- Nock specification, basic rules
- Constructing a trivial decrement operation
- Extended Nock
- Another hands-on construction
- Nock tree-walking interpreter
- Nock bytecode, if time

[0 1] to preserve current subject (kind of a pointer)

Nock Specification

Nock is an untyped combinator calculus. While it is aware

- A combinator calculus operates on constants.
    - Combinatory logic is free of variables, unlike the lambda calculus.
- The simplest combinator calculus consists of only:
    - **S** substitution
    - **K** constant
    - **I** identity
- Nock introduces a few more primitive operations as a practicality. It also uses a slightly different terminology for S, K, and I. There's also some subtle differences to its S or opcode 2 that we will elide.
    - Similar to Haskell Curry's BCKW system, which can be written in forms reducible to SKI, Nock provides a set of primitive rules and a set of economic extended rules for convenience in writing a compiler.
    - We simply assert without proof that Nock is a combinator calculus past this point, given the time constraints on our workshop today.
-

(We refer to the basic requirements for Turing completeness as exemplified in the μ-recursive functions which amount to constant, increment, variable access, program concatenation, looping per Raphael2012.) Paulsen1995

The Rules

Basic Nock

Nock 0

Every noun in Nock can be represented as a binary tree. Our first capability derives from the need to address values within that binary tree.
Nock's axiomatic convention is that all data are organized into binary trees addressable by natural numbers.
This corresponds to the S substitution operator of the SKI combinator.

Nock 1

Altho Nock is homoiconic, we need to be able to produce data as distinct from evaluated code; for instance, either as raw data inputs or as code for deferred evaluation. Opcode 1 fulfills this need. It simply throws away its subject and produces its argument.

```
> .*([1 2 3 4 5] [1 42])
42
```

This corresponds to the K constant operator of the SKI combinator.
We will skip lightly over opcode 2 for a moment in favor of the “axiomatic operators”.

Nock 3

It is convenient in some cases to be able to tell whether your noun is an atom or a cell. There is an axiomatic operator for this, and opcode 3 simply applies it.

```

```

However, we run into a quirk of Nock: the truth value is `0`, so we see that `?` wut returns `true` for an atom and `false` for a cell. (Think of these as being like C/Unix process return codes.)

Nock 4

```
> .*(0 [4 [0 1]])
1


> .*(0 [4 [1 41]])
42
```

- Are there any invalid inputs to this operator? (Yes, cells, but all atoms are valid.)

Nock 5

Like opcode 3, opcode 5 simply applies a rule: in this case, it checks `=` tis equality.

```
> .*([42 42] [5 [0 2] [0 3]])
0


> .*([42 43] [5 [0 2] [0 3]])
1
```

Opcodes 3, 4, and 5 are all designed to work on atoms. However, if you want to work with cells, you need to supply a cell of formulas; e.g.,

```
> .*([42 43] [[4 0 2] [5 [0 2] [0 3]]])
[43 1]
```

Nock 2

Unlike traditional compilers with labeled procedures, Nock lacks named functions or procedure calls, relying entirely on eval-like operations (opcode 2). The basic procedure is to either supply executable code directly with opcode 1 or look it up indirectly with opcode 0.

```
> .*([41 42] [2 [0 3] [1 [4 0 1]]])
43
```

- Talk through this one. In particular, note how the operational expression is pinned using opcode 1.

Opcode 2 is not used very much in Nock code produced by a compiler, in favor of opcode 9 which utilizes it.
This corresponds to the S constant operator of the SKI combinator.

Extended Nock

Nock 6

We are going to use a simpler, older definition for opcode 6 in this workshop.

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


.*(0 [2 [0 1] 2 [1 42 43] [1 0] 2 [1 2 3] 4 4 50])
```

`[0 0]` is the crash expression.

Nock 7

Nock 8

Nock 9

Exercises

1. Write a formula to return the subject unchanged.

2. Write a `[sut fol]` which returns itself (a quine).
3. Write a formula to return the constant `10` regardless of the subject.

Example: Increment

The direct application of the increment operator is straightforward:

```
.*(5 [4 [0 1]])
```

However, we cannot store such an expression in a convenient way for subsequent execution. That is, we want to treat a general increment formula as a stored procedure which we can invoke on demand in our program.
I will show you what such a stored procedure looks like, then we can talk through its pieces:

```
[8 [1 0] [1 4 0 6] 0 1]
```

- What do we see here?

Now that we have 9, we can refer to a longstanding convention in Nock that executable code sits in the head of a noun while data sits in the tail. We call this pair `[battery payload]`, and we subdivide the `payload` into `[sample context]`.

- What is the address of the battery? The sample? The context?

(We don't see these precisely here because we evaluate it “all at once” at the command line. But it will arise from the opcode 8 on its way.)

Example: Decrement

Almost everything interesting in our basic math library can be derived using increment and decrement. Since increment is built in, how can we do a decrement (subtraction of 1)? Think for a moment about an assembly language that didn't have a decrement or a subtraction operator. How would you proceed?

To build a decrement one piece at a time, let's write it in Jock first:

```
// Jock 0.1.0-alpha
func dec(a:Atom) -> Atom {
  let b = 0;
  loop;                        // starting point
  if a == +(b) {
    b
  } else {
    b = +(b);                  // increment operator
    recur
  }
}
```

Our Nock expression to decrement a function will proceed as follows.

First off it will have a few pieces, so let's build these:

- We need a counter variable starting at 0. How would we do this?
    - [8 [1 0] ...
- We need a loop recursion point. Fortunately, we don't have to mark it directly—we can simply mutate a copy of the subject when we get to the recursion point. `this` is the "body", `that` is the "recursion point".
    - [8 this that]
- We need a check for whether the value referred to as `a` is equal to the value referred to as `b` incremented by 1. For now, we don't know the axes yet, so let's punt on those with symbols.
    - [5 [0 a] 4 0 b]
- Those need to be embedded in a conditional branch.
    - [6 check [0 b] [4 0 b]]
- Finally, we need to recurse back to our loop start point. Since `8` modifies our subject, we will will be referring what's involved in our last `8`. (This is not the counter pin, but the loop recursion point. So `[9 2 0 1]`) is `that` from above.
- Put these together:

```
:: simplest possible invocation:
[8 [1 0] 8 [1 6 [5 [0 7] 4 0 6] [0 6] 9 2 [0 2] [4 0 6] 0 7] 9 2 0 1]
:: with zero check:
[6 [5 [0 1] [1 0]] [0 0] [8 [1 0] 8 [1 6 [5 [0 7] 4 0 6] [0 6] 9 2 [0 2] [4 0 6] 0 7] 9 2 0 1]]
:: with cell fence:
[6 [5 [0 1] [1 0]] [0 0] [6 [3 0 1] [0 0] [8 [1 0] 8 [1 6 [5 [0 7] 4 0 6] [0 6] 9 2 [0 2] [4 0 6] 0 7] 9 2 0 1]]]
```

Now, that is not exactly what a compiler would produce. Why is this different? It's defining a general-purpose function or gate, which changes the addresses in parts.

Hoon produces this code:

```
> !=
|=  a=@
=+  b=0
|-
?:  =(a +(b))  b
$(b +(b))
[8 [1 0] [1 8 [1 0] 8 [1 6 [5 [0 30] 4 0 6] [0 6] 9 2 10 [6 4 0 6] 0 1] 9 2 0 1] 0 1]
```

The Jock alpha compiler currently produces this code:

```
[8 [1 0] [1 8 [1 0] 8 [1 6 [5 [0 30] 4 0 6] [0 6] 7 [10 [6 4 0 6] 0 1] 9 2 0 1] 9 2 0 1] 0 1]
```

As with coding generally, we can arrive at a similar result by many paths.

Nock 10

In order to track more complicated moves, Nock 10 allows us to edit a subject and produced a mutated copy. This is most often called the “memory replacement” operator but I think it works better to call it an “edit” operator.

Example: Addition

The reader might wonder how an interpreter whose only arithmetic operation is increment can ever be practical.

The short answer is that a Nock interpreter doesn’t have to use the algorithm above. It just has to get the same result as the algorithm above. Likewise, when we work with

Nock is crash-only, which means that in practice we have to run it inside of a virtualized interpreter itself written in Nock. Almost everything in Urbit runs this way.

Event Loop

The beating heart of an operational system using Nock is the event loop formula. For each event that is injected, the system runs this formula in the kernel's `+aeon` arm:

```
.*([arvo -.epic] [9 2 10 [6 0 3] 0 2])
```

Let's talk through what is going on here.

- The subject consists of a pair of `arvo`, the evolution from underlying operating system and standard library until the current epitome state, and the head of `epic`, where `epic` is the remaining event log to be evaluated.
- The formula is a 9 which runs the head of the event log, `[0 3]`, against the whole subject, then replaces that into the head of the subject.
- The next event is then evaluated, until the list is empty.
- When it reaches the end of the event log, the runtime holds it in stasis until the next event is prepared to be injected.

Nock 11

Nock is a specification as well as an ISA, and features the ability to signal to its runtime arbitrarily. From the inside, Nock is a pure function: it knows nothing of the runtime's behavior or affordances and can only receive

(Note that we cannot just throw the hints away even if we do not use them, as a dynamic hint's code could crash.)

Jets

Jet dispatch does incur some overhead, which is generally worthwhile

Tree-Walking Interpreter

Nock is interpreted in the sense that it is produced to be executed by a virtual machine layer. A human end-user language will be parsed and interpreted into an abstract syntax tree, then converted into raw Nock for evaluation. This Nock is itself interpreted at runtime by a tree-walking interpreter.

This interpreter takes a Nock formula as input, recursively traverses the formula tree, evaluates each operation according to its subject, and returns the resulting noun.

The upside of Nock is that this process is fully general and portable. And since Nock is a both an ISA and a specification, when jetted code is available it can be evaluated in a faster way as much as possible.

So, some of you are going to ask me, what's the alternative to a tree-walking interpreter? To answer that, let's think about the Nock execution pattern. The major downside of Nock as an ISA is that it is a hyper-RISC that does not closely map to von Neumann architecture CPU operations. Modern computer architectures prefer to grab arrays in a hierarchy of caches, but a binary tree implementation does not lend itself well to this arrangement. Furthermore, because Nock is homoiconic, it is not straightforward to tell from a tree-walking interpreter if a given noun is code or data.

If tree-walking interpreters generally have poor cache locality [Nystrom], then Nock is hit with a double-whammy because it itself resolves to a tree even after interpretation.

- Nystrom

A newer paradigm, called subject knowledge analysis, proposes to fix this by a system of JIT-like affordances produced by a static analysis at compile time. SKA enables efficient analysis and execution of Nock code by unearthing the computational structure of the code even in an untyped procedure-free environment.

Nock Bytecode

There are currently two practical Nock interpreters in production, Urbit's Vere and Zorp's Sword. They both currently use a tree-walking interpreter, but a more sophisticated protocol is in active development. We will take a brief look at Vere's bytecode implementation.
Vere walks the Nock tree to produce a linear bytecode array for rapid evaluation. The bytecode interpreter is a stack machine built on top of an allocator. It uses a threaded code pattern or a computed GOTO. While it is highly optimized on its own terms, the gains are often clawed back by other parts of the system.
Vere can reveal the bytecode to us if we query it with an `%xray` Nock 11 hint.
For instance
We can see these by passing a Nock 11 `%xray` hint at the command line.

```
{[lils 65535] halt}
> ~>  %xray  65.535
65.535
{[libl i:0] halt}
> ~>  %xray  65.536
65.536
```

As with other bytecodes, these can be rather opaque. (In this case, we are sometimes left with labels and pointers to data in other locations rather than the complete data set.) Vere's bytecode interpreter is, I have been told, rather simple and primitive.

```
> !=  +(41)
[4 1 41]

> ~>  %xray  +(41)
{[lilb 41] bump halt}

> !=  (dec 43)
[8 [9 2.398 0 1.023] 9 2 10 [6 7 [0 3] 1 43] 0 2]

> ~>  %xray  (dec 43)
{[fask 1023] [kicb 1] snol head swap tail [lilb 43] musm [ticb 0] halt}
```