

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>154</b>
<b>2</b>	<b>Nock as a Combinator Calculus</b>	<b>154</b>
<b>3</b>	<b>Nock’s Decrement</b>	<b>158</b>
3.1	U . . . . .	159
3.2	Nock 13K . . . . .	165
3.3	Nock 12K . . . . .	168
3.4	Nock 11K . . . . .	170
3.5	Nock 10K . . . . .	172

25	3.6	Nock 9K . . . . .	174
26	3.7	Nock 8K . . . . .	176
27	3.8	Nock 7K . . . . .	178
28	3.9	Nock 6K . . . . .	179
29	3.10	Nock 5K . . . . .	181
30	3.11	Nock 4K . . . . .	182
31	<b>4</b>	<b>The Future of Nock</b>	<b>184</b>
32	<b>5</b>	<b>Conclusion</b>	<b>185</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 (Reitzig, 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:  $s$ ,  $\kappa$ , 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.  $\kappa xy = x$ , 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 exposi-  
tion 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 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. (~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.

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>6</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
(S)	(S \$a \$b \$c)	(\$a \$c (\$b \$c))

<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>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.”



263       A number is any natural number (ie, nonnegative  
264       integer).  
265  
266       A pair is an ordered pair of any two terms.  
267  
268       There is only one foo.  
269  
270   5 Syntax  
271       U is a computational model, not a programming  
272       language.  
273  
274       But a trivial ASCII syntax for terms is useful.  
275  
276   5.1 Trivial syntax: briefly  
277       Numbers are in decimal. Pairs are in parentheses  
278       that nest to the right. Foo is "~".  
279  
280       Whitespace is space or newline. Line comments  
281       use "#".  
282  
283   5.2 Trivial syntax: exactly  
284       term       : number  
285                | 40 ?white pair ?white 41  
286                | foo  
287  
288       number    : 48  
289                | [49-57] \*[48-57]  
290  
291       pair       : term white term  
292                | term white pair  
293  
294       foo        : 126  
295  
296       white     : \*(32 | 10 | (35 \*[32-126] 10))  
297  
298   6 Semantics  
299       U is a pure function from term to term.  
300  
301       This document completely defines U. There is no  
302       compatible way to extend or revise U.  
303  
304   6.1 Rules

	[name]	[pattern]	[definition]
305			
306			
307	(a)	(\$a 0 \$b)	\$b
308	(b)	(\$a 1 \$b \$c)	1
309	(c)	(\$a 1 \$b)	0
310	(d)	(\$a 2 0 \$b \$c)	\$b
311	(e)	(\$a 2 %n \$b \$c)	\$c
312	(f)	(\$a 3 \$b \$c)	=( \$b \$c)
313	(g)	(\$a 4 %n)	+%n
314			
315	(h)	(\$a 5 (~ ~ \$b) \$c)	\$b
316	(i)	(\$a 5 (~ \$b \$c) \$d)	*( \$a \$b \$c \$d)
317	(j)	(\$a 5 (~ ~) \$b)	~
318	(k)	(\$a 5 (~ \$b) \$c)	*( \$a \$b \$c)
319	(l)	(\$a 5 (\$b \$c) \$d)	*( \$a \$b \$d) *( \$a \$c \$d)
320			
321	(m)	(\$a 5 \$b \$c)	\$b
322			
323	(n)	(\$a 6 \$b \$c)	*( \$a *( \$a 5 \$b \$c))
324	(o)	(\$a 7 \$b)	*( \$a 5 \$a \$a \$b)
325	(p)	(\$a 8 \$b \$c \$d)	> ( \$b \$c \$d)
326			
327	(q)	(\$a \$b \$c)	*( \$a 5 *( \$a 7 \$b) \$c)
328	(r)	(\$a \$b)	*( \$a \$b)
329	(s)	\$a	*\$a

330  
331 The rule notation is a pseudocode, only used in  
332 this file. Its definition follows.

## 333 6.2 Rule pseudocode: briefly

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

## 336 6.3 Rule pseudocode: exactly

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

340  
341 An evaluation is a tree in which each node is a  
342 term, a term-valued variable, or a unary  
343 operation.

344

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

350

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

354

#### 355 6.4 Evaluation semantics

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

357

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

363

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

366

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

370

#### 371 6.5 Simple operators: equal, increment, evaluate

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

374

375 +%n is %n plus 1.

376

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

378

#### 379 6.6 The follow operator

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

382

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

384

385 if \*(\$a 5 \$b \$term) chokes:

386 \*(\$a 5 \$c \$term) chokes.

387

388 if \*(\$a 5 \$b \$term) completes:

```

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

```

431  
432 For example, packet loss in an unreliable packet  
433 protocol, such as UDP, is a voluntary failure  
434 mechanism.

435  
436 If the packet transfer function of a stateful UDP  
437 server is defined in terms of U, failure to  
438 compute means dropping a packet. If the server  
439 has no other I/O, its semantics are completely  
440 defined by its initial state and packet function.

## 441 7.2 Other unstandardized implementation details

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

446  
447 (Although trust failure, allocation failure or  
448 timeout, and choke detection all depend on what  
449 is presumably a single voluntary failure  
450 mechanism, they are orthogonal and should not be  
451 confused.)

452  
453 Also, because U is so abstract, differences in  
454 implementation strategy can result in performance  
455 disparities which are almost arbitrarily extreme.  
456 The difficulty of standardizing performance is  
457 well-known.

458  
459 No magic bullet can stop these unstandardized  
460 issues from becoming practical causes of lock-in  
461 and incompatibility. Systems which depend on U  
462 must manage them at every layer.

---

463  
464  
465 Source: ~sorreg-namtyv (2006)

## 466 3.2 Nock 13K

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

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

471 Axiomatic operator \* tar<sup>7</sup> is identified as a GOTO.<sup>8</sup>

---

472 Listing 2: Nock 13K, 8 March 2008.

---

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

474 Date: 3/8/2008

475 Version: 0.13

476

477 1. Manifest

478

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

481

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

485

486 2. Data

487

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

489

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

494

495 3. Pseudocode

496

497 nock is defined in a pattern-matching pseudocode.

498

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

502

503 4. Definition

504

505 4.1 Transformations

---

<sup>7</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>8</sup>One can see the influence of this version's naming scheme on Atman's Ax, pp. XX-XX herein.

```

506
507      *(a 0 b c) => *(*(a b) c)
508      *(a 0 b)   => /(b a)
509      *(a 1 b)   => (b)
510      *(a 2 b)   => **(a b)
511      *(a 3 b)   => &*(a b)
512      *(a 4 b)   => ^*(a b)
513      *(a 5 b)   => =*(a b)
514      *(a 6 b c d) => *(a 2 (0 1)
515                        2 (1 c d) (1 0)
516                        2 (1 2 3) (1 0) 4 4 b)
517      *(a b c)   => (*(a b) *(a c))
518      *(a)       => *(a)
519
520 4.2 Operators
521
522 4.2.1 Goto (*)
523
524      *(a)       -> nock(a)
525
526 4.2.2 Deep (&)
527
528      &(a b)      -> 0
529      &(a)        -> 1
530
531 4.2.3 Bump (^)
532
533      ^(a b)      -> ^(a b)
534      ^(a)        -> a + 1
535
536 4.2.4 Same (=)
537
538      =(a a)      -> 0
539      =(a b)      -> 1
540      =(a)        -> =(a)
541
542 4.2.5 Snip (/)
543
544      /(1 a)      -> a
545      /(2 a b)    -> a
546      /(3 a b)    -> b
547      /((a + a) b) -> /(2 /(a b))

```

```

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

```

---

551 Source: ~sorreg-namtyv (2008)

### 552 3.3 Nock 12K

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

---

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

---

```

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

```

#### 559 1. Introduction

560 This file defines one function, "nock."

561 nock is in the public domain.

#### 562 2. Data

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

#### 567 3. Semantics

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

#### 570 4. Pseudocode

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

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

#### 575 5. Definition



587

## 588 5.1 Transformations

589

590       \*(a (b c) d) =&gt; (\*(a b c) \*(a d))

591       \*(a 0 b)       =&gt; /(b a)

592       \*(a 1 b)       =&gt; (b)

593       \*(a 2 b c)     =&gt; \*\*(\*(a b) c)

594       \*(a 3 b)       =&gt; \*\*(a b)

595       \*(a 4 b)       =&gt; &amp;\*(a b)

596       \*(a 5 b)       =&gt; ^\*(a b)

597       \*(a 6 b)       =&gt; =\*(a b)

598

599       \*(a 7 b c d) => \*(a 3 (0 1) 3 (1 c d) (1 0)  
600                               3 (1 2 3) (1 0) 5 5 b)

601       \*(a 8 b c)     =&gt; \*(a 2 (((1 0) b) c) 0 3)

602       \*(a 9 b c)     =&gt; \*(a c)

603

604       \*(a)           =&gt; \*(a)

605

## 606 5.2 Operators

607

## 608 5.2.1 Goto (\*)

609

610       \*(a)                   -&gt; nock(a)

611

## 612 5.2.2 Deep (&amp;)

613

614       &amp;(a b)               -&gt; 0

615       &amp;(a)                 -&gt; 1

616

## 617 5.2.4 Bump (^)

618

619       ^(a b)               -&gt; ^(a b)

620       ^(a)                 -&gt; a + 1

621

## 622 5.2.5 Same (=)

623

624       =(a a)               -&gt; 0

625       =(a b)               -&gt; 1

626       =(a)                 -&gt; =(a)

627

## 628 5.2.6 Snip (/)

```

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

```

---

637 Source: ~sorreg-namtyv (2008)

## 638 3.4 Nock 11K

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

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

### Listing 4: Nock 11K, 25 May 2008.

---

```

643
644 Author: Mencius Moldebug (moldebug@gmail.com)
645 Date: 5/25/2008
646 Version: 11K
647

```

#### 648 1. Introduction

650 This file defines one function, "nock."

651 nock is in the public domain.

#### 654 2. Data

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

#### 661 3. Semantics

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

#### 666 4. Pseudocode

667

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

670

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

672

## 673 5. Definition

674

### 675 5.1 Transformations

676

677       \*(a (b c) d) => (\*(a b c) \*(a d))

678       \*(a 0 b)       => /(b a)

679       \*(a 1 b)       => (b)

680       \*(a 2 b c d) => \*(a 3 (0 1) 3 (1 c d) (1 0)  
681                       3 (1 2 3) (1 0) 5 5 b)

682       \*(a 3 b)       => \*\*\*(a b)

683       \*(a 4 b)       => &\*(a b)

684       \*(a 5 b)       => ^\*(a b)

685       \*(a 6 b)       => =\*(a b)

686

687       \*(a 7 b c)     => \*(a 3 (((1 0) b) c) 1 0 3)

688       \*(a 8 b c)     => \*(a c)

689

690       \*(a)           => \*(a)

691

### 692 5.2 Operators

693

#### 694 5.2.1 Goto (\*)

695

696       \*(a)           -> nock(a)

697

#### 698 5.2.2 Deep (&)

699

700       &(a b)         -> 0

701       &(a)          -> 1

702

#### 703 5.2.4 Bump (^)

704

705       ^(a b)         -> ^(a b)

706       ^(a)          -> a + 1

707

#### 708 5.2.5 Same (=)

709

```

710      = (a a)          -> 0
711      = (a b)          -> 1
712      = (a)            -> = (a)
713
714 5.2.6 Snip (/)
715
716      / (1 a)           -> a
717      / (2 a b)         -> a
718      / (3 a b)         -> b
719      / ((a + a) b)      -> / (2 / (a b))
720      / ((a + a + 1) b) -> / (3 / (a b))
721      / (a)             -> / (a)
722

```

---

723 Source: ~sorreg-namtyv (2008)

## 724 3.5 Nock 10K

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

### Listing 5: Nock 10K, 15 September 2008.

---

```

730 Author: Mencius Moldebug [moldebug@gmail.com]
731 Date: 9/15/2008
732 Version: 10K
733
734
735 1. Introduction
736
737     This file defines one function, "nock."
738
739     nock is in the public domain.
740
741 2. Data
742
743     A "noun" is either an "atom" or a "cell." An
744     "atom" is an unsigned integer of any size. A
745     "cell" is an ordered pair of any two nouns, the
746     "head" and "tail."
747

```

## 748 3. Semantics

749

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

752

## 753 4. Pseudocode

754

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

757

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

759

## 760 5. Definition

761

## 762 5.1 Transformations

763

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

765       \*[a 0 b]       => /[b a]

766       \*[a 1 b]       => [b]

767       \*[a 2 b c d] => \*[a 3 [0 1] 3 [1 c d]  
768                               [1 0] 3 [1 2 3] [1 0] 5 5 b]

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

770       \*[a 4 b]       => &\*[a b]

771       \*[a 5 b]       => ^\*[a b]

772       \*[a 6 b]       => =\*[a b]

773       \*[a]           => \*[a]

774

## 775 5.2 Operators

776

## 777 5.2.1 Goto [\*]

778

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

780

## 781 5.2.2 Deep [&amp;]

782

783       &[a b]           -> 0

784       &[a]             -> 1

785

## 786 5.2.4 Bump [^]

787

788       ^[a b]           -> ^[a b]

789       ^[a]             -> (a + 1)

790

791 5.2.5 Like [=]

792

```
793         =[ a a]           -> 0
794         =[ a b]           -> 1
795         =[ a]             -> =[ a]
```

796

797 5.2.6 Snip [/]

798

```
799         /[1 a]           -> a
800         /[2 a b]         -> a
801         /[3 a b]         -> b
802         /[(a + a) b]     -> /[2 /[a b]]
803         /[(a + a + 1) b] -> /[3 /[a b]]
804         /[a]             -> /[a]
```

806 Source: ~sorreg-namtyv (2008)

## 807 3.6 Nock 9K

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

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

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

- 824 • p is the function source;
- 825 • 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
```

```

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

```

---

887 Source: ~sorreg-namtyv (2010)

## 888 3.7 Nock 8K

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

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

897 Nock 8K received an uncharacteristic amount of commen-  
 898 tary, given a preprint document prepared for presentation at  
 899 the 42nd ISCIE International Symposium on Stochastic Systems  
 900 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.

---

```
908
909 1 Structures
910
911     A noun is an atom or a cell. An atom is any
912     unsigned integer. A cell is an ordered pair of
913     nouns.
914
915 2 Pseudocode
916
917     [a b c] is [a [b c]]; *a is nock(a). Reductions
918     match top-down.
919
920 3 Reductions
921
922     ?[a b]           0
923     ?a               1
924     ^a               (a + 1)
925     =[a a]           0
926     =[a b]           1
927
928     /[1 a]           a
929     /[2 a b]         a
930     /[3 a b]         b
931     /[(a + a) b]     /[2 /[a b]]
932     /[(a + a + 1) b] /[3 /[a b]]
933
934     *[a [b c] d]     [*[a b c] *[a d]]
935     *[a 0 b]         /[b a]
936     *[a 1 b]         b
937     *[a 2 b c]       [*[a b] *[a c]]
938     *[a 3 b]         ?*[a b]
939     *[a 4 b]         ^*[a b]
940     *[a 5 b]         =*[a b]
```

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

---

957       Source: ~sorreg-namtyv (2010)

### 958   3.8   Nock 7K

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

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

---

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

---

#### 966   1 Structures

967  
968   A noun is an atom or a cell. An atom is any  
969   natural number. A cell is any ordered pair of  
970   nouns.

#### 972   2 Pseudocode

973		
974	[a b c]	[a [b c]]
975	nock(a)	*a
976		
977	?[a b]	0
978	?a	1

```

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

```

---

```

1011     Source: ~sorreg-namtyv (2010)

```

### 1012 3.9 Nock 6K

```

1013     The axiomatic operator for increment was changed from ^ ket
1014     to + lus. Compound opcode syntax was reworked slightly.

```

---

Listing 9: Nock 6K, 6 July 2011.

---

```

1015     1 Structures
1016
1017

```

```

1018   A noun is an atom or a cell.  An atom is any
1019   natural number.  A cell is an ordered pair of
1020   nouns.
1021
1022   2 Reductions
1023
1024   nock(a)           *a
1025   [a b c]           [a [b c]]
1026
1027   ?[a b]            0
1028   ?a                1
1029   +a                1 + a
1030   =[a a]            0
1031   =[a b]            1
1032
1033   /[1 a]            a
1034   /[2 a b]          a
1035   /[3 a b]          b
1036   /[(a + a) b]      /[2 /[a b]]
1037   /[(a + a + 1) b]  /[3 /[a b]]
1038
1039   *[a [b c] d]      [*[a b c] *[a d]]
1040
1041   *[a 0 b]          /[b a]
1042   *[a 1 b]          b
1043   *[a 2 b c]        [*[a b] *[a c]]
1044   *[a 3 b]          ?*[a b]
1045   *[a 4 b]          +*[a b]
1046   *[a 5 b]          =*[a b]
1047
1048   *[a 6 b c d]      *[a 2 [0 1] 2 [1 c d] [1 0]
1049                      2 [1 2 3] [1 0] 4 4 b]
1050   *[a 7 b c]        *[a 2 b 1 c]
1051   *[a 8 b c]        *[a 7 [[0 1] b] c]
1052   *[a 9 b c]        *[a 7 c 0 b]
1053   *[a 10 b c]       *[a c]
1054   *[a 10 [b c] d]   *[a 8 c 7 [0 2] d]
1055
1056   +[a b]            +[a b]
1057   =a                =a
1058   /a                /a
1059   *a                *a
1060

```

---

1061 Source: ~sorreg-namtyv (2011)

### 1062 3.10 Nock 5K

1063 Compound opcode syntax was reworked slightly. All trivial  
1064 reductions of axiomatic operators were removed to the preface  
1065 of the specification.

1066 (For instance, a trivial “cosmetic” change was made to 5K’s  
1067 specification after it was publicly posted in order to synchro-  
1068 nize it with the VM’s behavior (dd779c1).)

#### Listing 10: Nock 5K, 24 September 2012.

```

1069 1 Structures
1070
1071     A noun is an atom or a cell.  An atom is any natural
1072     number.  A cell is an ordered pair of nouns.
1073
1074 2 Reductions
1075
1076     nock(a)           *a
1077     [a b c]           [a [b c]]
1078
1079     ?[a b]            0
1080     ?a                1
1081     +[a b]            +[a b]
1082     +a                1 + a
1083     =[a a]            0
1084     =[a b]            1
1085     =a                =a
1086
1087     /[1 a]            a
1088     /[2 a b]          a
1089     /[3 a b]          b
1090     /[(a + a) b]      /[2 /[a b]]
1091     /[(a + a + 1) b]  /[3 /[a b]]
1092     /a                /a
1093
1094     *[a [b c] d]      [*[a b c] *[a d]]
1095
1096     *[a 0 b]          /[b a]
1097     *[a 1 b]          b
1098

```

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

---

1114 Source: ~sorreg-namtyv (2012)

### 1115 3.11 Nock 4K

1116 The primary change motivating 5K to 4K was the introduction  
 1117 of an edit operator # hax, which ameliorated the proliferation  
 1118 of cells in the Nock runtime's memory.<sup>9</sup> The edit operator is an  
 1119 optimization which makes modifications to a Nock data struc-  
 1120 ture more efficient. It's a notable example of a change moti-  
 1121 vated by the pragmatics of the runtime rather than theoretical  
 1122 or higher-level language concerns.<sup>10</sup>

1123 Opcode 5 (equality) was rewritten to more explicit with ap-  
 1124 plication of the cell distribution rule. Opcodes 6–9 were rewrit-  
 1125 ten to utilize the \* tar operator rather than routing via opcode  
 1126 2. Opcode 11 (formerly opcode 10) was likewise massaged. In  
 1127 general, preferring to express rules using \* tar proved to be  
 1128 slightly more terse than utilizing opcode 2.

---

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

---

1129 Nock 4K  
 1130

1131  
 1132 A noun is an atom or a cell. An atom is a natural

---

<sup>9</sup>The date must be earlier than 27 September 2018; cf. `urbit/urbit` #1027.

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

```

1133 number. A cell is an ordered pair of nouns.
1134
1135 Reduce by the first matching pattern; variables match
1136 any noun.
1137
1138 nock(a)          *a
1139 [a b c]          [a [b c]]
1140
1141 ?[a b]           0
1142 ?a               1
1143 +[a b]           +[a b]
1144 +a               1 + a
1145 =[a a]           0
1146 =[a b]           1
1147
1148 /[1 a]           a
1149 /[2 a b]         a
1150 /[3 a b]         b
1151 /[(a + a) b]     /[2 /[a b]]
1152 /[(a + a + 1) b] /[3 /[a b]]
1153 /a               /a
1154
1155 #[1 a b]         a
1156 #[(a + a) b c]   #[a [b /[(a + a + 1) c]] c]
1157 #[(a + a + 1) b c] #[a [/[(a + a) c] b] c]
1158 #a               #a
1159
1160 *[a [b c] d]     [*[a b c] *[a d]]
1161
1162 *[a 0 b]         /[b a]
1163 *[a 1 b]         b
1164 *[a 2 b c]       *[*[a b] *[a c]]
1165 *[a 3 b]         ?*[a b]
1166 *[a 4 b]         +*[a b]
1167 *[a 5 b c]       =[*[a b] *[a c]]
1168
1169 *[a 6 b c d]     *[a *[[c d] 0 *[[2 3] 0 *[a 4 4 b]]]]
1170 *[a 7 b c]       *[*[a b] c]
1171 *[a 8 b c]       *[[*[a b] a] c]
1172 *[a 9 b c]       *[*[a c] 2 [0 1] 0 b]
1173 *[a 10 [b c] d]  #[b *[a c] *[a d]]
1174

```

```

1175 * [a 11 [b c] d]      * [[* [a c] * [a d]] 0 3]
1176 * [a 11 b c]          * [a c]
1177
1178 * a                      * a
1179

```

---

1180       Source: `~sorreg-namtyv` (2018-09-27)

## 1181 4 The Future of Nock

1182 While deviations from the trunk line of the Nock family have  
1183 been proposed at various points,<sup>11</sup> Nock itself has remained  
1184 the definitional substrate of Urbit since its inception. It has  
1185 also been adopted as the primary ISA of Nockchain and the  
1186 NockApp ecosystem.

1187       Why, then, do we contemplate further changes? The `skew`  
1188 proposal by `~siprel` and `~little-ponnys` argued that Nock 4K  
1189 represented an undesirable saddle point in the design space of  
1190 possible Nocks, itself a “ball of mud” (`~siprel` and `~little-`  
1191 `ponnys`, 2020). While `skew` itself was not adopted, it inspired  
1192 the development of `Plunder` and `PLAN` as a solid-state comput-  
1193 ing architecture sharing some ambitions with Urbit and Nock  
1194 (`~siprel` and `~little-ponnys`, 2023). A rigorously æsthetic  
1195 argument can thus be sustained that Nock is not yet “close  
1196 enough” to its final, diamond-perfect form to be a viable can-  
1197 didate.

1198       While some have found this argument compelling, Urbit’s  
1199 core developers have elected to maintain work in the “main  
1200 line” of traditional Nock as the system’s target ISA. The Nock  
1201 4K specification is a good candidate, in this sense, for a “final”  
1202 version of Nock, as it has been successfully used in produc-  
1203 tion for several years. It seems more likely that subsequent  
1204 changes to Nock will derive not from alternative representa-  
1205 tions but from either dramatically more elegant expressions  
1206 (e.g., of opcode 6 or a combinator refactor) or from an implicit  
1207 underspecification in the current Nock 4K which should be  
1208 made explicit.

---

<sup>11</sup>Notably, `Ax` (see pp. XX–XX, this volume), `skew`, and `PLAN` (see pp. XX–XX, this volume).



## 5 Conclusion

A13: If you don't completely understand your code and the semantics of all the code it depends on, your code is wrong.

A21: Prefer mechanical simplicity to mathematical simplicity. Often mechanical simplicity and mathematical simplicity go together.

F1: If it's not deterministic, it isn't real.

(~wicdev-wisryt, Urbit Precepts (2020))

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<sup>12</sup> 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).

---

<sup>12</sup>Modulo the vagaries of the von Neumann architecture, etc.

- 1240 Madore, David (2003) “The Unlambda Programming  
1241 Language”. URL:  
1242 <http://www.madore.org/~david/programs/unlambda/>  
1243 (visited on ~2025.5.19).
- 1244 Reitzig, Raphael (2012) “Are there minimum criteria for a  
1245 programming language being Turing complete?” URL:  
1246 [https://cs.stackexchange.com/questions/991/are-](https://cs.stackexchange.com/questions/991/are-there-minimum-criteria-for-a-programming-language-being-turing-complete)  
1247 [there-minimum-criteria-for-a-programming-](https://cs.stackexchange.com/questions/991/are-there-minimum-criteria-for-a-programming-language-being-turing-complete)  
1248 [language-being-turing-complete](https://cs.stackexchange.com/questions/991/are-there-minimum-criteria-for-a-programming-language-being-turing-complete) (visited on  
1249 ~2025.5.19).
- 1250 ~siprel, Benjamin and Elliot Glaysher ~littel-ponnys  
1251 (2020) “SKEW”. URL: [https://github.com/urbit/urbit/blob/skew/pkg/hs/urbit-](https://github.com/urbit/urbit/blob/skew/pkg/hs/urbit-skew/skew.md)  
1252 [skew/skew.md](https://github.com/urbit/urbit/blob/skew/pkg/hs/urbit-skew/skew.md) (visited on ~2025.5.19).
- 1253 — (2023) “What is Plunder?” URL: <https://plunder.tech/>  
1254 (visited on ~2025.7.6).
- 1255 ~soreg-namtyv, Curtis Yarvin (2006a) “U, a small model”.  
1256 URL: <http://lambda-the-ultimate.org/node/1269>  
1257 (visited on ~2024.2.20).
- 1258 — (2006b) “U, a small model”. URL:  
1259 <http://urbit.sourceforge.net/u.txt> (visited on  
1260 ~2024.2.20).
- 1261 — (2008a) “Nock 10K”. URL:  
1262 [https://github.com/urbit/archaeology/blob/](https://github.com/urbit/archaeology/blob/0b228203e665579848d30c763dda55bb107b0a34/Spec/nock/10.txt)  
1263 [0b228203e665579848d30c763dda55bb107b0a34/Spec/](https://github.com/urbit/archaeology/blob/0b228203e665579848d30c763dda55bb107b0a34/Spec/nock/10.txt)  
1264 [nock/10.txt](https://github.com/urbit/archaeology/blob/0b228203e665579848d30c763dda55bb107b0a34/Spec/nock/10.txt) (visited on ~2024.2.20).
- 1265 — (2008b) “Nock 11K”. URL:  
1266 [https://github.com/urbit/archaeology/blob/](https://github.com/urbit/archaeology/blob/0b228203e665579848d30c763dda55bb107b0a34/Spec/nock/11.txt)  
1267 [0b228203e665579848d30c763dda55bb107b0a34/Spec/](https://github.com/urbit/archaeology/blob/0b228203e665579848d30c763dda55bb107b0a34/Spec/nock/11.txt)  
1268 [nock/11.txt](https://github.com/urbit/archaeology/blob/0b228203e665579848d30c763dda55bb107b0a34/Spec/nock/11.txt) (visited on ~2024.2.20).
- 1269 — (2008c) “Nock 12K”. URL:  
1270 [https://github.com/urbit/archaeology/blob/](https://github.com/urbit/archaeology/blob/0b228203e665579848d30c763dda55bb107b0a34/Spec/nock/12.txt)  
1271 [0b228203e665579848d30c763dda55bb107b0a34/Spec/](https://github.com/urbit/archaeology/blob/0b228203e665579848d30c763dda55bb107b0a34/Spec/nock/12.txt)  
1272 [nock/12.txt](https://github.com/urbit/archaeology/blob/0b228203e665579848d30c763dda55bb107b0a34/Spec/nock/12.txt) (visited on ~2024.2.20).
- 1273 — (2008d) “Nock 13K”. URL:  
1274 [https://github.com/urbit/archaeology/blob/](https://github.com/urbit/archaeology/blob/0b228203e665579848d30c763dda55bb107b0a34/Spec/nock/13.txt)  
1275 [0b228203e665579848d30c763dda55bb107b0a34/Spec/](https://github.com/urbit/archaeology/blob/0b228203e665579848d30c763dda55bb107b0a34/Spec/nock/13.txt)  
1276 [nock/13.txt](https://github.com/urbit/archaeology/blob/0b228203e665579848d30c763dda55bb107b0a34/Spec/nock/13.txt) (visited on ~2024.2.20).
- 1277

- 1278 — (2010a) “Nock 7K”. URL:  
1279 [https://github.com/urbit/archaeology/blob/](https://github.com/urbit/archaeology/blob/0b228203e665579848d30c763dda55bb107b0a34/Spec/nock/7.txt)  
1280 [0b228203e665579848d30c763dda55bb107b0a34/Spec/](https://github.com/urbit/archaeology/blob/0b228203e665579848d30c763dda55bb107b0a34/Spec/nock/7.txt)  
1281 [nock/7.txt](https://github.com/urbit/archaeology/blob/0b228203e665579848d30c763dda55bb107b0a34/Spec/nock/7.txt) (visited on ~2024.2.20).
- 1282 — (2010b) “Nock 8K”. URL:  
1283 [https://github.com/urbit/archaeology/blob/](https://github.com/urbit/archaeology/blob/0b228203e665579848d30c763dda55bb107b0a34/Spec/nock/8.txt)  
1284 [0b228203e665579848d30c763dda55bb107b0a34/Spec/](https://github.com/urbit/archaeology/blob/0b228203e665579848d30c763dda55bb107b0a34/Spec/nock/8.txt)  
1285 [nock/8.txt](https://github.com/urbit/archaeology/blob/0b228203e665579848d30c763dda55bb107b0a34/Spec/nock/8.txt) (visited on ~2024.2.20).
- 1286 — (2010c) “Nock 9K”. URL:  
1287 [https://github.com/urbit/archaeology/blob/](https://github.com/urbit/archaeology/blob/0b228203e665579848d30c763dda55bb107b0a34/Spec/nock/9.txt)  
1288 [0b228203e665579848d30c763dda55bb107b0a34/Spec/](https://github.com/urbit/archaeology/blob/0b228203e665579848d30c763dda55bb107b0a34/Spec/nock/9.txt)  
1289 [nock/9.txt](https://github.com/urbit/archaeology/blob/0b228203e665579848d30c763dda55bb107b0a34/Spec/nock/9.txt) (visited on ~2024.2.20).
- 1290 — (2010d) “Nock: Maxwell’s equations of software”. URL:  
1291 [http://moronlab.blogspot.com/2010/01/nock-](http://moronlab.blogspot.com/2010/01/nock-maxwells-equations-of-software.html)  
1292 [maxwells-](http://moronlab.blogspot.com/2010/01/nock-maxwells-equations-of-software.html)  
1293 [equations-of-software.html](http://moronlab.blogspot.com/2010/01/nock-maxwells-equations-of-software.html) (visited on  
1294 ~2024.1.25).
- 1294 — (2010e) “Urbit: functional programming from scratch”.  
1295 URL: [http://moronlab.blogspot.com/2010/01/urbit-](http://moronlab.blogspot.com/2010/01/urbit-functional-programming-from.html)  
1296 [functional-programming-from.html](http://moronlab.blogspot.com/2010/01/urbit-functional-programming-from.html) (visited on  
1297 ~2024.1.25).
- 1298 — (2010f) “Why Nock?” URL:  
1299 [https://github.com/cgyarvin/urbit/blob/gh-](https://github.com/cgyarvin/urbit/blob/gh-pages/Spec/urbit/5-whynock.txt)  
1300 [pages/Spec/urbit/5-whynock.txt](https://github.com/cgyarvin/urbit/blob/gh-pages/Spec/urbit/5-whynock.txt) (visited on  
1301 ~2025.5.19).
- 1302 — (2011) “Nock 6K”. URL:  
1303 [https://github.com/urbit/archaeology/blob/](https://github.com/urbit/archaeology/blob/0b228203e665579848d30c763dda55bb107b0a34/Spec/nock/6.txt)  
1304 [0b228203e665579848d30c763dda55bb107b0a34/Spec/](https://github.com/urbit/archaeology/blob/0b228203e665579848d30c763dda55bb107b0a34/Spec/nock/6.txt)  
1305 [nock/6.txt](https://github.com/urbit/archaeology/blob/0b228203e665579848d30c763dda55bb107b0a34/Spec/nock/6.txt) (visited on ~2024.2.20).
- 1306 — (2012) “Nock 5K”. URL:  
1307 [https://github.com/urbit/archaeology/blob/](https://github.com/urbit/archaeology/blob/0b228203e665579848d30c763dda55bb107b0a34/Spec/nock/5.txt)  
1308 [0b228203e665579848d30c763dda55bb107b0a34/Spec/](https://github.com/urbit/archaeology/blob/0b228203e665579848d30c763dda55bb107b0a34/Spec/nock/5.txt)  
1309 [nock/5.txt](https://github.com/urbit/archaeology/blob/0b228203e665579848d30c763dda55bb107b0a34/Spec/nock/5.txt) (visited on ~2024.2.20).
- 1310 — (2025). “Urbit: Making the Future Real.” In: *LambdaConf*  
1311 *2025*. Accessed: 2025-05-19. Estes Park, Colorado:  
1312 LambdaConf. URL: <https://www.lambdaconf.us/> (visited  
1313 on ~2025.5.19).
- 1314 — (2018-09-27) “Nock 4K”. URL: [https://docs.urbit.org/](https://docs.urbit.org/language/nock/reference/definition)  
1315 [language/nock/reference/definition](https://docs.urbit.org/language/nock/reference/definition) (visited on  
1316 ~2024.2.20).

1317 Wolfram, Stephen (2021). *Combinators: A Centennial View*.  
1318 Accessed: 2025-05-19. Champaign, Illinois: Wolfram  
1319 Media. URL: <https://www.wolfram.com/combinators/>  
1320 (visited on ~2025.5.19).