Deriving Nock Opcodes 6–11

Thomas Lindstrom-Vautrin ~niblyx-malnus
Groundwire Syndicate

Abstract

Since Nock opcodes 0 through 5 are Turing complete, it should be possible to derive the operations of the remaining opcodes 6 through 11 from them. In this article, we expound on this idea and provides a detailed derivation of each opcode.

Contents

Urbit Systems Technical Journal II:1 (2025): 47–70.

Address author correspondence to ~niblyx-malnus.

1 Motivation

It is a fun exercise when learning Nock to try to derive opcodes 6 through 11 with opcodes 0 through 5. We know (or suspect) that this is possible since Nocks 0 through 5 are billed as a Turing-complete ruleset. In fact, Nocks 6 through 9 and Nock 11 are quite trivial (of these, Nock 6 is the most involved) since they are defined in terms of the `*` tar expression evaluator and Nocks 0 through 5. It is still instructive to write these entirely as nouns which can be evaluated against some subject using the Hoon `.*` dottar rune. (In the following, take `*[a ^]` to be the same as `.*(a ^)`.)

Nock 10 is not expressed in this way, however. It is expressed in terms of the # hax edit operator:

```
*[a 10 [b c] d] ==  #[b *[a c] *[a d]]
```

While opcode 10 was introduced in the decrement from Nock 5K to Nock 4K, it is still fundamentally based on the principles established by the earlier opcodes. Ultimately, we can express # hax in terms of opcodes 0 through 5, but we will find that it is not trivial. In fact, we have a hint as to how to do so in the definition of the # hax edit operator.

```
#[1 a b] ============ a
#[(a + a) b c] ====== #[a [b /[(a + a + 1) c]] c]
#[(a + a + 1) b c] == #[a [/[(a + a) c] b] c]
```

The # hax edit operator is defined in terms of the `/` fas slot operator, whose effect we can reproduce with base Nock opcodes using Nock 0. As long as we have a way to keep track of whether `a` is odd or even, we should be able to write this code in simple Nock. We will use this insight to create our own Nock 10 from scratch.

This process will be easier using Nock opcodes 6, 7, 8, and 9, however. So let us convince ourselves that these can be expressed as nouns composed of only Nocks 0 through 5.

2 Deriving Opcode 7

We will start with Nock 7 because it is the easiest:[1]

```
*[a 7 b c] == *[*[a b] c]
```

which looks a lot like Nock 2:

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

In fact, opcode 7 is basically just a glorified Nock 2 in order to allow for more direct function composition. We can easily see that if we replace our `c` from Nock 2 with a `[1 c]` we get:

```
*[a 2 b 1 c] == *[*[a b] *[a 1 c]] == *[*[a b] c]
```

Therefore `*[a 7 b c]` and `*[a 2 b 1 c]` are equivalent.

Nock 7 Primitive Equivalent

```
*[a 7 b c] == *[a 2 b 1 c]
```

---

3 Deriving Opcode 8

Nock 8 is also fairly straightforward.

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

This extends the noun by pinning another noun `*[a b]` to its head, and then runs a formula `c` on this new noun. We notice two things here. One is function composition. First we add a noun to the head. Then we apply a formula to the new noun. This suggests Nock 7 and thus Nock 2. We also notice the creation of a cell from two nouns.

Recall the rule:

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

---

[1] I use `==` and `:=` as equality and assignment operators, respectively, which are not to be confused with Hoon runes or terminators.

`[*[a b] a]` can be rewritten `[*[a b] *[a 0 1]]` which can be rewritten as `*[a b 0 1]`.

By substituting back into our original formula we get:

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

Nock 8 Primitive Equivalent

```
*[a 8 b c] == *[a 2 [b 0 1] 1 c]
```

4 Deriving Opcode 9

Nock 9 is the "function invocation" opcode. It performs some function `c` on the subject `a` and then extracts the function at slot `b` from this new subject and runs that function against that same new subject.

```
*[a 9 b c] == *[*[a c] 2 [0 1] 0 b]
```

Much of the work here is already done for us. We have a formula `[2 [0 1] 0 b]` being applied to the result of a formula `c` being applied to subject `a`. This is just Nock 7 `*[a 7 c 2 [0 1] 0 b]` which is just Nock 2 `*[a 2 c 1 2 [0 1] 0 b]`.

Nock 9 Primitive Equivalent

```
*[a 9 b c] == *[a 2 c 1 2 [0 1] 0 b]
```

As an aside, we can note that the Nock 9 is defined in terms of a combination of pseudocode operators and Nock opcodes. We can also go the other way to see what Nock 9 looks like in pure pseudocode operators.

```
*[a 9 b c]
== *[*[a c] 2 [0 1] 0 b]
== *[*[*[a c] 0 1] *[*[a c] 0 b]]
== *[*[a c] /[b *[a c]]]
```

Nock 9 Pseudocode Operator Equivalent

```
*[a 9 b c] == *[*[a c] /[b *[a c]]]
```

5 Deriving Opcode 6

Of all the opcodes so far, Nock 6 is the most complex. It is the if-then-else operator in Nock. It takes a function `b` which computes a loobean `*[a b]` on subject `a` and returns `*[a c]` if the loobean is yes (`0`) and `*[a d]` if the loobean is no (`1`).

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

Instead of working backwards from this formula, let's derive it ourselves. First, let's select `c` or `d` from a cell `[c d]` based on whether `*[a b]` is `0` or `1`. If `*[a b]` is `0` we want to slot into the head of `[c d]` at address `2` and if `*[a b]` is `1` we want to slot into the tail of `[c d]` at address `3`. With `*[a 4 4 b] == ++*[a b]` we get address `2` from loobean `0` and address `3` from loobean `1`. To slot into `[c d]` let's simply do `*[[c d] 0 *[a 4 4 b]]`. This gives us either `c` or `d` which we then want to apply to `a`:

```
*[a *[[c d] 0 *[[a 4 4 b]]]]
```

We are almost there, but where does the `*[2 3] 0 *[a 4 4 b]` come from? Isn't slotting into `[2 3]` with `2` or `3` from `*[a 4 4 b]` redundant? We get the same value back. However, if for some reason `*[a b]` is not a loobean but is still an atom, meaning it is an integer greater than `1` and if `[c d]` is a noun with many nested cells, `*[a 4 4 b]` will return an address greater than `3` which could still be a valid address. `*[[2 3] 0 *[a 4 4 b]]` forces this to return either `2` or `3` or to crash.

Thus

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

or

```
if *[a b] then *[a c] else *[a d]
```

Converting Opcode 6 to Opcodes 0 through 5

To convert this to a single noun is straightforward enough, but it's a bit of a grind. To keep things as clean as possible, like in timluc-miptev's piece,[2] we will use variables in the dojo. Here is where we start:

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

Let's start at the end. In order to construct `[0 *[a 4 4 b]]` on subject `a`, let's notice that it is a cell. When trying to construct a cell with respect to a subject `a` we want to put it in the form `[*[a m] *[a n]]` so that we can recover the original expression `*[a m n]`. The tail of our cell is already in the form `*[a n]`. So how about the head? `*[a 1 0] == 0`. So we can write:

```
[0 *[a 4 4 b]] == [*[a 1 0] *[a 4 4 b]] == *[a [1 0] 4 4 b]
```

Let's say `=x [[1 0] 4 4 b]`. This gives us:

```
[0 *[a 4 4 b]] == *[a x]
```

Excellent. We have started to simplify our expression. We now have:

```
*[a 6 b c d] == *[a *[[c d] 0 *[[2 3] *[a x]]]]
```

Hopefully you see how can we can repeat this process. Let's look at `*[[2 3] *[a x]]`. This has nested expression evaluations. This suggests Nock 2. For Nock 2, we want to get our expression in the form `*[*[a m] *[a n]]` so that we can recover the original expression `*[a 2 m n]`. (This looks like what we did in our first step, but don't be fooled. In the first step we were building a cell. In this step, we are building an expression evaluation.) Notice that `[2 3] == *[a 1 2 3]`. This gives us:

```
*[[2 3] *[a x]] ==
  *[*[a 1 2 3] *[a x]] ==
  *[a 2 [1 2 3] x]
```

Let's say `=y [2 [1 2 3] x]`. This gives us:

[2] See ~timluc-miptev, pp. 1–45 in this volume.

```
*[[2 3] *[a x]] == *[a y]
```

We now have:

```
*[a 6 b c d] == *[a *[[c d] 0 *[a y]]]
```

Let's look at `[0 *[a y]]`. Another cell. We know what to do here.

```
[0 *[a y]] == [*[a 1 0] *[a y]] == *[a [1 0] y]
```

So we now have:

```
*[a 6 b c d] == *[a *[[c d] *[a [1 0] y]]]
```

`*[[c d] *[a [1 0] y]]` has nested expression evaluations. We know what to do here.

```
*[[c d] *[a [1 0] y]] ==
  *[*[a 1 c d] *[a [1 0] y]] ==
  *[a 2 [1 c d] [1 0] y]
```

This is starting to get a little verbose again. Let's substitute: `=z [2 [1 c d] [1 0] y]`. This gives us:

```
*[[c d] 0 *[[2 3] 0 *[a 4 4 b]]] ==
  *[[c d] 0 *[a y]] == *[a z]
```

So now we have:

```
*[a 6 b c d] == *[a *[a z]]
```

More nested evaluation expressions.

```
*[a *[a z]] == *[*[a 0 1] *[a z]] == *[a 2 [0 1] z]
```

Finally, we have:

```
*[a 6 b c d] == *[a 2 [0 1] z]
```

And that's the whole expression. Let's substitute back in:

```
*[a 2 [0 1] z] == *[a 2 [0 1] 2 [1 c d] [1 0] y] ==
  *[a 2 [0 1] 2 [1 c d] [1 0] 2 [1 2 3] x] ==
  *[a 2 [0 1] 2 [1 c d] [1 0] 2 [1 2 3] [1 0] 4 4 b]
```

This is Nock 6 written with only Nock 0 through 5.

Nock 6 Primitive Equivalent

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

As an aside we can notice that Nock 6 (like Nock 9) is defined as a combination of pseudocode operators and Nock opcodes. We can also go the other way to see what Nock 9 looks like in pure pseudocode operators.

```
*[a 6 b c d] ==
  *[a *[[c d] 0 *[[2 3] 0 *[a 4 4 b]]]] ==
  *[a *[[c d] 0 *[[2 3] 0 ++*[a b]]]] ==
  *[a *[[c d] 0 /[++*[a b] [2 3]]]] ==
  *[a /[/[++*[a b] [2 3]] [c d]]]
```

Nock 6 Pseudocode Operator Equivalent

```
*[a 6 b c d] == *[a /[/[++*[a b] [2 3]] [c d]]]
```

This is not quite as intuitive as the Nock 9 pseudocode operator equivalent. The definition of Nock 6 is mostly to demonstrate the very cool fact that a branching operation can be derived using only the fundamental operators `*` tar, `/` fas and `+` lus. But we can also look at Nock 6 another way by introducing (in our own mental fork of the Nock spec) a new pseudocode operater `<` gal.

Nock 6 New Pseudocode Operator

```
<[0 a b]            a
<[1 a b]            b
<a                  <a

*[a 6 b c d]        <[*[a b] *[a c] *[a d]]
```

This is equivalent to the original definition. Like # hax, this pseudocode operator is not required for Turing completeness. `*` tar, `?` wut, `+` lus, `=` tis, and `/` fas are enough for this.

6 Deriving Opcode 11

Even though we don't need Nock 11 to build Nock 10, for the sake of our collective mania, let's note that Nock 11 can also trivially be written as a noun. The definition of Nock 11 is:

```
*[a 11 [b c] d] == *[[*[a c] *[a d]] 0 3]
*[a 11 b c] ====== *[a c]
```

The hint provided by this code is used in the process of jetting. In the first case, an atomic tag `b` is provided and a formula `c` is used to compute additional hint information on the subject. `d` is the formula we are actually trying to compute. In the second case, only the atomic tag `b` is passed and `c` is the formula we are actually trying to compute.

It is tempting to try to reduce the first case to `*[a d]`. However, this reduction is incorrect. You may have a case where `*[a c]` crashes but `*[a d]` does not. In this case, a correct interpreter will crash the whole expression if `*[a c]` crashes, whereas an incorrect interpreter will proceed with `*[a d]`. Thus, all correct interpreters must try to compute `*[a c]`.

```
*[a 11 [b c] d] == *[[*[a c] *[a d]] 0 3]
== *[*[a c d] *[a 1 0 3]]
== *[a 2 [c d] 1 0 3]

*[a 11 b c] == *[a c]
== *[*[a 0 1] *[a 1 c]]
== *[a 2 [0 1] 1 c]
```

Nock 11 Primitive Equivalent

```
*[a 11 [b c] d] == *[a 2 [c d] 1 0 3]
*[a 11 b c] ====== *[a 2 [0 1] 1 c]
```

Nock 11 Pseudocode Operator Equivalent

```
*[a 11 [b c] d] == /[3 [*[a c] *[a d]]]
*[a 11 b c] ====== *[a c]
```

7 Explaining Opcode 10

For convenience, let's restate Nock 10 and the # hax edit operator.

```
*[a 10 [b c] d] ==  #[b *[a c] *[a d]]

#[1 a b] ============ a
#[(a + a) b c] ====== #[a [b /[(a + a + 1) c]] c]
#[(a + a + 1) b c] == #[a [/[(a + a) c] b] c]
```

Nock 10 is defined in terms of the # hax edit operator. So let's derive this first.

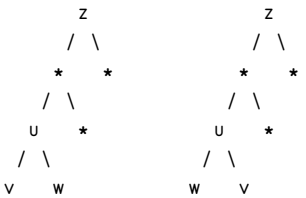
7.1 The # edit operator

Let's define # hax as a function of three inputs, `x`, `y`, and `z`. This says replace what's in slot `x` of `z` with `y`. In fact, let's rewrite the pseudocode with these inputs.

```
#[(x := 1) y z] ========= y
#[(x : x is even) y z] == #[x/2 [y /[x+1 z]] z]
#[(x : x is odd) y z] === #[(x-1)/2 [/[x-1 z] y] z]
```

What are we doing here? Let's first note that there is really no way for us to "edit" a noun. Really what we are doing is producing a new noun which is exactly the same as the old noun except in the specified way. If `x` is `1`, we are slotting into the root address of the noun `z` and replacing it with `y`, so we can get our "edited" noun simply by returning `y`. For any `x` greater than `1`, however, we have to create an entirely new noun. But creating complex nouns from scratch is hard. Creating a single *cell*, on the other hand, is easy.

To make this easier to understand, let's say `v := /[x z]`. `v` is the noun we will be replacing with `y`; the current occupant of slot `x` in noun `z`. This noun `v` has a "sibling noun" which we will call `w`. `v` is either the head or the tail of its "parent noun" `u`. Below, we diagram the two possible cases.



While we cannot create a modifed `z` in one step, we can create a modified `u` in one step. `u == [v w]` or `u == [w v]`. Since we are replacing `v` with `y`, our modified `u` looks like `u' == [y w]` in the first case and `u' == [w y]` in the second case. We already have `y`. We need to figure out how to get `w` and how to replace `u`. This requires the addresses of both. We diagram this below.

```
nouns          slots
-----          -----
  u             x/2
 / \            / \
v   w          x  x+1

  u           (x-1)/2
 / \            / \
w   v          x-1  x
```

If `x` is even, `w` is in slot `x+1` of `z` and `u' == [y /[x+1 z]]` can be slotted into slot `x/2` of `z`. If `x` is odd, `w` is in slot `x-1` of `z` and `u' == [/[x-1 z] y]` can be slotted into slot `(x-1)/2` of `z`. A more concise way to say this would be:

```
#[(x : x is even) y z] == #[x/2 [y /[x+1 z]] z]
#[(x : x is odd) y z] === #[(x-1)/2 [/[x-1 z] y] z]
```

This operation will recur, and since the slot is always getting smaller, we will eventually get to the base case of `x == 1` at which point we will return our new "edited" noun.

Now let's translate this into Nock. *To perform this translation, we are going to make liberal use of Dojo variables to keep things relatively simple and modular.*

We are trying to recreate the # hax edit operator in a function called # hax which will take a subject `[x y z]`. We should be able to run:

```
*[[x y z] hax] == #[x y z]
```

Let's crash immediately if x is a cell. This will make use of Nock 6. If `x-is-cell`, then `crash`; else `check-x-0` (we will proceed to check another degenerate case):

```
*[[x y z] 6 x-is-cell crash check-x-0]
```

For the sake of our OCD, we can start to convert this to opcodes 0 through 5 using our definition:

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

```
=hax [2 [0 1] 2 [1 crash check-x-0]
                [1 0]
                2 [1 2 3] [1 0] 4 4 x-is-cell]
```

To check if x is a cell, we use Nock 3:

```
*[[x y z] 3 0 2]
=x-is-cell [3 0 2]
```

To crash, we can simply try to slot into non-existing address `0`:

```
*[[x y z] 0 0]
=crash [0 0]
```

We should also crash if `x == 0`:

```
*[[x y z] 6 x-is-0 crash launch]
=check-x-0 [2 [0 1] 2 [1 crash launch]
            [1 0] 2 [1 2 3] [1 0] 4 4 x-is-0]
```

To check if `x == 0`, we use Nock 5:

```
*[[x y z] 5 [0 2] 1 0]
=x-is-0 [5 [0 2] 1 0]
```

In order to make recursive calls, we are going to want to store our formula in the head of our subject so that we can access it from within the formula.

```
[hax-formula x y z]
```

We can do this with Nock 8 and Nock 9. With Nock 8, we will pin `hax-formula` to the head of the subject and then call this head on the modified subject using Nock 9.

```
*[[x y z] 8 [1 hax-formula] call-head]
```

Recall that `*[a 8 b c] == *[a 2 [b 0 1] 1 c]`.

```
=launch [2 [[1 hax-formula] 0 1] 1 call-head]
```

To call the head of the modified subject on itself:

```
*[[hax-formula x y z] 9 2 0 1]
```

Recall that `*[a 9 b c] == *[a 2 c 1 2 [0 1] 0 b]`.

```
*[[hax-formula x y z] 9 2 0 1] ==
  [[hax-formula x y z] 2 [0 1] 1 2 [0 1] 0 2]
```

Notice that this simplifies to:

```
=call-head [2 [0 1] 0 2]
```

Now let's develop `hax-formula`. The subject we are now dealing with has the form:

```
[hax-formula x y z]
```

- `hax-formula` is at slot `2` (the head).
- `x` is at slot `6` (the head of the tail).
- `y` is at slot `14`.
- `z` is at slot `15`.

Let's deal with the first real branch. If `x == 1`, then we return `y`; else, we `recur-hax`:

```
*[[hax-formula x y z] 6 x-is-1 return-y recur-hax]
=hax-formula [2 [0 1] 2 [1 return-y recur-hax]
                        [1 0]
                        2 [1 2 3] [1 0] 4 4 x-is-1]
=x-is-1 [5 [0 6] 1 1]
=return-y [0 14]
```

Excellent. Now we are getting to the recursive part of our algorithm. We have already dealt with the base case. Now in the non-base case we want to call `hax-formula` on new parameters `[new-x new-y new-z]`. Let us imagine that there is a function `new-params` which takes subject `[x y z]` and

returns `[new-x new-y new-z]`. We can write this function later.

First, we want to create a cell:

```
[hax-formula new-x new-y new-z] ==
  [hax-formula *[[x y z] new-params]]

*[[hax-formula x y z] [0 2] 2 [0 3] 1 new-params]
=new-cell [[0 2] 2 [0 3] 1 new-params]
```

And we want to call the head of this new cell on itself:

```
*[[hax-formula x y z] 9 2 new-cell]
=recur-hax [2 new-cell 1 2 [0 1] 0 2]
```

When trying to extract new parameters from `[x y z]`, we come to another branch. We know at this point that `x` is not a cell and `x > 1`. Now we need to check if `x` is even:

- If `x` is even, return the following parameter updates:

  ```
  x := x/2
  y := [y /[x+1 z]]
  z := z
  ```

- Else (if `x` is odd), return the following parameter updates:

  ```
  x := (x-1)/2
  y := [/[x-1 z] y]
  z := z
  ```

So there are basically three things we need. We need a decremented `x`, `x-1`. We need a loobean which returns `0` if `x` is even and `1` otherwise (if `x` is odd; we have already dealt with all other cases). And we need the floor of half of `x`—the "parent address" of `x`.

We can actually get all of these at the same time. By using a modified version of the decrement algorithm, we can count up to `x-1` using some `counter` variable, keep track of whether `counter+1` is even or odd in variable `iseven` and keep track of the "parent address" of `counter+1` in variable `floorhalf`. When we get to `counter == x-1` we will have `x-1`, whether `x` is even, and the floor of half of `x`.

Because we are dealing with `x` values greater than `1`, the initial value of `counter` is `1`. Since the initial value of `counter+1` is `2` (the lowest possible value of `x`), the initial value of `iseven` is `0` and the initial value of `floorhalf` is `1`.

Here is what this looks like counting up to 10:

```
                    -------
                    x == 10
                    -------
+(counter)  2  3  4  5  6  7  8  9 10
counter     1  2  3  4  5  6  7  8  9
iseven      0  1  0  1  0  1  0  1  0
floorhalf   1  1  2  2  3  3  4  4  5
```

Let's look at the format of our function. Our subject is `[x y z]`. We are going to pin `[counter iseven floorhalf]` to the head of this noun and then our `params-formula` for recursion to the head of that. So by the time our recursion starts, our subject will look like this:

```
[params-formula [counter iseven floorhalf] x y z]
```

To pin `[[counter == 1] [iseven == 0] [floorhalf == 1]]` to the head of subject `[x y z]`, we run:

```
*[[x y z] 8 [1 1 0 1] pin-formula] ==
  *[[[1 0 1] x y z] pin-formula]
=new-params [2 [[1 1 0 1] 0 1] 1 pin-formula]
```

Next, we want to pin our `params-formula` to the head of this new subject:

```
*[[[1 0 1] x y z] 8 [1 params-formula] call-head] ==
  *[[params-formula [1 0 1] x y z] call-head]
=pin-formula [2 [[1 params-formula] 0 1] 1 call-head]
```

Finally, we want to run this `params-formula`, which is in the head of this new subject, against the whole subject. We actually already created a function which accomplishes this. We called it `call-head`.

```
call-head == [2 [0 1] 0 2]
```

Okay. Now we're ready to develop `params-formula`, which operates on subject:

```
[params-formula [counter iseven floorhalf] x y z]
```

(To save space we will write this as `[pf [c i f] x y z]`.)

- `params-formula` is at slot `2` (the head).
- `counter` is at slot `12` (the head of the head of the tail).
- `iseven` is at slot `26`.
- `floorhalf` is at slot `27`.
- `x` is at slot `14`.
- `y` is at slot `30`.
- `z` is at slot `31`.

Our formula should check if `counter+1 == x`. If this is the case, we have all the necessary data and we can `return-params`. Else, we should increment `counter`, flip `iseven` from `0` to `1` or from `1` to `0`, increment `floorhalf` only if `iseven == 1`, and then run `params-formula` again. You should be able to convince yourself that this works.

```
*[[pf [c i f] x y z] 6 is-at-x return-params recur-pf]
=params-formula [2 [0 1] 2 [1 return-params recur-pf]
                           [1 0]
                           2 [1 2 3]
                             [1 0]
                             4 4 is-at-x]

=is-at-x [5 [0 14] 4 0 12]
```

To return parameters from `[pf [c i f] x y z]` we need to branch on `iseven`.

```
*[[pf [c i f] x y z] 6 iseven even-params odd-params]
=iseven [0 26]
=return-params [2 [0 1] 2 [1 even-params odd-params]
                          [1 0] 2 [1 2 3]
                                  [1 0] 4 4 iseven]
```

Recall that if `x` is even, return the following parameter updates:

```
x := x/2
y := [y /[x+1 z]]
z := z
```

else (if `x` is odd), return the following parameter updates:

```
x := (x-1)/2
y := [/[x-1 z] y]
z := z
```

This translates to the logic: If `iseven` is `0`, return:

```
[floorhalf [y /[+(x) z]] z]
```

else, return:

```
[floorhalf [/[counter z] y] z]
```

Let's try to build these cells:

```
+(x) == *[[pf [c i f] x y z] 4 0 14]
counter == *[[pf [c i f] x y z] 0 12]
z == *[[pf [c i f] x y z] 0 31]

/[+(x) z] == *[z 0 +(x)] ==
  *[*[[pf [c i f] x y z] 0 31]
    *[[pf [c i f] x y z] [1 0] 4 0 14]] ==
  *[[pf [c i f] x y z] 2 [0 31] [1 0] 4 0 14]

/[counter z] == *[z 0 counter]
== *[*[[pf [c i f] x y z] 0 31]
     *[[pf [c i f] x y z] [1 0] 0 12]]
== *[[pf [c i f] x y z] 2 [0 31] [1 0] 0 12]

[floorhalf [y /[+(x) z]] z]
== *[[pf [c i f] x y z] [0 27]
     [[0 30] 2 [0 31] [1 0] 4 0 14] 0 31]
=even-params [[0 27] [[0 30] 2 [0 31] [1 0] 4 0 14]
              0 31]

[floorhalf [/[counter z] y] z]
== *[[pf [c i f] x y z] [0 27]
     [[2 [0 31] [1 0] 0 12] 0 30] 0 31]
=odd-params [[0 27] [[2 [0 31] [1 0] 0 12] 0 30]
             0 31]
```

All right, we now have a formula to return new parameters when `x` is even and a formula to return new parameters when `x` is odd.

Now we need to actually get our modified decrement to count up to `x-1`, updating our other data with it. Remember, our `recur-pf` formula will be operating on subject:

```
[params-formula [counter iseven floorhalf] x y z]
```

We want to call `params-formula` on:

```
[params-formula [new-counter new-iseven
                 new-floorhalf] x y z]
```

We can do this by calling:

```
*[[pf [c i f] x y z] 9 2 [0 2]
  [new-counter new-iseven new-floorhalf] 0 7]
```

`[[0 2] [new-counter new-iseven new-floorhalf 0 7]]` creates the new subject as a cell and the opcode 9 launches the head `2` on this new subject.

```
=recur-pf [2 [[0 2]
              [new-counter new-iseven new-floorhalf]
              0 7] 1 2 [0 1] 0 2]
```

Now let's actually update these values. `new-counter` is easy enough. It just increments the existing counter.

```
=new-counter [4 0 12]
```

`new-iseven` checks `iseven` then returns `1` if yes or `0` if no.

```
*[[pf [c i f] x y z] 6 iseven [1 1] 1 0]
=new-iseven [2 [0 1] 2 [1 [1 1] 1 0] [1 0]
             2 [1 2 3] [1 0] 4 4 iseven]
```

`new-floorhalf` checks `iseven` and returns `floorhalf` if yes or `+(floorhalf)` if no.

```
*[[pf [c i f] x y z] 6 iseven [0 27] 4 0 27]
=new-floorhalf [2 [0 1] 2 [1 [0 27] 4 0 27] [1 0]
                2 [1 2 3] [1 0] 4 4 iseven]
```

Okay! We have written our # hax edit operator as a Nock formula consisting of only Nocks 0 through 5. Very exciting! Let's put it together and see what it looks like. If you would like to try this yourself, copy and paste the following code in the dojo.

```
=iseven [0 26]
=new-counter [4 0 12]
=new-iseven [2 [0 1] 2 [1 [1 1] 1 0] [1 0]
             2 [1 2 3] [1 0] 4 4 iseven]
=new-floorhalf [2 [0 1] 2 [1 [0 27] 4 0 27] [1 0]
                2 [1 2 3] [1 0] 4 4 iseven]
=recur-pf [2 [[0 2]
              [new-counter new-iseven new-floorhalf]
              0 7] 1 2 [0 1] 0 2]
=odd-params [[0 27] [[2 [0 31] [1 0] 0 12] 0 30]
             0 31]
=even-params [[0 27] [[0 30] 2 [0 31] [1 0] 4 0 14]
              0 31]
=return-params [2 [0 1] 2 [1 even-params odd-params]
                [1 0] 2 [1 2 3] [1 0] 4 4 iseven]
=is-at-x [5 [0 14] 4 0 12]
=params-formula [2 [0 1] 2 [1 return-params recur-pf]
                 [1 0] 2 [1 2 3] [1 0] 4 4 is-at-x]
=call-head [2 [0 1] 0 2]
=pin-formula [2 [[1 params-formula] 0 1] 1 call-head]
=new-params [2 [[1 1 0 1] 0 1] 1 pin-formula]
=new-cell [[0 2] 2 [0 3] 1 new-params]
=recur-hax [2 new-cell 1 2 [0 1] 0 2]
=x-is-1 [5 [0 6] 1 1]
=return-y [0 14]
=hax-formula [2 [0 1] 2 [1 return-y recur-hax]
              [1 0] 2 [1 2 3] [1 0] 4 4 x-is-1]
=launch [2 [[1 hax-formula] 0 1] 1 call-head]
=x-is-0 [5 [0 2] 1 0]
=crash [0 0]
=check-x-0 [2 [0 1] 2 [1 crash launch] [1 0]
            2 [1 2 3] [1 0] 4 4 x-is-0]
=x-is-cell [3 0 2]

=hax [2 [0 1] 2 [1 crash check-x-0] [1 0] 2 [1 2 3]
      [1 0] 4 4 x-is-cell]
```

Great! What does this look like?

```
::  hax definition in raw Nock
[2 [0 1]
   2 [1 [0 0]
      2 [0 1]
        2 [1 [0 0]
           2 [[1 2 [0 1]
                   2 [1 [0 14]
                      2 [[0 2]
                         2 [0 3]
                         1 2 [[1 1 0 1] 0 1]
                             1 2 [[1 2 [0 1]
                                   2 [1 [2 [0 1]
                                         2 [1
  ↳ [[0 27] [[0 30] 2 [0 31] [1 0] 4 0 14] 0 31]
    [0 27]
    [[2 [0 31] [1 0] 0 12] 0 30]
                                                 0 31]
                                                [1 0]
                                              2 [1 2 3]
                                                [1 0]
                                                4 4 0 26]
                                           2 [[0 2]
                                              [[4 0 12]
  ↳ [2 [0 1]
       2 [1 [1 1] 1 0]
         [1 0]
         2 [1 2 3]
           [1 0]
           4 4 0 26]
     2 [0 1]
       2 [1 [0 27] 4 0 27]
         [1 0]
         2 [1 2 3]
           [1 0]
           4 4 0 26]
                                           0 7]
                                         1 2 [0 1]
                                         0 2]
                                        [1 0]
                                        2 [1 2 3]
                                          [1 0]
```

```
                                                  4 4 5 [0 14]
                                               4 0 12]
                                              0 1]
                                    1 2 [0 1]
                                 0 2]
                                1 2 [0 1]
                               0 2]
                           [1 0]
                  2 [1 2 3]
                    [1 0]
                    4 4 5 [0 6]
                      1 1]
                  0 1]
                1 2 [0 1]
            0 2]
          [1 0]
          2 [1 2 3]
            [1 0]
          4 4 5 [0 2]
          1 0]
    [1 0]
    2 [1 2 3]
      [1 0]
      4 4 3 0 2]
```

Quite a mouthful. I don't think this one will fit on a t-shirt. Maybe an evening gown, or perhaps a toga.

So, does it work?

```
> .*([1 [4 5] 6 7 8 9 10 11 12 13] hax)
[4 5]

> .*([2 [4 5] 6 7 8 9 10 11 12 13] hax)
[[4 5] 7 8 9 10 11 12 13]

> .*([3 [4 5] 6 7 8 9 10 11 12 13] hax)
[6 4 5]

> .*([62 [4 5] 6 7 8 9 10 11 12 13] hax)
[6 7 8 9 [4 5] 11 12 13]

> .*([17 [4 5] [[[[[[[6] 7] 8] 9] 10] 11] 12] 13]
     hax)
```

```
15  [[[[[[[6 7] 8] 9] 4 5] 11] 12] 13]
```

It appears so! We are inches away from the finish line.

8 Deriving Opcode 10

We now have our # hax operator. The last thing we have to do is use it to create a Nock 10 which runs on subject `a` with parameters `b`, `c`, and `d`:

```
*[a 10 [b c] d] == #[b *[a c] *[a d]]
```

Recall that we wrote # hax such that:

```
*[[x y z] hax] == #[x y z]
```

We can now rewrite this as:

```
*[a 10 [b c] d] ==
  *[[b *[a c] *[a d]] hax] ==
  *[[*[a 1 b] *[a c d]] *[a 1 hax]] ==
  *[*[a [1 b] c d] *[a 1 hax]] ==
  *[a 2 [[1 b] c d] 1 hax]
=nock-ten [2 [[1 b] c d] 1 hax]
```

And that's it. Congratulations! You have now derived all the Nock opcodes from 6 to 11 using only opcodes 0 through 5. What you will accomplish with this knowledge, I tremble to imagine. ⊠

Nock 10 Primitive Equivalent

```
*[a 10 [b c] d] ==
*[a 2 [[1 b] c d]
      1 2 [0 1]
          2 [1 [0 0]
             2 [0 1]
               2 [1 [0 0]
                  2 [[1 2 [0 1]
                       2 [1 [0 14]
                          2 [[0 2]
                             2 [0 3]
                           1 2 [[1 1 0 1] 0 1]
                           1 2 [[1 2 [0 1]
 ↳  2 [1 [2 [0 1]
          2 [1 [[0 27]
                [[0 30]
                  2 [0 31]
                    [1 0]
                    4 0 14]
                0 31]
                [0 27]
                [[2 [0 31]
                    [1 0]
                    0 12]
                  0 30]
               0 31]
               [1 0]
               2 [1 2 3]
                 [1 0]
                 4 4 0 26]
          2 [[0 2]
             [[4 0 12]
              [2 [0 1]
                  2 [1 [1 1] 1 0]
                    [1 0]
                    2 [1 2 3]
                      [1 0]
                      4 4 0 26]
               2 [0 1]
               2 [1 [0 27] 4 0 27]
                 [1 0]
```

```
                  2 [1 2 3]
                    [1 0]
                    4 4 0 26]
                0 7]
        1 2 [0 1] 0 2]
        [1 0]
        2 [1 2 3]
          [1 0]
          4 4 5 [0 14] 4 0 12]
                                0 1]
                             1 2 [0 1] 0 2]
                1 2 [0 1] 0 2]
           [1 0]
           2 [1 2 3]
             [1 0]
             4 4 5 [0 6]
           1 1]
         0 1]
      1 2 [0 1] 0 2]
    [1 0]
    2 [1 2 3]
      [1 0]
      4 4 5 [0 2] 1 0]
  [1 0]
  2 [1 2 3]
    [1 0]
    4 4 3 0 2]
```