A Documentary 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

Manuscript submitted for review.

Address author correspondence to ~lagrev-nocfep.

1 Introduction

Nock is a combinator calculus which serves as the computational specification layer for the Urbit and Nockchain/NockApp 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. Yet for all this, Nock was not born from a purely mathematical approach, but found its roots in practical systems engineering.

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 μ-recursive functions. In practice, these amount to operations for constant, increment, variable access, program concatenation, and looping (Raphael, 2012). 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.

  Some Nock opcodes alter the subject (for instance 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: S, K, and I (Wolfram, 2021). These combinators are:

1. S substitution. $\mathrm{S}xyz = xz(yz)$, returns the first argument applied to the third, then applies this to the result of the second argument applied to the third. 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. K constant. $\mathrm{K}xy = x$, consumes its argument and returns a constant in all cases. This corresponds to Nock 4K's opcode 1, which yields its argument as a constant noun.

3. I identity. $\mathrm{I}x = x$, returns its argument. This corresponds to a special case of 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 isomorphic to SKI, Nock provides a set of primitive rules and a set of economic extended rules for convenience in writing a compiler.[1]

In an early document, Yarvin explained two of his design criteria in producing Nock as a practical ISA target (`~sorreg-namtyv`, 2010):

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.

[1] See **Galebach2025 (Galebach2025)**, pp. 1–45 in this volume, for exposition on how to evaluate a Nock expression by hand or by interpreter.

This latter idea he particularly connected to the concept of what came to be called 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 decrement of the previous version.[2] 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 were derived from dated public posts (`U`, 9K), internal dating (13K, 12K, 11K, 10K), or from Git commit history data (8K, 7K, 6K, 5K).[3] 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 (`~sorreg-namtyv`, 2025). 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 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

[2] This system, called "kelvin decrementing", draws on analogy with absolute zero as the lowest possible temperature—and thus most stable state.

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

the specification to converge on a sufficiently good set of opcodes. 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, 2006)

The earliest extant Nock is `U`, a proto-Nock posted to the *Lambda the Ultimate* blog in 2006 (`~sorreg-namtyv` (2006); `~sorreg-namtyv` (2006)).[4] The draft is versioned 0.15; subsequent evidence indicates that this is a downward-counting kelvin-versioned document already. The full specification is reproduced in Listing 1.[5]

Extensive commentary on the operators is provided. Rightwards grouping of tuple expressions has already been introduced. Extension of the language is summarily ruled out.[6] Data are conceived of as Unix-like byte streams; 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
    (S)      (S $a $b $c)         ($a $c ($b $c))
```

[4] Curtis 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.

[5] The original text has been slightly modified for print.

[6] Compare Ax and Conk, pp. XX–XX herein.

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 Madore's Unlambda and Burger's 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
    This document defines the U function and its data
    model.

2 License
    U is in the public domain.

3 Status
    This text is a DRAFT (version 0.15).

4 Data
    A value in U is called a "term."  There are three
    kinds of term: "number," "pair," and "foo."

```

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

A pair is an ordered pair of any two terms.

There is only one foo.

5 Syntax

U is a computational model, not a programming language.

But a trivial ASCII syntax for terms is useful.

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
        | foo

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

pair    : term white term
        | term white pair

foo     : 126

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

6 Semantics

U is a pure function from term to term.

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

6.1 Rules

```
    [name]   [pattern]                [definition]

    (a)      ($a 0 $b)                $b
    (b)      ($a 1 $b $c)             1
    (c)      ($a 1 $b)                0
    (d)      ($a 2 0 $b $c)           $b
    (e)      ($a 2 %n $b $c)          $c
    (f)      ($a 3 $b $c)             =($b $c)
    (g)      ($a 4 %n)                +%n

    (h)      ($a 5 (~ ~ $b) $c)       $b
    (i)      ($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 $d))
    (m)      ($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)

    (q)      ($a $b $c)     *($a 5 *($a 7 $b) $c)
    (r)      ($a $b)                  *($a $b)
    (s)      $a                       *$a
```

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 "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 syntax.

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

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

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

        - 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."

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

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.

    *$a is U($a).

6.6 The follow operator
    >($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.

        if *($a 5 $b $term) completes:
```

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

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

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

7 Implementation issues

This section is not normative.

7.1 The follow operator

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

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 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 of U. However, this is equivalent to standardizing a fixed trusted code base. The problems with this approach are well-known.

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.

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

    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.

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.

    (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.)

    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.

    No magic bullet can stop these unstandardized
    issues from becoming practical causes of lock-in
    and incompatibility. Systems which depend on U
    must manage them at every layer.
```

Source: `~sorreg-namtyv` (2006)

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 `*` tar[7] is identified as a `GOTO`.[8]

Listing 2: Nock 13K, 8 March 2008.

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

1. Manifest

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

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

2. Data

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

    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."

3. Pseudocode

    nock is defined in a pattern-matching pseudocode.

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

4. Definition

4.1 Transformations
```

[7] We refer to Nock axiomatic operators via their modern aural ASCII pronunciations. While these evolved over time (to wit, `^` "hat" became "ket"), to attempt to synchronize pronunciation with the era of a Nock release is a fool's errand.

[8] One can see the influence of this version's naming scheme on Atman's Ax, pp. XX–XX herein.

```
*(a 0 b c)   => *(*(a b) c)
*(a 0 b)     => /(b a)
*(a 1 b)     => (b)
*(a 2 b)     => **(a b)
*(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))
*(a)         => *(a)
```

4.2 Operators

4.2.1 Goto (*)

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

4.2.2 Deep (&)

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

4.2.3 Bump (^)

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

4.2.4 Same (=)

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

4.2.5 Snip (/)

```
/(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)             -> /(a)
```

Source: `~sorreg-namtyv` (2008)

3.3 Nock 12K

Opcodes were reordered slightly. Compound opcodes were introduced, such as a conditional branch and a static hint opcode.

Listing 3: Nock 12K, 2008.

```
Author: Curtis Yarvin (curtis.yarvin@gmail.com)
Date: 3/28/2008
Version: 0.12

1. Introduction

    This file defines one function, "nock."

    nock is in the public domain.

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,
    the "head" and "tail."

3. Semantics

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

4. Pseudocode

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

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

5. Definition
```

5.1 Transformations

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

*(a 7 b c d) => *(a 3 (0 1) 3 (1 c d) (1 0)
                   3 (1 2 3) (1 0) 5 5 b)
*(a 8 b c)   => *(a 2 (((1 0) b) c) 0 3)
*(a 9 b c)   => *(a c)

*(a)         => *(a)
```

5.2 Operators

5.2.1 Goto (*)

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

5.2.2 Deep (&)

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

5.2.4 Bump (^)

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

5.2.5 Same (=)

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

5.2.6 Snip (/)

```
    /(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)               -> /(a)
```

Source: ~sorreg-namtyv (2008)

3.4 Nock 11K

Opcodes were reordered slightly. The conditional branch was moved to 2. Composition, formerly at 2, was removed.

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

1. Introduction

    This file defines one function, "nock."

    nock is in the public domain.

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, the
    "head" and "tail."

3. Semantics

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

4. Pseudocode

```

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

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

5. Definition

5.1 Transformations

```
*(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)
*(a 3 b)     => **(a b)
*(a 4 b)     => &*(a b)
*(a 5 b)     => ^*(a b)
*(a 6 b)     => =*(a b)

*(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 (*)

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

5.2.2 Deep (&)

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

5.2.4 Bump (^)

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

5.2.5 Same (=)

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

5.2.6 Snip (/)

    /(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)             -> /(a)
```

Source: ~sorreg-namtyv (2008)

3.5 Nock 10K

Parentheses were replaced by brackets. Opcodes were reordered slightly. Hint syntax was 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 Moldbug [moldbug@gmail.com]
Date: 9/15/2008
Version: 10K

1. Introduction

   This file defines one function, "nock."

   nock is in the public domain.

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, the
   "head" and "tail."
```

```
3. Semantics

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

4. Pseudocode

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

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

5. Definition

5.1 Transformations

      *[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]
      *[a 3 b]     => **[a b]
      *[a 4 b]     => &*[a b]
      *[a 5 b]     => ^*[a b]
      *[a 6 b]     => =*[a b]
      *[a]         => *[a]

5.2 Operators

5.2.1 Goto [*]

      *[a]             -> nock[a]

5.2.2 Deep [&]

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

5.2.4 Bump [^]

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

```
5.2.5 Like [=]

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

5.2.6 Snip [/]

      /[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]              -> /[a]
```

Source: `~sorreg-namtyv` (2008)

3.6 Nock 9K

The cell detection axiomatic operator underlying opcode 4 (cell detection) was changed from `&` pam to `?` wut. Versus 10K, 9K elides operator names in favor of definitions. Other differences are likewise primarily terminological, such as the replacement of `Deep` `&` pam with `?` wut.

This version of Nock was published on the Moron Lab blog in 2010 (`~sorreg-namtyv`, 2010) 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 expounded on the practicality of building a 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.

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)
```

This remains the essential pattern followed to this day by higher-level languages targeting Nock as an ISA.

Yarvin had prepared to virtualize Nock interpretation to expose a broader namespace for interaction with values than the "strict" subject of a formula (`~sorreg-namtyv`, 2010).

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

```
1 Context

    This spec defines one function, Nock.

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.

3 Pseudocode

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

    *a is Nock(a).  Reductions match top-down.

4 Reductions

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

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

    =[a a]           => 0
```

```
=[a b]             => 1
=a                 => =a

/[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                 => /a

*[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]
*[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
```

Source: ~sorreg-namtyv (2010)

3.7 Nock 8K

The compound opcodes reappeared. Opcode 6 defined a conditional branch. Opcode 7 was described as a function composition operator. Opcode 8 served to define variables. Opcode 9 defined a calling convention. The remaining opcodes are hints, but each serving a different purpose:

11. consolidate for reference equality.

12. yield an arbitrary, unspecified hint.

13. label for acceleration (jet).

Nock 8K received an uncharacteristic amount of commentary, given a preprint document prepared for presentation at the 42nd ISCIE International Symposium on Stochastic Systems Theory and Its Applications (SSS'10) (~sorreg-namtyv, 2010).

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 ubiquitous core pattern is a convention rather than a requirement). Watt was revealed to have a different ASCII pronunciation convention than Nock at this stage.

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.

2 Pseudocode

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

3 Reductions

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

```
*[a 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]
*[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]

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

Source: ~sorreg-namtyv (2010)

3.8 Nock 7K

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.

The three hint opcodes were refactored into two, a static and a dynamic hint, both at 10.

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.

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

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

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

Source: ~sorreg-namtyv (2010)

3.9 Nock 6K

The axiomatic operator for increment was changed from ^ ket to + lus. Compound opcode syntax was reworked slightly.

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
  nouns.

2 Reductions

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

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

  *[a 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 [[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]

  +[a b]              +[a b]
  =a                  =a
  /a                  /a
  *a                  *a
```

Source: ~sorreg-namtyv (2011)

3.10 Nock 5K

Compound opcode syntax was reworked slightly. 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 (dd779c1).)

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.

2 Reductions

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

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

  /[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                  /a

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

  *[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]
*[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 [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]
*[a 10 b c]         *[a c]

*a                  *a
```

Source: ~sorreg-namtyv (2012)

3.11 Nock 4K

The primary change motivating 5K to 4K was the introduction of an edit operator # hax, which ameliorated the proliferation of cells in the Nock runtime's memory.[9] The edit operator is an optimization which makes modifications to a Nock data structure more efficient. It's a notable example of a change motivated by the pragmatics of the runtime rather than theoretical or higher-level language concerns.[10]

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

Listing 11: Nock 4K, *terminus ad quem* 27 September 2018.

```
Nock 4K

A noun is an atom or a cell.  An atom is a natural
```

[9] The date must be earlier than 27 September 2018; cf. urbit/urbit #1027.

[10] See ~niblyx-malnus, pp. XX–XX, this volume, for a verbose derivation of the edit operator and opcode 10 from the primitive opcodes.

```
number.  A cell is an ordered pair of nouns.

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

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

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

#[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]
#a                  #a

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

*[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]
*[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]
*[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]
*[a 11 b c]         *[a c]

*a                  *a
```

Source: ~sorreg-namtyv (2018-09-27)

4 The Future of Nock

While deviations from the trunk line of the Nock family have been propsed at various points,[11] 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 NockApp ecosystem.

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" (~siprel and ~littel-ponnys, 2020). While SKEW itself was not adopted, it inspired the development of Plunder and PLAN as a solid-state computing architecture sharing some ambitions with Urbit and Nock (~siprel and ~littel-ponnys, 2023). 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 traditional Nock as the system's target ISA. The Nock 4K specification is a good candidate, in this sense, for a "final" version of Nock, as it has been successfully 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 a combinator refactor) or from an implicit underspecification in the current Nock 4K which should be made explicit.

[11] Notably, Ax (see pp. XX–XX, this volume), SKEW, and PLAN (see pp. XX–XX, this volume).

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 history of Nock serves as an index of refinement as Yarvin and contributors sought to balance conciseness, efficiency, and practicality.

The most recent version, Nock 4K, appears to provide all of the opcodes necessary for correct and efficient[12] 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.

References

Burger, Alexander (2006) "Pico Lisp: A Radical Approach to Application Development". URL: `https://software-lab.de/radical.pdf` (visited on ~2025.5.19).

Madore, David (2003) "The Unlambda Programming Language". URL: `http://www.madore.org/~david/programs/unlambda/` (visited on ~2025.5.19).

Raphael (2012) "Are there minimum criteria for a programming language being Turing complete?" URL: `https://cs.stackexchange.com/questions/991/are-there-minimum-criteria-for-a-programming-language-being-turing-complete` (visited on ~2025.5.19).

[12] Modulo the vagaries of the von Neumann architecture, etc.

~siprel, Benjamin and Elliot Glaysher ~littel-ponnys (2020) "SKEW". URL: https://github.com/urbit/urbit/blob/skew/pkg/hs/urbit-skew/skew.md (visited on ~2025.5.19).

— (2023) "What is Plunder?" URL: https://plunder.tech/ (visited on ~2025.7.6).

~sorreg-namtyv, Curtis Yarvin (2006a) "U, a small model". URL: http://lambda-the-ultimate.org/node/1269 (visited on ~2024.2.20).

— (2006b) "U, a small model". URL: http://urbit.sourceforge.net/u.txt (visited on ~2024.2.20).

— (2008a) "Nock 10K". URL: https://github.com/urbit/archaeology/blob/0b228203e665579848d30c763dda55bb107b0a34/Spec/nock/10.txt (visited on ~2024.2.20).

— (2008b) "Nock 11K". URL: https://github.com/urbit/archaeology/blob/0b228203e665579848d30c763dda55bb107b0a34/Spec/nock/11.txt (visited on ~2024.2.20).

— (2008c) "Nock 12K". URL: https://github.com/urbit/archaeology/blob/0b228203e665579848d30c763dda55bb107b0a34/Spec/nock/12.txt (visited on ~2024.2.20).

— (2008d) "Nock 13K". URL: https://github.com/urbit/archaeology/blob/0b228203e665579848d30c763dda55bb107b0a34/Spec/nock/13.txt (visited on ~2024.2.20).

— (2010a) "Nock 7K". URL: https://github.com/urbit/archaeology/blob/0b228203e665579848d30c763dda55bb107b0a34/Spec/nock/7.txt (visited on ~2024.2.20).

— (2010b) "Nock 8K". URL: https://github.com/urbit/archaeology/blob/0b228203e665579848d30c763dda55bb107b0a34/Spec/nock/8.txt (visited on ~2024.2.20).

— (2010c) "Nock 9K". URL: https://github.com/urbit/archaeology/blob/

0b228203e665579848d30c763dda55bb107b0a34/Spec/nock/9.txt (visited on ~2024.2.20).

— (2010d) "Nock: Maxwell's equations of software". URL: http://moronlab.blogspot.com/2010/01/nock-maxwells-equations-of-software.html (visited on ~2024.1.25).

— (2010e) "Urbit: functional programming from scratch". URL: http://moronlab.blogspot.com/2010/01/urbit-functional-programming-from.html (visited on ~2024.1.25).

— (2010f) "Why Nock?" URL: https://github.com/cgyarvin/urbit/blob/gh-pages/Spec/urbit/5-whynock.txt (visited on ~2025.5.19).

— (2011) "Nock 6K". URL: https://github.com/urbit/archaeology/blob/0b228203e665579848d30c763dda55bb107b0a34/Spec/nock/6.txt (visited on ~2024.2.20).

— (2012) "Nock 5K". URL: https://github.com/urbit/archaeology/blob/0b228203e665579848d30c763dda55bb107b0a34/Spec/nock/5.txt (visited on ~2024.2.20).

— (2025). "Urbit: Making the Future Real." In: *LambdaConf 2025*. Accessed: 2025-05-19. Estes Park, Colorado: LambdaConf. URL: https://www.lambdaconf.us/ (visited on ~2025.5.19).

— (2018-09-27) "Nock 4K". URL: https://docs.urbit.org/language/nock/reference/definition (visited on ~2024.2.20).

Wolfram, Stephen (2021). *Combinators: A Centennial View.* Accessed: 2025-05-19. Champaign, IL: Wolfram Media. URL: https://www.wolfram.com/combinators/ (visited on ~2025.5.19).