An Approach to Developing on Urbit

N E Davis

April 14, 2021

Malancandra & Sons

Colophon
This document was typeset with the help of KOMA-Script and LaTeX using the kaobook class.
The source code of this book is available at:
`https://github.com/davis68/urbit-textbook`

Publisher
First printed in July 2022 by Malancandra & Sons

Lights All Askew in the Heavens.
Stars Not Where They Seemed or Were Calculated to Be.
A BOOK FOR 12 WISE MEN.
No More in All the World Could Comprehend It.

– *The New York Times*, November 19, 1919

Contents

List of Figures

List of Tables

A Brief Introduction 1

1.1 What We Talk About When We Talk About Urbit

Urbit is a functional-as-in-language, network-first, compatibility-breaking operation function (or hosted operating system). But what does any of this mean? As we explore Urbit software development throughout this book, keep in mind that every piece of Urbit aims to solve a ambitious battery of critical problems with the existing legacy World Wide Web.

The Urbit project intends to cut the Gordian knot of user autonomy and privacy. To this end, the Urbit developers have articulated an approach prioritizing *a legible future-proof program stack*, *data security*, and *cryptographic ownership*. The ambitious scope of this project—and the evolution of the goals over the decade of the 2010s—has led many to have difficulty grasping what exactly Urbit is all about.

A Series of Unfortunate Events

Centralization . For most contemporary corporations, whether enterprise-scale or startup, the driving factor for growth and revenue became the number of customers (users) they were able to attract to their platform or app. Services like `del.icio.us` (founded 2003) and Flickr (founded 2004) betokened a wave of massive centralization, cemented by Facebook, Google, and Apple in the late aughts. TODO XXX number of users on each in 2010

As users jostled onto burgeoning social media platforms, their patterns of behavior changed, and more and more social interactions of significance took place within "walled gardens," service platforms that interfaced only poorly with the exterior web. Vendor lock-in and the nonportability of user data between platforms meant that consumer choice became a byword. It became (and remains) difficult for any user to find out just what a corporation or even an app knows about them, particularly given the rise of surveilling cookies and data trackers.

The shift to mobile computing starting with the 2007 launch of Apple's iPhone drove a rise in cloud computing and cloud storage. To many users, the data storage and access permissions on their data became largely illegible. Sometimes this led to poor assumptions, such as that the custodial corporation would never allow a leak, or that the data would always be backed up safely. As projects failed (like `del.icio.us`) or unilaterally changed policies (Tumblr), users permanently lost data. Given the effort involved in curating tags, bookmarks, images, contacts, and research data, these outcomes frequently amounted in the loss of years of human effort.

Data leaks . During the 2000s and 2010s, data leaks became so common as to hardly merit notice. As users flocked to corporate platforms for social media, publishing, photography, dating, and every other aspect of digital life, insufficient attention was given by corporations to both the practical security of user data and the potential fallout of leaks. Data breaches grew in number ever year, and affected corporations of every size in every industry.

- 2013: Evernote, 50 million records
- 2014: Ebay, 145 million records
- 2015: Ashley Madison, 32 million records
- 2016: Yahoo!, 1 billion records
- 2017: Experian, 147 million records
- 2019: Facebook, 850 million records
- 2019: CapitalOne, 106 million records

("Records" does not equal "people" or even "accounts," of course, rendering these numbers mutually incommensurable. Regardless, the scale staggers the mind.) Sometimes these breaches were the result of clever social engineering; more frequently, someone forgot to properly salt password hashes or just stored or transmitted them in unencrypted plaintext. Occasionally, the data were even just left available at a deprecated or forgotten API endpoint. Identity security is challenging to get right, and those who had custody of user data were frequently subject to moral hazard.

The looming software stack . A combination of practical manufacturing limits ending Moore's law and a complexifying operating system and software stack led to a long-term stagnation in the perceived speed and fluidity of user experience with computers. For the most part, even as multicore CPUs become more widespread, software bloat grows more acute with each new operating system version. For many enterprise developers, there have been insufficient incentives to simplify software rather than to continue making it more complex. Minimalist software by and large remained the demesne of hackers and code golf enthusiasts.

For instance, TODO MS Word menu structure

Even websites with visually minimalist aesthetics often presented **Ceglowski2015**
a - Optional Reading: [Maciej Cegłowski, "The Website Obesity Crisis"](https://idlewords.c
$OptionalReading : [MaciejCegowski, ”BuildaBetterMonster : Morality, Machine$
$//idlewords.com/talks/build_{ab}etter_{m}onster.htm)–OptionalReading :$
$[MarkTarver, ”TheCathedralandtheBizarre”](http : //marktarver.com/thecathed$

Security breaches .

heartbleed

Morbidity in open-source software projects .

FOSS lanternfish

Spam .

We have a cascading stack of legacy software and strange interdependencies. We have an anemic FOSS program in general. Taken together, actually getting the functionality you want as a user often requires a proprietary platform anyway.

For the Urbit developers, "security" means both guaranteed cryptographic ownership and access, and *security against future utilization* or future-proofing.

Data ownership

Urbit is a network-first, compatibility-breaking

As of this writing, Urbit runs on any of several interpreters as a "hosted OS," or a

Let us posit a social operating system, or SOS; a protocol for network-oriented platforms to utilize to ensure that user requirements are met securely. If we enumerate user-oriented desiderata for a social operating system, surely the following must rank prominently:

- Privacy
- Security
- Ownership

The Urbit project does not completely solve all of these problems—for instance, pwned hardware—but it offers a reasonable set of solutions for many of the social and software issues raised by contemporary corporate practice on the World Wide Web.

1.2 Azimuth, the Urbit Address Space

In **Zooko2001**, digital cash pioneer Zooko Wilcox-O'Hearn postulated that a namespace cannot simultaneously possess three qualities:

1. distributedness ("in the sense that there is no central authority which can control the namespace, which is the same as saying that the namespace spans trust boundaries"),
2. security ("in the sense that name lookups cannot be forced to return incorrect values by an attacker, where the definition of "incorrect" is determined by some universal policy of name ownership"), and
3. human legibility (or interpretable by human users).
   This trilemma, dubbed Zooko's triangle, laid down a challenge to cryptographic researchers, who spent some effort to empirically refute the postulate.
   In the case of Urbit,
   Many modern printed textbooks have adopted a layout with prominent margins where small figures, tables, remarks and just about everything else can be displayed. Arguably, this layout helps to organise the discussion by separating the main text from the ancillary material, which at the same time is very close to the point in the text where it is referenced.
   This document does not aim to be an apology of wide margins, for there are many better suited authors for this task; the purpose of

all these words is just to fill the space so that the reader can see how a book written with the kaobook class looks like. Meanwhile, I shall also try to illustrate the features of the class.

The main ideas behind kaobook come from this blog post, and actually the name of the class is dedicated to the author of the post, Ken Arroyo Ohori, which has kindly allowed me to create a class based on his thesis. Therefore, if you want to know more reasons to prefer a 1.5-column layout for your books, be sure to read his blog post.

Another source of inspiration, as you may have noticed, is the Tufte-Latex Class. The fact that the design is similar is due to the fact that it is very difficult to improve something wich is already so good. However, I like to think that this class is more flexible than Tufte-Latex. For instance, I have tried to use only standard packages and to implement as little as possible from scratch;[1] therefore, it should be pretty easy to customise anything, provided that you read the documentation of the package that provides that feature.

1: This also means that understanding and contributing to the class development is made easier. Indeed, many things still need to be improved, so if you are interested, check out the repository on github!

In this book I shall illustrate the main features of the class and provide information about how to use and change things. Let us get started.

In addition to the pronounceable @p, the sigil system affords a unique visual representation of each addressable point less than 2^{32}.

1.3 Accessing Urbit

The `kaobook` class focuses more about the document structure than about the style. Indeed, it is a well-known LaTeX principle that structure and style should be separated as much as possible (see also Section **??** on page ??). This means that this class will only provide commands, environments and in general, the opportunity to do things, which the user may or may not use. Actually, some stylistic matters are embedded in the class, but the user is able to customise them with ease.

The main features are the following:

Page Layout The text width is reduced to improve readability and make space for the margins, where any sort of elements can be displayed.

Chapter Headings As opposed to Tufte-Latex, we provide a variety of chapter headings among which to choose; examples will be seen in later chapters.

Page Headers They span the whole page, margins included, and, in twoside mode, display alternatively the chapter and the section name.[2]

2: This is another departure from Tufte's design.

Matters The commands `\frontmatter`, `\mainmatter` and `\backmatter` have been redefined in order to have automatically wide margins in the main matter, and narrow margins in the front and back matters. However, the page style can be changed at any moment, even in the middle of the document.

Margin text We provide commands `\sidenote` and `\marginnote` to put text in the margins.[3]

3: Sidenotes (like this!) are numbered while marginnotes are not

Margin figs/tabs A couple of useful environments is `marginfigure` and `margintable`, which, not surprisingly, allow you to put figures and tables in the margins (*cfr.* Figure **??**).

Margin toc Finally, since we have wide margins, why don't add a little table of contents in them? See `\margintoc` for that.

Hyperref `hyperref` is loaded and by default we try to add bookmarks in a sensible way; in particular, the bookmarks levels are automatically reset at `\appendix` and `\backmatter`. Moreover, we also provide a small package to ease the hyperreferencing of other parts of the text.

Bibliography We want the reader to be able to know what has been cited without having to go to the end of the document every time, so citations go in the margins as well as at the end, as in Tufte-Latex. Unlike that class, however, you are free to customise the citations as you wish.

Figure 1.1: Some sigils
https://media.urbit.org/site/posts/essays/help-the-environment.jpg

1.4 Developing for Urbit

Urbit development can be divided into three cases:

a) Kernel development
b) Userspace development, Urbit-side (`%gall` and generators)
c) Userspace development, client-side (Urbit API)

This guide focuses on getting the reader up to speed on the second development case early, then branches out.
We encourage the reader to approach each example and exercise in the following spirit:

a) Identify the input and outputs, preferably at the data type level and contents.
b) Reason analogically from other Hoon examples available in the text and elsewhere.
c) Create and complete an outline of the code content.
d) Devise and compose a suitable test suite.

Language Essentials

Nock, A Combinator Language 2

2.1 Primitive rules and the combinator calculus

A combinator calculus is one way of writing primitive computational systems. Combinatory logic allows one to eliminate the need for variables (unknown quantities like x) and thus deal exclusively (?) with pure functions.
one combinator calculus
To understand how Nock expressions produce nouns as pure stateless functions, we need to introduce the *subject*. The subject is somewhat analogous to a namespace in other programming languages; it encompasses the computational context and the arguments. Another way to put it is that the subject *is* the argument to the Nock formula: not all of the subject may be used in evaluating the formula, but it is all present.
Nock is a crash-only language; that is, while it can emit events that are interpretable by the runtime as errors that can be handled, Nock itself fails when an invalid operation occurs.
Nock is a standard of behavior, not necessarily an actual machine. (It is an actual machine, of course, as a fallback, but the point is that any Nock virtual machine should implement the same behavior.)
We like to think of this analogous to solving a matrix. Formally, given an equation

$$A\vec{x} = \vec{b}$$

the solution should be obtained as

$$A^{-1}A\vec{x} = A^{-1}\vec{b} \rightarrow \vec{x} = A^{-1}\vec{b}$$

This is correct, but often computationally inefficient to achieve. Therefore we use this behavior as a standard definition for $\vec{x}$, but may actually obtain $\vec{x}$ using other more efficient methods. Keep this in mind with Nock: one has to know the specification but doesn't have to follow suit to implement it this way (thus, jet-accelerated Nock, Section **??**).

Nock 4K

The current version of Nock, Nock 4K, consists of six primitive rules as well as a handful of compound adjuncts. The primitive rules are conventionally written in an explanatory pseudocode:

```
*[a 0 b]            /[b a]
*[a 1 b]            b
*[a 2 b c]          *[*[a b] *[a c]]
*[a 3 b]            ?*[a b]
*[a 4 b]            +*[a b]
```

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

with the following operations:

- ▶ `*` is the *evaluate* operator, which operates on a cell of `[subject formula]`;
- ▶ `/` is the *slot* operator or address `b` of [tree] `a`;
- ▶ `?` is the *cell* operator, testing whether its operand is a cell.
- ▶ `+` is the *increment* operator.
- ▶ `=` is the *equality* operator, checking for structural equality of the operands evaluated against the subject `a`.

It is also instructive to write these as mathematical rules:

$$\begin{aligned}
*_0[a](b) &:= a_b \\
*_1[a](b) &:= b \\
*_2[a](b,c) &:= *(*[a](b), *[a](c)) \\
*_3[a](b) &:= \begin{cases} 0 & \text{if cell} \\ 1 & \text{if atom} \end{cases} \\
*_4[a](b) &:= *(a,b) + 1 \\
_5[a](b,c) &:= ((a,b) \overset{?}{=} *(a,c))
\end{aligned}$$

where `*` is the generic evaluate operator.
Each rule is referred to by its number; *e.g.,* "Nock 3" refers to the cell test rule.
Nock operates on unsigned integers, with zero 0 expressing the null or empty value. Frequently this is written as a tilde, ˜ or ~. This value plays a complex role similar to `NULL` and `'\0'` in C and other programming languages—although, critically, it is still numeric.

Nock 0, Addressing

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

$$*_0(a,b) := a_b$$

Nock Zero allows the retrieval of nouns against the Nock subject. Data access requires knowing the address and how to retrieve the corresponding value at that address. The slot operator expresses this relationship using `a` as the subject and the atom `b` as the one-indexed address.
Every structure in Nock is a binary tree. Elements are enumerated left-to-right starting at 1 for the entire tree.
One common convention is to store values at the leftward leaves of rightward branches; this produces a cascade of values at addresses $2^n - 2$.
By hand,
In the Dojo, you may evaluate this statement using the `.*` "dot-tar" rune:

The address of a value in the Nock binary tree has no direct correspondence to its address in physical memory. This latter is handled by the Nock runtime, avoiding the use of pointers in Nock code.

`.*` implements Nock Two, which is of course *evaluate*.

```
.*(TODO)
```

You may also use `++mock`
virtualization arm computes a formula. '++mock' is Nock in Nock,

however, so it is not very fast or efficient.
‘++mock‘ returns a tagged cell, which indicates the kinds of things that can go awry:
- ‘- ‘- ‘
‘++mock‘ is used in Gall and Hoon to virtualize Nock calculations and intercept scrys. It is also used in Aqua, the testnet infrastructure of virtual ships.

Nock 1, Constant Reduction

`*[a 1 b]`	`b`

$$*_1(a,b) := b$$

Nock One simply returns the constant value of noun `b`.

Nock 2, Evaluate

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

$$*_2[a](b,c) := *(*[a](b), *[a](c))$$

Nock 3, Test Cell

Nock Three zero as true (because there is one way to be right and many ways to be wrong).

Although unusual, Nock is by no me
the only language to adopt 0 as the s
dard of truth. The POSIX-compliant sh
such as Bash adopt the convention th
is TRUE. So do Ruby and Scheme, althou
with caveats.

Nock 4, Increment

Nock 5, Test Equivalence

Let us examine some Nock samples by hand and see if we can reconstruct what they do. We will then create some new short programs and apply them by hand via the Nock rules.

2.2 Compound rules

For the convenience of programmers working directly with Nock (largely the implementers of Hoon), a number of compound rules were defined that reduce to the primitive rules. These implement slightly higher-order conventions such as a decision operator. Each of these provide syntactic sugar that render Nock manipulations slightly less cumbersome.

`*[a 6 b c d]`	`*[a *[[c d] 0 *[[2 3] 0 *[a 4 4 b]]]]`
`*[a 7 b c]`	`*[*[a b] c]`
`*[a 8 b c]`	`*[[*[a b] a] c]`
`*[a 9 b c]`	`*[*[a c] 2 [0 1] 0 b]`
`*[a 10 [b c] d]`	`#[b *[a c] *[a d]]`
`*[a 11 [b c] d]`	`*[[*[a c] *[a d]] 0 3]`
`*[a 11 b c]`	`*[a c]`

with the following operation:

- # is the *replace* operator, which edits a noun by replacing part of it with another piece.

As mathematical rules, these would be:

$$
\begin{aligned}
*_6[a](b,c,d) &:= \begin{cases} *[a](c) & \text{if } b \\ *[a](d) & \text{otherwise} \end{cases} \\
*_7[a](b,c) &:= *[*[a](b)](c) \\
*_8[a](b,c) &:= *[*[*[a](b)](a)](c) \\
*_9[a](b,c) &:= \begin{cases} 0 & \text{if cell} \\ 1 & \text{if atom} \end{cases} \\
*_{10}[a](b,c,d) &:= *(a,b)+1 \\
_{11}[a](b,c,d) &:= ((a,b) \overset{?}{=} *(a,c)) *_{11}\ [a](b,c) := (*(a,b) \overset{?}{=} *(a,c))
\end{aligned}
$$

where $*$ is the generic evaluate operator.

Nock 6, Conditional Branch

Nock 7, Compose

Nock 8, Declare Variable

Nock 9, Produce Arm of Core

Nock 10, Replace

Nock 11, Hint to Interpreter

With Nock under your belt, many of the quirks of Hoon become more legible. For instance, since everything in Nock is a binary tree, so also everything in Hoon. Nock also naturally gives rise to cores, which are a way of pairing operations and data in a cell.

There's also a "fake Nock" Rule Twelve, .^ "dotket", which exposes a namespace into Arvo. More details on this follow in Section **??**.

Although Nock is the runtime language of Urbit, developers write actual code using Hoon. Given a Hoon expression, you can produce the equivalent Nock formula using != "zaptis".

After this chapter, you may never write Nock code again. That's fine! We need to understand Nock to understand Hoon, but will not need to compose in Nock directly to do any work in Urbit, even low-level work. (There is no `inline` equivalent.)

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

> !=((add 1 1))
[8 [9 36 0 1.023] 9 2 10 [6 [7 [0 3] 1 1] 7 [0 3] 1 1] 0 2]
```

(Why do these differ so much? `++add` is doing a bit more than just adding a raw 1 to an unsigned integer. We'll walk through this function later in Section TODO.)

One last piece is necessary for us to effectively interpret Nock code: the implicit cons. Cons is a Lisp function to construct a pair, or what in Nock terms we call a cell. Many times we find Nock expressions in which the operand is a cell, and so TODO

Nock Examples

We will work through several Nock programs by hand. Since each Nock program is a pure function and emits no side effects, when we have applied all of the rules to achieve a final value, we are done calculating the expression.
Infamously, Nock does not have a native decrement operator, only an increment (Rule Four). Let us dissect a simple decrement operation in Nock:

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

which can be restated in one line as

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

or in many lines as

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

(It’s advantageous to see both.)
We can pattern-match a bit to figure out what the pieces of the Nock are supposed to be in higher-level Hoon. From the Hoon, we can expect to see a few kinds of structures: a trap, a test, a ‘sample‘. At a glance, we seem to see Rules One, Five, Six, Eight, and Nine being used. Let’s dig in.
(Do you see all those ‘0 6‘ pieces? Rule Zero means to grab a value

from an address, and what's at address '6'? The 'sample', we'll need that frequently.)
The outermost rule is Rule Eight '*[a 8 b c]→*[[*[a b] a] c]' computed against an unknown subject (because this is a gate). It has two children, the 'b' '[0 1]' and the 'c' which is much longer. Rule Eight is a sugar formula which essentially says, run '*[a b]' and then make that the head of a new subject, then compute 'c' against that new subject. '[0 1]' grabs the first argument of the 'sample' in the 'payload', which is represented in Hoon by 'a=@'.
The main formula is then the body of the gate. It's another Rule Eight, this time to calculate the 'b=0' line of the Hoon.
There's a Rule One, or constant reduction to return the bare value resulting from the formula.
Then one more Rule Eight (the last one!). This one creates the default subject for the trap `$`; this is implicit in Hoon.
Next, a Rule Six. This is an 'if'/'then'/'else' clause, so we expect a test and two branches.
- The test is calculated with Rule Five, an equality test between the address '30' of the subject and the increment of the 'sample'. In Hoon, '=(a +(b))'.
- The '[0 6]' returns the 'sample' address.
- The other branch is a Rule Nine reboot of the subject via Rule Ten. Note the '[4 0 6]' increment of the 'sample'.

Finally, Rule Nine is invoked with '[9 2 0 1]', which grabs a particular arm of the subject and executes it.
Contrast the built-in '++dec' arm:
"'nock > !=((dec 1)) [8 [9 2.398 0 1.023] 9 2 10 [6 7 [0 3] 1 1] 0 2] "'
for which the Hoon is:
"'hoon ++ dec |= a=@ ?< =(0 a) =+ b=0 |- ^@? := $(a + (b))b$(b +(b))
"'

Scan for pieces you recognize: the beginning of a cell is frequently the rule being applied.
In tall form,
"'hoon [8 [9 [2.398 [0 1.023]]] [9 2 [10 [6 7 [0 3] 1 1] [0 2]]]] "'
What's going on with the above `++dec` is that the Arvo-shaped subject is being addressed into at '2.398', then some internal Rule Nine/Ten/Six/Seven processing happens.

2.3 Kelvin versioning

Each version of Nock
telescopic versioning

2.4 Exercises

Compose a Nock interpreter in a language of your choice. (These aren't full Arvo interpreters, of course, since you don't have the Hoon, `%zuse`, and vane subject present.)

Elements of Hoon 3

3.1 Reading the Runes

The goals of this section are for you to be able to:

a) Identify Hoon runes and children in both inline and long-form syntax.
b) Trace a short Hoon expression to its final result.
c) Produce output as a side effect using the `&` rune.

For the first several exercises, we will suggest that you utilize one of these methods in particular so that you get a feel for how each works. After you are more comfortable working with Hoon code on Urbit, we will refrain.
Each rune accepts at least one child, except for `!!` "zapzap".

Although not a compiled language, the binary-tree structure of Hoon can lead to fairly involved programs which are difficult to type and parse as directly as some other languages afford. We instead encourage you to use one of three methods to run Hoon programs:

a) The Dojo REPL, which offers some convenient shortcuts to modify the subject for subsequent commands.
b) A tight loop of text editor and running fakezod.
c) The online interactive sandbox at `https://approaching-urbit.com`

3.2 Irregular Forms

Many runes in common currency are not written in their regular form (tall or wide), but rather using syntactic sugar as irregular. For instance, `%-` "cenhep" is most frequently written using parentheses `()` which permits a Lisp-like calling syntax:

```
(add 1 2)
```

is equivalent to

```
%- add [1 2]
```

is also equivalent to

```
%-(add [1 2])
```

Hoon parses to an abstract syntax tree (AST), which includes cleaning up all of the sugar syntax and non-primitive runes. To see the AST of any given Hoon expression, use `!,` "zapcom".

```
> !,(*hoon TODO)
TODO
```

3.3 Nouns

All values in Urbit are nouns, meaning either atoms or cells. An atom is an unsigned integer, which frequently has an *aura* associated with it. A cell is a pair of nouns.

3.4 Hoon as Nock Macro

The point of employing Hoon is, of course, that Hoon compiles to Nock. Rather than even say *compile*, however, we should really just say Hoon is a *macro* of Nock. Each Hoon rune, data structure, and effect corresponds to a well-defined Nock primitive form. We may say that Hoon is to Nock as C is to assembler, except that the Hoon-to-Nock transformation is completely specified and portable. Hoon is ultimately defined in terms of Nock; many Hoon runes are defined in terms of other more fundamental Hoon runes, but all runes parse unambiguously to Nock expressions.

Hoon values are addressed as elements in a binary tree.

Binary trees are explained in more detail in Section ??.

Be careful to not confuse `=(a b)`, which evaluates to `.=`, with the various `?` runes like `?=`.

3.5 Key Data Structures

Lists

Text

Hoon recognizes two basic text types: the *cord* or `@t` and the *tape*. Cords are single atoms containing the text as UTF-8 bytes interpreted as a single stacked number. Tapes are lists of individual one-element cords.

Both cords and tapes are casually referred to as strings.

Cords are useful as a compact primary storage and data transfer format, but frequently parsing and processing involves converting the text into tape format. There are more utilities for handling tapes, as they are already broken up in a legible manner.

Lists are null-terminated, and thus are tapes.

```
++  trip
  |=  a=@  ^-  tape
  ?:  =(0 (met 3 a))  ~
  [^-(@ta (end 3 1 a)) $(a (rsh 3 1 a))]
```

For instance, `trip` converts a cord to a tape; `crip` does the opposite. All text in Urbit is UTF-8 (*a fortiori* ASCII). The `@c` UTF-32 aura is only used by `%dill`and Hood (the Dojo terminal agent).

`++ crip |=(a=tape ‘@t‘(rap 3 a))`

`++rap` assembles the list interpreted as cords with block size of 2^3 (in this case).

Cores, Gates, Doors

> anyway you can explicitly set the sample in an iron core but you can't use it with +roll New messages below 11:54 (master-morzod) `%gold` is the default, read/write everything; `%iron` is for functions (write to the sample with a contravariant nest check), `%lead` is "hide the whole payload", `%zinc` completes the matrix but has probably never been used `%iron` lets you refer to a typed gate (without wetness), without depending on all the details of the subject it was defined against `%lead` lets you export a library interface but hide the implementation details

Molds

Maps, Sets, Tree

3.6 Generators

Naked Generators

`%say` generators

`%ask` generators

3.7 Libraries

3.8 Unit Tests

3.9 Building Code

Advanced Hoon 4

4.1 Cores

Variadicity

Genericity

4.2 Molds

Polymorphism

4.3 Rune Families

4.4 Marks and Structures

4.5 Helpful Tools

4.6 Deep Dives

Text Stream Parsing

JSON Parsing

HTML/XML Parsing

System Development

The Kernel 5

5.1 Arvo

Arvo is essentially an event handler which can coordinate and dispatch messages between vanes as well as emit `%unix` events to the underlying (presumed Unix-compatible) host OS. Arvo does not carry out several tasks specific to the machine hardware, such as memory allocation, system thread management, and hardware- or firmware-level operations. These are left to the king and serf, or the daemon processes which together run Arvo. Collectively, the system-level instrumentation of Arvo is described in Chapter **??**.

`%zuse` and `%lull`

`%zuse` and `%lull` define common structures and library functions for Arvo.
subject wrapped

5.2 `%ames`, A Network

In a sense, `%ames` is the operative definition of an urbit on the network. That is, from outside of one's own urbit, the only specification that must be hewed to is that `%ames` behaves a certain way in response to events.
`%ames` implements a system expecting—and delivering—guaranteed one-time delivery. This derives from an observation by **Yarvin2016** in the Whitepaper: "bus v. commands whatever"
UDP packet structure
network events acks & nacks

5.3 `%behn`, A Timer

`%behn` is a simple vane that promises to emit events after—but never before—their timestamp. This guarantee
As the shortest vane, we commend `%behn` to the student as an excellent subject for a first dive into the structure of a vane.
`%behn` maintains an event handler and a state.
Any task may have one of the following states:

```
%born   born:event-core
%rest   (rest:event-core date=p.task)
%drip   (drip:event-core move=p.task)
%huck   (huck:event-core syn.task)
%trim   trim:event-core
%vega   vega:event-core
%wait   (wait:event-core date=p.task)
%wake   (wake:event-core error=~)
```

5.4 `%clay`, A File System

`++ford`, A Build System

Scrying

Marks and conversions

5.5 `%dill`, A Terminal driver

5.6 `%eyre` and `%iris`, Server and Client Vanes

5.7 `%jael`, Secretkeeper

`%jael` weighs in as one of the shorter vanes

As of Arvo XXX K, the

s named after Jael, the wife of Heber,
ept mum and slew fleeing enemy
l Sisera in Judges 4.

5.8 Azimuth, Address Space Management

5.9 The Hoon Parser

6 Userspace

Modern %gall is sometimes called "static Gall," in contrast to an earlier specification "dynamic Gall." Dynamic Gall did not specify the arms and permitted each agent its own structure; in practice, this proved to be difficult for programmers to maintain in a consistent manner, leading to code refactors and maintenance of defunct arms for backwards compatibility of agents.

6.1 %gall, A Runtime Agent

6.2 Deep Dives in %gall

Each of the following case studies is drawn from published code, most of it incorporated into the Urbit userspace. In some cases, the original code uses conventions we have not yet introduced; we have simplified these to rely on the runes introduced in the main text through Chapter **??**.

Chat CLI

Drum and Helm

Bitcoin API

Bots

6.3 Threading with Spider

6.4 Urbit API

6.5 Deep Dives with Urbit API

Time (Clock)

Publish

%graph-store

Supporting Urbit 7

7.1 Booting and Pills

7.2 `%unix` Events

7.3 Nock Virtual Machines

`++mock`

King and Serf Daemons

Vere (Reference C Implementation)

King Haskell (Haskell Implementation)

Jaque (JVM Implementation)

7.4 Jetting

Jet matching and the dashboard

Concluding Remarks 8

8.1 Booting and Pills

Appendix

Appendices A

A.1 Comprehensive table of Hoon runes

A.2 Hoon versions

A.3 Nock versions

A.4 Hoon comparison with other languages

A.5 `%zuse`/`%lull` versions

A.6 Textbook changelog