Tree Notation: an antifragile document notation

Breck Yunits

***Abstract*—A new minimal notation is presented for encoding tree data structures. Tree Notation may be useful as a base notation for domain specific languages. A new conjecture on data structures is introduced and predictions are made.**

I. Introduction

Many widely used applications read and write documents in a domain specific language (DSL) based on XML,[1] JSON[2] or Racket.[3] This paper and accompanying source code (github.com/breck7/treenotation) introduce a new whitespace-based notation that can serve as an alternative base notation for DSLs. These DSLs, called ETNs (”Extends Tree Notation”), are easy to create and can be powerful. This paper describes Tree Notation (TN), three benefits and three downsides. It then introduces a new conjecture arising from TN and makes some falsifiable predictions.

II. Tree Notation

TN encodes one data structure, a **TreeNode**, with two members: an array of strings called **words** and an optional array of child TreeNodes called **children**.

TN defines two special characters, which in the canonical notation are node delimiter (”\n”) and node edge (” ”). Node edge is a space, not tab. Many ETNs also use a space as a word delimiter.

A comparison quickly illustrates nearly the entirety of the notation:

JSON:

```
{
 "title" : "About Ada",
 "stats": {
  "visits": 802
 }
}
```

Tree Notation:

```
title About Ada
stats
 visits 802
```

III. Practical Advantages

A. Easy composition

When a document is composed of blocks written in multiple languages, those blocks may require verbose encoding to accommodate the underlying base notation.

In the multi-lingual example below, a JSON-backed IPython notebook encodes Python to JSON. The resulting document is more complex:

```
{
 "source": [
  "import matplotlib.pyplot as plt\n",
  "print(\"ok\")"
  ]
}
```

With TN, the Python block is indented and requires no additional transformation:

```
source
 import matplotlib.pyplot as plt
 print("ok")
```

In this example, the ETN would define a ”source” node which interprets its child nodes as a single string.

B. Semantic diffs

JSON, XML, and Racket serializers can encode the same object to different documents by varying whitespace. Arbitrary whitespace is sometimes desirable. But a serializer’s whitespace whims often cause large diffs–and sometimes merge conflicts–for small or non-existent semantic changes.

In TN, editors have only one way to serialize an object. Diffs contain only semantic meaning.

C. Zero parse errors

Parse errors do not exist in TN. Every document is a valid TN document. Errors can only occur at the ETN level (e.g., a mistyped property name). Typos made at the location of a TN syntax character affect only the related node(s).

With other base notations, to get from a blank document to a certain valid document in keystroke increments requires stops at invalid documents. With TN all intermediate steps are valid.

A user can edit the nodes of a document at runtime with no risk of breaking the parsing of the entire document. If a node contains an ETN error, that node can handle the error itself and even autocorrect itself.

A developer working on an editor that allows a user to edit the document source does not have to worry about handling both errors at the DSL level and errors at the base notation level. TN eliminates the latter class of errors.

IV. Drawbacks

A. Lack of Tooling and Support

TN is new, and library and application support, compared to other popular base notations, rounds to zero.

Breck Yunits is a software engineer in Seattle (breck7@gmail.com).

B. *Lack of Primitive Data Types*

Some notations specify notations for common primitive types like booleans and numbers. Encoded documents can then be parsed directly to the matching in-memory structures. TN is relatively minimal and delegates the encoding and decoding of additional types to ETNs. Thus, parsing a TN document into the desired in-memory structures often requires the specification of an ETN.

In practice, however, rarely are the primitive data structures in a base level notation enough to fully parse a structure, and often an additional parse step is already used.[4]

C. *Aesthetic Differences*

Some developers dislike whitespace notations. In addition, without an ETN, a complex node in TN may be verbose, extending over multiple lines, with one node per line, whereas other base notations may benefit from a denser display of information out of the gate, with multiple nodes per line.

V. The Tree Conjecture (TTC)

Working with TN leads to some observations about the general nature of data. To the LISP programmer, all data structures are recursive lists (or perhaps "recursive cons"). With the introduction of TN, this general notion can be stated in a specific way: **All data structures are trees**.

A. *Steps to Disprove TTC*

To disprove TTC find a structure that can't serialize to TN. For example, a hash table could serialize to HashTableETN:

```
c 106
b 226
```

To the ETN Interpreter, this serialization represents a hash table, while at the same time to the TN interpreter it is a tree. Therefore, hash tables are a type of tree. TTC stands.

In a scan of over 300 programming languages and notations, no structure has been found that can't serialize to an ETN. Therefore, it is conjectured that all data structures are trees.

B. *Predictions from the Tree Conjecture*

TTC leads to many predictions. Two of them are listed below.

Prediction 1: Useful new ETNs may be found for every programming language.

Below is an example instance encoded in JTree, a JSON ETN.

```
o
 s:name Home
 o:street
  n:number 1145
```

This ETN is one of the many languages in the vast TN universe. Some simple ETNs have been around for decades.[5] ETNs are not limited to declarative data notations. There exists ETNs for C, Clojure, Ruby, et cetera, awaiting discovery. As soon as a new ETN is designed or found, any general ETN editor can then edit documents in that new ETN.

Prediction 2: The simplest 2D text encoding for neural networks will be an ETN.

By TTC, all trained neural networks are trees and therefore have ETNs. High level ETNs will be found to reduce these large trees into understandable notations.

VI. Discovery Process and Future Research

TN was discovered by an iterative process, where the goal was to find a useful language without violating certain natural Cartesian geometrical constraints. TN maps data structures to an XY plane. Perhaps an analogous process could be used for finding notations in higher dimensions.

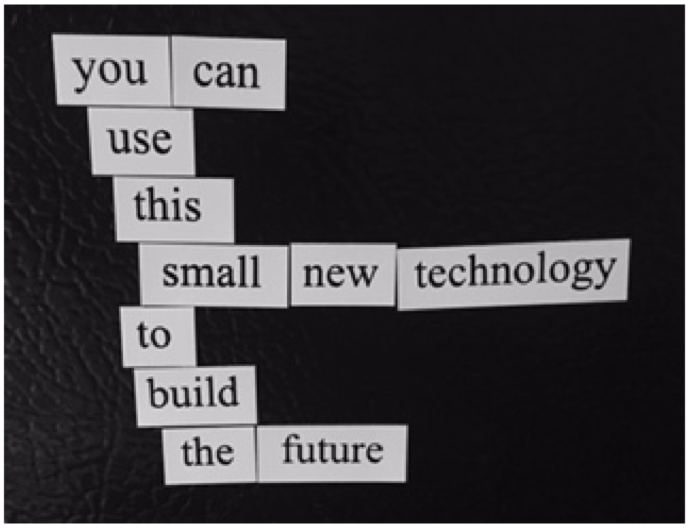

Fig. 1. Rearranging these fridge magnets is equivalent to editing a TN document. The fridge magnet set that includes parentheses is a poor seller.

VII. Conclusion

TN is a useful base level notation for DSLs. TN supports clean multi-lingual composition, aligns well with version control paradigms, and has an unbreakable base grammar that allows for robust runtime editing.

With more tooling support and the discovery of more ETNs, TN will improve developer productivity and enable faster and easier creation and maintenance of software.

TTC also makes certain predictions about future discoveries, and provides a basis on which to make further predictions.

References

[1] Bray, Tim, et al. "Extensible markup language (XML)." World Wide Web Consortium Recommendation REC-xml-19980210. http://www. w3. org/TR/1998/REC-xml-19980210 16 (1998): 16.

[2] Crockford, Douglas. "The application/json media type for javascript object notation (json)." (2006).

[3] Tobin-Hochstadt, Sam, et al. "Languages as libraries." ACM SIGPLAN Notices. Vol. 46. No. 6. ACM, 2011.

[4] Ooms, Jeroen. "The jsonlite package: A practical and consistent mapping between json data and r objects." arXiv preprint arXiv:1403.2805 (2014).

[5] Roughan, Matthew, and Jonathan Tuke. "Unravelling graph-exchange file formats." arXiv preprint arXiv:1503.02781 (2015).