An Approach to Developing on Urbit

N E Davis

June 11, 2021

Malancandra & Sons

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

Publisher
First published in the year of our Lord 2021 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

(quotation in Spanish from JLB about "to explain (or pass judgment on) an event is to link it to another; on Tlon, that joining-together is a posterior state of the subject, and can neither affect nor illuminate the prior states")

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.

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 and file bloat

Even websites with visually minimalist aesthetics often presented Ceglowski2015a - Optional Reading: Maciej Cegłowski, "The Website Obesity Crisis"

- Optional Reading: Maciej Cegłowski, "Build a Better Monster: Morality, Machine Learning, and Mass Surveillance"

- Optional Reading: Mark Tarver, "The Cathedral and the Bizarre"

Security breaches As the softwar stack grows, dependencies become opaque to downstream developers and users. Upstream vulnerabilities have led to zero-day exploits and security breaches. For instance, in 2014 the popular OpenSSL cryptography package had a bug of two years' standing revealed, Heartbleed. This flaw in the Transport Layer Security (TLS) exposed memory buffers adjacent to

Instant-messaging protocols relying on the `libpurple` were impacted by an out-of-bounds write flaw in 2017, potentially permitting denial-of-service attacks or arbitrary code execution.

These two examples are not cherry-picked: other examples abound. The point stands that security breaches in the software stack render reliant software vulnerable in unpredictable ways.

Morbidity in open-source software projects The rise of the free and open-source software (FOSS) movement has been enormously influential on software development and the end-user experience. Spearheaded by Richard Stallman's GNU Project and Linus Torvalds' Linux operating system, FOSS rapidly overtook enterprise software offerings in terms of feature parity and upstream utilization.

Unfortunately, open-source software products are frequently broken in ways that are opaque to relatively nontechnical users:

1. FOSS can be construed as operating under a parasitic model. Most real innovation happens outside of open-source projects, which are often clones of more successful proprietary software packages (LibreOffice/Microsoft Office, GIMP/Adobe Photoshop, Inkscape/Adobe Illustrator), and/or a clever way for a company to farm out development to free community labor (OpenOffice/Oracle, Ubuntu/Canonical, Darwin/Apple). Thus even FOSS successes are often copies of proprietary antecedents.
2. FOSS suffers from [what one observer has dubbed](http://marktarver.com/thecathedralandthebizarre.html) "financial deficiency disease." Even popular, well-used packages may have little oversight and funding for developers. As alluded to above, OpenSSL was found in 2014 to have only one full-time developer despite being used by 66% of Internet users. Very few companies have succeeded in being FOSS-first (as opposed to FOSS-sometimes).
3. FOSS has a hard time responding to customer demands. The DIY ethos espoused by FOSS developers has often led to demurrage when features are requested. This is the infamous response, "If you need it, why don't you build it yourself?" Many users are unable to commit the time to implement the necessary features, and most FOSS projects do not have full-time developers and existing market dynamics sufficient to motivate rapid development.

Even companies that loudly proclaimed support for "data liberation" used this FOSS openness like a lanternfish to later replace an open protocol (e.g., Google Talk) with a proprietary one (Google Hangouts).

Given the cascading stack of legacy software and strange interdependencies, actually getting the secure functionality a user wants often requires a proprietary platform anyway, undermining the aims of FOSS end-user applications and libraries.

Identity is cheap Identity itself is cheap: it costs botnets and spammers nothing to spin up new email addresses and new false identities. Game-theoretically, spammers thrive in an environment where identity is close to free.

Identity is also dear: losing a password in a breach can cause at best hours of resetting service logins and at worst the trauma and legal process of recovering from identity theft.

The foregoing summation may read as a bit emotional relative to what the reader is accustomed to reading in an academic textbook. This is because the structure of our digital life matters as much as the content, and we have been ill-served to date by the incentives and powers that be.

Enter Urbit, stage right.

Why Urbit

"Urbit is a clean slate reimagining of the operating system as an 'overlay OS', and a decentralized digital identity system including username, network address and crypto wallet." (Tlon)

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. Urbit has been built to provide an Internet where communities can thrive without meddling or interference by third parties, and where what you build truly belongs to you.

"[Urbit is] ultimately a hosted OS ([residing] on top of Linux) with an immutable file system with the additional purpose that you build applications distributed-first in a manner where clients store their own data." (['scare-junba'](https://news.ycombinator.com/item?id=21672481))

Your Urbit is a personal server built as a functional-as-in-language operating system that runs as a virtual machine on top of whatever. (Sometimes the developers refer to this arrangement as a "hosted OS," but they don't mean as in VMWare or VirtualBox or even containerization.) The Urbit vision is the unification of services and data around a scarce futureproof identity on an innately secure platform. Briefly put, Urbit requires you to have an Urbit OS (which runs your code, stores your data, etc.) and an Urbit ID (which secures your ownership of said code and data).

Urbit provides an excellent example of a visionary complex system which is radical (returning to the roots of computing) and forward-looking—and yet still small enough for us to grok all of the major moving parts in the system. As a "hundred-year computer," Urbit represents how computing could work when computing power approaches negligible cost and bandwidth becomes effectively unlimited (or at least not limiting TODO Ted disagrees), instead focusing on the quality of user experience and user security. We have found that Urbit is worthy of study in its own right as a compelling clean-state architecture embracing several innovative ideas at its base.

Legible future-proof program stack The core of Urbit is an operating function, or a functional-as-in-language operating system. That is, there is a lifecycle function which receives a state and an event, processes the event, and yields a new state.

$$L(\sigma, \varepsilon) \to \sigma'.$$

The lifecycle function and state are sometimes called the Urbit OS to distinguish them from other aspects of the Urbit project when ambiguity is present. The Urbit OS lifecycle function is written in a language called Nock and provides operational affordances through the Arvo operating core. A schematic representation is frequently used:

: width=25

At its base, Arvo is an encrypted event log yielding a particular state. The Nock virtual machine is like Urbit's version of assembler language, and it may in principle be implemented on top of any hardware. Hoon is Urbit's equivalent of C, a higher-level language with useful macros and APIs for building out software. Arvo runs atop these definitions. The Nock VM runs on a binary interpreter layer on top of actual hardware.

The user can think of Urbit OS as a virtual machine which allows everything upstack to be agnostic to the hardware, and handles everything downstack. (Urbit has sometimes been described as an operating function, and this is what that means.) Everything is implemented as a unique stateful instance, called a "ship".

: width=50

The vanes of Arvo provide services: `%ames` provides network interactivity, `%clay` provides filesystem services and builds, `%jael` provides cryptographic operations, and so forth. On top of these are built the userspace apps.

We will take a closer look at every part of this system in Chapter ??.

: width=100

As a "hosted OS," Urbit doesn't seek to replace mainline operating systems. Indeed, presumptively its Nock virtual machine could be run quite close to the bare metal, but Urbit itself would still require some provision of memory management, hardware drivers, and input/output services. The overarching goal of the Urbit project is instead to replace the insecure messaging and service platforms and protocols used across the current web.

Urbit was designed on the principle that inheriting old platform code is a developer antipattern, given the complexities, vagaries, and vulnerabilities of legacy OSs. In other words, things must break to be fixed. Thus Urbit interfaces with other systems, but is a world unto itself internally.

Data security TODO

Cryptographic ownership We noted above that Urbit OS is an encrypted event log. Urbit also acts as a universal single sign-on (SSO) for the platform and for services instrumented to work with Urbit calls. Since the Urbit address space is finite, each Urbit ID has inherent value within the system and should be a closely guarded secret. An instantiation of your Urbit ID is frequently called a ship, which lodges on your filesystem at a folder called a pier.

Following in the footsteps of other blockchain technologies, Urbit secures ownership of unique access points in the Urbit address space using Azimuth. Currently Azimuth is deployed on top of the Ethereum blockchain.

See Section ?? for more details.

Urbit IDs have mnemonic names attached to them, although fundamentally they are only a number in the address space. For instance, one example address on Urbit is `dopzod-binfyr`, the unique ID one

user owns, corresponding to the 32-bit address `0xeb2a.5a32` in hexadecimal.

On this network-oriented platform, users provide data to service endpoints, retaining their data rather than farming it out. While no control can be exercised over data once sent out, a proposed reputation system can penalize bad actors in the system with reduced network access and other sanctions.

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:

TODO

Thompson1984

The system is designed to be transparent. Something that runs on the Nock VM is of necessity open-source—no binary blobs! (As with Ken Thompson's "Reflections on Trusting Trust", one can't necessarily trust what's below completely, but that's a problem with any system one did not build oneself from the bare metal up.)

Identity on the current Web is frequently ephemeral and difficult to distinguish from spam. Identity on Urbit is scarce and stable, much like moving into a house. The SSO aspect of the system means that you have to remember and use many fewer passwords, and the cryptographic security layers means that as long as you treat your master key like your Bitcoin wallet you will have perpetual security.

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. Many think that it is better to attempt to fix the challenges of data control, privacy, and equity on the current web: Sovrin, WebAssembly, InterPlanetary File System, Holochain, Space, and Scuttlebutt each, in their own way, attack the same problems that Urbit seeks to solve, and each is worthy of the reader's further study.

All in all, Urbit like Bitcoin and (the best) blockchain applications seeks to securely deliver on the aims of the old Cypherpunk movement of the 1980s and 1990s: digital security, digital autonomy.

At this point, you may feel confused as to what exactly Urbit is. That's understandable: it's hard to explain a new system in full until it has started to manifest new and interesting features with broader repercussions. For comparison, consider the following two interviews from much earlier in the history of the public Internet:

- Bill Gates on David Letterman, 1995 (an attempt to explain the Internet before almost anyone grokked it)
- David Bowie on the BBC, 1999 (a prophecy which grasps the essence without the technicality)

On this basis, it's safe to say that Gates got it, but Bowie "got it." Their interlocutors did not.

1.2 Azimuth, the Urbit Address Space

Urbit address points are allocated sequentially from 0 to $2^{128} = 340\,282\,366\,920\,938\,463\,46$ (340 undecillion, 3.4×10^{38}). The maximum address space value in this representation is 128 bits wide, although most points in use today are 32 bits wide or smaller.

Urbit is structured with a hierarchy of addressable points, and bands of smaller values have more "heft" in the system and broker access for higher-addressed points. The structure of the address space reveals the governance structure of the Urbit project itself:

Bit width	Total number	Title	Role
8-bit points	256	Galaxies	Provide peer discovery and packet routing as well as network pro
16-bit points	$2^{16} - 256 = 65\,280$	Stars	Routing & Allocate peer discovery services, handle distribution o
32-bit points	$2^{32} - 2^{16} = 4\,294\,901\,760$	Planets	Act as primary single-user identities.
64-bit points	$2^{64} - 2^{32} \approx 1.84 \times 10^{19}$	Moons	Act as planet-bound points (devices, bots); each planet has 2^{32} m
128-bit points	$2^{128} - 2^{64} \approx 3.4 \times 10^{38}$	Comets	Act as anonymous disposable zero-reputation points (bots, single

Naming points

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.

The Urbit ID system resolves Zooko's triangle by using peer-to-peer routing after discovery, by strictly limiting identity as a scarce and reputation-bearing good, and by assigning each addressable point of the 128-bit address space a unique and memor(iz)able name.

Each point receives a unique pronounceable name constructed from a list of 256 prefixes and 256 suffixes. For instance, point 0 is `~zod`, the root sponsoring galaxy of the `%ames` network. In fact, today on Urbit you frequently see the mnemonic address used as the primary pseudonymous identity and username. The identity problem is thereby solved without restrictive username requirements and collision-avoidance strategies.

Urbit uses a system of mnemonic syllables to uniquely identify each address point. These mnemonic names, called "'patp's" after their Hoon representation '@p', occur in a set of 256 suffixes (such as "zod") and 256 prefixes (such as "lit"). They were selected to be pronounceable but not meaningful.

Number	Prefix	Suffix
0	doz	zod
1	mar	nec
2	bin	bud
3	wan	wes
4	sam	sev
...	...	...
254	mip	nev
255	fip	fes

The first 256 points (the galaxies) simply take their name from the suffix of their address point. Subsequent points combine values: for instance, point 256 is marzod, point 480 is marrem, and point 67,985 is fogfel-folden. (Conventionally, a sigma ' ' is used in front of an address.)

The 256 galaxies have suffix-only names, and all higher addresses have prefix–suffix names. Two-syllable names always mean the point is a

Sigils are a visual corollary to mnemonic patp. Each 32-bit or lo address has a unique sigil, based the 512 component syllables. Sigils not intrinsic to Urbit, but they fo part of the metatextual environm that Urbit inhabits and they are quently used as a means of ready ferentiation and identity. TODO ages

star; four-syllable names are planets. Comets have rather cumbersome names: 67,985,463,345,234,345 corresponds to doztes-nodbel-palleg-ligbep with eight syllables.

The stars which correspond to a galaxy are suffixed with the galaxy's name; planet names are mangled so that one cannot tell which star or galaxy a planet corresponds to at a glance.

If a planet needs to change its sponsor, there is support for changing one's sponsor, in which another star can assume the role of peer discovery in case a star goes offline (a "dark star").

TODO we in principle care about address when dealing with a strange star or planet for the first time. A reputation system is under development, but hasn't yet seemed to be necessary. This is called [‘Censures‘](https://urbit.org/docs/glossary/censure/). Plus, at this point, identity is fairly cheap, abundant if not infinite. (Notably, not so cheap that spammers can thrive.)

Peer discovery, the primary role of stars besides planet allocation, is an important step in responsibly controlling network traffic. "The basic idea is, you need someone to sponsor your membership on the network. An address that can't find a sponsor is probably a bot or a spammer" ([docs](https://urbit.org/understanding-urbit/)).

A reputation system is available called [‘Censures‘](https://urbit.org/docs/glossary/cen
As spammers and bots are not yet present on the Urbit network in any significant quantity, Censures is not heavily used today. Galaxies and stars can censure (or lower the reputation of) lower-ranked points as a deterrent to bad behavior (defined as spamming, scamming, and spreading malware). Since good behavior is the default, only lowering reputation is supported.

- Reading: [Philip Monk ‘ wicdev-wisryt‘, "Designing a Permanent Personal Identity"](https://urbit.org/blog/pki-maze/)

Azimuth On-Chain

Azimuth is a public-key infrastructure (PKI) and is currently deployed as a series of smart contracts operating on the Ethereum blockchain. “Azimuth is basically two parts, a database of who owns which points, and a set of rules about what points and their owners can do” (Wolfe-Pauly). Azimuth points are not interchangeable tokens like ETH or many other cryptocurrencies: each point has a unique ID, a type, and associated privileges. The technical blockchain term for this kind of points is a “non-fungible token” (NTF).

Azimuth is located on the Ethereum blockchain at address [‘0x223c067f8cf28ae173ee5ca
or [‘azimuth.eth‘](https://etherscan.io/address/azimuth.eth).

Point ownership is secured by the Urbit HD (Hierarchical Deterministic) wallet, a collection of keys and addresses which allows you fine-grained control over accessing and administering your asset. The Urbit HD wallet is described in more detail in Section ??.

There is nothing intrinsic about Azimuth which requires Ethereum to work correctly, and in the future Azimuth will probably be moved entirely onto Urbit itself.

- Reading: [Galen Wolfe-Pauly, "Azimuth is On-Chain", through section "Azimuth"](https://urbit.org/blog/azimuth-is-on-chain/) - Optional Reading: [Ameer Rosic, "What is An Ethereum Token?"](https://blockgeeks.com/guides/ethereum-token/) - Code: [‘azimuth-js‘](https://github.com/urbit/azimuth-js)

Urbit Ownership and the Crypto Community

Because Urbit address space is finite, it in principle bears value similar to cryptocurrencies such as Bitcoin. You should hold your Urbit keys like a dragon's hoard: once lost, they are irrecoverable. No one else has a copy of your master ticket, which is the cryptographic information necessary to sell, launch, or administer your planet.

To be clear, we do not herein promote Urbit or ownership of any part thereof as a speculative crypto asset. Like blockchain and cryptocurrencies, Urbit may carry intrinsic value or it may be only so much (digital) paper. Right now, you can purchase available Azimuth points on [OpenSea.io](https://opensea.io/assets/urbit-id?query=urbit) and [urbit.live](https://urbit.live/buy), or you can buy directly from someone who has some available.

- Reading: [Wolf Tivy, "Why do Urbit stars cost so much?" (Quora answer)](https://www.quora.com/Why-do-Urbit-stars-cost-so-much) - Resource: [Urbit Live, "Urbit Network Stats"](https://urbit.live/stats) (set the dates to a much broader range and the currency type to ETH)

Galaxy Ownership and Star Access The sale of galaxies formed a role in initially funding Urbit, but to prevent too early sale and to modulate access to the network, most stars are locked by Ethereum smart contracts and unsellable until their unlock date. Star contracts will unlock through January 2025, at which point the full address space will be available (but not necessarily activated). Galaxy owners can sell or distribute stars as they see fit, and star owners can parcel out planets. However, since a star provides peer discovery services, it is imperative that a star with daughter planets remain online and up-to-date.

Much of the Urbit address space is locked and unspawnable to provide an artificial brake on supply and prevent overrunning the available address space. See "The Value of Urbit Address Space, Part 3" for extensive details on star and planet limitations and the associated Ethereum smart contracts.

: width=100

If a star ceases to provide peer-to-peer lookup services and software updates, a planet may find itself in a pickle. "Dark stars" are stars which have spawned daughter planets but are not running anymore. To mitigate this situation, planets can switch from one sponsoring star and move to another.

- Optional Reading: [Erik Newton ‘ patnes-rigtyn‘, Galen Wolfe-Pauly ‘ ravmel-ropdyl‘, ”The Value of Urbit Address Space, Part 1”](https://urbit.org/blog/va of-address-space-pt1/) - Optional Reading: [Erik Newton ‘ patnes-rigtyn‘, Galen Wolfe-Pauly ‘ ravmel-ropdyl‘, ”The Value of Urbit Address Space, Part 2”](https://urbit.org/blog/value-of-address-space-pt2/) - Optional Reading: [Erik Newton ‘ patnes-rigtyn‘, Galen Wolfe-Pauly ‘ ravmel-ropdyl‘, ”The Value of Urbit Address Space, Part 3”](https://urbit.org/blog/va of-address-space-pt3/)

The Azimuth PKI is quite sophisticated, and the associated Urbit HD wallet allows for nuance in point management. The Bridge interface is used to manage point operations in the browser. We will revisit all three in technical detail in Section ??. We will furthermore examine the internal operations of ‘azimuth-js‘ and the Ecliptic contracts.

1.3 A Frozen Operating System

The philosophy underlying Urbit bears a strange resemblance to mathematics: rather than running always as fast as one can to stay in the same place (a Red Queen’s race), one should instead establish a firm foundation on which to erect all future enterprises. In this view, the operating system should provide a permanently future-proof platform for launching your applications and storing your data—rather than a pastiche of hardware platforms and network specifications, all of that is hidden, ”driver-like.” The OS should explicitly obscure all of that and no reaching beneath the OS should be allowed.

Urbit frequently refers to its way of doing things as ”Martian.”

From [the docs](https://web.archive.org/web/20140424223249/http://urbit.org/comm computing/):

> Normally, when normal people release normal software, they count by fractions, and they count up. Thus, they can keep extending and revising their systems incrementally. This is generally considered a good thing. It generally is.
>
> In some cases, however, specifications needs to be permanently frozen. This requirement is generally found in the context of standards. Some standards are extensible or versionable, but some are not. ASCII, for instance, is permafrozen. So is IPv4 (its relationship to IPv6 is little more than nominal—if they were really the same protocol, they’d have the same ethertype). Moreover, many standards render themselves incompatible in practice through excessive enthusiasm for extensibility. They may not be perma-frozen, but they probably should be.
>
> The true, Martian way to perma-freeze a system is what I call Kelvin versioning. In Kelvin versioning, releases count down by integer degrees Kelvin. At absolute zero, the system can no longer be changed. At 1K, one more modification is possible. And so on. ([Yarvin2017])

Urbit is not, of course, the only system to adopt an asymptotic approach to its final outcome. Donald Knuth, famous for many reasons but in this particular instance for the typesetting system TEX, has specified that TEX versions incrementally approach π. TEX will reach π definitively upon the date of Knuth’s death, at which point all remaining bugs are instantly transformed into features and the version becomes π. The current version of TeX is 3.14159265.

In other words, Urbit is intended to cool towards absolute zero, at which point its specification is locked in forever and no further changes

are countenanced. This doesn't apply to everything in the system—"there simply isn't that much that needs to be versioned with a kelvin" (nidsut-tomdun)—but it does apply to the most core components in the system.

In this light, when we talk about Urbit we talk about three things:

1. Crystalline Urbit (the promised frozen core, 0K)
2. Fluid Urbit (the practice, mercurial and turbulent but starting to take shape)
3. Mechanical Urbit (the under-the-hood elements, still a chaos lurching into being, although much less primeval than before)

Think of the hypothetical structure of Jupiter: clouds over a sea of metallic hydrogen over a diamond as big as Earth. TODO image

1.4 Developing for Urbit

The primary aim of this textbook is to expound Urbit in sufficient depth that you can approach it as an effective software developer. We assume previous programming experience of one kind or another, not necessarily in a functional language.

Urbit development can be divided into three cases:

1. Kernel development
2. Userspace development, Urbit-side (`%gall` and generators)
3. 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 into the two others. With a solid foundation in `%gall`, the reader will be well-equipped to handle demands in either of the other domains. We encourage the reader to approach each example and exercise in the following spirit:

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

Practicalities

We recommend organizing all development work discussed in this textbook into a single folder containing code folders, version control repositories of code, data, and so forth. For convenience, we locate this folder at `~/urbit`. Should you choose to install Urbit in this location, you should use the folder `~/urbit/bin` to contain the Urbit executables.

Development Ships (Fakezods) When started directly by the user, Urbit enters a read-evaluate-print loop (REPL) immediately after booting. This interface, called Dojo, can process many Hoon expressions and provides some support for administering apps, interacting with command-line interface (CLI) agents, and working with the Unix file system and input/output processes.

Urbit ships are commonly divided into live ships and fakezods (after `~zod`, the parent galaxy). Live ships are end-user instances operating on live `%ames`. Fakezods are disconnected and keyless ships frequently used in development. Most of our work in this book should be completed using fakezods, as one can lobotomize one's personal live ship by committing bad agent code to the Urbit file system.

To create a fakezod, one passes the `urbit` executable the `-F` 'fake keys' flag:

```
$ urbit -F zod
```

As downloading a boot sequence or pill can take some time, we recommend downloading the current pill once, storing it locally, and using it to boot fakezods for a while until it becomes outdated. The URL for the current pill can be found by starting a fakezod, observing the URL indicated in the boot sequence, and aborting the boot sequence before retrieving the pill using a tool like `wget` or a web browser. (Use `Ctrl`+`Z` to abort the process.)

```
$ urbit -F tex
~
urbit 1.0
boot: home is /home/davis68/tex
loom: mapped 2048MB
lite: arvo formula 79925cca
lite: core 59f6958
lite: final state 59f6958
boot: downloading pill https://bootstrap.urbit.org/urbit
    -v1.0.pill

[received keyboard stop signal, exiting]
$ https://bootstrap.urbit.org/urbit-v1.0.pill
```

Once the boot sequence has completed on a new fakezod, one can use the `-B` 'pill file' flag to start the fakezod:

```
$ urbit -F zod -B urbit-v1.0.pill
```

We will discuss the boot sequence in detail in Section ??.

After a few minutes of boot sequence, the Dojo REPL prompt appears.

```
$ urbit -F zod -B urbit-v1.0.pill
~
urbit 1.0
boot: home is /home/davis68/tex
loom: mapped 2048MB
lite: arvo formula 79925cca
lite: core 59f6958
lite: final state 59f6958
boot: loading pill urbit-v1.0.pill
```

```
loom: mapped 2048MB
boot: protected loom
live: logical boot
boot: installed 286 jets
boot: parsing %brass pill
---------------- playback starting ----------------
...
---------------- playback complete ----------------
vere: checking version compatibility
ames: live on 31455 (localhost only)
http: web interface live on http://localhost:8080
http: loopback live on http://localhost:12321
pier (20): live
ames: metamorphosis
~zod:dojo>
```

To verify that the fakezod has loaded correctly, the reader should type some simple primitives:

```
> 1
1
> 'hello mars'
'hello mars'
> "Hello Mars"
"Hello Mars"
```

The Dojo automatically parses inp for validity, so attempting to ty some sequences may fail. This is c fusing at first but soon becomes an dispensable validation of newly min code.

Since an Urbit ship is presumably always-on, shutting down the ship is simply a special case of suspending computation. Press `Ctrl+D` to stop the fakezod from running, or `Ctrl+Z` to force stop. To start the fakezod again, use the `urbit` executable with the pier name (the name of the folder created for the fakezod):

```
$ urbit zod
```

Since the behavior of a ship is determined by its state, and its state is determined by its boot sequence, a live ship or a fakezod will remain as such perpetually (although one can run a live ship disconnected from the network). Unless otherwise specified, we assume all development to take place on a fakezod.

Persistent sessions When running a ship to which one wishes to connect repeatedly without shutdown (the standard case for usage and an occasional case in development), one should employ a tool such as `screen` to persist sessions even when a terminal session is not actively connected.

One way to set up this situation is as follows:

1. Download and install `screen` if it is not already available on your OS platform.
2. Start a new screen session with appropriate name:

```
1 screen -S sampel-palnet
2
```

3. In this terminal session, start the Urbit ship, whether a fakezod or a live ship.

```
urbit sampel-palnet

```

4. Once the ship is running correctly, once you are ready to detach from the session, press `Ctrl`+`A`, then `d` to disconnect.
5. To connect to the session again, use

```
screen -r sampel-palnet

```

Be careful not to start multiple sessions with the same name at the same time or you will need to access your target session via a unique session number as well.

Sessions do not persist past system shutdown.

1.5 Exercises

1. Obtain an Urbit ID and set up the Urbit OS. Use the current installation procedure outlined at `urbit.org`. One does not need to use a hosting service if one prefers to run Urbit on one's own hardware, but maintaining the live ship in an always-connected state will improve the experience.
2. Set up a fakezod for software development. (We further recommend creating a copy of the pier so that a new fakezod can be started quickly in case of catastrophic failure.)

Language Essentials

Nock, A Combinator Language

2

2.1 Primitive rules and the combinator calculus

Objectives

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

1. Interpret Nock code as a combinator calculus.
2. Annotate Nock programs interpretively.

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 with pure functions on numeric quantities.

one combinator calculus SKEW

Nock as 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: the subject, or rather a subject, 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 qua virtual machine 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 relationship as analogous to solving a matrix. Formally, given an equation

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

the solution should be obtained as

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

This is correct, but is quite 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, we use jet-accelerated Nock, a computationally optimized interpreter for the particular machine on which Nock is running, covered in 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]]
```

subject to 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:

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

where $*$ is the generic evaluate operator. Furthermore, true is the integer 0 while false is the integer 1.

distinguish a
ock 5K, from
k Five.

Each rule is referred to by its number written out in prose; e.g., "Nock Three" 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 Zero, 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.

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.

One 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,

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

In the Dojo, you may evaluate this statement using the `.*` "dottar" rune:

```
.*(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 One, Constant Reduction

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

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

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

Nock Two, Evaluate

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

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

Nock Three, Test Cell

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

Nock Four, Increment

Nock Five, 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:

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

where $*$ is the generic evaluate operator.

Nock Six, Conditional Branch

Nock Seven, Compose

Nock Eight, Declare Variable

Nock Nine, Produce Arm of Core

Nock Ten, Replace

Nock Eleven, 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.

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:

```
> !=((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:

```
++  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,

```
[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 Objectives

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

1. Identify Hoon runes and children in both inline and long-form syntax.
2. Trace a short Hoon expression to its final result.
3. Execute Hoon code within a running ship.
4. Produce output as a side effect using the `~&` rune.

3.2 Reading the Runes

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.

The terminology used is often unfamiliar. Sometimes this means that you are dealing with a truly new concept (and overloading an older word like "subroutine" or "function" would obfuscate), and sometimes you are dealing with an internal aspect that doesn't really map well to other systems. The strangeness can be frustrating. The strangeness can make concepts fresh again. You'll encounter both as you move ahead.

Each rune accepts at least one child, except for `!!` "zapzap" (crash). Most runes accept a definite number of children, but a few can accept a variable number; these use `==` or `--` digraphs to indicate termination of the running series.

3.3 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]
```

which in turn is also equivalent to

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:

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

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

Developers balance expressiveness, comprehensibility, and pattern-matching in deciding how lapidary to compose a runic expression. Many extremely common patterns were soon subsumed by a sugar rune.

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 =/(n 4 +(n)))
[%tsfs p=term=%n q=[%sand p=%ud q=4] r=[%dtls p=[%wing
    p=~[%n]]]]
```

While parsing this AST is beyond the text at this point, note significant features like `%tsfs` for `=/` and `%sand` for aura-castable values.

3.4 Nouns

All values in Urbit are nouns, meaning either atoms or cells. An atom is an unsigned integer. A cell is a pair of nouns. Since all values are ultimately integers, we need a way to tell different "kinds" (or applications) of integers apart. Enter auras.

Atoms

Atoms have auras which are tagged types. In other words, an aura is a bit of metadata Hoon attaches to a value which tells Urbit how you intend to use a number. (Of course, ultimately an aura is itself an integer as well!) The default aura for a value is **`@ud`**, unsigned decimal, but of course there are many more. Aura operations are extremely convenient for converting between representations. They are also used to enforce type constraints on atoms in expressions and gates.

For what it may be worth, having integers isn't that different from a other digital machine, built on bin numbers. These all derive ultimat from Gödel numbering as introdu by Gödel in the proof of his fam incompleteness theorems. Urbit ma this about as apparent as C does (`union`, for instance), but it's first-or accessible via the Dojo REPL.

For instance, to a machine there is no fundamental difference between binary 0b1101.1001, decimal 217, and hexadecimal 0x*d*9. A human coder recognizes them as different encoding schemes and associates tacit information with each: an assembler instruction, an integer value, a memory address. Hoon offers two ways of designating values with auras: either directly by the formatting of the number (such as **`0b1101.1001`**) or using the irregular syntax **`` `@` ``**:

```
0b1101.1001
`@ud`0b1101.1001  :: yields 217
`@ux`0b1101.1001  :: yields 0xd9
```

Example 3.4.1 Try the following auras. See if you can figure out how each one is behaving.

```
1 `@ud`0x1001.1111
2 `@ub`0x1001.1111
3 `@ux`0x1001.1111
4 `@p`0x1001.1111
5
6 `@ud`.1
```

```
`@ux`.1
`@ub`.1
`@sb`.1

`@p`0b1111.0000.1111.0000

`@ta`'hello'
`@ud`'hello'
`@ux`'hello'
`@uc`'hello'
`@sb`'hello'
`@rs`'hello'
```

For a full table of auras, see Appendix ??.

The atom/aura system represents all simple data types in Hoon: dates, floating-point numbers, text strings, Bitcoin addresses, and so forth. Each value is represented in least-significant byte (LSB) order; for instance, a text string may be deconstructed as follows:

```
'Urbit'
0b111.0100.0110.1001.0110.0010.0111.0010.0101.0101
0b111.0100 0b110.1001 0b110.0010
0b111.0010 0b101.0101
```
116 105 98 114 85 (ASCII characters)
t i b r U

Note in the above that leading zeroes are always stripped. Since each atom is an integer, there is no way to distinguish 0 from 00 from 000 etc.

In this vein, it's worth mentioning that Dojo automatically parses any typed input and disallows invalid representations. This can lead to confusion until you are accustomed to the type signatures; for instance, try to type `0b0001` into Dojo.

Operators . Hoon has no primitive operators. Instead, aura-specific functions or gates are used to evaluate one or more atoms to produce basic arithmetic results. Gate names are conventionally prefixed with `++` which designates them as arms of a core. (More on this terminology in Section ??.) Some gates operate on any input atom auras, while others enforce strict requirements on the types they will accept. Gates are commonly invoked using a Lisp-like syntax and a reverse-Polish notation (RPN), with the operator first followed by the first and second (and following) operands.

Operation	Function	Example
Addition	`++add`	`(add 1 2)` → `3`
Subtraction	`++sub`	`(sub 4 3)` → `1`
Multiplication	`++mul`	`(mul 5 6)` → `30`
Division	`++div`	`(div 8 2)` → `4`
Modulus/Remainder	`++mod`	`(mod 12 7)` → `5`

Following Nock's lead, Hoon uses loobeans (0 = true) rather than booleans for logical operations. Loobeans are written `%.y` for true, 0, and `%.n` for false, 1.

Operation	Function	Example
Greater than, >	`++gth`	`(gth 5 6)` → `%.n`
Greater than or equal to, ≥	`++gte`	`(gte 5 6)` → `%.n`
Less than, <	`++lth`	`(lth 5 6)` → `%.y`
Less than or equal to, ≤	`++lte`	`(lte 5 6)` → `%.y`
Equals, =	`=`	`=(5 5)` → `%.y`
Logical `AND`, ∧	`&`	`&(%.y %.n)` → `%.n`
Logical `OR`, ∨	`|`	`|(%.y %.n)` → `%.y`
Logical `NOT`, ¬	`!`	`!%.y` → `%.n`

Since all operations are explicitly invoked Lisp-style within nested parentheses, there is no need for explicit operator precedence rules.

$$(a < b) \wedge ((b \geq c) \vee d)$$

```
\&((lth a b) (|((gte b c) d)))
```

The Hoon standard library, largely in `%zuse`, further defines bitwise operations, arithmetic for both integers and floating-point values (half-width, single-precision, double-precision, and quadruple-precision), string operations, and more. These are introduced incidentally as necessary and listed in more detail in Appendix ??.

Cells

Just as all structures in Nock are binary trees, so too with Hoon. This can occasionally lead to some awkward addressing when composing tetchy library code segments that need to interface with many different kinds of gates, but by and large is an extremely helpful discipline of thought.

binary tree format flattened representation/convention

Hoon values are addressed as elements in a binary tree.

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

Finally, the most general mold is `*` which simply matches any noun—and thus anything in Hoon at all.

3.5 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 expands on Nock primarily through the introduction of metadata

Each Hoon rune has an unambiguous mapping to a Nock representation. Furthermore, each rune has a well-defined binary tree structure and produces a similarly well-structured abstract syntax tree (AST). As we systematically introduce runes, we will expand on what this means in each case: for now, let's examine two runes without regard for their role.

1. `|=` "bartis" produces a gate or function. Every gate has the same shape, which means certain assumptions about data access and availability can be made.

```
::  XOR two binary atoms
|=  [a=@ub b=@ub]
`@ub`(mix a b)
```

maps to the Nock code

```
[8 [1 0 0] [1 8 [9 1.494 0 4.095] 9 2 10 [6 [0 28]
    0 29] 0 2] 0 1]
```

This Nock code is fully annotated in Example ??.

2. TODO

We call Hoon's data type specifications molds. Molds are more general than atoms and cells, but these form particular cases. Hoon uses molds as a way of matching Nock tree structures (including Hoon metadata tags such as auras).

3.6 Subject-Oriented Programming

TODO arms legs basics

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

3.7 Key Data Structures

Lists

Lests

Text

Both cords and tapes are casually referred to as strings.

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.

Lists are null are tapes.

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.

```
++  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 (and typically just 8-bit 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 and Derived Structures

Cores

The core is the primary nontrivial data structure of Nock: atoms, cells, cores. A core is defined as a cell of `battery payload`; in the abstract this simply divides the `battery` or code from the `payload` or data. Cores can be thought of as similar to objects in object-oriented programming languages, but possess a completely standard structure which allows for detailed introspection and "hot-swapping" of core elements. Everything in standard Hoon and Arvo that cannot be reduced to an atom or a cell is de facto a core. (Indeed, if one wished to separate code and data in a structure, the only other logical choice one would have available is to flip the order of battery and payload.)

Conventionally, most cores are either produced by the `|%` "barcen" rune or are instances of a more complex form (such as a door). Cores are a "live" type; they are not simply holders of data but are expected to operate on data, just as it is uncommon to see a C++ object which only holds data. (The role of a C `struct` is approximated by the `$%` tagged union.)

Some terminology is in order, to be expanded on subsequently. An arm is a Hoon expression to be evaluated against the arms legs Arms and legs are both limbs. (These must be distinguished from wings, which are resolution paths pointing to a limb.)

Traps

The trap creates the basic looping mechanism in Hoon, a special instance of a core which is capable of concisely recursing into itself. The trap is formally a core with one arm named `$`, and it is either created with `|-` "barhep" (for instant evaluation) or `|.` "bardot" (for deferred evaluation).

In practice, traps are used almost exclusively as recursion points, much like loops in an imperative language. For instance, the following program counts from 5 down to 1, emitting output via `~&` "sigpam" at each iteration, then return `~`.

```
=/  count  5
|-
  ?:  =(count 0)  ~
~&  count
$(count (dec count))
```

The `$()` notation is shorthand for `%=` "centis", which evaluates a wing given a set of changes.

The final line `$(count (dec count))` serves to modify the subject (at `count`) then to pull the `$` arm again. In practice the tree unrolls as follows, with indentation indicating code running "inside" of another rune.

```
=/  count  5
|-
  ?:  =(count 0)  ~
  ~&  count
  =/  count  (dec count)
  ?:  =(count 0)
  TODO think about how to unroll this in a visually
    pleasing manner that isn't
  a total lie
$(count (dec count))
```

recursive

Gates

Similar to how a trap is a core with one arm, a gate is a core with one arm and a sample. This means that new per-invocation data are available in the gate's subject.

A gate can recurse into itself like a trap when necessary. In this example, the gate accepts a value `n` (the gate's `spec`) and applies an arm (implicitly named `$`) which carries out the calculation. (Say it with me: `sample`/`payload`.) When the `$()` recursion point is reached, the entire `$` arm is re-evaluated with the specified change of $n \longrightarrow n - 1$.

Listing 3.1: Calculating a factorial using recursion. (Example from Tlon documentation.)

```
1 |=  n=@ud              :: accept a single value n for n!
2 ?:  =(n 1)  1          :: check n = 1; if so, return 1
3 (mul n $(n (dec n)))   :: multiply n times the product of
      this arm w/ n-1
```

That is, when one reads this program, one reads it falling into two components:

```
|=  n=@ud              :: accept a single value n for n!
?:  =(n 1)  1          :: check n = 1; if so, return 1
(mul n $(n (dec n)))   :: multiply n times the product of
     this arm w/ n-1
```

Doors

|_

The `|%` "barcen" rune produces a dry core, thus all arms it contains are dry.

The `|` "barpat" rune produces a wet core, so it can contain wet arms.

Doors are superficially similar to gates, but calling an arm in a door is more general: the calling convention for a gate replaces the sample

then pulls the `$` arm, while the calling convention for a door replaces the sample then pulls whichever arm has been requested.

and other cores frequently refers to the subject TODO:expand, this yields the ability to custom type definitions; these manipulate the sample without calling any arm directly. The following scussed below in Section ??. examples illustrate:

```
> (add [1 5]})                :: call a gate
6
> ~($ add [1 5])              :: call the $ arm in the
    door
6
> ~(. add [1 5])              :: update the sample (
    arguments) but don't evaluate
<1.otf [[@ud @ud] <45.xig 1.pnw %140>]>
> =<  $  ~(. add [1 5])       :: call the $ arm of the
    door with updated sample
6
> =/  addd  ~(. add [1 5])    :: do the same thing but in
    two steps
  =<  $  addd
6
```

At this point, we need to step back and contextualize the power afforded by the use of cores. In another language, such as C or Python, we specify a behavior but have relatively little insight into the instantiation effected by the compiler:

```
1 metals = ['gold', 'iron', 'lead', 'zinc']
2 total = 0
3 for metal in metals:
4     total = total + len(metal)
```

Listing 3.2: Python loop to sum the length of several strings.

```
1  1           0 LOAD_CONST               0 ('gold')
2              2 LOAD_CONST               1 ('iron')
3              4 LOAD_CONST               2 ('lead')
4              6 LOAD_CONST               3 ('zinc')
5              8 BUILD_LIST               4
6             10 STORE_NAME               0 (metals)
7
8  2          12 LOAD_CONST               4 (0)
9             14 STORE_NAME               1 (total)
10
11 3          16 LOAD_NAME                0 (metals)
12            18 GET_ITER
13       >>   20 FOR_ITER                16 (to 38)
14            22 STORE_NAME               2 (metal)
15
16 4          24 LOAD_NAME                1 (total)
17            26 LOAD_NAME                3 (len)
18            28 LOAD_NAME                2 (metal)
19            30 CALL_FUNCTION            1
20            32 BINARY_ADD
21            34 STORE_NAME               1 (total)
22            36 JUMP_ABSOLUTE           20
23       >>   38 LOAD_CONST               5 (None)
24            40 RETURN_VALUE
```

Listing 3.3: Python bytecode equ lent

Many popular programming languages specify behavior rather than

implementation. By specifying Hoon as a macro language over Nock, the Urbit developers collapse this distinction.

For Hoon (like Lisp), the structure of execution is always front-and-center, or at least only thinly disguised. When one creates a core with a given behavior, one can immediately envision the shape of the underlying Nock. This affords one immense power to craft efficient, effective, graceful programs.

Listing 3.4: Hoon trap to sum the length of several tapes.

```
=/  metals  `(list tape)`~["gold" "iron" "lead" "zinc"]
=/  count  0
=/  total  0
|-
  ?:  =(count (lent metals))  total
  =/  metal  `tape`(snag count metals)
$(total (add total (lent metal)), count +(count))
```

Listing 3.5: Nock equivalent

Molds

Most software programs require—or at least are greatly simplified by—custom type definitions. These are customarily defined with a `|%` core preceding the `|_` door definition. We commonly see three runes supporting this structure:

- `+$` "lusbuc" creates a type constructor arm to define and validate type definitions.
- `$%` "buccen" creates a collection of named values (type members).
- `$?` "bucwut" defines a union, a set validating membership across a defined collection of items. (This is similar to a `typedef` or `enum` in C-related languages.)

To illustrate these, we serially consider several ways to define a vehicle. In the first, we employ only `+$` to capture key vehicle characteristics. Using only `+$`, it's hard to say much of interest:

```
+$  vehicle  tape                :: vehicle
    identification number
```

By permitting collections of named type values with `$%`, we can produce more complicated structures:

```
+$  vehicle
  $%  vin=tape                   :: vehicle
    identification number
      owner=tape                 :: car owner's name
      license=tape               :: license plate
  ==
```

Type definition arms can rely on other type definition arms available in the subject:

```
+$  vehicle
  $%  vin=tape                   :: vehicle
    identification number
      owner=tape                 :: car owner's name
      license=tape               :: license plate
```

```
      kind=kind                   :: vehicle manufacture
    details
  ==
+$  kind
  $%  make=tape                   :: vehicle make
      model=tape                  :: vehicle model
      year=@da                    :: nominal year of
    manufacture (use Jan 1)
  ==
```

Finally, by introducing unions with `$?`, a type definition arm can validate possible values:

```
+$  vehicle
  $%  vin=tape                    :: vehicle
    identification number
      owner=tape                  :: car owner's name
      license=tape                :: license plate
      kind=kind                   :: vehicle manufacture
    details
  ==
+$  kind
  $%  make=tape                   :: vehicle make
      model=tape                  :: vehicle model
      year=@da                    :: nominal year of
    manufacture (use Jan 1)
  ==
+$  make                          :: permitted vehicle
    makes
  ?(%acura %chrysler %delorean %dodge %jeep %tesla %
    toyota)
```

In general, Hoon style does require you to be careful ab masking variable names in subject (using the same name the value as the mold). This ra introduces surprising bugs but typically contextually apparent to developer.

`@tas`-tagged text elements are extremely common in such type unions, as they afford a distinguishable human-readable categorization that is nonetheless rigorous to the machine.

Data Structures

Units

Every atom in Hoon is an unsigned integer, even if interpreted by an aura. Auras do not carry prescriptive power, however, so they cannot be used to fundamentally distinguish a `NULL`-style or `NaN`-style non-result. That is, if one receives a `~` back from a query, how does one distinguish a value of zero from a non-result (missing value)?

Units mitigate this situation by acting as a type union of `~` (for no result) and a cell `[~ u=item]` containing the returned item with face `u`.

```
++  unit
  |$  [item]
  $@(~ [~ u=item])
```

Maps

Sets

Trees

Dimes

3.8 Generators

Generators are standalone Hoon expressions that evaluate and may produce side effects, as appropriate. They are closely analogous to simple scripts in languages such as Bash or Python. By using generators, one is able to develop more involved Hoon code and run it repeatedly without awkwardness.

To run a generator on a ship, prefix its name with `+`. Arguments may be required or optional.

You may also see commands beginning with a `|` symbol; these are Hood commands instead.

```
+moon TODO
```

Running Developer Code on an Urbit Ship

Since the Urbit file system, called `%clay`, is independent of the Unix file system on which it is hosted, you must commit your Unix-side code into your pier.

If we `cd` into the ship's pier in Unix and `ls` the directory contents, by default we see nothing. With `ls -l`, a `.urb/` directory containing the ship's configuration and contents in obfuscated format becomes visible. This directory is not interpretable by us now, so we leave it until a later discussion of the Urbit binary. To move files into `%clay`, we must synchronize Urbit and Unix together. We initiate this inside of the running ship; run the Urbit system command:

```
|mount %
```

where `%` represents the current (home) path in `%clay`. Unix-side, run `ls` again and a `home/` directory appears with a number of children: `app/`, `gen/`, `lib/`, `mar/`, and so forth. This is the internal esoteric structure of `%clay` made manifest to Unix.

`%clay` implements several desks, which are like branches in version control systems; the most important of these are `%home` and `%kids`.

Generally speaking, we compose generators, which are short Hoon scripts. These are created in or copied into the `home/mar/` directory, and then must be synchronized with Urbit's `%clay`. Commit the change:

```
|commit %home
```

and the generator is now available within `%clay`.

Naked Generators

As we start to compose generators,

A naked generator is so called because it contains no metadata for the Arvo interpreter.

`%say` generators

— ‘++dicethrow‘

Write a `%say` generator which simulates scoring a simple dice throw of n six-sided dice. That is, it should return the sum of n dice as inputs. You may reuse code from ‘mp0‘ or you may use entropy following the discussion in []. If no number is specified, then only one die roll should be returned.

Since Hoon is functional but random number generators stateful, you should use the `=^` “tisket” rune to replace the current value in the RNG. `=^` is a kind of ”one-effect monad,” which allows you to change a single part of the subject.

For instance, here is a generator that returns a list of probabilities from 1–100

```
:-  %say
|=  [[* eny=@uv *] [n=@ud ~] ~]
  :-  %noun
  =/  values  `(list @ud)`~
  =/  count  0
  =/  rng  ~(. og eny)
  |-  ^-  (list @ud)
    ?:  =(count n)  values
    =^  r  rng  (rads:rng 100)
  $(count +(count), values (weld values ~[(add r 1)]))
```

Your command to run this generator in the Dojo should look like this:

“‘hoon +dicethrow, =n 5 “‘

(Note the comma separating optional arguments.)

—

%ask generators

3.9 Libraries

3.10 Unit Tests

3.11 Building Code

3.12 Exercises

The vertical direction presents a six-step process that prompts students to read the problem statement, figure out the data that is needed to represent the information of interest, and illustrate their insight with concrete examples;

articulate a purpose statement that concisely describes what the function or program is supposed to compute, including a signature;

work through functional examples, that is, explain what the function or program is supposed to produce when given certain inputs, based on steps 1 and 2;

create an outline of the program, based on steps 1 and 2;

fill in the outline from step 4, using steps 2 and 3; and

turn the examples from step 2 into a test suite for the program from step 5.

Advanced Hoon 4

4.1 Objectives

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

1. Identify Hoon runes and children in both inline and long-form syntax.
2. Trace a short Hoon expression to its final result.
3. Execute Hoon code within a running ship.
4. Produce output as a side effect using the `~&` rune.

4.2 Cores

Type Definition

Cores are defined in Hoon as

```
[$core p=type q=coil]
+$  coil
  $:
    p=[p=(unit term) q=?(%wet %dry) r=?($gold $iron
    $lead $zinc)]
    q=type
    r=[p=seminoun q=(map term (pair what (map term hoon)
    ))]
```

where a `+$type` is any or zero nouns. (The rest we will discuss in this section.)

Variadicity

Variadicity describes how an object TODO

Cores can hew to a variety of variadic models:

1. **`%gold`** cores are covariant, meaning that they allow universal reading from and writing to the core. **`%gold`** cores are ubiquituous as they are the default core variadic model.
2. **`%iron`** cores
3. **`%lead`** cores
4. **`%zinc`** cores are rarely used in practice, but complete the system of variadic relationships.

> 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

Genericity

TODO Hoon koan about boiling tea

Similar to how cores vary by variadicity (the metal type), some cores (gates) can also differ by genericity or their type analysis behavior. Gates can be classified as either dry or wet:

1. Dry gates perform type analysis at the call site. When arguments are passed to the gate, the compiler attempts to cast each argument to the type specified by the gate. In a sense, a strong typing system is enforced, although nesting rules apply (e.g.,, `@ud` nests in `@u`).
   The `|=` "bartis" rune produces a dry gate as a pair of a specification and a resulting value: that is, a cell containing the arguments and the operative binary tree (`battery` specifying values, `payload` specifying operations).
2. Wet gates perform type analysis at the gate definition. When arguments are passed to the gate, the argument types are preserved and the compiler carries out type analysis at the definition of the gate. This allows wet gates to be much more flexible (and correspondingly dangerous!) in parsing arbitrary inputs.
   The `|*` rune produces wet gates
   Wet arms and wet cores behave a bit like C++ overloaded operators and templates or Haskell typeclasses. However, there is no multiple dispatch (as with C++) so there's not a way to, e.g.,, just make '++add' and '++add:rs' compatible.)

Wet gates can be thought of as C-style macros while dry gates are like C-style functions.

Hearken back to the type definition of a core.

```
1 [$core p=type q=coil]
2 +$  coil
3   $:
4     p=[p=(unit term) q=?(%wet %dry) r=?($gold $iron
      $lead $zinc)]
5     q=type
6     r=[p=seminoun q=(map term (pair what (map term hoon)
      ))]
```

Here one can see that

> The full core stores both payload types. The type that describes the payload currently in the core is `p`. The type that describes the payload the core was compiled with is `q.q`. (Documentation, "Cores")

TODO more on doors etc.

4.3 Molds

As we saw when discussing auras, molds are the most general category of type in Hoon.

soft atoms, seeing atoms, etc.
)

Polymorphism

4.4 Rune Families

These runes are up-to-date as of Hoon `%140`. For an exhaustive list of runes, please check the appendix.

Runes play the role of structural keywords in Hoon. Somewhat conveniently, runes are classified into semantic families by their first character. (The second character rarely carries specific information, and only occasionally do "opposite" characters like `+` and `-` correspond.) When reading or composing Hoon code, the ability to identify runes by family can quickly help you structure a program.

It can be helpful to think of runes in two ways: as branch points in binary trees or as modifiers and thence yielders of new subjects or expressions. (Almost all runes except `!!` "zapzap" and the `~` "sig" runes can be considered in both ways.)

Below, we discuss the most significant runes in each family and give examples of their usage. In subsequent discussion, you should refer back to this discussion in order to interpret kernel and userspace code containing unfamiliar runes. We defer a comprehensive catalogue of runes to Appendix ??.

`|` "bar": Core Definition

`|` "bar" runes produce cores.

The `|-` rune expands to an evaluated `|.` trap and thence to a one-armed `|%` core with `$` arm.

```
=<  $  |.  a=hoon
=<  $  |%  ++  $  a=hoon
```

`§` "buc": Mold Definition

`$` "buc" runes produce mold definitions

`%` "cen": Core Evaluation

`:` "col": Cell Construction

`.` "dot": Nock Evaluation

`.` "dot" runes allow direct evaluation of certain Nock expressions. These are variously useful for direct operations such as atom increment, fast equality checking, and cell/atom differentiation.

.^ “dotket”

Although Nock itself has twelve rules denominated from Zero to Eleven, a fake Nock Twelve rules allows Arvo to scry or request out-of-subject information directly.

Each vane has its own unique set of scry types; a few of these are briefly demonstrated here but scrys discussed in depth in the section on each vane.

Most scrys are requests to `%clay` or `%gall` for state information, such as subscription data.

master-morzod:

from the inside out: - a (working) **.^** means you’re being virtualized (+mock) - **.^** (”mock” opcode %12) simply calls the ”scry-handler” gate provided to the virtual nock interpreter - so **.^** is just a way to read state from your caller - arvo implements a namespace over its state (+peek:le) - significant portions of that namespace are deferred to vanes (‘+scry) - to read arvo’s namespace from outside: arvo’s external +peek arm - to read from within arvo proper: +peek:le - to read from within a vane, call the provided $roof - to wire up arvo’s namespace to **.^**, invoke +mock with (look roof) as the scry handler

.+ “dotlus”

Irregular form: `+(a)`

.* “dottar”

.= “dottis”

Irregular form: `=(a1 a2)`

.? “dotwut”

^: Core Typecasting

~ “sig”: Hinting

Compare `#pragma` expressions in C-family languages.

~ “sig” runes represent directives to the runtime. Nock Eleven provides a way for the

; “mic”: Macro

; runes are mainly utilized to produce calling structures (mainly monadic binds) and XML elements.

`=` “tis”: Subject Alteration

`=` runes modify the subject and yield a new subject with the specified changes.

`?` “wut”: Comparison

`!` “zap”: Wildcard

Like `;` runes, `!` “zap” runes constitute a miscellany of effects.

4.5 Marks and Structures

4.6 Helpful Tools

”so =- is inverted =+ so the second part of the expression actually executes before the first part”

See also Section ?? which discus
common ”factory patterns” in subj
oriented programming.

```
=-  (~(put by some-map) key -)
... really long product that goes into the map
```

Example 4.6.1 Defining a Rune

It is fairly straightforward to produce a sugar rune.

We proceed analogically from `|?` “barwut”, a sugar rune which converts a core to a lead core (bivariant). We will produce `|#` “barhax” which will convert a core to a zinc trap (covariant). (We punt on the question of why one would want a zinc trap, which may explain why such a rune does not exist.)

Any new rune should not be reduced by the parser to

4.7 Deep Dives

Text Stream Parsing

Text parsing in Urbit is largely synonymous with the Hoon parser itself. `%zuse` offers built-in support for some structured text data, notably JSON and HTML/XML, which are discussed in subsequent sections. Other text is parsed using a `++rule`.

“‘py [(lent ”foo”) (met 3 ’foo’) (lent ” ”) (lent (tuba ” ”))] “‘

rules, nails, etc.

More information can be found in the `%shoe` tutorial TODO.

JSON Parsing

JavaScript Object Notation (JSON) data structures have become a lingua franca of the modern Web. More compact than XML and related languages, natively parsed by Javascript, Python, and several other languages, and readily human-readable, JSON data are provided and processed by many APIs.

HTML/XML Parsing

System Development

The Kernel 5

The system-level instruction set of Arvo is described in Chapter ??.

5.1 Objectives

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

1. Diagram the structure of Arvo as stateful event processor.
2. Follow a move trace to see how an event is processed by Arvo.
3. Enumerate the primary vanes of Arvo and their roles.
4. Scry into each vane for run-time information.
5. Query and manipulate ship keys from Azimuth and `%jael`.

5.2 Arvo

We could do worse than to start our exploration of the Urbit kernel than to quote the Whitepaper itself on the subject of Arvo:

> The fundamental unit of the Urbit kernel is an event called a `+$move`. Arvo is primarily an event dispatcher between moves created by vanes. Each move therefore has an address and other structured information. ([Yarvin2017])

Every vane defines its own structured events (`+$move`s). Each unique kind of structured event has a unique, frequently whimsical, name. This can make it challenging to get used to how a particular vane behaves.

Arvo is essentially an event handler which can coordinate and dispatch messages between vanes as well as emit `%unix` events (side effects) to the underlying (presumed Unix-compatible) host OS. Arvo as hosted OS does not carry out any 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, the daemon processes which together run Arvo.

Arvo is architected as a state machine, the deterministic end result of the event log. We need to examine Arvo from two separate angles:

1. Event processing engine and state machine (vane coordinator)
2. Standard noun structure ("Arvo-shaped noun")

Event Processing Engine

A vanilla event loop scales poorly in complexity. A system event is the trigger for a cascade of internal events; each event can schedule any number of future events. This easily degenerates into "event spaghetti." Arvo has "structured

> events"; it imposes a stack discipline on event causality, much like imposing subroutine structure on `goto`s. ([Yarvin2017])

Arvo events are known as `++move`s, containing metadata and data. Arvo recognizes four types of moves:

1. `%pass` events are forward calls from one vane to another (or back to itself, occasionally), and
2. `%give` events are returned values and move back along the calling duct.
3. `%slip` events [TODO think of like ref counting?]. (`%slip` events are common only in the initial system boot process or in over-the-air updates to Arvo.)
4. `%unix` events communicate from Arvo to the underlying binary in such a way as to emit an external effect (an `%ames` network communication, for instance, or text input and output).

show structure of a card

"Each vane defines a protocol for interacting with other vanes (via Arvo) by defining four types of cards: tasks, gifts, notes, and signs." "In other words, there are only four ways of seeing a move: (1) as a request seen by the caller, which is a note. (2) that same request as seen by the callee, a task. (3) the response to that first request as seen by the callee, a gift. (4) the response to the first request as seen by the caller, a sign." (TODO move trace work here)

Without reference to the particular content of vanes, let us briefly diagram a "move trace", or examination of how an event generated by a vane produces results via Arvo.

TODO remote call via `%ames`?

> "An interrupted event never happened. The computer is deterministic; an event is a transaction; the event log is a log of successful transactions. In a sense, replaying this log is not Turing complete. The log is an existence proof that every event within it terminates. ([Yarvin2017])

Example 5.2.1 Tradecraft: Tracing Now that we have discussed how Arvo processes events, we can turn our attention to how particular real events are actually processed.

Urbit supports tracing the behavior of each event using a command-line flag, conventionally `-j`. Start an existing Urbit ship using

```
$ urbit -j sampel-palnet
```

Initiate an event:

```
> (add 1 2)
```

Tracefiles will appear in `sampel-palnet/.urb/put/trace`. Copy these JSON files to a working directory and open them using an appropriate utility. There may be some lag in generating the tracefiles, and it requires discernment to locate the relevant move trace elements.

TODO

Example 5.2.2 Tradecraft: Profiling Urbit supports profiling the performance of event handling using a command-line flag, conventionally `-P`. Start an existing Urbit ship using

```
$ urbit -P sampel-palnet
```

Initiate an event:

```
> (add 1 2)
```

Profiling files will appear in `sampel-palnet/.urb/put/profile`. Copy these text files to a working directory and open them using an appropriate utility.

TODO

Standard Noun Structure

Arvo defines five standard arms for vanes and the binary runtime to use:

1. `++peek` grants read-only access to `%clay`; this is called a scry.
2. `++poke` accepts `++move`s and processes them; this is the only arm that actually alters Arvo's state.
3. `++wish` accepts a core and parses it against `%zuse`.
4. `++come` and `++load` are used in kernel upgrades, allowing Arvo to update itself in-place.

https://urbit.org/docs/arvo/overview/

`%zuse` and **`%lull`**

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

subject wrapped

5.3 Arvo Vanes

Each vane has a characteristic structure which identifies it as a vane to Arvo and allows it to handle moves consistently.

5.4 **`%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

Scrying into **%ames**

`%ames` scry

Table 5.1: `%ames .^` Calls.

Symbol	Meaning	Example
`%x`	Get ship and peer information: protocol version, peers, ship state, etc.	

5.5 **%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=~)
```

Scrying into **%behn**

Table 5.2: `%behn .^` Calls.

Symbol	Meaning	Example
`%x`	Get timers, timestamps, next timer to fire, etc.	

5.6 **%clay**, A File System

https://github.com/davis68/martian-computing/blob/78dce6435f645f2135f09e228062a1371cf2ef9d/lessons/lesson22-clay-2.md

“‘ tinnus-napbus 3:57 PM what is the correct way to read a file on a remote ship? I’ve tried both warp and werp and I’m not getting a

response, just messages in the target about clay something something indirect and then I get crash on fragment errors sometimes with an rovnys-ricfer 4:02 PM warp should work, I think make sure the file is actually there i.e. can you do the same warp on the local ship? “‘

- `aeon` is
- `arch` is
- `care` is the `%clay` submode, defined in `%lull`.
- `desk` is
- `dome` is mark cast file

Data Model

paths

merges

marks

Scrying into `%clay`

`%clay` has the most sophisticated scry taxonomy.

Table 5.3: `%clay .^` Calls.

Symbol	Meaning	Example
`%a`	Expose file build into namespace.	`.^(vase %`
`%b`	Expose mark build into namespace.	
`%c`	Expose cast build into namespace.	
`%d`	diff; ask clay to validate .noun as .mark	
`%e`		
`%f`		
`%p`	Get the permissions that apply at path.	
`%r`	Get the data at a node (like `%x`) and wrap it in a vase.	
`%s`	produce yaki or blob for given tako or lobe	
`%t`	produce the list of paths within a yaki with :pax as prefix	
`%u`	Check for existence of a node at aeon.	
`%v`	Get the desk state at a specified aeon.	
`%w`	Get all cases referring to the same revision as the given case.	
`%x`	Get the data at a node.	
`%y`	Get the `arch` (directory listing) at a node.	
`%z`	Get a recursive hash of a node and its children.	

`++ford`, A Build System

runes

Table 5.4: `++ford` Runes.

Symbol	Meaning	Example
`/-`	Import structure file from `sur`.	
`/+`	Import library file from `lib`.	
`/=`	Import user-specified file.	
`/*`	Import contents of file converted by mark.	

importing with * is w/o face, foo=bar

Marks and conversions

Exercises

CSV Conversion Mark Compose a mark capable of conversion from a CSV file to a plain-text file (and vice versa).

The ++grad arm can be copied from the hoon mark, since we are not concerned with preserving CSV integrity.

Conversion Tube Use ++ford to produce a tube from hoon to txt.

Answer this question with the expression.

5.7 %dill, A Terminal driver

Scrying into %dill

%dill scrys are unusual, in that they are typically only necessary for fine-grained Arvo control of the display. Even command-line apps instrumented with %shoe do not call into %dill commonly. The only instance of use in the current Arvo kernel is in Herm, the terminal session manager.

Table 5.5: %dill .^ Calls.

Symbol	Meaning	Example
%x	Get the current line or cursor position of default session.	

5.8 %eyre and %iris, Server and Client Vanes

Scrying into %eyre

%eyre

Table 5.6: %jael .^ Calls.

Symbol	Meaning	Example
%x	Get CORS etc TODO.	
%$	Get .	

Scrying into %iris

Table 5.7: %iris .^ Calls.

Symbol	Meaning	Example
%$	Get .	

5.9 **%jael**, Secretkeeper

%jael keeps secrets, the cryptographic keys that make it possible to securely control your Urbit. Among other cryptographic facts, **%jael** keeps track of the following:

1. Subscription to **%azimuth-tracker**, the current state of the Azimuth PKI.
2. Initial public and private keys for the ship.
3. Public keys of all galaxies.
4. Record of Ethereum registration for Azimuth.

%jael weighs in as one of the shorter vanes, but is critical to Urbit as a secure network-first operating system. **%jael** is in fact the first vane loaded after **%dill** when bootstrapping Arvo on a new instance.

%jaelis name
Heber, who k
enemy gener
also puns on

---

record the Ethereum block the public key is registered to, record the URL of the Ethereum node used, save the signature of the parent planet (if the ship is a moon), load the initial public and private keys for the ship, set the DNS suffix(es) used by the network (currently just arvo.network), save the public keys of all galaxies, set Jael to subscribe to

master-morzod 1:00 PM the private key (ring) is 64 bytes, plus a tag byte 1:00 (met 3 sec:ex:(pit:nu:crub:crypto 512 eny)) 65 the ship is up to 64 bits (or 128, if you want to support comet keypairs, which you could)

```
1 =/  ,seed:jael  [who=`@p`0 lyf=1 sek=(bex 520) ~]  (met
      3 (scot %uw (jam -)))
2 =/  ,seed:jael  [who=`@p`(bex 127) lyf=(bex 31) sek=(bex
       520) ~]  (met 3 (scot %uw (jam -)))
```

likely lower and upper bounds

%jael API Reference

Scrying into **%jael**

Table 5.8: **%jael** ^Calls

Symbol	Meaning	Example
%$	Get TODO.	
%$	Get .	

Sometimes residual elements of vane development leak into release code and yield insight into how the kernel developers produce new vanes. For instance, when a new version of **%jael** was being developed, it was dubbed **kale** and used the scry path **%k**. This made it into a release version of **lib/ring** in Arvo 309 K.

Launching a Moon

It is instructive, as a brief examination of how **%jael** works, to see how to launch and operate a moon (a 64-bit identity tied to a parent planet). We begin our enquiry with an examination of Hood's **moon** generator. This generator optionally accepts a particular moon name. When run, **|moon** returns a private key appropriate to registering a new moon ship with the planet's **%jael**. This key is then used to launch the moon ship using **%jael**to authenticate against the planet.

```
~sampel-palnet:dojo> |moon ~fosfyr-dinler-sampel-palnet
moon: ~fosfyr-dinler-sampel-palnet
0wayzmv.OyFW6.FL6bt.GYKeZ.tPPfc.vJ7av.nFvA0.210dK.iAHiq.
    cSOGh.MCTuz.RiDIV.A3Pd8.pPZcV.djykA.kzrxV.P6M0s.~
    oZge.Nsjpr.K-WAh.QoUOP
```

Let's take a look at |moon:

Listing 5.1: gen/hood/moon.hoon, T

```
::  Create a private key-file for a random (or specified
    ) moon
::
::::  /hoon/moon/hood/gen
  ::
/-  *sole
/+  *generators
::
::::
  ::
:-  %say
|=  $:  [now=@da eny=@uvJ bec=beak]
        arg=?(~ [mon=@p ~])
        public-key=pass
    ==
:-  %helm-moon
^-  (unit [=ship =udiff:point:jael])
=*  our  p.bec
=/  ran  (clan:title our)
?:  ?=([?(%earl %pawn)] ran)
  %-  %-  slog  :_  ~
      leaf+"can't create a moon from a {?:(?=(%earl ran)
     "moon" "comet")}"
  ~
=/  mon=ship
  ?^  arg
    mon.arg
  (add our (lsh 5 (end 5 (shaz eny))))
=/  seg=ship  (sein:title our now mon)
?.  =(our seg)
  %-  %-  slog  :_  ~
      :-  %leaf
      "can't create keys for {(scow %p mon)}, which
    belongs to {(scow %p seg)}"
  ~
=/  =pass
  ?.  =(*pass public-key)
    public-key
  =/  cub  (pit:nu:crub:crypto 512 (shaz (jam mon life=1
     eny)))
  =/  =seed:jael
    [mon 1 sec:ex:cub ~]
  %-  %-  slog
      :~  leaf+"moon: {(scow %p mon)}"
          leaf+(scow %uw (jam seed))
      ==
  pub:ex:cub
`[mon *id:block:jael %keys 1 1 pass]
```

TODO

The `%helm-moon` tag tells Helm to invoke its `++poke-moon` arm and cycle the keys of the moon (or create them).

Listing 5.2: `lib/hood/helm.hoon`, line 211

```
%helm-moon              =;(f (f !<(_+<.f vase)) poke-moon
    )
```

Listing 5.3: `lib/hood/helm.hoon`, lines 75–80

```
++  poke-moon                                          ::
     rotate moon keys
  |=  sed=(unit [=ship =udiff:point:jael])
  =<  abet
  ?~  sed
    this
  (emit %pass / %arvo %j %moon u.sed)
```

where `++emit` is the Helm event processor:

Listing 5.4: `lib/hood/he` 21–24, 29

```
|=  [=bowl:gall sat=state]
=|  moz=(list card)
|%
++  this  .
++  emit  |=(card this(moz [+< moz]))
```

When a new moon is launched as a ship, as part of the Arvo bootstrapping sequence the moon's status as a daughter point of the given planet is checked. First, whether the sponsoring point is validly a moon (as `%earl`); then whether said moon can in fact be validated as belonging to a planet we control (`sein:title` checks for sponsorship). If both of these checks pass, then the private-key update event is issued using the `++new-event` arm of `%jael`. The most salient part of this code, from a cryptographic standpoint, is `udiff`, a non-invertible `diff` or the new public key.

Azimuth points underwent several shifts in nomenclature. Although currently organized into the intuitive hierarchical galaxy/star/planet/moon/comet syntax, the original scheme involved `%duke/%earl/%lord/%pawn/%wolf`. Sometimes this terminology remains visible in internal kernel types such as `%earl` for moon; this is from the `%czar/%king/%duke/%earl/%pawn` phase, which itself predates the carrier/cruiser/destroyer/yacht/sub

Note the … call form … arm in a door, last line … snippet. `new-event` is a… (part of `%jael`) invoked al… duct `hen`, the time `now`, … public-key state `pki`, and … node `etn`.

```
::
::  update private keys
::

  %moon
?.  =(%earl (clan:title ship.tac))
  ~&  [%not-moon ship.tac]
  +>.$
?.  =(our (^sein:title ship.tac))
  ~&  [%not-our-moon ship.tac]
  +>.$
%-  curd  =<  abet
(~(new-event su hen now pki etn) [ship udiff]~:tac)
```

As for the rest—please see Section ?? for more information on the boot sequence of a new ship.

A moon can scry into `%jael`

```
::
  %earl
?.  ?=([@ @ ~] tyl)  [~ ~]
?.  =([%& our] why)
  [~ ~]
=/  who  (slaw %p i.tyl)
=/  lyf  (slaw %ud i.t.tyl)
?~  who  [~ ~]
?~  lyf  [~ ~]
```

```
?:  (gth u.lyf lyf.own.pki.lex)
  ~
?:  (lth u.lyf lyf.own.pki.lex)
  [~ ~]
:: XX check that who/lyf hasn't been booted
::
=/  sec  (~(got by jaw.own.pki.lex) u.lyf)
=/  moon-sec  (shaf %earl (sham our u.lyf sec u.who))
=/  cub  (pit:nu:crub:crypto 128 moon-sec)
=/  =seed  [u.who 1 sec:ex:cub ~]
``[%seed !>(seed)]
```

(Now that you have a moon, what can you do with it?)

5.10 Azimuth, Address Space Management

Urbit HD wallet

Comet keys . Comets do not have an associated Urbit HD wallet, and their keys work slightly differently.

> yeah thats how comet mining works. so you'd just put the private key you generated for a comet on the card, and this would be the ames DH exchange private key. i suppose you could still obfuscate it with a master ticket @q, by just picking a 128 bit hash, but it would be used differently than a normal azimuth master key, which is a @q used to derive a bunch of ethereum wallets private keys (and ultimately the initial network key, but that isnt required). 9:15 >and yeah whether a key works at the time it is mined is dependent on whether the routing node automatically assigned to the comet public key is currently working > lagrev-nocfep: the comets name is its public key

5.11 The Hoon Parser

Userspace 6

Modern Gall apps 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

Userspace applications are conceptually (but not architecturally) divisible into three categories. It is recommended as a burgeoning best practice that new apps hew to this division for clarity and forward compatibility.

1. Stores. Independent local collections of data exposed to other apps. Think of Groups for Landscape.
2. Hooks. Store manipulations. Think of subscriptions.
3. Views. User-interface applications. Think of Landscape itself (the browser interface).

Logan Allen, Matt, Matilde Park ‘ haddef-sigwen‘, ”Userspace Architecture”

All userspace apps are mediated by `%gall`, which provides a standard set of operations to interface with userspace applications. `%gall`'s domain essentially consists of user-visible state machines. Almost everything you interact with as a user is mediated through `%gall`: Chat, Publish, Dojo, and so forth.

Every complex structure in Hoon is a core; a `%gall` agent structurally requires ten arms in its main core with an optional helper core. The agent itself is a door with two components in its subject:

1. `bowl:gall` defined `%gall`-standard tools and data structures.
2. Agent state information (with version number)

A `%gall` agent can use default arms provided by the `default-agent` library in case they are not needed. For instance, a minimalist `%gall` agent looks like this:

```
/+  default-agent
^-  agent:gall
=|  state=@  :: TODO still valid?
|_  =bowl:gall
+*  this      .
    default   ~(. (default-agent this %|) bowl)
::
++  on-init   on-init:default
++  on-save   on-save:default
++  on-load   on-load:default
++  on-poke
  |=  [=mark =vase]
  ~&  >  state=state
  ~&  got-poked-with-data=mark
  =.  state  +(state)
  `this
::
```

Table 6.1: `%gall` ^ Calls

Symbol	Meaning	Example

```
%e
%x      ; returns a vase compatible with the mark at the end of the scry request.
%y      ; returns a cage with a mark of %arch and a vase of %arch.
```

```
++  on-watch  on-watch:default
++  on-leave  on-leave:default
++  on-peek   on-peek:default
++  on-agent  on-agent:default
++  on-arvo   on-arvo:default
++  on-fail   on-fail:default
```

For `%gall`, a standard move is a pair `[bone card]`. A `bone` is a cause while a `card` is an action. (`[cause action]`, it's always `[cause action]`.) A `card`, in turn, is a pair `[tag noun]`. The tag is a `@tas`token to indicate which event is being triggered while the `noun` contains any necessary information.

Compare this to other core structures: a gate is '[battery [sample context]]'; a move is '[bone [tag noun]]'.

`%gall` apps run like services, so they can always talk to each other. This leads to a very high potential for app interactivity rather than siloization.

Scrying into `%gall`

`%gall` possesses several scry types, although not as rich as `%clay`:

Structures, Patterns, and Factories

Structures define common patterns of interaction which are shared across system components and agents. A structure file lives in `/sur`, and is generally distinguished from being a `/lib` library file in that it only presents molds as type definitions, whereas a library contains operable code as gates and doors and the like. (This is by convention not force.) Structures are imported using the `/+` "faslus" rune. For instance, consider the `/sur/file-server/hoon`, which defines `action`s, `configuration`s, and `update`s for the `%file-server` `%gall` app.

Listing 6.1: `/sur/file-server/hoon`

```
/-  glob
|%
+$  action
  $%  [%serve-dir url-base=path clay-base=path public=?
    spa=?]
      [%serve-glob url-base=path =glob:glob public=?]
      [%unserve-dir url-base=path]
      [%toggle-permission url-base=path]
      [%set-landscape-homepage-prefix prefix=(unit term)
    ]
  ==
::
+$  configuration
  $:  landscape-homepage-prefix=(unit term)
  ==
::
```

```
+$  update
  $%  [%configuration =configuration]
  ==
--
```

This in turn relies on the `/sur/glob/hoon` structure defining a `glob`. (In Unix parlance, a glob is a collection of files, such as that obtained by matching a regular expression, such as `*.txt`.)

Listing 6.2: `/sur/glob/hoon`

```
|%
+$  glob  (map path mime)
--
```

Factories are code patterns that are commonly employed in creating In object-oriented programming, these are object-constructing objects. In subject-oriented programming, one uses wet cores to create the appropriate newly patterned core.

One simple constructed example (that is, not a true factory) is the use of `default` core to produce `%gall` agents.

TODO

6.2 Deep Dives in **%gall**

Several 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 ??. Other examples are new to this chapter.

%shoe/**%sole** CLI Libraries

Prefatory to examining the mechanics of `%chat-cli` and the Dojo terminal parser, we need to consider what a command-line interface is and what functionality it offers. We will then construct a simple reverse-Polish notation (RPN) calculator as a minimal working example of a `%shoe`-instrumented CLI app.

See also the https://github.com/urbit/docs/hoon/guides/cli-tutorial/Tlon `%shoe` tutorial for an alternative explanation of these principles.

Reverse-Polish Notation Calculator, **%pyth**

We wish to create a calculator which accepts one or two numbers as input, then an operator which evaluates the one or two preceding numbers on the stack, as appropriate. (For instance, unary negation could be input as `5 -`.) We will implement the following operators:

1. `+`, addition (binary operator)
2. `-`, subtraction (unary if only one number, else binary)
3. `*`, multiplication (binary operator)
4. `/`, division (binary operator)
5. `^`, exponentiation (binary operator)
6. `_`, square root (unary operator)

Like Dojo, we do not need to accept any invalid input, but we do need to deal with some special cases: automatically promoting integer input values (`@ud`) to double-precision floating-point numbers (`@rd`), and deciding whether `-` refers to a unary or binary operation. Input of a number should pause until completion is indicated by pressing Return, but an operator may be evaluated instantly.

For instance, here is an expected session, with `%` being the CLI prompt:

```
% 5
5.0
% 4
5.0 4.0
% -
1.0
% _
1.0
% -
-1.0
% 6
-1.0 6.0
% 12
-1.0 6.0 12.0
% *
-1.0 72.0
% /
-0.013888888888888888
```

We construct `%pyth` beginning with the default `%gall` core. Save this text as a file `app/pyth.hoon`.

Listing 6.3: Default `%gall` core `%pyth`

```
/+  default-agent
^-  agent:gall
=|  state=@  :: TODO still valid?
|_  =bowl:gall
+*  this      .
    default   ~(. (default-agent this %|) bowl)
::
++  on-init   on-init:default
++  on-save   on-save:default
++  on-load   on-load:default
++  on-poke
  |=  [=mark =vase]
  ~&  >  state=state
  ~&  got-poked-with-data=mark
  =.  state  +(state)
  `this
::
++  on-watch  on-watch:default
++  on-leave  on-leave:default
++  on-peek   on-peek:default
++  on-agent  on-agent:default
++  on-arvo   on-arvo:default
++  on-fail   on-fail:default
```

`%pyth` needs to set the following arms to play their roles:

1. `++on-init` should initialize an empty stack.
2. `++on-load` should clear the stack.

3. `++on-save` should push the current state for an upgrade.
4. `++on-poke`
5. `++on-agent`

Other arms may remain with their default settings.

`++on-init` is responsible to set up the prompt symbol `%` and define the parsing rules.

`++on-poke` is the workhorse which takes any input and decides how to parse it, as well as either pushing numeric values onto the stack or effecting operators.

Chat CLI

Drum, Helm, Hood, and Herb

Collectively, these agents are components of or adjacent to Dojo which permit text input (from the keyboard via `%dill` or through a plaintext API).

Bitcoin API

Ranked Voting

Ranked voting describes a class of voting systems wherein participants can designate primary, secondary, and perhaps more votes in an election, ranked by preference. Various schemes consolidate these votes into a final assessment of the winning item. In this example, we will create a `%gall` agent to implement a positional voting algorithm. We permit voters to rank three choices of many, with the first choice receiving 3 points, the second receiving 2 points, and the third choice receiving 1 point. Other choices receive no points. (This variant is called a Borda count.) There are two roles in the system: a sponsor `%sponsor` who proposes a ballot, including populating the selection candidates and finalizing the vote; and one or more voters `%voter`, possibly including the sponsor, who vote on the items.

Example 6.2.1 The `%gall` agent

Furthermore, the agent needs to interact with an external browser-based interface or CLI. In this case, we elect to compose a CLI.

:ranked-choice

```
/+  default-agent, dbug
|%
+$  versioned-state
    $%  state-0
    ==
+$  state-0  [%0 counter=@]
--
%-  agent:dbug
=|  state-0
=*  state  -
^-  agent:gall
```

```
|_  =bowl:gall
+*  this      .
    default   ~(. (default-agent this %|) bowl)
::
++  on-init   >  'on-init'
  `this(state [%0 3])
++  on-save  ^-  vase  !>(state)
++  on-load
  ~&  >  'on-load'
  on-load:default
++  on-poke
  |=  [=mark =vase]
  ~&  >  state=state
  ~&  got-poked-with-data=mark
  =.  state  +(state)
  `this
  :: =/  vote
  :: ?:  (in '')
  ::
::
++  on-watch  on-watch:default
++  on-leave  on-leave:default
++  on-peek   on-peek:default
++  on-agent  on-agent:default
++  on-arvo   on-arvo:default
++  on-fail   on-fail:default
```

Bots

6.3 Threading with Spider

6.4 Urbit API

6.5 Deep Dives with Urbit API

Time (Clock)

Publish

%graph-store

Logan: they're all enumerated in the +on-peek arm of graph-store

WebRTC Applications

Initialize Urbit airlock http-api Create UrbitRTCApp instance Register handler for incomingcall event reject/answer Call initialize() Place calls using call() or Start with peer connection from answer() Create Icepond instance Register handler for iceserver events Add media or data streams Call initialize on Icepond and peer connection

commoditizes external infrastructure dependencies unifies personal identity and address removes centralized coordination for call negotiation (peer-to-peer media calls) (bootstrapping from Urbit peer-to-peer cnxn)

6.6 Exercises

1. Produce a ranked-choice voting agent which implements instant-runoff voting. In this version, voters rank all items. A majority item wins outright (> 50%); otherwise all votes for the item with the fewest first choices are redistributed based on which item is next on each voter's ballot. This proceeds until a winner is determined.
2. Produce a command-line calculator which may be accessed by anyone.

Supporting Urbit 7

7.1 Objectives

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

1. Outline the Urbit boot process and over-the-air live updating process.
2. Explain how off-Mars calls are made with `%unix` events.
3. Evaluate Nock using a virtual machine.
4. Distinguish king and serf processes and enumerate the responsibilities of each.
5. Discuss how jet-powered code enables fast evaluation of formal Nock code.
6. (Optionally) produce a jet matching Hoon/Nock code.

7.2 Booting and Pills

Any new ship is a freshly created instance of Arvo, which requires creating Arvo then playing a standard boot sequence of events.

7.3 **%unix** Events

7.4 Nock Virtual Machines

++mock

7.5 King and Serf Daemons

Vere (Reference C Implementation)

Loom: https://rovnys-public.s3.us-east-1.amazonaws.com/rovnys-ricfer/2021.5.24..20.11.19-cgy-urbit-zeno-2-loom-memory-allocator.txt

King Haskell (Haskell Implementation)

Jaque (JVM Implementation)

7.6 Jetting

Jet matching and the dashboard

Jet composition

Example 7.6.1 Given a Hoon gate, how can a developer produce a matching C jet? Let us illustrate the process using a simple `|%` core, then with a `|_` door.

This library provides a few transcendental functions useful in many mathematical calculations. The `~%` "sigcen" rune registers the jets (with explicit arguments, necessary at the highest level of inclusion). The `~/` "sigfas" rune indicates which arms will be jetted.

Listing 7.1: `lib/trig-rs.`

```
::  Transcendental functions library, compatible with
    @rs
::
=/  tau   .6.28318530717
=/  pi    .3.14159265358
=/  e     .2.718281828
=/  rtol  .1e-5
|%
~%  %trig  ..part  ~
:: Factorial, $x!$
::
++  factorial
  ~/  %factorial
  |=  x=@rs  ^-  @rs
  =/  t=@rs  .1
  |-  ^-  @rs
  ?:  =(x .1)
    t
  $(x (sub:rs x .1), t (mul:rs t x))
:: Absolute value, $|x|$
::
++  absolute
  |=  x=@rs  ^-  @rs
  ?:  (gth:rs x .0)
    x
  (sub:rs .0 x)
:: Exponential function, $\exp(x)$
::
++  exp
  ~/  %exp
  |=  x=@rs  ^-  @rs
  =/  rtol  .1e-5
  =/  p   .1
  =/  po  .-1
  =/  i   .1
  |-  ^-  @rs
  ?:  (lth:rs (absolute (sub:rs po p)) rtol)
```

```
    p
  $(i (add:rs i .1), p (add:rs p (div:rs (pow-n x i) (
    factorial i))), po p)
:: Integer power, $x^n$
::
++  pow-n
  ~/  %pow-n
  |=  [x=@rs n=@rs]  ^-  @rs
  ?:  =(n .0)  .1
  =/  p  x
  |-  ^-  @rs
  ?:  (lth:rs n .2)
    p
  ::~&  [n p]
  $(n (sub:rs n .1), p (mul:rs p x))
--
```

We will create a generator which will pull the arms and slam each gate such that we can assess the library's behavior. Later on we will create unit tests to validate the behavior of both the unjetted and jetted code.

ng 7.2: gen/trig.hoo

```
/+  *trig-rs
!:
:-  %say
|=  [[* eny=@uv *] [x=@rs n=@ud ~] ~]
::
~&  (factorial n)
~&  (absolute x)
~&  (exp x)
~&  (pow-n n)
[%verb ~]
```

Mise en place Jet development can require frequent production of new Urbit binaries and , so let us pay attention to our mise en place. We will primarily work in the fakezod on the two files just mentioned, and in the `pkg/urbit` directory of the main Urbit repository, so we need a development process that allows us to quickly access each of these, move them into the appropriate location, and build necessary components. The basic development cycle will look like this:

1. Compose correct Hoon code.
2. Hint the Hoon code.
3. Register the jets in the Vere C code.
4. Compose the jets.
5. Compile and troubleshoot.
6. Repeat as necessary.

You should consider using a terminal utility like `tmux` or `screen` which allows you to work in several locations on your file system simultaneously: one for file system operations (copying files in and out of the `home` directory), one for running the fakezod, and one for editing the files, or an IDE or text editor if preferred.

![One recommended screen layout for jet composition and development.](TODO)

To start off, you should obtain a clean up-to-date copy of the Urbit repository, available on GitHub at `https://github.com/urbit/urbit`github.com/urbit/urbit. Make a working directory `~/tlon`. Create a new branch within the repo named `trigjet`:

Listing 7.3: Terminal 1 (I

```
cd
mkdir tlon
git clone https://github.com/urbit/urbit.git
cd urbit
git branch trigjet
```

Test your build process to produce a local executable binary of Vere:

Listing 7.4: Terminal 1 (I

```
make
```

This invokes Nix to build the Urbit binary. Take note of where that binary is located (typically in `/tmp` on your main file system) and create a new fakezod using a downloaded pill.

Listing 7.5: Terminal 1 (I

```
cd ..
wget https://bootstrap.urbit.org/urbit-v1.5.pill
<Nix build path>/bin/urbit -B urbit-v1.5.pill -F zod
```

Inside of that fakezod, sync `%clay` to Unix.

Listing 7.6: Terminal 2 (I

```
|mount %
```

Then copy the entire `home` desk out so that you can work with it and copy it back in as necessary.

Listing 7.7: Terminal 1 (I

```
cp -r zod/home .
```

Save the foregoing library code in `home/lib` and the generator code in `home/gen`. Whenever you work in your preferred editor, you should work on the `home` copies, then move them back into the fakezod and synchronize before execution.

Listing 7.8: Terminal 1 (I

```
cp -r home zod
```

Listing 7.9: Terminal 2 (I

```
|commit %home
```

Jet composition Now that you have a developer cycle in place, let's examine what's necessary to produce a jet. A jet is a C function which replicates the behavior of a Hoon (Nock) gate. Jets have to be able to manipulate Urbit quantities within the binary, which requires both the proper affordances within the Hoon code (the interpreter hints) and support for manipulating Urbit nouns (atoms and cells) within C.

Jet hints must provide a trail of symbols for the interpreter to know how to match the Hoon arms to the corresponding C code. Think of these as breadcrumbs. The current jetting documentation demonstrates a three-deep jet; here we have a two-deep scenario. Specifically, we mark the outermost arm with `~%` and an explicit reference to the Arvo core (the parent of `part`). We mark the inner arms with `~/` because their parent symbol can be determined from the context. The `@tas` token will tell Vere which C code matches the arm. All symbols in the nesting hierarchy must be included.

Listing 7.10: Jet lib/trig-rs.hoon

```
|%
~%  %trig  ..part  ~
++  factorial
```

```
  ~/  %factorial
  |=  x=@rs  ^-  @rs
  ...
++  exp
  ~/  %exp
  |=  x=@rs  ^-  @rs
  ...
++  pow-n
  ~/  %pow-n
  |=  [x=@rs n=@rs]  ^-  @rs
  ...
--
```

We also need to add appropriate handles for the C code. This consists of several steps:

1. Register the jet symbols and function names in `tree.c`.
2. Declare function prototypes in headers `q.h` and `w.h`.
3. Produce functions for compilation and linking in the `pkg/urbit/jets/e` directory.

The first two steps are fairly mechanical and straightforward.

Register the jet symbols and function names. A jet registration may be carried out at in point in `tree.c`. The registration consists of marking the core

ng 7.11: Add
urbit/jets/tree.c

```
/* Jet registration of ++factorial arm under trig */
static u3j_harm _140_hex__trig_factorial_a[] = {{".2",
     u3we_trig_factorial, c3y}, {}};
/* Associated hash */
static c3_c* _140_hex__trig_factorial_ha[] = {
  "903
    dbafb8e59427eced0b35379ad617c2eb6083a235075e9cdd9dd80e732efa4
    ",
  0
};

/* Jet registration by token for ++factorial */
static u3j_core _140_hex__trig_d[] =
  { { "factorial", 7, _140_hex__trig_factorial_a, 0,
    _140_hex__trig_factorial_ha },
    {}
  };
/* Associated hash */
static c3_c* _140_hex__trig_ha[] = {
  "0
    bac9c3c43634bb86f6721bbcc444f69c83395f204ff69d3175f3821b1f679ba
    ",
  0
};

/* Core registration by token for trig */
static u3j_core _140_hex_d[] =
{ /* ... pre-existing jet registrations ... */
  { "trig", 31, 0, _140_hex__trig_d,
    _140_hex__trig_ha  },
  {}
```

```
25 };
```

Jet hashes are not active at the moment time.

The numeric component of the title, `140`, indicates the Hoon Kelvin version. Library jets of this nature are registered as `hex` jets, meaning they live within the Arvo core. Other, more inner layers of `%zuse` and `%lull` utilize `pen` and other three-letter jet tokens. The core is conventionally included here, then either a `d` suffix for the function association or a `ha` suffix for a jet hash.

The particular flavor of C mandated by the Vere kernel is quite lapidary, particularly when shorthand functions (such as `u3z`) are employed. In this code, we see the following `u3` elements:

1. `c3_c`, the platform C 8-bit `char` type
2. `c3y`, loobean yes, `%.y`
3. `u3j_core`, C representation of Hoon/Nock cores
4. `u3j_harm`, an actual C jet ("Hoon arm")

The numbers `7` and `31` refer to relative core addresses. In most cases—unless you're building a particularly complicated jet or modifying `%zuse` or `%lull`—you can follow the pattern laid out here. `".2"` is a label for the axis in the core `[battery sample]`, so just the battery. The text labels for the `|%` core and the arm are included at their appropriate points. Finally, the jet function entry point `u3we_trig_factorial` is registered.

Declare function prototypes in headers.

A `u3w` function is always the entry point for a jet. Every `u3w` function accepts a `u3noun` (a Hoon/Nock noun), validates it, and invokes the `u3q` function that implements the actual logic. The `u3q` function needs to accept the same number of atoms as the defining arm (since these same values will be extricated by the `u3w` function and passed to it).

In this case, we have cited `u3we_trig_factorial` in `tree.c` and now must declare both it and `u3qe_trig_factorial`:

Listing 7.12: Adding to pkg/urbit/include/w.h

```
u3_noun u3we_trig_factorial(u3_noun);
```

Listing 7.13: Adding to pkg/urbit/include/q.h

```
u3_noun u3qe_trig_factorial(u3_atom);
```

Produce functions for compilation and linking.

Since this function deals with floating-point values, the SoftFloat library is included. This library represents `float`s as `struct`s which can be examined using a `union`.

Given these function prototype declarations, all that remains is the actual definition of the function. Both functions will live in their own file; we find it the best convention to associate all arms of a core in a single file. In this case, create a file `pkg/urbit/jets/e/trig.c` and define all of your `trig` jets therein.

Listing 7.14: pkg/urbit/j

```
/* jets/e/trig.c
**
*/
#include "all.h"
#include <softfloat.h>  // necessary for working with software-defined floats
#include <stdio.h>      // helpful for debugging, removable after development
#include <math.h>       // provides library fabs() and ceil()
```

```
8
9   union sing {
10    float32_t s;    //struct containing v, uint_32
11    c3_w c;         //uint_32
12    float b;        //float_32, compiler-native,
      useful for debugging printfs
13  };
14
15  /* ancillary functions
16  */
17    bool isclose(float a,
18                 float b)
19    {
20      float atol = 1e-6;
21      return ((float)fabs(a - b) <= atol);
22    }
23
24  /* factorial of @rs single-precision floating-point
      value
25  */
26    u3_noun
27    u3qe_trig_factorial(u3_atom u)  /* @rs */
28    {
29      union sing a, b;
30      a.c = u3r_word(0, u);  // extricate value from
      atom as 32-bit word
31
32      if (ceil(a) != a) {
33        // raise an error if the float has a nonzero
      fractional part
34        return u3m_bail(c3__exit);
35      }
36
37      if (isclose(a.b, 1.0)) {
38        return u3i_word(a.b);
39      }
40      else {
41        // naive recursive algorithm
42        b.b = a.b - 1.0;
43        return u3i_word((c3_w)f32_mul(a.s, b.s).v);
44      }
45    }
46
47    u3_noun
48    u3w_trig_factorial(u3_noun cor)
49    {
50      u3_noun a;
51
52      if ( c3n == u3r_mean(cor, u3x_sam_2, &a, 0) ||
53           c3n == u3ud(a) )
54      {
55        return u3m_bail(c3__exit);
56      }
57      else {
58        return u3q_trig_factorial(a);
59      }
60    }
```

This code merits ample discussion. Without focusing on the particular types used, read through the logic and look for the skeleton of a standard simple factorial algorithm.

We omit from the current discussion a few salient points:

1. Reference counting with transfer and retain semantics. (For everything the new developer does outside of real Hoon shovel work, one will use transfer semantics.)
2. The structure of memory: the loom. This is discussed in Section ??.
3. Many details of C-side atom declaration and manipulation from the `u3` library.

We commend to the reader the exercise of selecting particular library functions provided with the system, such as `++cut`, locating the corresponding jet code, and learning in detail how particular operations are realized in `u3` C. Note in particular that jets do not need to follow the same solution algorithm and logic as the Hoon code; they merely need to reliably produce the same result. Thus, you should extensively test your Hoon code and your C jet code.

8 Concluding Remarks

8.1 Booting and Pills

Appendix

Appendices A

A.1 Comprehensive table of Hoon runes

These runes are up-to-date as of Hoon `%140`. Runes are ordered by alphabetical pronunciation of Urbit-standard aural ASCII. Standard definitions (such as `+$map` and `+$unit`) are defined in Section ??. Of those remaining, the most common is `+$hoon`, which is used as a shorthand for any valid Hoon expression. A `+$term` is TODO tome spec value

Digraphs

`~`	"sig"	`!`	"zap"	"pat"

Table A.1: Aural ASCII Pronunciation

`|` "bar": Core Definition

`|` "bar" runes produce cores.

`|$ (lest term) spec`

`|$` "barbuc" produces a mold (a type definition) given a non-empty list or `+$lest` of `+$term`s (`@tas` ASCII symbols, the labels) and a structure definition or `+$spec`. In other words, `|$`

Definition

Examples

```
++  lest
  |$  [item]
  ::    null-terminated non-empty list
  ::
  ::  mold generator: produces a mold of a null-
    terminated list of the
  ::  homogeneous type {a} with at least one element.
  [i=item t=(list item)]
```

Listing A.1: `++lest` (non-empty l from `hoon.hoon`

```
++  pair
  |$  [head tail]
  ::    dual tuple
  ::
  ::  mold generator: produces a tuple of the two types
    passed in.
  ::
  ::  a: first type, labeled {p}
  ::  b: second type, labeled {q}
  ::
  [p=head q=tail]
```

Listing A.2: `++pair` from `hoon.hoon`

`|_ spec alas (map term tome)`

`|_` “barcab”produces a door (a core with sample) given a spec alas and a

Examples

```
TODO
```

Listing A.3: `+$mk-item` frc

```
::  $mk-item: constructor for +ordered-map item type
++  mk-item
  |$  [key val]
  [key=key val=val]
```

`|% (unit term) (map term tome)`

`|_` “barcab” produces a core (battery and payload) given a `unit` of `term`s (`@tas` ASCII symbols, arm labels, the battery) and a dictionary of named arms, or `map` from `term` keys to `tome` values (). In other words,

`|: [p=hoon q=hoon]`

`|:` “barcol” produces a gate with a custom sample given a

Definition

```
1 [%tsls p.gen [%brdt q.gen]]
2 =+  p  |. q
```

`|. hoon`

produces a trap (a core with one arm)

`|- hoon`

produces a trap (a core with one arm) and evaluates it

`|^ hoon (map term tome)`

produces a core whose battery includes a $ arm and computes the latter

`| (unit term) (map term tome)`

produces a wet core (battery and payload)

`|~ [spec value]`

produces an iron gate

`|* [spec value]`

produces a wet gate (a one-armed core with sample)

`|= [spec value]`

produces a dry gate (a one-armed core with sample)

`|? hoon`

`|?` “barwut” produces a lead trap (bivariant) or core with a single arm
`$`

`%lead` lets you export a library interface but hide the implementation details

`$` “buc”: Mold Definition

`$` “buc” runes produce mold definitions.

`%` “cen”: Core Evaluation

`%` “cen” runes evaluate cores (similar to function calls in other languages).

```
(~(rad og eny) 5)
(rad:og 5)  :: TODO figure out tersest working
    equivalent here
```

`:` “col”: Cell Construction

`:` “col” runes produce cells of various structures.

`.` “dot”: Nock Evaluation

`.` “dot” runes directly evaluate as Nock expressions.

`^` “ket”: Core Typecasting

`^` “ket” runes are used to alter cores.

`~` “sig”: Hinting

`~` “sig” runes represent directives to the runtime.

`;` “mic”: Macro

`;` runes produce calling structures (mainly monadic binds) and XML elements.

`=` “tis”: Composition

`=` runes modify the subject.

`?` “wut”: Conditional

`?` runes compare subtrees.

`!` “zap”: Wildcard

`!` runes perform a miscellaneous set of auxiliary operations.

A.2 Hoon versions

A.3 Nock versions

A.4 Hoon comparison with other languages

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

A.6 Textbook changelog