University of California
Santa Barbara

A New Human-Readability Infrastructure for Computing

A dissertation submitted in partial satisfaction
of the requirements for the degree

Doctor of Philosophy
in
Computer Science

by

Christopher Kyle Hall

Committee in charge:

Professor Tobias Höllerer, Chair
Professor Tevfik Bultan
Professor Chandra Krintz

September 2017

The Dissertation of Christopher Kyle Hall is approved.

Professor Tevfik Bultan

Professor Chandra Krintz

Professor Tobias Höllerer, Committee Chair

March 2017

A New Human-Readability Infrastructure for Computing

I dedicate this manuscript to my father Stan, who has always stoked the fire of my curiosity, and to Kyle, my late grandfather whom I have never met, but who would have also been deeply invested in this particular academic pursuit.

Acknowledgements

I am eternally grateful to my advisor, Prof. Tobias Höllerer, for his trust and generosity in creating a research environment that enabled me to incubate a long-term agenda. I thank Trevor for accepting nothing less than excellence from me, for helping to keep my priorities balanced with a complementary perspective, and for all the sleepless nights we spent writing. I thank Erica for her support, enthusiasm, and patience through this process. She and I have shared the strains of a long-distance relationship as well as the celebrations of momentous milestones. I thank my mother and grandmother for keeping me housed and fed during the final push to completion.

Thanks for all your encouragement!

Curriculum Vitæ
Christopher Kyle Hall

Education

2017	Ph.D. in Computer Science (Expected), University of California, Santa Barbara.
2012	M.S. in Computer Science, University of California, Santa Barbara.
2010	B.S. in Computer Science, University of California, Santa Barbara.

Publications

Hall, C., Standley, T., & Hollerer, T. (2017, October). Infra: Structure All the Way Down - Structured Data as a Visual Programming Language. In Proceedings of the 2017 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (to appear). ACM.

Hall, C. (2015, October). Rethinking the human-readability infrastructure. In Proceedings of the Workshop on Future Programming (pp. 1-6). ACM.

Hall, C. (2014, October). HCI metacomputing: universal syntax, structured editing, and deconstructible user interfaces. In Proceedings of the companion publication of the 2014 ACM SIGPLAN conference on Systems, Programming, and Applications: Software for Humanity (pp. 21-24). ACM.

Bostandjiev, S., O'Donovan, J., Hall, C., Gretarsson, B., & Hollerer, T. (2011, September). Wigipedia: A tool for improving structured data in wikipedia. In Semantic Computing (ICSC), 2011 Fifth IEEE International Conference on (pp. 328-335). IEEE.

Gretarsson, B., O'Donovan, J., Bostandjiev, S., Hall, C., & Hllerer, T. (2010, June). Smallworlds: visualizing social recommendations. In Computer Graphics Forum (Vol. 29, No. 3, pp. 833-842). Blackwell Publishing Ltd.

Abstract

A New Human-Readability Infrastructure for Computing

by

Christopher Kyle Hall

We present Infra, a new baseline medium for representing data. With Infra, arbitrarily-complex structured data can be encoded, viewed, edited, and processed, all while remaining in an efficient non-textual form. It is suitable for the full range of information modalities, from free-form input, to compact schema-conforming structures. With its own equivalent of a text editor and text-field widget, Infra is designed to target the domain currently dominated by flat character strings while simultaneously enabling the expression of sub-structure, inter-reference, dynamic dependencies, abstraction, computation, and context (metadata).

Existing metaformats fit neatly into two categories. They are either textual for human readability (such as XML and JSON) or binary for compact serialization (such as Thrift and Protocol Buffers). In contrast, Infra unifies those two paradigms. In order to have the desirable properties of binary formats, Infra has no textual representation. And yet, it is designed to be easily read and authored by end-users.

We show how the organization Infra brings to data makes a new non-textual programming paradigm viable. Programs that modify data can now be embedded into the data itself. Furthermore, these programs can often be authored by demonstration. We argue that Infra can be used to improve existing software projects and that bringing direct authoring and human readability to a binary data paradigm could have rippling ramifications on the computing landscape.

Contents

Chapter 1

Introduction

Early computers used electromechanical typewriters to interface with humans. That legacy is alive today. Printable character codes are the sole building blocks of source code files, command line languages, form fields, Web formats (HTTP, URL, HTML, JSON, CSS) and all other "human-readable" formats. "A coded character set provides the basic elements of communication between humans and data-processing or telecommunication machines." [1] Though all CPUs and runtime data-structures rely on binary-encoded quantities to structure information for random access, formats that need to be able to have a direct relationship with users are stuck with essentially one option - encoding their structure indirectly via contrived patterns of co-opted character codes. This is unfortunate because, text, as an infrastructure and encoding paradigm, comes not only with compromises to efficiency, but functionality in general.

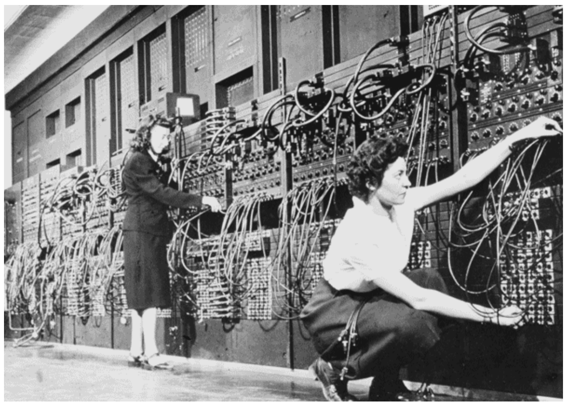
Let us define "plaintext infrastructure" as the set of text encoding, display, manipulation, and processing artifacts currently ubiquitous in computing: ASCII, UTF8, text editors, text-field or text-area UI widgets, terminals / consoles, keyboards (physical and virtual), String types, object-to-String rendering functions, human-readable format libraries, tokenizers, parsers, escape sequences and input sanitization, Base64 encoding, line-ending and whitespace conventions, and the fallback data-flavor of the copy/paste clipboard. We believe that an alternative infrastructure, centered around metadata, can be positioned to fill the same role that the plaintext infrastructure currently plays, while bringing with it the building blocks of software (*structure* and *abstraction*), time and space encoding efficiencies, improved human-readability, and richer ways to author and interact with raw and persistable information. With a new medium such as this made common across user environments, the natures of file formats, data structures, hypermedia, source code, and graphical user-interfaces, can be unified. This could simplify computing by reducing the number of distinct modalities that users need to learn and allow the best properties of each to exist evenly throughout. By removing the need for formats to choose between an efficient structured binary type of encoding and a compromised but human readable type of encoding, every modality of computing could have a parsed nature, data-driven presentation abstractions, and interactive computation.

In the next chapter, we present Infra, our medium that is simultaneously suitable for use as a high-efficiency binary metaformat, a human-readable markup language, a programming language, and as plain text - positioned to play any role that UTF8 currently plays. Infra aims to make data more powerful and easier to deal with. We start by describing how Infra can be used to replace existing human-readable data formats even though Infra is binary encoded. We show that this approach has a number of benefits over text-based formats. Following that, we describe one of these benefits, Patch, in detail. Patch is a non-textual programming language that targets the domain of data metaprogramming. In fact, Infra's structured approach to data representation is what makes Patch viable. The major contributions in this work include the design and implementation of Infra's universal metaformat, a runtime system for Patch, and a universal structured editor to bring direct editing and human readability to Infra.

1.1 Revolutions in End-User Media

By our accounting, there have been roughly three major revolutions in the nature of common end-user media throughout computing history thus far. We identify them as the following:

1. **Text**: The modalities of computer operators transitioned from direct physical manipulation of a computer's patch board, dials, and lights, to reading programmatically formatted output and typing in human-readable command languages.

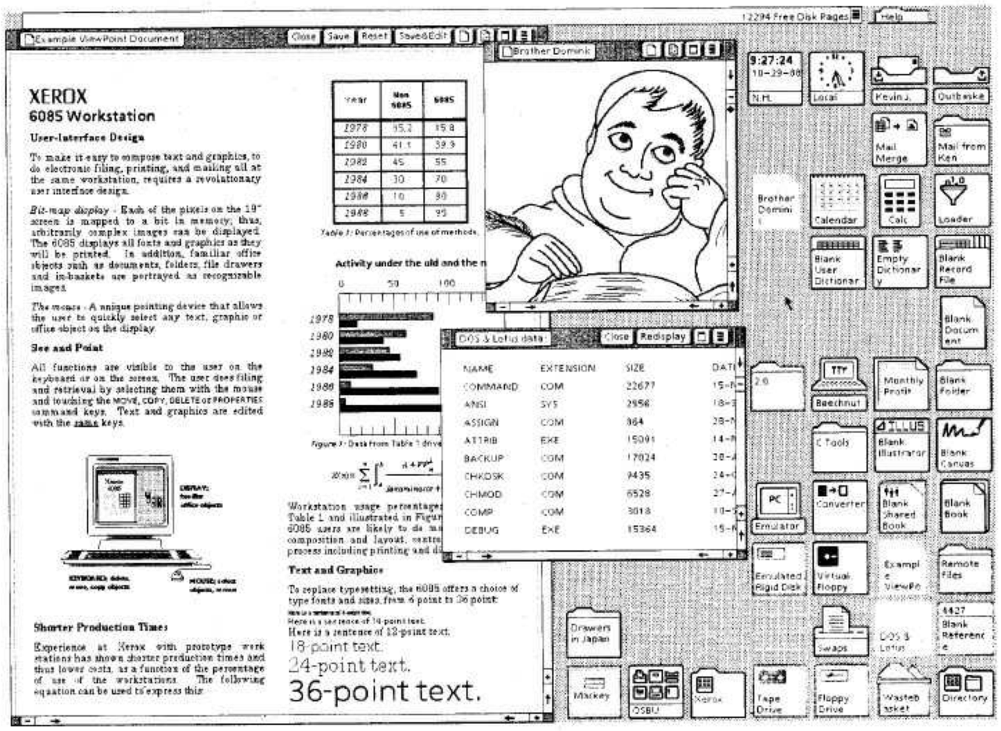
   

   Text brought with it the concepts of character encodings, line editors, text editors, transcript-based user interfaces, parsers, and interpreters. Streams and arrays of character codes have formed the common medium between humans and software ever since computers started leveraging electromechanical typewriters as command consoles in the 1950s.

   Once interactive graphics started to become commonplace, this situation evolved in a significant but isolated way. The history of encoding text in an enumerated form dates back to the optical telegraph semaphore-line of 1792. This influenced the electrical telegraph Morse code of 1837, the Baudot Code of 1870, the Extended Binary Coded Decimal Interchange Code (EBCDIC) in the 1950s, and the American Standard Code for Information Interchange (ASCII) character encoding in the

1960s [2]. The physical human-interface portion of early-computing's infrastructure is a direct adaptation of the existing typewriter culture and electromechanical teletypes, which were designed around EBCDIC and ASCII.

2. **GUI**: The Graphical User Interface revolution brought richer ways to communicate information hierarchy to users and modeless interaction to navigate it. GUI's brought with them bit-mapped displays, spatial syntax, and recognition over recall. Its influences include Jean Piaget's Constructivism'37 [3], Ivan Sutherland's Sketchpad'64 [4], the Simula 67 programming language [5], the GRaIL graphical input language [6], Jerome Bruner's perspective on developmental psychology [7], Seymour Papert's LOGO [8], the interaction hardware and bootstrapping approach of Douglas Engelbart's augmented human intellect project [9], and their culmination in the Smalltalk system [10].



XEROX
6085 Workstation

User-Interface Design

To make it easy to compose text and graphics, to do electronic filing, printing, and mailing all at the same workstation, requires a revolutionary user interface design.

Bit-map display - Each of the pixels on the 19" screen is mapped to a bit in memory; thus, arbitrarily complex images can be displayed. The 6085 displays all fonts and graphics as they will be printed. In addition, familiar office objects such as documents, folders, file drawers and in-baskets are portrayed as recognizable images.

The mouse - A unique pointing device that allows the user to quickly select any text, graphic or office object on the display.

See and Point

All functions are visible to the user on the keyboard or on the screen. The user does filing and retrieval by selecting them with the mouse and touching the MOVE, COPY, DELETE or PROPERTIES command keys. Text and graphics are edited with the same keys.

Shorter Production Times

Experience at Xerox with prototype work stations has shown shorter production times and thus lower costs, as a function of the percentage of use of the workstations. The following equation can be used to express this

Year	Non 6085	6085
1978	85.2	15.8
1980	41.1	39.9
1982	45	55
1984	30	70
1986	10	90
1988	5	95

Table 1: Percentages of use of methods.

Figure 1: Data from Table 1 drive

Workstation usage percentage
Table 1 and illustrated in Figur
6085 users are likely to do m
composition and layout, centra
process including printing and di

Text and Graphics

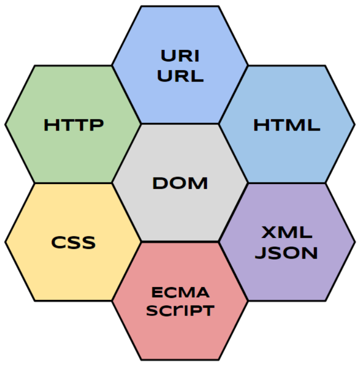
To replace typesetting, the 6085 offers a choice of type fonts and sizes, from 6 point to 36 point:

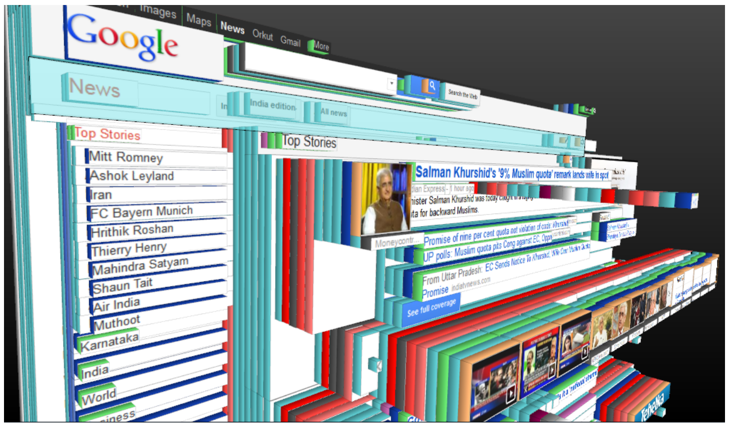
Here is a sentence of 14-point text.
Here is a sentence of 12-point text.
18-point text.
24-point text.
36-point text.

NAME	EXTENSION	SIZE	DATE
COMMAND	COM	22677	15-N
ANSI	SYS	2956	18-3
ASSIGN	COM	864	28-N
ATTRIB	EXE	15091	14-M
BACKUP	COM	17024	28-A
CHKDSK	COM	9435	24-C
CHMOD	COM	6528	27-A
COMP	COM	3018	10-5
DEBUG	EXE	15364	15-N

This allowed for high-level abstractions and domain-tailored visualizations within graphical applications, but these are imposed from the top down by the specific application's logic. If and when data is exported and interacted with directly from outside of the application, those abstractions do not have a means to 'travel' along with it.

3. **DOM**: (World-Wide-Web Document Object Model). More recently, computing has entered an era where a majority of cross-platform content is consumed through a web browser. The DOM brought with it, a simplified form of hypermedia, document-based user interfaces, data-driven presentation and behavior, platform-independent standardized semantics, direct reference to/from any document in a global address space, and cache-as-needed web-surfing (no installation). Its influences include Vannevar Bush's Memex [11], Theodore Nelson's Xanadu project [12], Douglas Engelbart's oN-Line System [9], and Tim Berners-Lee's ENQUIRE System [13].





The web browser acts as an outsourced user-interface framework, factoring the overarching business logic into 'back ends' and the tight-loop interaction logic into portable 'front ends'. A major component of this is the wide-spread standardization of flexible semantics for presentation (CSS), content structure (HTML), and sandboxed scripting of behavior (ECMAScript). However, web browsers are designed around the display and consumption of web page documents only, not around authorship, editing, or publishing of those documents. While sophisticated web-page authoring tools have existed all along, the trend lines do not point towards HTML eventually becoming the backing format for all information authored by end-users. In other words, the web stack is not poised to bring its revolutionary properties to the lowest-level medium of computing by replacing plain text. After all, it is itself made of a set of UTF8 encoded formats.

Looking forward: Notice that these three media revolutions have all been additive, i.e. each new media has supplemented the previous and all continued to have their unique place in parallel. We see a path forward to a *fourth* end-user common-media revolution that closes the gap between the properties of high-level domain-tailored user interfaces and low-level manipulation of individual printable characters in strings. By this we mean putting those two layers on the same continuum by bringing the text medium's transparency to user interfaces, and imbuing the text medium with an application's capacity for data structures and abstraction. These maneuvers are one in the same, but require two general technical artifacts: a self-describing universal binary metaformat, and free-form structure editors to make it human-readable. This could have the capacity to unify the previous three into a single media.

1.2 Why Another Revolution is Needed

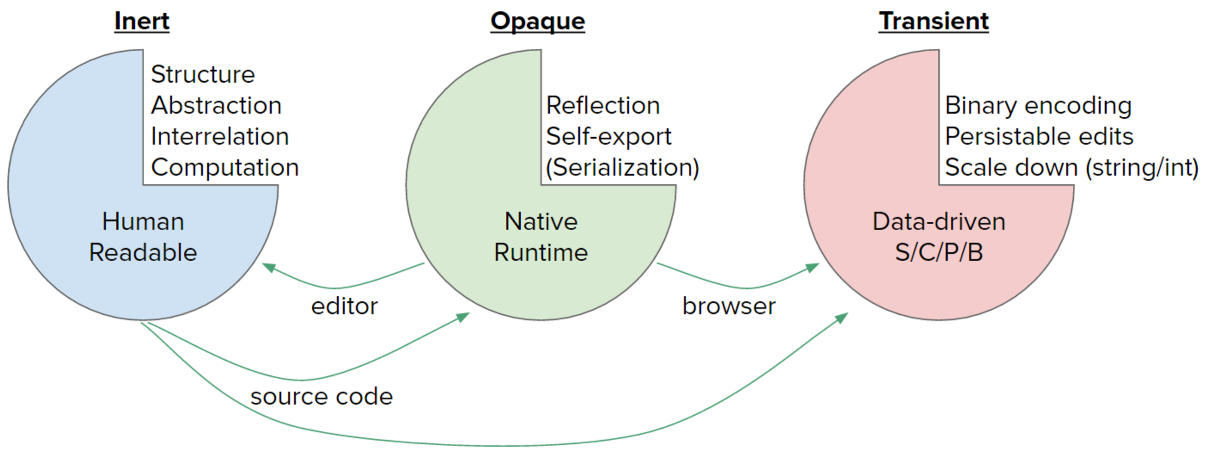
This work is motivated by many trains of thought and frustrations with computing both from a developer perspective and from an end-user perspective. Hopefully a rough gestalt can be communicated through a brief discussion of the following two themes - one from academia, and one from industry:

- “The computer revolution has not happened yet” –Alan Kay 1997 [14]
- “Everyone should learn to code” –The general zeitgeist of the 2010s [15]

1.2.1 “The computer revolution has not happened yet”

On many occasions, Alan Kay (of Xerox PARC fame) has decried the trajectory of personal computing for not evolving itself beyond the paper-centric metaphors to continue becoming its own thing. Computers are still used mostly to simulate ‘old media’. This is devastating in the sense that it takes a general-purpose machine and assembles it into less powerful building blocks. He advises us to “take the powerful thing you’re working on and not loose it by partitioning up your design space”. On a computer, computational capacity should be ubiquitous, and development (writing source code) does not have to happen outside the language itself. This way, computation can help itself scale by “allowing objects to make experiments with other objects in a safe way to see how they respond to various messages”. By moving from a protocol approach to a universal interface language approach, we can move from a clockwork-like paradigm to a biology-like paradigm for ‘growing’ systems.

Our takeaway from this is that computing should be self-similar, all the way down to its rawest form of media. None of the forms of media that came out of the revolutions outlined above have this property. Each medium is incomplete on its own; each counts on being propped up by another.

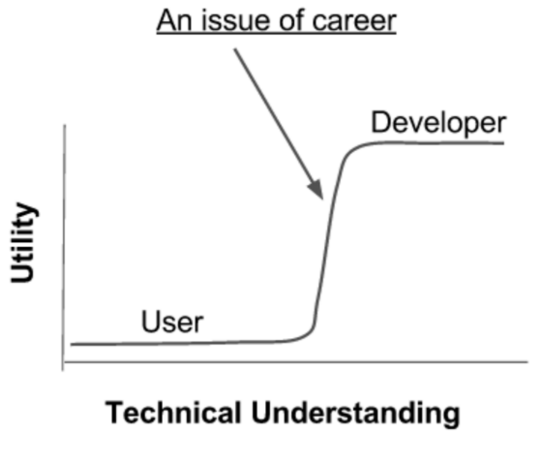


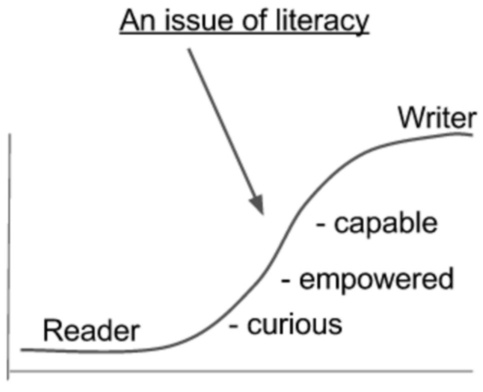
In the above figure, the items listed in the missing wedges represent the concepts missing from that medium. The arrows between media indicate critical roles one media plays for another.

1.2.2 "Everyone should learn to code"

In 2016, former president Barack Obama announced a 'Computer Science for All' initiative stating that "computer science (CS) is a 'new basic' skill necessary for economic opportunity and social mobility", mobilizing K-12 teachers to be trained in CS curricula, and enlisting the help of state governments. Several mayors ended up publicly announcing that they will be learning to code. Computer Science curriculum should certainly be available to anyone with an interested in it, but should everyone learn to code?

Alan Kay named Smalltalk after the prediction that in the not-too-distant future, programming and computational metaphors would become such an everyday form of literacy that it would be a thing of small talk. Smalltalk has had its name for over 45 years now, and that vision has not yet come to fruition. We believe that this is because **computational literacy does not earn proportional utility**, yet. By this we mean that having a casual or cursory engagement with computer science or software development buys you nothing in the other verticals of your interactions with software throughout your day. An average smartphone app does not have any inlet for a user to apply their computational fluency to bridge correspondingly-sized gulfs of execution [16]. Unlike physical artifacts, user-interfaces cannot be broken open as they are running to reveal their live material and witness their inner choreography. User interfaces are opaque and routinely leave existing useful functionality unexposed. The reader may be thinking that source code is the analog of this, but we find source code akin to only being able to see an organism's DNA strands in an X-ray, and source code does not capture the dimension of memory or live state. Also, unlike tangible human-scale phenomenon, such as dance, there is no obvious way or immediately intuitive place to mimic, as an inspired amateur, the interesting fragments of what was just seen performed so expertly by one's favorite software. It is not unheard of for curious mechanically inclined children who grow up with an abundance of access to cars, to become a decent self-taught auto-mechanic. On the other hand, it cannot be expected that a child who grows up with an iPad with an abundance of access to apps, to become a decent self-taught software developer. The media of computing, from an end-user perspective, is just not conducive to its own curriculum.





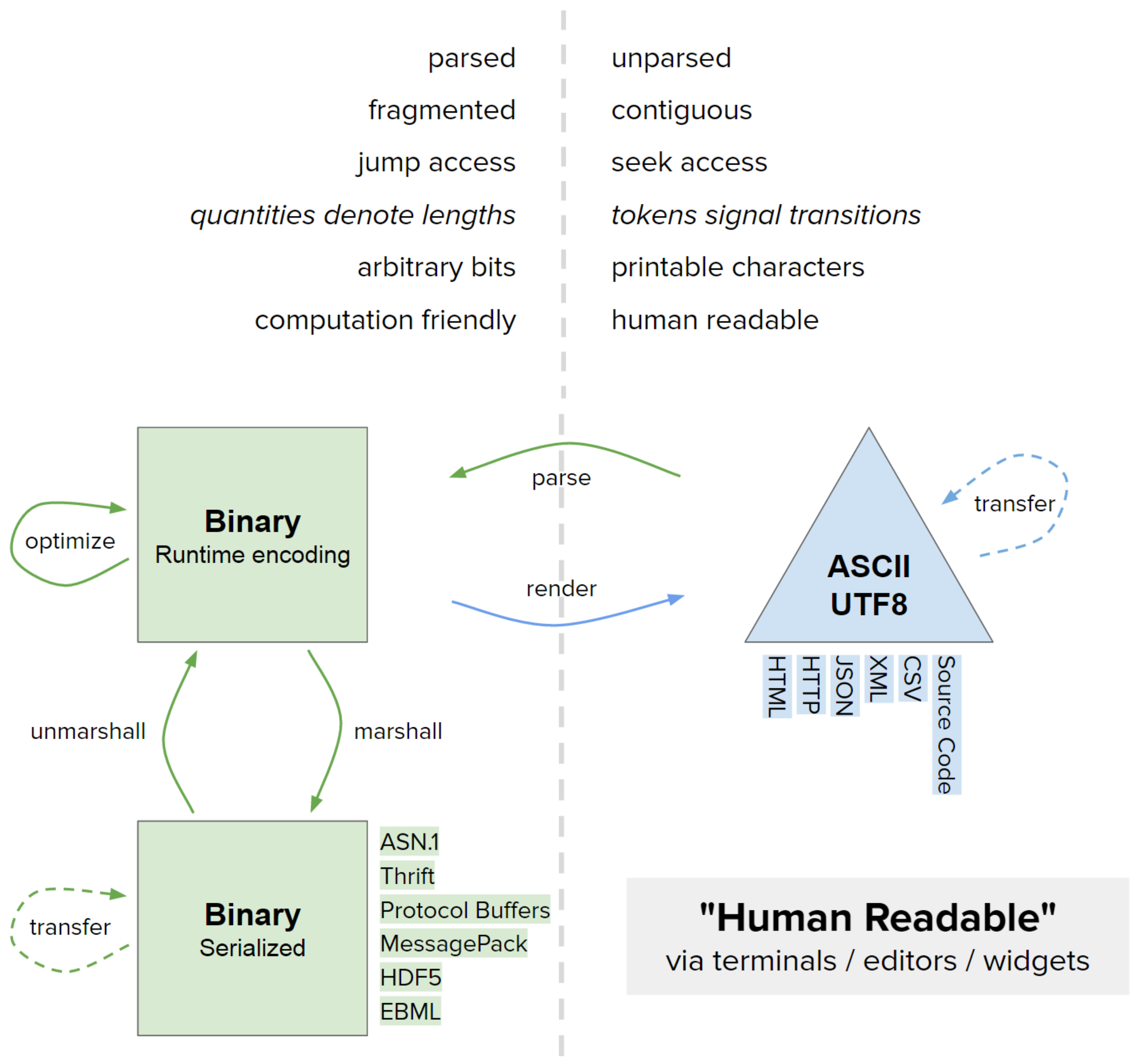
Once a user has an end-to-end level of computer science fluency, they are highly empowered to make their own software from scratch or modify existing open source tools, but this jump in personal utility for their daily lives comes all at once after prolonged investment (depicted by the curve on the left). We believe there is potential for a different computing landscape that would naturally support more incremental empowerment (more like the hypothetical curve on the right). This would mean that partial fluencies would stand on their own dividends, and any further investment toward developer-level skill could be stopped without it all being for naught.

1.3 The Opportunity

1.3.1 Observation 1: There are two paradigms of syntax

Software uses binary quantities to structure information and to jump around in a fragmented memory model (runtime heap). This is important for mechanical speed and simplicity. But of course, that does not resemble the paradigm of written natural language, which humans find important for speed and simplicity. An ecosystem of parsers and renderers are employed to continuously convert between the two so that humans and computers can each use the paradigm most intuitive to them. Though all CPUs and runtime data-structures rely on binary-encoded quantities to structure information for random access, formats that need to be able to have a direct relationship with users are stuck with essentially one option - encoding their structure indirectly via contrived patterns of character codes. This is unfortunate because, text, as a UI paradigm, comes not only with compromises to efficiency, but functionality in general.

Textual formats have a serialized nature, and each language grammar designates particular character sequences to signal mode changes in its parsers. This is necessary to encode any semblance of structure or type variation, but punches holes in the content space because control characters (meta-characters) can occur at any position.



These two different natures for encoding information have a set of mutually exclusive properties. The above figure outlines their major properties in the form of a life-cycle. On the left is the territory of computation (runtimes), and on the right is the territory of human readability and writability (communication). Though it is often swept under the table, it is important to acknowledge that the output of a parser is indeed also an encoding with a syntax, there is just never a direct awareness of it outside of very low-level debugging. Parsed data takes on the internal encoding conventions of the programming languages own runtime. While there is some variety in runtime structures between languages, the outlined properties hold universally.

In a parsed encoding, there is no need for data to remain continuous. It can be fragmented throughout a large memory space without having to be moved around as other data comes and goes. Items can be reached with jump access and offsets. The fundamental building block for this structure is quantities. Computing hardware is built for handling binary quantities, and programs are compiled to solve problems in numerical machine-readable representations. When exporting information out of a runtime, it is rendered to some stand-alone and serialized form. If there is any anticipation that it will, or could, be for human consumption / communication, that renderer will need to target a human-readable encoding. As discussed earlier, this means that printable characters will be the sole building blocks in the representation and will be structured by tokens in some higher-level grammar. When reading text, every character must be scanned because any one of them has equal opportunity to be a control character from the grammar, which alters the meaning of the characters to follow. Thus, seek access must be employed (no skipping over anything). A direct consequence is that text is contiguous with no gaps.

Quantitative Syntax Now that these two paradigms have been identified, they should be given names for later reference. Since the computation-anchored paradigm is most characterized by its use of quantities, we refer to it as *quantitative syntax.* In quantitative syntax, the run-lengths for units of structure are provided upfront in one form or another. In other words, the amount of addressing to skip in order to reach the end of that item. At its simplest, this directly encodes a tokenization. Here is a contrived example assuming a simple textual single-character length preamble preceding each 'segmented' element: **3www2w33org** would encode the domain name for the W3C ("www.w3.org") without the need to ever have chosen a particular character to act as the domain name delimiter. Note that this keeps the structural part of syntax from being content or grammar sensitive. This paradigm requires some degree of mechanical logic to author because there are invariants that need to be upheld. If edited directly, length values in the headers must be updated to match any changes in content size. Thus, this paradigm tends not to exist outside the context of digital automation - i.e. a passive piece of paper does not coordinate the updating of previous ink strokes in lockstep with the addition of new ones.

Qualitative Syntax *Qualitative syntax* is characterized by its use of predesignated symbols/tokens (a finite lexicon) reserved to represent both structure and type control signals. Most anything textual is this flavor, including source code, and thus it is usually what is implied when referring to the concept of 'syntax' in general. Units of structure extend until encountering a symbol designated as its terminator. e.g. A sentence continues until a period, a C string continues until a null byte, a file in DOS continues until character 26 (EOF). In many cases a token continues until a value that is not in the alphabet for that type. In order to nest hierarchies of data, a different token is used to distinguish each end. e.g. In C-based languages, code blocks deepen with and shallow with . Comment blocks use /* and */ but sadly do not nest. The nature of context specific delimiters cause local lexical analysis to be married to overarching grammar.

1.3.2 Observation 2: Human-readability is about ubiquity

The closer one analyzes the criteria for a format to be considered human-readable, the more it seems to be a misnomer. A romanticized notion might expect a quality reminiscent of ink on paper - that the information exists in such a way that it is 'directly' intelligible in its passive physical form. Obviously, a spatial analog encoding is not amenable (efficiently) to digital computers, and as a result, the convention is to map binary quantities to abstract enumerations of characters. This is very compact and has the benefit of being semantically precise and decouples presentation from content.

The technical definition of a human-readable format, is a bit-stream consisting solely of these character codes. Due to evolving byte-widths, the arbitrarity of character orderings, and the many written languages used around the world, many character encodings have come and gone over the years. To make a long story short, it can be said that Unicode and its ASCII backwards-compatible form, UTF8, represent a major success in satisfying most parties in a unified extensible way. UTF8 is the official character encoding of the Web and is quickly becoming the standard across all of computing.

As a result, the *working* definition of a human-readable format is a string of ASCII or UTF8 that can be displayed by a *text console* or *text editor*. The caveat of having to be displayable in a text editor is to address the details of non-printable characters (control codes / modifiers) and languages that are supported by few fonts. The first 32 ASCII values were originally allocated as abstract control signals with no printable characters associated with them.

As stated in The Art of Unix Programming, "the transparency and interoperability benefits of textual formats are sufficiently strong that most designers have resisted the temptation to optimize for performance at the cost of readability" [6]. When a specific language is built out of characters, a text editor can render it to the point of legibility without an awareness of the higher-level meanings. Some examples of human-readable formats include: TXT (text), CSV (comma separated values), RTF (rich text format), SGML (standard generalized markup language), HTTP (hypertext transfer protocol), HTML (hypertext markup language), JSON (javascript object notation), XML (extensible markup language), TeX (the typesetting system).

The critical observation is that software infrastructure is heavily involved in supporting the human-readability of text. It is not the case that the bit sequences of UTF8 or any other text encodings are somehow intrinsically understandable to a human. An application interprets the bytes as character codes as per a known standard, which are mapped to glyphs in a font, and rendered to a grid of pixels. This chain of interpretation and transformation starts with clusters of electrons and ends with clusters of photons before the human nervous system takes over. The point being that there is still a necessary software layer performing a transformation in the middle.

e- $\Longrightarrow bits \Longrightarrow charactercodes \Longrightarrow glyphs \Longrightarrow pixels \Longrightarrow photons$

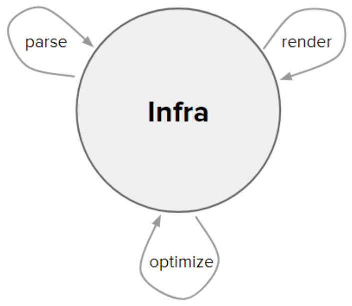
“Human readability” just colloquially implies that it is a standard encoding understood by most text editors. The sense of inherent readability merely comes from the ready availability of tools that render ASCII and Unicode. One can assume that a text editor or some text rendering infrastructure exists in the target system. Therefore, any encoding could technically achieve the same ‘human readable’ status as ASCII if it and its editors were general-purpose enough to warrant an equally ubiquitous install base. Thus, there is an opportunity to expand or upgrade the realm of what can be considered human readable.

The following sections introduce a candidate encoding beleived to be worth of such a pursuit. The graphical user interface revolution of the 70s and 80s made bitmapped displays common. Flat-text infrastructure is the most entrenched aspect of computing, and it did not undergo the revolution it could have along with the GUI. Computing is no longer technologically constrained by purely textual typewriters and teletypes as the common denominator for input/output convenient to human users.

Chapter 2

Infra: A New Infrastructure

Infra aims to make data more powerful and easier to deal with for both humans and computers. All types of data can be viewed, edited, processed, transferred, and stored entirely in Infra. Therefore, developers, runtimes, and end-users could theoretically share a common foundational medium across the computing landscape.



Infra is composed of a novel encoding and a novel type of editor/browser. These two components are intended to supplant the use cases of text encodings and text editors respectively, and since the encoding is compact binary, it also addresses the needs of transfer formats. Infra editors make reading and writing Infra's binary metaformat simple for end-users, and can even style the presentation and taylor editing in response to metadata, resembling a Web Browser or IDE. Beyond the common metaformat features, Infra's encoding includes three critical primitives: Metadata, Free, and Patch.

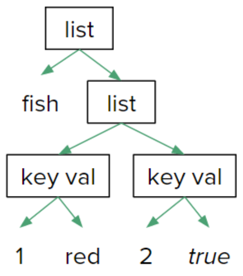
Metadata allows any data element, including other metadata, to be decorated with arbitrary Infra information to add context. For example, metadata is useful for providing IDs to support referencing values by name, style markup to assist presentation in an editor, or schema/type info to constrain or validate data.

Free allows encoded information to contain unallocated memory regions. This can be useful for aligning data to fixed-widths or improving the efficiency of localized edits to large structures on disk.

Patch elements are programs that can inline another Infra object and optionally modify the shallow copy, forming a generalization of graphs. This primitive turns out to be a powerful building block toward general computation in the domain of data metaprogramming. In many situations, Patches and the Infra-encoded statements within them, can be conveniently authored indirectly via programming by demonstration.

The Infra encoding is designed to be an end-user-authorable universal metaformat. It is schemaless like text, yet binary like RPC formats. Currently, plaintext is used as a medium for *indirectly* supporting the building blocks that computing needs, rather than using one that offers those essential building blocks directly. Infra is about providing a new higher-level baseline medium to simplify the infrastructure of computing. This is based on the above observation that the quantitative nature of encoding structure in software for a processor, and the qualitative nature of encoding structure through a teletype for humans, can actually be unified now that typewriters have been out of the picture - as they have been for several decades.

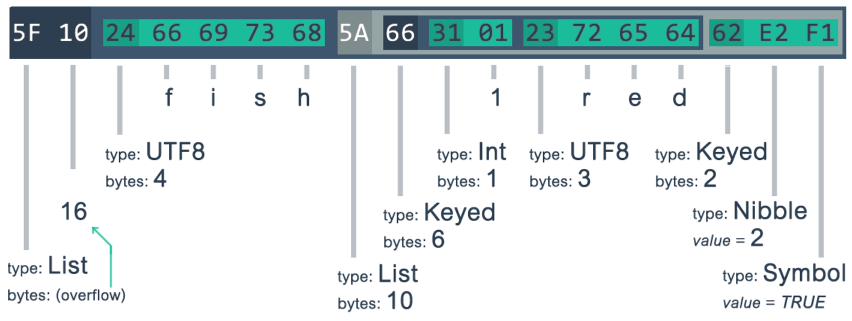
As a preliminary introduction to our alternative infrastructure, the following is a side-by-side comparison of some structured data expressed equivalently in four different languages - three textual and one Infra. This data is a toy example engineered to exercise a range of element types. On the left is the abstract syntax tree of the structure.



JSON	["fish", {"1":"red", "2":true}]
Python	['fish', {1: 'red', 2: True}]
Lisp	('fish ((1 . 'red) (2 . t))
Infra	fish 1:red 2:True

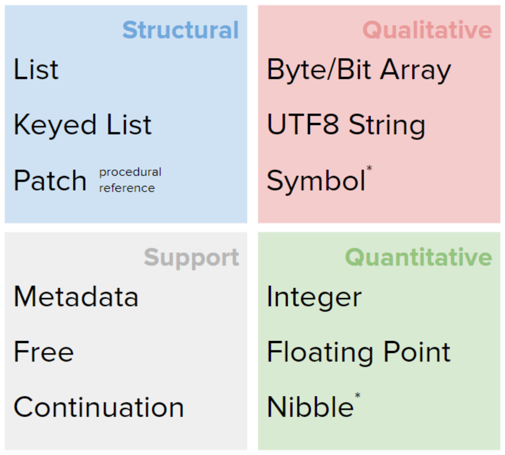
Infra provides structured scaffolding for holding data, but it does not attempt to invent a new character encoding, so 'fish' and 'red' are encoded as UTF8 strings. On the other hand, 'True' is directly encoded as a boolean value, and is shown in blue. Typical formats communicate structure passively using characters such as '[' and '('. Infra communicates the abstract syntax present in the data actively, using general graphical elements. For example, the span of lists is indicated by a blue line above the items it contains.

Before we dive into the details, here is the byte encoding for the Infra representation (shown in hexadecimal). A critical component of Infra is its feasibility to be authored straight into this representation intuitively through an editor. Other binary formats can only be rendered procedurally and often require pre-existing schema.



2.1 The Metaformat

Infra's metaformat encoding consists of a nestable sequence of 'segments', each made of a header byte and a body of variable length. The header byte indicates the type of the segment and the length of its body. This pattern is sometimes called type-length-value, tag-length-value, or key-length-value. Due in part to its simplicity and processing efficiency, type-length-value is a common paradigm, used by many binary file formats such as Portable Network Graphics (PNG) [17], Audio Video Interleave (AVI), Matlab's MAT, Protocol Buffers [18], MessagePack [19], and most modes of the complex ASN.1 [20] format. The finer details vary among these formats, but the most significant decision that differentiates the utility of these formats is the set of first-order primitives, or base types, that they define.

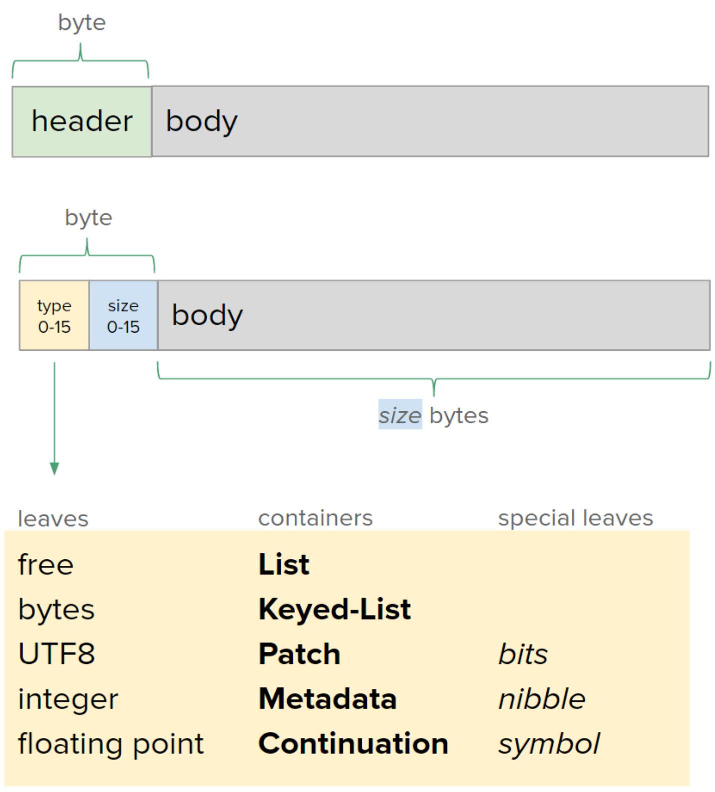


Structural	Qualitative
List	Byte/Bit Array
Keyed List	UTF8 String
Patch procedural reference	Symbol*
Support	**Quantitative**
Metadata	Integer
Free	Floating Point
Continuation	Nibble*

After a wide survey of primitives appearing in programming languages and metaformats, we carefully define 13 base types, which support a flexible combination of direct authoring, processing efficiency, and extensibility. Each of these will be defined in detail in the next chapter. The most disruptive of these base types are *Metadata* and *Patch*. They will come up throughout the various sections, and Patch has its own chapter in this document.

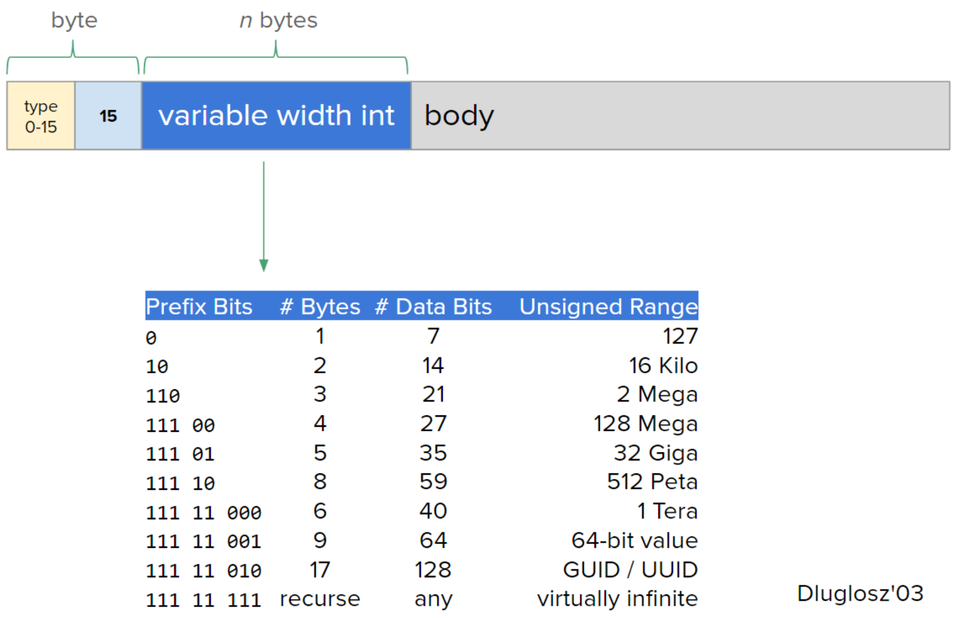
2.1.1 Header Format

Only half a byte is needed to account for 13 base types (plus 3 unallocated). Segment headers can be a single byte when body lengths are no longer than 14 bytes. When a body length is greater than 14 bytes, additional bytes must be used to indicate the length.



leaves	containers	special leaves
free	**List**	
bytes	**Keyed-List**	
UTF8	**Patch**	*bits*
integer	**Metadata**	*nibble*
floating point	**Continuation**	*symbol*

Body length 15 is reserved to signal a multi-byte header mode where the length is instead encoded using a variable length unsigned integer encoding. We find Dluglosz' encoding [21] to be efficient and well designed for this purpose. This is the encoding used for VLIs in the ZIP2 format.



Prefix Bits	# Bytes	# Data Bits	Unsigned Range
`0`	1	7	127
`10`	2	14	16 Kilo
`110`	3	21	2 Mega
`111 00`	4	27	128 Mega
`111 01`	5	35	32 Giga
`111 10`	8	59	512 Peta
`111 11 000`	6	40	1 Tera
`111 11 001`	9	64	64-bit value
`111 11 010`	17	128	GUID / UUID
`111 11 111`	recurse	any	virtually infinite

2.2 The Editor

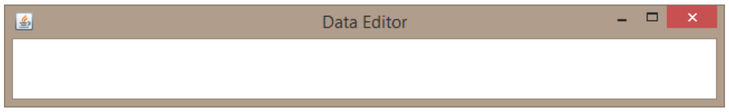
With Infra, our first priority is supporting interactive authoring by end users. An Infra editor aims to fill at least the same breadth of role in computing that text editors and text-field widgets currently play, ultimately superseding them. Like a text editor, which tries to be as adept and general-purpose as possible when it comes to enabling users to view and manipulate a buffer of character codes, an Infra editor tries to make authoring structured data easy. This includes having mechanisms to facilitate the authoring of spans/hierarchy, quantities, references, metadata, and padding.

Using an Infra editor feels like using a text editor. Unlike text editors, however, Infra editors have the opportunity to add useful structure as users type. Users can type their intended structure along with their content. For example, pressing ‘spacebar’ between words defaults to tokenizing the text into lists of strings. Furthermore, recognizable fields such as numbers can be parsed on-the-spot and converted into the appropriate Infra element type (such as Floating-point, which is binary-encoded).

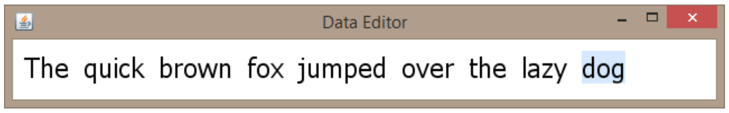
2.2.1 Editor Demo

This brief demonstration will introduce the general idea of using a free-form paradigm of editing to author structured information of any kind. It is not meant to explain everything upfront, but just to provide some intuition for going into the later sections.

Here is an editor window open to an empty document.



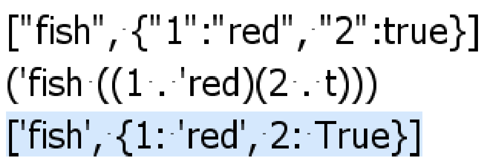
Let us type "The quick brown fox jumped over the lazy dog"



The quick brown fox jumped over the lazy dog

We have just authored a list of nine UTF8 encoded strings. The spacebar naturally tokenized our sentence as we typed it, rather than writing a space character like it normally would in a text editor. Since the cursor is at the word level, pressing delete will delete an entire word. We will press it nine times to get back to an empty document.

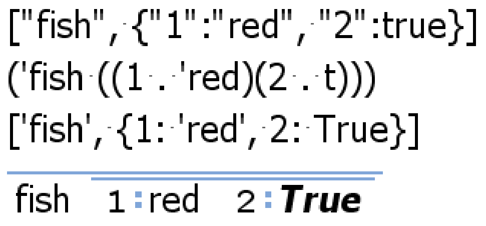
Now, let us paste in the text from the human-readable-format comparison example we used before. These three strings all represent the same (equivalent) data structure as represented in JSON, Lisp, and Python respectively. (Note: The frame of the application window will be omitted from screen shots from this point on.)



```
["fish", {"1":"red", "2":true}]
('fish ((1 . 'red)(2 . t)))
['fish', {1: 'red', 2: True}]
```

This toy data structure is designed to densely cover a range of syntactic elements: Strings, numbers, lists, key-values, and a boolean value. However, in their current (and typical) form, they are *only* made up of character codes. For all of the similarity in their structure, they are far from being interchangeable between parsers because each of these languages makes different choices in how to encode abstract syntax indirectly through character codes.

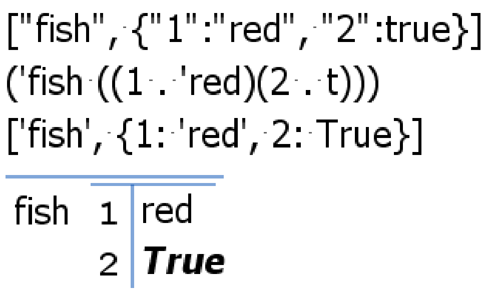
Now we will type a fourth instance of this data, this time using more than just a string element:



```
["fish", {"1":"red", "2":true}]
('fish ((1 . 'red)(2 . t)))
['fish', {1: 'red', 2: True}]
fish  1:red  2:True
```

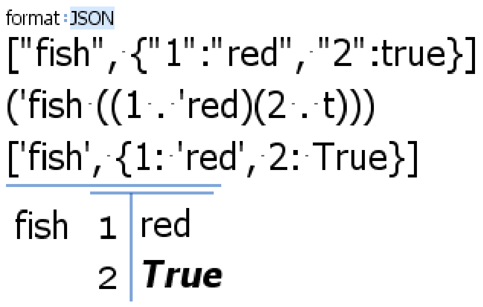
In this case, the values, ‘1’ and ‘2’, are not being encoded as text characters. They are binary encoded integers. And this True has been authored explicitly as Infra’s boolean symbol for true. With Infra, data can stay parsed in this way right from the time of authoring. This means that editor UIs can have richer, more precise, interactive dialog with the author about the content, its parts, metadata associated with its parts, inter-relationships, and its presentation.

Here you can see that the editor is able to layout any subtree as a table:



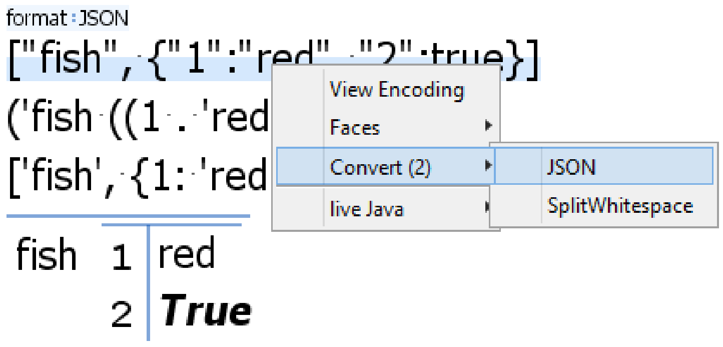
```
["fish", {"1":"red", "2":true}]
('fish ((1 . 'red)(2 . t)))
['fish', {1: 'red', 2: True}]
fish  1 | red
      2 | True
```

Now we will add metadata to the first line to give the editor more confidence that this string is in-fact representing JSON.



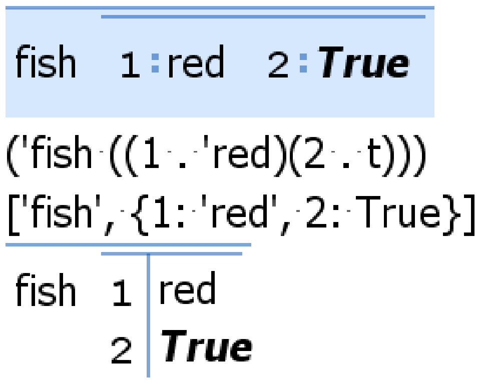
```
format:JSON
["fish",{"1":"red","2":true}]
('fish ((1 . 'red)(2 . t)))
['fish', {1: 'red', 2: True}]
fish  1 red
      2 True
```

This editor happens to have a JSON parser and supports metadata labeled as "format" metadata, so here we can see the editor now offering to parse it as JSON.



```
format:JSON
["fish",{"1":"red","2":true}]
('fish ((1 . 'red
['fish', {1: 'red
fish  1 red
      2 True
```

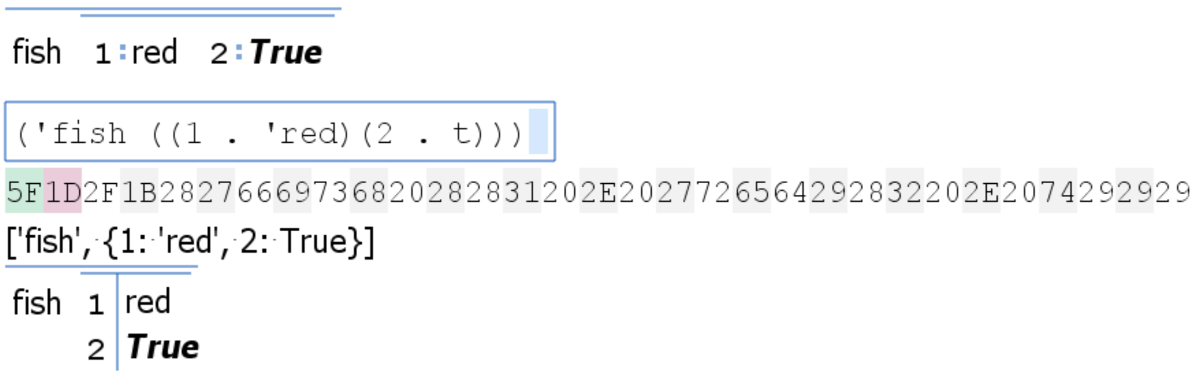
And now it is in parsed form:



```
fish  1:red  2:True
('fish ((1 . 'red)(2 . t)))
['fish', {1: 'red', 2: True}]
fish  1 red
      2 True
```

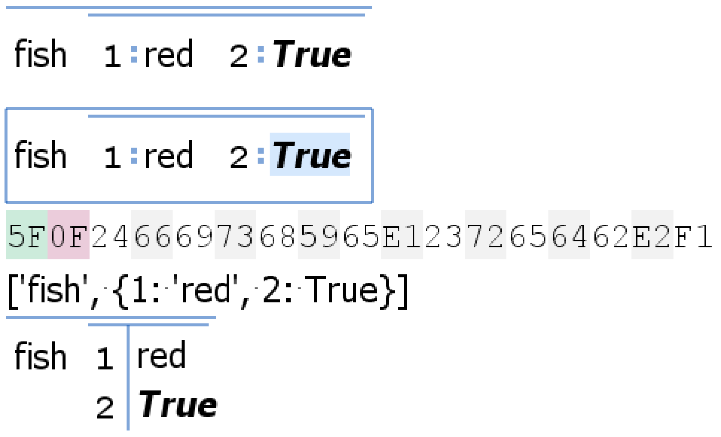
Next, to demonstrate more editing, we will *manually* parse the next one of these string-based representations. But first, we can rig up a side-by-side view of the byte encoding for this object so we can see how the representation changes with each keystroke.

We can drop the cursor down to character-editing level and start chopping the string up into data components.



```
fish   1:red   2:True

('fish ((1 . 'red)(2 . t)))
5F1D2F1B2827666973682028283120 2E20277265642928322 02E2074292929
['fish', {1: 'red', 2: True}]
fish  1 red
      2 True
```

From Infra's perspective, a traditional text editor is just an editor whose cursor is locked down at the character-level, only edits String objects, and has tunnel vision on one string at a time.



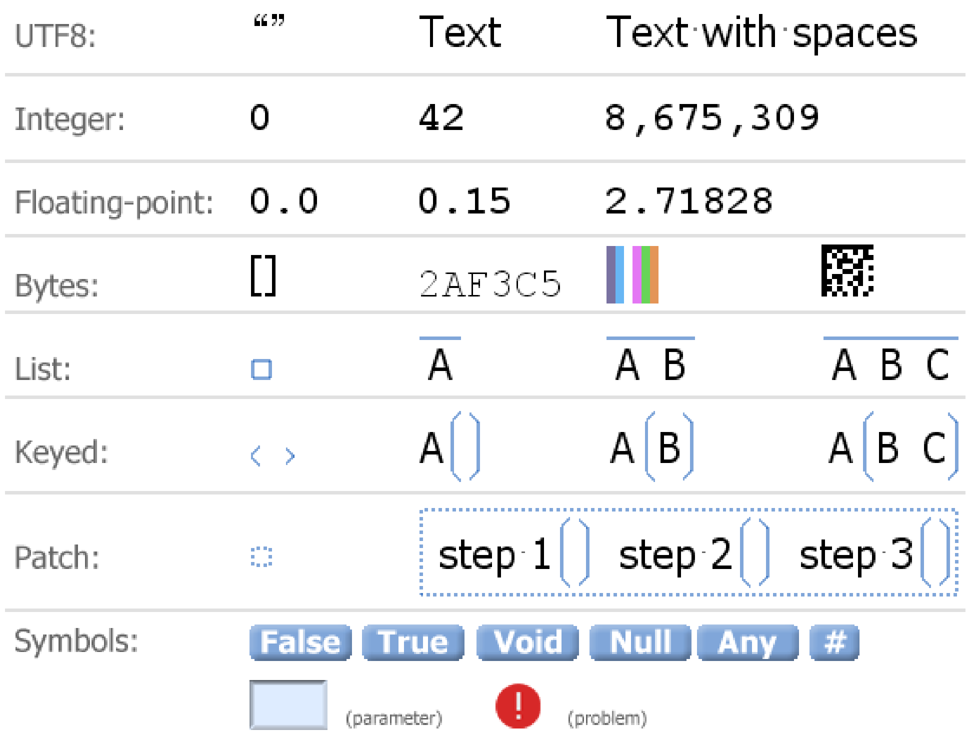
```
fish   1:red   2:True

fish   1:red   2:True
5F0F2466697368596 5E12372656462E2F1
['fish', {1: 'red', 2: True}]
fish  1 red
      2 True
```

Just like raw UTF8, Infra's encoding is easy to make human friendly in editors, and usable by user-interface widgets and libraries everywhere. All while rivaling the machine-readability of binary metaformats such as ASN.1 or Protocol Buffers.

2.2.2 Visualizing Primitives

An Infra editor conveys information structure by displaying Infra's abstract syntax with some sort of visual convention. Each editor can vary in the details, but should at minimum cover basic editing of a hierarchical information structure that can be recursively interleaved with metadata and computed transclusions. (A transclusion is a referentially transparent inlining of content from another location. In the case of Infra, these are combined with a model for computation called 'Patch'.) In the UI for the editor, this amounts to displaying a handful of primitive types in a way that is distinguishable from each other. These include: empty strings, strings with spaces (as opposed to padding between segmented words), integer and floating-point binary encoded quantities, regular List containers, Patch containers, Keyed-list containers, the built-in Symbols, and the Side-Effect objects returned by Patches. (A definition of each is given in the next chapter.)

The following table outlines our prototype editor's approach to visualizing these types. The specific values are just for the purposes of demonstration. The first column of the first six rows exemplifies 'empty' versions of the respective types.



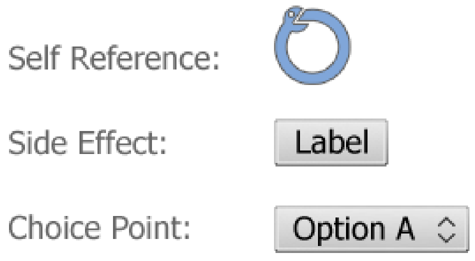
UTF8:	“”	Text	Text·with·spaces	
Integer:	0	42	8,675,309	
Floating-point:	0.0	0.15	2.71828	
Bytes:	[]	2AF3C5		
List:	□	A	A B	A B C
Keyed:	‹ ›	A( )	A(B)	A(B C)
Patch:	⬚	step·1( ) step·2( ) step·3( )		
Symbols:	False True Void Null Any #			
	▭ (parameter)	● (problem)		

Having an explicit difference in base type between numbers and number-like strings also allows for quantity specific pretty-printing such as dynamic localized digit-grouping. Similar to how spreadsheet programs have a convention of left justifying text within a cell and right justifying numeric types within a cell, our prototype editor, renders text and quantities in different font families. Quantities default to a monospaced font, while strings do not.

The span of a List can be conveyed in a human-readable way with a simple overline. When contents are displayed vertically, or must line-wrap, a more complete outline is drawn around the items.

Byte arrays can be displayed in many creative ways. Since a user is rarely inspecting long binary values digit-by-digit, compact views are often most useful. Projects such as Data Matrix [22], Chroma-Hash [23], Mnemonic Encoder [24], and PGPfone [25] provide a number of useful lenses that an Infra editor could support for a user to toggle between, each with different use cases and advantages. The figure above depicts hexadecimal, a variant of Chroma-Hash, and Data Matrix as the main options in our prototype editor. The Chroma-Hash type approach is useful for making binary sequences quick to compare visually. The Data Matrix approach is highly compact with a consistent form factor, playing the useful role of signaling the type of primitive all while abstracting the details of the value from a human user and staying technically readable.

Patches can evaluate to three additional types of first-class value: self reference, side effect, and choice point. These values only occur at runtime and are only encoded indirectly through Patch instructions, but they still must be accounted for in the editor user-interface.

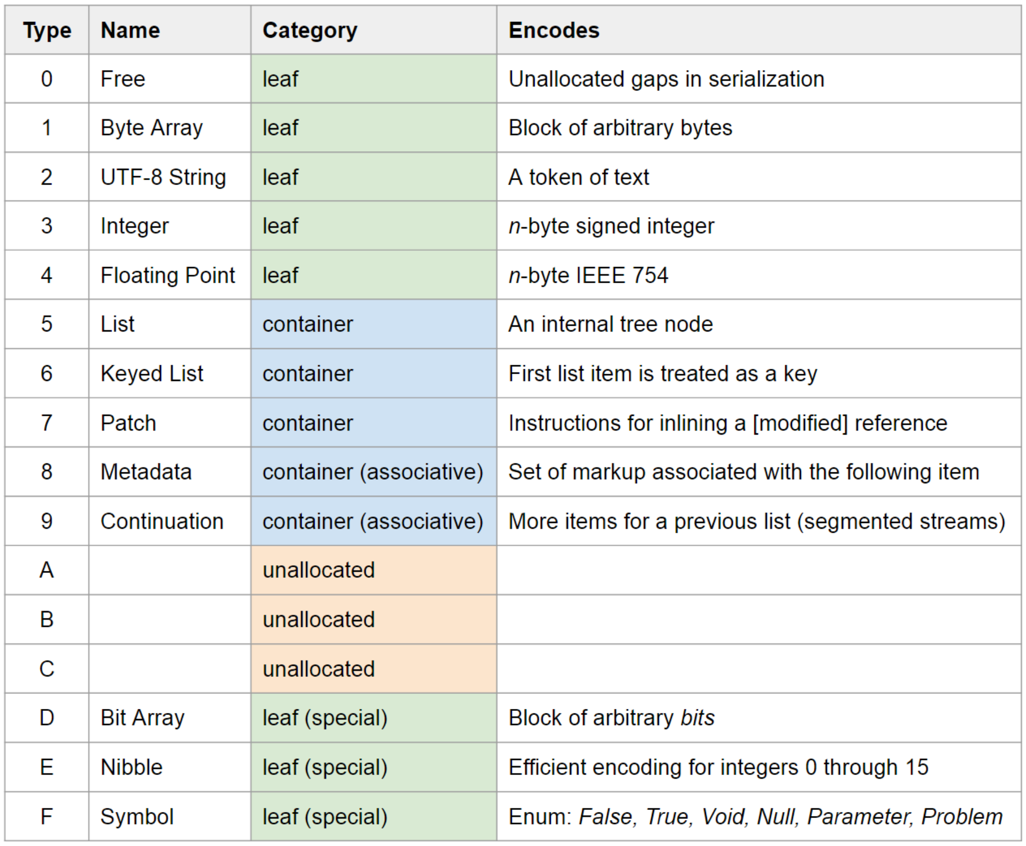


When a Patch node has no value due to a circular self-reference situation, a simplified depiction of Ouroboros (an ancient symbol for cyclical interconnection), resembling a circular arrow is displayed as a place-holder. Its ideal to display Side-Effect objects (returned by Patches that attempt mutation) as interactive buttons because they require some form of synchronous event to trigger them. These buttons can display the label provided by the Patch to elucidate the nature of its side-effect to the user. See the chapter on Patch for details on Patch, Self-Reference, Side-Effect, and Choice-Point semantics.

Additional niceties for an editor to support include things like a CSS implementation for reacting to CSS-labeled metadata to style the document, a suite of abstractions for common high-level type descriptors, a library of uniquely identifiable objects that can be referenced for things like operating system integration, or Programming-By-Demonstration and Programming-By-Example engines to help a user intuitively author computed transclusions. Our prototype implementation includes each of these to a degree. They allow the Infra medium to scale up to the roles of Web Browsers, application user-interfaces, and integrated development environments (IDEs).

Chapter 3

Base Semantics and Interaction



Type	Name	Category	Encodes
0	Free	leaf	Unallocated gaps in serialization
1	Byte Array	leaf	Block of arbitrary bytes
2	UTF-8 String	leaf	A token of text
3	Integer	leaf	*n*-byte signed integer
4	Floating Point	leaf	*n*-byte IEEE 754
5	List	container	An internal tree node
6	Keyed List	container	First list item is treated as a key
7	Patch	container	Instructions for inlining a [modified] reference
8	Metadata	container (associative)	Set of markup associated with the following item
9	Continuation	container (associative)	More items for a previous list (segmented streams)
A		unallocated	
B		unallocated	
C		unallocated	
D	Bit Array	leaf (special)	Block of arbitrary *bits*
E	Nibble	leaf (special)	Efficient encoding for integers 0 through 15
F	Symbol	leaf (special)	Enum: *False, True, Void, Null, Parameter, Problem*

The table above is a comprehensive listing of Infra's base types. This chapter will define the details of each as well as their unique relationship with human readability in an Infra editor.

The order in which they are covered in the following sections follows a progression that ensures examples only build on types that have been covered up to that point. The Patch base type is covered in its own chapter due to the number of subtopics it elicits.

These base types are what can appear at the encoding layer of the interpretation hierarchy, but four of these types are strictly for controlling degrees-of-freedom in the encoding and no not appear as literal data elements in the logical data tree.

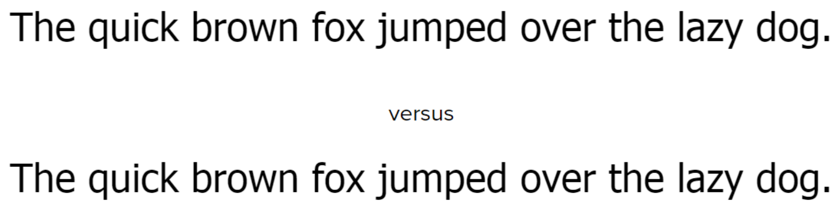
Element Type	**Encoding Layer**	**Tree Layer**	**Graph Layer**
Byte Array	✓	✓	✓
UTF-8 String	✓	✓	✓
Integer	✓	✓	✓
Floating Point	✓	✓	✓
List	✓	✓	✓
Keyed List	✓	✓	✓
Bit Array	✓	✓	✓
Symbol	✓	✓	✓
Patch	✓	✓	Transparent
Nibble	✓	Integer	
Metadata	✓	List	
Free	✓		
Continuation	✓		
Choice / Union / Enum			✓ Patch
Side-Effect Object			✓ Patch
Native-Service Object			✓ Patch

Since Patches are referentially transparent, they do not appear at the evaluated /

graph layer of interpretation. As a result, Patches are also the gateway for otherwise non-encodable types.

3.1 Base type: UTF-8

The UTF-8 base type marks a string of UTF8-encoded text. These strings can have any length, including zero. A series of UTF-8 segments naturally forms a tokenized sequence of text. In an Infra editor, the default margin between items in a layout is the width of a would-be space character. This results in a parity between the display of a traditional text buffer in a text editor, and the display of tokenized text in Infra.

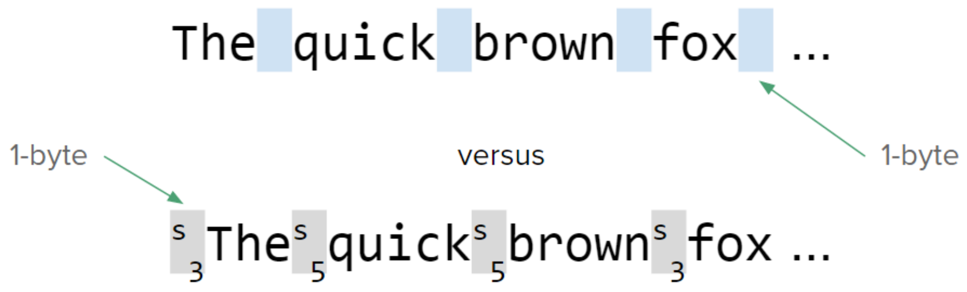


The quick brown fox jumped over the lazy dog.

versus

The quick brown fox jumped over the lazy dog.

As you can see above, a side-by-side comparison can be identical though the byte encoding is of a parsed paradigm. In an Infra editor, a UTF-8 segment is authored simply by typing characters as usual. The characters are appended to the selected element and its size field is updated accordingly. Note that this kind of editing technically requires coordinated changes in multiple places in the buffer with each insertion/deletion to keep element sizes in sync. This is not something that was viable with typewriter-based input/output. Pressing the spacebar key creates a new segment, such that normal typing naturally results in tokenized sentences. Our prototype used shift+space to type a literal space character (without creating a new segment).



The byte overhead of per-token headers is made up for in most cases by the freedom to forgo most delimiters and whitespace characters that would normally be necessary between tokens. In the case of strings less than 15 bytes long, the single-byte preamble overhead is exactly equivalent to that of a C-style string with a trailing zero-byte terminator, but Infra strings are unambiguous to parse or skip because the length is known up front.

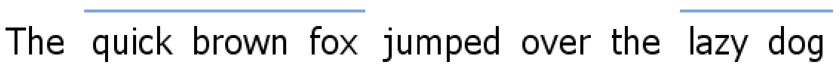
This means that Infra can be used to encode language expressions in tokenized form, which bypasses the typical parsing constraints that are imposed on identifiers and the like. For example, variable names in various programming languages cannot usually contain spaces or symbols due to the parsing ambiguities they would cause, but these limitations are endemic to flat text. Infra, as an infrastructure, can shift most parsing steps to the time of authoring. Not only does this lift character constraints on things like identifiers (including spaces), but it also allows user interfaces and rich assisted interaction to take place within the task of communicating structured information.

Rich-text styling can be supported via metadata. We will visit this topic in the section on Metadata.

3.2 Base type: List

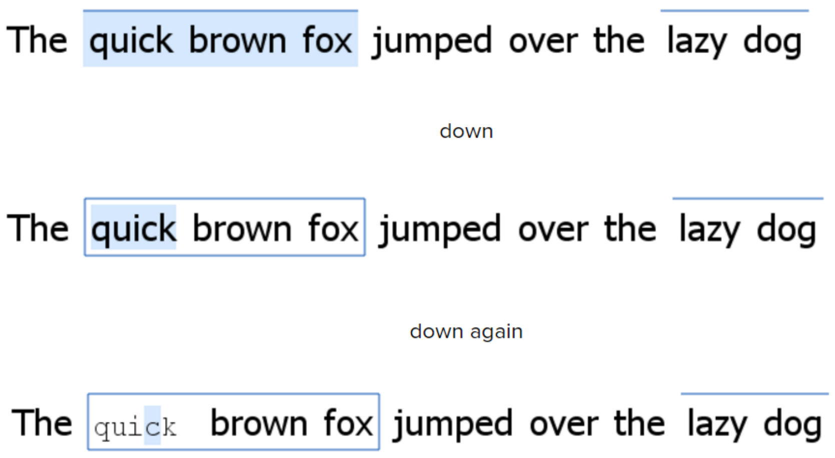
A list is a container to group zero or more data structures together. Lists can contain elements of any type, including other lists. General trees can be built using lists of lists.

Unlike text editors, which edit flat character arrays, Infra editors are designed to work well with hierarchical data. In our prototype editor, lists are represented simply as a line spanning over the items it contains.



In the above example, 'quick', 'brown', and 'fox' are grouped together in a List. 'lazy' and 'dog' are within another List. 'The', 'jumped', 'over', and 'the' are at the root level.

The selection cursor can also be hierarchical in order to edit at any level of granularity present in the data's structure. In addition to moving the cursor between siblings, a central user-interface action is to move the selection down to a child or up to a parent container.


The quick brown fox jumped over the lazy dog
down
The quick brown fox jumped over the lazy dog
down again
The quick brown fox jumped over the lazy dog

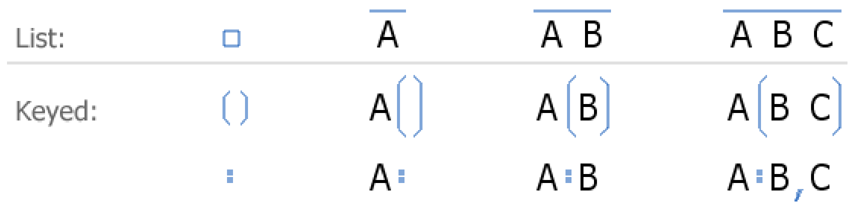
In the above figure, the top row depicts selection of the second element as a whole. Moving the cursor *down* or *in* results in the second row, where 'quick' is selected, and moving the cursor to the right would now select siblings of 'quick' as opposed to siblings of the List (i.e. 'jumped'). As seen in the third row, the cursor can be moved *in* again to operate at the character level in the familiar way.

It is worth noting that any tree / sub-tree can be displayed using a grid-like layout, forming a column-aligned table. (See section **??** - Alternate View Faces: table-view)

3.3 Base type: Keyed List

Keyed List is a variant of **List** that associates the first child with the container itself. This is similar to the concept of a key-value pair where the first child is taken as the key, except Keyed Lists can have any number of values, like an n-tuple. Keyed nodes are also similar to Lisp's S-Expressions and are used to encode Patch instructions, which we describe further in the chapter on Patch. Infra's Keyed Lists are different than the concept of "Keyed Lists" in the Tcl programming language, which are lists containing key-value pairs.

Our prototype editor displays Keyed Lists in either of two visual styles: a parentheses-like style or a colon-like style. Since it is only a presentation layer decision, a user can spontaneously switch at any time. The following figure compares the visual differences between Lists and Keyed Lists of zero through three items respectively.



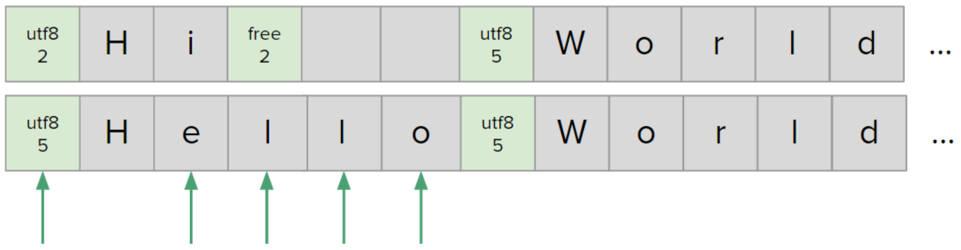
We find that the colon-like style is the more appealing for when there are exactly two items in the Keyed List (including the key), i.e. A:B. But we find that the parentheses-like style is generally less visually ambiguous for cases of fewer or greater than two items total. (To be clear, such ambiguities would be strictly user-interface-level issues, not encoding-level ambiguity.) In our figures, you will find that we mix the use of the two styles for best readability on a case-by-case basis.

3.4 Base type: Free

Free segments are placeholders that span unallocated bytes. They do not contribute data to the data tree, they merely allow the serialized encoding to have padded gaps. This is analogous to the function of a *free list* in dynamic memory allocation, which tracks the unallocated byte ranges of a runtime heap. It was deliberately chosen to be the type value zero so that any zeroed byte buffers parse as valid Infra. A header value of 0x00 means a **Free** segment of length 0. As a result, one way to delete a segment entirely is to simply zero out its binary representation. To our knowledge, no other metaformat natively supports this concept at the byte level. Byte-level support is critical in order to manage single byte gaps.

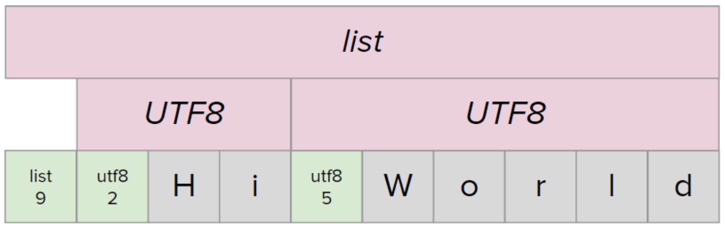
Files storing Infra encoded data do not have to be streamed in and parsed like textual formats do. The element headers allow jumping through the tree. If only one element of a large file needs to be updated or read, only the segments on the path to that node would have to be examined. They do not even have to be kept in memory. Then, the element can be changed by writing directly into the file in the correct location. If its byte-size is the same, the update is trivial. If the size is shorter, the gap between it and the rest of what was in the file can be filled with a **Free** segment. If the change results in a size *increase*, hopefully there is an immediately adjacent **Free** segment to pull from. Nearby **Free** segments can be strategically inserted and maintained by the file writer or heap manager in advance, based on predicted usage. When **Free** segments are nearby, but not quite adjacent to an expanding write location, they can be pre-shifted into place with a number of writes proportional to the tree depth of the segment. Clever use of the **Free** segment allows enough efficiency to support a live file editing mode where the Data Editor continuously saves edits as they are made, including to very large files. If the serialized encoding were held in RAM rather than disk, it would be immediately on-hand for writing to a file or sending over a network.

Furthermore, free segments can be used on an in-memory representation to improve runtime efficiency. Segments can be padded to ensure that they always align to regular intervals. Or the elements of a particular list can be padded to a homogeneous length to enable calculating its direct offset with multiplication.



Metadata can be used to specify the ideal pre-allocations of free space that should be included in the serialization. See section 6.2 for more on this.

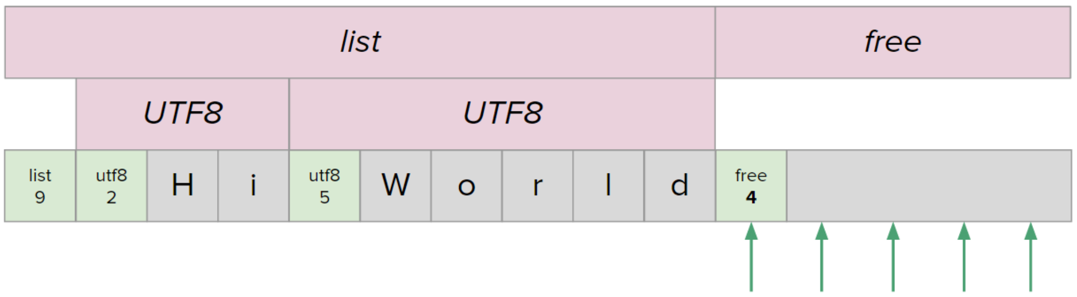
The following is an example of making a modification to the encoding that requires extending the value of a leaf node while rigorously maintaining parsability between every atomic mutation. Certain steps of the following algorithm can be skipped if there is no concern over critical interruptions such as a power outage or system crash. We assume that overwriting a single byte is atomic. Allocating and shifting memory can be done in fewer steps when not requiring this level of rigor.



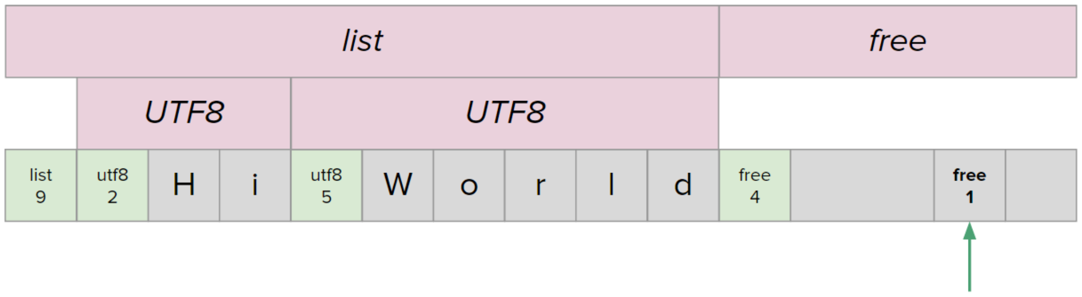
Currently encoding: Hi World

Goal encoding: Hello World

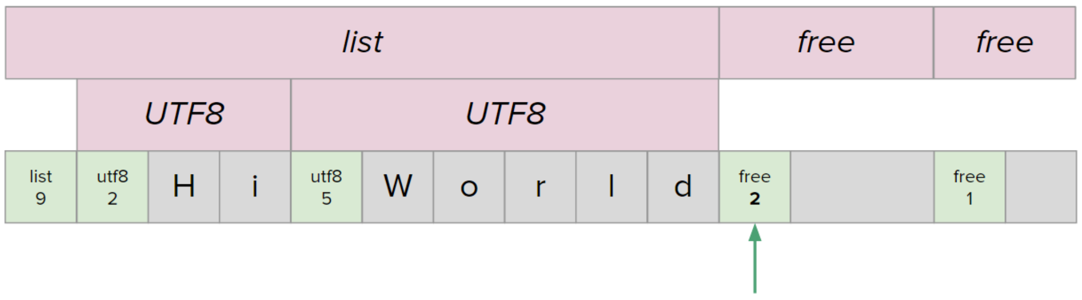
1. Extend the buffer with free space. This can always be done easily at the end of a file. We only need to allocate three more bytes for our target, but we allocate more for the purpose of demonstration.



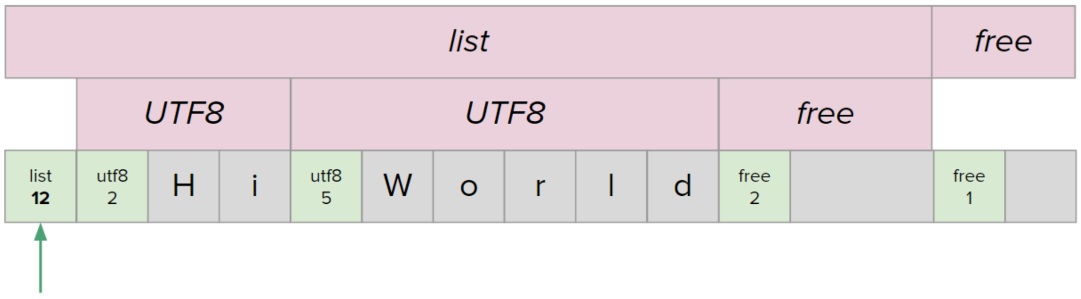
2. Prepare to slice the needed amount from the free pool.



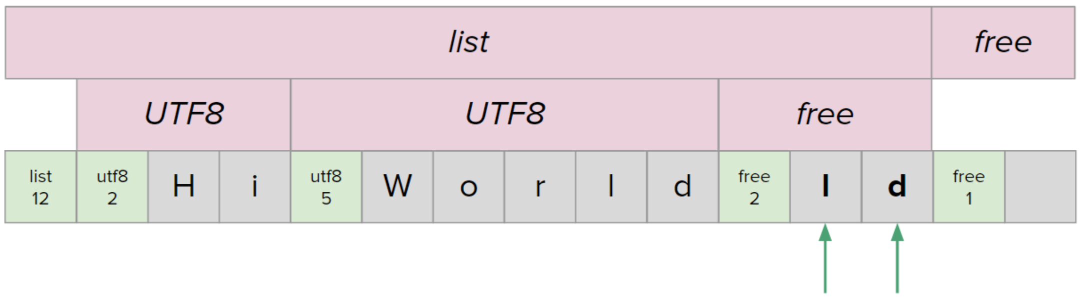
3. Slice. The encoding is valid before and after updating header byte value.



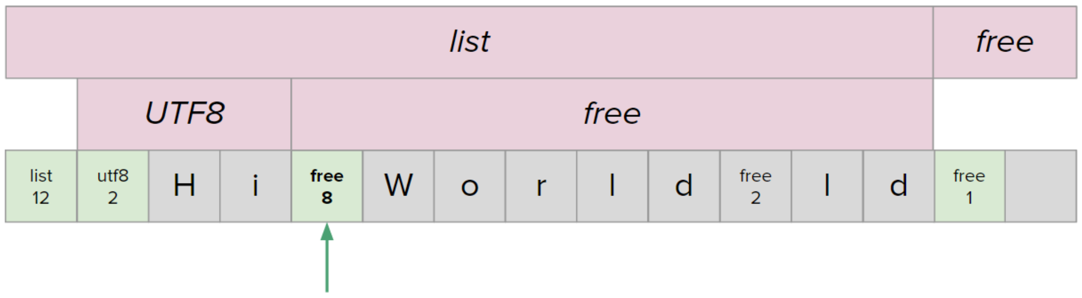
4. Move allocation into the scope of the adjacent list by changing its span.



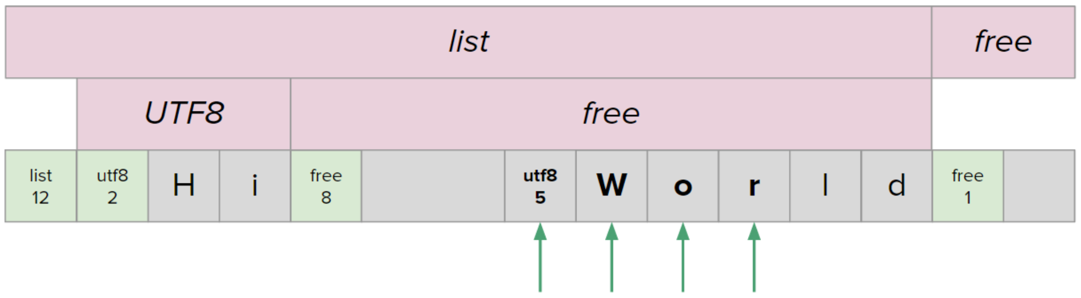
5. Prepare to shift 'World' down. This minimizes the amount of time spent in step 7.



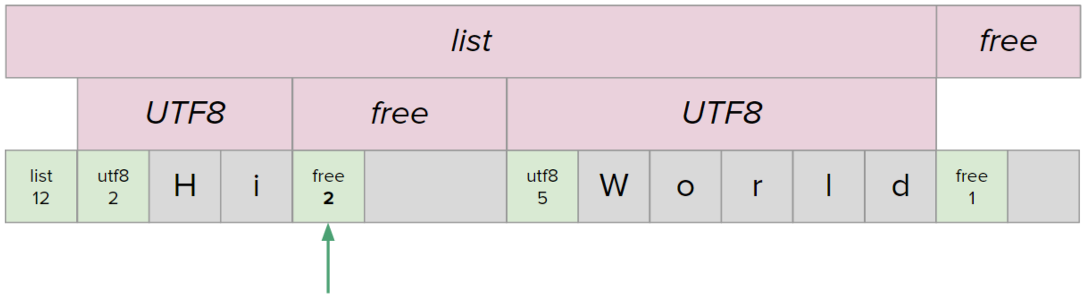
6. Temporarily disable this region. This allows the encoding to stay parsable during modification, but does risk a string being dropped from the tree if the process gets interrupted at this point.



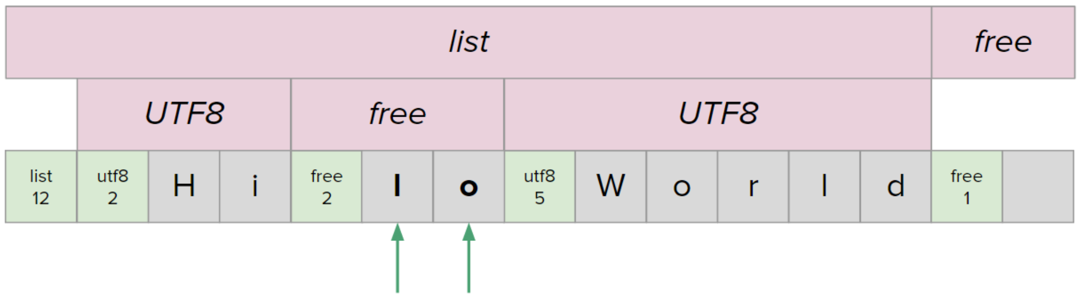
7. Perform the remainder of the shift while the region is disabled.



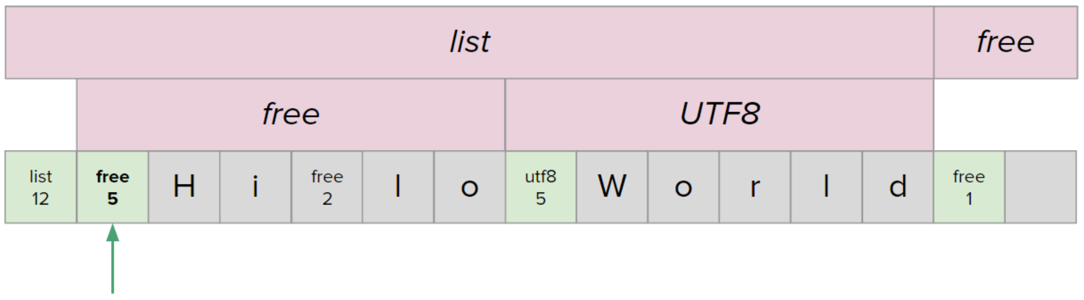
8. Re-enable region. The three allocated bytes are now swapped over to its left side.

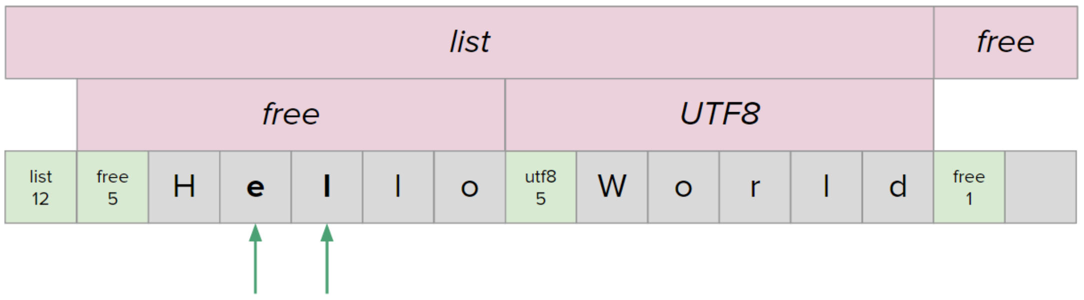


9. Prepare for the extension of our target string to ‘Hello’.

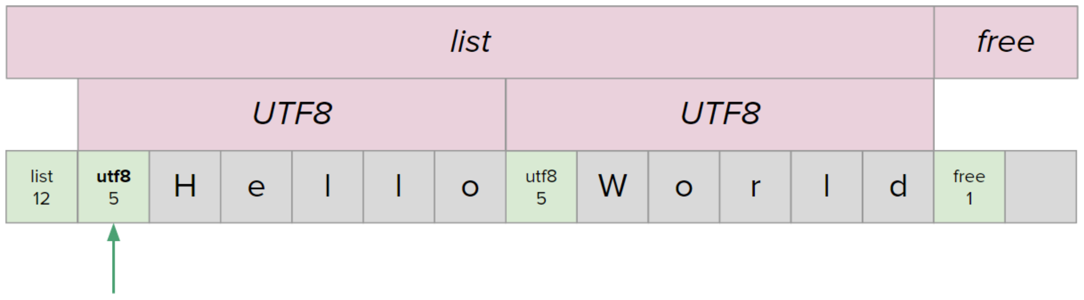


10. Disable region for string update.





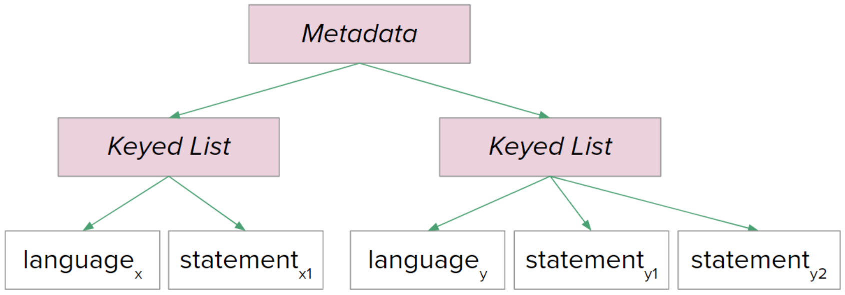
11. Re-enable string at new length.



3.5 Base type: Metadata

In Infra, metadata can be associated with any element, including other metadata. Metadata can be any data, but should describe or expound upon the data to which it is associated. The Metadata base-type provides a capacity to carry information that is not necessarily accounted for in the data's schema. The addition of metadata will never disrupt or confuse a parser that only looks at the data, since metadata is unambiguously peripheral.

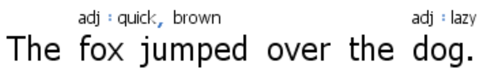
Metadata elements are keyed with a language identifier. This not only provides a means to anchor interpretation of the metadata to some recognizable semantics, but also to allow any number of different kinds of markup to coexist on the same data node.



The language and statement branches of a metadata tree can of course be any data structure. Therefore, the language name can be a flat string or an elaborate structure, but should be qualified enough as to be globally uniquely. We expect a common convention to be using a list containing the segments of a namespace hierarchy, *i.e.* what a parsed version of a domain name or Java package name would look like after being split on the period character. See section 7.1 to see this kind of pre-parsed freedom applied to domain names in the context of URLs.

When it comes to display in the editor, metadata can be laid out in various ways or selectively hidden. In our prototype editor, metadata is shown in a smaller font over the element it is associated with. Another option is to display the metadata for the currently selected element in a side panel.

In the following example, metadata has been authored onto the strings ‘fox’ and ‘dog’. The metadata values are keyed as ‘adj’ markup using Keyed Lists.



This particular example is equivalent to viewing / authoring the following HTML in a text editor (with HTML-specific syntax highlighting):

```
The <span adj="quick" adj="brown">fox</span> jumped over the <span adj="lazy">dog</span>.
```

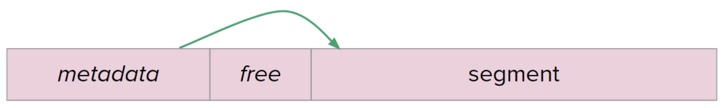
However, this HTML is malformed because repeated attribute tags are not supported. In practice, “quick” and “brown” would have to be combined into one value using a one-off syntax scheme to indirectly retain their boundaries. At that point, the HTML parser, syntax highlighting, and editor assistance stop helping, and custom parsing must be added wherever the values are used. Note that in HTML, attributes cannot themselves also have attributes (no recursive metadata).

Metadata can be used for an endless variety of purposes. These include providing context, self-description, or provenance to data to coordinate its processing across applications. Metadata allows data to accumulate augmentations when they are available, and transport them intact through stages of processing that may not even understand them. Information is generally better able to be kept in one place, such as storing the history of accesses made to a piece of data along side that data, without having to alter its format in the process. Metadata can even be used to enable stateless processing, in the same way that web servers have users store their own cookies so that load-balancing servers can be stateless. Metadata is altogether critical for general extensibility.

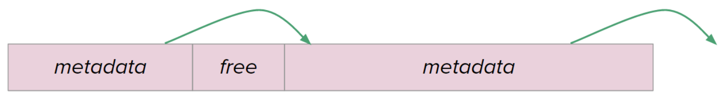
See section 7.2 for an example using metadata for CSS (Cascading Style Sheets) support in an editor.

3.5.1 Matadata Segment Association Rules

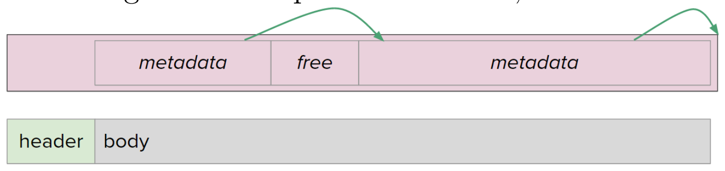
Metadata segments associate with the segment immediately following it in the stream (skipping any segments of type Free).



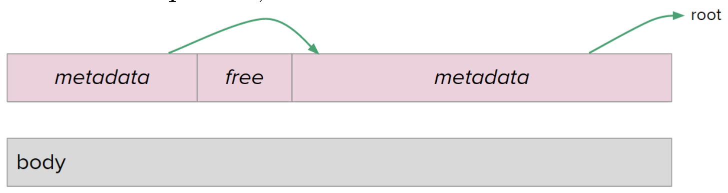
Note that if no metadata exists on a node, then the entire container is simply absent from the encoding. The capacity to have metadata costs nothing if it is not used. Metadata can be associated with other metadata with no issue (meta-metadata).



If the last segment in a span is metadata, it associates with its container.



If it is in the top level, it is metadata for the tree itself.



3.5.2 Standard Metadata Language Channels

There are a number of simple metadata languages we define and support out-of-the-box as part of a 'standard library implementation' of the editor. Their language semantics enrich the system on a number of levels, from low-level control over the encoding to high-level presentation and adaptability. For the sake of this document, the names (identifiers) of the languages appear without being qualified with namespaces. Currently, the standard library defines:

- **ID**(): for specifying the name of a node (can be any Infra tree). Used by Patch's cursor navigation instructions for referencing nodes by name (using lexical scoping rules).

- **UID**(): for registering a node as a global with a name that is unique within the tree.
- **encoding**(): marking numeric types as fixed-width, hierarchically declaring little-endian and big-endian byte-order modes, managing the use of **Free** segments for padding.
- **schema**(): specifying a node's type. Type descriptions are prototype-based and are literally a structure of the default values. This information can be used to reason about and enforce strong type safety. They can inform an editor how to validate a structure and guide edits towards valid configurations. Type schemas can be given to byte arrays for headerless data packing. Schema metadata on a Patch node pertains to the structure of its result.
- **format**(): specifying list display-mode: orientation (vertical/horizontal), table view. Number display modes: digit grouping, significant figures. Node spacing/margins.
- **CSS**(): an *Infra* dialect of CSS - the style sheet language of choice for the web. Support for CSS blurs the lines between Infra editors and web browsers, and position Infra as more efficient and author-friendly alternative to the Document Object Model (DOM). Out prototype currently only supports a subset of CSS semantics.
- **markup markup**(): to mark other metadata as transient (i.e. do not serialize it persistently), or to mark another metadata entry as being critical to understand. Critical to understand means that the library should not proceed with confidence if it knows it does not support this particular markup.
- **comment**(): for general comments - just like code comments, yet these are explicitly associated with the node being commented on.

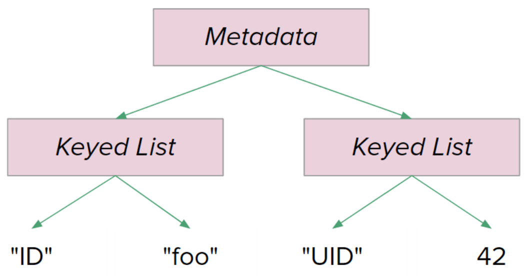
- **closet**(): for housing data just for the sake of referring to it from some other location. We inherit the name from Boxer's similar concept [26].
- **inbound references**(): a log of external references to this data. This can be used to help keep system-wide inter-references between information in sync.

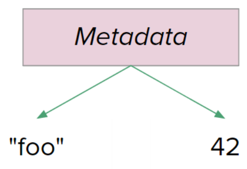
3.5.3 Shorthand for ID and UID metadata

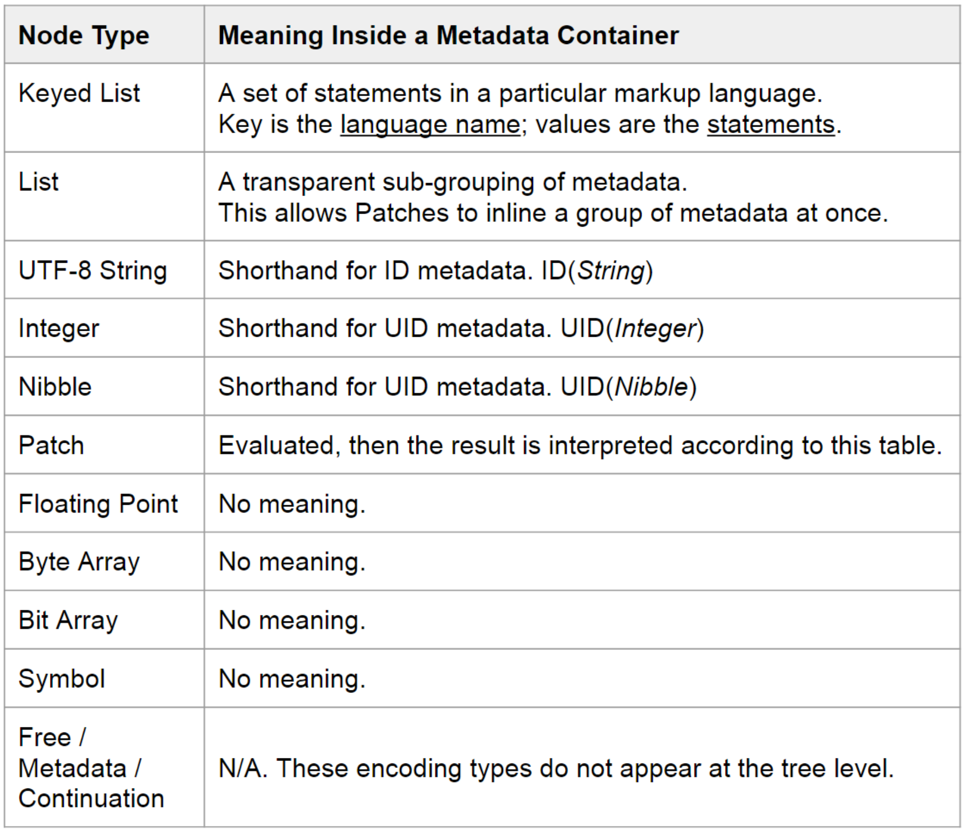
There are two exceptions to the rule of metadata items being Keyed Lists:

- a UTF8 String element found as a direct child of a Metadata node is taken implicitly to be 'ID' metadata, i.e. shorthand for the Keyed List **ID**:*string*.
- an Integer element found as a direct child of a Metadata node is taken implicitly to be 'UID' metadata, i.e. shorthand for the Keyed List **UID**:*number*.

This is nice because using metadata to give a node a name (an ID) is common, and using UIDs to de-duplicate elements for efficient storage is assisted further by this compact form. Thus, the following two Metadata trees are equivalent in terms of meaning:





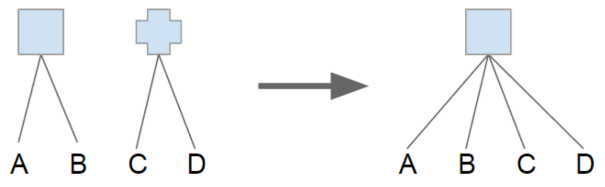


Node Type	Meaning Inside a Metadata Container
Keyed List	A set of statements in a particular markup language. Key is the language name; values are the statements.
List	A transparent sub-grouping of metadata. This allows Patches to inline a group of metadata at once.
UTF-8 String	Shorthand for ID metadata. ID(*String*)
Integer	Shorthand for UID metadata. UID(*Integer*)
Nibble	Shorthand for UID metadata. UID(*Nibble*)
Patch	Evaluated, then the result is interpreted according to this table.
Floating Point	No meaning.
Byte Array	No meaning.
Bit Array	No meaning.
Symbol	No meaning.
Free / Metadata / Continuation	N/A. These encoding types do not appear at the tree level.

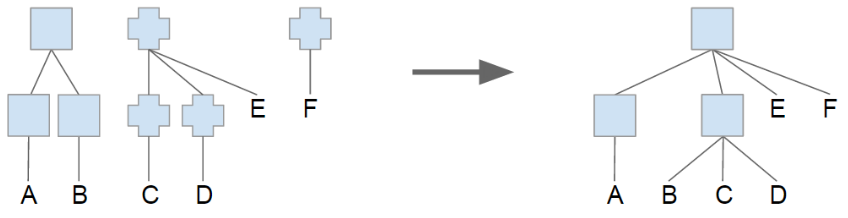
3.6 Base type: Continuation

The purpose of this container type is to enable partitioning of long-length segments into portions that can each remain valid *Infra* trees. This is useful when storing or transmitting a tree that is being generated slowly over time or has unbounded length. Using continuation containers, portions of known size can be associated together into a tree whose total size is not known in advance. **Continuation** is like a concatenation command, concatenating its children to the previous top-level container. If a child of the continuation is itself a **Continuation** segment, then it concatenates its contents to the existing second-level container (whatever its type), and so on. This mechanism allows for any Infra segment to be continued.

The Continuation base type allows an Infra tree to be segmented into smaller portions for streaming while allowing each chunk to be valid stand-alone trees that could be parsed on their own. A Continuation is just the same as a List except its children get merged into the container immediately previous to it in the byte stream. Here is a simple scenario:



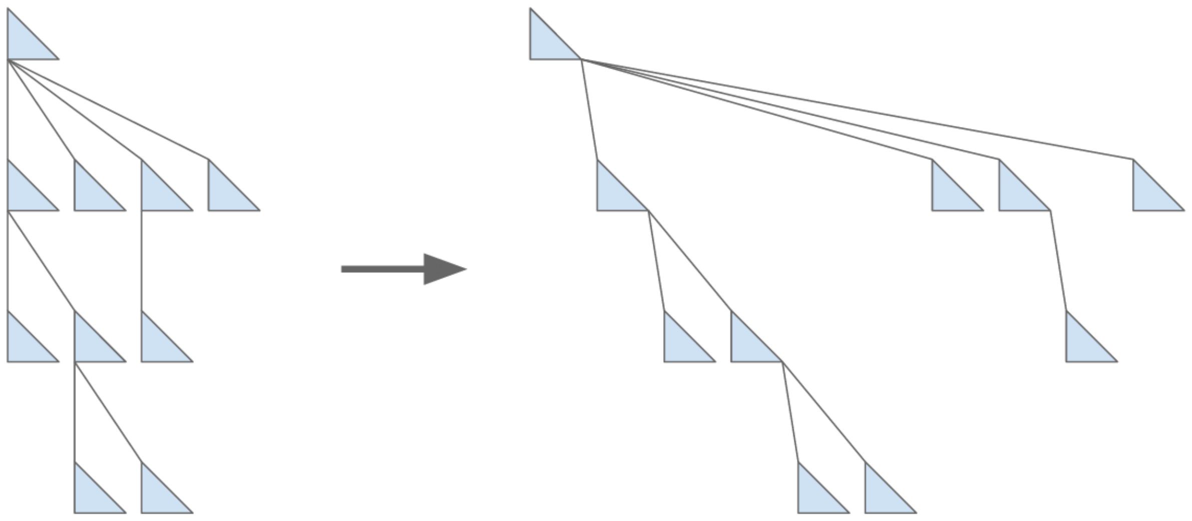
The square represents a List, and the plus represents a Continuation. In this example, the stream contains a List with two items A and B, followed by a Continuation list with two items C and D. On the receiving end, the Continuations's two items get appended to the List, making one list with four items. Here is a multi-tiered scenario:



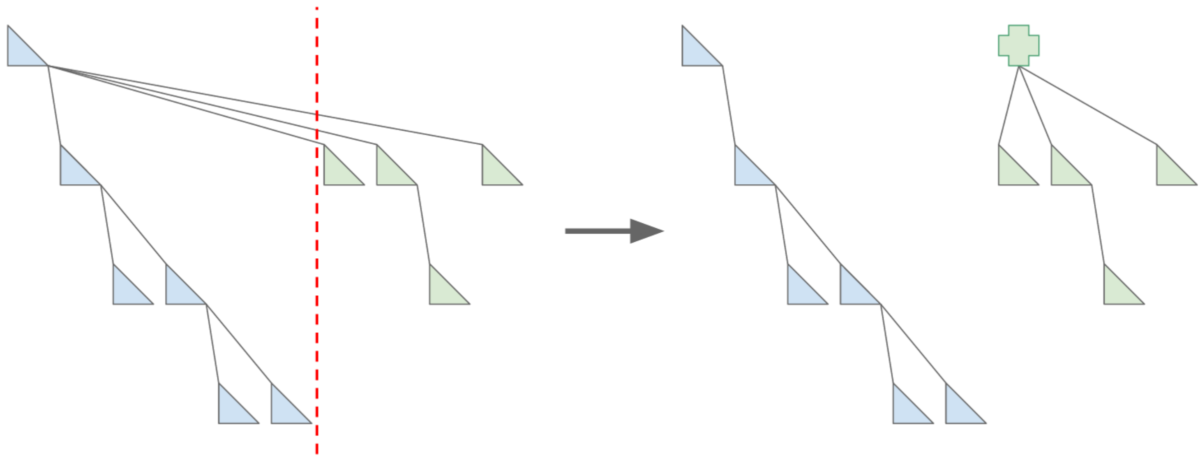
This use of Continuations is contrived for the sake of example. We have the first tree as our base and the second tree 'continues' it, so let us focus on the second tree for now. Within the second tree, there is a Continuation continuing a Continuation. This would be the first to merge as it is parses, so now imagine that D is merged into the list with C. As a result of this property in general, the only Continuation nodes that will remain in a parsed tree are going to be along the 'left edge'. The continuations along the left edge of the second tree line up with the continers on the right edge of the first tree. Merging them hierarchically and then doing the same with the third tree, we get the result on the right. Note that there would be no way to continue the List that contains item A.

It is permanently 'shadowed' by its right sibling, but its okay, the tree has moved on to newer things.

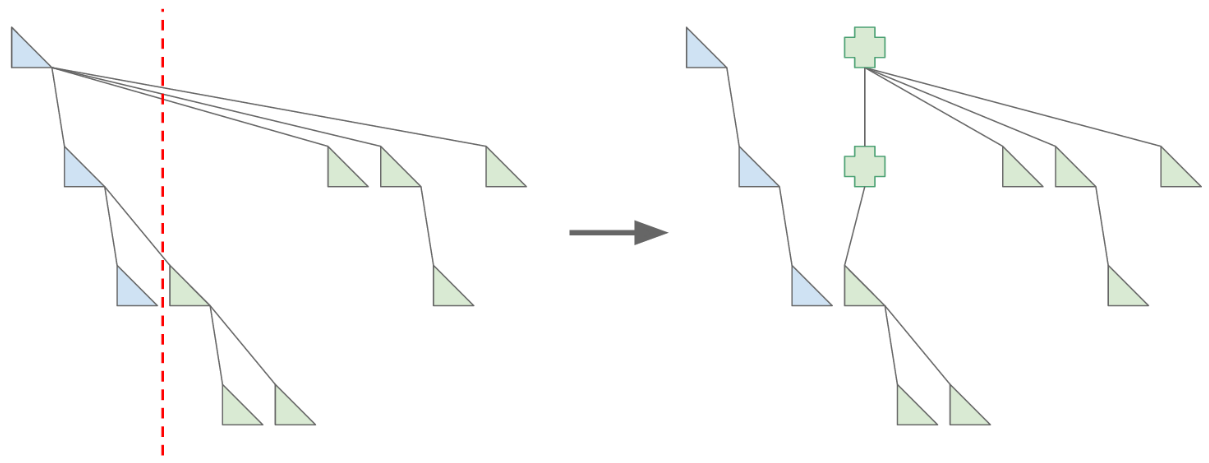
Since Infra's encoding is essentially a pre-order traversal, splitting an encoding at an arbitrary point looks like splitting a tree diagram along a vertical axis. To visualize this, we give every node its own column, in prefix order.



Now we can say that the tree can be split vertically at any point. The split operation just requires adding a Continuation header for each tier being cut. Here is an example involving just the top level:



Here is another example involving the first two tiers:



We have been speaking about the role of Continuations in terms of splitting pre-existing trees, but the expected main use case is when a real-time application is repeatedly waiting on further items to be processed/generated and it wants to submit the partial batches as they are available. This is useful for debugging, real-time visualizing, and for interactively editing partial states.

3.7 Base types: Integer and Floating-Point

The most fundamental contrast between binary and textual data formats is whether the bytes are meant to represent quantities or character codes. Nearly all hardware and programming language primitives have long since settled on two genres of number encoding: two's complement integer and IEEE-754 floating-point. The major source of variety in programming language primitives stems from the various byte widths. Since the **Integer** and **Floating-Point** segment headers have a size field, a single segment type for integer can cover all byte-widths, going beyond the commonly named widths: byte, short, int, long, int64, octaword, etc. Likewise, **Floating-Point** covers all byte-widths defined by the IEEE-754 standard: half, single, float, double, quadruple, etc.

Byte endian-ness comes into play when encoding binary quantities. The encoding defaults to little-endian since that has become most common, but the endian mode can be overriden using metadata. The mode setting cascades in a hierarchical manner, so placing a single little-endian declaration at the root of the tree (in metadata) will apply to all quantities. The nature of this metadata is described in section 3.5.2 on metadata language channels. If fine control is specified, any subtree can override the mode. This allows each platform to use whatever byte order is more efficient for it, and to unambiguously convert data received from an architecture with an opposite byte order, and only to the extent needed.

3.7.1 Quantity Types in the Editor

Unlike plaintext, Infra can be sophisticated in how it deals with numbers. Numeric elements are displayed in a monospaced font to help line up decimal points in tabular data as well as to distinguish them from string elements. An editor can make authoring quantities seamless by automatically parsing to the shortest binary encoding when possible.

As a demonstration, we will demonstrate by typing negative-one-thousand-point-five. There will be a figure to show a snapshot after each keystroke. The right side of the figures will show the current byte encoding.

```
-          212D
```

At this point, this is a one character string.

```
-1         31FF
```

Now, infra recognizes ‘dash one’ as a number and encodes it efficiently. Its header byte signifies that the following byte be interpreted as an integer.

```
-10        31F6
```

By simply pressing zero, negative one became negative ten while remaining binary encoded.

```
-100          319C
```

Now it is negative 100. Any typing a third zero...

```
-1,000          32FC18
```

Negative one thousand. Because this is explicitly a number in the encoding, the editor can show digit grouping based on the user's localization settings. Text formats often cannot tolerate commas in numbers because they use them as delimiters to separate list items.

```
-1000.          262D313030302E
```

Because the string representation of "-1000" does not contain a decimal point, Infra must revert to encoding this as a string. In general, numbers are tested for round-trip stability before being converted to a binary encoded number.

```
-1,000.5          44C47A2000
```

Now that the character 5 has been added, converting the string to a float and back to a string again results in the original string. Thus Infra can safely encode -1000.5 as a floating point number.

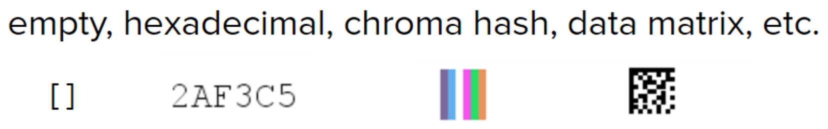
The widely-used Grisu3 and Dragon4 [27] shortest-decimal-representation algorithms are used to reverse decimal-to-binary rounding loss when performing the round-trip test.

3.8 Base type: Byte Array

Byte Arrays are generally for opaque binary data, such as embedding a non-human-readable format within Infra. There are many ways to display a byte array in an editor however. A traditional way to view and edit raw bytes is in hexadecimal.

```
"A"    → 10    → "0A"  ✗
"3F"   → 63    → "3F"  ✓
```

But when manual editing is not necessary, it can be convenient to use a more visually-compact form. The forms included in our prototype are hexadecimal, chroma hash, and data matrix.



Byte Arrays can also be used as more compact encoding for an Infra List, if its elements are all of the same structure and footprint. If the data type is described in the Byte Array's 'schema' metadata, then the headers can be omitted from the data in the array while still allowing a parser to understand the structure of the bytes. (See section: Type System.)

3.9 Special Leaf Types: Bits, Nibble, Symbol

Special leaves differ from the other primitives by using the length field portion of the header byte in a different way.

Bit Array In the case of **Bit Array**, the length is not specified in bytes, it is specified in bits. This allows for encoding a block of opaque data that is not a multiple of eight bits long. (Extra bits are encoded as zeros at the end of the segment so that the next segment starts on a whole-byte boundary.)

Nibble In the case of **Nibble** and **Symbol**, the length field is used as immediate data. **Nibble** uses the immediate to encode a 4-bit unsigned integer, allowing 0-15 to be encoded without any bytes in the body of the segment.

Symbol **Symbol** uses the immediate to define 16 special enumeration constants. We currently only allocate 6 of them as the following: *False*, *True*, *Void*, *Null*, *Parameter*, *Problem*. This list should contain reasonably common data constants that can offer useful special meaning to program logic.

Value	Name	Concept
0x0	False	Logical falsehood
0x1	True	Logical truthhood
0x2	Void	An intended / explicit nothingness
0x3	Null	An unintended / implicit nothingness
0x4	Parameter	Expecting to be replaced with a value
0x5	Problem	Marks an unintended halt on dependent computation

Parameter is used as a placeholder for a value that is waiting to be specified. It can be used to invite the user to fill in a value. *Mathematical constant* stands in for a named literal value. See the mathematical constant language in the Metadata Language Channels Section for how the name of the constant is specified. This is especially useful for representing irrational numbers that cannot be encoded precisely. A system can substitute a best-effort finite-value only when needed. *Problem* is returned by a patch when an undefined operation or type-error is encountered. Metadata on the *problem* node can be used to identify the issue. Otherwise legal operations with some error parameters concatenate the metadata of their error parameters and return the conjoined error. This mechanism can take the place of syntax error reporting.

3.10 Singleton Unwrapping

When writing an Infra tree to a file, it is always possible to append yet another element. For this reason, the file buffer itself is taken as a root List element, housing whatever series of elements happen to appear in its body. This implies that a completely empty file is encoding an empty List as opposed to non-representable nothingness. If this was not the case, then any data encoded after the first Infra element read from a file would have to be ignored. This issue comes up in the design of metaformats like XML, where there is no meaning past the point of the closing of the root tag, and an empty string is malformed, causing a parse error.

A benefit of treating the file itself as a root List container is that is avoids an otherwise redundant size field that would have to be written for the root of an Infra tree. The disadvantage, however, is that there is a potential ambiguity regarding trees that do not have a List element as its root. For example, if the 'tree' being encoded is just an Integer, it will appear to the parser as a List containing an Integer. Thus, a convention has to be established to enable disambiguation of these cases.

The rule is that once parsing completes, if the tree is a List of exactly one element, and the root list was never associated with a metadata segment (not even an empty one), then the single item should be 'unwrapped' and returned as the tree root. So, on the flip side, if the tree being encoded is in fact a List containing exactly one element, the presence of a trailing metadata segment in the encoding (even if there are no metadata values in it) can be used to suppress the notion that the lone item was never meant to be wrapped in a List in the first place. The logic behind this is that trailing metadata ends up being associated with the root (List), and if the lone item was meant to be the root all along, then there would have been nothing for such metadata to have been coming from. Note that the lone item can optionally have its own metadata, because that metadata segment would precede it in the encoding and not confuse matters.

3.11 Delayed Loading and Incremental Decoding

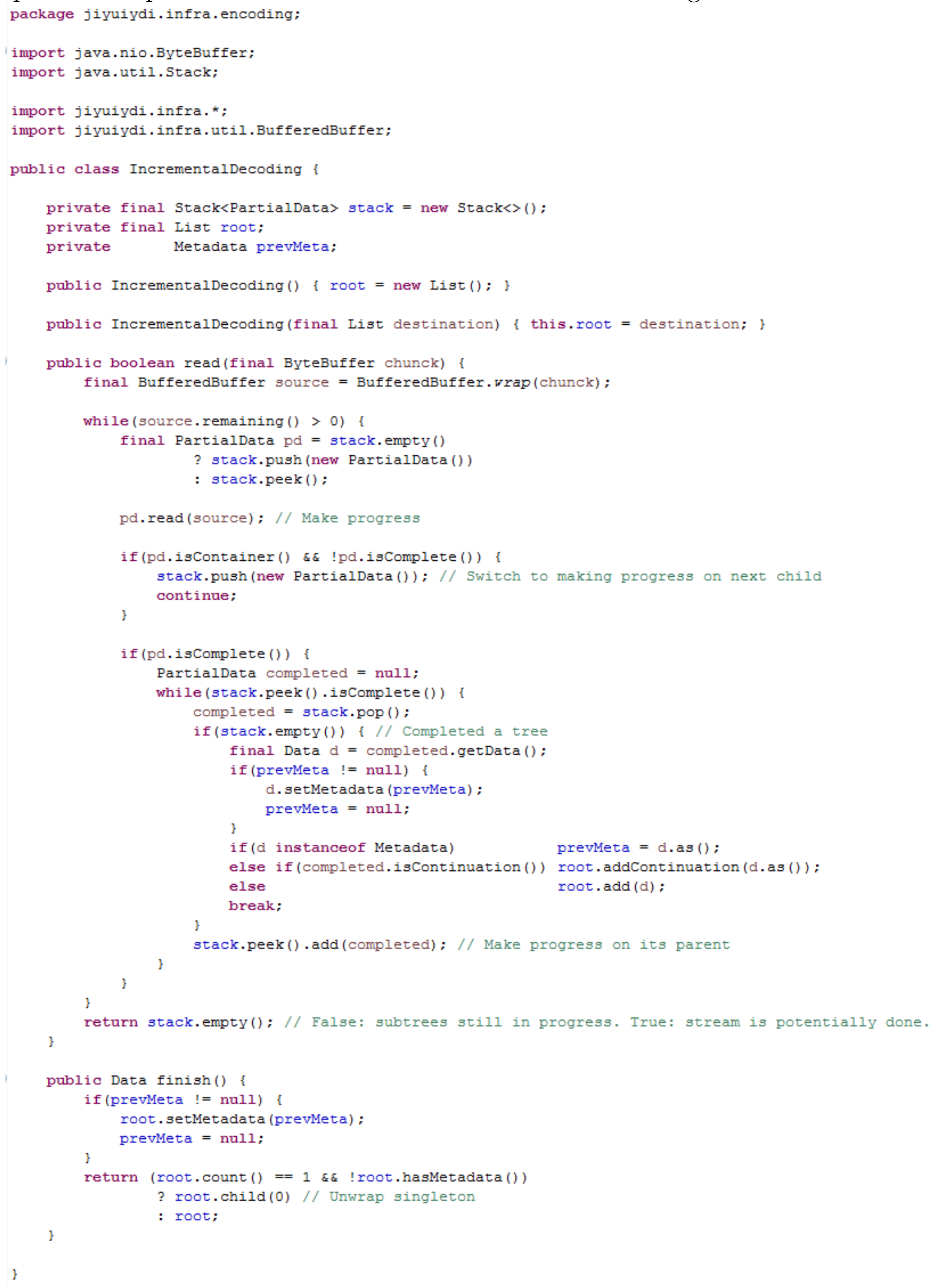
In order for Infra to function as a general purpose media, it is critical that some portions of data be loaded only on demand, and that data streams be asynchronous.

Delayed Loading When loading large Infra trees, it can be sluggish for the system and overwhelming for the user to load or display the whole thing right away. Loading subtrees as they are needed, and unloading least-frequently-used subtrees, can bring efficiencies and the possibility of exploring trees that do not fit in memory. Also, not all Infra nodes are representing data from a byte stream loaded from a file. There are native-service objects that, for example, represent the file system tree as Infra nodes. It would be detrimental if generating an outline of the entire file system had to be a natural side-effect of viewing a individual directory.

Incremental Decoding Even though the Infra encoding format has explicitly-sized nested segments, it can still be parsed in a manner that allows arbitrarily few bytes at a time to be ingested. This is critical for the subtrees completed so far to be immediately usable while waiting for the rest to stream in.

This is possible using a stack, where each stack element represents a segment in progress. To enable safe asynchronous use of a partially assembled tree, our library and editor implementation support an interface for detecting partially loaded and on-demand-loaded containers.

When reading the next available chunk of bytes from a stream, it is known in advance which subtrees will load completely within that number of bytes. This can be used to avoid the overhead of incremental loading for elements that do not need it. The following is a partial sample of our source code for incremental decoding.



```java
package jiyuiydi.infra.encoding;

import java.nio.ByteBuffer;
import java.util.Stack;

import jiyuiydi.infra.*;
import jiyuiydi.infra.util.BufferedBuffer;

public class IncrementalDecoding {

    private final Stack<PartialData> stack = new Stack<>();
    private final List root;
    private       Metadata prevMeta;

    public IncrementalDecoding() { root = new List(); }

    public IncrementalDecoding(final List destination) { this.root = destination; }

    public boolean read(final ByteBuffer chunck) {
        final BufferedBuffer source = BufferedBuffer.wrap(chunck);

        while(source.remaining() > 0) {
            final PartialData pd = stack.empty()
                    ? stack.push(new PartialData())
                    : stack.peek();

            pd.read(source); // Make progress

            if(pd.isContainer() && !pd.isComplete()) {
                stack.push(new PartialData()); // Switch to making progress on next child
                continue;
            }

            if(pd.isComplete()) {
                PartialData completed = null;
                while(stack.peek().isComplete()) {
                    completed = stack.pop();
                    if(stack.empty()) { // Completed a tree
                        final Data d = completed.getData();
                        if(prevMeta != null) {
                            d.setMetadata(prevMeta);
                            prevMeta = null;
                        }
                        if(d instanceof Metadata)            prevMeta = d.as();
                        else if(completed.isContinuation()) root.addContinuation(d.as());
                        else                                 root.add(d);
                        break;
                    }
                    stack.peek().add(completed); // Make progress on its parent
                }
            }
        }
        return stack.empty(); // False: subtrees still in progress. True: stream is potentially done.
    }

    public Data finish() {
        if(prevMeta != null) {
            root.setMetadata(prevMeta);
            prevMeta = null;
        }
        return (root.count() == 1 && !root.hasMetadata())
                ? root.child(0) // Unwrap singleton
                : root;
    }

}
```

Chapter 4

Patch: Reference and Modify

A Patch is a program that evaluates to a data structure. The simplest type of Patch simply references another node in the tree, returning a shallow copy; this expands the domain of infra from trees to fully-general graph structures by providing cross-links. Patch programs can also make modifications to what they return (without modifying the original); this enables pure-functional programming at the encoding layer. In the encoding, Patch nodes are containers like List nodes, except their children are interpreted as instructions for assembling a return value. Thus, a Patch's program statements are made up of the exact entities that Patches manipulate, not merely a one-to-one conversion between them. This qualifies Infra as having a stronger sense of homoiconicity than is enjoyed by Lisp-like languages.

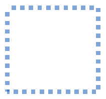
Like a formula in a cell of a spreadsheet, a Patch always has an implicit return value, has no explicit input parameters, has no need of a name (anonymous function / lambda), can reference its surrounding data environment (closure), can perform no visible side-effects (pure-function), and is therefore referentially transparent. Unlike a formula in a cell, Patch instructions form a chain of object-oriented commands, and can make edits to their subjective view of data (like a 'lens' in functional programming [28]).

A Patch's edits operate on a private overlay of the structure so that edits are reflected only in its return value and not in the original source material, such as in immutable or persistent data structures. Later, we will see that Patches can return Side-Effect objects which can, in turn, perform controlled side effects.

The basic role of a Patch is to reference data at some other location in the tree. In a spreadsheet, the uniform grid provides cells with an implicit name for formulas to leverage. In hierarchical structures like XML, a language such as XPath [29] is used to query elements matching structural patterns. In Infra, Patches are a bit of a hybrid of those two models. Being one of the first-class base-types, Patches are containers just like List nodes, and as such, contain further Infra nodes as its children. These children form an ordered sequence of commands that operate much like the individual segments of an XPath expression - performing iterative tree navigation through parent and child relationships. Patch commands can also behave similarly to spreadsheet formulae - operating from within the reference frame of the data environment and being able to jump discontinuously to nodes tagged with unique identifiers.

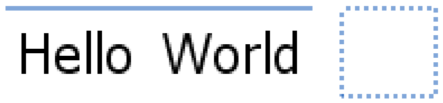
The Patch execution model is centered around the metaphor of a virtual cursor, equivalent in nature to the cursor in an Infra editor. A Patch's virtual cursor begins execution at the Patch's own location in the data tree. Starting at that point, Patch instructions navigate to the desired node, and then describe the modifications that should be made to the returned version of the referenced node. Each Patch instruction is a Keyed List made up of an opcode (key) and a set of arguments (values). The set of available opcodes is the same as the set of edit operations available in the Infra editor. Not only is this set of opcodes general enough to make any modification to a return value, but it allows for a user-friendly way to author simple Patch programs. A modified reference can be authored by demonstration without ever needing to see or write Patch code. A user's modifications to the Patch output are appended to the Patch's instructions like a macro recording.

Here is what an empty Patch looks like in the prototype; it is characterized by a dashed outline, distinct from the solid lines of Lists:

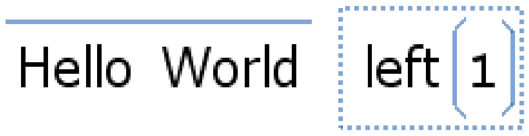


Since a Patch's virtual cursor initializes to its own location in the structure, an empty Patch naturally evaluates to itself. In this way, an empty Patch node in the encoding is the symbol for the concept of self reference. It is simultaneously, fully evaluated, and an infinite recursion. We will return to the topic of self-reference in the section on circular reference.

Let us add some other content. Here, a list with the strings 'Hello' and 'World' have been added before the Patch:

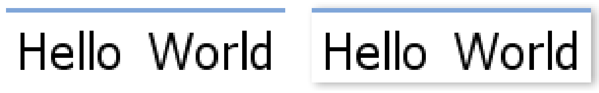


Now let us add an instruction to the Patch:



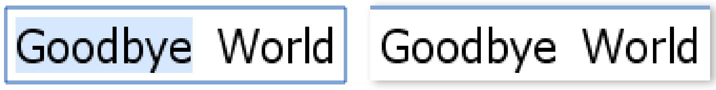
Note that this instruction is a Keyed List node. The key (first child) is 'left' and the value (second child) is 1. Keyed Lists are a natural fit for modelling the familiar function call syntax, where all children after the first are the function arguments. It turns out that the key of a Keyed-List Patch instruction will be substituted for a binary encoded integer representing the cardinal value for the opcode of that name. This is essentially an *author-time* compilation step for runtime efficiency and also allows for very straightforward internationalization of the opcode names. This can also be done in a way that provides a drop-down widget for the user to be able to discover the available opcode values.

Once this Patch is 'closed' in the editor (by moving the selection focus up off the children), it evaluates, looking like the following:



The instruction we added was the 'left' instruction, which moves the Patch's virtual cursor to a sibling of a lower index - in this case '1' lower. This moves the focus to the List [Hello World]. Since that is the only instruction, the Patch evaluation is over and the return value is whatever the cursor ended on. In this rendering, the second view of [Hello World] has a drop-shadow as an indication that it is a Patched-in value. It is best for the differentiation between a literal value and an Patched value to be subtle because, for all intents and purposes, Patches are referentially transparent. An editor can have various means for indicating underlying Patches to the user and for 'opening' them back up. In our prototype, a result's source-Patch code can be opened through a right-click context menu or keyboard shortcut.

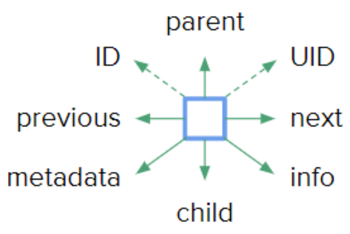
Before we introduce the other opcodes, we would like to foreshadow a few things and address some potential concerns. Patch's semantics are carefully designed to allow Patch programs to be authored indirectly in most cases. With basic editor support, common use cases of Patch do not require manually writing Patch instructions. For example, the Patch in the Hello World example above could instead be assembled from scratch using Copy/Paste: select target, hit Copy, select location, hit Paste. The reference instructions are calculated relative to the location automatically behind the scenes, never to be seen, but always available for inspection. In that case the copy would be a second live-view of the first rather than a forked clone as it would have to be in plaintext encodings. The following screenshot shows us replacing Hello with Goodbye; note that both views reflect this change.



As we will cover later, references can also leverage ID metadata to refer to named nodes directly, or schema metadata to provide typing information for stongly-typed scenarios. No matter how a Patch reference is constructed, editor infrastructure should maintain correct paths whenever the relationship between a Patch and target changes in the course of general editing (just as spreadsheets do when moving cells around).

Infra's user-centric philosophy aligns its core abstraction primitives with those of user-interface interaction. Patch opcodes are a procedural mirror of the actions a user performs in a text editor, or rather, a tree-oriented editor. The central metaphor is a cursor to navigate and make edits through. From this perspective, a Patch program is like a recorded macro of user activity, and a Patch's modified return values are like the transient unsaved changes of an open document. The name 'Patch' refers both to patch cords, which form connections by patching in signals, and to patches in the sense of updating documents via diffing operations.

4.1 Opcodes for Navigation



parent($\boldsymbol{n}$) shifts the focus cursor up the tree by n tiers. If the argument is omitted, the behavior is equivalent to parent(1). If there is no nth parent, the Patch instead evaluates to the Problem symbol with metadata describing the issue and execution is halted.

child($\boldsymbol{i}$) shifts the focus cursor to its ith child. Or if the argument is a Keyed List, the focus will shift to the first child Keyed-List with a matching key (the same key as in the argument). If the argument is omitted, the behavior is equivalent to child(0). If multiple arguments are provided, each will be considered an index for successive applications of child(i). In other words, child(2 0 1) would be equivalent to the sequence: child(2) child(0) child(1).

previous($\boldsymbol{n}$) shift focus to the sibling with the index n less than the focus' own index in its parent.

next($\boldsymbol{n}$) shift focus to the sibling with the index n more than the focus' own index in its parent.

metadata(*channel*) shifts the focus to its associated metadata container, and then to a Keyed List within it whose key matches *channel*. If the argument is omitted, focus just moves to the metadata container in general. Focus shifts to the Problem symbol if no Keyed List matches the given *channel* value.

ID(*id*) jumps the focus cursor to the 'nearest node' with an ID-metadata value matching *id*. The search order resembles classical scoping rules for identifiers in most programming languages. To start, the first level of children of the focus are searched. If none have ID metadata that matches, siblings are searched. And then, the search resorts to siblings of the parent, grandparent, etc.

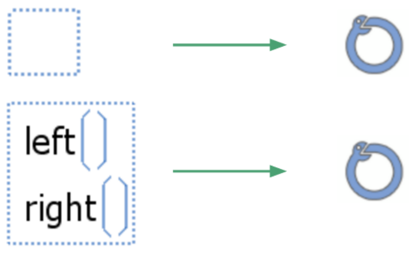
UID(*uid*) jumps the focus cursor to the unique node with a UID-metadata value matching *uid*. UID-labeled data have their own namespace and do not have to avoid name collisions with ID-labeled data.

info() shift focus to a synthetic tree populated with information about the element that was currently in focus, such as its number of children, its index position in its parent container, and its encoding type.

4.2 Circular Reference

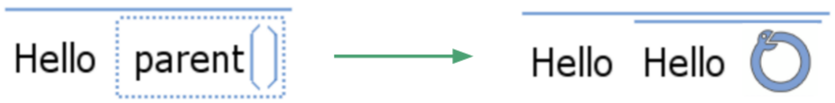
A Patch can end up involved in its own result. We break this down in to three cases:

1. A Patch refers directly to itself. This can occur when a Patch never moves the cursor (no operations), or if it moves the cursor off itself and back on. During evaluation, the Patch result is temporarily set to the self-reference symbol (a canonical instance of an empty Patch), so that there is something for the cursor to interact against if it to attempt an operation other than navigating away.

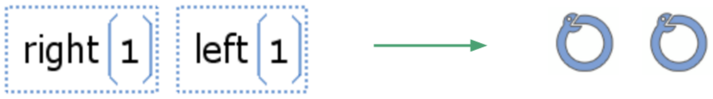


2. A Patch's result *contains* itself. If a Patch ends up as a descendant of its own result, the output value is still clearly defined, albeit an infinite tree. Of the three cases, this is the only one where the output value of the Patch is not the self-reference

symbol. Only the editor/view layer needs to detect these scenarios and detection is straight forward.



3. A Patch refers to itself through a chain of one or more other Patches that form a chain that eventually refer back to the first, forming a cycle.



Indirect self reference can be detected through the use of the dual-pointer or "tortoise and the hare algorithm" for detecting cycles. The following is a code sample of such an algorithm from our Java implementation.

```
public class Patch extends List {

    static public final Patch selfReference = new Patch();

    protected Data directResult;  // Can be an error value
    protected Data reducedResult; // Never an error value

    public Data directResult() {
        if(directResult == null) execute();
        return directResult;
    }

    public Data reducedResult() {
        if(reducedResult != null) return reducedResult;
        Data temp, cycleProbe = reducedResult = this;

        do { // This loop is unrolled by a factor of 2
            temp = ((Patch)reducedResult).directResult();
            if(temp.getClass() == Symbol.Error.class) break;
            if(temp == cycleProbe) { reducedResult = selfReference; break; }
            reducedResult = temp; // OK to reduce further

            if(!(reducedResult instanceof Patch)) break;

            temp = ((Patch)reducedResult).directResult();
            if(temp.getClass() == Symbol.Error.class) break;
            if(temp == cycleProbe) { reducedResult = selfReference; break; }
            reducedResult = temp; // OK to reduce further

            if(cycleProbe instanceof Patch) // advance probe by one step for every two taken by reducedResult
                cycleProbe = ((Patch)cycleProbe).directResult();
        } while(reducedResult instanceof Patch);
        return reducedResult;
    }

    public Data fullyEvaluatedResult() {
        Data ans = reducedResult();
        return ans instanceof Patch ? ((Patch)ans).directResult() : ans;
    }

    protected void execute() {
        directResult = selfReference; // To terminate indirect self-reference cycle
        directResult = new PatchExecution(this).run();
        if(directResult == this) directResult = selfReference; // Direct self-reference
    }

    ...

}
```

4.3 Opcodes for Modification

The secondary role of Patch instructions is to perform edits, modifying the value being referenced, but only from that Patch's perspective. This is akin to concepts such as: copy-on-write, persistent data structures, and 'modifiable references' in [30]. Since the original reference material is guaranteed not to be modified, and that material is the Patch's only input source, Patches behave like 'pure functions'.

insert(v) modifies the focus' parent to contain v at the same index as the focus cursor. If multiple arguments are provided, all will be inserted at successive index positions.

remove(*n*) removes the next n items starting at the position of the focus cursor. In the argument is omitted, the behavior is equivalent to remove(1).

write(*v*) overwrites the value at the current focus with the value v. A critical detail is that any metadata on the old value is retained. If metadata exists at the target v is cloned. If v also includes metadata, a metadata container is also cloned to perform a merge. If no metadata is involved, this acts like a remove() followed by an insert(v).

append(*v*) inserts value v as a new last child of the list in focus.

sync(*label*) halts execution and causes the Patch to evaluate to a Side-Effect object (See section 4.8 on Infra's Effect System).

In our prototype editor, we represent a Side-Effect object as a clickable button labeled with the value of the *label* argument. A good label describes concisely to the user what the goal of the particular mutation is. If the argument is omitted, we default to using the value being saved as a label. When the user clicks the button, the Patch is resumed in a context where it is safe to mutate the subtree it references.

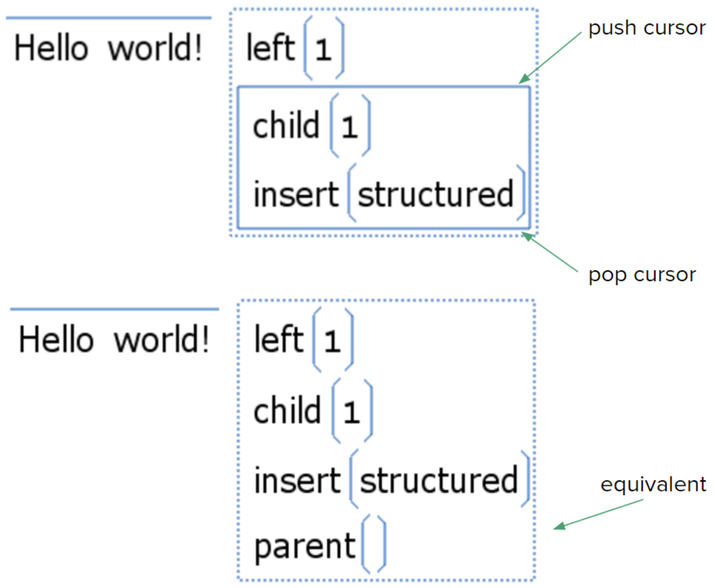
4.4 Nesting Statements and Shorthand

Normally, Patch instructions are Keyed List nodes; the first element is the operation and the rest are the parameters. We have the opportunity to define meaning to the use of other base types as a direct child of a Patch. Thus far, we only found the need to define special meaning for four other types, three of which are merely constrained shorthand for existing opcodes.

Node Type	Meaning Inside a Patch
Keyed List	An instruction. Key is opcode number or service-object name; values are input arguments.
List	Group of instructions nested in a cursor-position stack frame.
UTF-8 String	Shorthand for an ID instruction. ID(*String*)
Integer	Shorthand for a UID instruction. UID(*Integer*)
Nibble	Shorthand for a UID instruction. UID(*Nibble*)
Patch	Evaluated, then the result is interpreted according to this table.
Floating Point	No meaning.
Byte Array	No meaning.
Bit Array	No meaning.
Symbol	No meaning.
Free / Metadata / Continuation	N/A. These encoding types do not appear at the tree level.

List elements are treated as a group of nested instructions. Each child element in the List is executed as a normal instruction. When execution completed within the block, the focus cursor is restored to its location before the block was executed.

In the scenario where a Patch wishes to modify a subtree of the result in progress, the general idea would be to move the cursor down to the subtree, perform the edits, and walk the cursor back up through parent nodes. Nesting instructions in a List makes this easier and eliminates instruction clutter by automatically performing the 'walk back up'.



As a consequence of these semantics, nested instructions have no net effect if not performing any modification, and nesting is extraneous if not performing any navigation.

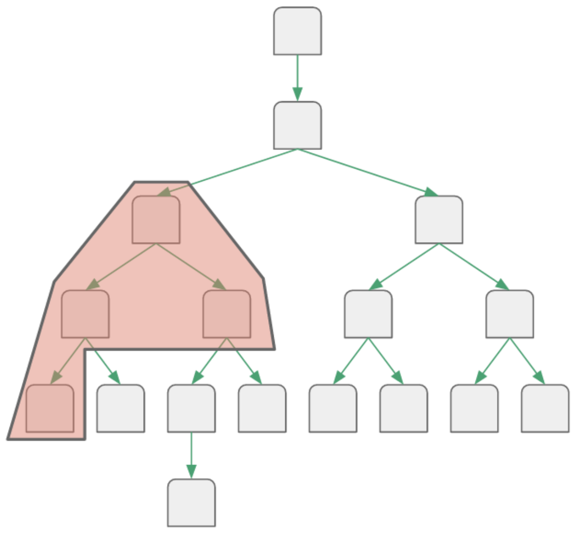
UTF-8 elements are treated as a shorthand for ID(*string*). This shorthand is convenient for the common case of using a string as an ID value.

Integer elements are treated as shorthand for UID(*number*). This shorthand is convenient for the common case of using string table lookups to de-duplicate string storage across an entire document.

4.5 Persistent Data Structure

The role of a persistent data structure is to allow mutation to occur in such a way that the previous state is left intact from the perspective of those still referencing it. A trivial means to achieve this is to create a new deep-copy of the structure before every edit, but this is prohibitively expensive in both time and memory for large structures.

More commonly, immutable data structures perform shallow copies along the spine of the structure, from the point of the edit up to the root. This copying occurs each time the structure is returned, i.e. with each individual edit. Because of the semantics of Patch, we can avoid performing any shallow copies above the highest point in the tree that will end up in the Patch's result. All edits can be batched into a lazily-expanded truncated-subtree overlay. It is a subtree because it does not have to be rooted any higher in the tree than is relevant to the final result value. It is 'truncated' because it can share/reuse any values that remain below the region of mutations.

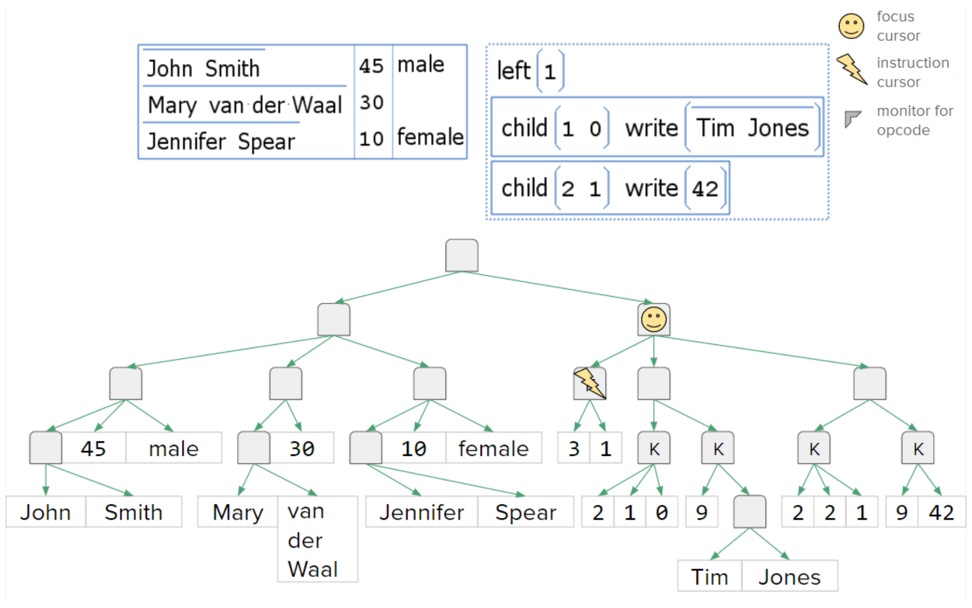


Each Patch execution context has one of these overlays. The overlay is empty until the first edit operation occurs. With each successive edit, or navigation after an edit, it expands just enough to include that location. Any value that is already part of the overlay can be mutated over and over with amortized cost, such as when inserting and removing a series of items in a list.

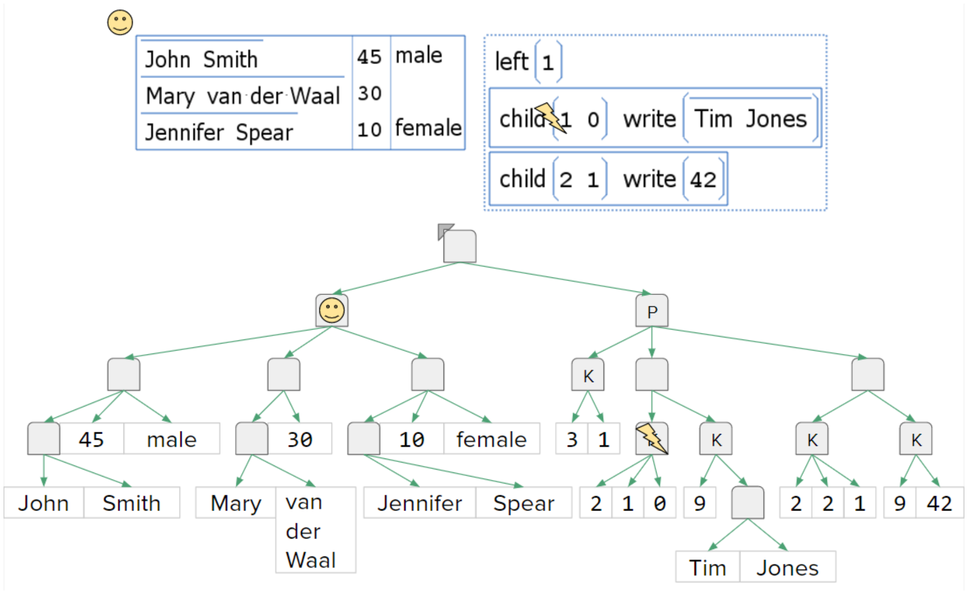
This batched-overlay approach assumes that a Patch's edits tend to have high locality with respect to each other and to the final resting point of the cursor. However, if they do not, the performance only degrades to that of a typical non-batched immutable data structure.

An overlay data structure need only contain a Parents List and a Cloning Map. The Parents List is a stack for remembering the previous focus when navigating to a child. The Cloning Map is a tree that remembers the extents of the overlay. Its role is essentially to remember which children have been cloned (shallow-copied) so far.

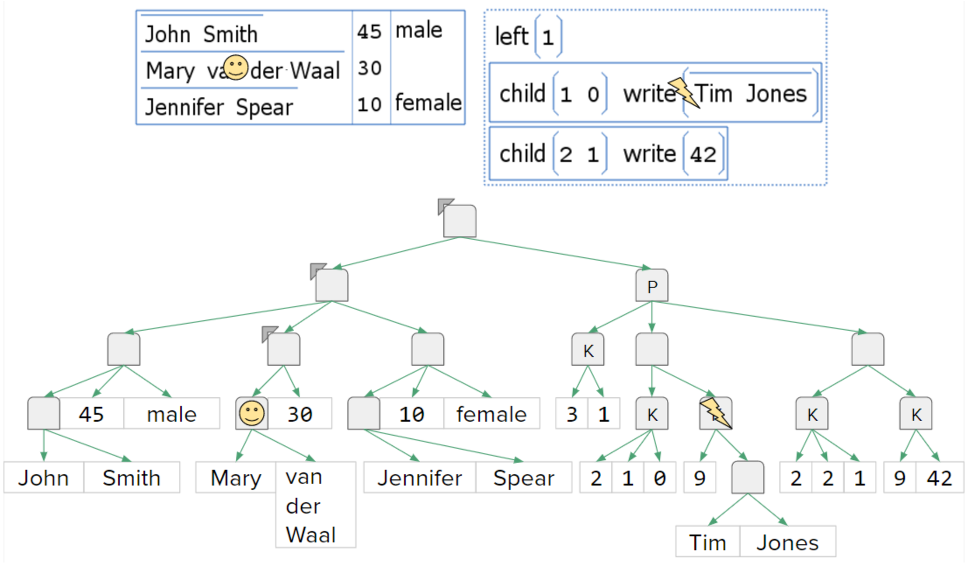
Let us walk through an example using specific data. The following tree contains some tabular data and a Patch that references and modifies one entry in that table. In each figure, eight in total, the data will be depicted in two forms: a typical editor view, and an abstract directed graph. This particular Patch example results in a view of the table with the name changed in the second entry and the number (age) changed in the third. Note that the opcodes are encoded at integers in the encoding tree, using their enumerated values.



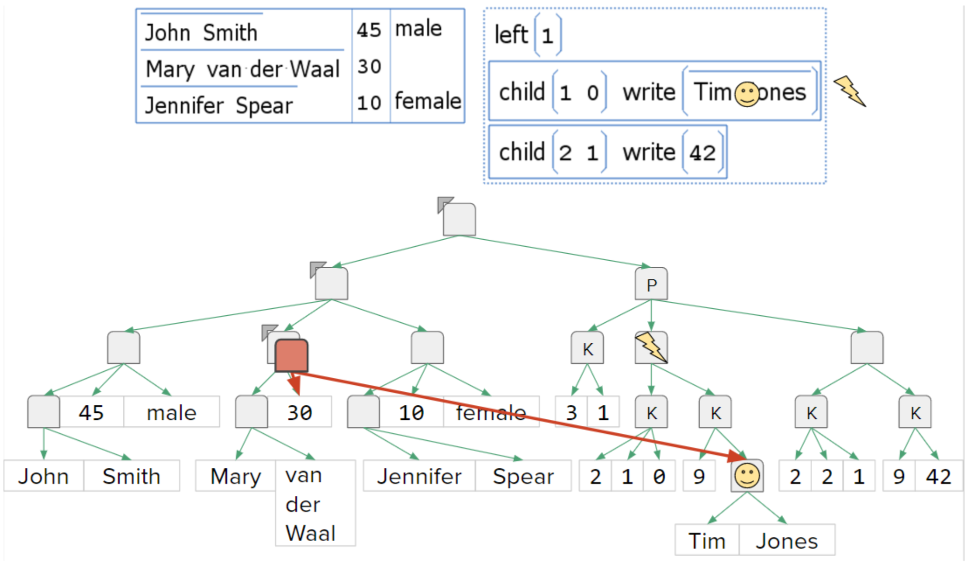
As described before, a Patch starts out focused on itself in the tree, as depicted by the location of the smiley-face in the figure above. Value Monitors are detailed in section 6.1.



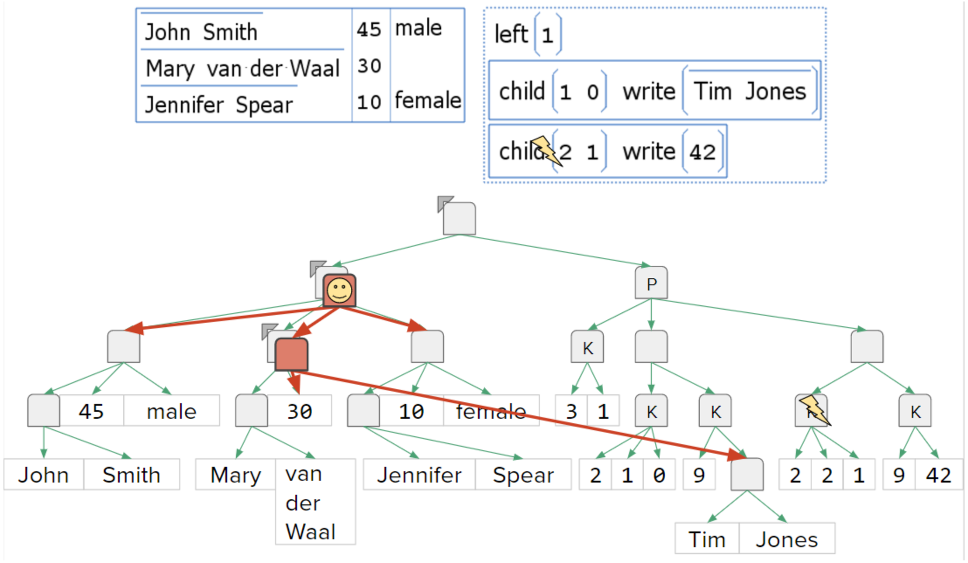
After the instruction to move the cursor to the previous sibling comes an instruction to move to child index 1 and then child index 0 after that.



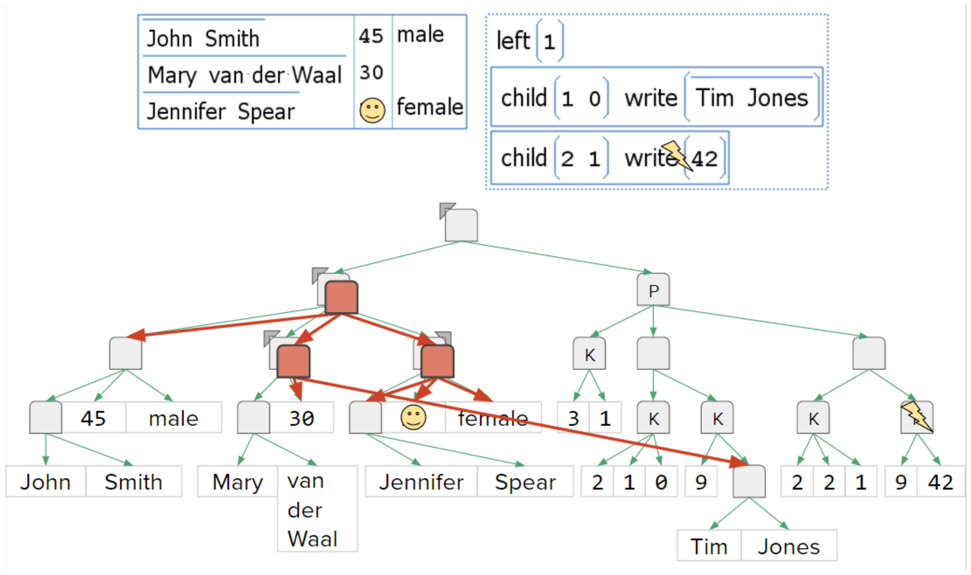
At this point, the Patch execution is at a write instruction. This will cause the parent to be cloned, since it is not already.



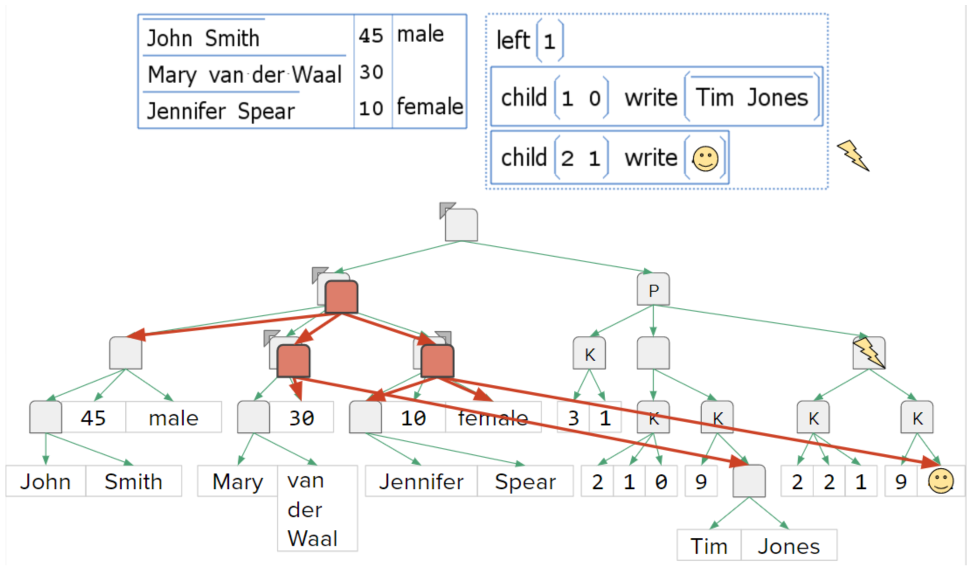
After the write, this virtual tree is technically a directed-acyclic graph because the value written is only referenced from its place in the Patch code.



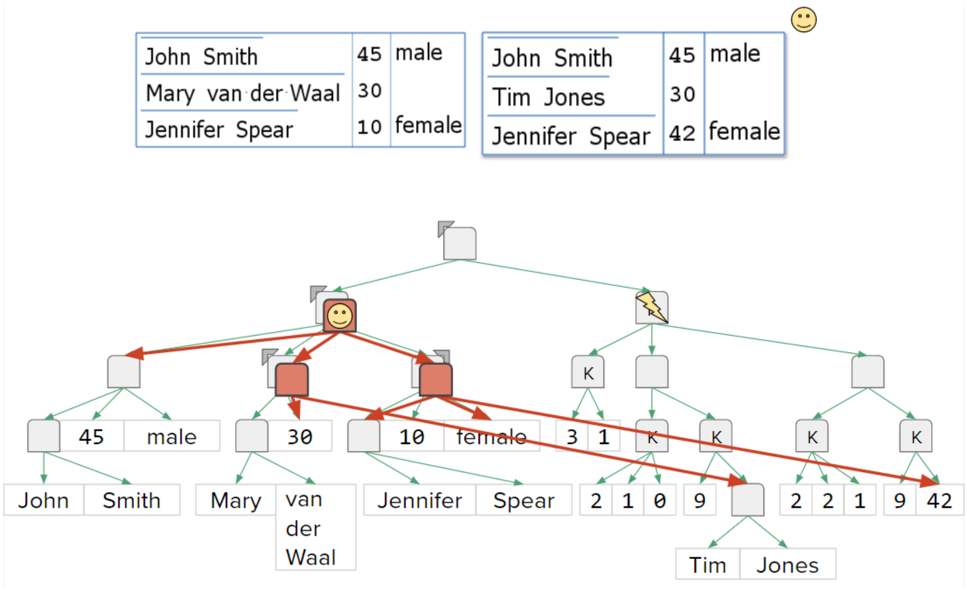
Since the last two statements were nested, and the end of the inner list has been reached in the execution, the cursor is 'popped' back to were it was when that list started. This has caused an additional node to be cloned so that the cursor does not end up outside of the cloned region.



Now cloning to expand the overlay has happened in the other direction - to prevent the focus from moving too deep below the bottom edge of the cloned region.



When this write occurs, no additional cloning need occur since the parent being changed is already part of the overlay.



John Smith	45	male
Mary van der Waal	30	
Jennifer Spear	10	female

John Smith	45	male
Tim Jones	30	
Jennifer Spear	42	female

Again, reaching the end of a nested list returns the cursor to its prior position. The instruction pointer has also reached the end of the Patch program and thus, the overlay root is the Patch's return value.

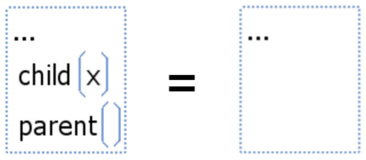
4.6 Parent Pointers / Closures

In order for a Patch's virtual cursor to navigate around the tree, it must rely on a node's pointer to its parent container. This is not a completely trivial concept because, due to Patches, a node can be considered to have multiple parents, depending on the definition. This is because any node can be patched into another part of the tree many times over by Patches in those locations. However, Patch semantics for parent navigation are defined along the lines of a *lexical closure*. We define parent pointers from an encoding perspective. At the encoding level, the structure is only ever a tree, even though it can express an arbitrary graph at a logical level. To reflect this, we call these pointers *Encoding Parents*.

As it turns out, only container types need an Encoding Parent, not leaf types. This is because only Patches need to use these pointers to access the surrounding tree, and only containers, by definition, can *contain* a Patch. So, from an implementation standpoint, in-memory representations of Infra structures with long arrays of strings or numbers do not incur any significant overhead from the general support of Patch semantics.

The marriage between the multi-parent aspect and the Encoding Parent aspect happens in the semantics of a Patch cursor. Navigation down to a child and back up to its parent must result in the same focus as if neither operation occurred.

(The ellipses is the following figure represent an arbitrary set of Patch commands.)



This has to be true no matter whether that child happened to be the output of a Patch or not. If the child is a Patch that evaluates to some distant node, that result's Encoding Parent is not going to be the node we approached it from. Thus, expressed as an implementation, the full semantics are:

- each Patch execution cursor has its own stack that starts empty
- *push* the focus to a stack when/before moving to a child
- *pop* the focus to move to a parent if the stack is not empty
- use the Encoding Parent to move to a parent if the stack is empty

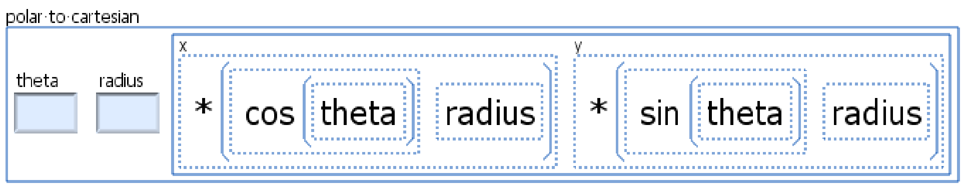
The implication is that even when a Patch is referenced by another, its evaluation still takes place relative to its static lexical context. This is essentially an implementation for closures, and enables Patch source codes to be written from a stable predictable frame of reference.

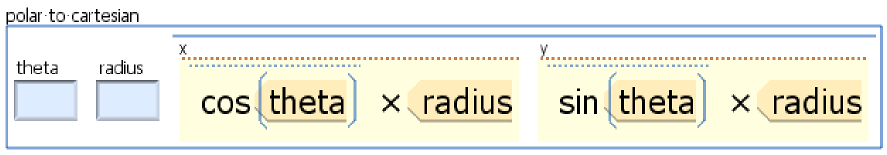
4.7 Patch as Function Application

Infra has no need for a native concept of a function or a function call. Since Patches can be defined inline, Patch semantics act simultaneously in the role of a function *and* a call site. Conceptually, function application is the process of taking an instance of a function and substituting argument values in for parameter placeholders. That very process can be performed *by* a Patch using the building blocks we have already introduced.

The convention for mimicking a traditional function is to have a list of default parameter values followed by a result value that is composed of one or more Patches that reference those parameter values. Instead of providing a default value for any particular parameter, Infra's Parameter symbol can be used as a valueless placeholder. ID or schema metadata can optionally provide names and type constraints and on any or all parameters.

Let us look at a concrete example function. To do this, we are going to have to get slightly ahead of ourselves and use an arithmetic object for multiplication, which is not introduced until the section on Native-Service Objects. The following example is a function that converts an angle and radius to an x-y pair.

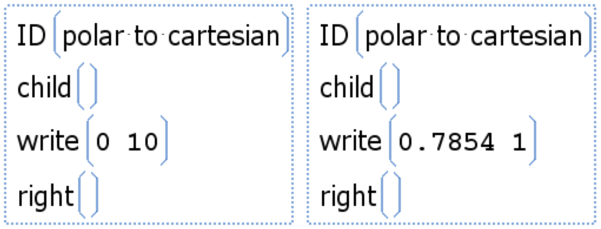




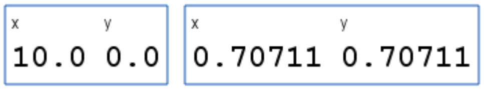
The top row shows the Patch tree without any evaluation, and the bottom row is the reduced (semi-evaluated) form that would appear in the editor (see the discussion on

Native-Service Objects for clarification). The pale blue boxes labeled 'theta' and 'radius' are *Parameter* symbol elements, which are meant to be used as placeholders for a future value - perfect to play to role of an input parameter.

Next let us look at two different elements that each call this 'polar to Cartesian' function with their own arguments.


```
ID polar to cartesian    ID polar to cartesian
child                    child
write 0 10               write 0.7854 1
right                    right
```

And here is what these Patches evaluate to:


```
x     y          x        y
10.0 0.0         0.70711 0.70711
```

There is an air of machine-code level programming to this use of Patch, but note that users will be viewing the evaluated output of Patches most of the time and author them indirectly through direct manipulation of their result value. Taking advantage of live interaction and iterative authoring of input values can enable a Patch to dispense its own documentation dynamically in response to the partial values as they are being assembled.

A general benefit of leaving function application as an emergent ability of Patch semantics, is that it itself is programmable. A variety of function-application semantics can be supported while keeping the number of first-class concepts in the Infra specification minimal.

- **default arguments** are achieved simply by not overwriting some of the hard-coded values in a function interface.

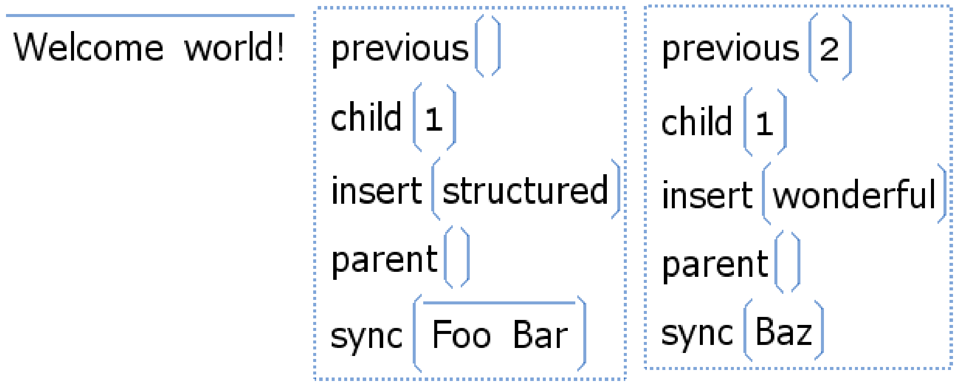
- **call by value** is achieved when writing literal values as input arguments.
- **call by name / need** is achieved when writing Patched values as input arguments. (lazily evaluated arguments)
- **currying** is achieved when writing values to only a subset of locations and leaving the rest to be referred to and written by other Patches.
- **named arguments** are achieved when using ID metadata on arguments to locate them.

4.8 Effect System

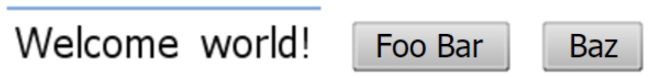
As described above, a Patch's edits normally only effect the result value of the Patch, leaving the referenced source material unchanged. On the occasions that it is useful for a Patch execution to have a stateful effect on the world, an effect system helps this to happen while keeping side effects explicit and controllable.

Infra has an effect system made up of three components: the Side-Effect object-type that can be returned as the result of a Patch, the sync() opcode that exports a Patch's attempted mutations as a Side-Effect object, and a permissions system to regulate automatic execution of the side-effects described by a Side-Effect object.

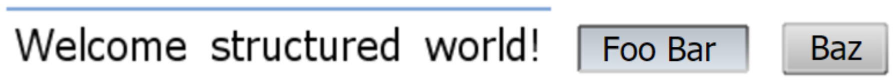
A Side-Effect object is analogous to a Pull Request in the popular Git version control system. They are inert return values until they are accepted/triggered. They encapsulate the edits that a Patch has made to the data it was referencing, such that those edits can be applied to the original (a destructive change) at the discretion of the runtime system. Because Side-Effect objects are represented as an interactive button in our prototype, they also resemble and behave like toolbar or drop-down-menu buttons in a graphical user interface, which trigger specific useful state changes on demand. An Infra editor UI allows users to trigger Side-Effect objects directly. Any Patches that have a dependency on a value being mutated are invalidated, and will be re-evaulated as needed. This same mechanism already has to be in place for normal edits made directly by the user.



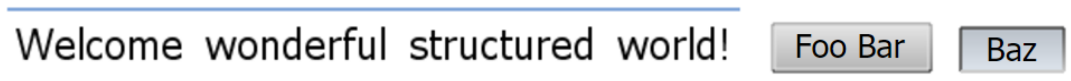
This tree evaluates to the following:



After the first button is activated, the addition of 'structured' occurs on the actual tree, persistently:



After the second button is activated:



When a Patch contains within it a Patch that returns a Side-Effect object, the top-level Patch will also evaluate to an Effect as well. Logically, this is because it is not only roadblocked waiting for *its* side effects to occur before proceeding, but it also needs to have a context for synchronizing mutations for the nested one to inherit. If schema metadata appears on a Patch that evaluates to an Effect, the schema value itself needs to be an Effect object in order to be consistent. This is akin to a print function in Haskell including “->IO ()” at the end of its type declaration to indicate that it returns an action in the form of the input/output monad.

4.9 Extended Opcodes

patch(*n*) shift focus to the Patch ‘behind’ the current focus. If the value at the current focus is not a Patch result, this operation halts evaluation with a Problem symbol.

data(*v*...) does nothing as an instruction besides ensure that the values v have a safe place to exist without being interpreted as instructions. This allows Patches to contain internal data in an aggregated place that is more clearly for reuse throughout the Patch.

choice(*i*) is identical in behavior to child(), however use of this code is meant to signal that any sibling would be equally valid. If the focus cursor is not moved again, the Patch’s resulting value will be the value of that child, but flagged as a Choice Point. In our prototype editor, we represent Choice Points with interactive drop-down list widgets. When used in schema metadata, Choice Points capture the concept of a disjoint union or sum type. Adding strong-typing via schema metadata is discussed in the Type System section.

fill(*v*...) replaces occurrences of the parameter symbol with the values of the arguments v. The parameter symbols are searched for in the source code of the Patch in focus. The search occurs in the depth first order.

write patch(*p*) replaces the current focus with the literal (un-evaluated) value p. It is assumed that p is a Patch, ready to be re-rooted in the new position. This results in ‘dynamic scoping’ for the references occurring within Patch p.

move to(*i*) moves the current focus to index i within its same parent.

move by(*n*) moves the current focus to its own index plus n.

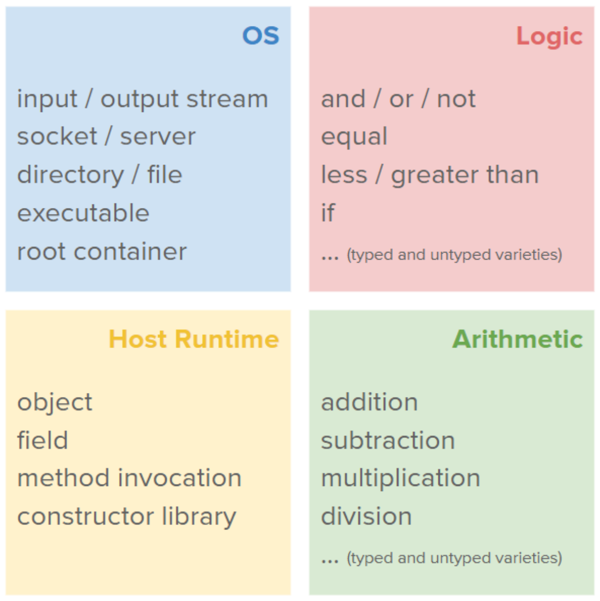
case(*match result*) shift focus to *result* if the value at the focus cursor matches *match*. In general, wild cards can be used to influence value comparison (see chapter: Native Service Objects). The case() opcode is meant to assist programming-by-example by capturing the conditions of end-user edits that replace the entirety of a Patch's output value (see section on programming-by-example).

value(*v*) shifts focus to v itself. This allows Patches to originate the value of their output entirely from within.

registry(*i...*) shifts focus to a tree of pre-populated values. These values are intended to provide byte-efficient access to verbose fully-qualified names for metadata languages. Reminder: the opcodes themselves are encoded as integers using a standard enumeration (to be finalized at a future date). This also offers a tree form of 'string interning' for constant-time equality checking at runtime. The exact order and structure of the standardized portion of the registry are left as future work.

Chapter 5

Native-Service Objects



As mentioned above, Patches can refer to in-scope named data using the ID opcode. Patches can also refer to native services by using their unique ID as if it were an opcode value. When a Patch interpreter does a lookup for an opcode value, it references a library of registered objects. This library contains at least the standard native service objects which extend Infra into a full programming language.

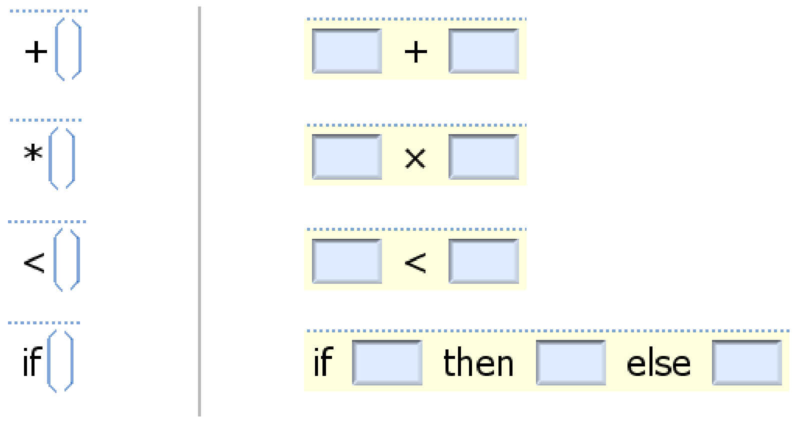
As we have seen thus far, Patch opcodes just perform tree navigation, insertions, and deletions. On their own, those operations can not perform computation that is sensitive to the values they operate on, which is to say, they are not Turing complete. However, those operations *are* sufficient for performing 'function application'. The right primitive functions just need to be available in order to bootstrap a capacity for computation. This is where the native service objects in the standard library come in.

Since their logic cannot be expressed in terms of virtual cursor manipulations, these built in functions must be built in to the standard just like the primitive operations in other programming languages are. They are loaded by name, just like any other named entity, but they each override Patch evaluation or mutation semantics with their own native logic rather than execute their contents as standard Patch cursor instructions. This allows them to use their contents merely as a presentation layer for their parameters - free to act like a domain-specific language.

We have explored Native Service Objects for performing logic and arithmetic, for performing operating system input/output (with the help of Infra's effect system), and for inspecting and manipulating Java runtime objects and methods.

5.1 Logic and Arithmetic

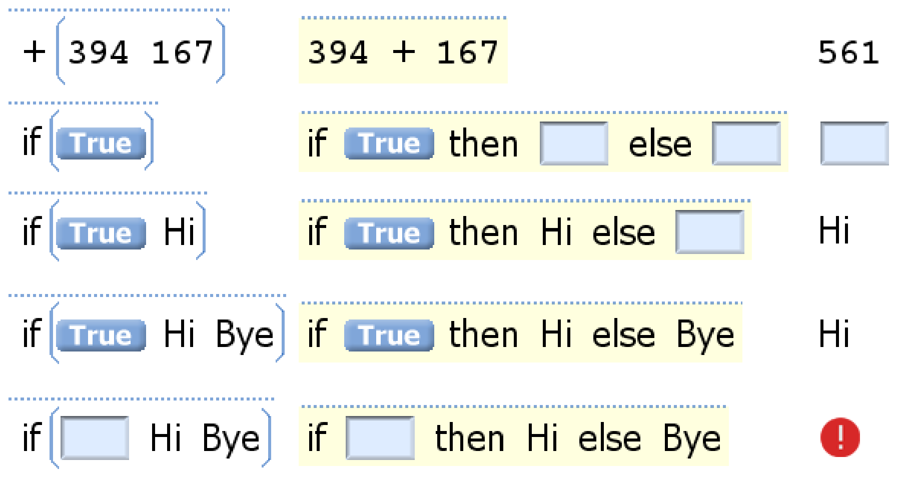
The logic and arithmetic entries in the standard library include boolean operators such as conjunction and disjunction, mathematical operators such as addition and subtraction, and control flow structures such as if-then-else.



In the table above, the first column contains four example Patches. The second column displays their corresponding evaluations, all of which are native-service objects. In our prototype editor, subclassed types are given a yellow background tint to remind the user that they evaluate according to overridden semantics. These native-service values cannot be serialized directly in the Infra encoding. This means that they are always a result value of a Patch that references their *a-priori* existence in a way that can be serialized.

Native-service Patches act like native functions by assembling their interface out of Parameter symbols. By replacing the parameters with values, the Patch performs its native logic on custom data. Note that the text elements in these objects are purely decorative for the sake of their user interface. They are not necessary for the object to perform its function, but would be nondescript without them. In the case of the math operators, the interface is able to resemble a familiar infix notation without the need for any explicit support or syntax for distinctions between prefix, infix, and postfix operators. Also note that, in the case of the multiplication example, the decorative elements can help expressions to use more appropriate Unicode characters without requiring the user to deal with the round-about ways to type them manually.

As usual, the parameter values can be written in with Path's modification opcodes, or the shorthand notation can be used if it is sufficient to fill parameters in depth-first order. The following table depicts the shorthand notation of listing argument values in the body of the instruction.



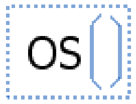
+ 394 167	394 + 167	561
if True	if True then ☐ else ☐	☐
if True Hi	if True then Hi else ☐	Hi
if True Hi Bye	if True then Hi else Bye	Hi
if ☐ Hi Bye	if ☐ then Hi else Bye	❗

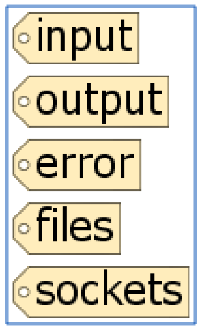
The first column is the directly encodable form of a call to a native object with argument values. The second column shows the values after one evaluation. The third column shows the values after a second evaluation.

In the prototype editor, Patches are displayed only as evaluated as they can be without error. In other words, the editor automatically evaluates Patches up to a point. This refers to Patches that evaluate to a Patch, that in turn evaluate potentially to a Patch. Once an evaluation chain results in a Problem symbol, the previous stage is the one displayed. Therefore, the example in the fifth row would be displayed in the form of the second column, while the others would be displayed in the form of the third column. This assists the user in filling in missing values or addressing errors. The lazy evaluation of Patches means that even deeply-nested issues could be easily addressed on the surface, one at a time, without the rest of the clutter.

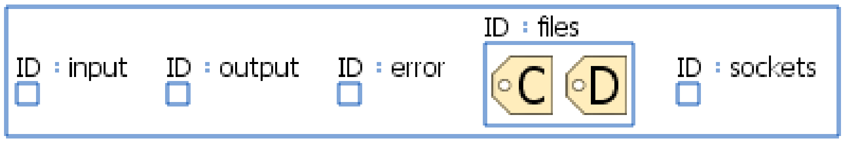
5.2 Operating System Integration

Native-service objects can provide external forms of input and output. Operating System integration is addressed by four categories of native-service object: standard input/output console streams, a file system tree, executable process interface, and socket binding interface. All instances of these services all grouped under a single object registered as "OS".





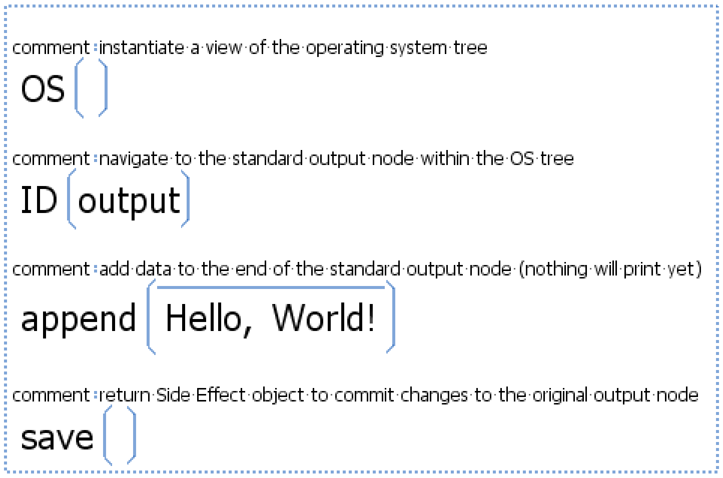
On the left is a Patch that jumps to the OS tree and stops. On the right is its evaluated value. These five items look the way they do because they have ID metadata values and our prototype defaults to displaying named elements as their ID value. To help avoid confusing the ID as the actual value at that location, it is rendered to look like a luggage tag. This is an example of a View Face (See Alternate View Faces). With the default Face we can see the actual values, and the tree would look like:



Each of these items are Lists that keep their contents in sync with the external reality of their corresponding data model in the operating system. Each are a special Native-Service Objects instantiated as singletons inside the OS Service Object.

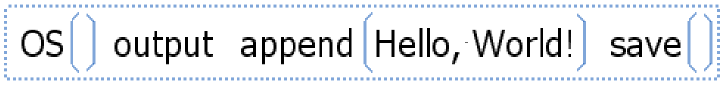
Standard I/O If an element is inserted into 'output', the element is also written to the process' standard output stream. Bytes written to the process' standard input stream get interpreted as Infra elements and appear in the 'input' list. If input bytes do not successfully parse as Infra, they appear in an additional child named 'raw input' (not shown), which is a byte array. Input is 'consumed' by actually removing elements it from their input container.

Now we have the building blocks we need to write a true "Hello, World!" program in Infra. The following Patch results in a button that, when clicked, prints the string to the standard output stream. (Reminder: printing to standard out is a state mutation and therefore requires the use of the save opcode.)



```
comment·instantiate·a·view·of·the·operating·system·tree
OS( )

comment·navigate·to·the·standard·output·node·within·the·OS·tree
ID(output)

comment·add·data·to·the·end·of·the·standard·output·node·(nothing·will·print·yet)
append(Hello, World!)

comment·return·Side·Effect·object·to·commit·changes·to·the·original·output·node
save( )
```

Here is is again without comments and using the shorthand form for ID lookup:



```
OS( ) output append(Hello,·World!) save( )
```

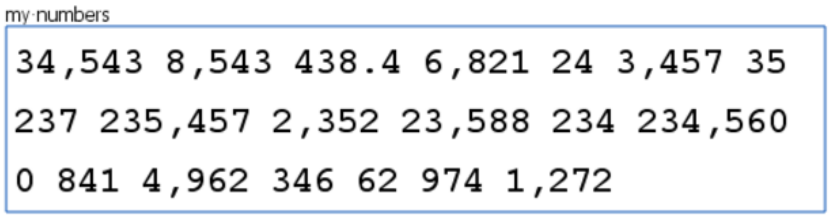
File System Tree In the OS figure, 'C' and 'D' represent drive letters (the root file system objects on our machine). These are actually List elements with ID metadata and, just as before, they are being displayed in a mode where their ID represents their whole.

In the UI, the actual children of a directory will be rendered when selecting or drilling down into that element. To avoid implicitly rendering the whole file system tree at once, it is important that Infra editors delay the loading of sub-trees until they are expanded.

File Loading File loading can be as simple as navigating the file tree through the 'files' OS child, but opening a file technically causes side effects, such as changing its 'last accessed' time-stamp and taking a read lock from the operating system. Thus, the native service object representing a file is also a Side-Effect Object (displayed as a button).

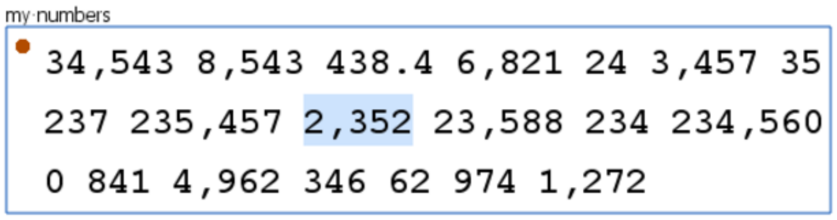
An example, regarding a file named 'my numbers.infra', containing a list of numbers might look like:

And once opened:



```
34,543 8,543 438.4 6,821 24 3,457 35
237 235,457 2,352 23,588 234 234,560
0 841 4,962 346 62 974 1,272
```

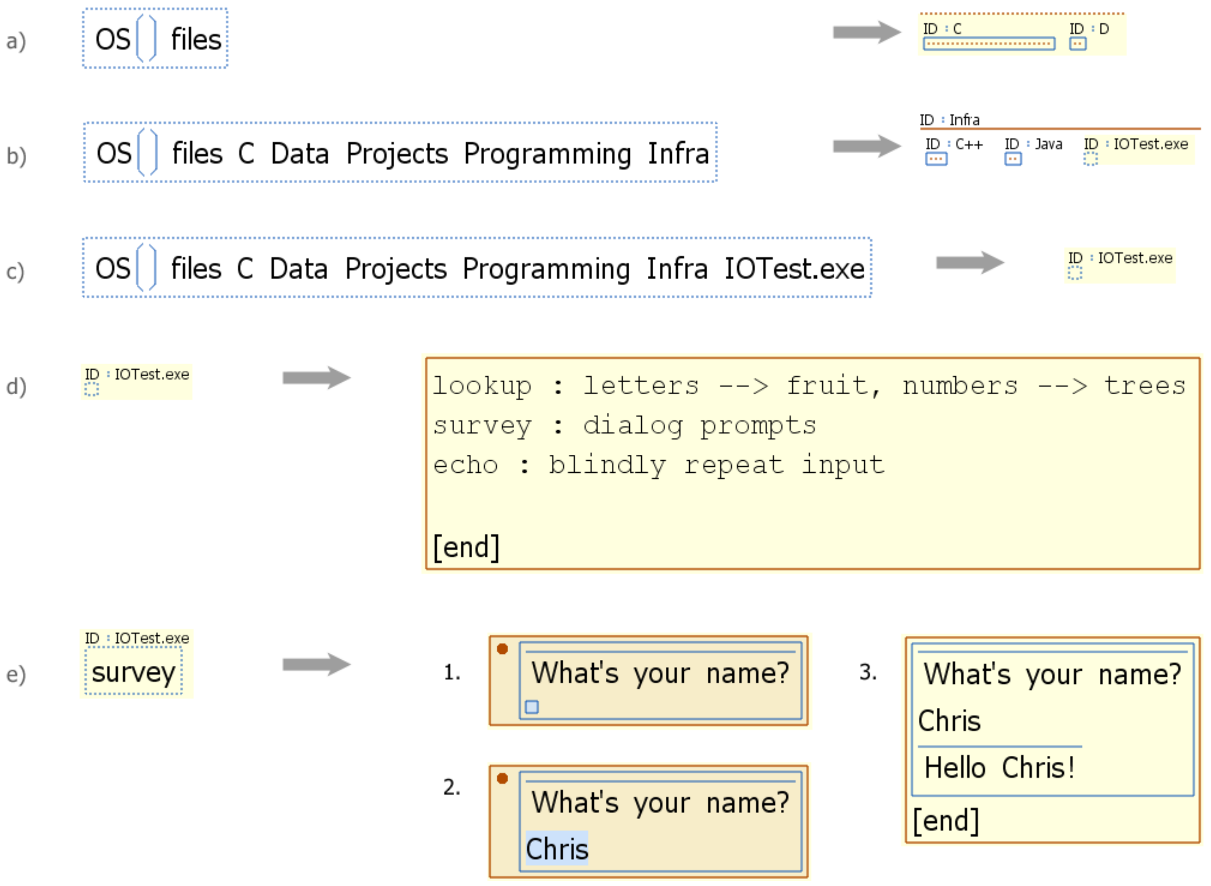
Editing the values in a file can function exactly like editing the result of a Patch. The red 'recording light' has the same semantics - namely batched editing, resembling that of the classical document load-edit-save paradigm. Similarly, when disabling the recording light, edits become direct to the live file buffer (continuously saved). As discussed in section 3.4, Free segments support opportunity for constant-time file updates.



```
● 34,543 8,543 438.4 6,821 24 3,457 35
  237 235,457 2,352 23,588 234 234,560
  0 841 4,962 346 62 974 1,272
```

Executable-Process Interface Infra-encoded files can by definition be loaded as Infra trees. Plantext files can be loaded as single long strings. Generic binary data can be loaded as a single long byte array. But when it comes to an executable, it may best be loaded as a Patch. This is to provide a place to provide input arguments - on the un-evaluated Patch.

The following example figure shows five snapshots along the path of interacting with an external process.



```
lookup : letters --> fruit, numbers --> trees
survey : dialog prompts
echo : blindly repeat input

[end]
```

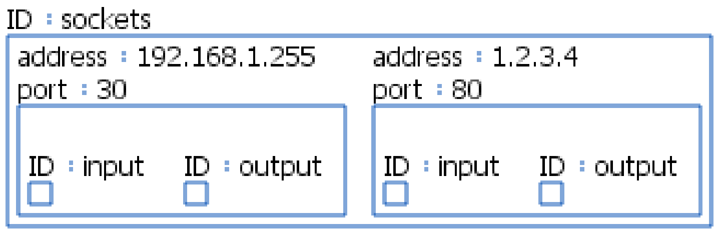
1.
```
What's your name?
```

2.
```
What's your name?
Chris
```

3.
```
What's your name?
Chris
Hello Chris!
[end]
```

The first two steps are just partial paths to the file. The fourth is execution with no input arguments, resulting in a usage string. The fifth step shows ‘survey’ as an input argument and the second sub-step is the user typing ‘Chris’ and submitting the value using the save key.

Socket Binding Interface The following figure is an example of what the 'sockets' tree might looks like after inserting two children, which spawns individual socket native-service objects, and giving them appropriate 'address' and 'port' metadata for the socket to actually bind. (No actual data has yet been send or received.) The acquisition of the necessary metadata for objects like this can be form-driven to help the user know what metadata is relevant. Valid forms for any subtree can be described using 'schema' metadata (not shown), and the editor can use a schema to direct/assist a user's editing.



From this point on, the input and output lists behave the same as the standard IO streams.

5.3 Runtime Language Reflection

Since Infra is an infrastructure based around directly authoring and editing structured data, there is a natural mapping between the internal data structures of a programming language runtime and human-readable Infra data structures. For programming languages that feature runtime reflection, these mappings can be automatically supported without having to prepare an adapter for each data type in advance. Reflection makes it possible for the library to dynamically assemble representations for any object without the need to run pre-processors on source code or be involved at compile-time. Runtime objects can be visualized on demand, by the Infra medium. The Infra medium also naturally brings with it the interactions necessary to manually assemble argument values into and invocation of native functions/methods. This essentially allows Infra to act as a visual debugger for the runtime environment that the editor implementation is running in.

We leave the details of these Native-Service Objects for future work.

Chapter 6

Second-Order Infrastructure

6.1 Monitoring Edits for Smart Patch Recalculation

When a user causes mutations to the tree, Patch output values may need to be recalculated. Changes made to the source values appearing in an output value do not need to trigger recalculation (those are handled by regular model-view updates). Only changes that influence *which* value a Patch outputs need to trigger recalculation. Without any kind of smart filtering, all Patches would have to be marked for recalculation with every edit occurring anywhere in the tree. It is especially important to only recalculate Patches when necessary because Patches can depend on the results of other Patches, causing cascading recalculations. This problem is similar to formula recalculation in spreadsheets, but expanded to a hierarchical tree structure.

Since Patches have to have been evaluated for there to be a value to refresh in the first place, dependencies can be gathered empirically during execution. As navigation commands move a virtual cursor around the tree, each potentially introduces an additional dependency.

Navigation Codes: Dependency Description

- **parent**(): depends on the continued presence of a specific child in a specific container.
- **count**(): depends on the total number of children in a container.
- **index**()/**child**(i): depends on the index position of a specific child.
- **previous**()/**next**(): depends on the spacing between two children in a container.
- **UID**(v): depends on the continued presence of UID metadata, and its specific value.
- **child**(key()): depends on the continued presence of the matching child, that its key does not change, and watch for the introduction of new left-siblings with a matching key.
- **ID**(v): depends on the continued presence of ID metadata on the target, that its value does not change, and that no closer descendants introduce matching ID metadata.
- **metadata**(): No monitoring necessary

Event subscriptions can be registered with each node as the are encountered during navigation. There are four events that occur when committing edits to the tree, any subset of which can be subscribed to on any node. These events are also leveraged by the user interface to refresh views. The goal will be to combine these in a manner to address the dependency categories above.

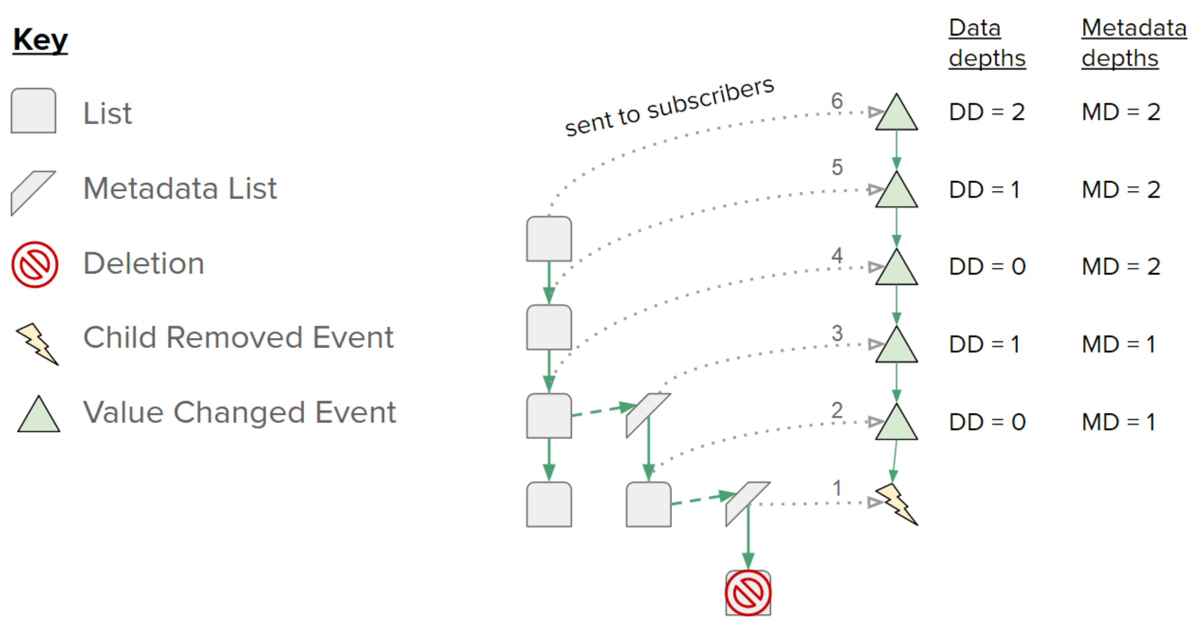
Subscribe-able Edit Events:

- **Child Replaced** - This event is always followed by a **Child Removed** event and then a **Child Inserted** event for the same index. Thus, it allows a deletion and insertion to be treated as atomic in update logic. Removal and insertion events are always issued in addition, so that update logic can safely ignore this event type while handling insertions and removals individually. Events are batched in a global event queue so that events can be delivered in a breadth-first type manner guaranteeing that **Child Removed** and **Child Inserted** do always appear back to back. A direct and immediate delivery scheme would potentially allow other events to be inserted between them, when an edit event just as immediately triggers further edit events.

- **Child Removed** - contains the index and reference of the removed child

- **Child Inserted** - contains the index and reference of the inserted child

- **Descendant Changed** - the only event that bubbles up the tree from an edit site, firing in each encoding-parent up to the root. As it travels up, it increments the relative tree-depth of the edit, and the number of ascensions from metadata along the way. This informs the receiver of how far away (data depth) and how many layers of metadata (metadata depth) the change was from it. See the following diagram for details of depth counting.



A Patch execution can build higher-level observers out of combinations of subscriptions to these basic events. We call these higher-level observers monitors. We identify six classes of monitor that precisely cover the conditions that can cause a change in Patch result.

- **Monitor Index**(*parent, i*) - subscribes to **Child Inserted** and **Child Removed** events from *parent*. It triggers only if the inserted/removed child has an index less than or equal to *i*.

- **Monitor Presence**(*parent, i*) - subscribes to **Child Removed** and **Child Inserted** events from *target*. It triggers when the child at index *i* is removed. In the mean time, removal events for an index less than *i* decrement the value of *i* being monitored, and insertion events for an index *less than or equal to i* increment it.

- **Monitor Index Spacing**(*parent, i1, i2*) - subscribes to **Child Inserted** and **Child Removed** events from *parent*. It triggers only if an insertion/removal occurs at an index value in the range between *i1* and *i2* (inclusive). Also, each value *i1* and *i2* are incremented or decremented respectively in response to insertion and removal events taking place at an index value below their own.

- **Monitor Count**(*target*) - subscribes to **Child Replaced**, **Child Inserted**, and **Child Removed** events from *target*. It triggers if insertions or deletions happen without having been preceded by a **Child Replaced** event.
- **Monitor Value**(*target*) - subscribes only to **Descendant Changed** events from *target*.
- **Monitor Descendant Metadata**(*root, metadata entry, depth cap*) - subscribes only to **Descendant Changed** events from *root*. It triggers only if the event occurred above the relative tree-depth of *depth cap*, and if the metadata depth of the change is exactly 1, and if the changed metadata value at that depth matches the value of *metadata entry*.

Now we will outline the mapping between navigation commands and monitor objects. In the following listing, *src* refers to the tree node being navigated from, and *dst* refers to the tree node being navigated to.

Navigation Commands: Subscribes to Monitor

- parent(): **Monitor Presence**(*dst, src.index*)
- count(): **Monitor Count**(*src*)
- index(): **Monitor Index**(*src.parent, src.index*)
- child(*i*): **Monitor Index**(*src, i*)
- previous()/next(): **Monitor Index Spacing**(*src.parent, src.index, dst.index*)

- UID(v): **Monitor Presence**(*dst.metadata, i*), and **Monitor Value**(*dst.meta.child[i]*), where i is the index of the matching UID(v) in *dst.metadata*.
- child(*key*()): **Monitor Key Query**(*src, dst.index, key*)
- ID(v): **Monitor Presence**(*dst.metadata, i*), and **Monitor Value**(dst.metadata.child[i]), where i is the index of the matching ID(v) in *dst.metadata*. Also, to detect shadowing, **Monitor Descendant Metadata**(*dst.parent*, ID(v), *dst.parent.depth - src.depth*).
- metadata(): No monitor necessary

By the time a Patch completes evaluation, it has built up a collection of monitors. Once any of these monitors trigger, the Patch will be notified. It will then mark itself as needing recalculation, and clear all of its monitors and have them cancel their respective subscriptions.

Instruction Pointer As a Patch executes, its internal instruction pointer moves through its own tree to track which command to execute next. If edits were to occur to this tree, they could also obsolete the Patch's result. It is easiest to handle detecting this with a single subscription to **Descendant Changed** events from the Patch's root. However, if tighter precision were required, specific monitors could be analogously placed on the instructions themselves as they are executed, following the empirical control flow of that execution.

6.2 Encoding Plan Models

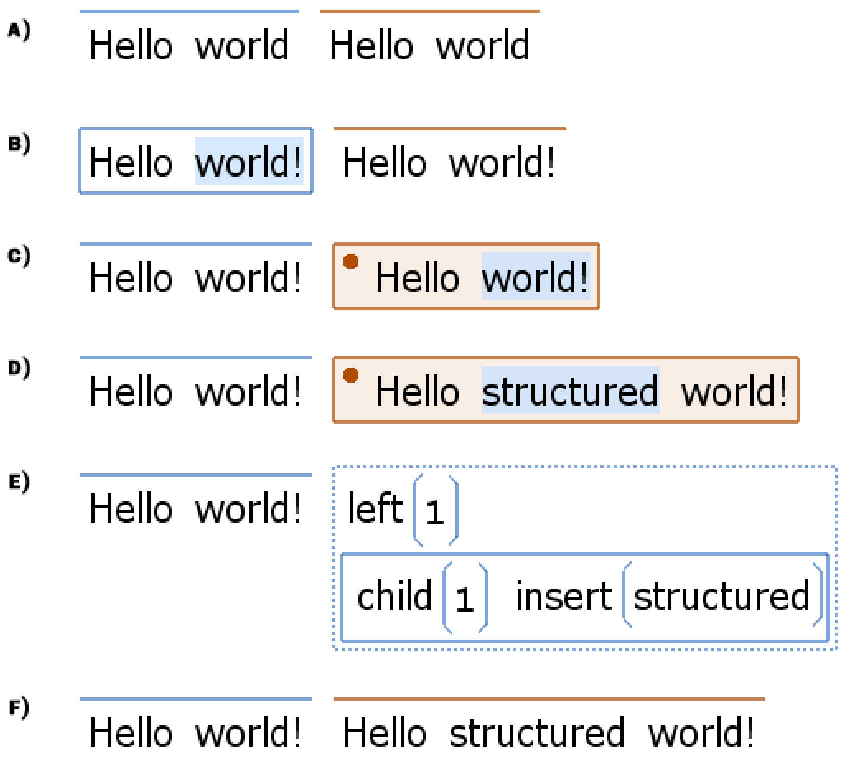
An Infra encoding in a byte buffer can be updated incrementally, or lazily all at once. When writing out an encoding from scratch, the byte lengths of the various segments have to be tabulated first since the segment headers must be specified up front. Lengths can be influenced by metadata - encoding metadata can request certain kinds of Free segment padding. Such padding values may even be informed by past editing statistics and expected reallocation sizes that will avoid the need to shift data in the future. (See Base type: Free)

We refer to structures of pre-calculated headers as Encoding Plans. Once built, a Plan is used in tandem with the in-memory tree to output the tree's byte encoding.

6.3 Authoring Patches By Demonstration

The commands within a Patch can of course be authored manually like any other data, but more conveniently, an editor can automatically synthesize them in response to a user's attempted edits to the Patch's output value. This is similar to the concept of recording a macro to automate tasks in applications such as Microsoft Excel, except modeless and local to a specific subtree. A Patch can be initially created using the familiar Copy and Paste commands. This seeds the Patch with a path to the node that was copied.

but the design of Patch semantics is such that the opcodes mimic a similar set of discrete operations that the editor user-interface provides to users. This means that editors can allow users to edit the output value of a Patch directly, and trivially generate corresponding commands. However, Patch models can be built up without ever having to directly write or see their internal instructions. By 'recording' the modifications users make to a Patch's output value, the necessary Patch commands to make those same changes can be synthesized programmatically.



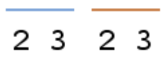
Edit Capture When a direct edit is attempted in the UI, the edit event is passed up the tree. If it makes it out the top of the root node, the edit event is applied. If the event encounters a Patch that is in 'record mode', the edit will be transformed before it continues on its way up the tree. The transformation is a conversion from a direct edit, to the appending of Patch code.

6.4 Authoring Function Calls by Example

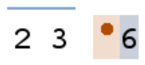
When a user edits a Patch result, they are effectively providing input/output mappings of some potentially more general transformation. This forms the basis for natural programming-by-example features in an Infra editor. We have implemented a toy version of automatically generalizing individual operations made by the user.

As an example, we will go through how a user can author a program to multiply a list of numbers together without writing any ‘code’. First, let us just walk through what a user might see, then go through it again from the perspective of the editor.

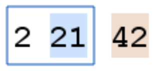
1. Type a list of numbers, say 2 and 3, then reference them with a Patch (using copy/paste).



2. Edit the pasted value to be a ‘6’ instead of the list.

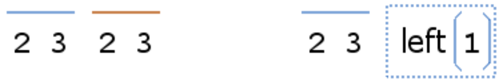


3. Modify the source list to exhibit new input data. See the updated product immediately.

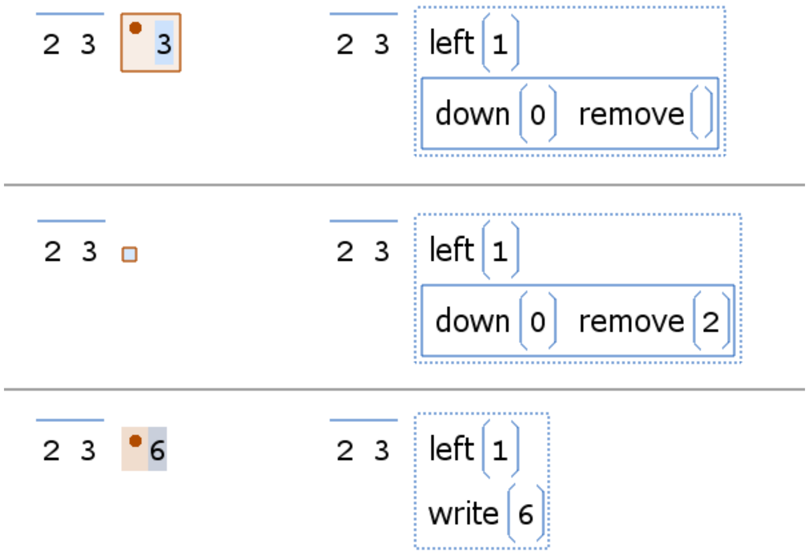


Now let us go through that sequence again with behind-the-scenes detail. In the following figures, the left side will reiterate the evaluated view, and the side side will show its un-evaluated view.

1. After the copy and paste.



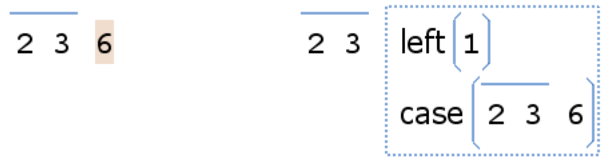
2. Changing the patched view of the list may consist of several incremental changes, such as deleting each of the two children before typing the ‘6’ to replace the now-empty list. Reminder: the remove command’s argument is the quantity of sequential elements to remove (defaulting to one if omitted).



```
2 3 3        2 3  left 1
                  down 0  remove

2 3          2 3  left 1
                  down 0  remove 2

2 3 6        2 3  left 1
                  write 6
```

Note that the refactoring engine is attempting to keep the code simple by catching opportunities to merge or eliminate statements. The second ‘remove’ was able to be merged in with the first, and ultimately the ‘remove’ commands are rendered irrelevant by a write that replaces their root.

When the cursor is moved off the Patch, the engine detects that the material being referenced (the list of ‘2’ and ‘3’) is discarded entirely by the modifications assembling the output value - the ‘write(6)’. The logic of explicitly referencing a value that you deliberately do not use is the basis for the opportunity to record a ‘case’ statement to capture the input-output pair as a training example. The situation would be entirely different had the navigation instructions never been there - the ‘left(1)’.

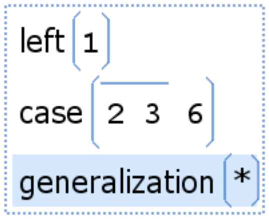


```
2 3 6        2 3  left 1
                  case 2 3  6
```

Reminder: the second argument of a case statement is the value that is returned by the Patch if the Patch's current focus matches the case's first argument. Therefore, it reads as, if the virtual cursor is currently pointing to a list with '2' and '3', then halt and return '6'.

If the user came back to edit '6' further while the input list was still '2 3', this same case statement's second argument would be updated directly. If the input list was not '2 3' and the output value were edited, an additional case statement would be generated for the new input-output pairing.

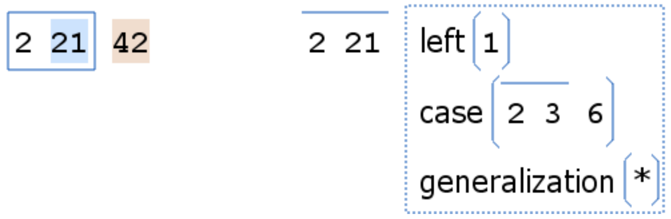
The collection of case statements form a suite of constraints that must be adhered to by any attempt at automatically generalizing the user's algorithm. A background thread opportunistically runs through the library of known functions, looking for one that returns each case's second argument when fed each case's first argument respectively. (Because of the Effect System, these functions can be treated safely as pure functions.) All functions that pass all case statements are recorded in the Patch as the arguments of a 'generalization()' instruction.


```
left 1
case 2 3 6
generalization *
```

When a generalization statement has more than one argument, there is a definite need for more constraints to narrow things down. The editor could also draw the user's attention to the list of found options for their insight on which was intended. Such a dialog can even be posed in terms of having the user provide one or more 'answers' to intelligently synthesized hypothetical inputs detected to resolve the ambiguities in as few trials as possible. We leave this kind of high-level functionality for future work.

In this particular case however, the system will find only one library function that returns a '6' when given a list with '2' and '3'. When there is only one option in a generalization, the editor can apply it confidently and transparently.

3. When the original list is modified, dependent Patches are updated. When this Patch's code runs, the case statement will not match and execution will pass through to the generalization command. It applies a multiplication to the value in focus, and since that is the last statement in the Patch, its focus ends on '42' and is the final output.

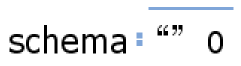


This same sequence applies equally well to a range of useful interactions such as sorting a small list to establish sorting for a big list, and even string manipulations such as auto-capitalization of the first word in a sentence (and therefore auto de-capitalized when moved out of first position), as long as the editors function library is creative enough to include useful word-processing functions.

6.5 Type System

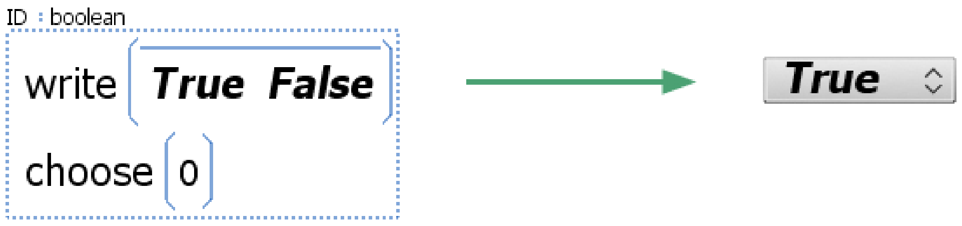
A type system for Infra can be homoiconic (looks like the thing it describes) because the Infra encoding is self describing. Since each Infra segment already includes a type header, having a static type description is just a matter of isolating the tree of headers from a prototypical instance. In other words, an Infra tree can be used directly as a description of a structural contract for another. There is no need to define a new language specifically for defining types. The standard way to associate such descriptions with an element is through 'schema' metadata.

For example, to assert that an element consists of a string followed by an integer, its metadata would include:



Like any value, the schema value can exist in a shared part of the tree and get patched in (referred to) for every occurrence. If the type-checking engine were to use object identity for type compatibility testing, the behavior would match that of static-typing rules. On the other hand, if the type-checking engine were to use equality, the behavior would match duck-typing rules.

Disjoint union types (or sum types) can be expressed with Choice-Point objects (generated by Patches with the 'choose' opcode). A good example of this is a definition for boolean.



6.6 Schema-Guided Editing

If an element has schema metadata associated with it, an Infra editor can use that information to help the user during editing. This can be as simple as pointing out when the shape of the data is not compliant with its schema, or as heavy handed as constraining invalid edits completely. Integrated development environments (IDEs) often limit themselves to the former, using red underlining to indicate compilation errors. At the other extreme, graphical user interfaces often aim for the latter, aiming to guide interaction as much as possible to prevent inconsistent states from occurring in the first place.

See section Related Work: Structured Editors.

6.7 Suite of Converters (Parsers and Renderers)

Converters transform an Infra tree into a different Infra tree. This can be as simple as reversing the order of items in a list, or as elaborate as recognizing/parsing the structure of a binary file format and generating its representation as an Infra graph. Having a suite of converters on hand in an editor helps Infra interface with the existing landscape of text-based and non-human-readable binary formats. Converters also include renderers, which specialize in the opposite process - taking a labeled structure and flattening them to a text or byte array using the syntax of a specific external format. A best practice for a parser would be to include a format tag in metadata on its output so that a corresponding renderer can later bring the data full circle, targeting the original syntax.

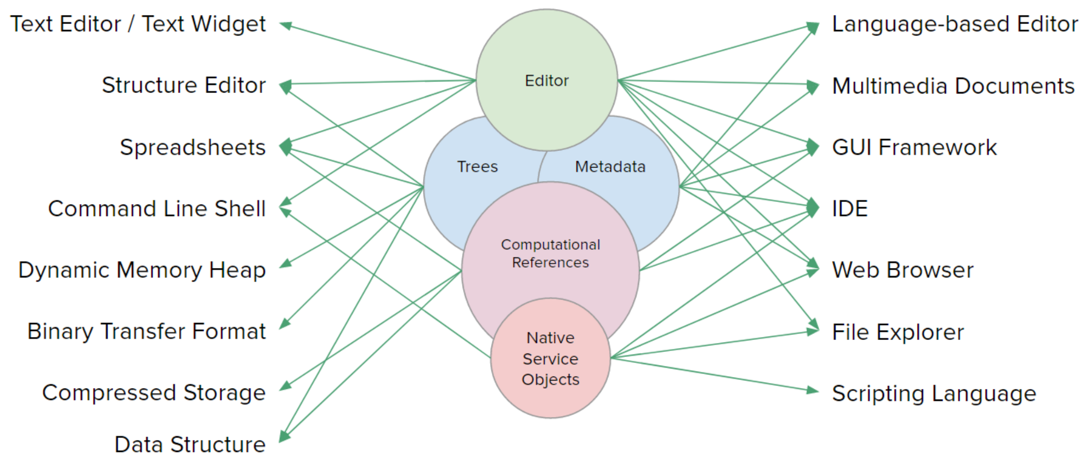
We found it useful to interface converters to the editor with a two-phase pipeline. A converter can be asked to provide an assessment of its ability and confidence to perform its conversion on any given Infra tree. These assessments can be aggregated and ranked by confidence, then applied at the discretion of the user interface design. The confidence rankings are useful on two fronts: for applying certain conversions automatically, for filtering irrelevant conversions when the user right clicks for a list of actions pertaining to their current selection.

Chapter 7

Applications

The purpose of Infra is to be a better lowest-level building block for computing. By nature, it is applicable to any scenario involving encoded information, which is to say that it is technically relevant to any computing task. Infra's unique angle is in bringing the properties for human readability and direct authorability to the realm of machine-friendly binary encoding. The common encodings used across computing do not always worry about having general properties beyond those necessary for their specific role, and are stuck making compromising choices between prioritizing efficiency or human readability. Infra is best used as a unifier and equalizer - reducing the need for as many varieties of encoding and, most importantly, raising the common denominator of how they can be interacted with.

As a baseline, Infra can absorb significant portions of traditional application scenarios and modalities across personal computing. What would normally be a unique software stack for each, can now be mostly or entirely overlapped. Infra is designed as a cohesive mutually-reinforcing whole, but can be seen as roughly four component layers: the encoding (trees with metadata), Patch evaluation (runtime and effect system), the extensible library of native-service objects (computation and I/O), and Infra editors (human readability, direct authoring of the encoding, and data-driven presentation). The following two diagrams break down fifteen traditional application types in terms of what Infra components are *most* salient in superseding their characteristic functionality.



Traditional Application	Structure (Trees)	Metadata	Computational References	Native-Service Objects	Editor
Binary Transfer Format	All				
Dynamic Memory Heap	Free regions				
Data Structure	All		Inter-reference		
Compressed Storage		Identities	De-duplication		
Scripting Language				All	
Text Editor / Text Widget					✓
Structure Editor	All				✓
Language-based Editor		Schema			✓
Multimedia Documents		Presentation			✓
Spreadsheets	Table		Inter-reference	Logic / Arithmetic	✓
GUI Framework		Presentation	Abstraction	All	✓
Command Line Shell				OS integration	✓
File Explorer				File System Tree	✓
Web Browser		Presentation		Sockets	✓
IDE		Schema	Refactoring	OS integration	✓

Beyond emulating a spectrum of traditional personal computing modalities, the fact that Infra unifies their fundamental ingredients into a single medium enables novel combinations of their expression (leading to a much higher bar of consistency and amortized learning curves for interaction) - such as hierarchical spreadsheets containing structured data, or a file explorer with inline editing of files, or a user interface that can be extended on the fly, or a command-line shell with syntax-directed editing and high-level abstractions.

7.1 Case Study 1: URL Syntax

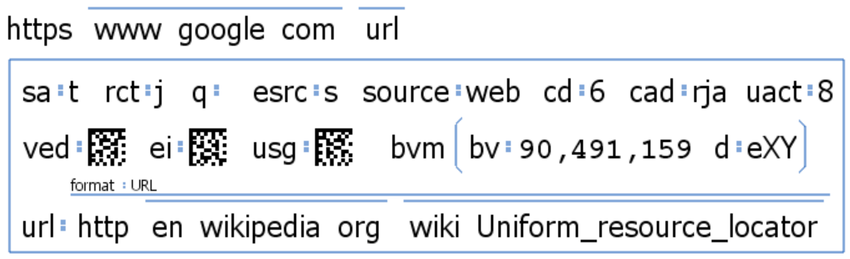
Let us explore a hypothetical alternative reality where computing's text infrastructure never consisted of only characters codes, and was built up around a parametrized syntax such as Infra. The UI widgets used for everyday tokens of input/output (such as Text Fields) would be Infra editor widgets and literacy around using keyboards would think in terms of authoring structure along with values. In this section, we explore the effect this would have on the nature of computing by focusing on an everyday unit of structured information - a URL.

The following URL is a link provided by a Google search result. It is the link to the Wikipedia article on Uniform Resource Locator, but the actual URL is a redirect through Google's servers for accounting purposes. This kind of URL is chosen because it is representative of complex stateful URLs as well as the fact that it contains a URL inside itself.

```
https://www.google.com/url?sa=t&rct=j&
q=&esrc=s&source=web&cd=6&cad=rja&uact
=8&ved=0CDkQFjAF&url=http%3A%2F%2Fen.w
ikipedia.org%2Fwiki%2FUniform_resource
_locator&ei=tE8sVe-iF4m8ggSZi4T4AQ&usg
=AFQjCNFVKoOa_HlcuDeUu8wYS_g70me4Kw&bv
m=bv.90491159,d.eXY
```

Note that this URL is not very readable, and that the embedded URL is escaped and does not work if copied to a browser address bar as is. The bulk of the characters in URLs like this are Base64-encoded bit strings. Base64 encoding is born out of the fact that human-readable formats are unfriendly to binary data, and in the web world, there is even further need for compromises to encoding in URLs, to avoid having to escape plus and forward slash characters.

Now let us jump to looking at how URLs could have formed differently if Infra boxes existed before text boxes did.


https
www google com
url
sa t rct j q esrc s source web cd 6 cad rja uact 8
ved ei usg bvm bv 90,491,159 d eXY
format URL
url http en wikipedia org wiki Uniform_resource_locator

- The elements of a domain name do not have to be separated by punctuation. It can simply be a list.
- This applies the same way to the path component of a URL.
- The query fields are key-value pairs and can be grouped together.
- Numbers stay binary encoded. (As they were in the memory of the computer that constructed the URL.)

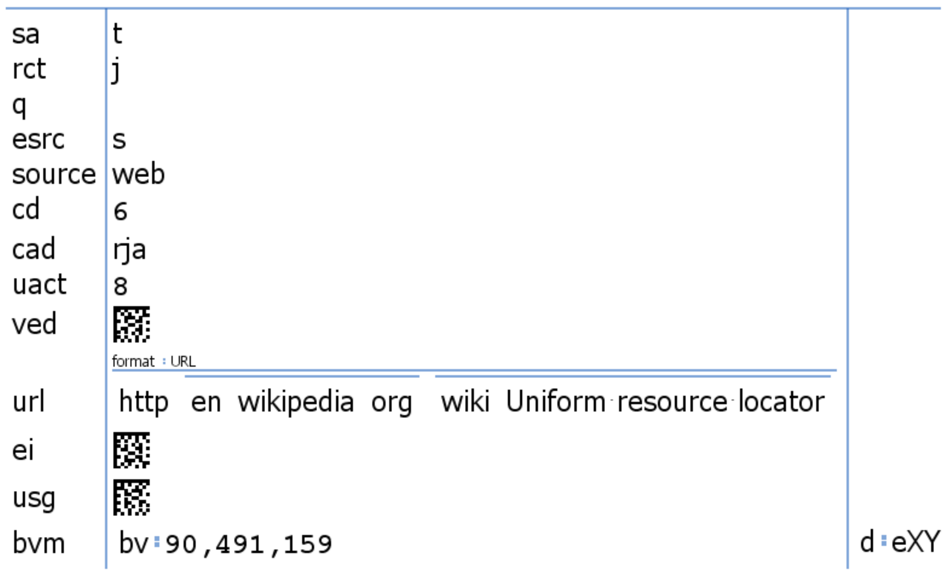
- Bit strings can stay bit strings without the need to use indirect representations such as Base64. These bit strings are displayed as a Data Matrix (one of many possible visualizations at the disposal of the editor UI such as Chroma Hash). The use of a data matrix allows for a compact display of a binary value that does not necessarily need to be readily deciphered by a human, while giving some ability to judge equality. In this case of reverse engineering Google's URL, It is not obvious if the values 't', 'j', 's', and 'rja' are also meant to be treated as Base64. With Infra, such an ambiguity would not have to exist.

- The nested URL does not have to be escaped, in fact, it is also parsed, and even labeled as being a URL with 'format' metadata.

- Underscores do not have to be used as a substitute for spaces.

- The 'bvm' value can have the substructure it seems to want. In this case it was parsed into two values separated by comma, and then sub-split into key-vals by period. The '90491159' portion is numeric and is encoded more usefully as a binary integer - able to be displayed according to the user's preference for localized digit-grouping.

Since the query fields are grouped together, they can conveniently be displayed in a tabular arrangement at the request of the user. From this layout, the information structure is quite clear, and it is easy to notice that 'bvm' is the only field to have more than one associated value.

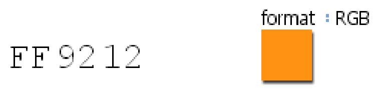


sa	t	
rct	j	
q		
esrc	s	
source	web	
cd	6	
cad	rja	
uact	8	
ved		
	format : URL	
url	http en wikipedia org wiki Uniform resource locator	
ei		
usg		
bvm	bv:90,491,159	d:eXY

This particular example case up in real life when we were trying to extract the forwarding address, by hand, from a URL like this. Not only is unescaping an escaped URL by hand tedious and cryptic, but it also requires using an ASCII table for reference. However, in the hypothetical case of Infra-based URL syntax, extracting the forwarding address is trivial. As would quickly performing surgery on a URL before sending it to someone. We find ourselves often manually tweaking video links (such as YouTube) to either remove the playlist portion (so it only takes them to the specific video) or to nudge the timecode it will take them to (because we hit pause a little late before copying the generated link). In the text world, doing these simple kinds of things requires familiarity with URL's specific meta-characters, rather than just being the same kind of structured editing across all user interfaces.

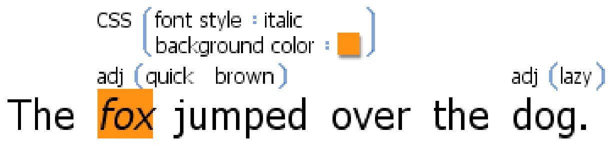
7.2 Case Study 2: Data-Driven Presentation

One of the many uses for metadata is to hint to Infra editors/browsers what abstractions are appropriate when displaying a particular piece of data.



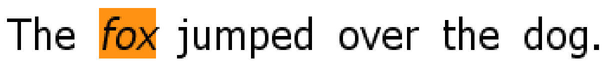
On the left side of the figure above is a byte array of size three displayed in hexadecimal by the editor. On the right side is the same element after the user added 'format' metadata. As it happens, this editor recognizes 'format' markup, and the value 'RGB' gives the editor confidence to instead display the byte array as a color swatch, which can even be interacted with as a color picker, making editing the value much more intuitive.

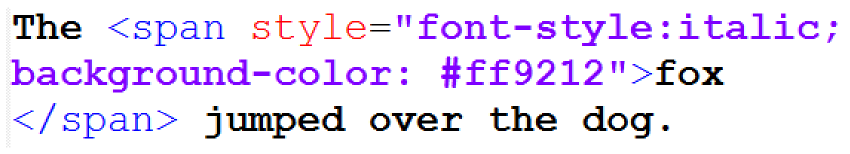
Our prototype editor also supports a subset of the CSS standard, which makes use of color values, so let us combine this example with the previous one. In this scenario, the same metadata exists on the color value in the "background color" property, which happens to itself be metadata. (The 'format' metadata is not shown here because it is at least two levels removed from the current position of the selection cursor. Moving the cursor to the first metadata level will expand it.)



There are several noteworthy aspects to this structure. Several grammatical constraints are relaxed relative to typical CSS due to Infra circumventing the bottlenecks of a tokenizer. The style property names can have spaces in them, rather than being forced to use hyphens to separate words. The byte encoding of the color value is in binary, which is more compact than "#ff9212" by a factor of three and moves the parsing to author time rather than render time. As we will explore later, Infra encodings can also use its Patch base type and metadata layers to bring string de-duplication and value computations to CSS or any other application.

If an editor displays metadata layers off in a side panel or the user toggles their display, this rendering in the editor will resemble the browser output for the following HTML. This is powerful because the true direct comparison of human-readable directly-authorable encodings is between Infra and raw HTML, not between Infra and rendered Web pages.


The *fox* jumped over the dog.


```
The <span style="font-style:italic;
background-color: #ff9212">fox
</span> jumped over the dog.
```

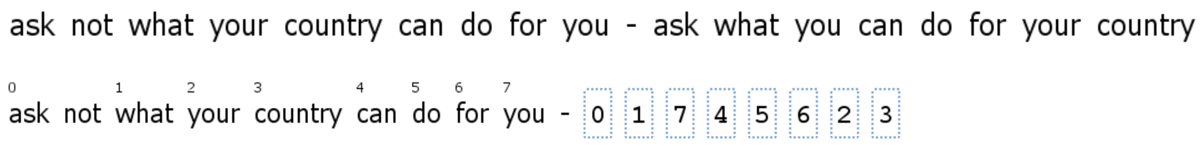
7.3 Case Study 3: Plain Text at Scale

This section briefly explores the cost of storing abstract structure within content in the way that Infra proposes all data be authored and stored. There are varying degrees of structural breakdown, hierarchy, and interconnection possible with any kind of data. For starters, we will look at just a basic first pass of sub-structure that can be given to most plain-text content - tokenization.

We have tokenized a sample of English texts and source code files within Infra to measure an average byte overhead introduced by Infra's element headers, which segment each word. Infra editors display whitespace padding between elements, so actual space characters are not needed between words. Elements of fewer than 15 bytes only require a 1-byte header, and so most of the time, the presence of the header byte is made up for with the lack of need for a space character. However, newlines are a common occurrence in text and are not usually paired with adjacent whitespace. In all cases tried, the byte overhead was less than 4%.

For the full text of Lewis Carroll's "Alice's Adventures in Wonderland", the byte size increased from 163,815 bytes to 169,096 bytes when tokenized simply by splitting on space characters. This is an Infra overhead of **3.2%** to have structure at the word level. But, now that there is word-level structure, Patch can be used to de-duplicate strings by encoding a common word once and referring to them from the locations where they are used. As long as the byte size of the Patches themselves is smaller than the word they reference (minus the one-time cost of metadata to number the word), memory will be saved. In the case of Alice in Wonderland, the storage size can be reduced by 44,206 bytes (**26.1%**) through basic string de-duplication.

For an example of what this kind of Patch usage looks like, let us take the famous quote from JFK:



ask not what your country can do for you - ask what you can do for your country

ask^0 not what^1 your^2 country^3 can^4 do^5 for^6 you^7 - $\boxed{0}$ $\boxed{1}$ $\boxed{7}$ $\boxed{4}$ $\boxed{5}$ $\boxed{6}$ $\boxed{2}$ $\boxed{3}$

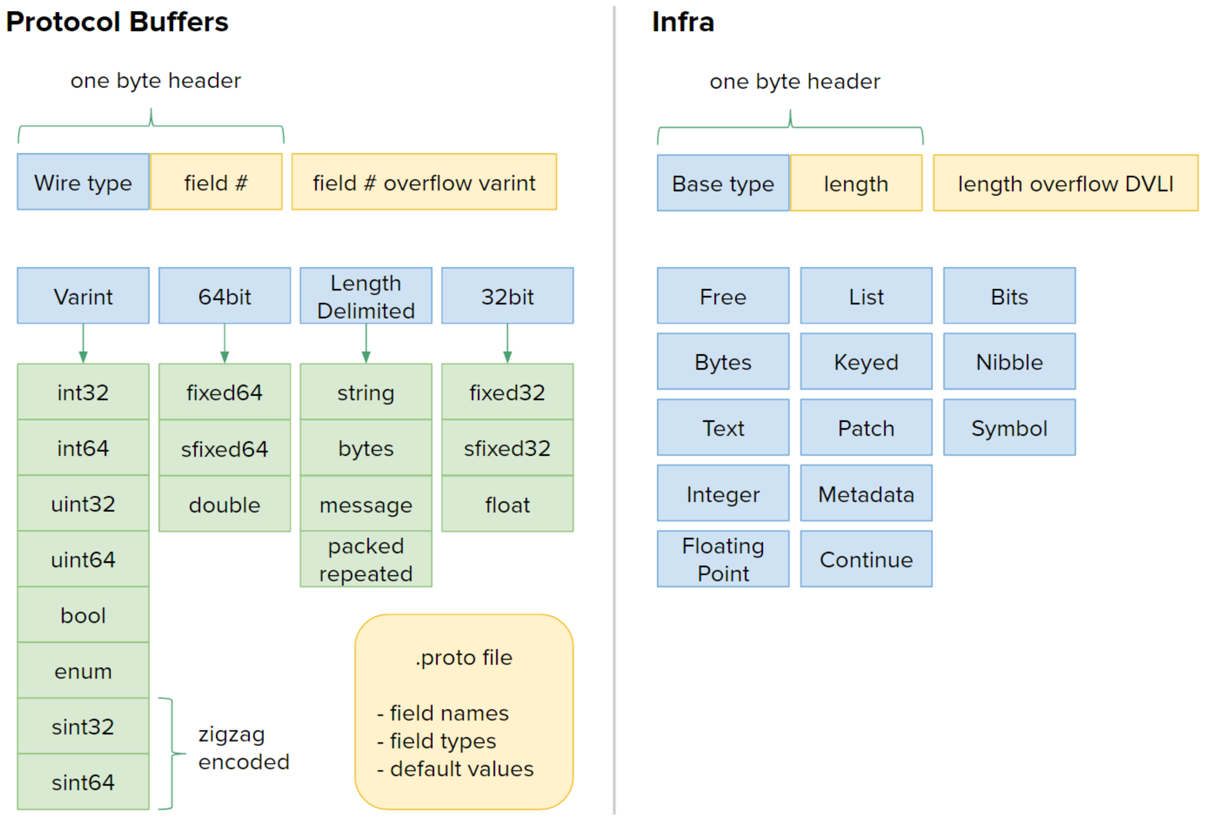
The second row shows 'UID' metadata and Patches unevaluated. (Reminder: Metadata and Patch both have shorthand for ID and UID when using strings and numbers respectively, which is why the metadata does not appear as "UID:4" and why the Patch commands do not appear as "UID(4)".)

String de-duplication is a simple form of data compression, but importantly, this is not a compression scheme that obfuscates the data format. Patches are referentially transparent, and so substituting an element for a reference to the same value is a non-disruptive transformation.

7.4 Case Study 4: Protocol Buffers Replacement

Infra can be leveraged as a transfer format - a library for marshaling and unmarshaling data in the traditional metaformat sense. It has an efficient binary encoding on par with the RPC-oriented metaformats, yet always comes with the option to view and edit data in serialized form.

As far as compact high-efficiency serialization formats go, Google's Protocol Buffers [18] are, by our estimation, the most widely known, used, and supported in a modern setting. In this section, we will refer to it simply as 'Proto'. Overall, Infra has roughly the same byte efficiency as Proto. Both Infra and Proto precede elements with a one-byte header split into a type enumeration portion, and a scalar quantity portion. Also in both cases, the header is conditionally followed by a variable-length unsigned-integer encoding to allow the scalar quantity to overflow into more bits.

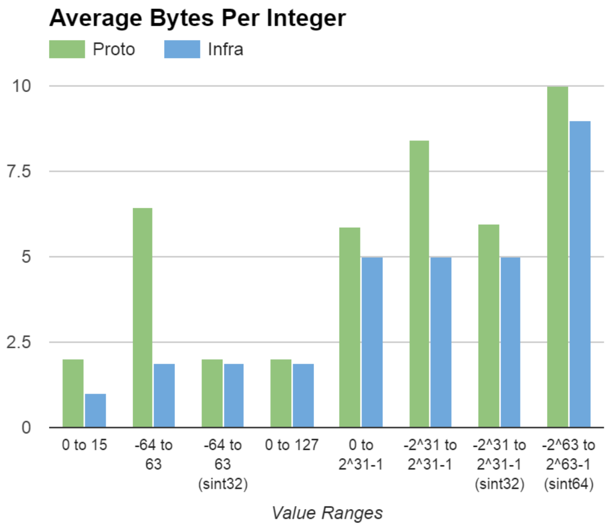


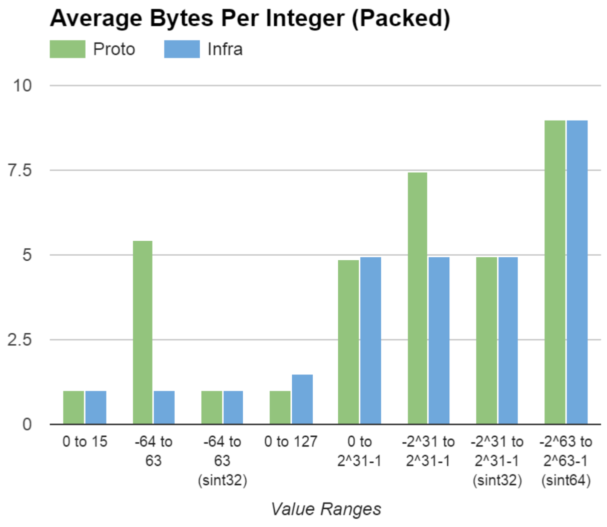
The performance of Infra and Proto are tricky to compare directly because they are designed for nearly opposite circumstances. Proto was designed to be manipulated procedurally by pre-compiled code, and to eliminate as much unnecessary exposition of the data on the wire as possible. Infra was designed to be viewed and authored directly in its encoded form, and to allow for as much exposition of the data on the wire as the user wishes to include. That being said, Infra can still be used in a constrained way as not to embed any more than the bare minimum necessary for Proto-like use cases.

7.4.1 Integer Encoding

In this comparison, we focus on the efficiency of encoding integer values, since that is where bit widths are the most dynamic, and where the bulk of the design complexity resides in Proto. Infra has two integer base types (Integer and Nibble), while Proto has ten (int32, int64, uint32, uint64, sint32, sint64, fixed64, sfixed64, fixed32, sfixed32). Infra can get away with effectively one integral base type because Infra's headers are parametrized by byte length, whereas Proto's headers are parametrized by field number.

For the following measurements, various trials of encoding a list of ten thousand integers in each encoding were performed. The integers were randomly chosen from a flat distribution. Trials vary in the range of random integer values chosen (small, large, negative) in order to exercise various phase changes in the encodings. The list is serialized using each encoding, and the total byte length of the serialization is divided by the number of elements (ten thousand) to arrive at an average number of bytes per integer. This averaging amortises away the one-time-cost portions of their byte overhead.



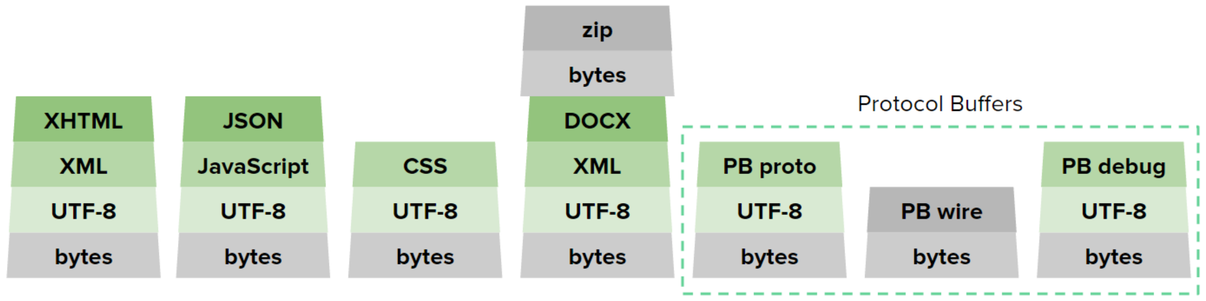


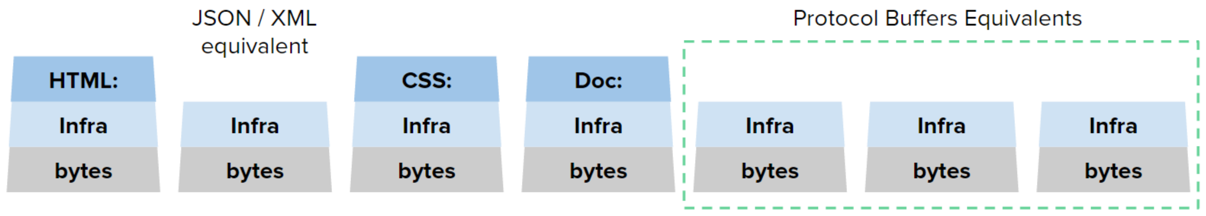
Value Ranges: Proto has unsigned integer types, signed integer type, and *sign-unspecified* integer types. Proto's variable-length integer encoding ends up being highly inefficient for negative values (two's complement) if a *signed* type is not specified explicitly. It falls on the user defining the .proto file to make a judgement call regarding the frequency of negative values that will appear in future data. This is the issue responsible for the measured spikes of inefficiency for Proto in the second and sixth value ranges. Those two ranges are run again with 'sint32' and 'sint64' specified in the .proto file respectively.

'Packed' Mode (Second Chart): Proto has a 'packed' option for repeated fields, in which it forgoes repeating the same tag header for every element in a list. 'Packed' can be specified in the definition of each repeated field of a scalar type or by declaring "proto3" syntax mode. Infra can do something analogous using the Bytes base type. Infra is able to have schema metadata associated with the byte array to inform how to decode it, but since Proto is built to only assume that any reader of the data also has the corresponding schema on hand, we make the same assumption during this phase of our comparison.

7.4.2 Schema Encoding

Infra defines one encoding while Proto defines three: its C-like .proto definition language, its wire encoding, and its JSON-like 'debug' strings. The .proto definitions and debug strings are human readable, the former is meant to be authored directly, and the latter is meant only for reading. The following diagram outlines how various formats layer their encoding semantics. The three Protocol Buffer encodings are grouped on the right. Other familiar formats are included for context and comparison.



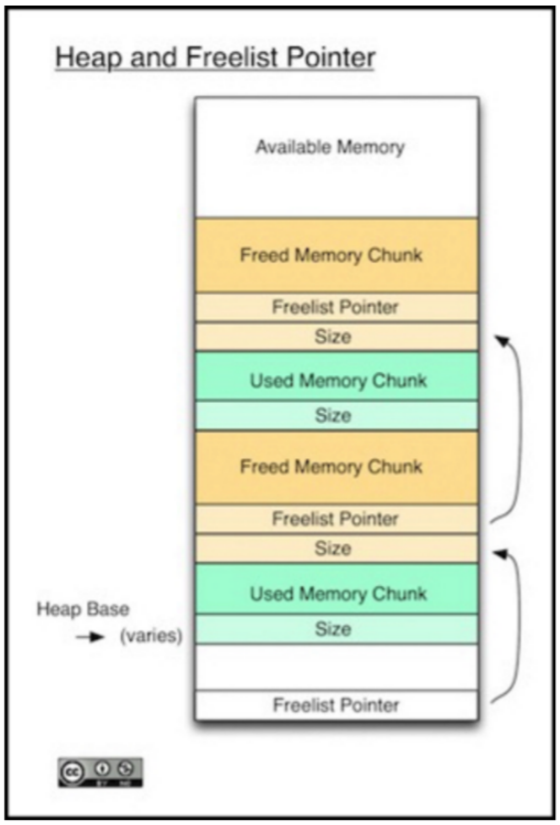


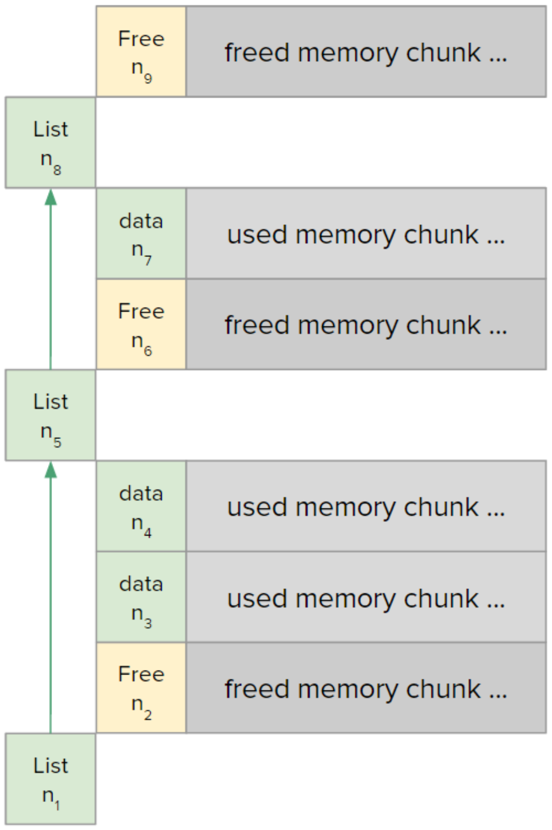
The second row describes a kind of alternate reality, where Infra is the encoding foundation of these formats instead of UTF-8 or a one-off binary scheme. In this arrangement, formats such as HTML and CSS are simply metadata channels on Infra encoded data structures. Metaformats such as JSON, XML, and Protocol Buffers would have much less reason to exist if Infra were an existing baseline offering both the high performance of binary encodings and even greater human-readability than textual encodings. Thus, vanilla Infra is shown as equivalent rather than imagining JSON, XML, and Protocol Buffers semantics on top of Infra. Infra can mimic any one of the three encodings defined by Proto simply by virtue of what layer(s) of information is included in a structure.

- Infra instead of **PB's .proto**: a data structure acting as a prototypical data instance (data that can later be used as schema metadata)
- Infra instead of **PB's wire**: a stream of data instances containing no copy or reference to a prototypical instance (data without schema metadata)
- Infra instead of **PB's debug**: a stream of data instances with a copy or reference to a prototypical instance as schema metadata

7.5 Runtime-Heap Encoding

Application Transparency: If it became common for applications to use Infra for internal objects and data structures, then the internals of applications could be inspected on demand. Any data missing or un-selectable in the user interface, could be manually obtained and processed by end-users. Users would have a much higher-level 'hood to lift' like they do with physical artifacts. A similar thing has happened with the web. Browsers now offer developer consoles so users can inspect and modify the DOM.





7.6 Infra Dialects of Existing Languages

Programming language grammars could be re-designed to use Infra rather than text to store their syntax. Such Infra dialects would not have to be parsed in the conventional sense since they could already qualify as abstract syntax trees, and their grammar could take advantage of encoding-level graph structure and meta-programming. Version control and merging could be done at the node level rather than at the line level, leading to fewer merge steps and fewer merge related bugs. Patches could even be used to communicate change lists.

The concrete syntax for a language like Java could have manifested many different ways, but had a long history of languages with C-like syntax to leverage. When it comes to Infra, which provides even more degrees of freedom for grammar design and encoding-level semantics, there is no obvious preferred way to express the equivalent structure of a Java program while taking advantage of Infra's benefits - yet. In other words, questions like: 'what aspects of the grammar should be relegated to metadata?', and 'should semantics like variable reference and behavior inheritance be relegated to Patches at the encoding-level?' have no objective answer. It must be left to usage patterns and natural evolution to establish a canonical, but uniquely Infra, way to formulate highly usable ASTs.

Imagine Infra encoding a language with identifiers. Each identifier could be a reference to the canonical reference name. Then the identifier could be renamed globally from one location without the editor having to understand any aspect of the language. The name of another identifier could even be defined in terms of the first, using Patch. If the first identifier were to change, all instances of the second identifier would change, as well as all instances of the first.

Patch can even be used to improve efficiency. When a user copies and pastes a node, the system keeps track of a shallow copy, saving memory. If the user makes a change to the copy or to the original, the system can choose to make a deep copy (copy-on-write), or even just "patch out" the change. The system could even be instructed to let the change propagate, if the user intends for the copy to be kept in sync. Furthermore, encoding size could be reduced by running a compression step that looks for similar or duplicated data in the stream and replaces it with an equivalent patch. No decompression algorithm would have to be written because the compressed representation remains valid Infra.

7.7 Designed User-Interfaces

Reconstructible UIs Once all data has metadata capacity, markup languages can be mixed-in to augment any data at any time with high-level types and presentation descriptions. Object-oriented agents can then be automatically attached to individual subtrees to provide various layouts, abstractions, visualizations, or interactive APIs that uphold invariants across operations. This can include enforcing a particular grammar and guiding edits toward only valid expressions. Agent implementations live on the client and take care of mapping to platform specific modalities. This paradigm is about feeding a multitude of micro applications and user interfaces to data as opposed to the other way around. These data-driven layers can themselves remain hierarchical because the metaformat/data-editor pair solve a fractal problem of structured interaction in a fractal way. They can be combined and layered recursively, maintaining the provenance of each model and view's inputs. One can imagine incrementally approximating a full-fledged traditional graphical user interface, which can then, by construction, be peeled back apart on the fly for any number of reasons.

Alternate Web Stack This is along the lines of a browser DOM that is more layered, persistent, directly authorable, suitable to any encoding, not tied to any one markup/computation language, does not represent just one fixed chunk of document at a time, and whose selective presentation, loading, collapse, and expansion of subsections does not rely on a designer's script.

7.8 Synthetic User-Interfaces

Ad-hoc UIs: Developing highly user friendly, aesthetic, and simple UIs is an important aspect of an application's usability. But for many applications and many application features, this level of polish is cost-prohibitive in terms of developer effort. Infra editors can provide the interaction and presentation of internal data, acting as a fallback user interface for portions of an application. This requires very little code and almost no design work.

Development UIs: To a developer, there is value in prototyping and testing each layer in an application's architecture incrementally. Being able to have interactive displays for structured data assists nearly every development task. Thus, there is an incentive to approach development as structural data modelling with adaptable views and APIs for each milestone, starting very general and finishing very specialized. This approach not only results in applications that are technically usable even before top level GUIs are designed for them, but also the user benefits from having a cascade of surfaces to fall back on in exceptional circumstances by no perception of additional effort on the part of the developers to provide. In fact, it may often result in less work since more of the pieces are likely to be reusable across communities. This gives applications a spreadsheet-like quality. Suitable for many gradations of end-user programming.

7.9 Backwards Compatibility and Adoption

Though the ultimate goal of Infra is to be a better *alternative* to the classical plaintext infrastructure rather than a *supplement* to it, Infra can still provide value along side exiting technology. The following paragraphs each outline progressively deeper adoption scenarios.

Infra-Agnostic Use An Infra library can offer a suite of parsers (and renderers) to convert existing flat text data to structured dialects for easier manipulation and to render them back out in the original encoding. This category includes using Infra merely as an alternative to JSON, XML, YAML, and Protocol Buffers, for encoding and transferring data.

Stand-Alone Use A fully isolated use case is to use Infra as a generalization of spreadsheets. Beyond the usual grid of flat (unstructured) values, its semantics offer trees of any depth. Infra's Patch references function much like cell formulae do.

Semi-Integrated Use Using Infra as an alternative to XML or YAML, Infra can serve as a beneficial format for directly-editable configuration files. Also, since metadata can be used to include CSS markup, Infra can bring richer web-like presentation and interaction to textual system reports, such as in console logs.

Transparent Adoption If Infra widgets replaced text field UI widgets, but continued to be used only for plaintext uses, Infra could be made to be 'invisible' while its extra capacities were available on an optional basis. The widget API only needs to offer automatic 'flattening' to plaintext strings (perhaps dropping metadata and concatenating list items using spaces) for applications that must use the entered data in a strictly character-array form. This would allow Infra to be leveraged more and more over time with a pre-existing install base.

Social-Media Adoption If a user-centric team-communication app (e.g. Google Hangouts, Slack, FlowDock, Twitter) were to adopt Infra as its medium, the situation would be similar to the 'transparent adoption' above, but would take place within the context of a community that is accustomed to authoring and exchanging constructive elements beyond text on a regular basis.

Neo HTML Infra itself is a more presentable markup language than HTML, and could be used as an always-binary-encoded alternative. Either, Infra editors can be used in the role of a web browser, or Infra extensions could be written for existing web browsers. This would unify the concepts of viewing the source and viewing the rendered page. Parsed dialects of CSS and Javascript content can exist in metadata. HTTP headers can also be switched to be Infra-based. Since the majority of an HTTP header consists of numerical values, this would save an especially appealing amount of parsing and byte overhead.

Post-Web Web Stack Since Infra editors are already similar to web browsers, and Infra's encoding comprises a markup language, binary transfer format, and dynamically allocated heap memory (using the 'Free' element type), it is a small leap to use Infra for every layer of a typical web stack, including database storage.

Patch semantics lend themselves well to mimicking an HTTP request. With the addition of a new opcode for 'navigating' the virtual cursor to a remote host, a Patch can continue executing on the remote host, acting exactly like a URL as it specifies the rest of the path with the rest of its instructions to name a resource. The Patch result is the reply from the host, performing the equivalent of a GET request. And a Patch that performs modifications would be the equivalent of a PUT, POST, or DELETE request.

To complete Infra's ability to act as a back-end database, an additional native service object can be added to act as a database driver.

Full Adoption With full adoption we envision a compiled programming language designed with infra at its core. Infra would make up the code, the data structures, and possibly the heap. New programming language semantics would be invented that leverage Infra and Patch.

Chapter 8

Related Work

The problems that Infra aims to solve require expanding the notion of human-readability beyond only character codes by generalizing it on two levels: encoding and editing. As a result, our contributions touch several domains. Existing metaformats are related work because we provide a general-purpose data encoding upon which higher-level formats can be built. Structured Editors are related work because we provide structure-informed display and editing tools. Software systems that merge programmable presentation with computational elements are related work because our goal is to scale seamlessly from raw data to high-level user interfaces within the same medium. We have organized related work by domain.

8.1 Textual Metaformats

Human-readable metaformats such as XML [31], JSON [32], Comma-Separated Values (CSV) [33], and Lisp's S-Expressions [34], are the most popular and supported formats for structured information. Yet, it would be a far cry to imagine any one of them ever becoming the universal default for all text written by all end-users. In other words, it would be overly dysfunctional if every form field, search query, and command-line expected all users to type only valid XML structures. Even in the case of wide standardization, providing structured editors to abstract away the syntax would not really be worth doing since the encoding is not machine-friendly to begin with.

8.2 Binary Metaformats

Binary metaformats such as Abstract Syntax Notation One (ASN.1) [20], Thrift [35], Google's Protocol Buffers [18], Cap'n Proto, MessagePack [19], Binary JSON [36], and Extensible Binary Markup Language (EBML) [37], swing so far in the other direction that they forgo the ability to be easily edited at all. The vast majority are explicitly designed as RPC frameworks, prioritize only byte-efficiency, and require predefined schemata or at least formal field names before any data can be encoded. Even if one of these formats had editors that would help them mimic freeform editing, few of them are designed to encode graphs and none of them support recursive metadata or fragmenting a block of memory with unallocated byte-gaps between elements. The former is critical for data-driven processing and the latter is a critical part of all programming language runtime heaps for tracking dynamically allocated memory.

8.3 Structured Editors

Structured editors have a long history of repeated attempts to assist users in the syntactic tasks of editing formal languages. Systems from the 70s and 80s, such as Emily [38], Gandalf [39], Centaur [40], as well as contemporary systems such as Subtext [41], TouchDevelop [42], and Prune [43], are billed solely as source code editors specializing in a particular language. JetBrains' Meta Programming System (MPS) [44], has generalized this by using meta grammars to allow the same system to be used to program in additional languages. However, structured editors end up limiting themselves by being so high-level. They constrain edits to prevent data from ever entering grammatically invalid states. In these systems, source code must be modified such that it is compilable before and after each atomic edit. This is cited as the main cause of the usability problems that have historically plagued structured editors [45]. Programmers' editing habits routinely find that the path of least resistance for compound edits passes through grammatically invalid intermediate states. This issue is amplified further in *Source Code in Database* (SCID) systems that customize even their storage representation to a particular language. Syntactically invalid programs do not have a way of being represented. Intentional Programming [46] utilizes such a system. In contrast, our goal with Infra is to apply the concept of structured editing to a general-purpose metaformat that languages can be built upon.

8.4 Programmable User-Environments

Programmable user-Environments such as The On-Line System (NLS) [9], the Smalltalk and Squeak User Environments [47], Berkeley's Boxer Project [26], and Wolfram's Computable Document Format (CDF), all empower end-users with authorship of and access to the descriptions responsible for the entities and abstractions present in front of them. However, each still use text characters as their only fundamental building blocks within those descriptions. This means that their 'liveness' and helpful abstractions bottom out at the source code level. This is unfortunate for users and developers alike because both programming learning curves and API documentation could benefit from those properties - recursively. A partial exception to this is Boxer, the spiritual successor to Logo

[48], which includes 'boxes' of three varieties[1] as distinct primitives in addition to text to give structure to raw data and source code statements. This still leaves out basic types like binary-encoded quantities, which are critical to the internal representations in all software systems.

[1]'Data', 'Doit' (code), 'Port' (transparent reference)

Chapter 9

Conclusions and Further Work

Infra is designed to make working with data more direct, consistent, and efficient for both humans and computers. Before Infra, developers had to either choose a human-readable format for their data and forgo processing simplicity and efficiency, or choose a binary format and forgo human readability. With Infra, developers can have both. Furthermore the organization Infra brings to data makes Patch viable as a new type of programming language targeting the domain of in-stream data metaprogramming.

In summary, Infra defines a dozen base types, a handful of Patch opcodes, a range of native-service interfaces, and an effect system to mediate side effects so that untrusted data can at least perform computation while trusted data can be useful for general-purpose programming. It provides the components needed to provide modest free-form data input widgets that can scale into a user environment and programming environment on a whim (since Patches can exist anywhere and metadata can embellish any data with appropriate editor abstractions). A capacity for structure, metadata, and computation can become an ambient part of everyday computer usage.

Infra, as a single lightweight tech stack, can be leveraged to various degrees to perform in a wide variety of critical roles across the computing landscape. To highlight a few,

Infra can provide:

- text-editor-like availability for users to quickly read, write, and manipulate Infra-encoded information
- a spreadsheet-like experience: by displaying hierarchical data in a tabular layout, Patch's data-flow-like programming model functions like cell formulae
- a rich-text document-like experience: data-driven styling and presentation (via 'CSS' metadata)
- an IDE-like experience: data-driven structure and type checking (via 'schema' metadata)
- a GUI application-like experience: high-level interfaces and abstractions of specific data models (via 'format' metadata)
- a suite of parsers (and renderers) to assist in migrating existing flat text data to structured dialects
- a Web-browser-like experience (via 'HTML' metadata): Infra naturally provides the opportunity for a binary-HTML based Web (and binary-HTTP headers), but in the meantime, a traditional parsing step can be used to interface with the existing Web.
- a command-shell-like experience: the Native-Service Objects for operating system integration allow inline browsing of the file system, invocation of executable files with input arguments, and standard I/O streams (mediated by the effect system).

We find that Infra, as a medium, enables myriad opportunities to bring richer direct-manipulation and user-interface support to much lower-level layers of computing than are normally available. Because a structured encoding opens the door to explicitly-demarcated metadata, and structured metadata opens the door to new heights of extensibility, we believe Infra is the best kind of root-striking evolution computing can make right now. In an ever more connected world with accelerating trends towards ubiquitous computing, Internet of things, semantic web[49], and an active push for broader cultural reach in Computer Science education, Infra is well poised to make the needed kind of impact. Infra raises computing's currently-low bar for what is expected from the rawest-of-the-raw tokens of encoding, in terms of their ability to standardize the building blocks of information and computation, as well as the richness of the means for interacting with them.

Bibliography

[1] J. Patterson, *Coded character sets, history and development*, *IEE Proceedings E-Computers and Digital Techniques* **128** (1981), no. 4 173.

[2] C. E. Mackenzie, *Coded-Character Sets: History and Development.* Addison-Wesley Longman Publishing Co., Inc., 1980.

[3] J. Piaget, *La construction du réel chez l'enfant.*, .

[4] I. E. Sutherland, *Sketchpad a man-machine graphical communication system*, *Transactions of the Society for Computer Simulation* **2** (1964), no. 5 R–3.

[5] O.-J. Dahl, B. Myhrhaug, and K. Nygaard, *Some features of the simula 67 language*, in *Proceedings of the second conference on Applications of simulations*, pp. 29–31, Winter Simulation Conference, 1968.

[6] T. O. Ellis, J. F. Heafner, and W. Sibley, *The grail language and operations*, tech. rep., DTIC Document, 1969.

[7] J. S. Bruner, *Toward a theory of instruction*, vol. 59. Harvard University Press, 1966.

[8] S. Papert and C. Solomon, *Twenty things to do with a computer*, .

[9] D. C. Engelbart and W. K. English, *A research center for augmenting human intellect*, in *Proceedings of the December 9-11, 1968, Fall Joint Computer Conference, Part I*, AFIPS '68 (Fall, part I), (New York, NY, USA), pp. 395–410, ACM, 1968.

[10] A. Goldberg, *SMALLTALK-80: the interactive programming environment.* Addison-Wesley Longman Publishing Co., Inc., 1984.

[11] V. Bush *et. al.*, *As we may think*, *The atlantic monthly* **176** (1945), no. 1 101–108.

[12] T. Nelson, *On the xanadu project*, *BYTE Magazine* **15** (1990), no. 9 298–299.

[13] T. Berners-Lee, *The enquire system–short description (1.1)*, tech. rep., Technical report, European Organisation for Nuclear Research. Available at http://www. w3. org/History/1980/Enquire/manual, 1980.

[14] A. C. Kay, *The computer revolution hasn't happened yet (keynote session)*, in *Proceedings of the eighth ACM international conference on Multimedia*, p. 1, ACM, 2000.

[15] E. Shein, *Should everybody learn to code?*, *Communications of the ACM* **57** (2014), no. 2 16–18.

[16] D. A. Norman, *Cognitive engineering*, *User centered system design: New perspectives on human-computer interaction* **3161** (1986).

[17] T. Boutell, *Png (portable network graphics) specification version 1.0*, .

[18] J. Dean and S. Ghemawat, *Mapreduce: a flexible data processing tool*, *Communications of the ACM* **53** (2010), no. 1 72–77.

[19] S. Furuhashi, *Messagepack: Its like json. but fast and small, 2014*, .

[20] I. ITU-T, *Iec: Abstract syntax notation one (asn. 1) specification of basic notation*, *Report no. ITU-T Rec. X* **680** (2002) 8824–1.

[21] "Dlugosz' Variable-Length Integer Encoding - Revision 2." `http://www.dlugosz.com/ZIP2/VLI.html`.

[22] D. G. Priddy and R. S. Cymbalski, *Dynamically variable machine readable binary code and method for reading and producing thereof*, July 3, 1990. US Patent 4,939,354.

[23] N. Gruschka and L. L. Iacono, *Password visualization beyond password masking.*, in *INC*, pp. 179–188, 2010.

[24] G. P. Kusnick, *Method and system for mnemonic encoding of numbers*, Apr. 6, 1999. US Patent 5,892,470.

[25] P. Juola, *Whole-word phonetic distances and the pgpfone alphabet*, in *Spoken Language, 1996. ICSLP 96. Proceedings., Fourth International Conference on*, vol. 1, pp. 98–101, IEEE, 1996.

[26] A. A. diSessa and H. Abelson, *Boxer: a reconstructible computational medium*, *Communications of the ACM* **29** (1986), no. 9 859–868.

[27] F. Loitsch, *Printing floating-point numbers quickly and accurately with integers*, *ACM Sigplan Notices* **45** (2010), no. 6 233–243.

[28] A. Steckermeier, *Lenses in functional programming*, .

[29] J. Clark, S. DeRose, *et. al.*, *Xml path language (xpath) version 1.0*, 1999.

[30] U. A. Acar, G. E. Blelloch, and R. Harper, *Adaptive functional programming*, vol. 37. ACM, 2002.

[31] T. Bray, J. Paoli, C. M. Sperberg-McQueen, E. Maler, and F. Yergeau, *Extensible markup language (xml)*, *World Wide Web Consortium Recommendation REC-xml-19980210. http://www. w3. org/TR/1998/REC-xml-19980210* **16** (1998).

[32] T. Bray, *The javascript object notation (json) data interchange format*, .

[33] Y. Shafranovich, *Common format and mime type for comma-separated values (csv) files*, .

[34] J. McCarthy, *Recursive functions of symbolic expressions and their computation by machine, part i*, *Communications of the ACM* **3** (1960), no. 4 184–195.

[35] M. Slee, A. Agarwal, and M. Kwiatkowski, *Thrift: Scalable cross-language services implementation*, *Facebook White Paper* **5** (2007), no. 8.

[36] cc, "Binary json."

[37] M. Nilsson, *Extensible binary markup language*, *Draft specification, Matroska* (2004).

[38] W. J. Hansen, *Creation of hierarchic text with a computer display*, .

[39] A. N. Habermann and D. Notkin, *Gandalf: Software development environments*, *Software Engineering, IEEE Transactions on* (1986), no. 12 1117–1127.

[40] P. Borras, D. Clément, T. Despeyroux, J. Incerpi, G. Kahn, B. Lang, and V. Pascual, *Centaur: the system*, vol. 13. ACM, 1989.

[41] J. Edwards, *Subtext: uncovering the simplicity of programming*, in *ACM SIGPLAN Notices*, vol. 40, pp. 505–518, ACM, 2005.

[42] N. Tillmann, M. Moskal, J. de Halleux, and M. Fahndrich, *Touchdevelop: programming cloud-connected mobile devices via touchscreen*, in *Proceedings of the 10th SIGPLAN symposium on New ideas, new paradigms, and reflections on programming and software*, pp. 49–60, ACM, 2011.

[43] "Prune: A Code Editor that is Not a Text Editor." `https://www.facebook.com/notes/kent-beck/prune-a-code-editor-that-is-not-a-text-editor/1012061842160013`.

[44] M. JetBrains, *Meta programming system*, 2014.

[45] A. J. Ko, H. H. Aung, and B. A. Myers, *Design requirements for more flexible structured editors from a study of programmers' text editing*, in *CHI'05 extended abstracts on human factors in computing systems*, pp. 1557–1560, ACM, 2005.

[46] C. Simonyi, *The death of computer languages, the birth of intentional programming*, in *NATO Science Committee Conference*, pp. 17–18, 1995.

[47] D. Ingalls, T. Kaehler, J. Maloney, S. Wallace, and A. Kay, *Back to the future: the story of squeak, a practical smalltalk written in itself*, in *ACM SIGPLAN Notices*, vol. 32, pp. 318–326, ACM, 1997.

[48] S. Papert, *Mindstorms: Children, computers, and powerful ideas.* Basic Books, Inc., 1980.

[49] T. Berners-Lee, J. Hendler, O. Lassila, *et. al.*, *The semantic web*, *Scientific american* **284** (2001), no. 5 28–37.