2007−NL−178 (9)
2007／3／28

Trees as Paths: Lessons from File Systems and Unix in Dealing with Language Ttrees

Noah Evans, Masayuki Asahara and Yuji Matsumoto

ABSTRACT

Natural language parsers provide a variety of output formats. The rationale for these output formats vary. Some formats emphasize readability, others provide a format that is easily parsed by other programs. This paper proposes a method of language tree output that enumerates tree node structure similar to Unix paths. This approach emphasizes the software tools methodology of early versions of Unix system and more recently the plan 9 operating system. This method, by providing the structure of the tree in the description of each output node, allows easy interoperates easily with traditional Unix tools, allowing tree queries and other tree operations to be performed by programs like `grep` and `sort` interoperate with a wide variety of scripting languages.

Keywords: NLP, tree querying, software tools

Motivation

NLP tools typically produce two forms of output, human readable and machine readable.

Human readable output typically uses a display engine or text to represent the tree in a form that visually represents the tree structure:

```
   魚を-D
食べると---D
        頭が-D
   よくなる。
```

This representation allows visual, intuitive analysis of tree structures by the user; however, it is difficult to perform any kind of automated analysis on this type of data. To compensate for this problem NLP tools also produce machine readable output.

Machine readable output is typically expressed in a standardized format, commonly s-expressions like the Penn Treebank style

```
(S (SBAR (WHADVP When)
         (S (NP your back)
            (VP is
                (PP against
                    (NP the whiteboard)))))
   , (S (NP I)
      (VP 'll
          (VP be
              (PRT back) (S (VP to
                                (VP back
                                    (NP you) (PRT up)))))))
```

or the XML output of CaboCha(edited for clarity):

```
<sentence>
 <chunk id="0" link="1" rel="D" >
  <tok id="0" read="サカナ" >魚</tok>
  <tok id="1" read="ヲ" >を</tok>
 </chunk>
</sentence>
```

These output formats can be parsed by any tool supporting the relevant tree format. Tools parse the structured data and convert the input tree into a data structure. Programs can then perform various tasks like semantic role labeling or named entity recognition on the data structure. Tree querying programs, like `tgrep` and `lpath`, or programs which analyze tree data like Bact use these tree structures as the basis of their operation (Rohde 2004)(Bird 2005)(Kudo 2004).

While this reparsing is simple given the proper parser it is computationally expensive. The structure of the data must be reinferred after each tree is generated by the parsing tool. In addition, to support any one of these formats a tool requires specialized parsers for each format or a conversion utility to convert trees to the proper format type.

To avoid the computational cost of reparsing and the difficulty of supporting a variety of input and output formats many NLP tools also provide libraries to access the NLP tool's capabilities directly using a scripting language like Perl, Python or Ruby. However only languages supported by the tool itself can be used. Implementing this support requires a great deal of effort, using the Foreign Function Interface or some other mechanism to connect the tool to the supported language.

These two traditional methods of interaction between NLP tools discourages interoperability and the reuse, leading to a variety of incompatible and complex data formats. This incompatibility and complexity compounds the difficulty of implementing tools like morphological analyzers or language parsers. This has lead to a situation where tools are constantly reimplemented and each tool implementer must devote a great deal of effort to supporting different formats. This implementation time could be better spent developing functionality directly related to solving NLP problems.

How would it be possible to allow NLP tools to support a variety of different languages and formats without placing an undue burden on the implementer? An ideal system would be language and format agnostic, allowing the implementer to concentrate on implementing NLP tool functionality without worrying about supporting various formats.

The Software Tools Philosophy

The "software tools" philosophy developed by Bell Labs for the Unix operating system is one way of allowing this kind of interoperability between programs.

A software tool is a program that is meant to take input from and give output to other programs with no explicit connections between the individual programs. Connections are created automatically by a command interpreter, in the case of Unix, the shell(Pike 1984). These connections, called pipes, allow programs to communicate without any explicit communication protocols. Instead programs read and write binary streams using flat(non recursive) text.

Using streams provides a variety of implementation and interface advantages. Because the connections between programs are provided transparently by the operating system, changes to one program do not alter any of the other programs used. Even if one tool understands a different text format, tools with differing textual interfaces can be combined by altering their inputs using an intermediate pipe which changes the input and output of programs to a common format. These intermediates are typically created using a pattern processing language like awk or `sed`

Tools can use this freedom of interconnection to specialize. Since any tool can be used with any other tool users only need one tool for one task.

The troff typesetting system is perhaps the best example of the tools philosophy in action. Various tool pipelines take a variety of different domain specific graphical, tabular and equation layout languages and convert the output of these languages into the more general troff language. For an in depth description of troff see (Ossanna 1977).

Most Unix users are at least somewhat familiar with this style of interaction i.e. using the cat program with the more pager. More detailed explanations of the tools philosophy can be found in(Pike 83)(Plauger and Kernighan 76).

A Software Tools approach to structuring NLP data.

So how would you structure an NLP tool using this philosophy? Unix tools seems inappropriate for processing the output of NLP tools. NLP tools deal with structured data; Unix tools deal with flat text. However Unix does deal with structured data in one situation, Unix paths.

Unix paths are a representation of Unix's hierarchical file system, which itself is a rooted tree. A Unix path is a depth first enumeration of the path from the root to a node of a tree structure. Paths represent the root of the tree with "/" and each child node is the child's name separated by a "/"'s for each level in the hierarchy.

This following examples show a Unix path structure:

```
/usr/
/usr/npe/
/usr/npe/.bashrc
```

Although Unix path representation can describe structured data, the ordering of Unix paths is still inappropriate for NLP data. Unix paths are ordered alphabetically; NLP parse trees are ordered by sentence structure. Since Unix paths cannot represent arbitrarily ordered trees, to effectively represent NLP tree structures the order of the tree must be represented explicitly. This ordering can be expressed by representing the position of each node with a numerical prefix. Note that this path is intentionally unordered.

```
echo '
/2.usr/1.npe/
/2.user/2.glenda/
/2.usr/
/2.usr/1.npe/1.file.txt
/1.bin/1.sed
```

One interesting aspect of this prefixed format is that the tree can be sorted with the unix command `sort` to obtain the proper order of the tree structure.

```
% sort tree
/1.bin/1.sed
/2.user/2.glenda/
/2.usr/
/2.usr/1.npe/
/2.usr/1.npe/1.file.txt
```

This explicit expression of the order of a tree means that any NLP tree can be represented in the same style as Unix paths.

The parse tree fragment

```
(S (SBAR (WHADVP When)
         (S (NP your back)
            (VP is
                (PP against
                    (NP the whiteboard)))))
    , (S (NP I)
      (VP 'll
          (VP be
              (PRT back) (S (VP to
                    (VP back
                        (NP you) (PRT up))))))))
```

Can be represented using the following path format.

```
/1.S/1.SBAR/1.WHADVP/1.When
/1.S/1.SBAR/2.S/1.NP/1.your
/1.S/1.SBAR/2.S/1.NP/2.back
/1.S/1.SBAR/2.S/2.VP/1.is
/1.S/1.SBAR/2.S/2.VP/2.PP/1.against
/1.S/1.SBAR/2.S/2.VP/2.PP/2.NP/1.the
/1.S/1.SBAR/2.S/2.VP/2.PP/2.NP/2.whiteboard
/1.S/2.,
/1.S/3.S/1.NP/1.I
/1.S/3.S/2.VP/1.'ll
/1.S/3.S/2.VP/2.VP/1.be
/1.S/3.S/2.VP/2.VP/2.PRT/1.back
/1.S/3.S/2.VP/2.VP/3.S/1.VP/1.to
/1.S/3.S/2.VP/2.VP/3.S/1.VP/2.VP/1.back
/1.S/3.S/2.VP/2.VP/3.S/1.VP/2.VP/2.NP/1.you
/1.S/3.S/2.VP/2.VP/3.S/1.VP/2.VP/3.PRT/1.up
```

Advantages

Formatting NLP trees using a path based structure provides a variety of advantages.

First, unlike traditional tree representations, representing trees as paths explicitly expresses the location of each node in relation to the root for every path node. The explicit nature of the representation allows the extraction of tree patterns using regular languages —the extracting program does not require recursive state to express tree patterns.

This ability to use regular languages to express tree structures allows the user to search and process trees using traditional Unix tools and scripting languages. Any language or tool supporting line based input and regular expressions can be used as a tree querying and processing language. This gives the user a set of de facto tools(i.e. `sort` , `grep` , `sed` , `awk`) to deal with trees and allows the user to use a variety of languages to process trees without any explicit support from the tool providing the trees.

For instance simple vertical tree queries are trivial using a path based notation. Let tpaths be a list of trees in path representation then the command

```
% grep '.NP/' tpaths
```

will return all paths containing noun phrases.

Horizontal tree queries are more difficult but possible using a multi-pass approach, one pass for every adjacent node. Each pass narrows a list of candidates to ensure that a particular structure contains the specified elements.

To get all noun phrases containing "New England":

First find all of the noun phrases containing "New" and generate regular expressions uniquely identifying that noun phrase with the second term.

```
% grep '.NP/.*.New$' tpaths |
sed 's#^/([0-9]+.S).*([0-9]+.NP/)[^/]*#^/1.*2England#' >pats
```

Then use the pattern file as a basis to search for England. Generate a regular expression for all noun phrases and sentences matched(which will contain New and England by default)

```
% grep -f pats tpaths |
sed  '
s#^/([0-9]+\.S).*([0-9]+\.NP/)[^/]*#^/\1.*\2England#
  ' >pats2
```

Finally return all of the trees matched:

```
% grep -f pats2 tpaths
```

the horizontal query process is much easier using an environment providing regular expressions that aren't limited by line boundaries like the structural regular expressions of the plan 9 system detailed in (Pike 1987). The same query using the sam text editor is

```
% sam -d tpaths <<!
/\.NP\/[0-9]+\.New\n.*\.NP\/[0-9]+\.England\n/p
!
```

Leaf names can be extracted with Unix tools as well:

```
% basename  '/1877.S/3.PP/3.NP/2.own' |
sed 's/^[0-9]+.//'
own
```

In addition to being able to use Unix tools to query trees It is just as easy to apply scripting languages to the parsed data.

We will use awk in the following examples because awk notation allows examples to be expressed without any explicit record handling logic.

The first example is the same as our earlier examples it does a horizontal tree query for "New England". The script performs the following actions, treating the record as a doubly linked list

1. Iterate backwards over the list starting from the last field(tree node) first.
2. Note when "New" and "England" are adjacent and in the end of their respective fields.
3. Keep iterating backwards until the noun phrase ends.
4. Traverse the list forwards printing node elements until the end of the noun phrase.

The script makes the following assumptions about the structure of the path stream in addition to the path structure proposed earlier. Complete trees are records separated by a blank line. Individual paths are fields separated by newlines. Note that this example omits searching for multiple matches in the same tree for brevity.

```
awk 'BEGIN { FS="\n"; RS=""; OFS="\n" }
hasne(0,0) {
        for(i=NF; i; i--) {
                if(i-1 &&  hasne(i-1, i)) {
                        foundpat=1; i--
                        len=split($i,a,"/")
                        np=a[len-1]
                        break
                  }
        }
        if(foundpat) {
                while(i && split($(--i),a,"/")
                                && a[len-1] ~ np);
                split($i,a,"/")
                for(j=i; j &&  a[len-1] ~ np; ) {
                        print $j
                        split($(++j),a,"/")
                }
                foundpat=0
        }
}
function hasne(fld1,fld2) {
        return $fld1 ~ /New/ && $fld2 ~ /England/
}
```

The second examples shows the ease in which the path format can be converted to other formats. Because the relationship between root to node is inherent in the path format, a simple approach comparing the previous to the current path can generate the appropriate xml or s-expression.

The following awk script generates a Penn TreeBank style tree from one path expression.

```
% awk ' BEGIN { FS="/"; nprev=0 }
{
        if(NR != 1) {
            for(j=2; $j ~ prev[j] && NR != 1; j++);
            for(i=j; i<=nprev; i++) printf ")"
           print ""
        } else
                j=2
        for(i=0;i<nprev;i++) printf "\t"
        for(i=j; i <= NF-1; i++) {
                sub("[0-9]+\.","",$i)
                printf "("$i"\t"
        }
        sub("[0-9]+\.","",$NF)
        printf $NF
        nprev=NF-1
        split($0,prev,"[ \t]")
}
END {   for(i=1; i<nprev; i++) printf ")"
                print "" } ' tpath
```

This script has important implications. Any program that adheres to this path based system can generate a recursive tree format by only referring to the previous tree path. Thus path based tools can support a variety of tree formats by providing a common set of shell scripts that use path based input to produce tree based output. This obviates the need to support output formats in NLP tools themselves.

Disadvantages

While the software tools method of programming interaction is powerful and general the tools approach has significant problems.

The software tools philosophy, specifically as implemented in Unix, is frequently described as unintuitive and representative of bad human engineering principles(Norman 1981).The creators of Unix themselves believe that the tools philosophy is difficult for users to understand. Due to this lack of understanding the tools philosophy has gradually become less common even as the number of systems using Unix based operating systems has grown(Pike 2001).

These difficulties are apparent when using tree based path representations. Paths are neither as concise as other tree formats nor as regular to query using Unix tools compared query languages. Tree querying and processing using Tree Paths also requires a facility with the Unix shell beyond what a tool implementer can expect of casual users of NLP programs.

Conclusion

Because of these disadvantages path based representation of tree structures should be considered an intermediate format. As the earlier shell and scripting examples show paths can be used as a means of rapidly prototyping and laying the foundation of general interfaces to interact with NLP data. By encoding the state of the tree into a line based structure paths allow the systems administrator or the tool implementer to easily tailor and manipulate tree structures without depending on the features of any one language. The tool implementer can avoid specific implementations of format types, creating format types as needed using the extra information encoded in the path structure. Simple languages such as `tgrep` or `lpath` could be implemented on top of tree path expressions either using the output converted to another format or by implementing native support for a tree based approach.

Future Work

The large amount of redundant data in paths makes paths an ideal candidate for compression. Compressing paths while still allowing them to be searched would eliminate one of the central problems of path representation. (Woods 1983) describes such a method using unix paths. Future work could study the feasibility of this approach to tree paths.

In addition to compression the self ordering and flat representation of paths makes them an ideal candidate for parallelization. Future research could examine methods of parallel tree querying and tree processing. The lack of dependence between tree nodes would also seems to be an ideal candidate for processing using Google's Map Reduce parallel algorithm.

References

Rohde D.L.T. (2004). "TGrep2 User Manual." **http://tedlab.mit.edu/dr/Tgrep2**

Bird S., Chen Y., Davidson S., Lee H. and Zheng Y.(2005). "Extending XPath to Support Linguistic Queries." In *Workshop on Programming Language Technologies for XML (PLAN-X).*

T. Kudo and Matsumoto Y. (2004). "A boosting algorithm for classification of semi-structured text." In *Proceedings of the 2004 Conference on Empirical Methods in Natural Language Processing.*

Woods, J.A. (1983). "Finding Files Fast." *;login, 8:1*

Ossanna J.F. (1977). "Troff User's Manual". *Bell Laboratories Computing Science Technical Report*

Pike, R. and Kernighan B. (1984). "Program design in the UNIX environment." *AT&T Bell Laboratories Technical Journal*

Pike R. (1987). "Structural Regular Expressions." In *EUUG Spring 1987 Conference Proceedings.*

Pike R.(2001) "The Good, the Bad, and the Ugly: The Unix Legacy." In *Commemoration of the Billionth Unix Second Since the Epoch*

Norman D.A. (1981). "The trouble with UNIX." *Datamation*

Kernighan, BW and Plauger PJ(1976). *Software tools. ACM Press New York, NY, USA*