
Hot Nock: A History of the Nock Combinator Calculus

N. E. Davis `~lagrev-nocfep`
Zorp Corp

Abstract

Nock is a family of computational languages derived from the `SKI` combinator calculus. It serves as the ISA specification layer for the Urbit and NockApp systems. This article outlines the extant historical versions of the Nock combinator calculus and reconstructs the motivation for the changes made at each kelvin decrement. It begins with an exposition of Nock as a tool of computation, outlines the history of Nock’s decrements, and speculates on motivations for possible future developments.

Contents

1	Introduction	2
2	Nock as a Combinator Calculus	2
3	Nock’s Decrement	6
3.1	U	7
3.2	Nock 13K	15
3.3	Nock 12K	17
3.4	Nock 11K	19
3.5	Nock 10K	21

3.6	Nock 9K	23
3.7	Nock 8K	25
3.8	Nock 7K	27
3.9	Nock 6K	28
3.10	Nock 5K	29
3.11	Nock 4K	31
4	The Future of Nock	32
5	Conclusion	33

1 Introduction

Nock is a combinator calculus which serves as the computational specification layer for the Urbit and Nockchain/Nock-App systems. It is a hyper-RISC instruction set architecture (ISA) intended for execution by a virtual machine (but see **Mopfel2025** (**Mopfel2025**), pp. XX–XX herein). Nock’s simplicity and unity of expression make it amenable to proof-based reasoning and guarantees of correctness. Its Lisp-like nature surfaces the ability to introspect on the code itself, a property which higher-level languages compiling to it can exploit. Nock permits itself a finite number of specification changes, called “decrements” or “kelvins”, which allow it to converge on a balance of expressiveness and efficacy. This article outlines the extant historical versions of the Nock combinator calculus and reconstructs the motivation for the changes made at each kelvin decrement. It begins with an exposition of Nock as a tool of computation, outlines the history of Nock’s decrements, and speculates on motivations for possible future developments.

2 Nock as a Combinator Calculus

Fundamental computer science research has identified a family of universal computers which may be instantiated in a variety of ways, such as the Turing machine, the lambda calculus, and the combinator calculus. Equivalence theorems such as

the Church–Turing thesis show that these systems are equivalent in their computational power, and that they can be used to compute any computable function. The combinator calculus is a family of systems which use a small set of combinators to express computation. The most well-known member of this family is the **SKI** combinator calculus, which uses only three combinators: **S**, **K**, and **I**. Other members of this family include the **BCKW** combinator calculus and the **H** combinator calculus. These systems are all equivalent in their computational power, but they differ in their syntax and semantics. The Nock combinator calculus is an extension of the **SKI** combinator calculus which adds a few axiomatic rules to navigate and manipulate binary trees, carry out a very primitive arithmetic, and provide for side effects.

Perhaps better put, Nock is a family of combinator calculi that sequentially converge on an “optimal” expressiveness for certain design desiderata. This includes an economy of expression (thus several macro opcodes) and consideration of how a higher-level language would invoke stored procedural expressions. Furthermore an opcode exists which produces and then ignores a computation, intended to signal to a runtime layer that a side effect may be desired by the caller.

Nock bears the following characteristics:

- Turing-complete. Put formally, Turing completeness (and thus the ability to evaluate anything we would call a computation) is exemplified by the **-recursive functions** (**WikipediaMuRecursive**, **WikipediaMuRecursive**). In practice, these amount to operations for constant, increment, variable access, program concatenation, and looping (**Raphael2012**, **Raphael2012**). Nock supports these directly through its primitive opcodes.
- Functional (as in language). Nock is a pure function of its arguments. In practice, the Urbit operating system provides a simulated global scope for userspace applications, but this virtualized environment reduces to garden-variety Nock. (See **Davis2025b** (**Davis2025b**), pp. XX–XX in this volume, for details of a Nock virtualized interpreter.)

- Subject-oriented. Nock evaluation consists of a formula as a noun to be evaluated against a subject as a noun. Taken together, these constitute the entire set of inputs to a pure function.

Everything about a scope, including name bindings, aliases, and docstrings, is stored in the subject's type. This allows Hoon's compilation discipline to be similarly minimal: the compiler is a function from subject type and Hoon source to product type and compiled Nock. Running this Nock against a value of the subject type produces a value of the result. It's hard to imagine a more streamlined formalism for compilation. (**Blackman2020**, **Blackman2020**)

Some Nock opcodes alter the subject, such as a variable declaration, by producing a new subject which is utilized for subsequent axis lookups.

- Homoiconic. Nock unifies code and data under a single representation. A Nock atom is a natural number, and a Nock cell is a pair of nouns. Every Nock noun is acyclic, and every Nock expression is a binary tree. For example, Nock expressions intended to be evaluated as code are often pinned as data by the constant opcode until they are retrieved by evaluating the constant opcode at that axis.
- Untyped. Nock is untyped, meaning that it does not impose any type system on the expressions it evaluates. Nock "knows" about the natural numbers in two senses: such are used for addressing axes in the binary tree of a noun, and such are manipulated and compared using the increment and equality opcodes.
- Solid-state. A Nock interpreter is a solid-state machine, meaning that it operates from a state to a new state strictly according to inputs as a pure lifecycle function. The Nock

interpreter must commit the results of a successful computation as the new state before subsequent computations, or events, can be evaluated. Transient evaluations (uncompleted events) and crashes (invalid evaluations) may be lost without consequence, and the Nock interpreter layer persists the underlying state of the machine.

We have asserted without demonstration thus far that Nock is a combinator calculus. We now show that this is the case, with reference to Nock 4K, the latest specification. The simplest combinator calculus consists of only three combinators (**SomeoneSmart**, **SomeoneSmart**), *s*, *κ*, and *ι*. These combinators are defined as follows:

1. *s* substitution. This corresponds to Nock 4K's opcode 2, which substitutes the second argument into the first argument at the third argument's axis. (There are some subtle differences to Nock's expression of *s* as opcode 2 that we will elide as being fundamentally similar, but perhaps worthy of its own monograph.)
2. *κ* constant. This corresponds to Nock 4K's opcode 1, which yields its argument as a constant noun.
3. *ι* identity. This corresponds to Nock 4K's opcode 0, a generalized axis lookup operator, which can trivially retrieve the current subject or expression as well as any children.

While Nock introduces a few more primitive operations as a practicality, the above identities establish its bona fides as a combinator calculus capable of general computation. 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. (See **Galebach2025 (Galebach2025)**, pp. XX–XX in this volume, for exposition on how to evaluate a Nock expression by hand or by interpreter.)

In an early document, Yarvin explained two of his design criteria in producing Nock as a practical ISA target (**Yarvin2010, Yarvin2010**):

1. Natural conversion of source to code without other inputs.
2. Metacircularity without deep stacks; i.e., the ability to extend Nock semantics without altering the underlying substrate.

This latter idea he particularly connected to the concept of a “scry namespace”: “dereferencing Urbit paths is as natural (and stateless) a function as increment or equals” (ibid.). Indeed, Urbit’s current userspace utilizes such an affordance to replicate a global scope environment for accessing system and remote resources. (See **Davis2025b (Davis2025b)**, pp. XX–XX in this volume, for a discussion of the Nock virtualized interpreter.)

3 Nock's Decrements

The Nock family survives in a trail of breadcrumbs, with each version of the specification being a kelvin decrement of the previous version. Early versions were produced exclusively by Curtis Yarvin, eventually involving the input of other developers after the 2013 founding of Tlon Corporation. In this section, we present each extant version of the Nock specification and comment on the changes and their motivations. Only the layouts have been changed for print. Dates for Nock specifications are derived from dated public posts (U, 9K), internal dating (13K, 12K, 11K, 10K), or from Git commit history data (8K, 7K, 6K, 5K).¹ No version of 14K survives publicly, nor does any primordial version prior to U (15K) appear to exist.

Yarvin’s background as a systems engineer with systems like Xaos Tools (for SGI Irix), Geoworks (on DoCoMo’s iMode), and Unwired Planet (on the Wireless Application Protocol, WAP) inclined him towards a formal break with Unix-era computing (**Yarvin2025, Yarvin2025**). He sought to produce a system enabling server-like behavior rather than a network of clients dependent on centralized servers for a functional Internet. This

¹In at least one case (7K), Yarvin claims to have finished the proposal a month earlier but to not have posted it until this date.

required a deep first-principles rederivation of computing; the foundational layer was a combinator calculus which became Nock. Nock was intended from the beginning to become less provisional over time, encoding a kelvin decrement which forced the specification to converge on a sufficiently good set of op-codes. Many downstream consequences of Urbit and NockApp as systems derive directly from the affordances encoded into Nock.

3.1 U

I have not really worked with combinator models, but my general impression is that it takes essentially an infinite amount of syntactic sugar to turn them into a programming language. U certainly takes some sweetener, but not, I think, as much. (Curtis Yarvin)

The earliest extant Nock is U, a proto-Nock posted to the *Lambda the Ultimate* blog in 2006 (Yarvin2006, Yarvin2006).² The draft is versioned 0.15; subsequent evidence indicates that this is a downward-counting kelvin-versioned document already. The specification is produced in full in Listing 1.

Extensive commentary on the operators is provided. Rightwards grouping of tuple expressions has already been introduced. Extension of the language is summarily ruled out.³ Details of parsing and lexing are considered. Terms (the ancestor of nouns) include a NULL-like “foo” type ~ distinguishable by value rather than structure. ASCII is built in as numeric codes, similar to Gödel numbering.

As commenter Mario B. pointed out, the U specification permits SKI operators with the simple expressions,

[name]	[pattern]	[definition]
(I)	(I \$a)	\$a
(K)	(K \$a \$b)	\$b

²Curtis Yarvin was consulted for elements of this history. Unfortunately many elements of the original prehistory of Nock appear to be lost to the sands of time on unrecoverable hard drives.

³Compare Ax and Conk, pp. XX–XX herein.

(S) (S \$a \$b \$c) (\$a \$c (\$b \$c))

While early work (1940s–50s) had been carried out on “minimal instruction set computers” (MISCs), it is more likely that Yarvin was influenced by contemporaneous work on “reduced instruction set computers” (RISCs) in the 1980s and 90s. Language proposals like that of **Burger2006 (Burger2006)** (Pico Lisp) may have influenced Yarvin’s design choices throughout this era.

The U specification is in some ways the single most interesting historical document of our series. Yarvin particularly identified a desire to avoid baking abstractions like variables and functions into the U cake, and an emphasis on client–server semantics. The scry namespace appears *avant la lettre* as a referentially transparent immutable distributed namespace. U expresses a very ambitious hyper-Turing operator, acknowledging that its own instantiation from the specification is impossible and approximate. Yarvin grapples in U with the halting problem (via his follow operator) and with the tension between a specification and an implementation (a gulf he highlighted as a human problem in his 2025 LambdaConf keynote address). Furthermore, asides on issues like the memory arena prefigure implementation details of Vere as a runtime.

Listing 1: U, 31 January 2006. The earliest extant patriarch of the Nock family.

U: Definition

1 Purpose

5 This document defines the U function and its data model.

2 License

 U is in the public domain.

10 3 Status

 This text is a DRAFT (version 0.15).

4 Data

 A value in U is called a “term.” There are three

15 kinds of term: "number," "pair," and "foo."

A number is any natural number (ie, nonnegative integer).

20 A pair is an ordered pair of any two terms.

There is only one foo.

5 Syntax

25 U is a computational model, not a programming language.

But a trivial ASCII syntax for terms is useful.

30 5.1 Trivial syntax: briefly

Numbers are in decimal. Pairs are in parentheses that nest to the right. Foo is "~".

Whitespace is space or newline. Line comments use "#".

5.2 Trivial syntax: exactly

term : number
 | 40 ?white pair ?white 41
40 | foo

number : 48
 | [49-57] *[48-57]

45 pair : term white term
 | term white pair

foo : 126

50 white : *(32 | 10 | (35 *[32-126] 10))

6 Semantics

U is a pure function from term to term.

55 This document completely defines U. There is no compatible way to extend or revise U.

6.1 Rules

	[name]	[pattern]	[definition]
60			
	(a)	(\$a 0 \$b)	\$b
	(b)	(\$a 1 \$b \$c)	1
	(c)	(\$a 1 \$b)	0
	(d)	(\$a 2 0 \$b \$c)	\$b
65	(e)	(\$a 2 %n \$b \$c)	\$c
	(f)	(\$a 3 \$b \$c)	=(\$b \$c)
	(g)	(\$a 4 %n)	+%n
	(h)	(\$a 5 (~ ~ \$b) \$c)	\$b
70	(i)	(\$a 5 (~ \$b \$c) \$d)	*(\$a \$b \$c \$d)
	(j)	(\$a 5 (~ ~) \$b)	~
	(k)	(\$a 5 (~ \$b) \$c)	*(\$a \$b \$c)
	(l)	(\$a 5 (\$b \$c) \$d)	*(\$a \$b \$d) *(\$a \$c \$d)
75	(m)	(\$a 5 \$b \$c)	\$b
	(n)	(\$a 6 \$b \$c)	*(\$a *(\$a 5 \$b \$c))
	(o)	(\$a 7 \$b)	*(\$a 5 \$a \$a \$b)
	(p)	(\$a 8 \$b \$c \$d)	> (\$b \$c \$d)
80			
	(q)	(\$a \$b \$c)	*(\$a 5 *(\$a 7 \$b) \$c)
	(r)	(\$a \$b)	*(\$a \$b)
	(s)	\$a	*\$a

85 The rule notation is a pseudocode, only used in this file. Its definition follows.

6.2 Rule pseudocode: briefly

Each line is a pattern match. "%" means
90 "number." Match in order. See operators below.

6.3 Rule pseudocode: exactly

Both pattern and definition use the same evaluation language, an extension of the trivial
95 syntax.

An evaluation is a tree in which each node is a term, a term-valued variable, or a unary

operation.

100

Variables are symbols marked with a constraint.
A variable "\$name" matches any term. "%name"
matches any number.

105

There are four unary prefix operators, each of
which is a pure function from term to term: "=",
"+", "*", and ">". Their semantics follow.

6.4 Evaluation semantics

110

For any term \$term, to compute U(\$term):

115

- find the first pattern, in order, that
matches \$term.
- substitute its variable matches into its
definition.
- compute the substituted definition.

Iff this sequence of steps terminates, U(\$term)
"completes." Otherwise it "chokes."

120

Evaluation is strict: incorrect completion is a
bug. Choking is U's only error or exception
mechanism.

125

6.5 Simple operators: equal, increment, evaluate

=(\$a \$b) is 0 if \$a and \$b are equal; 1 if they
are not.

+%n is %n plus 1.

130

*\$a is U(\$a).

6.6 The follow operator

135

>(\$a \$b \$c) is always 0. But it does not always
complete.

We say "\$c follows \$b in \$a" iff, for every \$term:

- if *(\$a 5 \$b \$term) chokes:
- *(\$a 5 \$c \$term) chokes.

140

```

    if *($a 5 $b $term) completes:
        either:
            *($a 5 $c $term) completes, and
145         *($a 5 $c $term) equals
            *($a 5 $b $term)
        or:
            *($a 5 $c $term) chokes.
```

```
150 If $c follows $b in $a, >($a $b $c) is 0.
```

```

    If this statement cannot be shown (ie, if there
    exists any $term that falsifies it, generates an
    infinitely recursive series of follow tests, or is
155 inversely self-dependent, ie, exhibits Russell's
    paradox), >($a $b $c) chokes.
```

7 Implementation issues

```

    This section is not normative.
```

160

7.1 The follow operator

```

    Of course, no algorithm can completely implement
    the follow operator. So no program can completely
    implement U.
```

165

```

    But this does not stop us from stating the
    correctness of a partial implementation - for
    example,
    one that assumes a hardcoded set of follow cases,
    and
    fails when it would otherwise have to compute a
    follow
170 case outside this set.
```

```

    U calls this a "trust failure." One way to
    standardize trust failures would be to standardize
    a fixed set of follow cases as part of the
    definition
175 of U. However, this is equivalent to
    standardizing a
    fixed trusted code base. The problems with this
    approach are well-known.
```

180 A better design for U implementations is to
depend on a voluntary, unstandardized failure
mechanism. Because all computers have bounded
memory,
and it is impractical to standardize a fixed
memory
size and allocation strategy, every real computing
environment has such a mechanism.

185 For example, packet loss in an unreliable packet
protocol, such as UDP, is a voluntary failure
mechanism.

190 If the packet transfer function of a stateful UDP
server is defined in terms of U, failure to
compute
means dropping a packet. If the server has no
other
I/O, its semantics are completely defined by its
initial state and packet function.

195

7.2 Other unstandardized implementation details

A practical implementation of U will detect and
log common cases of choking. It will also need a
timeout or some other unspecified mechanism to
abort undetected infinite loops.

200

(Although trust failure, allocation failure or
timeout, and choke detection all depend on what
is presumably a single voluntary failure
mechanism, they are orthogonal and should not be
confused.)

205

Also, because U is so abstract, differences in
implementation strategy can result in performance
disparities which are almost arbitrarily extreme.
The difficulty of standardizing performance is
well-known.

210

No magic bullet can stop these unstandardized
issues from becoming practical causes of lock-in

215

and incompatibility. Systems which depend on U must manage them at every layer.

In plain language, the rules of U are as follows:

- (a) axis lookup
- (b) cell detection
- (c) atom detection
- (d) constant
- (e)
- (f) equality
- (g) increment
- (h)
- (i)
- (j)
- (k)
- (l) cell distribution
- (m) head
- (n)
- (o)
- (p) hyper-Turing follow operator
- (q)
- (r) cell distribution
- (s) trivial evaluation

3.2 Nock 13K

At some point between January 2006 and March 2008, Nock acquired its cognomen.

The only compound opcode is opcode 6, the conditional branch opcode.

Axiomatic operator `*` is identified as a GOTO.

Listing 2: Nock 13K, 8 March 2008.

Author: Curtis Yarvin (curtis.yarvin@gmail.com)
Date: 3/8/2008
Version: 0.13

5 1. Manifest

This file defines one Turing-complete function,
"nock."

nock is in the public domain. So far as I know,
it is
10 neither patentable nor patented. Use it at your
own risk.

2. Data

Both the domain and range of nock are "nouns."
15

A "noun" is either an "atom" or a "cell." An
"atom" is an unsigned
integer of any size. A "cell" is an ordered pair
of any two nouns,
the "head" and "tail."

20 3. Pseudocode

nock is defined in a pattern-matching pseudocode.

Match precedence is top-down. Operators are
prefix. Parens
25 denote cells, and group right: (a b c) is (a (b
c)).

4. Definition

4.1 Transformations

```

30      *(a 0 b c)  => *(*(a b) c)
      *(a 0 b)    => /(b a)
      *(a 1 b)    => (b)
      *(a 2 b)    => ***(a b)
35      *(a 3 b)    => &*(a b)
      *(a 4 b)    => ^*(a b)
      *(a 5 b)    => =*(a b)
      *(a 6 b c d) => *(a 2 (0 1) 2 (1 c d) (1 0) 2
(1 2 3) (1 0) 4 4 b)
      *(a b c)    => (*(a b) *(a c))
40      *(a)       => *(a)

```

4.2 Operators

4.2.1 Goto (*)

```

45      *(a)          ->nock(a)

```

4.2.2 Deep (&)

```

50      &(a b)         -> 0
      &(a)            -> 1

```

4.2.3 Bump (^)

```

55      ^ (a b)        -> ^ (a b)
      ^ (a)           -> a + 1

```

4.2.4 Same (=)

```

60      =(a a)         -> 0
      =(a b)         -> 1
      =(a)           -> =(a)

```

4.2.5 Snip (/)

```

65      /(1 a)         -> a
      /(2 a b)        -> a

```



```
70      /(3 a b)      -> b
      /((a + a) b)   -> /(2 /(a b))
      /((a + a + 1) b) -> /(3 /(a b))
      /(a)           -> /(a)
```

3.3 Nock 12K

Opcodes were reordered slightly. A dynamic hint opcode 8 and a static hint opcode 9 were added.

TODO is that 8?

Listing 3: Nock 12K, 2008.

Author: Curtis Yarvin (curtis.yarvin@gmail.com)

Date: 3/28/2008

Version: 0.12

5 1. Introduction

This file defines one function, "nock."

nock is in the public domain.

10

2. Data

A "noun" is either an "atom" or a "cell." An
"atom" is an unsigned integer of any size. A
15 "cell" is an ordered pair of any two nouns,
the "head" and "tail."

3. Semantics

20

nock maps one noun to another. It doesn't
always terminate.

4. Pseudocode

25

nock is defined in a pattern-matching
pseudocode, below.

Parentheses enclose cells. (a b c) is
(a (b c)).

30

5. Definition

5.1 Transformations

35

$\star(a\ (b\ c)\ d) \Rightarrow (\star(a\ b\ c)\ \star(a\ d))$

$\star(a\ 0\ b) \Rightarrow /(b\ a)$

$\star(a\ 1\ b) \Rightarrow (b)$

$\star(a\ 2\ b\ c) \Rightarrow \star(\star(a\ b)\ c)$

$\star(a\ 3\ b) \Rightarrow \star\star(a\ b)$

40

$\star(a\ 4\ b) \Rightarrow \&\star(a\ b)$

$\star(a\ 5\ b) \Rightarrow ^\star(a\ b)$

$\star(a\ 6\ b) \Rightarrow =\star(a\ b)$

$\star(a\ 7\ b\ c\ d) \Rightarrow \star(a\ 3\ (0\ 1)\ 3\ (1\ c\ d)\ (1\ 0))$

45

$\qquad\qquad\qquad 3\ (1\ 2\ 3)\ (1\ 0)\ 5\ 5\ b)$

$\star(a\ 8\ b\ c) \Rightarrow \star(a\ 2\ (((1\ 0)\ b)\ c)\ 0\ 3)$

$\star(a\ 9\ b\ c) \Rightarrow \star(a\ c)$

$\star(a) \Rightarrow \star(a)$

50

5.2 Operators

5.2.1 Goto (\star)

55

$\star(a) \rightarrow \text{nock}(a)$

5.2.2 Deep ($\&$)

$\&(a\ b) \rightarrow 0$

60

$\&(a) \rightarrow 1$

5.2.4 Bump ($^$)

$^ (a\ b) \rightarrow ^ (a\ b)$

65

$^ (a) \rightarrow a + 1$

5.2.5 Same ($=$)

$= (a\ a) \rightarrow 0$

70

$= (a\ b) \rightarrow 1$

$= (a) \rightarrow =(a)$

5.2.6 Snip (/)

```
75      /(1 a)          -> a
      /(2 a b)         -> a
      /(3 a b)         -> b
      /((a + a) b)     -> /(2 /(a b))
      /((a + a + 1) b) -> /(3 /(a b))
80      /(a)           -> /(a)
```

3.4 Nock 11K

TODO

The kelvin versioning system here became explicit (rather than implicitly decreasing minor versions).

Listing 4: Nock 11K, 25 May 2008.

Author: Mencius Moldbug (moldbug@gmail.com)

Date: 5/25/2008

Version: 11K

5 1. Introduction

This file defines one function, "nock."

nock is in the public domain.

10

2. Data

A "noun" is either an "atom" or a "cell." An
"atom" is an unsigned
integer of any size. A "cell" is an ordered pair
of any two nouns,
15 the "head" and "tail."

3. Semantics

nock maps one noun to another. It doesn't always
terminate.

20

4. Pseudocode

nock is defined in a pattern-matching pseudocode, below.

25 Parentheses enclose cells. (a b c) is (a (b c)).

5. Definition

5.1 Transformations

```

30      *(a (b c) d) => (*(a b c) *(a d))
      *(a 0 b)      => /(b a)
      *(a 1 b)      => (b)
      *(a 2 b c d) => *(a 3 (0 1) 3 (1 c d) (1 0)
35                      3 (1 2 3) (1 0) 5 5 b)
      *(a 3 b)       => **(a b)
      *(a 4 b)       => &*(a b)
      *(a 5 b)       => ^*(a b)
      *(a 6 b)       => =*(a b)
40
      *(a 7 b c)     => *(a 3 (((1 0) b) c) 1 0 3)
      *(a 8 b c)     => *(a c)

      *(a)           => *(a)

```

5.2 Operators

5.2.1 Goto (*)

```
50      * (a)      ->  nock(a)
```

5.2.2 Deep (&)

```

55      &(a b)      -> 0
      &(a)         -> 1

```

5.2.4 Bump (^)

$$\begin{array}{ll} \text{\texttt{^ (a b)}} & \rightarrow \text{\texttt{^ (a b)}} \\ \text{\texttt{^ (a)}} & \rightarrow \text{\texttt{a + 1}} \end{array}$$

5.2.5 Same (=)

```
65      = (a a)          -> 0
      = (a b)          -> 1
      = (a)            -> =(a)
```

5.2.6 Snip (/)

```
70      / (1 a)          -> a
      / (2 a b)         -> a
      / (3 a b)         -> b
      / ((a + a) b)     -> / (2 / (a b))
      / ((a + a + 1) b) -> / (3 / (a b))
75      / (a)           -> / (a)
```

3.5 Nock 10K

Parentheses were replaced by brackets. Opcodes were reordered slightly. Compound opcodes for composition and extension (variable definition) were removed. Functionally, 11K and 10K appear very similar, particularly if the Watt (proto-Hoon) compiler is set up to produce variable declarations and compositions as the compound opcodes had them.

Listing 5: Nock 10K, 15 September 2008.

Author: Mencius Molddbug [molddbug@gmail.com]
Date: 9/15/2008
Version: 10K

5 1. Introduction

This file defines one function, "nock."

nock is in the public domain.

10

2. Data

A "noun" is either an "atom" or a "cell." An
"atom" is an unsigned
integer of any size. A "cell" is an ordered pair
of any two nouns,
15 the "head" and "tail."

3. Semantics

nock maps one noun to another. It doesn't always terminate.

20

4. Pseudocode

nock is defined in a pattern-matching pseudocode, below.

25

Brackets enclose cells. [a b c] is [a [b c]].

5. Definition

5.1 Transformations

30

```
*[a [b c] d] => [*[a b c] *[a d]]
*[a 0 b]      => /[b a]
*[a 1 b]      => [b]
*[a 2 b c d] => *[a 3 [0 1] 3 [1 c d]
                                     [1 0] 3 [1 2 3] [1 0] 5 5 b]
35 *[a 3 b]      => **[a b]
   *[a 4 b]      => &*[a b]
   *[a 5 b]      => ^*[a b]
   *[a 6 b]      => =*[a b]
40 *[a]          => *[a]
```

5.2 Operators

5.2.1 Goto [*]

45

```
*[a]          ->nock[a]
```

5.2.2 Deep [&]

50

```
&[a b]        -> 0
&[a]          -> 1
```

5.2.4 Bump [^]

55

```
^[a b]        ->^[a b]
```

```
^ [a]                -> (a + 1)
```

5.2.5 Like [=]

```
60    = [a a]          -> 0
      = [a b]          -> 1
      = [a]            -> = [a]
```

5.2.6 Snip [/]

```
65    / [1 a]          -> a
      / [2 a b]        -> a
      / [3 a b]        -> b
      / [(a + a) b]    -> / [2 / [a b]]
70    / [(a + a + 1) b] -> / [3 / [a b]]
      / [a]            -> / [a]
```

3.6 Nock 9K

The cell detection axiomatic operator underlying opcode 4 (cell detection) was changed from & to ?.

This version of Nock was published on the Moron Lab blog in 2010 (**Yarvin2010**, **Yarvin2010**) as “Maxwell’s equations of software”. Yarvin emphasized that Nock was intended to serve as “foundational system software rather than foundational metamathematics” (ibid.).

Yarvin also publicly exposed his thoughts about building a practical higher-level language on top of Nock at this point (ibid.):

To define a language with Nock, construct two nouns, q and r , such that $*[q\ r]$ equals r , and $*[s\ *[p\ r]]$ is a useful functional language. In this description,

- p is the function source;
- q is your language definition, as source;
- r is your language definition, as data;
- s is the input data.

TODO bug in expression?

More concretely, Watt (the predecessor to Hoon) is defined as:

```

urbit-formula == Watt(urbit-source)
               == Nock(urbit-source watt-formula)
watt-formula  == Watt(watt-source)
               == Nock(watt-source watt-formula)

```

TODO

Versus 10K, 9K elides operator names in favor of definitions. Other differences are likewise primarily terminological, such as the replacement of Deep & with ?.

Yarvin has been to think about virtualizing Nock interpretation to expose a broader namespace for interaction with values than the “strict” subject of a formula (**Yarvin2010**, **Yarvin2010**).

Listing 6: Nock 9K, *terminus ad quem* 7 January 2010.

1 Context

 This spec defines one function, Nock.

5 2 Structures

 A noun is an atom or a cell. An atom is any unsigned integer.
 A cell is an ordered pair of any two nouns.

10 3 Pseudocode

 Brackets enclose cells. [a b c] is [a [b c]].
 *a is Nock(a). Reductions match top-down.

15

4 Reductions

```

?[a b]          => 0
?a              => 1

^ [a b]         => ^ [a b]
^a              => (a + 1)

```

20


```

25      = [a a]          => 0
      = [a b]          => 1
      = a              => =a

      /[1 a]           => a
      /[2 a b]         => a
30      /[3 a b]         => b
      /[(a + a) b]     => /[2 /[a b]]
      /[(a + a + 1) b] => /[3 /[a b]]
      /a               => /a

35      *[a 0 b]        => /[b a]
      *[a 1 b]         => b
      *[a 2 b c d]     => *[a 3 [0 1] 3 [1 c d]
                        [1 0] 3 [1 2 3] [1 0] 5 5 b]
40      *[a 3 b]        => **[a b]
      *[a 4 b]         => ?*[a b]
      *[a 5 b]         => ^*[a b]
      *[a 6 b]         => =[a b]
      *[a [b c] d]     => [*[a b c] *[a d]]
      *a               => *a

```

3.7 Nock 8K

The compound opcodes reappear for the first time since U in Nock 8K. An opcode 12 appears as TODO

8K received an uncharacteristic amount of commentary, given a document prepared for its presentation at sss10 (**Yarvin2010a**, **Yarvin2010a**).

Lambda was highlighted as a design pattern (a “gate” or stored procedure call) enabled by the “core” convention. Notably, `[[sample context] battery]` occurred in a different order than has been conventional since 2013 (emphasizing that the common core pattern is a convention rather than a requirement of Nock).

Listing 7: Nock 8K, 25 July 2010.

1 Structures

A noun is an atom or a cell. An atom is any unsigned integer.

A cell is an ordered pair of nouns.

5

2 Pseudocode

[a b c] is [a [b c]]; *a is nock(a). Reductions match top-down.

10 3 Reductions

	?[a b]	0
	?a	1
	^a	(a + 1)
15	= [a a]	0
	= [a b]	1
	/[1 a]	a
	/[2 a b]	a
20	/[3 a b]	b
	/[(a + a) b]	/[2 /[a b]]
	/[(a + a + 1) b]	/[3 /[a b]]
	[a [b c] d]	[[a b c] *[a d]]
25	*[a 0 b]	/[b a]
	*[a 1 b]	b
	*[a 2 b c]	*[*[a b] *[a c]]
	[a 3 b]	?[a b]
	[a 4 b]	^[a b]
30	*[a 5 b]	=*[a b]
	*[a 6 b c d]	*[a 2 [0 1] 2 [1 c d] [1 0] 2
	[1 2 3] [1 0] 4 4 b]	
	*[a 7 b c]	*[a 2 b 1 c]
	*[a 8 b c]	*[a 7 [7 b [0 1]] c]
35	*[a 9 b c]	*[a 8 b 2 [[7 [0 3] d] [0 5]] 0
	5]	
	*[a 10 b c]	*[a 8 b 8 [7 [0 3] c] 0 2]
	*[a 11 b c]	*[a 8 b 7 [0 3] c]
	*[a 12 b c]	*[a [1 0] 1 c]
40	^[a b]	^[a b]

= a	= a
/ a	/ a
* a	* a

3.8 Nock 7K

TODO

During this era, substantial development took place on the early Urbit operating system. Nock began to be battle-tested in a way it had not previously been stressed. Several decrements occurred in short order.

Opcode 12 is reduced to TODO

Listing 8: Nock 7K, *terminus ad quem* 14 November 2010.

1 Structures

A noun is an atom or a cell. An atom is any
natural number.

A cell is any ordered pair of nouns.

5

2 Pseudocode

[a b c]	[a [b c]]
nock(a)	*a
?[a b]	0
?a	1
^a	1 + a
= [a a]	0
= [a b]	1
/[1 a]	a
/[2 a b]	a
/[3 a b]	b
/[(a + a) b]	/[2 /[a b]]
/[(a + a + 1) b]	/[3 /[a b]]
[a [b c] d]	[[a b c] *[a d]]
*[a 0 b]	/[b a]

25

```

    * [a 1 b]          b
    * [a 2 b c]        * [* [a b] * [a c]]
    * [a 3 b]          ? * [a b]
    * [a 4 b]          ^ * [a b]
30  * [a 5 b]          = * [a b]

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

    * [a 7 b c]        * [a 2 b 1 c]
35  * [a 8 b c]        * [a 7 [[7 [0 1] b] 0 1] c]
    * [a 9 b c]        * [a 7 c 0 b]
    * [a 10 b c]       * [a c]
    * [a 10 [b c] d]   * [a 8 c 7 [0 3] d]

40  ^ [a b]           ^ [a b]
    = a               = a
    / a               / a
    * a               * a

```

3.9 Nock 6K

TODO

The axiomatic operator for increment was changed from to +. The definition of opcode 8 (extend) was simplified somewhat, and the order XXX [o 2]

Listing 9: Nock 6K, 6 July 2011.

1 Structures

```

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

```

2 Reductions

```

nock(a)          * a
10  [a b c]      [a [b c]]

?[a b]           0
?a              1
+a              1 + a

```

15	= [a a]	0
	= [a b]	1
	/ [1 a]	a
	/ [2 a b]	a
20	/ [3 a b]	b
	/ [(a + a) b]	/ [2 / [a b]]
	/ [(a + a + 1) b]	/ [3 / [a b]]
	* [a [b c] d]	* [a b c] * [a d]]
25		
	* [a 0 b]	/ [b a]
	* [a 1 b]	b
	* [a 2 b c]	* [* [a b] * [a c]]
	* [a 3 b]	? * [a b]
30	* [a 4 b]	+ * [a b]
	* [a 5 b]	= * [a b]
	* [a 6 b c d]	* [a 2 [0 1] 2 [1 c d] [1 0] 2 [1
		2 3] [1 0] 4 4 b]
	* [a 7 b c]	* [a 2 b 1 c]
35	* [a 8 b c]	* [a 7 [[0 1] b] c]
	* [a 9 b c]	* [a 7 c 0 b]
	* [a 10 b c]	* [a c]
	* [a 10 [b c] d]	* [a 8 c 7 [0 2] d]
40		
	+ [a b]	+ [a b]
	= a	= a
	/ a	/ a
	* a	* a

3.10 Nock 5K

All trivial reductions of axiomatic operators were removed to the preface of the specification.

(For instance, a trivial “cosmetic” change was made to 5K’s specification after it was publicly posted in order to synchronize it with the VM’s behavior)

Listing 10: Nock 5K, 24 September 2012.

1 Structures

A noun is an atom or a cell. An atom is any natural number.

A cell is an ordered pair of nouns.

5

2 Reductions

	nock(a)	*a
	[a b c]	[a [b c]]
10	[a b]	0
	?a	1
	+ [a b]	+ [a b]
	+ a	1 + a
15	= [a a]	0
	= [a b]	1
	= a	= a
	/ [1 a]	a
20	/ [2 a b]	a
	/ [3 a b]	b
	/ [(a + a) b]	/ [2 / [a b]]
	/ [(a + a + 1) b]	/ [3 / [a b]]
	/ a	/ a
25	* [a [b c] d]	* [a b c] * [a d]
	* [a 0 b]	/ [b a]
	* [a 1 b]	b
30	* [a 2 b c]	* [* [a b] * [a c]]
	* [a 3 b]	? * [a b]
	* [a 4 b]	+ * [a b]
	* [a 5 b]	= * [a b]
35	* [a 6 b c d]	* [a 2 [0 1] 2 [1 c d] [1 0] 2 [1 2 3] [1 0] 4 4 b]
	* [a 7 b c]	* [a 2 b 1 c]
	* [a 8 b c]	* [a 7 [[7 [0 1] b] 0 1] c]
	* [a 9 b c]	* [a 7 c 2 [0 1] 0 b]
	* [a 10 [b c] d]	* [a 8 c 7 [0 3] d]
40	* [a 10 b c]	* [a c]

*a

*a

3.11 Nock 4K

The primary change motivating 5K to 4K was the introduction of an edit operator, which ameliorated the proliferation of cells in the Nock runtime's memory. The edit operator is an optimization which makes modifications to a Nock data structure more efficient. It's an interesting example of a change motivated by the runtime rather than theoretical or higher-level language concerns.

Opcode 5 (equality) was rewritten to more explicit with application of the cell distribution rule. Opcodes 6–9 were rewritten to utilize the * operator rather than routing via opcode 2. Opcode 11 (formerly opcode 10) was likewise massaged. In general, preferring to express rules using * proved to be slightly more terse than utilizing opcode 2.

Listing 11: Nock 4K, October 2018.

Nock 4K

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

5  Reduce by the first matching pattern; variables match
    any noun.

nock(a)          *a
[a b c]          [a [b c]]

10 ?[a b]         0
   ?a            1
   +[a b]        +[a b]
   +a            1 + a
   =[a a]        0
15 =[a b]        1

/[1 a]           a
/[2 a b]         a
```

```

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

#[1 a b]          a
25 #[(a + a) b c] #[a [b /[(a + a + 1) c]] c]
#[(a + a + 1) b c] #[a [/[(a + a) c] b] c]
#a              #a

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

30 *[a 0 b]        /[b a]
*[a 1 b]          b
*[a 2 b c]        [*[a b] *[a c]]
*[a 3 b]          ?*[a b]
35 *[a 4 b]        +*[a b]
*[a 5 b c]        =[*[a b] *[a c]]

*[a 6 b c d]      *[a *[[c d] 0 *[[2 3] 0 *[a 4 4
                    b]]]]
*[a 7 b c]        [*[a b] c]
40 *[a 8 b c]      [*[*[a b] a] c]
*[a 9 b c]        [*[*[a c] 2 [0 1] 0 b]
*[a 10 [b c] d]    #[b *[a c] *[a d]]

*[a 11 [b c] d]   *[[*[a c] *[a d]] 0 3]
45 *[a 11 b c]     *[a c]

*a              *a

```

4 The Future of Nock

While deviations from the trunk line of the Nock family have been proposed at various points,⁴ Nock itself has remained the definitional substrate of Urbit since its inception. It has also been adopted as the primary ISA of Nockchain and the Nock-App ecosystem.

⁴Notably, Ax/Conk (see pp. XX-XX, this volume), SKEW, and PLAN (see pp. XX-XX, this volume).

Why, then, do we contemplate further changes? The `skew` proposal by `~siprel` and `~little-ponnys` argued that Nock 4K represented an undesirable saddle point in the design space of possible Nocks, itself a “ball of mud” (**Skew2020**, **Skew2020**). While `skew` itself was not adopted, it inspired the development of `PLAN` as a solid-state computing architecture sharing some ambitions with `Urbit` and `Nock`. A rigorously æsthetic argument can thus be sustained that Nock is not yet “close enough” to its final, diamond-perfect form to be a viable candidate.

While some have found this argument compelling, `Urbit`’s core developers have elected to maintain work in the “main line” of Nock as an ISA. The Nock 4K specification is a good candidate for a “final” version of Nock, as it is the most recent version and has been used in production for several years. It seems more likely that subsequent changes to Nock will derive not from alternative representations but from either dramatically more elegant expressions (e.g., of opcode 6) or from an implicit underspecification in the current Nock 4K which should be made explicit.

5 Conclusion

Nock began life as a hyper-Turing machine language, a theoretical construct for the purpose of defining higher-level programming languages with appropriate affordances and semantics. While its opcodes and syntax have gradually evolved over the course of two decades, the ambition to uproot the Unix “ball of mud” and replace it with a simple operating function amenable to reason has remained the north star of `Urbit` and `Nock`.

The most recent version, Nock 4K, appears to provide all of the opcodes necessary for correct and efficient⁵ evaluation. It is likely that future versions of Nock will be based genetically on Nock 4K, but with some changes to improve its performance and usability. The road to zero kelvin is likely very long still, given an abundance of caution, but it also appears to be straight.

⁵Modulo the vagaries of the von Neumann architecture, etc.