Rethinking the Human-Readability Infrastructure

Christopher Hall
UC Santa Barbara
chall01@cs.ucsb.edu

Abstract

Character encodings and text editing widgets and applications together form the current human-readability infrastructure. We outline an alternative human-readability infrastructure that is simultaneously appropriate for textual formats and binary runtime data structures. Our approach consists of a binary metaformat and corresponding language-independent structure editor. Here we describe our prototype editor, its suitability to supersede text editors, and several case studies that explore its immediate and repeat benefit to various content domains.

Keywords human-readability, structure editing, metaformat, computer literacy, computational literacy

Categories and Subject Descriptors D.2.6 [*Programming Environments*]: Interactive environments; E.2 [*Data Storage Representations*]: Object representation; I.7.2 [*Document/Text Processing*]: hypermedia, markup languages; K.3.2 [*Computer and Information Science Education*]: Literacy

1. Introduction

In general, text encodings are considered "human readable" and binary encodings are not. Examples of human-readable formats include: source code files, comma separated values (.csv), nearly every encoding used in the web stack (HTTP, .html, .css, .js, .json, .xml), TeX typesetting (.tex), and of course flat Text Files (.txt). Each of these have an encoding defined to consist solely of character codes (in some character encoding) and can thus be displayed legibly in a text editor or on a terminal without specific understanding of the language of the content. Increasingly, UTF8 is the universal standard text encoding because it combines the extensibility of Unicode with a backwards compatibility to ASCII. The ubiquity of UTF8 represents a major unification in the long and complex history of character encodings and internationalization.

Command line interfaces, text editors, and text field widgets in graphical user interfaces, each allow a user to directly author strings of characters (flat text) and therefore author content in any *human-readable format*. Flat text appears at the borders between the user and the application, and often between two distinct programs as communication over network, clipboard, or inter-process interfaces. Parsers take flat text in and convert it to a machine-friendly encoding paradigm. Renderers perform the inverse, transforming data out to flat text for human-readability. While software itself has a generally recursive nature, interaction with flat text is where structure and abstraction (the major ingredients of software) bottom out. Specifically, strings are often components of compound structures, yet strings cannot themselves have arbitrarily complex inner structures; flat text is very bare and literal in its presentation. The lack of structure and abstraction brings many disadvantages, including the need for content-altering escape sequences, the overloading of whitespace as a token delimiter, the lack of binary encoded numbers, and the line endings debacle (differences between operating system conventions), but worst of all, the inability to express arbitrarily elaborate contextual information or explicit cross-links and indirect derivations between data. Regardless of its limitations, flat text is all that users are given (as far as open-ended asynchronous interactions with raw data go) because of its monopoly on human-readability. Even when software has some choice in the matter "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" (Raymond 2003).

Based on three observations, we find that there is an opportunity to alleviate this bottleneck along with its engineering and human-computer interaction trade-offs:

1. The bit sequences of textual format encodings like UTF8 are not somehow intrinsically understandable to a human; "human readability" just colloquially implies that it is a standard encoding understood by most text editors. The sense of intrinsic readability merely comes from the ubiquity of tools that render ASCII and unicode. (You are guaranteed to find a text rendering stack in

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.

FPW'15, October 26, 2015, Pittsburgh, PA, USA
© 2015 ACM. 978-1-4503-3905-6/15/10...$15.00
http://dx.doi.org/10.1145/2846656.2846657

the core of every OS / user environment) **Technically, any format could achieve this same status if it and its editors are general-purpose enough to warrant an equal ubiquity.**

2. The machine-friendly output of parsers is itself a class of encoding with consistent characteristics. **Binary formats and programming language runtimes use quantities to structure information.** This is true whether those quantities (scalars to indicate lengths/sizes/spans) are implicit (imposed by the context) such as with the bit widths of fields in a network protocol header, or explicit (exist within the data itself) such as with the *free-list* or byte allocation records of dynamically allocated arrays in a runtime heap.
3. Computing is no longer technologically constrained by purely textual typewriters and teletypes as the common denominator for input/output convenient to human users. **Bitmapped displays are ubiquitous, and parametric graphics can abstract quantity-based syntax (outlined in observation 2) in a human-readable way.**

We see a path forward which can lead to an unprecedented unification of runtime binary data, human-readable text encodings, user interfaces, and program code, where everything on the spectrum can be made of the same primitives and live within each other. This paper will focus on the foundational ingredients to this end.

2. Approach

Our approach is to place quantity-based syntax *underneath* text encoding to give them the general structure and computational benefits of runtime data structures, while *also* placing graphical syntax on top of text editors (and text entry widgets) to preserve (or buy back) the human-readability that would otherwise be lost. We define *quantity-based syntax* as the category of encoding that declares its structural segmentation using scalar quantities as opposed to token delimiters (a common pattern in binary formats and runtime heaps but has never been given a categorical name). And we refer to *graphical syntax* as any visual non-character-based means to express the organization or structure of information in a human-readable way, such as nested box outlines (a common pattern in diagramming and graphical user interfaces). This two-part strategy enables a *higher* low-water mark for general information such that flat text never again has to be resorted to for the sake of human interactivity.

The encoding must be extremely light weight in order to be suitable for use in place of any format, including individual strings, and the editor paradigm must be suitable for use in place of any text editing, including terminal I/O and text fields in GUIs. We have designed a novel metaformat called Infra (as in *infrastructure*) to address these shortcomings and to unify the universal properties of runtime data structures and transport formats, with the properties of human-readable formats. A detailed description of Infra will be presented in a future paper.

Below, we will focus on this new human-readable medium from the perspective of a paradigm of editor that is as free-form as flat-text editors, but still edits at a syntactic level. We will refer to our reimagined text editors as **Data Editors**. They can be considered to edit data structures where text editors are only for editing text. Their graphical syntax abstracts-away the binary quantities used by the encoding to express tree structures and leaf primitives. This paper focuses on the design of our prototype Data Editor and the general concept of Language-Independent Structure Editing as a paradigm to supersede text editing.

2.1 Related Work

The most related type of free-form editor for abstract syntax is The Berkeley Boxer Project (diSessa and Abelson 1986), which is a successor in spirit to Logo (Papert 1980), as reimagined from the perspective of the graphical user interface era. It does not have a grammar in mind for *all* content because it is not primarily designed around being a structure editor; only the lines of source code that the user attempts to execute are required to be valid statements in its language. The Boxer project refers to itself as a computational medium and shares our long term goals of making personal computing more self-similar by blurring the lines between documents, data structures, source code, applications, and user interfaces, offering a transparent, exploratory, and reconstructible end-user environment. We are particularly fond of Boxer's vision, but find that its model has not quite achieved generality and that its development has been discontinued. Symptoms of this include: box names cannot themselves contain boxes, there is no structural distinction for metadata, the output box generated by an execution and the hideable "closet" box have invisible special-form structural relationships that can not be inspected or emulated by the user, the user interface prevents certain characters from being typed literally yet has no provisions in the model to escape them. The major limitation of highly unified and highly constructive environments such as Boxer, Wolfram's Computable Document Format (CDF), and the Smalltalk user environment (Goldberg 1984), is that they require wholesale adoption of an exclusive programming language and monolithic runtime stack, making them unsuitable for use in place of the individual "strings" or user interfaces within a software application.

2.2 Prototype Editor and UI Widget

We have implemented a prototype *data editor* and ported it to a custom widget in a user interface toolkit as a proof-of-concept. This has helped us to gather a list of the minimum features and design decisions an implementation needs to make in order to support content agnostic / language-independent structure editing. Each of the following sections will name a general consideration and give a brief descrip-

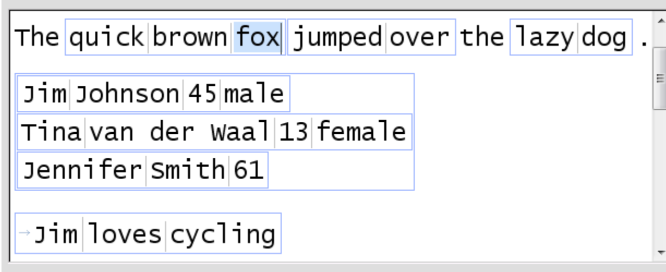


Figure 1. Our Data Editor as a UI toolkit widget, displaying some raw data and visualizing its inline parametrized structure codes as language independent graphics made primarily of recursively nested boxes outlining spans of content. Here, "The" and "the" happen to be direct children of the top level and are therefore not boxed. The cursor is currently *on* the "fox" token.

tion of our prototype's means of addressing it. By no means is there only one way of addressing them. Design polish and enhancements in the usability space are left as future work.

Creating text segments Simply typing characters will append them to the currently selected text segment, or create a UTF8 segment if the selection is not a text segment or if the tree is empty. Pressing *spacebar* will insert a new text segment to follow the current one and move the cursor to it. This segments each 'word' into its own logical token by default. *Shift-space* will instead insert a literal space character into the token without moving to a new one. When the cursor is set to character-scope editing (discussed later), the mapping for these two actions of spacebar are reversed for convenience.

Creating / converting quantity segments The creation of Integer and Floating-point segments is done by typing the number into its own segment as if it were meant to be text, then upon focus leaving the segment, it is detected as being a whole or fractional quantity and converted to the most efficient binary encoding. In the case of fractional numbers, since not every decimal value can be expressed exactly as a binary floating point number, the binary encoding is tested for round-trip coherence by converting it back to the 'shortest decimal representation' using the Grisu3 or Dragon4 algorithms (Loitsch 2010). Numbers are converted to the shortest byte-width encoding that can contain the value without loss of precision. The byte-width will be incrementally increased if the Dragon4 algorithm fails to convert back to the literal digits originally entered by the user, and will simply remain a textual segment if no level of conventional byte-width binary encoding suffices. Numeric segments are rendered in a distinct style for user awareness and default to a monospaced font for alignment. At any time the user can manually override the segment type chosen by the automatic decision through context menus. This policy provides the same numeric stability that we are used to from classical text editing, but does take advantage of the encoding efficiency when it can, as well as allow any software reading the data to unambiguously understand the intended magnitude of a value. As there are many ways to textually format a number, with preferred decimal markers, digit groupings, scientific notation, etc., the user's local region settings are used to initially interpret the intended magnitude as well as tailor the display of the numbers they view to use those preferences to create consistency. This is an example of pulling semantic negotiation to the authoring user's end of the pipeline to give them more capacity for being explicit, while also allowing the consuming user to benefit from increased tailoring.

Visualize a List's items horizontally When segments are laid out end to end, padding is left on either side to serve the role of whitespace. Line wrap or horizontal scrolling modes can be toggled by the user. Tall thin vertical bars are drawn between segments upon selection or hover in order to provide visual delineation as well as to discern actual space characters in tokens versus interstitial padding.

Honor 'new lines' symbol to display a List's items vertically Pressing the *enter* key adds the 'new lines' symbol into the metadata of the currently selected List segment if it is not already present. *Shift-enter* removes it if it is present. Double tapping *enter* toggles its presence. The layout algorithm performs horizontal distribution of a list's segments by default, and vertical when the symbol is found in metadata. This simulates newline characters in classical text editing, but exists only once for the list as a whole rather than after each line. While this assumes that a list is intended to be either entirely vertical or horizontal in its layout, mixed orientations can be achieved through nesting sub-groups together.

Visualize nesting (containment) A thin outline is drawn around the segment to visually group it. This straightforward approach is ubiquitous across mediums for communicating containment - from whiteboard illustrations, to UML, engineering diagrams, and graphical user interface toolkits in general. Background color can be used to communicate relative nesting depth, which helps to visually identify all the items that sit at the same depths. Figure 1 depicts boxes outlining user-entered content containers. They chose to communicate a grouping across "quick brown fox", "jumped over" and "lazy dog". The list of three profile records are each grouped internally as well as grouped as a mini database. Note that the explicitly typed segmentation makes the scope of Tina's last name, "van der Waal", unambiguous regardless of the fact that it contains spaces.

Show/hide subtrees At the moment, the *tab* key toggles between full and minimized display for a subtree. A similar interaction acts as The Berkeley Boxer Project's main means of navigating through content. It has no scrolling functionality and is designed around small screen resolutions.

Balance line-wrapping within and across segments Line wrapping rules treat segments as atomic if they can, but

try to balance the line-wrapping within the segment (if it is composite) with line-wrapping across the other segments at that hierarchy level.

Manage the scope / depth of the cursor Since the four directional arrow keys are tied up with performing the intuitive spatial cursor navigation expected from classical text editing, *shift-up* and *shift-down* or *page-up* and *page-down* are used to move along the additional dimension of scope.

Creating container (List) segments The *ctrl* key is the mascot for 'creating' structure in our prototype. Holding *ctrl* and pressing an arrow key will create a List segment. Left and Right will create a List to the left or right of the cursor respectively. Down will create a List as a child within the selection. Up will wrap the selection in a List, similarly to alt-down discussed below.

Hierarchical rearrangement The *alt* key is the mascot for 'moving' structure in our prototype. Holding *alt* and pressing an arrow key will move the selection through the tree. Up will pull it up to the parent List. Left and Right will swap the selection with the left and right siblings respectively. Down will wrap the selection in a List (pulling it down in the hierarchy depth) similarly to ctrl-up discussed above.

Visualize empty subtree placeholders We use a centered dot, similar to an interpunct, to give 'nothingness' a representation that can be used as a handle for that location in the structure.

Visualizing Meta Finding a robust layout for organizing metadata in relation to its data has been tricky. We find the most appropriate general means is to give it a full structural sense of separation. Our prototype uses a unique pane within the workspace to stack each layer of meta. However, during experiments, inline layouts were strongly desired to make certain kinds of data structure use cases more readable.

Navigate to Meta and back The *escape* key moves the cursor to the meta-container of the current selection. The meta segment is created if necessary. *Shift-escape* does the inverse, moving the cursor back towards the root by one level.

Create and visualize internal references Since references are about having the same content in more than one place at the same time (at least logically), we figured copy/paste might be a reasonable model to leverage. When pasting content, it starts out as an inline reference to that original subtree and is visually marked as such by an arrow pointing in from the left edge of the segment's frame. Figure 1 shows the the second occurrence of "Jim" is a reference. If either location displaying "Jim" is edited, both will change. A special use for *alt-up* is to pull a stand-alone copy of the content 'up' out of the reference frame, to no longer be a reference.

3. Case Studies

The following are some specific *before* and *after* type scenarios to illustrate the immediate potential of the paradigm we have conceptualized and implemented. An exploration of long term potential after we have defined an *Infra*-based computational model is left as future work. Any one of the sub-domains we explore below could of course be equivalently or better served by domain specific tools and designs, but our emphasis is on the ability for a strategic initial investment in the right single paradigm to lift the general foundations required by all domains. A consistency of tools is valuable for literacy, transparency, algorithm reuse, and reduction of per-domain implementations of non domain-specific interfaces (such as basic GUI properties dialogs that essentially just act as form-field editors but for domain-specific schema).

3.1 URL

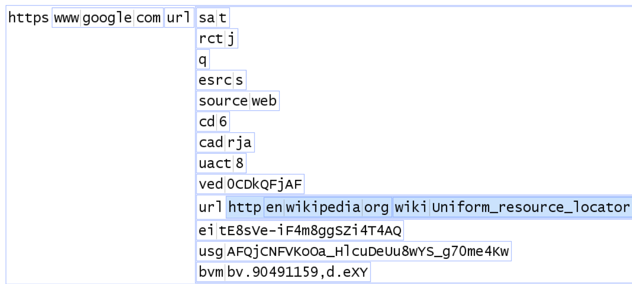
URLs are one of the most formally structured type of plain text that end-users encounter on a daily basis, potentially on-screen a vast majority of the time spent using a web browser. They are also an interesting case study because the practicality of both reading and writing them by hand has diminished as they tend to increasingly include embedded URLs (such as redirection URLs for tracking purposes), which are mangled by escape sequences. A common situation we can use to study particularly unreadable URLs is a Google search result. A query of "URLs" turns up a reference to **`http://en.wikipedia.org/wiki/Uniform_resource_locator`** but actually links to:

`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.wikipedia.org%2Fwiki%2FUniform_resource_locator&ei=tE8sVe-iF4m8ggSZi4T4AQ&usg=AFQjCNFVKoOa_HlcuDeUu8wYS_g70me4Kw&bvm=bv.90491159,d.eXY`

If a user or developer wanted or needed to fetch the actual destination URL, finding it is one thing (especially being careful about which character it ends on), but using it is another. Copying and pasting the URL gives you **`http%3A%2F%2Fen.wikipedia.org%2Fwiki%2FUniform_resource_locator`** which is not a valid URL due to the escaping. Perhaps a savvy user has memorized that %2F is a '/' and %20 is a space, etc. and can perform the necessary surgery to resuscitate the URL.

If browsers used the LfS paradigm, the infra-URLs would be something along the lines of:

The URL could of course still be displayed as a long scrolling line, but for this figure we pressed *enter* on the 'query' field to display its items in a vertical orientation. The cursor is currently highlighting the embedded URL as a whole, which has its own substructure intact. The metadata tags that mark the 'types' of elements with URL taxonomy are not being shown. The canonical use of metadata



for describing URLs would be up to the hypothetical specification for infra-URLs. An obvious candidate would be a straight conversion of the original syntax. The following is a diagram of what that looks like with the metadata tags displayed above the cell they describe, and then again, as a template exercising the full taxonomy of URLs:

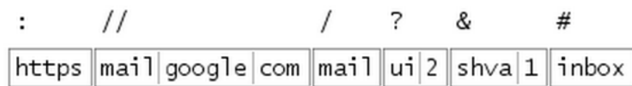


Figure 2. Original url: **`https://mail.google.com/mail?ui=2&shva=1#inbox`**

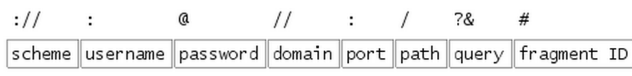


Figure 3. Full taxonomy

Not only are infra-URLs easier to read, they remain naturally in a 'parsed' state and are also easier and faster to manage programmatically as well. They would no longer have byte value constraints and can shed the need for Base64 encoding and decoding. Had we also simulated that projection in our case study, the hashcode-like portions of the URLs would have been rendered as some more compact summary visualization for large integers. Language-independent structure editors could offer all sorts of graphics enabled visualization to help summarize or collapse subtrees, especially if it can be known what portions of content are not necessarily interesting to the user at the moment.

The hypothetical study imagines that URLs would always be written, generated, viewed, edited, and processed as infra-URLs, but for this mock-up, we pasted classical plain text URLs into our prototype widget, which happens to have a small library of parsers for automatically converting non-*infra* languages to language-independent structure, including a module for URLs.

3.2 Twitter Hashtags

There are a number of characters that Twitter hashtags cannot contain. Originally, Twitter did not have the concept of hashtags and were formalized through popular convention. The originator of the folk 'syntax' (use of the '#' prefix) was a software engineer with a specific interest in social media. Though the invented convention is simple and stands out well enough with a rare but easily typed character, it is far from general. A user cannot express where the hashtag content is meant to end, and whitespace is the *de facto* token terminator. Since only single words are unambiguous in this context, the convention is to chain words together in the portmanteau fashion familiar to programmers. Hashtags have come to serve such a critical purpose for spontaneously aggregating topics, that Twitter has solidified them with semantics and parses them for automatic conversion to hyperlinks in the user interface. Besides being limited to 140 characters, Twitter is a free-form natural language domain for users to express themselves, but that freedom is bottlenecked by low-level syntactic issues.

#opportunityisnowhere
#childrenslaughter
#everyoneshavingbabies
#needsatan

Figure 4. Two-faced hashtags

Figure 4 features a collection of hashtags that have actually been used by users that may not have noticed that they were expressing themselves ambiguously and could be misinterpreted as well as grouped in with tweets of very different sentiment. In the case of #opportunityisnowhere, the intended optimistic phrase "opportunity is now here", can actually be misread as the opposite and rather pessimistic phrase "opportunity is nowhere". We leave it as an exercise for the reader to explore the multiple expansions of the other examples.

In a hypothetical world that regularly used LfS, the widgets used to compose tweets would have been future-proof and ready for a folk syntax to come along and take advantage of *Infra*'s self-similarity. A sub-structure has all the expressive potential as the root structure. The pound-sign # could still be used to demarcate a hashtag of course, but to eliminate the need to escape it when not intending to write a hashtag (such as "we're #1!"), the convention could be to put the # in the tag's metadata. Either way, these case study hashtags might have looked something like figure 5.

opportunity is now here
children's laughter
everyone 's having babies
needs a tan

Figure 5. Un-impoverished communication

Even though hashtags are of little consequence in the bigger picture, we find this case study to be a compelling representative of unfortunate moments when modern communications technology ironically reduces our ability to express ourselves the way we intend. Mainstream pop-culture writing has not been devoid of spaces between words since their

invention over a thousand years ago (Wingo 1972). Web domain names and identifier names in programming languages are similarly impoverished.

3.3 Type Sensitive Sorting

When sorting a list of files/strings that have numbers in front (e.g. a track number before the name). When the number rolls over to an additional digit, string sorting rules would place this item earlier in the list than the intended (non textual) interpretation would. Some users have padded values with leading zeros to coax the desired sorting. In current file managers special case logic has been added to handle common variants of this issue. LfS would naturally allow table-like sorting and would have avoided the need for this complexity.

3.4 Copy / Paste Form Data

In general, copy and paste does not work across form fields. Each field has to be transferred individually. LfS would allow the structure between fields to be part of the selection and therefore a part of the copy / paste.

4. Conclusion

We realize the importance of minimizing the end-user apparent differences to classical text editing. This paradigm intrinsically has extra dimensions of expression for the user to manage, but having complete literacy is only necessary to the extent that the user wishes to take advantage of them. We feel that the minimum increase in computer literacy that Language-Independent Structure requires in order to achieve basic competence, is reasonable and fairly marginal, especially when considering them as a percentage of the existing learning-curves and nuances of computer literacy that are common place now. The value proposition is fairly high: being able to express information more explicitly, earning new domains for human-readability and transparency, and for the potential unifications between command line shells, word processing, spreadsheets, web browsers, etc.

Overall, this approach scales down to represent a single classical string, with a single byte of header overhead (equivalent footprint of a null-terminated C string). At its most ambitious, it scales up to approximate full application UIs that can be, by nature, deconstructed by end users if the polished surface they have been given ever falls short of their needs. They will have “a hood to lift” and a surface to apply their computational literacy to without ever having to leave the realm of structure and abstraction. This platform has the potential to unify all classes of editing, readily absorbing the roles of calculators, spreadsheets, word processors, CLIs, IDEs, and web browsers alike, in a more fluid, granular, interoperable, and reusable way.

References

A. A. diSessa and H. Abelson. Boxer: a reconstructible computational medium. *Communications of the ACM*, 29(9):859–868, 1986.

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

F. Loitsch. Printing floating-point numbers quickly and accurately with integers. *ACM Sigplan Notices*, 45(6):233–243, 2010.

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

E. S. Raymond. *The art of Unix programming*. Addison-Wesley Professional, 2003.

E. O. Wingo. *Latin punctuation in the classical age*, volume 133. Walter de Gruyter, 1972.