

1  
2  
3  
4  
5  
6  
7  
8  
9  
10  
11  
12  
13  
14  
15  
16  
17  
18  
19  
20  
21  
22  
23  
24

---

# 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

<b>1</b>	<b>Introduction</b>	<b>2</b>
<b>2</b>	<b>Nock as a Combinator Calculus</b>	<b>2</b>
<b>3</b>	<b>Nock’s Decrement</b>	<b>6</b>
3.1	U . . . . .	7
3.2	Nock 13K . . . . .	13
3.3	Nock 12K . . . . .	16
3.4	Nock 11K . . . . .	18
3.5	Nock 10K . . . . .	20

25	3.6	Nock 9K . . . . .	22
26	3.7	Nock 8K . . . . .	24
27	3.8	Nock 7K . . . . .	26
28	3.9	Nock 6K . . . . .	27
29	3.10	Nock 5K . . . . .	29
30	3.11	Nock 4K . . . . .	30
31	<b>4</b>	<b>The Future of Nock</b>	<b>32</b>
32	<b>5</b>	<b>Conclusion</b>	<b>33</b>

## 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. 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  $\mu$ -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, κ, and I (Wolfram, 2021). These combinators are:

- 131 1. *s* substitution.  $sxyz = xz(yz)$ , returns the first argu-  
132 ment applied to the third, then applies this to the re-  
133 sult of the second argument applied to the third. This  
134 corresponds to Nock 4K's opcode 2, which substitutes  
135 the second argument into the first argument at the third  
136 argument's axis. (There are some subtle differences to  
137 Nock's expression of *s* as opcode 2 that we will elide as  
138 being fundamentally similar, but perhaps worthy of its  
139 own monograph.)
- 140 2. *κ* constant.  $κxy = x$ , consumes its argument and returns  
141 a constant in all cases. This corresponds to Nock 4K's  
142 opcode 1, which yields its argument as a constant noun.
- 143 3. *i* identity.  $ix = x$ , returns its argument. This corre-  
144 sponds to a special case of Nock 4K's opcode 0, a gener-  
145 alized axis lookup operator, which can trivially retrieve  
146 the current subject or expression as well as any children.

147 While Nock introduces a few more primitive operations as a  
148 practicality, the above identities establish its bona fides as a  
149 combinator calculus capable of general computation. Similar  
150 to Haskell Curry's BCKW system, which can be written in forms  
151 isomorphic to SKI, Nock provides a set of primitive rules and  
152 a set of economic extended rules for convenience in writing a  
153 compiler.<sup>1</sup>

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

- 157 1. Natural conversion of source to code without other in-  
158 puts.
- 159 2. Metacircularity without deep stacks; i.e., the ability to  
160 extend Nock semantics without altering the underlying  
161 substrate.

---

<sup>1</sup>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.<sup>2</sup> 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).<sup>3</sup> 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

---

<sup>2</sup>This system, called “kelvin decrementing”, draws on analogy with absolute zero as the lowest possible temperature—and thus most stable state.

<sup>3</sup>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 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. (~sorreg-namtyv<sup>4</sup>, 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)).<sup>5</sup> 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.<sup>6</sup>

Extensive commentary on the operators is provided. Rightwards grouping of tuple expressions has already been introduced. Extension of the language is summarily ruled out.<sup>7</sup> 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

---

<sup>4</sup>*Avant la lettre.*

<sup>5</sup>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.

<sup>6</sup>The original text has been slightly modified for print.

<sup>7</sup>Compare Ax and Conk, pp. XX–XX herein.

225 (S) (S \$a \$b \$c) (\$a \$c (\$b \$c))  
226

---

227 While early work (1940s–50s) had been carried out on  
228 “minimal instruction set computers” (MISCs), it is more likely  
229 that Yarvin was influenced by contemporaneous work on “re-  
230 duced instruction set computers” (RISCs) in the 1980s and  
231 90s. Language proposals like that of Madore’s Unlambda and  
232 Burger’s Pico Lisp may have influenced Yarvin’s design choices  
233 throughout this era.

234 The U specification is in some ways the single most inter-  
235 esting historical document of our series. Yarvin particularly  
236 identified a desire to avoid baking abstractions like variables  
237 and functions into the U cake, and an emphasis on client–server  
238 semantics. The scry namespace appears *avant la lettre* as a re-  
239 ferentially transparent immutable distributed namespace. U ex-  
240 presses a very ambitious hyper-Turing operator, acknowledg-  
241 ing that its own instantiation from the specification is impos-  
242 sible and approximate. Yarvin grapples in U with the halting  
243 problem (via his follow operator) and with the tension between  
244 a specification and an implementation (a gulf he highlighted as  
245 a human problem in his 2025 LambdaConf keynote address).  
246 Furthermore, asides on issues like the memory arena prefigure  
247 implementation details of Vere as a runtime.

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

---

248 U: Definition  
249  
250  
251 1 Purpose  
252 This document defines the U function and its data  
253 model.  
254  
255 2 License  
256 U is in the public domain.  
257  
258 3 Status  
259 This text is a DRAFT (version 0.15).  
260  
261 4 Data  
262 A value in U is called a “term.” There are three



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

264

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

267

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

269

270 There is only one foo.

271

## 272 5 Syntax

223 U is a computational model, not a programming  
274 language.

275

276 But a trivial ASCII syntax for terms is useful.

277

### 278 5.1 Trivial syntax: briefly

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

281

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

284

### 285 5.2 Trivial syntax: exactly

286 term : number  
287 | 40 ?white pair ?white 41  
288 | foo

289

290 number : 48  
291 | [49-57] \*[48-57]

292

293 pair : term white term  
294 | term white pair

295

296 foo : 126

297

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

299

## 300 6 Semantics

301 U is a pure function from term to term.

302

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

305

## 306 6.1 Rules

307	[name]	[pattern]	[definition]
308			
309	(a)	(\$a 0 \$b)	\$b
310	(b)	(\$a 1 \$b \$c)	1
311	(c)	(\$a 1 \$b)	0
312	(d)	(\$a 2 0 \$b \$c)	\$b
313	(e)	(\$a 2 %n \$b \$c)	\$c
314	(f)	(\$a 3 \$b \$c)	=( \$b \$c )
315	(g)	(\$a 4 %n)	+%n
316			
317	(h)	(\$a 5 (~ ~ \$b) \$c)	\$b
318	(i)	(\$a 5 (~ \$b \$c) \$d)	*( \$a \$b \$c \$d )
319	(j)	(\$a 5 (~ ~) \$b)	~
320	(k)	(\$a 5 (~ \$b) \$c)	*( \$a \$b \$c )
321	(l)	(\$a 5 (\$b \$c) \$d)	*( \$a \$b \$d ) *( \$a \$c \$d )
322			
323	(m)	(\$a 5 \$b \$c)	\$b
324			
325	(n)	(\$a 6 \$b \$c)	*( \$a *( \$a 5 \$b \$c ) )
326	(o)	(\$a 7 \$b)	*( \$a 5 \$a \$a \$b )
327	(p)	(\$a 8 \$b \$c \$d)	> ( \$b \$c \$d )
328			
329	(q)	(\$a \$b \$c)	*( \$a 5 *( \$a 7 \$b ) \$c )
330	(r)	(\$a \$b)	*( \$a \$b )
331	(s)	\$a	*\$a

332

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

335

## 336 6.2 Rule pseudocode: briefly

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

339

## 340 6.3 Rule pseudocode: exactly

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

344

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

```

347     operation.
348
349     Variables are symbols marked with a constraint.
350     A variable "$name" matches any term.  "%name"
351     matches any number.
352
353     There are four unary prefix operators, each of
354     which is a pure function from term to term: "=",
355     "+", "*", and ">". Their semantics follow.
356
357 6.4 Evaluation semantics
358     For any term $term, to compute U($term):
359
360         - find the first pattern, in order, that
361           matches $term.
362         - substitute its variable matches into its
363           definition.
364         - compute the substituted definition.
365
366     Iff this sequence of steps terminates, U($term)
367     "completes." Otherwise it "chokes."
368
369     Evaluation is strict: incorrect completion is a
370     bug. Choking is U's only error or exception
371     mechanism.
372
373 6.5 Simple operators: equal, increment, evaluate
374     =($a $b) is 0 if $a and $b are equal; 1 if they
375     are not.
376
377     +%n is %n plus 1.
378
379     *$a is U($a).
380
381 6.6 The follow operator
382     >($a $b $c) is always 0. But it does not always
383     complete.
384
385     We say "$c follows $b in $a" iff, for every $term:
386
387         if *($a 5 $b $term) chokes:
388             *($a 5 $c $term) chokes.

```

```

389
390         if *($a 5 $b $term) completes:
391             either:
392                 *($a 5 $c $term) completes, and
393                 *($a 5 $c $term) equals
394                 *($a 5 $b $term)
395             or:
396                 *($a 5 $c $term) chokes.
397
398     If $c follows $b in $a, >($a $b $c) is 0.
399
400     If this statement cannot be shown (ie, if there
401     exists any $term that falsifies it, generates an
402     infinitely recursive series of follow tests, or is
403     inversely self-dependent, ie, exhibits Russell's
404     paradox), >($a $b $c) chokes.
405
406 7 Implementation issues
407     This section is not normative.
408
409 7.1 The follow operator
410     Of course, no algorithm can completely implement
411     the follow operator. So no program can completely
412     implement U.
413
414     But this does not stop us from stating the
415     correctness of a partial implementation - for
416     example, one that assumes a hardcoded set of
417     follow cases, and fails when it would otherwise
418     have to compute a follow case outside this set.
419
420     U calls this a "trust failure." One way to
421     standardize trust failures would be to standardize
422     a fixed set of follow cases as part of the
423     definition of U. However, this is equivalent to
424     standardizing a fixed trusted code base. The
425     problems with this approach are well-known.
426
427     A better design for U implementations is to
428     depend on a voluntary, unstandardized failure
429     mechanism. Because all computers have bounded
430     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<sup>8</sup> is identified as a GOTO.<sup>9</sup>

---

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

---

<sup>8</sup>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.

<sup>9</sup>One can see the influence of this version's naming scheme on Atman's Ax, pp. XX-XX herein.

506

## 507 4.1 Transformations

508

```

509      *(a 0 b c) => *((*(a b) c)
510      *(a 0 b)   => /(b a)
511      *(a 1 b)   => (b)
512      *(a 2 b)   => ***(a b)
513      *(a 3 b)   => &*(a b)
514      *(a 4 b)   => ^*(a b)
515      *(a 5 b)   => =*(a b)
516      *(a 6 b c d) => *(a 2 (0 1)
                        2 (1 c d) (1 0)
517                        2 (1 2 3) (1 0) 4 4 b)
518      *(a b c)   => (*(a b) *(a c))
519      *(a)        => *(a)

```

521

## 522 4.2 Operators

523

## 524 4.2.1 Goto (\*)

525

```

526      *(a)          -> nock(a)

```

527

## 528 4.2.2 Deep (&amp;)

529

```

530      &(a b)         -> 0
531      &(a)           -> 1

```

532

## 533 4.2.3 Bump (^)

534

```

535      ^(a b)         -> ^(a b)
536      ^(a)           -> a + 1

```

537

## 538 4.2.4 Same (=)

539

```

540      =(a a)         -> 0
541      =(a b)         -> 1
542      =(a)           -> =(a)

```

543

## 544 4.2.5 Snip (/)

545

```

546      /(1 a)         -> a
547      /(2 a b)       -> a

```

```

548      /(3 a b)          -> b
549      /((a + a) b)      -> /(2 /(a b))
550      /((a + a + 1) b) -> /(3 /(a b))
551      /(a)              -> /(a)
552

```

---

553 Source: ~sorreg-namtyv (2008)

### 554 3.3 Nock 12K

555 Opcodes were reordered slightly. Compound opcodes were in-  
556 troduced, such as a conditional branch and a static hint opcode.

#### Listing 3: Nock 12K, 2008.

---

```

557 Author: Curtis Yarvin (curtis.yarvin@gmail.com)
558 Date: 3/28/2008
559 Version: 0.12
560

```

#### 561 1. Introduction

```

562
563
564     This file defines one function, "nock."
565
566     nock is in the public domain.
567

```

#### 568 2. Data

```

569
570     A "noun" is either an "atom" or a "cell." An
571     "atom" is an unsigned integer of any size. A
572     "cell" is an ordered pair of any two nouns,
573     the "head" and "tail."
574

```

#### 575 3. Semantics

```

576
577     nock maps one noun to another. It doesn't
578     always terminate.
579

```

#### 580 4. Pseudocode

```

581
582     nock is defined in a pattern-matching
583     pseudocode, below.
584
585     Parentheses enclose cells. (a b c) is
586     (a (b c)).

```



```

587
588 5. Definition
589
590 5.1 Transformations
591
592      *(a (b c) d) => (*(a b c) *(a d))
593      *(a 0 b)      => /(b a)
594      *(a 1 b)      => (b)
595      *(a 2 b c)     => **(*(a b) c)
596      *(a 3 b)       => **(a b)
597      *(a 4 b)       => &*(a b)
598      *(a 5 b)       => ^*(a b)
599      *(a 6 b)       => =(a b)
600
601      *(a 7 b c d)   => *(a 3 (0 1) 3 (1 c d) (1 0)
602                        3 (1 2 3) (1 0) 5 5 b)
603      *(a 8 b c)     => *(a 2 (((1 0) b) c) 0 3)
604      *(a 9 b c)     => *(a c)
605
606      *(a)           => *(a)
607
608 5.2 Operators
609
610 5.2.1 Goto (*)
611
612      *(a)           -> nock(a)
613
614 5.2.2 Deep (&)
615
616      &(a b)          -> 0
617      &(a)            -> 1
618
619 5.2.4 Bump (^)
620
621      ^ (a b)         -> ^ (a b)
622      ^ (a)           -> a + 1
623
624 5.2.5 Same (=)
625
626      =(a a)          -> 0
627      =(a b)          -> 1
628      =(a)            -> =(a)

```

```

629
630 5.2.6 Snip (/)
631
632     /(1 a)           -> a
633     /(2 a b)         -> a
634     /(3 a b)         -> b
635     /((a + a) b)     -> /(2 /(a b))
636     /((a + a + 1) b) -> /(3 /(a b))
637     /(a)             -> /(a)
638

```

---

639 Source: ~sorreg-namtyv (2008)

## 640 3.4 Nock 11K

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

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

Listing 4: Nock 11K, 25 May 2008.

---

```

645
646 Author: Mencius Moldbug (moldbug@gmail.com)
647 Date: 5/25/2008
648 Version: 11K
649

```

### 650 1. Introduction

651 This file defines one function, "nock."

652  
653 nock is in the public domain.

### 656 2. Data

657  
658 A "noun" is either an "atom" or a "cell." An  
659 "atom" is an unsigned integer of any size. A  
660 "cell" is an ordered pair of any two nouns, the  
661 "head" and "tail."

### 663 3. Semantics

664  
665 nock maps one noun to another. It doesn't always  
666 terminate.

667

## 668 4. Pseudocode

669

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

672

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

674

## 675 5. Definition

676

### 677 5.1 Transformations

678

```

679       *(a (b c) d) => (*(a b c) *(a d))
680       *(a 0 b)     => /(b a)
681       *(a 1 b)     => (b)
682       *(a 2 b c d) => *(a 3 (0 1) 3 (1 c d) (1 0)
683                      3 (1 2 3) (1 0) 5 5 b)
684       *(a 3 b)     => **(a b)
685       *(a 4 b)     => &*(a b)
686       *(a 5 b)     => ^*(a b)
687       *(a 6 b)     => =*(a b)
688
689       *(a 7 b c)    => *(a 3 (((1 0) b) c) 1 0 3)
690       *(a 8 b c)    => *(a c)
691
692       *(a)          => *(a)

```

693

### 694 5.2 Operators

695

#### 696 5.2.1 Goto (\*)

697

```

698       *(a)               -> nock(a)

```

699

#### 700 5.2.2 Deep (&)

701

```

702       &(a b)            -> 0
703       &(a)              -> 1

```

704

#### 705 5.2.4 Bump (^)

706

```

707       ^(a b)            -> ^(a b)
708       ^(a)              -> a + 1

```

709

710 5.2.5 Same (=)

```
711
712     = (a a)           -> 0
713     = (a b)           -> 1
714     = (a)             -> =(a)
```

715

716 5.2.6 Snip (/)

```
717
718     / (1 a)           -> a
719     / (2 a b)         -> a
720     / (3 a b)         -> b
721     / ((a + a) b)     -> / (2 / (a b))
722     / ((a + a + 1) b) -> / (3 / (a b))
723     / (a)             -> / (a)
```

725 Source: ~sorreg-namtyv (2008)

726 3.5 Nock 10K

727 Parentheses were replaced by brackets. Opcodes were re-  
 728 ordered slightly. Hint syntax was removed. Functionally, 11K  
 729 and 10K appear very similar, particularly if the Watt (proto-  
 730 Hoon) compiler is set up to produce variable declarations and  
 731 compositions as the compound opcodes had them.

---

Listing 5: Nock 10K, 15 September 2008.

---

```
732
733 Author: Mencius Moldbug [moldbug@gmail.com]
734 Date: 9/15/2008
735 Version: 10K
```

736

737 1. Introduction

738

739 This file defines one function, "nock."

740

741 nock is in the public domain.

742

743 2. Data

744

745 A "noun" is either an "atom" or a "cell." An  
 746 "atom" is an unsigned integer of any size. A  
 747 "cell" is an ordered pair of any two nouns, the

748 "head" and "tail."

749

### 750 3. Semantics

751

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

754

### 755 4. Pseudocode

756

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

759

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

761

### 762 5. Definition

763

#### 764 5.1 Transformations

765

766 \*[a [b c] d] => [\*[a b c] \*[a d]]

767 \*[a 0 b] => /[b a]

768 \*[a 1 b] => [b]

769 \*[a 2 b c d] => \*[a 3 [0 1] 3 [1 c d]

770 [1 0] 3 [1 2 3] [1 0] 5 5 b]

771 \*[a 3 b] => \*\*[a b]

772 \*[a 4 b] => &\*[a b]

773 \*[a 5 b] => ^\*[a b]

774 \*[a 6 b] => =\*[a b]

775 \*[a] => \*[a]

776

#### 777 5.2 Operators

778

##### 779 5.2.1 Goto [\*]

780

781 \*[a] -> nock[a]

782

##### 783 5.2.2 Deep [&]

784

785 &[a b] -> 0

786 &[a] -> 1

787

##### 788 5.2.4 Bump [^]

789

```

790      ^[a b]          -> ^[a b]
791      ^[a]            -> (a + 1)
792
793 5.2.5 Like [=]
794
795      =[a a]          -> 0
796      =[a b]          -> 1
797      =[a]             -> =[a]
798
799 5.2.6 Snip [/]
800
801      /[1 a]           -> a
802      /[2 a b]          -> a
803      /[3 a b]          -> b
804      /[(a + a) b]      -> /[2 /[a b]]
805      /[(a + a + 1) b] -> /[3 /[a b]]
806      /[a]              -> /[a]

```

---

808 Source: ~sorreg-namtyv (2008)

## 809 3.6 Nock 9K

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

815 This version of Nock was published on the Moron Lab blog  
816 in 2010 (~sorreg-namtyv, 2010) as “Maxwell’s equations of  
817 software”. Yarvin emphasized that Nock was intended to serve  
818 as “foundational system software rather than foundational  
819 metamathematics” (ibid.). Yarvin also publicly expounded on  
820 the practicality of building a higher-level language on top of  
821 Nock at this point (ibid.):

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

- 826 • p is the function source;

- 827 • *q* is your language definition, as source;
- 828 • *r* is your language definition, as data;
- 829 • *s* is the input data.

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

---

```
831 urbit-formula == Watt(urbit-source)
832               == Nock(urbit-source watt-formula)
833 watt-formula  == Watt(watt-source)
834               == Nock(watt-source watt-formula)
```

---

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

839 Yarvin had prepared to virtualize Nock interpretation to ex-  
840 pose a broader namespace for interaction with values than the  
841 “strict” subject of a formula (~sorreg-namtyv, 2010).

---

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

---

```
842 1 Context
843
844     This spec defines one function, Nock.
845
846 2 Structures
847
848     A noun is an atom or a cell. An atom is any
849     unsigned integer. A cell is an ordered pair of
850     any two nouns.
851
852 3 Pseudocode
853
854     Brackets enclose cells. [a b c] is [a [b c]].
855
856     *a is Nock(a). Reductions match top-down.
857
858 4 Reductions
859
860     ?[a b]          => 0
861     ?a              => 1
862
863     ^[a b]          => ^[a b]
864     ^a              => (a + 1)
```

```

866
867   = [a a]           => 0
868   = [a b]           => 1
869   = a               => = a
870
871   /[1 a]            => a
872   /[2 a b]          => a
873   /[3 a b]          => b
874   /[(a + a) b]      => /[2 /[a b]]
875   /[(a + a + 1) b] => /[3 /[a b]]
876   /a                => /a
877
878   *[a 0 b]          => /[b a]
879   *[a 1 b]          => b
880   *[a 2 b c d]      => *[a 3 [0 1] 3 [1 c d] [1 0]
881                        3 [1 2 3] [1 0] 5 5 b]
882   *[a 3 b]          => **[a b]
883   *[a 4 b]          => ?*[a b]
884   *[a 5 b]          => ^*[a b]
885   *[a 6 b]          => =*[a b]
886   *[a [b c] d]      => [*[a b c] *[a d]]
887   *a                => *a
888

```

---

889       Source: ~sorreg-namtyv (2010)

## 890   3.7   Nock 8K

891   The compound opcodes reappeared. Opcode 6 defined a con-  
892   ditional branch. Opcode 7 was described as a function compo-  
893   sition operator. Opcode 8 served to define variables. Opcode 9  
894   defined a calling convention. The remaining opcodes are hints,  
895   but each serving a different purpose:

- 896       11. consolidate for reference equality.
- 897       12. yield an arbitrary, unspecified hint.
- 898       13. label for acceleration (jet).

899       Nock 8K received an uncharacteristic amount of commen-  
900   tary, 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]
```

```

941      *[a 4 b]          ^*[a b]
942      *[a 5 b]          =*[a b]
943
944      *[a 6 b c d]       *[a 2 [0 1] 2 [1 c d] [1 0]
945                          2 [1 2 3] [1 0] 4 4 b]
946      *[a 7 b c]         *[a 2 b 1 c]
947      *[a 8 b c]         *[a 7 [7 b [0 1]] c]
948      *[a 9 b c]         *[a 8 b 2 [[7 [0 3] d] [0 5]]
949                          0 5]
950      *[a 10 b c]        *[a 8 b 8 [7 [0 3] c] 0 2]
951      *[a 11 b c]        *[a 8 b 7 [0 3] c]
952      *[a 12 b c]        *[a [1 0] 1 c]
953
954      ^[a b]             ^[a b]
955      = a                 = a
956      /a                  /a
957      *a                  *a
958

```

---

959 Source: ~sorreg-namtyv (2010)

## 960 3.8 Nock 7K

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

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

---

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

---

### 967 1 Structures

968  
969 A noun is an atom or a cell. An atom is any  
970 natural number. A cell is any ordered pair of  
971 nouns.  
972

### 973 2 Pseudocode

```

974      [a b c]             [a [b c]]
975      nock(a)             *a
976
977
978

```

```

979  ?[a b]           0
980  ?a              1
981  ^a             1 + a
982  =[a a]          0
983  =[a b]          1
984
985  /[1 a]           a
986  /[2 a b]         a
987  /[3 a b]         b
988  /[(a + a) b]     /[2 /[a b]]
989  /[(a + a + 1) b] /[3 /[a b]]
990
991  *[a [b c] d]      [*[a b c] *[a d]]
992
993  *[a 0 b]          /[b a]
994  *[a 1 b]          b
995  *[a 2 b c]        [*[a b] *[a c]]
996  *[a 3 b]          ?*[a b]
997  *[a 4 b]          ^*[a b]
998  *[a 5 b]          =*[a b]
999
1000  *[a 6 b c d]      *[a 2 [0 1] 2 [1 c d] [1 0]
1001                        2 [1 2 3] [1 0] 4 4 b]
1002
1003  *[a 7 b c]         *[a 2 b 1 c]
1004  *[a 8 b c]         *[a 7 [[7 [0 1] b] 0 1] c]
1005  *[a 9 b c]         *[a 7 c 0 b]
1006  *[a 10 b c]        *[a c]
1007  *[a 10 [b c] d]    *[a 8 c 7 [0 3] d]
1008
1009  ^[a b]            ^[a b]
1010  =a                =a
1011  /a                /a
1012  *a                *a

```

---

1013     Source: ~sorreg-namtyv (2010)

### 1014   3.9   Nock 6K

1015   The axiomatic operator for increment was changed from ^ ket  
1016   to + lus. Compound opcode syntax was reworked slightly.

---

Listing 9: Nock 6K, 6 July 2011.

---

1017

1018 1 Structures

1019

1020 A noun is an atom or a cell. An atom is any  
1021 natural number. A cell is an ordered pair of  
1022 nouns.

1023

1024 2 Reductions

1025

1026 `nock(a)` `*a`  
1027 `[a b c]` `[a [b c]]`

1028

1029 `?[a b]` `0`

1030 `?a` `1`

1031 `+a` `1 + a`

1032 `=[a a]` `0`

1033 `=[a b]` `1`

1034

1035 `/[1 a]` `a`

1036 `/[2 a b]` `a`

1037 `/[3 a b]` `b`

1038 `/[(a + a) b]` `/[2 /[a b]]`

1039 `/[(a + a + 1) b]` `/[3 /[a b]]`

1040

1041 `*[a [b c] d]` `[*[a b c] *[a d]]`

1042

1043 `*[a 0 b]` `/[b a]`

1044 `*[a 1 b]` `b`

1045 `*[a 2 b c]` `*[*[a b] *[a c]]`

1046 `*[a 3 b]` `?*[a b]`

1047 `*[a 4 b]` `++[a b]`

1048 `*[a 5 b]` `==[a b]`

1049

1050 `*[a 6 b c d]` `*[a 2 [0 1] 2 [1 c d] [1 0]`  
1051 `2 [1 2 3] [1 0] 4 4 b]`

1052 `*[a 7 b c]` `*[a 2 b 1 c]`

1053 `*[a 8 b c]` `*[a 7 [[0 1] b] c]`

1054 `*[a 9 b c]` `*[a 7 c 0 b]`

1055 `*[a 10 b c]` `*[a c]`

1056 `*[a 10 [b c] d]` `*[a 8 c 7 [0 2] d]`

1057

1058 `+ [a b]` `+ [a b]`

1059 `= a` `= a`

```

1060      / a                / a
1061      * a                * a
1062

```

---

```

1063      Source: ~sorreg-namtyv (2011)

```

### 1064 3.10 Nock 5K

1065 Compound opcode syntax was reworked slightly. All trivial  
 1066 reductions of axiomatic operators were removed to the preface  
 1067 of the specification.

1068 (For instance, a trivial “cosmetic” change was made to 5K’s  
 1069 specification after it was publicly posted in order to synchro-  
 1070 nize it with the VM’s behavior (dd779c1).)

Listing 10: Nock 5K, 24 September 2012.

---

#### 1071 1 Structures

```

1072
1073
1074      A noun is an atom or a cell. An atom is any natural
1075      number. A cell is an ordered pair of nouns.

```

#### 1077 2 Reductions

```

1078
1079      nock(a)                * a
1080      [a b c]                [a [b c]]
1081
1082      ?[a b]                  0
1083      ?a                      1
1084      +[a b]                  +[a b]
1085      +a                      1 + a
1086      =[a a]                  0
1087      =[a b]                  1
1088      =a                      =a
1089
1090      /[1 a]                  a
1091      /[2 a b]                a
1092      /[3 a b]                b
1093      /[(a + a) b]            /[2 /[a b]]
1094      /[(a + a + 1) b]        /[3 /[a b]]
1095      /a                      /a
1096
1097      *[a [b c] d]            *[a b c] *[a d]]

```

```

1098
1099   * [a 0 b]           / [b a]
1100   * [a 1 b]           b
1101   * [a 2 b c]         * [* [a b] * [a c]]
1102   * [a 3 b]           ? * [a b]
1103   * [a 4 b]           + * [a b]
1104   * [a 5 b]           = * [a b]
1105
1106   * [a 6 b c d]        * [a 2 [0 1] 2 [1 c d] [1 0] 2
1107                           [1 2 3] [1 0] 4 4 b]
1108   * [a 7 b c]          * [a 2 b 1 c]
1109   * [a 8 b c]          * [a 7 [[7 [0 1] b] 0 1] c]
1110   * [a 9 b c]          * [a 7 c 2 [0 1] 0 b]
1111   * [a 10 [b c] d]     * [a 8 c 7 [0 3] d]
1112   * [a 10 b c]         * [a c]
1113
1114   * 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.<sup>10</sup> 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.<sup>11</sup>

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.

---

<sup>10</sup>The date must be earlier than 27 September 2018; cf. urbit/urbit #1027.

<sup>11</sup>See ~niblyx-malnu, pp. XX–XX, this volume, for a verbose derivation of the edit operator and opcode 10 from the primitive opcodes.

```

1132 Nock 4K
1133
1134 A noun is an atom or a cell. An atom is a natural
1135 number. A cell is an ordered pair of nouns.
1136
1137 Reduce by the first matching pattern; variables match
1138 any noun.
1139
1140 nock(a)          *a
1141 [a b c]          [a [b c]]
1142
1143 ?[a b]           0
1144 ?a               1
1145 +[a b]           +[a b]
1146 +a               1 + a
1147 =[a a]           0
1148 =[a b]           1
1149
1150 /[1 a]           a
1151 /[2 a b]          a
1152 /[3 a b]          b
1153 /[(a + a) b]      /[2 /[a b]]
1154 /[(a + a + 1) b]  /[3 /[a b]]
1155 /a               /a
1156
1157 #[1 a b]          a
1158 #[(a + a) b c]    #[a [b /[(a + a + 1) c]] c]
1159 #[(a + a + 1) b c] #[a [/[(a + a) c] b] c]
1160 #a                #a
1161
1162 *[a [b c] d]      [*[a b c] *[a d]]
1163
1164 *[a 0 b]           /[b a]
1165 *[a 1 b]           b
1166 *[a 2 b c]          [*[a b] *[a c]]
1167 *[a 3 b]            ?*[a b]
1168 *[a 4 b]            +*[a b]
1169 *[a 5 b c]          =[*[a b] *[a c]]
1170
1171 *[a 6 b c d]        *[a [*[c d] 0 *[[2 3] 0 *[a 4 4 b]]]]
1172 *[a 7 b c]          [*[a b] c]
1173 *[a 8 b c]          *[[*[a b] a] c]

```

```

1174  * [a 9 b c]          * [* [a c] 2 [0 1] 0 b]
1175  * [a 10 [b c] d]     # [b * [a c] * [a d]]
1176
1177  * [a 11 [b c] d]      * [[* [a c] * [a d]] 0 3]
1178  * [a 11 b c]          * [a c]
1179
1180  * a                    * a
1181

```

---

1182       Source: ~sorreg-namtyv (2018-09-27)

## 1183   4   The Future of Nock

1184   While deviations from the trunk line of the Nock family have  
1185   been proposed at various points,<sup>12</sup> Nock itself has remained the  
1186   definitional substrate of Urbit since its inception. It has also  
1187   been adopted as the primary ISA of Nockchain and the Nock-  
1188   App ecosystem.

1189       Why, then, do we contemplate further changes? The *skew*  
1190   proposal by ~siprel and ~little-ponnys argued that Nock 4K  
1191   represented an undesirable saddle point in the design space of  
1192   possible Nocks, itself a “ball of mud” (~siprel and ~little-  
1193   ponnys, 2020). While *skew* itself was not adopted, it inspired  
1194   the development of *Plunder* and *PLAN* as a solid-state comput-  
1195   ing architecture sharing some ambitions with Urbit and Nock  
1196   (~siprel and ~little-ponnys, 2023). A rigorously æsthetic  
1197   argument can thus be sustained that Nock is not yet “close  
1198   enough” to its final, diamond-perfect form to be a viable can-  
1199   didate.

1200       While some have found this argument compelling, Urbit’s  
1201   core developers have elected to maintain work in the “main  
1202   line” of traditional Nock as the system’s target ISA. The Nock  
1203   4K specification is a good candidate, in this sense, for a “final”  
1204   version of Nock, as it has been successfully used in produc-  
1205   tion for several years. It seems more likely that subsequent  
1206   changes to Nock will derive not from alternative representa-  
1207   tions but from either dramatically more elegant expressions

---

<sup>12</sup>Notably, *Ax* (see pp. XX–XX, this volume), *skew*, and *PLAN* (see pp. XX–XX, this volume).



1208 (e.g., of opcode 6 or a combinator refactor) or from an implicit  
1209 underspecification in the current Nock 4K which should be  
1210 made explicit.

## 1211 5 Conclusion

1212 Nock began life as a hyper-Turing machine language, a theo-  
1213 retical construct for the purpose of defining higher-level pro-  
1214 gramming languages with appropriate affordances and seman-  
1215 tics. While its opcodes and syntax have gradually evolved over  
1216 the course of two decades, the ambition to uproot the Unix  
1217 “ball of mud” and replace it with a simple operating function  
1218 amenable to reason has remained the north star of Urbit and  
1219 Nock. The history of Nock serves as an index of refinement  
1220 as Yarvin and contributors sought to balance conciseness, effi-  
1221 ciency, and practicality.

1222 The most recent version, Nock 4K, appears to provide all  
1223 of the opcodes necessary for correct and efficient<sup>13</sup> evaluation.  
1224 It is likely that future versions of Nock will be based genet-  
1225 ically on Nock 4K, but with some changes to improve its per-  
1226 formance and usability. The road to zero kelvin is likely very  
1227 long still, given an abundance of caution, but it also appears to  
1228 be straight.

## 1229 References

- 1230 Burger, Alexander (2006) “Pico Lisp: A Radical Approach to  
1231 Application Development”. URL:  
1232 <https://software-lab.de/radical.pdf> (visited on  
1233 ~2025.5.19).
- 1234 Madore, David (2003) “The Unlambda Programming  
1235 Language”. URL:  
1236 <http://www.madore.org/~david/programs/unlambda/>  
1237 (visited on ~2025.5.19).

---

<sup>13</sup>Modulo the vagaries of the von Neumann architecture, etc.

- 1238 Raphael (2012) “Are there minimum criteria for a  
 1239 programming language being Turing complete?” URL:  
 1240 [https://cs.stackexchange.com/questions/991/are-](https://cs.stackexchange.com/questions/991/are-there-minimum-criteria-for-a-programming-language-being-turing-complete)  
 1241 [there-minimum-criteria-for-a-programming-](https://cs.stackexchange.com/questions/991/are-there-minimum-criteria-for-a-programming-language-being-turing-complete)  
 1242 [language-being-turing-complete](https://cs.stackexchange.com/questions/991/are-there-minimum-criteria-for-a-programming-language-being-turing-complete) (visited on  
 1243 ~2025.5.19).
- 1244 ~siprel, Benjamin and Elliot Glaysher ~littel-ponnys  
 1245 (2020) “SKEW”. URL: [https:](https://github.com/urbit/urbit/blob/skew/pkg/hs/urbit-skew/skew.md)  
 1246 [//github.com/urbit/urbit/blob/skew/pkg/hs/urbit-](https://github.com/urbit/urbit/blob/skew/pkg/hs/urbit-skew/skew.md)  
 1247 [skew/skew.md](https://github.com/urbit/urbit/blob/skew/pkg/hs/urbit-skew/skew.md) (visited on ~2025.5.19).
- 1248 — (2023) “What is Plunder?” URL: <https://plunder.tech/>  
 1249 (visited on ~2025.7.6).
- 1250 ~sorreg-namtyv, Curtis Yarvin (2006a) “U, a small model”.  
 1251 URL: <http://lambda-the-ultimate.org/node/1269>  
 1252 (visited on ~2024.2.20).
- 1253 — (2006b) “U, a small model”. URL:  
 1254 <http://urbit.sourceforge.net/u.txt> (visited on  
 1255 ~2024.2.20).
- 1256 — (2008a) “Nock 10K”. URL:  
 1257 [https://github.com/urbit/archaeology/blob/](https://github.com/urbit/archaeology/blob/0b228203e665579848d30c763dda55bb107b0a34/Spec/nock/10.txt)  
 1258 [0b228203e665579848d30c763dda55bb107b0a34/Spec/](https://github.com/urbit/archaeology/blob/0b228203e665579848d30c763dda55bb107b0a34/Spec/nock/10.txt)  
 1259 [nock/10.txt](https://github.com/urbit/archaeology/blob/0b228203e665579848d30c763dda55bb107b0a34/Spec/nock/10.txt) (visited on ~2024.2.20).
- 1260 — (2008b) “Nock 11K”. URL:  
 1261 [https://github.com/urbit/archaeology/blob/](https://github.com/urbit/archaeology/blob/0b228203e665579848d30c763dda55bb107b0a34/Spec/nock/11.txt)  
 1262 [0b228203e665579848d30c763dda55bb107b0a34/Spec/](https://github.com/urbit/archaeology/blob/0b228203e665579848d30c763dda55bb107b0a34/Spec/nock/11.txt)  
 1263 [nock/11.txt](https://github.com/urbit/archaeology/blob/0b228203e665579848d30c763dda55bb107b0a34/Spec/nock/11.txt) (visited on ~2024.2.20).
- 1264 — (2008c) “Nock 12K”. URL:  
 1265 [https://github.com/urbit/archaeology/blob/](https://github.com/urbit/archaeology/blob/0b228203e665579848d30c763dda55bb107b0a34/Spec/nock/12.txt)  
 1266 [0b228203e665579848d30c763dda55bb107b0a34/Spec/](https://github.com/urbit/archaeology/blob/0b228203e665579848d30c763dda55bb107b0a34/Spec/nock/12.txt)  
 1267 [nock/12.txt](https://github.com/urbit/archaeology/blob/0b228203e665579848d30c763dda55bb107b0a34/Spec/nock/12.txt) (visited on ~2024.2.20).
- 1268 — (2008d) “Nock 13K”. URL:  
 1269 [https://github.com/urbit/archaeology/blob/](https://github.com/urbit/archaeology/blob/0b228203e665579848d30c763dda55bb107b0a34/Spec/nock/13.txt)  
 1270 [0b228203e665579848d30c763dda55bb107b0a34/Spec/](https://github.com/urbit/archaeology/blob/0b228203e665579848d30c763dda55bb107b0a34/Spec/nock/13.txt)  
 1271 [nock/13.txt](https://github.com/urbit/archaeology/blob/0b228203e665579848d30c763dda55bb107b0a34/Spec/nock/13.txt) (visited on ~2024.2.20).
- 1272 — (2010a) “Nock 7K”. URL:  
 1273 [https://github.com/urbit/archaeology/blob/](https://github.com/urbit/archaeology/blob/0b228203e665579848d30c763dda55bb107b0a34/Spec/nock/7.txt)  
 1274 [0b228203e665579848d30c763dda55bb107b0a34/Spec/](https://github.com/urbit/archaeology/blob/0b228203e665579848d30c763dda55bb107b0a34/Spec/nock/7.txt)  
 1275 [nock/7.txt](https://github.com/urbit/archaeology/blob/0b228203e665579848d30c763dda55bb107b0a34/Spec/nock/7.txt) (visited on ~2024.2.20).

- 1276 — (2010b) “Nock 8K”. URL:
- 1277 [https://github.com/urbit/archaeology/blob/](https://github.com/urbit/archaeology/blob/0b228203e665579848d30c763dda55bb107b0a34/Spec/nock/8.txt)
- 1278 [0b228203e665579848d30c763dda55bb107b0a34/Spec/](https://github.com/urbit/archaeology/blob/0b228203e665579848d30c763dda55bb107b0a34/Spec/nock/8.txt)
- 1279 [nock/8.txt](https://github.com/urbit/archaeology/blob/0b228203e665579848d30c763dda55bb107b0a34/Spec/nock/8.txt) (visited on ~2024.2.20).
- 1280 — (2010c) “Nock 9K”. URL:
- 1281 [https://github.com/urbit/archaeology/blob/](https://github.com/urbit/archaeology/blob/0b228203e665579848d30c763dda55bb107b0a34/Spec/nock/9.txt)
- 1282 [0b228203e665579848d30c763dda55bb107b0a34/Spec/](https://github.com/urbit/archaeology/blob/0b228203e665579848d30c763dda55bb107b0a34/Spec/nock/9.txt)
- 1283 [nock/9.txt](https://github.com/urbit/archaeology/blob/0b228203e665579848d30c763dda55bb107b0a34/Spec/nock/9.txt) (visited on ~2024.2.20).
- 1284 — (2010d) “Nock: Maxwell’s equations of software”. URL:
- 1285 [http://moronlab.blogspot.com/2010/01/nock-](http://moronlab.blogspot.com/2010/01/nock-maxwells-equations-of-software.html)
- 1286 [maxwells-equations-of-software.html](http://moronlab.blogspot.com/2010/01/nock-maxwells-equations-of-software.html) (visited on
- 1287 ~2024.1.25).
- 1288 — (2010e) “Urbit: functional programming from scratch”.
- 1289 URL: [http://moronlab.blogspot.com/2010/01/urbit-](http://moronlab.blogspot.com/2010/01/urbit-functional-programming-from.html)
- 1290 [functional-programming-from.html](http://moronlab.blogspot.com/2010/01/urbit-functional-programming-from.html) (visited on
- 1291 ~2024.1.25).
- 1292 — (2010f) “Why Nock?” URL:
- 1293 [https://github.com/cgyarvin/urbit/blob/gh-](https://github.com/cgyarvin/urbit/blob/gh-pages/Spec/urbit/5-whynock.txt)
- 1294 [pages/Spec/urbit/5-whynock.txt](https://github.com/cgyarvin/urbit/blob/gh-pages/Spec/urbit/5-whynock.txt) (visited on
- 1295 ~2025.5.19).
- 1296 — (2011) “Nock 6K”. URL:
- 1297 [https://github.com/urbit/archaeology/blob/](https://github.com/urbit/archaeology/blob/0b228203e665579848d30c763dda55bb107b0a34/Spec/nock/6.txt)
- 1298 [0b228203e665579848d30c763dda55bb107b0a34/Spec/](https://github.com/urbit/archaeology/blob/0b228203e665579848d30c763dda55bb107b0a34/Spec/nock/6.txt)
- 1299 [nock/6.txt](https://github.com/urbit/archaeology/blob/0b228203e665579848d30c763dda55bb107b0a34/Spec/nock/6.txt) (visited on ~2024.2.20).
- 1300 — (2012) “Nock 5K”. URL:
- 1301 [https://github.com/urbit/archaeology/blob/](https://github.com/urbit/archaeology/blob/0b228203e665579848d30c763dda55bb107b0a34/Spec/nock/5.txt)
- 1302 [0b228203e665579848d30c763dda55bb107b0a34/Spec/](https://github.com/urbit/archaeology/blob/0b228203e665579848d30c763dda55bb107b0a34/Spec/nock/5.txt)
- 1303 [nock/5.txt](https://github.com/urbit/archaeology/blob/0b228203e665579848d30c763dda55bb107b0a34/Spec/nock/5.txt) (visited on ~2024.2.20).
- 1304 — (2025). “Urbit: Making the Future Real.” In: *LambdaConf*
- 1305 *2025*. Accessed: 2025-05-19. Estes Park, Colorado:
- 1306 LambdaConf. URL: <https://www.lambdaconf.us/> (visited
- 1307 on ~2025.5.19).
- 1308 — (2018-09-27) “Nock 4K”. URL: [https://docs.urbit.org/](https://docs.urbit.org/language/nock/reference/definition)
- 1309 [language/nock/reference/definition](https://docs.urbit.org/language/nock/reference/definition) (visited on
- 1310 ~2024.2.20).
- 1311 Wolfram, Stephen (2021). *Combinators: A Centennial View*.
- 1312 Accessed: 2025-05-19. Champaign, IL: Wolfram Media.
- 1313 URL: <https://www.wolfram.com/combinators/> (visited
- 1314 on ~2025.5.19).