

1

2

3

4

5
6
7
8
9
10
11
12
13
14
15
16
17
18
19

20

Nock for Everyday Coders

Tim Galebach ~timluc-miptev
Hyperware

Abstract

Nock is not super complex, and most programmers can learn the basics of it rapidly. The mental model gained by learning Nock turns out to be very useful in learning Hoon and understanding Urbit. While the Urbit docs generally suggest to not worry about Nock, Nock is very simple and small. Most programmers will feel more comfortable in Hoon having learned Nock’s basics. The goal of this tutorial is to explain Nock clearly in terms most programmers will relate to, to impart a feeling of confidence with very basic Nock, and to give a knowledge of Nock’s idioms and big wins so that they carry over to learning Hoon.

This tutorial was originally published online by ~timluc-miptev in early 2020 and remains an excellent exposition of Nock’s affordances.

Contents

21	1 Getting Started	2
22	1.1 Phases of Nock	3
23	1.2 What Is a Nock Interpreter?	3
24	1.3 How to Run Nock Code	4
25	1.4 Subject, Formula?	4
26	1.5 Evaluating Our First Nock Code	4

27	2 Fundamental Opcodes	5
28	2.1 Nock's Simplest Functions, 0 and 1	5
29	2.2 Binary Tree Addressing	5
30	2.3 0, the "Memory Slot" Function	7
31	2.4 1, the "Quoter" Function	9
32	2.5 4, the Incrementing Function	10
33	2.6 The Cell-Maker (AKA the Distribution Rule) . .	13
34	2.7 3, the Cell Detector, and 5, the Equality Tester	15
35	2.8 2, the "Subject-Altering" Function	19
36	2.9 Summary of Fundamental Opcodes	23
37	3 Composite Opcodes	23
38	3.1 An Aside About the 1 (Quoter Function) . . .	24
39	3.2 6, "If/Else" Conditional Branch	24
40	3.3 7, the "Composition" Opcode	28
41	3.4 8, the "Variable Push" Opcode	29
42	3.5 9, Run a Stored Procedure Arm in a Core . . .	30
43	3.6 10, Replace a Memory Slot	32
44	3.7 Real Nock Code	33
45	3.8 Summary	35
46	4 Interlude	35
47	4.1 Order of Operations	36
48	4.2 11, Hints and Side Effects for the Interpreter .	37
49	4.3 Nock in Hoon	38
50	5 The Core as Design Pattern	38
51	5.1 Building a Core Manually	39
52	6 Conclusion	46

53 1 Getting Started

54 When people first look at Nock, they see the definition, which
 55 is fairly intimidating. I'm talking about lines like this:

56 `/[(a + a) b]` `/[2 /[a b]]`
 57

The problem is, the programmer may already know that Nock code looks more like the below – just lists of numbers, with no symbols:

```
[6
 [5 [0 6] [1 0]]
 [0 7]
 [9 4 [[0 2] [2 [0 6] [0 5]] [4 0 7]]]
]
```

What gives? Which one is the “real” Nock?

1.1 Phases of Nock

We are looking at two different things in the examples above:

1. Pseudocode for *how to interpret* Nock.
2. Code to be interpreted (written as lists of numbers).

The symbols and spec are pseudocode, not real Nock code. They could just as easily be written in English, and they will never be written down as actual Nock code and given to an interpreter. They represent what an interpreter should do to turn Nock code into interpreter instructions.

The lists of numbers are the actual Nock code. This is what you feed to an interpreter to get some result.

1.2 What Is a Nock Interpreter?

An interpreter can be a computer program, or it can be a human manually expanding Nock code into results. In both cases, the program and human have to know the Nock pseudocode in order to do the right thing with incoming Nock code.

So a Nock interpreter is any entity that takes Nock code as input, and gives a noun as output. A noun can be:

```
:: a number
782
:: a cell (pair with two elements)
[782 9872]
:: each element can itself be a pair
```

```
94 [782 [9872 89728]]
95 :: the above can be written as
96 [782 9872 89728]
```

98 1.3 How to Run Nock Code

99 We will be expanding Nock pseudocode manually in the exam-
100 ples that follow, in effect acting as our own interpreter.

101 If we want to check that we're getting the right results from
102 our manual interpretation, we need to run a Nock interpreter,
103 such as the Urbit Dojo.

- 104 1. Start up a Dojo session on a fake ship.¹
- 105 2. At the prompt, we can execute Nock using `. * dottar`, e.g.,
106 `. * (NOCK_SUBJECT, NOCK_FORMULA)`.

107 1.4 Subject, Formula?

108 Let's keep this simple:

- 109 • subject = an argument to a function
- 110 • formula = the function

111 That's it. We'll see below how this works, going really
112 slowly with examples.

113 1.5 Evaluating Our First Nock Code

114 OK, so the interpreter takes two arguments, a "subject" and a
115 "formula". Both are nouns (a number or a cell). Let's run some
116 insanely simple Nock code in the Dojo:²

```
117 > . * (42 [0 1])
118 42
```

¹Consult `docs.urbit.org` for details.

²Code samples beginning with `>` are Dojo inputs (to wit, Hoon expressions).

121 In the above, 42 is our subject. [0 1] is our formula.
122 Formulas are always cells, and the first element of the cell
123 is a number that you can think of as *the name of the function*.
124 In this case, our function name is 0, which is the memory
125 slot function. It is always followed by 1 number, in this case 1,
126 which is the number of the memory slot to fetch in the subject.
127 Whenever we look at Nock code, we want to ask:

- 128 • What is the subject (function argument)? In this case,
129 it's 42.
- 130 • What is the formula (function)? In this case, it's [0 1].
- 131 • What value does that formula (function) produce when
132 called on this subject (argument)? In this case, the return
133 value is 42.

134 Why is the return value 42? How does this formula work?

135 2 Fundamental Opcodes

136 2.1 Nock's Simplest Functions, 0 and 1

137 The two most basic Nock functions are 0 address and 1 con-
138 stant. The goal here is to get strong intuitions of what they do,
139 how they handle edge cases, and how this relates to the Nock
140 spec/pseudocode.

141 2.2 Binary Tree Addressing

142 Before getting started on Nock proper, we should understand
143 how Nock and Hoon handle addresses in binary trees. If you
144 already understand why memory slot 5 of [['apple' %pie]
145 [0b1101 0xdad]] is %pie, you are good to go and can skip
146 ahead to the Section 2.3.

147 Every noun in Nock can be thought of as a tree, which
148 means we can give an exact number to access any position in
149 the tree. This means that, no matter how big our subject (ar-
150 gument) is, we can yank a value out of any part of it.

Noun tree addressing is directly addressed in the Nock specification:

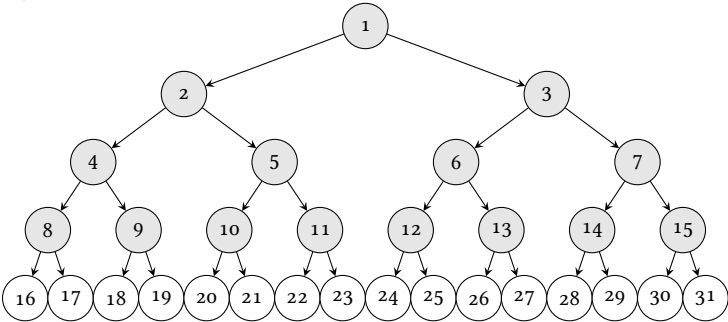
```

/[1 a]          a
/[2 a b]        a
/[3 a b]        b
/[(a + a) b]    /[2 /[a b]]
/[(a + a + 1) b] /[3 /[a b]]

```

This permits the address to be defined within a particular subtree as well as within the overall tree (Figure 1).

Figure 1: A binary tree with labeled node addresses to several layers.



That is, how do you say which slot number you want from a given tree? We say that:

- The tree root is address 1.
- The head of every node n is $2n$.
- The tail of every node n is $2n + 1$.

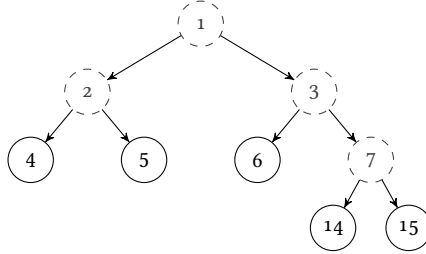
Let's take an example tree to illustrate. In Nock cell form, the tree is:

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

Diagrammatically, the tree looks like Figure 2.

- 1 is the address of the whole tree, `[[4 5] [6 14 15]]`.

Figure 2: A binary tree with some labeled node addresses.



- 174 • 2 is the address of the left branch, [4 5].
- 175 • 3 is the address of the right branch, [6 14 15].
- 176 • 15 is the value 15.

177 Let's play around now in the Dojo Nock interpreter so that
 178 we can confirm this. In each example, our subject (argument)
 179 will be the tree [[4 5] [6 14 15]].

```

180
181 :: formula (function) [0 1]: get the whole tree
182 > .*([[4 5] [6 14 15]] [0 1])
183 [[4 5] 6 14 15]
184
185 :: formula (function) [0 2]: get the left branch
186 > .*([[4 5] [6 14 15]] [0 2])
187 [4 5]
188
189 :: formula (function) [0 7]: get the subtree in slot 7
190 > .*([[4 5] [6 14 15]] [0 7])
191 [14 15]
192
```

193 2.3 @, the "Memory Slot" Function

194 The pseudocode for the @ opcode³ is as follows:

³From this point on, we will write Nock opcodes in special type so that they can be distinguished from other numbers in Nock. This reflects Hoon practice, which marks Nock rules as constants like %0. While not strictly necessary, it can be helpful when learning or coding Nock to see an explicit distinction.

```
195
199 *[_a @ b] /[_b a]
```

198 /[_b a] is pseudocode. In English, it means “a is a binary
199 tree. Get the memory slot numbered b.

200 So if a were the tree [9 10], and b were 1, we’d get the
201 memory slot 1 in the tree [9 10], which is just the tree itself.

202 Written in pseudocode, that’s /[_1 [9 10]]. We can also
203 do /[_2 [9 10]], which grabs memory slot 2, aka 9.

204 Let’s look at some examples. I’ve put them first in Dojo
205 form so that you can see how they run in that interpreter, and
206 then I show the “human interpreter” in pseudocode below that.

207 2.3.1 Example: Retrieve the Subject

```
208
209 :: get memory slot 1
210 > .*([50 51] [_ 1])
211 [50 51]
212 :: PSEUDOCODE -- replaces the Dojo's () with []
213 *[_[50 51] [_ 1]]
214 :: *[_a @0@ b] -> a = [50 51], b = 1
215 /[_1 [50 51]]
216 [50 51]
```

218 2.3.2 Example: Retrieve the Head of the Subject

```
219
220 > .*([50 51] [_ 2])
221 50
222 :: PSEUDOCODE -- replaces the Dojo's () with []
223 *[_[50 51] [_ 2]]
224 :: *[_a @0@ b] -> a = [50 51], b = 2
225 /[_2 [50 51]]
226 50
```

228 2.3.3 Example: A Crash

```
229
230 > .*([50 51] [_ [0 1]])
231 dojo: hoon expression failed
```



```

232 :: PSEUDOCODE
233 *[[50 51] [0 [0 1]]]
234 /[[0 1] [50 51]]
235 :: can't evaluate this--a memory slot must be a
236     number like 2, not a cell like [0 1]
237 CRASH
238

```

2.3.4 Summary of @

@ is everywhere in Nock, because “get something from a memory slot” really means “store stuff in a place and get it whenever I want,” which is another name for creating variables. These memory slot numbers take the place of variable names.

If you’re familiar with assembly or C, they’re conceptually similar to memory pointers or registers in how Nock uses them. Keep in mind though, Nock is functional/immutable, so it doesn’t update memory locations: it creates copies of data structures with altered values.

We’ve also seen that the memory slot function can’t take just anything as the memory slot to fetch: it must receive an atom (number).

2.4 1, the “Quoter” Function

This is another really simple function. You can think of it as a “quoter”: it just returns any value passed to it exactly as it is. It ignores the subject, and just quotes the value after it. Let’s look at a couple examples.

```

257 > .*([20 30] [1 67])
258 67
259
260 :: [@1@ 2 587] is same as [@1@ [2 [587]]]
261 > .*([20 30] [1 [2 [587]]]
262 [2 587])
263

```

It doesn’t matter how much information is after the 1: 1 is a dumb function that just returns it all.

The pseudocode for 1 is:

```

267
268 *[a 1 b]    b
269

```

In English, this means: “ignore the subject ‘a’, and just return everything after the 1 exactly as it is.

Let’s look at our first example, `.*([20 30] [1 67])`. The subject `a` is `[20 30]`, so we ignore that. What’s after the 1? `67`, so we return that.

In the second example, “everything after the 1” is longer, but the same rule applies: the Nock interpreter just returns it exactly as it is, after stripping out unnecessary brackets.

2.4.1 Summary of 1

1 is a simple function that just returns whatever is after it (“quoter” or “constant”). It’s useful for quoting values that you want to use later in your Nock code.

2.5 4, the Incrementing Function

0 and 1 are simple functions that don’t have any nested behavior. Now we’re going to move to a function that *does* have nested children, opcode 4. In these examples, pay attention to how 4 operates on its arguments; we’ll look at pseudocode and break down the examples in a moment.

```
> .*(50 [4 0 1])
51
> .*(50 [4 4 0 1])
52
> .*([100 150] [4 4 0 3])
152
> .*(50 [4 1 98])
99
> .*(50 [4 1 [0 2]])
dojo: hoon expression failed
```

Here’s 4’s pseudocode, juxtaposed with that of 0 and 1:

```
*[a 4 b]  **[a b]
*[a 0 b]  /[b a]
```

308 `*[a 1 b] b`

310 In English, this says “when we have subject `a`, function `4`,
 311 and Nock code `b`, first evaluate `[a b]` as `[subject formula]`,
 312 and then add `1` to the result.”

313 If we contrast with `0` and `1`, we see that the right side has
 314 a `*` symbol. This symbol means “evaluate the expression again
 315 in the Nock interpreter.” `0` and `1` *did not* have this symbol, and
 316 that’s why they couldn’t evaluate nested Nock expressions.

317 2.5.1 Example: Walking Through Increment

318 Let’s start by translating the Dojo’s `.*(subject formula)` to
 319 pseudocode of the form `*[a b]`, and then expand it line by
 320 line:

```

321 > .*(50 [4 0 1])
322 51
323 :: change to pseudocode
324 *[50 [4 0 1]]
325 :: move the 4 to the outside as +
326 **[50 [0 1]]
327 :: expand the 0 opcode to the memory slot operator "/"
328 +/[1 50]
329 :: grabs the memory slot
330 +(50)
331 :: evaluate
332 51
333 
```

335 The `[a b]` part of `**[a b]` expands to:

336 `[50 [0 1]]`

339 OK, this we know how to handle! It’s just our `0` function, and
 340 it wants the value in memory slot `1` of the subject. That’s `50`.

341 Now we know that `*[a b]=50`, and we just have to add `1`
 342 to it (the `+` in `**[a b]`). That is `51`, which is exactly what the
 343 interpreter gave us.

344 2.5.2 Example: Walking Through Increment

345 This one is similar, we just have an extra `4`. We again start by
 346 translating the Dojo’s `.*(subject formula)` to pseudocode,

347 and then expand

```

348
349 > .* (50 [4 4 0 1])
350 52
351 *[50 [4 4 0 1]]
352 :: the first 4 moves outside as a '+'
353 **[50 [4 0 1]]
354 :: 2nd 4 becomes a '+'...we have opcode 0 again!
355 ***[50 [0 1]]
356 ++/[1 50]
357 ++(50)
358 +(51)
359 52
360

```

361 2.5.3 Example: Walking Through Serial Increment

362 Here we again see lots of 4s applied consecutively, and we also
363 see how we can yank values out of a more complicated subject
364 and manipulate them. Notice how the subject is a cell, not an
365 atom. The rest is the same as in the previous example.

```

366
367 > .* ([100 150] [4 4 0 3])
368 152
369 a (the subject) = [100 150]
370 *[ [100 150] [4 4 0 3] ]
371 **[ [100 150] [4 0 3] ]
372 :: Now we've extracted all the increments,
373 :: so we just grab the value at memory slot 3
374 ***[ [100 150] [0 3] ]
375 ++/[3 [100 150]]
376 ++(150)
377 +(151)
378 152
379

```

380 2.5.4 Example: Walking Through Incrementing a Constant

381 In the below example, we see how we can use the quote/con-
382 stant function 1 to generate the value 98 and increment it. We
383 ignore the subject 50 completely.

```

384
385 > .* (50 [4 1 98])

```

```

386 99
387 *[50 [4 1 98]]
388 :: formula is the "quoter" function
389 **[50 [1 98]]
390 +(98)
391 99
392

```

393 2.5.5 Example: A Crash When Incrementing a Cell

394 Just as opcode 0 had values it couldn't handle (non-atoms), so
 395 opcode 4 needs the nested value inside it to evaluate to an atom.

```

396 > .*(50 [4 1 [0 2]])
397 dojo: hoon expression failed
398 *[50 [4 1 [0 2]])
399 :: OK cool, the nested value is a 1 opcode
400 **[50 [1 [0 2]])
401 :: 1 ignores the subject (50) and just returns [0 2]
402 +([0 2])
403 :: [0 2] isn't an atom, so how can we increment it???
404 :: We can't, so we crash.
405 dojo: hoon expression failed
406
407

```

408 2.5.6 Summary of 4

409 In these examples, we've seen that function 4 can be called as
 410 many times in a row as we want. At the end of those calls, it
 411 always ends up incrementing a number that either is yanked
 412 from the subject (memory slot function 0) or quoted as it is
 413 (quote function 1).

414 2.6 The Cell-Maker (aka the Distribution Rule)

415 The Nock interpreter is allowed to return nouns, which are
 416 atoms (positive numbers) or cells (pairs of nouns). What have
 417 our functions/opcodes been returning so far?

- 418 • 0: atoms or cells, depending what's in the memory slot
 419 that we yoink.

420 • 1: atoms or cells, depending on what we quote.

421 • 4: just atoms.

422 But what if my subject was [51 67 89], and I wanted to
423 increment every value and return that as [52 68 90]? How
424 can I do that when it's a cell, and 4 only seems able to return
425 atoms?

426 The answer is something that the Nock docs call the “distri-
427 bution rule” or “implicit cons” (hello, fellow LISPer!), but that
428 I find easiest to think of as the “Cell-Maker Rule”.

429 2.6.1 A Quick Detour into Nock Formulas

430 We haven't talked much yet about what values are legal to feed
431 into the Nock interpreter (the `.*(subject formula)` func-
432 tion in the Dojo). So far, we've only been using formulas that
433 start with numbers (our functions/opcodes 0/1/4).

434 Let's now fully solidify our understanding of what's a legal
435 formula by taking a quick look at the three possible cases. (The
436 format is `*(subject formula)`.)

437 • The formula is a cell starting with an opcode. We know
438 this is OK.

```
439 > .*(50 [0 1])
440 50
441
442
```

443 • The formula is just an atom (o). This is not valid.

```
444 > .*(50 0)
445
446 dojo: hoon expression failed
447
```

448 • Finally, we try a formula that is a cell starting with an
449 atom. This looks invalid, but—it works!

```
450 > .*(50 [[0 1] [1 203]])
451 [50 203]
452
453
```

454 So apparently a formula cell can start with a cell. The Cell-
455 Maker rule is:

```

456
457 * [subject [formula-x formula-y]]→
458   [*[subject formula-x] *[subject formula-y]]
459

```

460 In our example above, `formula-x` is `[0 1]`, and `formula-y`
 461 is `[1 203]`. They each evaluate individually against the sub-
 462 ject, and the end result is a cell.

463 We can make as many cells in a row as we want:

```

464
465 > .* (50 [[0 1] [1 203] [0 1] [1 19] [1 76]])
466 [50 203 50 19 76]
467

```

468 We can put any operation inside each cell:

```

469
470 > .* ([19 20] [[0 1] [1 76] [4 4 0 3]])
471 [[19 20] 76 22]
472

```

473 If we take the returned collection `[[19 20] 76 22]` in or-
 474 der, we can write in English how they connect to our collection
 475 of formulas that we passed:

- 476 • `[0 1]`: get memory slot 1
- 477 • `[1 76]`: return the quoted value 76
- 478 • `[4 4 0 3]`: increment twice the value in memory slot 3
 479 (20)

480 So we can pass one small subject `[[19 20]]` and make an
 481 arbitrarily long collection of values from it, using any functions
 482 we want. Cell-Maker FTW!

483 2.7 3, the Cell Detector, and 5, the Equality Tester

484 Now we come to functions/opcodes 3 and 5, which are pretty
 485 straightforward after we've seen how 4 and the Cell-Maker
 486 work. Functions 3 and 5, like 4, allow nested evaluation. Let's
 487 put all their pseudocode definitions together to compare:

```

488
489 :: function/opcode 3
490 * [a 3 b]    ?*[a b]
491 :: function/opcode 4
492 * [a 4 b]    **[a b]

```

```

493 :: function/opcode 5
494 *[a 5 b c] =[*[a b] *[a c]]
495

```

496 First of all, notice how the right side of all these “equations”
 497 has the evaluation operator, *. This means that these functions
 498 can have nested formulas, since they keep evaluating all the
 499 way down.

500 There are some new pseudocode symbols here that we need
 501 to translate into English. We already know +: “increment the
 502 value after this”. Now we also see:

- 503 • ?: “check whether the value after this is a cell. Return o
 504 if yes, 1 if no”.⁴
- 505 • =: “first run the function in b with subject a as the argu-
 506 ment, and same for the function in c. If the results are
 507 equal, return o, if not, return 1.

508 2.7.1 Example: Not a Cell

```

509
510 > .* (50 [3 0 1])
511 1
512 :: PSEUDOCODE OF THE ABOVE
513 *[50 [3 0 1]]
514 ?*[50 [0 1]]
515 :: get memory slot 1 of the subject
516 ?(50)
517 :: is 50 a cell? No, so return 1
518 1
519

```

520 2.7.2 Example: A Cell

```

521
522 > .* ([50 51] [3 0 1])
523 0
524 :: PSEUDOCODE OF THE ABOVE
525 *[[50 51] [3 0 1]]
526 ?*[[50 51] [0 1]]

```

⁴Nock conventionally explains that there is one “true” case and (potentially) many “false” cases, thus o/true and 1/false.


```

527 ::get memory slot 1 of the subject: [50 51]
528 ?([50 51])
529 :: is [50 51] a cell? Yes, so return 0
530 0
531

```

532 2.7.3 Example: Nested Cell Evaluation with 4

```

533
534 > .*([50 51] [4 4 3 0 1])
535 2
536 :: PSEUDOCODE OF THE ABOVE
537 *([50 51] [4 4 3 0 1])
538 **([50 51] [4 3 0 1])
539 :: whatever comes out of the 3 function,
540 :: we're gonna increment it twice
541 ***([50 51] [3 0 1])
542 :: down to just fetching memory slot 1
543 ++?([50 51] [0 1])
544 ++?([50 51])
545 :: is [50 51] a cell? Yes, so return 0
546 ++(0)
547 +(1)
548 2
549

```

550 2.7.4 Example: Check Multiple Cases of Cells

```

551
552 > .*([50 51] 52) [[3 0 2] [3 0 3]]
553 [0 1]
554 *([50 51] 52) [[3 0 2] [3 0 3]]
555 ?[*([50 51] 52) [0 2]]
556   *([50 51] [0 3])
557 :: yank memory slots 2 and 3
558 ?*([50 51] 52)
559 :: first is a cell, second is not
560 [0 1]
561

```

562 2.7.5 Example: Equality Test

563 Because 5 compares the results of 2 formulas, it *always* makes
 564 2 inner evaluations of the subject. It's similar to Cell-Maker in

565 this way.

```

566
567 > .*([50 51] [5 [0 2] [0 2]])
568 0
569 :: PSEUDOCODE
570 *([50 51] [5 [0 2] [0 2]])
571 :: factor out the =
572 =[*([50 51] [0 2]) *([50 51] [0 2])]
573 :: get memory slot 2 twice
574 =(50 50)
575 0
576

```

577 2.76 Example: Comparing Two Unequal Values

```

578
579 > .*([50 51] [5 [0 2] [0 3]])
580 1
581 :: PSEUDOCODE
582 *([50 51] [5 [0 2] [0 3]])
583 :: factor out the =
584 =[*([50 51] [0 2]) *([50 51] [0 3])]
585 :: get memory slot 2 and memory slot 3
586 =(50 51)
587 1
588

```

589 2.77 Example: Compare Values with Function Calls

```

590
591 > .*([50 51] [5 [4 0 2] [0 3]])
592 0
593 :: PSEUDOCODE
594 *([50 51] [5 [4 0 2] [0 3]])
595 :: factor out the =
596 =[*([50 51] [4 0 2])
597   *([50 51] [0 3])]
598 :: factor out the +
599 =[+*([50 51] [0 2])
600   *([50 51] [0 3])]
601 :: get memory slots 2 and 3
602 =[+50 51]
603 :: evaluate the +
604 =[51 51])

```

605 0
606

607 2.7.8 Example: Compare Cells with Function Calls

```
608 > .*([99 99] [5 [1 [99 99]] [0 1]])
609
610 0
611 :: PSEUDOCODE
612 *[[99 99] [5 [1 [99 99]] [0 1]]]
613 =[*[99 99] [1 [99 99]]]
614   *[[99 99] [0 1]]]
615 :: first cell is the result of quoter function
616 :: second cell is the result of memory fetch
617 =[99 99] [99 99]]
618 :: true
619 0
620
```

621 2.7.9 Summary of 3 and 5

622 These along with 4 are known as the “axiomatic” functions.
623 They exist to implement definitional logic for the Nock spec-
624 ification. In particular, the cell check and the equality check
625 provide for structural analysis of nouns.

626 2.8 2, the “Subject-Altering” Function

627 In all our examples so far, the subject has been defined at the
628 start when we call the interpreter, and never changes. But what
629 if we want to change the subject?

630 A different subject? Why would we want that? Here’s an
631 easy example. Say you found the following piece of Nock code
632 on the interwebz:

```
633 [8 [1 0] 8 [1 6 [5 [0 7] 4 0 6] [0 6] 9 2 [0 2]
634 [4 0 6] 0 7] 9 2 0 1]
635
```

637 This is the code for a Nock function that expects a subject
638 that is an atom, and decrements that subject by 1. You can
639 actually enter it in the Dojo right now:

```

640
641 > .*(100 [8 [1 0] 8 [1 6 [5 [0 7] 4 0 6] [0 6] 9 2 [0
642      2] [4 0 6] 0 7] 9 2 0 1])
643 99
644

```

645 This works! You do not have to understand the code right now;
646 just try entering different numbers instead of 100 to see the
647 program working.

648 But now imagine that I have a Nock program with a differ-
649 ent subject

```

650
651 > .*([50 51] some-formula)
652

```

653 And somewhere inside `some-formula`, I want to decre-
654 ment the number 51. I can't pass `[50 51]` as the subject to
655 my decrementer code above though, because that's a cell, and
656 it expects just an atom (a number).

657 2.8.1 Subject Altering to the Rescue

658 The function/opcode `2` is designed to handle this problem for us.
659 First the pseudocode:

```

660 :: PSEUDOCODE
661 *[a 2 b c]  *[*[a b] *[a c]]
662
663

```

664 `2` expects 2 formulas after the subject `a`: `b` and `c`. With
665 those, it:

- 666 1. runs formula `b` against the subject to set up a new envi-
667 ronment/subject derived from the subject
- 668 2. runs formula `c` against the subject to prepare a 2nd func-
669 tion
- 670 3. run that 2nd function against the new environment/sub-
671 ject from step (1)

672 Note that the pseudocode for `2` has nested “*”s.

```

673
674 *[*[a b] *[a c]]
675

```

676 The two inner *s run steps (1) and (2), and the outer one,
677 around the whole expression, runs step (3).

678 2.8.2 Example: Change the Subject and Call a Constant Value

```

679
680 > .*([50 51] [2 [0 3] [1 [4 0 1]]])
681 52
682 :: PSEUDOCODE
683 *[[50 51] [2 [0 3] [1 [4 0 1]]]]
684 :: separate b and c to each run against the subject
685     (steps 1 and 2)
686 *[*[[50 51] [0 3]] *[[50 51] [1 [4 0 1]]]]
687 :: after steps 1 and 2, we have a new subject, 51!
688 :: note how we're back in normal *[subject formula]
689     form
690 *[[51 [4 0 1]]
691 :: apply the 4 function as we're used to
692 **[[51 [0 1]]
693 :: grab 51 from memory slot 1
694 +(51)
695 52
696

```

697 2.8.3 Example: Grab a Block of Code from the Subject and Run It

698 Think of this as grabbing a “stored procedure” from the subject.

```

699
700 > .*([[4 0 1] 51] [2 [0 3] [0 2]]])
701 52
702 :: PSEUDOCODE, subject is [[4 0 1] 51]
703 *[[[4 0 1] 51] [2 [0 3] [0 2]]]
704 *[*[[[4 0 1] 51] [0 3]
705     *[[[4 0 1] 51] [0 2]]]]
706 :: step 1 gets memory slot 3, step 2 grabs memory
707     slot 2
708 *[[51 [4 0 1]]
709 :: looks like a normal 4 opcode to me!
710 **[[51 [0 1]]
711 :: grab memory slot 1
712 +(51)
713 52
714

```

715 2.8.4 Example: Back to Our Motivating Case

716 Remember our decrementing block of code that we couldn't
 717 use when the subject was [50 51], instead of just an atom?
 718 Opcode 2 makes handling that issue a piece of cake.

719 We simply use 2 to transform our subject into an atom, and
 720 use 1 to quote the decrement block of code before it evaluates
 721 in step (3).

```

722 > .*([50 51] [2 [0 2] [1 [8 [1 0] 8 [1 6 [5 [0 7] 4 0
723      6] [0 6] 9 2 [0 2] [4 0 6] 0 7] 9 2 0 1]]])
724
725 49
726 :: PSEUDOCODE (subject is [50 51])
727 :: ("decrement-formula" substituted for clarity)
728 *[[50 51] [2 [0 2] [1 decrement-formula]]]
729 *[*[[50 51] [0 2]] *[[50 51] [1 decrement-formula]]]
730 *[50 decrement-formula]
731 ::decrement-formula has the atom subject that it wants
732 49
733 
```

734 2.8.5 Summary of 2

735 For those who know a bit of Hoon, the second example above
 736 is rather similar to calling an arm that produces a gate, and
 737 then running the gate. Most of Hoon runs this type of stored-
 738 procedure plus subject-altering Nock, and it all uses opcode 2
 739 at its base.

740 And for those who know Hoon, [[4 0 1] 51] should al-
 741 ready be looking a lot like [battery payload]... that's not
 742 a coincidence. We're starting to see the first glimpses of how
 743 Hoon's cores, arms and subjects/subject mutations flow out of
 744 the fundamental structure of Nock.

745 We also see how Hoon/Nock lend themselves well to throw-
 746 ing around chunks of code, and adjusting the subject as neces-
 747 sary to create the correct subject/environment against which
 748 to run that code.

749 2.9 Summary of Fundamental Opcodes

750 You now have seen all the fundamental functions/opcodes in
751 Nock. In the next part, I'll introduce the remaining functions,
752 which no longer need pseudocode: we have enough scaffold-
753 ing now to build the rest of Nock from Nock itself. Instead,
754 these new opcodes will just be shortcuts/macros/code expan-
755 sions building on opcodes 0-5.

756 We'll also start to connect Nock to Hoon, and see how most
757 of the fundamental (and slightly weird) features of Hoon flow
758 directly from Nock's structure, and make a lot more sense in
759 combination with it.

760 3 Composite Opcodes

761 A word of encouragement: we're done with the hard part now.
762 Every Nock function we learn in this section will be built from
763 pieces in the preceding section.

764 None of these new functions are *necessary* to make Nock
765 work. All of them except Nock 10 and 11 are syntactic sugar
766 that is part of the Nock definition, and must be built into every
767 correct Nock implementation. If you have seen macros or code
768 expansions in other languages, that's another word for what's
769 happening here.

770 Nock 10 makes it easier to replace a memory value some-
771 where in a tree, and Nock 11 allows passing hints to the inter-
772 preter.

773 One point of clarity: this syntactic sugar/code-expansion
774 system is *not* extensible. This means that you can't invent your
775 own Nock opcodes and still have that language be Nock; you're
776 making a higher-level language on top of Nock at that point.

777 In fact, that's not a bad way to think of Hoon: it's a higher-
778 level language that adds missing syntactic sugar/macros and
779 human-readable names to Nock. (That's not the whole story,
780 but it's a decent mental peg to initially hang Hoon on.)

781 Nock is intentionally very, very small, such that you can
782 always walk through and analyze what is happening in a block
783 of code if you know Nock's opcodes.

784 3.1 An Aside About the 1 (Quoter Function)

785 You're going to see the 1 opcode (the "Quoter") appear in a lot
 786 of examples below for a simple reason: the 6-9 opcodes expect
 787 formulas in a lot of places, not just atoms. And, as we've seen
 788 already, formulas have to be cells.

789 Whenever a formula is required, but you really just want
 790 to return a number, you use the quoter function.

791 3.1.1 Example: Incrementing a Constant

```
792 :: We just want to run 4 on the number 5,  
793 :: but 4 expects a formula after it, so we use [1 5]  
794 > .* (0 [4 1 5])  
795 6  
796  
797
```

798 3.1.2 Example: Comparing a Memory Slot to a Constant Value

```
799 :: we grab memory slot 2  
800 :: then it has to be compared to the result of a  
801 formula  
802 :: so we just use the formula [1 23] to return 23  
803 > .* ([23 45] [5 [0 2] [1 23]])  
804 0  
805  
806
```

807 3.2 6, "If/Else" Conditional Branch

808 The remaining Nock opcodes are "sugar". This means that the
 809 functions in this next part will use only * and Nock code in
 810 their pseudocode. For example, here is the code for 6, the
 811 "If/Else" function.

```
812 * [a 6 b c d]      * [a * [[c d] 0 * [[2 3] 0 * [a 4 4 b]]]]  
813
```

815 The prose explanation of what's happening is straightfor-
 816 ward:

- 817 1. Evaluate b against the subject a (* [a b]) to see whether
 818 it's 0/true or 1/false.

- 819 2. If b equals 0/true, run formula c against subject a.
- 820 3. If b equals 1/false, run formula d against subject a.
- 821 4. If b is not equal to 0 or 1, the code crashes, for reasons
- 822 we'll see in the code explanation.

823 3.2.1 Code Explanation (Way More Fun)

824 OK, so that's the English version. The code explanation (the
825 right side of the definition) is *really fun* now that we know the
826 basic Nock opcodes.

827 The pseudocode has four nested “[subject formula]’s, so
828 I’m going to unwrap those to the bottom, and then build it up
829 again. The layers are, in order:

- 830 1. `*[a ...]`, i.e. subject a evaluated against that long for-
831 mula starting with `*[[c d] 0 ...]`.
- 832 2. `*[c d]`, i.e. subject c evaluated against the formula start-
833 ing with `[[2 3 ...]`.
- 834 3. `*[2 3]`, i.e. subject 2 evaluated against the lowest-level
835 formula.
- 836 4. Finally, subject a evaluated against formula `[4 4 b]`.

837 3.2.2 Step 4

838 Remember, our English explanation was “see if `*[a b]` is true
839 or false, and do different actions depending on that. Step Four
840 is that check.

841 Let’s say we have a as 59, and b as the quoted value 0/true.

```

842 *[ a 4 4 b ]
843 :: a: 59
844 :: b: [1 0]
845 *[59 4 4 [1 0]]
846 :: substitute out the two increment operators
847 ***[59 [1 0]]
848 :: ignore subject, return quoted value 0
849 ++(0)
850 2
851
852
```

In summary, because `*[a b]` evaluated to `0/true`, we get the number 2. If `*[a b]` had been `1/false`, we'd get 3 (because we'd evaluate `++(1)`).

What the heck? Why are we getting `2` or `3` back? How does that help us?? Well, Nock uses that returned `2` or `3` as a *memory slot*, and executes the code in that memory slot.

3.2.3 Step 3

Now we go up to step 3, with the subject `[2 3]` evaluated against our return value from step 4. Let's imagine that 2 had been returned:

```

:: remember, "result-of-step-4" was 2
*[[2 3] 0 result-of-step-4]
*[[2 3] 0 2]
:: get memory slot 2
2

```

This grabs memory slot 2 if `*[a b]` was true, and memory slot 3 if `*[a b]` was false.

Wait, isn't this redundant? We are just using our 2 or 3 generated in step 4 to generate a 2 or a 3. Seems dumb.

The answer is that we make sure that a crash happens if `*[a b]` yields any answer other than `0/true` or `1/false`. If `*[a b]` returned `10`, for example, we'd have the following code in step 3:

```

*[[2 3] 0 10]
:: there's no slot 10
CRASH!!

```

This is exactly what we want: the program crashes unless we are doing a loobean⁵ test that returns a 0 or 1 (converted to an indicial 2 or 3) in step 4.

3.2.4 Step 2

We now have our validated 2 or 3 to plug into step 2. Let's imagine `c` is the simple formula `[0 1]` and `d` is `[1 203]`.

⁵A boolean truth value but with the typical truth values of 0 and 1 reversed to `0/true` and `1/false`.

```

889
890 :: if step 3 returned "2" (true)
891 *[[[_ 1] [_ 203]] _ 2]
892 [_ 1]
893

```

894 Not much to see here: we just grab memory slot 2 or 3
895 depending on whether our initial b was true or false.

896 3.2.5 Step 1

897 And now we're back at the top level, where we just use whichever
898 formula we yonked in step 2 and run it against a.

```

899
900 *[a formula-from-step-2]
901 :: let's say we returned formula [_ 1]
902 :: our original a, from step 4, was 59
903 *[59 [_ 1]]
904 59
905

```

906 3.2.6 Example: Code Expansion of 6

```

907
908 > .*(1 [_ 6 [_ 1] [_ 1] [4 0 1]])
909 :: PSEUDOCODE
910 :: *[a *[[c d] 0 *[[2 3] 0 *[a 4 4 b]]]]
911 *[_ 1 *[[[_ 1] [4 0 1]] _ *[[2 3] _ *[_ 4 4 [0 1]]]]]
912 :: factor out the 4 opcodes
913 *[_ 1 *[[[_ 1] [4 0 1]] _ *[[2 3] _ **[_ 1 [_ 1]]]]]
914 :: b evaluates to 1 (yank memory slot 1)
915 *[_ 1 *[[[_ 1] [4 0 1]] _ *[[2 3] _ ++(1)]]]
916 :: evaluate the two increments
917 *[_ 1 *[[[_ 1] [4 0 1]] _ *[[2 3] _ 3]]]
918 :: get memory slot 3 from [2 3]
919 *[_ 1 *[[[_ 1] [4 0 1]] _ 3]]
920 :: [0 3] means get memory slot 3 from the subject
921 (formula [4 0 1])
922 *[_ 1 [4 0 1]]
923 :: factor out the 4 opcode
924 **[_ 1 [_ 1]]
925 +(1)
926 2
927

```

928 3.2.7 Summary of 6

929 Now I'm going to make you a little sad. Most Nock interpreters
 930 don't do this whole awesome code expansion. They just see 6,
 931 and implement an if/else check with a crash if `*[a b]` isn't a
 932 loobean.

933 However, the pattern of storing chunks of code in memory
 934 and pulling them out when you want them is the most impor-
 935 tant part of Nock. In fact, those of you who know Hoon are
 936 probably already seeing the makings of the "core" code pat-
 937 tern.

938 3.3 7, the "Composition" Opcode

939 7 is so simple we barely need to spend time on it. All it does is
 940 create a new subject/environment (using 2), and immediately
 941 runs a formula against that new subject. Let's compare it to 2:

```
942 :: opcode 7
943
944 *[a 7 b c]  *[*[a b] c]
945
946 :: opcode 2
947 *[a 2 b c]  *[*[a b] *[a c]]
948
```

949 This is almost exactly the same except with `c` instead of
 950 `*[a c]`. This means that you can write a "subject-changing"
 951 formula in `b`, and then run a simple function against that in `c`.

952 3.3.1 Example: Opcode 2 vs. Opcode 7

```
953
954 :: using opcode 2
955 > .*([23 45] [2 [0 3] [1 4 0 1]])
956 46
957
958 :: using opcode 7 -- we get to remove the quoter
959   opcode "1"
960 :: wow amazing /sarcasm
961 > .*([23 45] [7 [0 3] [4 0 1]])
962 46
963
```

964 Opcode 7 is clearly trivial, so why does it exist? It allows
 965 clean expression of function composition: this is really $c(b(a))$,
 966 where a (the subject) is the initial argument, and b and c are
 967 functions. This pattern comes up a ton, and it's nice to not use
 968 1s everywhere.

969 3.4 8, the "Variable Push" Opcode

970 If you're ever writing Nock and think "how can I add a new
 971 variable to the subject?", opcode 8 is what you want. The new
 972 variable can be based on either the existing subject, or be a new
 973 value you add on.

```
974 *[a 8 b c]          *[[*[a b] a] c]
```

977 This says to run $*[a\ b]$, and then make that the head of a
 978 new subject, with the old subject a as the new subject's tail.

979 3.4.1 Example: Add Variable as a Copied Value from an Existing Sub- 980 ject

981 In English, the below code first yanks the variable from mem-
 982 ory slot 3, copies it to the head of a new subject, and then in-
 983 crements the value in that new head.

```
984 > . *([67 39] [8 [0 3] [4 0 2]))
```

```
985 40
```

```
986 :: PSEUDOCODE
```

```
987 *[[*[67 39] [0 3]] [67 39]] [4 0 2]]
```

```
988 :: yanks mem slot 3 and pins it to the front of the
```

```
989 old subject
```

```
990 *[[39 [67 39]] [4 0 2]]
```

```
991 +*[[39 [67 39]] [0 2]]
```

```
992 +(39)
```

```
993 40
```

996 3.4.2 Example: Add Variable as a New Value

```
997 > . *([67 39] [8 [1 0] [4 0 2]))
```

```
998 1
```

```

1000 :: PSEUDOCODE
1001 *[[*[[67 39] [1 0]] [67 39]] [4 0 2]]
1002 *[[0 [67 39]] [4 0 2]]
1003 **[[0 [67 39]] [0 2]]
1004 +(0)
1005 1
1006

```

1007 Why do we want this? In the first example, we pin a copy
1008 of a value so that we can manipulate it without changing the
1009 original. In the second example, we add a 0 to the front; maybe
1010 we want to increment it until some condition is met?

1011 The above should be starting to feel *very* Hoon-ish: we're
1012 a minor code transform away from pinning new values to the
1013 head of a payload.

1014 3.5 9, Run a Stored Procedure Arm in a Core

1015 We're almost at full Hoon now, although still at a very low/raw
1016 level. 9 looks a little complicated...

```

1017
1018 *[a 9 b c]  *[*[a c] 2 [0 1] 0 b]
1019

```

1020 ... but in English, this is just saying:

- 1021 1. Use formula *c* to make a new subject from *a* (*[*a* *c*]).
- 1022 2. Grab the formula located at memory slot *b* in that new
1023 subject.
- 1024 3. Run that formula against the new subject *[*a* *c*].

1025 The fact that the above description has the words “make a new
1026 subject” tells us right away that it's syntactic sugar for opcode
1027 2, and that's indeed what we see in the pseudocode.

1028 3.5.1 Example: Using 9 to Run an Increment Arm

1029 The initial expression here likely looks cryptic, but if you fol-
1030 low the pseudocode, it will become clear.

1031 To stay oriented, remember that, if we're thinking of 9 as
1032 *[*a* 9 *b* *c*], then

```

1033
1034 a: 45
1035 b: 2
1036 c: [[1 4 0 3] 0 1]
1037

```

```

1038
1039 > .* (45 [9 2 [1 4 0 3] 0 1])
1040 46
1041 :: PSEUDOCODE
1042 * [45 [9 2 [1 4 0 3] [0 1]]]
1043 * [* [45 [1 4 0 3] [0 1]] 2 [0 1] 0 2]
1044 :: new subject is [[4 0 3] 45]
1045 :: that is a formula to increment mem slot 3 in the
1046 :: head; 45 in the tail
1047 * [[[4 0 3] 45] 2 [0 1] 0 2]
1048 :: now we expand opcode 2
1049 * [* [[[4 0 3] 45] 0 1]
1050      * [[[4 0 3] 45] 0 2]]
1051 :: mem slot 2 of the subject becomes the new formula
1052 * [[[4 0 3] 45] 4 0 3]
1053 + * [[[4 0 3] 45] 0 3]
1054 :: grab mem slot 3
1055 + (45)
1056 46
1057

```

1058 We start above with a subject that is not a core; it's just the
 1059 atom 45. The code for c is then:

```

1060
1061 [[1 4 0 3] [0 1]]
1062

```

1063 This formula uses the Cell Maker (Distribution Rule) to in-
 1064 sert [4 0 3] as the head of the new subject, and puts mem
 1065 slot 1 of the old subject as the tail, so we get a new subject of
 1066 [[4 0 3] 45].

1067 What we do next is use b (here, 2) to select memory slot 2
 1068 from that new subject. Memory slot 2 is a formula: [4 0 3].
 1069 We run that formula against the new subject.

1070 3.5.2 Key Point/Possible Confusion

1071 When I first saw 9, I thought: “why didn’t they just use 2 to
 1072 make the transformed subject? Why is there an extra step to

1073 use [0 1] to pull the new *[a c] subject? Why can't we do
 1074 *[a 2 c [0 b]]?

1075 The answer is that we actually use the *[a c] subject *twice*:

- 1076 1. We extract from it the formula located at memory slot b.
- 1077 2. Then we run *that* formula against the *[a c] subject.

1078 If we just did *[a 2 c [0 b]], then [0 b] would try to
 1079 look up the b memory slot in a, NOT *[a c]. So 9 gives us
 1080 one extra step to set up the core itself.

1081 3.5.3 How to Think of 9

1082 I like to think of 9 as having three parts:

- 1083 1. c: our formula to set up the subject, making a new sub-
 1084 ject.
- 1085 2. b: the memory slot in our new subject where an arm is.
- 1086 3. Run the arm located at b against the new subject.

1087 You can think of this as setting up a subject, pulling a stored
 1088 procedure from it, and then running that procedure against the
 1089 subject.

1090 3.6 10, Replace a Memory Slot

1091 Before explaining 10, we need to introduce a new operator, #.
 1092 # is the “edit” operator. It has the form

```
1093 #[mem-slot new-val target-tree]
```

1096 It replaces the memory slot mem-slot in target-tree with
 1097 new-val.

```
1098 :: Example
1099 #[2 [4 5] [99 88 77]]
1100 [[4 5] 88 77]
```

1103 The pseudocode for 10 is:


```
1104
1105 *[a 10] [b c] d]  #[b *[a c] *[a d]]
```

1107 In English, this first calculates `*[a c]` and `*[a d]`, and
 1108 then replaces memory slot `b` in the latter with the result of the
 1109 former.

1110 3.6.1 Example: Using `10` to Replace a Memory Slot

```
1111
1112 > .*(50 [10 [2 [0 1]] [1 8 9 10]))
1113 [50 9 10]
1114 :: PSEUDOCODE
1115 *[50 [10 [2 [0 1]] [1 8 9 10]]]
1116 #[2 *[50 0 1] *[50 1 8 9 10]]
1117 #[2 /[1 50] [8 9 10]]
1118 #[2 50 [8 9 10]]
1119 :: expanded as far as we can, now do the edit
1120 :: replace mem slot 2 of [8 9 10] with the value 50
1121 [50 9 10]
```

1123 We will defer discussion of opcode `11` to a later section be-
 1124 cause it is not part of strict Nock semantics.

1125 3.6.2 Towards Hoon

1126 If “grab a chunk of code from a subject and then run it against
 1127 the subject” sounds a lot like getting an arm from a core in
 1128 Hoon, that’s because it is. Nock opcode `9` is hand-in-glove
 1129 with Hoon’s core/arm structure. The “new subjects” created by
 1130 `*[a c]` are cores, and the things selected from memory slots
 1131 by `b` are arms.

1132 3.7 Real Nock Code

1133 To finish this off and take your new powers for a spin, let’s look
 1134 at some real Nock code from the wild. I got the below example
 1135 from the Dojo. It’s a mold: a function that takes a noun and
 1136 returns it if it’s the correct type, and crashes if not. The mold
 1137 here checks whether the input noun is a loobean (`0/true` or
 1138 `1/false`).

1139 3.71 Example: A Loobean Mold

```

1141 :: loobean mold gate
1142 > =loobean-mold ?
1143
1144 :: below output truncated for our purposes here
1145 > loobean-mold
1146 < 1.fxf ... >
1147
1148 :: grab the code for the battery
1149 > -.loobean-mold
1150 [8 [6 [5 [1 0] 0 6] [1 0] 6 [5 [1 1] 0 6] [1 1] 0 0]
1151 8 [5 [0 14] 0 2] 0 6]

```

1153 So we:

- 1154 1. Assign the mold gate denoted by ? to the face loobean-mold.
- 1155 2. Examine what's inside, seeing that the head is an arm.
- 1156 3. Print out that source code by calling the head.

1157 We know the below code is a gate that evaluates its sample,
 1158 and returns it if it's a loobean, and crashes otherwise. Let's see
 1159 how that works.

```

1161 [6 [5 [1 0] 0 6] [1 0] 6 [5 [1 1] 0 6] [1 1] 0 0]
1162

```

1163 We start with 6, which means this is an if-else. The true/false
 1164 test is the next element:

```

1165 [5 [1 0] 0 6]
1166

```

1168 This compares the quoted value 0 (from [1 0]) with the
 1169 value at memory slot 6 in the subject ([0 6]).

1170 What is the subject and what is at mem slot 6? This code is
 1171 the arm of a gate, so the subject is that gate/core: [battery
 1172 [sample payload]]. We are looking at the battery right now,
 1173 and so [0 6] yanks the head of the tail, or the sample!

1174 We know the sample will be a noun that we're testing for
 1175 loobeanness, so this code starts by seeing whether the sample
 1176 is the value 0/true. If it is, the next element is [1 0], so we
 1177 return 0 if the sample is 0.

1178 Otherwise, we run the second branch of the `if-else` con-
1179 ditional:

```
1180       [6 [5 [1 1] 0 6] [1 1] 0 0]
```

1183 This is also an opcode `6` if-else. Once again, it compares
1184 something to the value at mem slot 6 (the sample), but this
1185 time it checks whether that is the value 1. If it is, it runs the
1186 formula `[1 1]` to return 1.

1187 If not, it means our input was neither 0 nor 1 and is not a
1188 loobean, so we run the formula `[0 0]`, which always crashes
1189 (as there is no memory slot 0).⁶ This is exactly what we want—
1190 crash if the sample is not a loobean!

1191 3.8 Summary

1192 In this section, we’ve seen how to use almost all of the remain-
1193 ing opcodes, which build Nock up to a slightly more expres-
1194 sive level. We also saw how Hoon cores start to arise pretty
1195 naturally out of the Nock primitives, especially `8` and `9`. Then
1196 we walked through real production Nock code to show that
1197 everything we’ve learned so far works exactly as expected in
1198 the wild. In the next section, you’ll learn how to write real
1199 programs in Nock, and compose those programs to make new
1200 ones. Finally, hopefully you now see that, whatever its other
1201 limitations, Nock is not particularly obscurantist, and is fairly
1202 straightforward to parse, once you understand its syntax and
1203 idioms.

1204 4 Interlude

1205 In this section, we cover some loose ends deriving from the
1206 previous discussion.

⁶In the binary tree addressing scheme of Nock, 1 refers to the entire noun,
2 to its head if it exists, and 3 to its tail if it exists.

1207 4.1 Order of Operations

1208 At this point, we can answer the question of what order Nock
 1209 evaluates expressions in. The answer is straightforward: it
 1210 starts with code that has only one pseudocode operator and
 1211 all Nock code, as below:

```
1212 *[50 4 4 [5 [0 1] [1 50]]]
```

1215 The interpreter can then be thought of as moving left-to-right,
 1216 expanding according to its rules, until it can't expand any fur-
 1217 ther into pseudocode:

```
1218 *[50 4 4 [5 [0 1] [1 50]]]
1219 **[50 4 [5 [0 1] [1 50]]]
1220 ***[50 5 [0 1] [1 50]]
1221 +=[*[50 [0 1]] *[50 [1 50]]]
1222 +=[/[1 50] 50]
```

1225 Once we get to that last line, +=[/[1 50] *[50 [1 50]]],
 1226 no further expansion into pseudocode is possible.

1227 At this point, the interpreter expands “inside-out”, starting
 1228 from the deepest operators. In this case, those are /[1 50]
 1229 and 50, so it does those:

```
1230 +=[50 50]
```

1233 Then it moves to the next-furthest-inside operator: =, and then
 1234 to the + operators:

```
1235 +=[50 50]
1236
1237 ++(0)
1238 +(1)
1239 2
```

1241 4.1.1 Summary of Nock Order of Operations

- 1242 1. On the first pass, when we have Nock code `*(nock-code)`,
 1243 we expand left-to-right into pseudocode.
- 1244 2. On the second pass, when we've fully expanded pseu-
 1245 docode, we evaluate operators from the inside-out.

1246 4.2 11, Hints and Side Effects for the Interpreter

1247 Opcode 11 lets you compute side effects and pass informa-
 1248 tion to Nock's interpreter, for that interpreter to do what it
 1249 wants. The most common uses are things like jets and debug-
 1250 ging prints. In theory, this opcode is fairly dangerous, since it
 1251 tells the interpreter that it's free to do anything.

1252 There are two different versions of 11, depending on whether
 1253 the head following 11 is an atom or a cell

```
1254 :: head is an atom (b)
1255
1256 *[a 11 b c] *[a c]
1257
1258 :: head is a cell ([b c])
1259
1260 *[a 11 [b c] d]      *[[*[a c] *[a d]] 0 3]
```

1261 This pseudocode lets 11 handle two separate use cases:

- 1262 1. We just want the interpreter to process a message in its
 1263 own language/environment.
- 1264 2. We want the interpreter to compute some Nock code
 1265 with Nock semantics before processing the hint.

1266 4.2.1 Option #1: Interpreter Processes a Message

1267 Imagine the following Nock:

```
1268 . *([50 51] [11 369 0 2])
1269
1270
```

1271 This passes the value 369 to the interpreter, where maybe it
 1272 would be the key in a hashmap. The value associated with the
 1273 could be the function to debug print the whole operation. Or
 1274 the value could be a jet function that specifically knows how
 1275 to compute the 0 opcode faster when it's followed by a 2.

1276 After the interpreter does whatever, it needs to then discard
 1277 the value 292 . 989, and computes

```
1278 > . *([50 51] [0 2])
1279
1280 50
1281
```

1282 4.2.2 Option #2: Run Nock Code and then Process a Message

1283 In this case, instead of passing a static value to the interpreter,
 1284 we pass it a Nock computation against the subject. The result
 1285 of that computation will then be available to the interpreter to
 1286 use as a message.

1287 4.2.3 Practical Use of 11: Jet Registration

1288 Jet registration in Hoon usually looks like this:

```
1289 ++ my-function
1290 ~/ %my-function
1291 |= var= *
1292 ...
1293
1294
```

1295 ~/ “fassign” uses Nock 11 to pass the value %my-function to the
 1296 interpreter, where it is associated with the C code that produces
 1297 the product of my-function.

1298 4.3 Nock in Hoon

1299 In Hoon, you can acquire the Nock code for any function by
 1300 using the != “zaptis” rune. Since this produces Nock code com-
 1301 plete with gate calls, you may also find it helpful to cast a data
 1302 type as a noun using ^ - *, which will show you the raw Nock
 1303 code for that value.

1304 As you have seen, you can also evaluate raw Nock using
 1305 the .* “dottar” rune, which evaluates to Nock opcode 2. Other
 1306 Nock runes in Hoon include .+, which is opcode 4; .=, which
 1307 is opcode 5; and .?, which is opcode 3. Fake opcode 12 is not
 1308 actually Nock, but is used in a virtualization context to resolve
 1309 values that may not be available in the current subject.

1310 5 The Core as Design Pattern

1311 At this point, you now know everything there is to know about
 1312 Nock syntax and basic code manipulation. However, if I asked
 1313 you to go write a simple program in Nock, you’d probably have

1314 a hard time getting started. It's really helpful to see some ex-
 1315 amples of how Nock programs are designed and patterns that
 1316 tend to recur in them.

1317 We can set variables to formulas in the Urbit Dojo. We will
 1318 use that in this lesson to make code easy to follow, and to show
 1319 how Nock programs are made out of discrete chunks.

```
1320
1321 > =increment-formula [4 @ 1]
1322 > .* (50 increment-formula)
1323 51
1324 > =increment-formula
1325
```

1326 5.1 Building a Core Manually

1327 Every Nock program is just a formula that takes in a subject
 1328 (argument). In order to run full programs against a subject, we
 1329 follow the following pattern:

- 1330 1. First pass: set up the dominos inside an opcode like 2
 1331 (or 7/8/9). This will create a core, and evaluate it on the
 1332 second pass.
- 1333 2. Second pass: evaluate that core to get our result.

1334 The pseudocode for opcode 2, this looks like:

```
1335 *[subject 2 core-formula call-formula]
1336
1337
```

1338 Almost always, however, we'll use the 9 opcode (which is
 1339 sugar on top of 2) to actually start our programs running, since
 1340 it supports the concept of “cores” and “arms” very naturally.
 1341 (For those who know Hoon, this “set up a core and run it” pat-
 1342 tern should feel similar to | ^ or =>.)

1343 5.1.1 Example: A Simple Increment Gate

1344 Let's take code to increment the subject and turn it into a Hoon-
 1345 style gate (a core with one arm and a sample). The code itself
 1346 is simple: [4 @ 1].

1347 Our code expects the subject to be an atom, but Hoon gates
 1348 expect their subject to be the gate itself, which has the form

1349 [battery payload]. So we need to adjust our formula to ac-
1350 cept this subject format.

```
1351 > =inc [7 [0 3] [4 0 1]]
1352
1353 :: verify that it increments value in the subject's
1354     tail (payload)
1355 > .*([1 74] inc)
1356 75
1357
```

1358 We put our formula in the head, and put a default (“bunted”)
1359 sample of 0 as the payload.

```
1360 > =inc-gate [inc 0]
1361 > inc-gate
1362 [[7 [0 3] 4 0 1] 0]
1363
1364
```

1365 Arms are best invoked using the 9 opcode. Here we use the
1366 “standard” version of it: [9 2 0 1]. Recall that 9 is $\star[a\ 9\ b\ c]$, where c is the “new subject” formula, and b is the memory
1367 slot of that new subject to take an arm from. Here we use [0
1368 1] to keep our subject (the gate) unchanged.

```
1369 > .*(inc-gate [9 2 0 1])
1370
1371 1
1372
1373
```

1374 To “pass a parameter” to our gate, we use the 10 opcode to
1375 edit the subject (the inc-gate core) and replace the payload
1376 with a value. We make our subject [inc-gate val-to-increment]:

```
1377 :: notice how opcode 10 can replace the last element
1378     of inc-gate
1379 :: the tail of inc-gate was 0; now it's 36
1380 > .*([inc-gate 36] [10 [3 [0 3]] 0 2])
1381 [[7 [0 3] 4 0 1] 36]
1382
1383
1384 :: now use that to set up the subject with a new
1385     tail (sample)
1386 :: and call arm in memory slot 2 (our increment
1387     formula)
1388 > .*([inc-gate 36] [9 2 10 [3 [0 3]] 0 2])
1389 37
1390 > .*([inc-gate 562] [9 2 10 [3 [0 3]] 0 2])
1391 563
1392
```

1393 5.1.2 Summary

1394 What was the point of this? We took a perfectly good incre-
1395 ment formula that worked on atoms, and made it more com-
1396 plicated for the same result. *Lame!*

1397 In fact, however, this setup will come in really handy when
1398 we have multiple arms in our core, not just one. We've made
1399 it easy to set up a clean environment for increment to find its
1400 sample in, no matter what other data is present. This will come
1401 in *very* handy in our third example program.

1402 First, however, let's look at putting a more complicated for-
1403 mula inside a gate. (Hint: the process is the same).

1404 5.1.3 Example: A Decrement Gate

1405 Here we take a piece of code that decrements the subject when
1406 the subject is an atom, and turn it into a gate. For now, we
1407 will take it as a given that the decrement function given below
1408 works. (The code will crash when the input is a cell or 0)

```
1409 :: the code: set a Dojo face =dec
1410 > =dec [6 [5 [0 1] [1 0]] [0 0] [6 [3 0 1] [0 0] [8
1411       [1 0] 8 [1 6 [5 [0 7] 4 0 6] [0 6] 9 2 [0 2] [4 0
1412       6] 0 7] 9 2 0 1]]]
1413
1414
1415 :: testing in the Dojo
1416 > .* (100 dec)
1417 99
1418
1419 :: with non-atom input we crash
1420 .* ([100 101] dec)
1421 dojo: hoon expression failed
```

1423 Let's re-jigger the decrement formula to accept the subject
1424 in format [battery payload]:

```
1425 :: gets value in mem slot 3 and applies the
1426 decrement formula to it
1427 > =dec-arm [7 [0 3] dec]
1428
1429
1430 :: verify that dec-arm decrements the value in the
1431 tail
```

```

1433 > .*([1 88] dec-arm)
1433 87
1434
1435 :: output the full dec-arm formula code
1436 > dec-arm
1436 [7 [0 3] 6 [5 [0 1] 1 0] [0 0] 6 [3 0 1] [0 0] 8 [1
1438 0] 8 [1 6 [5 [0 7] 4 0 6] [0 6] 9 2 [0 2] [4 0 6]
1439 0 7] 9 2 0 1]
1440

```

Next, assemble the core:

```

1441
1442
1443 :: all we need to do is add the sample! (we give it a
1444 default value of 0)
1445 > =dec-gate [dec-arm 0]
1446 > dec-gate
1447 [[7 [0 3] 6 [5 [0 1] 1 0] [0 0] 6 [3 0 1] [0 0] 8 [1
1448 0] 8 [1 6 [5 [0 7] 4 0 6] [0 6] 9 2 [0 2] [4 0 6]
1449 0 7] 9 2 0 1] 0]
1450 [[7 [0 3] 6 [5 [0 1] 1 0] [0 0] 6 [3 0 1] [0 0] 8 [1
1451 0] 8 [1 6 [5 [0 7] 4 0 6] [0 6] 9 2 [0 2] [4 0 6]
1452 0 7] 9 2 0 1] 0]
1453

```

As in the first example, we use the 10 opcode to edit the subject (the `dec-gate` core) and replace the payload with a value. We make our subject `[dec-gate val-to-decrement]`.

```

1454
1455
1456
1457
1458 :: confirm that our 10 code places 36 in the tail
1459 > .*([dec-gate 36] [10 [3 [0 3]] 0 2])
1460 [[7 [0 3] 6 [5 [0 1] 1 0] [0 0] 6 [3 0 1] [0 0] 8 [1
1461 0] 8 [1 6 [5 [0 7] 4 0 6] [0 6] 9 2 [0 2] [4 0 6]
1462 0 7] 9 2 0 1] 36]
1463
1464 :: now use that to set up the subject with a new
1465 tail (sample)
1466 :: and call arm in memory slot 2 (our decrement
1467 formula)
1468 > .*([dec-gate 36] [9 2 10 [3 [0 3]] 0 2])
1469 35
1470 > .*([dec-gate 562] [9 2 10 [3 [0 3]] 0 2])
1471 561
1472

```

1473 5.1.4 Example: Comparing Two Numbers

1474 In this example we will build a Nock core that compares two
 1475 numbers and returns whether they are greater than, less than,
 1476 or equal to each other.

1477 The program specification is to take two numbers a and b
 1478 and return:

- 1479 • 0 if $a = b$.
- 1480 • 1 if $a > b$.
- 1481 • 2 if $a < b$.

1482 The trick is that the only mathematical operators we have
 1483 for this are the 4 opcode for increment, 5 for equals, and a
 1484 decrement Nock function that we will supply. The basic algo-
 1485 rithm is:

- 1486 • If $a = b$, return 0.
- 1487 • If $a = 0$, return 2. (I.e., $a < b$).
- 1488 • If $b = 0$, return 1. (I.e., $a > b$).
- 1489 • Recur with a and b both decremented.

1490 The above works because if a is smaller than b , it hits 0
 1491 first. If b is smaller than a , it hits 0 first. If they are equal, we
 1492 never decrement in the first place, so there are no issues with
 1493 numbers going negative.

1494 We will construct a core that has two arms: one arm with
 1495 the algorithm logic, and a second arm with the decrement gate
 1496 from the second example. These arms will live in the head of
 1497 the core (think “battery”).

1498 We will put our variables a and b in the tail of the core
 1499 (think “payload”). They will be updated before the core is called
 1500 each time. (This is similar to a trap or door in Hoon).

1501 The final core we will create will have the structure `[battery`
 1502 `payload]`, which can be broken down as:

```
1503 [[main-logic dec-gate] a b]
```

1506 Once the core is set up, we'll invoke the algorithm logic arm
 1507 to set it running. We will build the main logic “inside-out”, by
 1508 first defining the “recur” code, and then inserting it into our
 1509 if/else tests that implement the algorithm.

1510 This recursion logic assumes the decrement gate lives in the
 1511 tail of the battery (the battery is the head), i.e. [0 5]. We are
 1512 using [10 [3 [0 memory-slot]] 0 5] to extract the decre-
 1513 ment gate from our core, and then we invoke it with [9 2
 1514 ...].

1515 This is identical to what we did in the second example—the
 1516 only difference is that we now do it twice: once for memory
 1517 slot 6 (a) and again for memory slot 7 (b).

```

1518 > =dec-a [9 2 10 [3 [0 6]] 0 5]
1519 > =dec-b [9 2 10 [3 [0 7]] 0 5]
1520
1521
1522 :: we can test our formulas by making a
1523 :: dummy core (a = 33, b = 77)
1524 :: first formula in battery just returns
1525 :: the current subject
1526 > =dummy-core [[0 1] dec-gate] 33 77]
1527 > .*(dummy-core dec-a)
1528 32
1529 > .*(dummy-core dec-b)
1530 76
1531
```

1532 The recursion formula here operates by invoking its subject—
 1533 itself—and applying the dec-a and dec-b formulas to it. Then
 1534 we use opcode 9 for the new subject's setup: replace payload
 1535 with new a and b run the current core

```

1536
1537 > =recur [9 4 [[0 2] dec-a dec-b]]
1538
1539 :: test it -- we want to see same battery,
1540 :: with payload being replaced with 32 and 76
1541 > .*(dummy-core recur)
1542 [[0 1] [7 [0 3] 6 [5 [0 1] 1 0] [0 0] 6 [3 0 1] [0
1543       0] 8 [1 0] 8 [1 6] [5 [0 7] 4 0 6] [0 6] 9 2 [0 2]
1544       [4 0 6] 0 7] 9 2 0 1] 0] 32 76]
1545
1546 :: clean up the dummy-core
1547 > =dummy-core
1548
```

The main logic is then:

```

1549
1550
1551 :: check whether a is 0; return 2 if true
1552 :: else check whether b is 0; return 1 if true
1553 :: else recur
1554 > =main-logic [6 [5 [0 6] [1 0]] [1 2] [6 [5 [0 7] [1
1555     0]] [1 1] recur]]
1556
1557 :: test (we aren't doing the equality case here)
1558 :: a < b
1559 > .*([[main-logic dec-gate] 9 10] [9 4 0 1])
1560 2
1561 :: a > b
1562 > .*([[main-logic dec-gate] 10 9] [9 4 0 1])
1563 1
1564

```

We build an outer test for whether $a=b$, and then if it's not, we use the 1 opcode to return the core.

```

1565
1566
1567 > =battery [main-logic dec-gate]
1568
1569
1570 :: payload for our core is memory slots 2 and 3 (a
1571     and b)
1572 > =comparison [6 [5 [0 2] [0 3]] [1 0] [9 4 [[1
1573     battery] [0 2] 0 3]]]
1574
1575 :: test our 3 cases
1576 > .*([0 8] comparison)
1577 2
1578 > .*([0 0] comparison)
1579 0
1580 > .*([8 0] comparison)
1581 1
1582

```

We can take this code and use it *anywhere* that our subject is a cell of two numbers, and it will “just work”.

```

1583
1584
1585 [6 [5 [0 2] 0 3] [1 0] 9 4 [1 [6 [5 [0 6] 1 0] [1 2]
1586     6 [5 [0 7] 1 0] [1 1] 9 4 [0 2] [9 2 10 [3 0 6] 0
1587     5] 9 2 10 [3 0 7] 0 5] [7 [0 3] 6 [5 [0 1] 1 0]
1588     [0 0] 6 [3 0 1] [0 0] 8 [1 0] 8 [1 6 [5 [0 7] 4 0
1589     6] [0 6] 9 2 [0 2] [4 0 6] 0 7] 9 2 0 1] 0] [0 2]
1590     0 3]
1591
1592

```

1593 One common convention used in languages like Hoon is
1594 to denote Nock rules as constants, e.g. %0 (corresponding to
1595 our 0). This makes it much more straightforward to interpret
1596 hand-rolled Nock code with addresses and constants present.

1597 6 Conclusion

1598 After this tutorial, you should be prepared to interpret Nock
1599 code when it is produced by Hoon. Although there is only
1600 rarely any call to write Nock code directly, you should now
1601 have a good understanding of how to read and interpret it. As
1602 you write your own interpreters in the future, you may even
1603 have call to hand-roll Nock expressions on occasion.