing is that since we're trying to make the Nock specification small, we can get rid of the parentheses:

```
?cell               0
?atom               1
+cell               ! exit
+atom               1 + atom
=[a a]              0
=[a !a]             1
=atom               ! exit
```

Then we can remove the words ‘cell’ and ‘atom.’ Since the rules in the nock spec match top to bottom, we can specify matching a rule to a cells or atom by putting [a b] above a. A cell will match [a b] and and atom will not, therefore because all cells will match [a b], only atoms will match a below [a b].

```
?[a b]              0
?a                  1
+[a b]              ! exit
+a                  1 + atom
=[a a]              0
=[a !a]             1
=a                  ! exit
```

Next little thing we can do, along the same principle, is change !a to b, because [a b] below [a a] will only match to a pair of unequal nouns.

```
?[a b]              0
?a                  1
+[a b]              ! exit
+a                  1 + atom
=[a a]              0
=[a !a]             1
=a                  ! exit
```

Then there’s these lines, which we call our crash defaults:

```
10 ::    +[a b]           +[a b]
14 ::    =a               =a
21 ::    /a               /a
39 ::    *a               *a
```

Basically the crash defaults determine when Nock needs to return an ! exit, because something is nonsensical. In theory these lines imply that Nock spins forever in an infinite loop, in practice, Nock will just crash.

```
10 ::    +[a b]           +[a b]
```

means that Nock crashes if you try to increment a cell

```
14 ::    =a               =a
```

means that Nock crashes if you try run an equality test on an atom

```
   21 ::    /a               /a
```

means that Nock crashes if you try to reference a noun axis that doesn't exist

```
39 ::    *a               *a
```

means that if you try to run something that's not a valid formula (i.e. doesn't match any of the preceding lines 1 through 38) through Nock, you guessed it, Nock crashes.

Replacing ! exit in our rules with the appropriate crash default, we get the canoncial Nock specification:

```
1  ::    A noun is an atom or a cell.
2  ::    An atom is a natural number.
3  ::    A cell is an ordered pair of nouns.
4  ::
5  ::    nock(a)          *a
6  ::    [a b c]          [a [b c]]
7  ::
8  ::    ?[a b]           0
```

```
9  ::    ?a                1
10 ::    +[a b]            +[a b]
11 ::    +a                1 + a
12 ::    =[a a]            0
13 ::    =[a b]            1
14 ::    =a                =a
15 ::
16 ::    /[1 a]            a
17 ::    /[2 a b]          a
18 ::    /[3 a b]          b
19 ::    /[(a + a) b]      /[2 /[a b]]
20 ::    /[(a + a + 1) b]  /[3 /[a b]]
21 ::    /a                /a
22 ::
23 ::    *[a [b c] d]      [*[a b c] *[a d]]
24 ::
25 ::    *[a 0 b]          /[b a]
26 ::    *[a 1 b]          b
27 ::    *[a 2 b c]        *[*[a b] *[a c]]
28 ::    *[a 3 b]          ?*[a b]
29 ::    *[a 4 b]          +*[a b]
30 ::    *[a 5 b]          =*[a b]
31 ::
...
38 ::
39 ::    *a                *a
```

And that's it! That's really all there is to Nock. Everything else in Urbit, including the elided lines 32 through 38, is just a structure built on top of Nock. All your playing around in Urbit reduces to some combination of what you now already know.

Chapter 4

Nock Macros

> *You get used to it. I don't even see the code. All I see is blonde, brunette, redhead.*
>
> —**The Matrix**

In the last chapter we showed you the Nock specification with lines 32 through 37 removed. This chapter will explain what these lines are and what they do:

```
32 ::    *[a 6 b c d]     *[a 2 [0 1] 2 [1 c d] [1 0] 2 [1 2 3] [1 0] 4 4 b]
33 ::    *[a 7 b c]       *[a 2 b 1 c]
34 ::    *[a 8 b c]       *[a 7 [[7 [0 1] b] 0 1] c]
35 ::    *[a 9 b c]       *[a 7 c 2 [0 1] 0 b]
36 ::    *[a 10 [b c] d]  *[a 8 c 7 [0 3] d]
37 ::    *[a 10 b c]      *[a c]
```

These lines describe the Nock operators 6 through 10, which are just macros that reduce to operators 0 through 5. As you can see, these operators are defined in terms of Nock operators covered in the last chapter.

They add no new functionality to Nock, but we include them in the specification because they are essential to making Nock a practical basis for computation.

4.1 Nock 7

We'll start with the easiest macro: Nock 7

```
33 ::    *[a 7 b c]       *[a 2 b 1 c]
```

We actually already used Nock 7 in the last chapter while we were explaining Nock 2.

```
~tomsyt-balsen/try=> .*([42 43] [2 [4 0 3] 1 [3 0 1]])
1
```

Which sequentially applies the formulas `[4 0 3]` and `[3 0 1]` to our subject `[42 43]`. We use Nock 7 whenever we evaluate two formulas on our subject in series. The Nock 1 operator in `*[a 2 b 1 c]` is, as we know, the constant operator, which means that when Nock 2 applies, as in the above example, '[1 [3 0 1]]' to the subject, [3 0 1] is produced.

Let's work through a full reduction of the above example, refer to the specification at the end of the last chapter to get a better sense of where these rules are coming from. And as always, please copy along on pen and paper or in a text file.

```
*[[42 43] [2 [4 0 3] 1 [3 0 1]]]

      27 ::    *[a 2 b c]       *[*[a b] *[a c]]


*[*[[42 43] [4 0 3]] *[[42 43] 1 [3 0 1]]]

      26 ::    *[a 1 b]         b


*[*[[42 43] [4 0 3]] [3 0 1]]

      29 ::    *[a 4 b]         +*[a b]
```

```
*[+*[[42 43] 0 3] [3 0 1]]

      25 ::    *[a 0 b]          /[b a]

*[+/[3 [42 43]] [3 0 1]]

      18 ::    /[3 a b]          b

*[+43 [3 0 1]]

      11 ::    +a                1 + a

*[44 [3 0 1]]

      28 ::    *[a 3 b]          ?*[a b]

?*[44 0 1]

      25 ::    *[a 0 b]          /[b a]

?/[1 44]

      16 ::    /[1 a]            a

?44

      9  ::    ?a                1
1
```

This reduction is overly long because we've included the full reduction sequences for the Nock 0, 3 and 4 operators, which can usually be omitted without loss of clarity. The notation is often helpful, but must be balanced against the needs of conciseness.

Like so:

```
*[[42 43] [2 [4 0 3] 1 [3 0 1]]]

    27 ::    *[a 2 b c]        *[*[a b] *[a c]]


*[*[[42 43] [4 0 3]] *[[42 43] 1 [3 0 1]]]

    26 ::    *[a 1 b]          b

*[*[[42 43] [4 0 3]] [3 0 1]]
```

```
    29 ::    *[a 4 b]         +*[a b]

*[44 [3 0 1]]

    28 ::    *[a 3 b]         ?*[a b]
1
```

The student of Nock should be able to fill in the missing steps in their minds.

Informally, the formula `[7 b c]` composes the formulas `b` and `c`. To use a bit of math notation, if `d` is `[7 b c]`,

```
d(a) == c(b(a))
```

Let's see how this works by applying some reductions to the definition of `7`, and producing a simpler definition that doesn't look like a macro:

`7` Reduction:

```
29 ::    *[a 7 b c]        *[a 2 b 1 c]

*[a 2 b 1 c]

23 ::    *[a 2 b c]        *[*[a b] *[a c]]

*[*[a b] *[a 1 c]]

22:    *[a 1 b]          b

*[*[a b] c]
```

`7` Reduced:

```
7r ::     *[a 7 b c]         *[*[a b] c]
```

We'll skip the full reduction process for the rest of these macros and simply use the reduced forms to explain their function. Skip to the Nock reference guidechapter 4 if you want to see the full reduction sequence.

4.2 Nock 8

```
34 ::    *[a 8 b c]       *[a 7 [[7 [0 1] b] 0 1] c]

34r ::    *[a 8 b c]       *[[*[a b] a] c]
```

Nock 8 evaluates the formula `c` with the cell of `*[a b]` and the original subject `a`. In other words, in math notation, if `d` is `[8 b c]`,

```
d(a) == c([b(a) a])
```

But why? Suppose, for the purposes of `c`, we need not just `a`, but some intermediate noun computed from `a` that will be useful in `c`'s calculation. We apply `c` with a new subject that's a cell of the intermediate value and the old subject - not at all unlike pushing a new variable on the stack.

Let's work through some examples:

```
~tomsyt-balsen/try=> .*(42 [8 [4 0 1] [0 1]])
[43 42]

~tomsyt-balsen/try=> .*([42 45] [8 [[4 0 2] [4 0 3]] [0 1]])
[[43 46] 42 45]
```

For extra credit, a good question to ask yourself: In line 34,

```
   34 ::    *[a 8 b c]       *[a 7 [[7 [0 1] b] 0 1] c]
```

why do we need to write `[7 [0 1] b]` and not just `b`?

4.3 Nock 6

```
32 ::    *[a 6 b c d]     *[a 2 [0 1] 2 [1 c d] [1 0] 2 [1 2 3] [1 0] 4 4 b]

32r ::    *[a 6 b c d]     *[a *[[c d] [0 *[[2 3] [0 ++*[a b]]]]]]
```

Actually, `6` is a primitive known to every programmer - good old “if.” If `b` evaluates to `0`, we produce `c`; if `b` evaluates to `1`, we produce `d`; otherwise, we crash.

For instance:

```
~tomsyt-balsen/try=> .*(42 [6 [1 0] [4 0 1] [1 233]])
43
```

and

```
~tomsyt-balsen/try=> .*(42 [6 [1 1] [4 0 1] [1 233]])
233
```

We can actually simplify the semantics of `6`, at the expense of breaking the system a little, by creating a macro that works as “if” only if `b` is a proper boolean and produces `0` or `1`.

This simpler “if” would be:

```
32s::    *[a 6 b c d]    *[a 2 [0 1] 2 [1 c d] [1 0] [4 4 b]]
```

This reduces to

```
32sr::   *[a 6 b c d]     *[a *[[c d] [0 ++*[a b]]]]
```

Let's describe what each of `a`, `b`, `c` and `d` are.

`a` is our subject, some data that we want to run our `if` on

`b` is our test formula, which returns a `yes` or a `no` when we apply it to our subject.

`c` is our `then` formula, which we want to apply to our subject if our test formula produces a `yes`, i.e. a 0.

`d` is our `else` formula, which we want to apply to our subject if our test formula produces a `no`, i.e. a 1.

Let's say we have the following Nock expression:

```
~tomsyt-balsen/try=> .*(42 [6 [3 0 1] [4 0 2] [4 0 1]])
43
```

Our `test` is the formula [3 0 1], which tests if the subject is a cell.

Our `then` is the formula [4 0 2], which increments the head of a cell.

Our `else` is the formula [4 0 1], which increments an atom.

If we changed our subject to a cell:

```
~tomsyt-balsen/try=> .*([40 43] [6 [3 0 1] [4 0 2] [4 0 1]])
41
```

With our simpler reduced if rule:

```
32sr::   *[a 6 b c d]     *[a *[[c d] [0 ++*[a b]]]]
```

we could rewrite

```
*[42 [6 [3 0 1] [4 0 2] [4 0 1]]]
```

as

```
*[42  *[[[4 0 2] [4 0 1]] [0 ++*[42 [3 0 1]]]]]
```

Since `*[42 [3 0 1]]` produces a `no`, i.e a `1`:

```
*[42  *[[[4 0 2] [4 0 1]] [0 ++1]]]]
```

which gets incremented twice

```
*[42  *[[[4 0 2] [4 0 1]] [0 3]]]
```

and goes into a Nock 0 to select the tail of `[[4 0 2] [4 0 1]]`

```
*[42 [4 0 1]]
```

which increments 42 to produce 43. You should be able to see how changing the subject to the cell `[40 43]` would cause a `0` to be produced by the test, and how that would cascade into the `then` formula `[4 0 2]` being selected and applied instead.

Our real `if` is only slightly more complicated:

```
32r ::    *[a 6 b c d]     *[a *[[c d] [0 *[[2 3] [0 ++*[a b]]]]]]
```

There appears to be an extra step here, using Nock 0 twice, first to select from the cell [2 3], and then to select from the cell [c d].

The reason is fairly simple, if we just used the simpler version of `if`, tests that returned values other than 0 or 1 would have unexpected behaviour. If our our test produced a `3` we would then try to reference the axis /5 in our cell [c d]. Since the tail of `d` could very well be a valid formula, strange things could result.

We add the step of selecting from [2 3] because trying to reference anything other than /2 or /3 within [2 3] will crash, which is exactly what we want. (/1 of [2 3] won't crash, but this is fine since our test can never produce a -1)

4.4 Nock 9

```
9r ::     *[a 9 b c]        *[*[a c] *[*[a c] 0 b]]
```

We'll discuss what Nock 9 does in the next chapter, when we introduce how to use *cores*, which are subjects containing both code and data. If you have a really fine instinctive sense of Nock, you might understand what `9` is for.

Succinctly, we use Nock 9 to call a formulas held inside of the subject itself, and apply them to the subject. If you've been wondering how one writes Nock expressions that loop, this is how.

4.5 Nock 10

```
36 ::     *[a 10 [b c] d]  *[a 8 c 7 [0 3] d]
37 ::     *[a 10 b c]      *[a c]
```

The second case of 10 is so easy it's puzzling:

```
37 ::    *[a 10 b c]      *[a c]
```

For any `b`, the formula `[10 b c]` seems to be perfectly equivalent to the formula `c`. But why? Why would we say `[10 b c]` when we could just say `c`?

The answer is that `10` is a hint to the interpreter. It's true that `[10 b c]` has to be *semantically* equivalent to `c`, but it doesn't have to be *practically* equivalent. Since whatever information is in `b` is discarded, a practical interpreter is free to ignore it, or to use it in any way that does not affect the results of the computation.

And the other reduction of `10`:

```
   36 ::    *[a 10 [b c] d]  *[a 8 c 7 [0 3] d]
```

Reducing:

```
*[a 8 c 7 [0 3] d]

  <<34r::    *[a 8 b c]       *[[*[a b] a] c]>>

*[[*[a c] a] [7 [0 3] d]]

  <<33r::    *[a 7 b c]       *[*[a b] c]>>

*[*[[*[a c] a] 0 3] d]
```

If you've assimilated a bit of Nock already, you may feel the temptation to reduce this to

```
*[a d]
```

since it would be very reasonable to think that

```
*[[*[a c] a] 0 3]
```

is just `a`. And it seems to be - given the semantics of 8 as we've explained them.

But there's a problem, which is that `c` might not terminate. If `c` terminates, this reduction is correct. Otherwise it's not. So the best we can do is:

```
36r::    *[a 10 [b c] d]  *[*[[*[a c] a] 0 3] d]
```

And that's it! That's the entirety of the Nock specification!

```
1  ::    A noun is an atom or a cell.
2  ::    An atom is a natural number.
3  ::    A cell is an ordered pair of nouns.
4  ::
5  ::    nock(a)          *a
6  ::    [a b c]          [a [b c]]
7  ::
8  ::    ?[a b]           0
9  ::    ?a               1
10 ::    +[a b]           +[a b]
11 ::    +a               1 + a
12 ::    =[a a]           0
13 ::    =[a b]           1
14 ::    =a               =a
15 ::
16 ::    /[1 a]           a
17 ::    /[2 a b]         a
18 ::    /[3 a b]         b
19 ::    /[(a + a) b]     /[2 /[a b]]
20 ::    /[(a + a + 1) b] /[3 /[a b]]
21 ::    /a               /a
22 ::
23 ::    *[a [b c] d]     [*[a b c] *[a d]]
24 ::
25 ::    *[a 0 b]         /[b a]
26 ::    *[a 1 b]         b
27 ::    *[a 2 b c]       *[*[a b] *[a c]]
28 ::    *[a 3 b]         ?*[a b]
29 ::    *[a 4 b]         +*[a b]
30 ::    *[a 5 b]         =*[a b]
31 ::
```

```
32 ::    *[a 6 b c d]     *[a 2 [0 1] 2 [1 c d] [1 0] 2 [1 2 3] [1 0] 4 4 b]
33 ::    *[a 7 b c]       *[a 2 b 1 c]
34 ::    *[a 8 b c]       *[a 7 [[7 [0 1] b] 0 1] c]
35 ::    *[a 9 b c]       *[a 7 c 2 [0 1] 0 b]
36 ::    *[a 10 [b c] d]  *[a 8 c 7 [0 3] d]
37 ::    *[a 10 b c]      *[a c]
38 ::
39 ::    *a               *a
```

Chapter 5

Cores and Gates

But are you crazy enough?

—**Point Break**

5.1 Playing with Nock

Now we're going to actually do some cool stuff with Nock.

Fortunately, we have an entire OS, Arvo, which is built on Nock. Unfortunately, there's really no practical reason to work directly in Nock when you're using Arvo - except for learning Nock, which you do once and never again. So the things we'll have to do are a little bit cumbersome.

What Arvo is good at is evaluating Hoon. And it's possible to evaluate Nock from Hoon, much the way you can put inline assembly in C. Through this indirection, we have two ways to run Nock in Hoon: on the command line and via an app file.

5.1.1 Command line

From the Arvo command line, you can run one-liners with the Hoon rune `.*`:

```
~tomsyt-balsen/try=> .*(42 [4 0 1])
43
```

5.1.2 Application file

Unfortunately, the Arvo command line doesn't do multiline input well, and if there's any hope of writing complex Nock formulas it's by using plenty of whitespace and linebreaks.

So we've constructed a template for writing Nock formulas as Arvo applications. Unfortunately Arvo is a young OS and has no way to edit a file. But Arvo runs on Unix and Unix is a very old OS. Arvo syncs its filesystem with your `$URBIT_HOME` directory, propagating changes on either side.

Let's assume your `$URBIT_HOME` is `urb/`, and your ship is `~tomsyt-balsen`. The Nock application template is in

```
urb/tomsyt-balsen/try/bin/nock.hoon
```

Its text should be:

```
!:              ::  To write Nock as an Arvo application in Hoon
|=  *           ::
|=  [a=* ~]     ::  For educational purposes only
:_  ~  :_  ~    ::
:-  %la         ::  Preserve this mysterious boilerplate square
%+  sell  %noun::
.*  a           ::  Replace the formula with your own
::::::::::::::::
                ::  Formula: increment
[4 0 1]
```

For the rest of this document we'll simply assume you can copy boilerplate, and write the rest of the file:

```
[4 0 1]                                             ::    bump /1
```

(The pseudocode in the comments is not in any way described. If you have trouble figuring it out, that's okay, because it should make itself obvious by the end of the document.)

Test this by running:

```
~tomsyt-balsen/try=> :nock 42
43
```

Our first complex example will be a decrement function. With or without `vere` running, copy the template from Unix:

```
$ cp urb/tomsyt-balsen/try/bin/nock.hoon urb/tomsyt-balsen/try/bin/dec.hoon
```

Then, use a Unix editor to change "Formula: increment" to "Formula: decrement" in `dec.hoon`.

Either next time you start `vere`, or on your next keyboard event if you're already running it, you'll see something like

```
 + /~tomsyt-balsen/try/1/bin/dec/hoon
```

Arvo has slurped up dec.hoon from your filesystem. To test it,

```
~tomsyt-balsen/try=> :dec 42
43
```

Well, we didn't change the formula, so it still increments. But it's a start.

5.2 Decrement

The only arithmetic operation in Nock is increment. So how do we decrement? The algorithm is obvious: to decrement `n`, start from `0`, and count up to `n-1`. Or rather, count up to a number `m` such that `m+1` equals `n`.

(Is this going to be an `O(n)` algorithm? It is. How do we compute effectively in a platform where decrement is `O(n)`? Gosh, it seems difficult, doesn't it? We'll get to that.)

The first thing we're going to need is a counter. Right now our subject is just the atom we're trying to decrement - `/1`, referenced with the formula `[0 1]`. Thus, to increment it, the formula is `[4 0 1]`.

Let's try to put the counter into the subject with one of our macros operators, `8`. Recall our revised rule for `8`:

```
34r::    *[a 8 b c]       *[[*[a b] a] c]
```

The formula `c` is applied to the subject `[*[a b] a]`. What is our `b`? It should just produce our initial counter value - 0. So, use operator `1` to produce a constant - `[1 0]`. Let's put this counter in the subject, and then increment as usual.

Edit `dec.hoon` so that the formula reads

```
[ 8                                                 ::  push
  [1 0]                                             ::  just 0
  [4 0 1]                                           ::  bump /1
]                                                   ::
```

Note that for these tall bracket structures, the space after `[` is essential. Then, you'll see the file automatically update in Arvo:

```
: /~tomsyt-balsen/try/2/bin/dec/hoon
~tomsyt-balsen/try=> :dec 42
```

Whoops! It crashed:

```
! /~tomsyt-balsen/try/~2013.11.26..00.01.38..499b/bin/dec/:<[4 1].[13 2]>
! /~tomsyt-balsen/try/~2013.11.26..00.01.38..499b/bin/dec/:<[4 8].[13 2]>
! /~tomsyt-balsen/try/~2013.11.26..00.01.38..499b/bin/dec/:<[5 1].[13 2]>
! /~tomsyt-balsen/try/~2013.11.26..00.01.38..499b/bin/dec/:<[6 1].[13 2]>
! /~tomsyt-balsen/try/~2013.11.26..00.01.38..499b/bin/dec/:<[7 1].[13 2]>
! exit
```

What did we do wrong? We forgot that the subject had changed. When we get to `[4 0 1]`, the subject is not `42`, but `[0 42]` - the counter is there. So our original argument, `42`, is actually at `/3`:

```
[ 8                                                 ::  push
  [1 0]                                             ::  just 0
  [4 0 3]                                           ::  bump /3
]                                                   ::

: /~tomsyt-balsen/try/3/bin/dec/hoon
~tomsyt-balsen/try=> :dec 42
43
```

Okay, at least it increments again. (Constantly readjusting tree addresses by hand is one good reason to use a higher-level language, like Hoon.) But now, perhaps, we can build a decrement that works for at least one input value - `1`.

Obviously at some point we'll have to build a loop. But for now, all we need is an `if` statement that compares the incremented counter to the original argument. We know the original argument is at `/3`, and the counter is at `/2`; we use the if operator, `6`, and the equality test operator `5`. If the comparison fails, we shrug our shoulders and keep incrementing the argument.

```
[ 8                                                 ::  push
  [1 0]                                             ::  just 0
  [ 6                                               ::  pick
    [5 [4 0 2] [0 3]]                               ::  same (bump /2) /3
    [0 2]                                           ::  /2
    [4 0 3]                                         ::  bump /3
  ]                                                 ::
]                                                   ::

: /~tomsyt-balsen/try/4/bin/dec/hoon
~tomsyt-balsen/try=> :dec 42
43
~tomsyt-balsen/try=> :dec 1
0
```

We're getting closer. But now, that loop:

Up till now, our subject has contained only data. If we want to loop, we're obviously going to have to bite the bullet and put code in our subject - which will become a `[code data]` cell. In Nock (and Hoon) this is called a `core`.

Suppose we take our `6` formula and put it in the subject. Then, with this core subject `[formula counter argument]`, we'll run the formula itself. With this subject, the formula is `/2`, and of course the core itself is `/1`. So we can activate the core with `[2 [0 1] [0 2]]`.

Of course, since the subject has changed again, we need to change the addresses again. The counter is now `/6` and the argument is now `/7`:

```
[ 8                                                 ::  push
  [1 0]                                             ::  just 0
  [ 8                                               ::  push
    [ 1                                             ::  quid
      [ 6                                           ::  pick
        [5 [4 0 6] [0 7]]                           ::  same (bump /6) /7
        [0 6]                                       ::  /6
        [4 0 7]                                     ::  bump /7
      ]                                             ::
    ]                                               ::
    [2 [0 1] [0 2]]                                 ::  nock /1 /2
  ]                                                 ::
]                                                   ::
```

This does exactly the same thing as before:

```
: /~tomsyt-balsen/try/5/bin/dec/hoon
~tomsyt-balsen/try=> :dec 42
43
~tomsyt-balsen/try=> :dec 1
0
```

But somehow, we feel it *could* do better. Why? Because where we do the useless `[4 0 7]`, we have a subject containing the code we want to invoke. It's just that the counter is wrong.

We need to do the same thing as `[2 [0 1] [0 2]]`, but the subject is not `[0 1]`. That would be `[formula counter argument]`. We need `[formula (counter + 1) argument]`.

So, `formula` is `[0 2]`, `counter` is `[0 6]`, and `argument` is `[0 7]`. With autocons, we can just put them together to make a (superfluous) formula for `[formula counter argument]` - ie,

```
[[0 2] [0 6] [0 7]]               ::  cons /2 /6 /7
```

But we actually want to increment the counter:

```
[[0 2] [4 0 6] [0 7]]             ::  cons /2 (bump /6) /7
```

And to invoke our formula on this modified core:

```
[2 [[0 2] [4 0 6] [0 7]] [0 2]]   ::  nock (cons /2 (bump /6) /7) /2
```

If we put this into the decrement, it should actually work:

```
[ 8                                                 ::  push
  [1 0]                                             ::  just 0
  [ 8                                               ::  push
    [ 1                                             ::  quid
      [ 6                                           ::  pick
        [5 [4 0 6] [0 7]]                           ::  same (bump /6) /7
        [0 6]                                       ::  /6
        [ 2                                         ::  nock
           [[0 2] [4 0 6] [0 7]]                    ::  (cons /2 (bump /6) /7)
           [0 2]                                    ::  /2
        ]                                           ::
      ]                                             ::
    ]                                               ::
    [2 [0 1] [0 2]]                                 ::  nock /1 /2
  ]                                                 ::
]                                                   ::
```

And it does:

```
: /~tomsyt-balsen/try/6/bin/dec/hoon
~tomsyt-balsen/try=> :dec 42
41
```

But there's one more step. Remember operator `9`?

```
35 ::    *[a 9 b c]       *[a 7 c 2 [0 1] 0 b]
35r::    *[a 9 b c]       *[*[a c] *[*[a c] 0 b]]
```

Suppose `c` is a formula that produces a core. Then we see immediately what `9` does: it activates a core, using the formula at `/b` within the core.

So we can rewrite our decrement to use `9`:

```
[ 8                                                 ::  push
  [1 0]                                             ::  just 0
  [ 8                                               ::  push
    [ 1                                             ::  quid
      [ 6                                           ::  pick
        [5 [4 0 6] [0 7]]                           ::  same (bump /6) /7
        [0 6]                                       ::  /6
```

```
        [9 2 [0 2] [4 0 6] [0 7]]          ::  call.2 (cons /2 bump /6 /7)
      ]                                    ::
    ]                                      ::
    [9 2 0 1]                              ::  call.2 /1
  ]                                        ::
]                                          ::
```

Seems to work nicely:

```
: /~tomsyt-balsen/try/6/bin/dec/hoon
~tomsyt-balsen/try=> :dec 42
41
```

Of course, there are limits:

```
~tomsyt-balsen/try=> :dec 0
```

You'll have to hit `^C`, and you'll see a big ugly error stack. Nock can work wonders but it can't decrement 0. (Yes, you can build signed integers in Hoon - they are represented as atoms with the sign bit low.)

5.2.1 A function

As we start to build up toward language-level primitives, it behooves us to do things the way a higher-level language would do them. Well, more exactly, the way Hoon does things.

Surprisingly, although a formula defines a function of the subject, a function - at the language level - is not the same thing as a formula. Or rather, the argument is not the same thing as the subject.

For instance, as we saw in decrement, the subject for the loop needs to contain the code itself. If we apply a formula which can't call back into itself, our ability to loop is sorely diminished. So at the very least, when we

call a function, the subject can't just be `argument` - it has to be the cell `[formula argument]`, so that the function can recurse.

Actually, it's confusing to say `argument`, because this implies a special status for single and multiple arguments. In Nock and Hoon, we say `sample`, which is always one thing, but can be a cell for "functions of two arguments", a triple for three, etc. Eg, the sample for a decrement function is an atom; the sample for an add function is a cell of two atoms; etc.

Furthermore, a function needs more data than just the argument - it might, for instance, want to call other functions. Where's it going to get them? There is no external environment in Nock.

So the standard convention for a Nock function - or a Hoon function - is

```
[formula sample context]
```

Where `formula` is the code, `sample` is the argument(s), and `context` is any other data and/or code that may be useful.

It's a bit irregular that we are taking the external subject and using it directly from our formula. Let's try to build a function with this convention and call it directly.

First, we'll build an increment function to keep things simple. We actually don't need anything in the context, so we'll put 0.

```
[ 8                                          ::  push
  [                                          ::  cons
    [1 [4 0 6]]                              ::  quid bump /6  ::  formula
    [1 0]                                    ::  just 0        ::  sample
    [1 0]                                    ::  just 0        ::  context
  ]                                          ::
  [ 9                                        ::  call
    2                                        ::  .2
    [0 4] [0 3] [0 11]                       ::  cons /4 /3 /11
  ]                                          ::
]                                            ::
```

Why `[[0 4] [0 3] [0 11]]`? Our goal in calling the function is to take the blank default core we've created at `/2`, and substitute in the original subject of the outer formula, which before the outer `8` was `/1` and is now `/3`. Around this we wrap the formula from the default core, at `/4`, and the (dummy) context, at `/11` - that is, `/7` within `/2`.

Let's fit our decrement into this framework:

```
[ 8                                             ::  push
  [                                             ::  cons
    [ 1                                         ::  quid    ::  formula
      [ 8                                       ::  push
        [1 0]                                   ::  just 0
        [ 8                                     ::  push
          [ 1                                   ::  quid
            [ 6                                 ::  pick
              [5 [4 0 6] [0 30]]                ::  same /6 /30
              [0 6]                             ::  /6
              [9 2 [0 2] [4 0 6] [0 7]]         ::  call.2 /2 (bump /6) /11
            ]                                   ::
          ]                                     ::
          [9 2 0 1]                             ::  call.2 /1
        ]                                       ::
      ]                                         ::
    ]                                           ::
    [1 0]                                       ::  just 0  ::  sample
    [1 0]                                       ::  just 0  ::  context
  ]                                             ::
  [9 2 [0 4] [0 3] [0 11]]                      ::  call.2 /4 /3 /11
]                                               ::
```

Observe that nothing has changed from the way we called our increment function, and only one thing has changed within the decrement formula - the axis of the argument. Now at `/7` is not the naked argument to decrement, but our outer core. The sample is at `/6` within this `/7`, ie, at `/30`.

5.3 A library

Frankly, this is getting close to the limits of anything you'd want to do in hand-generated Nock. But why not press on?

What we'd really like to do is build a library of functions that can call each other. It's easy to guess that this library will be a core. But what does this core look like?

A function core, `[formula sample context]`, is a very useful kind of core, but it's not the only kind of core. (Actually, because the word "function" is too easy to throw around, we have a special name for a function core: we call it a `gate`. Compare to "lambda" or "closure.")

But in general, a core is just `[code data]` - or, to use more lingo, `[battery payload]`. The payload can be anything - it's just data.

The battery can be one *or more* formulas, each of which is applied with the core as its subject. This is why `9` takes the axis operand `b`. If the core is a gate, the battery is just one formula; this is the head of the core, so `b` is 2.

But not every core is a gate. Suppose we want to build a library? We could assemble a bundle of cores and put it in the context. So, let's say we need to write subtract, which obviously is going to use decrement. So, the context will be

```
[subtract-gate decrement-gate]
```

But wait. Each gate is `[formula sample context]`. So, because Nock doesn't do cycles, there's no way the subtract gate and the decrement gate can each reference each other through the context. It happens to be the case here that subtract needs decrement, but decrement doesn't need subtract. But we're not looking for ugly at this point - we know Nock is more than capable of that.

To support general mutual recursion, our library needs to be a battery in which each formula produces a gate. The context of that gate is the library core.

Let's repeat this again because it's so important. Our library will be a battery in which each formula produces a gate. The context of that gate is the library core.

Let's build a trivial library core of this form, with one function, good old increment. Then, we'll call it.

```
[ 8                                          ::  push
  [                                          ::  cons
    [ 1                                      ::  quid          ::  battery
      [1 [4 0 6]]                            ::  quid bump /6
      [1 0]                                  ::  just 0
      [0 1]                                  ::  /1
    ]                                        ::
    [1 0]                                    ::  just 0        ::  payload
  ]                                          ::
  [ 8                                        ::  push
    [9 2 0 2]                                ::  call.2 /2
    [9 2 [0 4] [0 7] [0 11]]                 ::  call.2 /4 /7 /11
  ]                                          ::
]                                            ::
```

Compare this to the standalone increment above. It's obviously more complex and it should be.

First of all, what we put in the library core is not the function gate directly, but a formula that generates the gate. This way, and only this way, we can put the library itself in the context.

Second, what's the payload of the library core? It's `0`, because the library doesn't depend on anything. It certainly doesn't depend on the argument to our application.

Third, now we can't just call the gate directly. We have to actually build it. So we need another `8` to "push it on the stack", and then we call it with the usual `9`. Since the subject at this point is `[gate library argument]`, the sample we use is `[0 7]` rather than `[0 3]` - everything else is the same.

But does it work? C'mon, you know it works:

```
~tomsyt-balsen/try=> :dec 42
43
```

Okay, let's go ahead and put our actual decrement function in the library.

```
[ 8                                             ::  push
  [                                             ::  cons
    [ 1                                         ::  quid     :: formula
      [ 1                                       ::  quid     :: decrement
        [ 8                                     ::  push
          [1 0]                                 ::  just 0
          [ 8                                   ::  push dec
            [ 1                                 ::  quid
              [ 6                               ::  pick
                [5 [4 0 6] [0 30]]              ::  same (bump /6) /30
                [0 6]                           ::  /6
                [9 2 [0 2] [4 0 6] [0 7]]       ::  call.2 /2 (bump /6) /7
              ]                                 ::
            ]                                   ::
            [9 2 0 1]                           ::  call .2 /1
          ]                                     ::
        ]                                       ::
      ]                                         ::
      [1 0]                                     ::  just 0   :: sample
      [0 1]                                     ::  /1       :: context
    ]                                           ::
    [1 0]                                       ::  just 0   :: payload
  ]                                             ::
  [ 8                                           ::  push
    [9 2 0 2]                                   ::  call.2 /2
    [ 9                                         ::  call
      2                                         ::  .2
      [0 4] [0 7] [0 11]                        ::  /4 /7 /11
    ]                                           ::
  ]                                             ::
]                                               ::

~tomsyt-balsen/try=> :dec 42
43
```

Then, let's go crazy and add a subtract function, which calls decrement.

```
[ 8                                             ::  push
  [                                             ::  cons
    [                                           ::  cons
      [ 1                                       ::  quid       :: subtract
        [ 1                                     ::  quid       :: formula
          [ 8                                   ::  push
            [9 5 0 7]                           ::  call.5 /7
            [ 6                                 ::  pick
```

```
              [5 [1 0] [0 29]]                ::  same just 0 /29
              [0 28]                          ::  /29
              [ 9                             ::  call
                2                             ::  .2
                [0 6]                         ::  /6
                [ [9 2 [0 4] [0 28] [0 15]]   ::  call .2 /4 /28 /15
                  [9 2 [0 4] [0 29] [0 15]]   ::  call .2 /4 /29 /15
                ]                             ::
                [0 15]                        ::  /15
              ]                               ::
            ]                                 ::
          ]                                   ::
        ]                                     ::
        [1 0]                                 ::  just 0    ::  sample
        [0 1]                                 ::  /1        ::  context
      ]                                       ::
      [ 1                                     ::  quid      :: decrement
        [ 1                                   ::  quid      :: formula
          [ 8                                 ::  push
            [1 0]                             ::  just 0
            [ 8                               ::  push
              [ 1                             ::  quid
                [ 6                           ::  pick
                  [5 [4 0 6] [0 30]]          ::  same bump /6 /30
                  [0 6]                       ::  /6
                  [9 2 [0 2] [4 0 6] [0 7]]   ::  call.2 /2 (bump /6) /7
                ]                             ::
              ]                               ::
              [9 2 0 1]                       ::  call.2 /1
            ]                                 ::
          ]                                   ::
        ]                                     ::
        [1 0]                                 ::  just 0
        [0 1]                                 ::  /1
      ]                                       ::
    ]                                         ::
    [1 0]                                     ::  just 0    :: payload
  ]                                           ::
  [ 8                                         ::  push
    [9 4 0 2]                                 ::  call.4 /2
    [ 9                                       ::  call
      2                                       ::  .2
      [0 4] [0 7] [0 11]                      ::  /4 /7 /11
    ]                                         ::
  ]                                           ::
]
```

Note that the call to build the gate is `[9 4 0 2]`, because the subtract arm is the head of the battery, which is the head of the core - ie, `/2` within `/2` -

ie, `/4`.

Does this work? Really?

```
~tomsyt-balsen/try=> :dec [42 12]
30
```

##Exercises##

Do you actually know Nock now? Well, possibly.

A good exercise is to add more simple math functions to this battery. Try add, multiply, and divide. One way to start is by walking through the routines above and figuring out what they're doing.

Computing axes is slightly arduous (which is why we use Hoon, generally). We are torturing ourselves by using Nock, but we might as well use Hoon to calculate axes:

```
    ~tomsyt-balsen/try=> (peg 3 3)
    7
    ~tomsyt-balsen/try=> (peg 3 5)
    13
```

Ie, `(peg a b)` is `/b` within `/a`. Writing Nock without this would be pretty tough.

Chapter 6

Nock Reference

> *Tlön será un laberinto, pero es un laberinto urdido por hombres, un laberinto destinado a que lo descifren los hombres.*
>
> *Tlön is surely a labyrinth, but it is a labyrinth devised by men, a labyrinth destined to be deciphered by men.*
>
> —***Tlön, Uqbar, Orbis Tertius***, Jorge Luis Borges

This chapter is designed for the experienced programmer and can be read without reference to the previous tutorial chapters.

It's important to recognize that the pseudocode of the Nock spec is just that: pseudocode. It looks a little like Hoon. It isn't Hoon - it's just pseudocode. Or in other words, just English. At the bottom of every formal system is a system of axioms, which can only be written in English. (Why pseudocode, not Hoon? Since Hoon is defined in Nock, this would only give a false impression of nonexistent precision.)

The logic of this pseudocode is a pattern-matching reduction, matching from the top down. To compute Nock, repeatedly reduce with the first line that matches.

6.1 Nock Specification:

```
1  ::    A noun is an atom or a cell.
2  ::    An atom is a natural number.
3  ::    A cell is an ordered pair of nouns.
4  ::
5  ::    nock(a)           *a
6  ::    [a b c]           [a [b c]]
7  ::
8  ::    ?[a b]            0
9  ::    ?a                1
10 ::    +[a b]            +[a b]
11 ::    +a                1 + a
12 ::    =[a a]            0
13 ::    =[a b]            1
14 ::    =a                =a
15 ::
16 ::    /[1 a]            a
17 ::    /[2 a b]          a
18 ::    /[3 a b]          b
19 ::    /[(a + a) b]      /[2 /[a b]]
20 ::    /[(a + a + 1) b]  /[3 /[a b]]
21 ::    /a                /a
22 ::
23 ::    *[a [b c] d]      [*[a b c] *[a d]]
24 ::
25 ::    *[a 0 b]          /[b a]
26 ::    *[a 1 b]          b
27 ::    *[a 2 b c]        *[*[a b] *[a c]]
28 ::    *[a 3 b]          ?*[a b]
29 ::    *[a 4 b]          +*[a b]
30 ::    *[a 5 b]          =*[a b]
31 ::
32 ::    *[a 6 b c d]      *[a 2 [0 1] 2 [1 c d] [1 0] 2 [1 2 3] [1 0] 4 4 b]
33 ::    *[a 7 b c]        *[a 2 b 1 c]
34 ::    *[a 8 b c]        *[a 7 [[7 [0 1] b] 0 1] c]
35 ::    *[a 9 b c]        *[a 7 c 2 [0 1] 0 b]
36 ::    *[a 10 [b c] d]   *[a 8 c 7 [0 3] d]
37 ::    *[a 10 b c]       *[a c]
38 ::
39 ::    *a                *a
```

6.2 1. Structures

```
A noun is an atom or a cell.  An atom is any natural number.
A cell is any ordered pair of nouns.
```

Nouns are the dumbest data model ever. Nouns make JSON look like XML and XML look like ASN.1. It may also remind you of Lisp's S-expressions - you can think of nouns as "S-expressions without the S."

To be exact, a noun *is* an S-expression, except that classic S-expressions have multiple atom types ("S" is for "symbol"). Since Nock is designed to be used with a higher-level type system (such as Hoon's), it does not need low-level types. An atom is just an unsigned integer of any size.

For instance, it's common to represent strings (or even whole text files) as atoms, arranging them LSB first - so "foo" becomes `0x6f6f66`. How do we know to print this as "foo", not `0x6f6f66`? We need external information - such as a Hoon type. Similarly, other common atomic types - signed integers, floating point, etc - are all straightforward to map into atoms.

It's also important to note that, unlike Lisp, Nock cannot create cyclical data structures. It is normal and common for nouns in a Nock runtime system to have acyclic structure - shared subtrees. But there is no Nock computation that can make a child point to its parent. One consequence: Nock has no garbage collector. (Nor can dag structure be detected, as with Lisp `eq`.)

There is also no single syntax for nouns. If you have nouns you have Nock; if you have Nock you have Hoon; if you have Hoon, you can write whatever parser you like.

6.3 Line 5: Nock Function

```
5  ::    nock(a)          *a
```

Nock is a pure (stateless) function from noun to noun. In our pseudocode (and only in our pseudocode) we express this with the prefix operator `*`.

Normally `a` in `nock(a)` is a cell `[s f]`, or as we say

```
[subject formula]
```

Intuitively, the formula is your function and the subject is its argument. We call them something different because Hoon, or any other high-level language built on Nock, will build its own function calling convention which *does not* map directly to `*[subject formula]`.

6.4 Line 6: Bracket grouping

```
6  ::    [a b c]          [a [b c]]
```

Ie, brackets (in our pseudocode, as in Hoon) associate to the right. For those with Lisp experience, it's important to note that Nock and Hoon use tuples or "improper lists" much more heavily than Lisp. The list terminator, normally 0, is never automatic. So the Lisp list

```
(a b c)
```

becomes the Nock noun

```
[a b c 0]
```

which is equivalent to

```
[a [b [c 0]]]
```

Note that we can and do use unnecessary brackets anyway, for emphasis.

6.5 Lines 8-13: Pseudocode notation

```
8  ::    ?[a b]           0
9  ::    ?a               1
10 ::    +[a b]           +[a b]
11 ::    +a               1 + a
12 ::    =[a a]           0
13 ::    =[a b]           1
```

Here we define more pseudocode operators, which we'll use in reductions further down. So far we have four built-in functions: `*` meaning Nock itself, `?` testing whether a noun is a cell or an atom, `+` incrementing an atom, and `=` testing for equality. Again, no rocket science here.

We should note that in Nock and Hoon, `0` (pronounced "yes") is true, and `1` ("no") is false. Why? It's fresh, it's different, it's new. And it's annoying. And it keeps you on your toes. And it's also just intuitively right.

6.6 Lines 16-20: Noun Axes and Trees

```
16 ::    /[1 a]           a
17 ::    /[2 a b]         a
18 ::    /[3 a b]         b
19 ::    /[(a + a) b]     /[2 /[a b]]
20 ::    /[(a + a + 1) b] /[3 /[a b]]
```

Slightly more interesting is our tree numbering. Every noun is of course a tree. The `/`, or axis, operator imposes an address space on that tree, mapping every nonzero atom to a tree position.

1 is the root. The head of every node `n` is `2n`; the tail is `2n+1`. Thus a simple tree:

```
     1
  2      3
4   5  6   7
         14 15
```

If the value of every leaf is its tree address, this tree is

```
  [[4 5] [6 14 15]]
```

6.7 Line 23:

```
23 ::    *[a [b c] d]      [*[a b c] *[a d]]
```

Um, what?

Since Nock of an atom just crashes, the practical domain of the Nock function is always a cell. Conventionally, the head of this cell is the “subject,” the tail is the “formula,” and the result of Nocking it is the “product.” Basically, the subject is your data and the formula is your code.

We could write line 23 less formally:

```
*[subject [formula-x formula-y]]
=>  [*[subject formula-x] *[subject formula-y]]
```

In other words, if you have two Nock formulas `x` and `y`, a formula that computes the pair of them is just `[x y]`. We can recognize this because no atom is a valid formula, and every formula that *does not* use line 19 has an atomic head.

If you know Lisp, you can think of this feature as a sort of "implicit cons." Where in Lisp you would write `(cons x y)`, in Nock you write `[x y]`.

6.8 Line 26: Constant

```
26 ::    *[a 1 b]          b
```

`1` is the constant operator. It produces its argument without reference to the subject.

6.9 Line 27: Recursion

```
27 ::    *[a 2 b c]        *[*[a b] *[a c]]
```

Line 22 brings us the essential magic of recursion. `2` is the Nock operator. If you can compute a subject and a formula, you can evaluate them in the interpreter. In most fundamental languages, like Lisp, `eval` is a curiosity. But Nock has no `apply` - so all our work gets done with `2`.

6.10 Lines 28-30: Cell Test, Increment, Equality

```
28 ::    *[a 3 b]          ?*[a b]
29 ::    *[a 4 b]          +*[a b]
30 ::    *[a 5 b]          =*[a b]
```

In lines 23-25, we meet our axiomatic functions again:

For instance, if `x` is a formula that calculates some atomic product, `[4 x]` calculates that product plus one.

6.11 Lines 10, 14, 21, 39: Crash defaults

```
10 ::    +[a b]            +[a b]
14 ::    =a                =a
21 ::    /a                /a
39 ::    *a                *a
```

The nock function is defined for every noun, but on many nouns it does nothing useful. For instance, if `a` is an atom, `*a` reduces to... `*a`. In theory, this means that Nock spins forever in an infinite loop. In other words, Nock produces no result - and in practice, your interpreter will stop.

(Another way to see this is that Nock has “crash-only” semantics. There is no exception mechanism. The only way to catch Nock errors is to simulate Nock in a higher-level virtual Nock - which, in fact, we do all the time. A simulator (or a practical low-level interpreter) can report, out of band, that Nock would not terminate. It cannot recognize all infinite loops, of course, but it can catch the obvious ones - like `*42`.)

6.12 Line 32: Nock 6

6 Reduction:

```
32 ::    *[a 6 b c d]      *[a 2 [0 1] 2 [1 c d] [1 0] 2 [1 2 3] [1 0] 4 4 b]

*[a 2 [0 1] 2 [1 c d] [1 0] 2 [1 2 3] [1 0] 4 4 b]

27 ::    *[a 2 b c]        *[*[a b] *[a c]]

*[*[a 0 1] *[a 2 [1 c d] [1 0] 2 [1 2 3] [1 0] 4 4 b]]

25 ::    *[a 0 b]          /[b a]

*[a *[a 2 [1 c d] [1 0] 2 [1 2 3] [1 0] 4 4 b]]

27 ::    *[a 2 b c]        *[*[a b] *[a c]]

*[a *[*[a [1 c d]] *[a [1 0] 2 [1 2 3] [1 0] 4 4 b]]]

26 ::    *[a 1 b]          b

23 ::    *[a [b c] d]      [*[a b c] *[a d]]

*[a *[[c d] [*[a 1 0] *[a 2 [1 2 3] [1 0] 4 4 b]]]]

26 ::    *[a 1 b]          b

*[a *[[c d] [0 *[a 2 [1 2 3] [1 0] 4 4 b]]]]

27 ::    *[a 2 b c]        *[*[a b] *[a c]]

*[a *[[c d] [0 *[*[a [1 2 3]] *[a [1 0] 4 4 b]]]]]

26 ::    *[a 1 b]          b

*[a *[[c d] [0 *[[2 3] *[a [1 0] 4 4 b]]]]]

23 ::    *[a [b c] d]      [*[a b c] *[a d]]

*[a *[[c d] [0 *[[2 3] [*[a [1 0]] *[a 4 4 b]]]]]]

26 ::    *[a 1 b]          b

*[a *[[c d] [0 *[[2 3] [0 *[a 4 4 b]]]]]]

29 ::    *[a 4 b]          +*[a b]

*[a *[[c d] [0 *[[2 3] [0 ++*[a b]]]]]]
```

`6` Reduced:

```
       32r ::   *[a 6 b c d]              *[a *[[c d] [0 *[[2 3] [0 ++*[a b]]]]]]
```

`6` is a primitive known to every programmer - good old "if." If `b` evaluates to `0`, we produce `c`; if `b` evaluates to `1`, we produce `d`; otherwise, we crash. We could have defined "if" as a built-in function, like increment by arbitrarily inventing another operator:

```
::    $[0 b c]         b
::    $[1 b c]         c
```

Then we could restate `6` quite compactly:

```
32c::    *[a 6 b c d]     *[a $[*[a b] c d]]
```

However, since we *can* write `if` as a macro, albeit a funky macro, we do so as to keep the Nock specification small.

In real life, of course, the Nock implementor knows that `6` is "if" and implements it as such. It's worth noting that in practical, compiler-generated Nock, we never do anything as funky as these `6` macro internals. There's no reason we couldn't build formulas at runtime, but we have no reason to and we don't - except when actually metaprogramming. As in most languages, normally code is code and data is data.

6.13 Line 29: Nock 7

`7` Reduction:

```
33 ::    *[a 7 b c]        *[a 2 b 1 c]

*[a 2 b 1 c]

27 ::    *[a 2 b c]        *[*[a b] *[a c]]

*[*[a b] *[a 1 c]]

26:    *[a 1 b]          b

*[*[a b] c]

```text
**`7` Reduced:**

```text
33r ::     *[a 7 b c]         *[*[a b] c]
```

`7` is an old mathematical friend, function composition. It's easy to see how this is built out of `2`. The data to evaluate is simply `b`, and the formula is `c` quoted.

6.14 Line 30: Nock 8

`8` Reduction:

```
34 ::    *[a 8 b c]        *[a 7 [[7 [0 1] b] 0 1] c]

*[a 7 [[7 [0 1] b] 0 1] c]

33r ::     *[a 7 b c]         *[*[a b] c]

*[*[a [7 [0 1] b] 0 1]] c]

23 ::    *[a [b c] d]      [*[a b c] *[a d]]

*[[*[a [7 [0 1] b]] *[a 0 1]] c]

25 ::    *[a 0 b]          /[b a]

*[[*[a [7 [0 1] b]] /[1 a]] c]
```

```
16 ::    /[1 a]            a

*[[*[a [7 [0 1] b]] a] c]

33r ::     *[a 7 b c]         *[*[a b] c]

*[[*[*[a 0 1]] b] a] c]
```

`8` Reduced:

```
34r ::     *[a 8 b c]        *[[*[a b] a] c]
```

`8` is similar to `7`, except that the subject for `c` is not simply the product of `b`, but the ordered pair of the product of `b` and the original subject.

Why would we want to do this? Imagine a higher-level language in which the programmer declares a variable. This language is likely to generate an `8`, because the variable is computed against the present subject, and used in a calculation which depends both on the original subject and the new variable.

6.15 Line 31: Nock 9

`9` Reduction:

```
31 :: *[a 9 b c]        *[a 7 c [2 [0 1] [0 b]]]

*[a 7 c [2 [0 1] [0 b]]]

33r ::     *[a 7 b c]         *[*[a b] c]

*[*[a c] [2 [0 1] [0 b]]]

27 ::     *[a 2 b c]        *[*[a b] *[a c]]

*[*[*[a c] [0 1]] *[*[a c] [0 b]]]

25 ::     *[a 0 b]          /[b a]
```

`9` Reduced:

```
31r ::     *[a 9 b c]        *[*[a c] *[*[a c] 0 b]]
```

`9` is a calling convention. With `c`, we produce a noun which contains both code and data - a *core*. We use this core as the subject, and apply the formula within it at slot `b`.

A *core* is a construct which is behind any kind of interesting control flow in Hoon. Of course, the Nock programmer is not constrained to use the same techniques as the Hoon compiler, but it is probably a good idea.

In Hoon, all the flow structures from your old life as an Earth programmer become cores. Functions and/or closures are cores, objects are cores modules are cores, even loops are cores.

The core is just a cell whose tail is data (possibly containing other cores) and whose head is code (containing one or more formulas). The tail is the *payload* and the head is the *battery*. Hence your core is

```
[battery payload]
```

To activate a core, pick a formula out of the battery, and use the entire core (*not* just the payload) as the subject.

(A core formula is called an *arm*. An arm is almost like an object-oriented method, but not quite - a method would be an arm that produces a function on an argument. The arm is just a function of the core, ie, a computed attribute.)

6.16 Line 36 and 37: Nock 10

** `10` Reduction:

```
 36 :: *[a 10 [b c] d]   *[a 8 c 7 [0 3] d]

34r ::     *[a 8 b c]        [[*[a b] a] c]

*[[*[a c] a] 7 [0 2] d]

33r ::     *[a 7 b c]        *[*[a b] c]

*[*[[*[a c] a] 0 3] d]
```

`10` reduced:

```
36r ::    *[a 10 [b c] d]   *[*[[*[a c] a] 0 3] d]

33 ::    *[a 10 b c]       *[a c]
```

If `b` is an atom and `c` is a formula, the formula `[10 b c]` appears to be equivalent to `c`. Likewise if `[b c]` is a cell, `[10 [b c] d]` appears to be equivalent to `d`.

`10` is actually a hint operator. The `b` or `[b c]` is discarded information - it is not used, formally, in the computation. It may help the interpreter compute the expression more efficiently, however.

Every Nock computes the same result - but not all at the same speed. What hints are supported? What do they do? Hints are a higher-level convention which do not, and should not, appear in the Nock spec. Some are defined in Hoon. Indeed, a naive Nock interpreter not optimized for Hoon will run Hoon quite poorly. When it gets the product, however, the product will be right.

(Why is the c in `[b c]` is computed? Because `c` could crash. A correct Nock cannot simply ignore it, and treat both variants of `10` as equivalent.)