A Development Environment for Large-scale Multi-lingual Parsing Systems

Hisami Suzuki
Microsoft Research
One Microsoft Way
Redmond WA 98052 USA
hisamis@microsoft.com

Abstract

We describe the development environment available to linguistic developers in our lab in writing large-scale grammars for multiple languages. The environment consists of the tools that assist writing linguistic rules and running regression testing against large corpora, both of which are indispensable for realistic development of large-scale parsing systems. We also emphasize the importance of parser efficiency as an integral part of efficient parser development. The tools and methods described in this paper are actively used in the daily development of broad-coverage natural language understanding systems in seven languages (Chinese, English, French, German, Japanese, Korean and Spanish).

1 Introduction

The goal of the grammar development at Microsoft Research is to build robust, broad-coverage analysis and generation systems for multiple languages. The runtime system is referred to as NLPWin, which provides the grammar development environment described in this paper. The graphical user interface of NLPWin is shown in Figure A in the Appendix. The system is modular in that the linguistic code is separate from non-linguistic code. All languages share the same parsing engine, which is a bottom-up chart parser and is fully Unicode-enabled. Linguistic code itself is also modular, in that it can be specific to a particular language (e.g., syntax rules) or can be largely shared across languages (e.g., semantic mapping rules). Linguistic rules are written in a proprietary language called G; a sample syntax rule written in G is given in Figure B in the Appendix. G-rules are translated into C, yet they are more convenient for a linguist to use than C, as it gives special notational support to attribute-value data structures used within the system. The rule and data structure formalisms are shared by all languages; for details, see Jensen *et al*. (1993) and Heidorn (2000).

In this paper, we describe the tools and methods for the cross-linguistic development of analysis components of our system, which consists of three major modules: (i) the tokenization component, which performs word segmentation (in the case of Chinese and Japanese) and morphological analysis; (ii) the parsing component, which performs phrase-structure analysis and creates parse tree(s)[1]; (iii) the Logical Form (LF) component, which computes the basic predicate-argument structure from parse tree(s)[2]. In this paper, we focus on the tools and the methods for the development of parsing and LF components, which are essentially the same[3].

For an efficient development of a computational grammar of these modules, we find it necessary to have a development environment that can provide immediate feedback to the grammar-writer of the changes he or she has made. We have three types of tools in our system to meet these requirements:

- Tools for linguistic rule writing: these include the tools that let linguists navigate through the final and intermediate parse trees, and trace rule application (Section 2).
- Tools for grammar testing: these tools allow linguists to compare results of two versions of

[1] This component is further divided into the *Sketch* component, which produces trees with default attachment of constituents, and the *Portrait* component, which finds the best attachment sites (Heidorn, 2000).

[2] LF is computed from a surface syntax tree via a level of representation called *Language-Neutral Syntax* (LNS), which serves as an interface to various semantic representations including predicate-argument structure. For a more detailed description of LNS, see Campbell and Suzuki (2002).

[3] Similar tools and methods are also available for the development of sentence realization component.

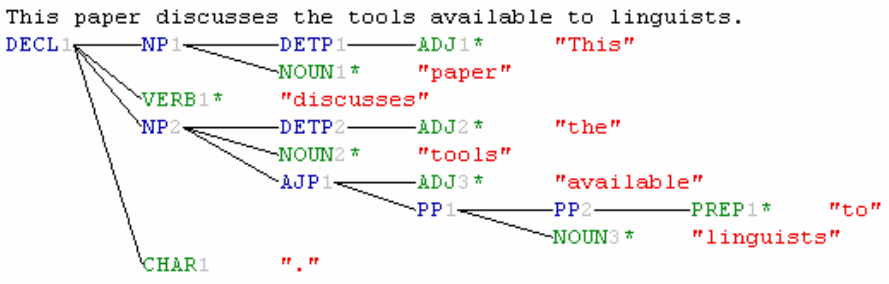


Figure 1: A parse tree

```
>display record VERB1*
(Segtype     VERB
 Nodetype    VERB
 Nodename    VERB1
 Ft-Lt       3-3
 String      "discusses"
 CopyOf      REC152
 Lex         "discusses"
 Lemma       "discuss"
 Bits        Pers3 Sing Pres Tl T6
             Hsubj Loc_sr Recip
 Prob        1.00000
 Parent      DECL1 "This paper discusses the tools
             available to linguists."
 Infl        Verb-tax )
```

Figure 2: Lexical record for VERB1 "discusses"

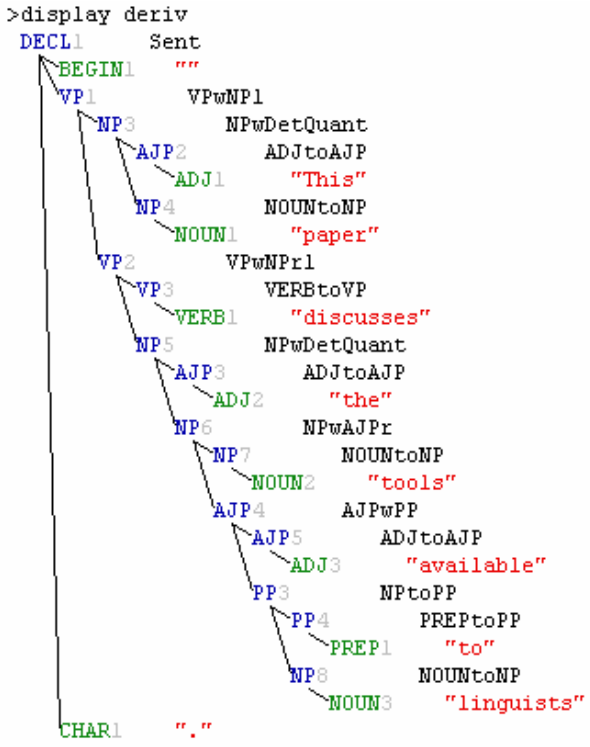


Figure 3: A derivational tree

grammar, and update the database of desired output structures (called *regression suites*, Section 3).

- A very fast processing environment (Section 4).

These tools are described in the following three sections of this paper. Section 5 gives a summary and suggests directions for future research.

2 Tools for linguistic rule writing

In this section we present the tools available for the development of the parsing component. The output structure of parsing is graphically represented as a phrase-structure tree, as in Figure 1 above. Various functionalities are available to navigate through this tree as well as intermediate (or failed) representations, by simple operations such as double-clicking the node in the user interface, or by typing in commands in the Command window, which can be invoked by the Command menu in the user interface (see Figure A). Below is a selected set of examples of tree navigation functionalities which are essential to the fast development of linguistic rules:

Accessible records

At any point, a linguist can access the records underlying the parse tree by double-clicking the node. The *record* for a node is comprised of lexical and morphological information, syntactic and functional features and attributes, as well as pointers to the sub-constituents and parent of the node. For example, double-clicking on the VERB1 node in Figure 1 will display the record structure in Figure 2.

Derivational tree

We can also display the history of rule application in graphical form, as in Figure 3. Any node in the history tree (called the *derivational tree*) can also be double-clicked in order to access the record underlying it.

Apply Rule and Rule Explain

Rule Explain shows the application of the rule underlying the formation of a node in the tree. The rule application is displayed using a color-coded display to highlight successful conditions (green), failed conditions (red) and the actions performed on the resulting record (purple) on the rules such as the one displayed in Figure B. The display is available for both successful and failed rule applications: we can access the Rule Explain display by double-clicking the resulting node, or we can manually apply any rule to any constituent to bring up Rule Explain.

Compare

Parsed trees can be quite large and it may be difficult to determine exactly where two trees differ from each other. In such a case, trees and nodes can be easily compared to detect subtle differences in composition or rule history by the Compare function.

Display trees

This command is particularly useful in checking the edges of possible, intermediate constituents. It displays all the partial trees with a certain label that includes a particular node or spans over specified nodes. The following are some examples of possible variations in the query:

(a) `display trees VP 1 5`
(b) `display trees NP NOUN4`
(c) `display trees AJP`

(a) displays all VPs that span from position 1 to 5; (b) displays all NPs that include the node NOUN4; and (c) displays all possible subtrees whose nodetype (label) is AJP.

Tree filters

This functionality does not directly assist the grammarian in writing rules, yet is extremely useful in collecting and examining particular linguistic constructions of interest that are output by the parser. The linguistic developer can write custom-made tree filters in G, which traverses the parse trees or LF structures and exports only the information needed for a particular purpose, or only those sentences with particular linguistic configurations. Tree filters are also convenient in creating a linguistic annotation for external applications.

The tools described in this section enable linguists to inspect the effect of grammar changes in detail, with the information of how exactly a particular rule applied or failed. These tools are used in the context of daily grammar development, which we describe in the next section.

3 Process of grammar development

3.1 Incremental grammar testing and creation of regression suites

The standard practice of parser development within our group is schematically shown in Figure 4. The grammarian for each language processes a text file with input sentences and adds only the sentences with desired parses to what we call a *master file*, which contains the sentences and their target structure. A collection of master files is called a *regression suite*. A regression suite thus contains the target structures given a particular version of the grammar. When new grammar changes are made in order to accommodate a new sentence or construction, the linguist runs the new grammar against the regression suite (called *regression testing*) to examine the consequences of the changes to the grammar. When differences are found, they are kept in **.dff* files and are displayed in two colors, highlighting the differences. Figure C in the Appendix is an example display of a difference (unfortunately in black and white): the highlights in green (here the first three lines) correspond to the analysis in the master file, while those in red (the next three lines) indicate the new

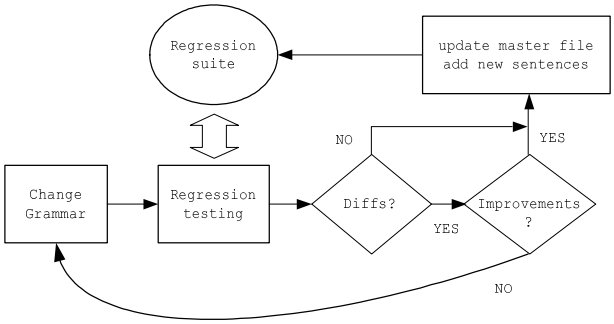


Figure 4. Flow diagram of daily grammar development

analysis introduced by the new grammar rules. The lines that did not change are grayed out. If the change is an improvement, the developer can choose to update the master file by double-clicking on the sentence number (in purple), adding the sentence or construction that is newly accommodated by the parser to the regression suite. If the change is evaluated as negative, the linguistic developer reworks the rules that caused the regression.

3.2 Testing against relative standards

As is described above, we run regression tests against the machine-created master files rather than against an independent set of hand-annotated target corpora. The test is therefore *incremental* and *relative*, in that new sentences and their target structures are constantly added as the grammar develops, and what it measures is not the coverage against an absolute standard, but the coverage improvements relative to the output of an old version of the grammar.

The incremental and relative testing method has proven to facilitate the development of a broad-coverage parsing system in some important respects. First, it ensures that the desired structures in the master files are always *current*. Because the master files are constantly incremented and updated using the most recent version of the grammar, they will never become obsolete should the target structures change. The ease of maintenance of the regression suite is one of the key features contributing to the usefulness of the regression suite in our daily development work.

Secondly, because the master files are created automatically rather than by hand, the resultant annotation is guaranteed to be *consistent*. Creating a test corpus for parser evaluation by hand is known to be an extremely laborious, inconsistency-prone task, especially when the tagging is performed on real-life data [4]. In addition, a broad-coverage grammar must also work with input strings that are not necessarily well-formed, including sentence fragments, ungrammatical sentences and extreme colloquialisms. Hand-annotating these structures may either be impossible or extremely error-prone. In contrast, by annotating them automatically using the output of the parser, these structures can be added to our regression suite easily and consistently. Effects of later grammar changes can easily be detected by running the regression testing as part of the regular development process.

Finally, incremental and relative testing makes the parser development *data-driven* rather than being dictated by a theory. This is an important feature for a large-scale system. Though it is eventually up to the grammarian to accept or reject a particular analysis, the system always provides a candidate analysis for any input string, which facilitates the rapid creation of the master files. It also allows linguistic developers to experiment on the grammar code in the following sense: assume that there is a sentence or a construction that allows multiple linguistically valid analyses, and that there is no obvious reason to prefer one to the other, a situation that arises often in the development of a broad-coverage grammar. In this case, the grammarian can temporarily choose one of the structures as a target, and add it to the regression suite. If the target structure the grammarian has selected is inconsistent with the rest of the grammar, it will constantly come back as a regression (difference) when further changes to the other parts of the grammar are made, because the assumption implicit in the tentative target structure is not consistent with the rest of the grammar. Once the change is made to the target structure that is consistent with the remainder of the grammar, it typically stops appearing as a difference in regression tests. The data-driven nature of development therefore helps the grammarians to proceed with grammar development even when there is indeterminacy in the target structure. Regular regression testing over large corpora

[4] One piece of evidence for this statement is that the bracketing guidelines for Penn Treebank project (Bies *et al*. 1995) consist of over 300 pages of documentation for annotating relatively homogeneous text.

ensures that any outlandish analyses have only a short life span in our regression suites.

Possible disadvantages of testing against a relative standard include: (i) it is difficult get a feel for how mature the grammar is in general; (ii) it makes the comparison across different systems difficult. The first problem is addressed partially by running evaluation testing against blind benchmark corpora, which consists of sentences never used in the grammar development. The parser coverage is automatically measured in terms of the number of sentences that received at least one spanning parse, versus those that failed to receive any spanning analysis.

Testing and comparing parser performance across different systems is an extremely difficult task, given different aims and grammatical foundations. One possibility, which is currently pursued in our group, is to develop a metric that enables comparison with manually created golden standards, as they have become more widely available for various languages, such as the Penn Treebank for Chinese and English, NEGRA corpus for German, and Kyoto Corpus for Japanese.

Ultimately, the parser output must be compared and evaluated at the level of an application that uses the result of linguistic analysis. Campbell *et al*. (2002) is an attempt to use machine translation as a test bed for a multi-lingual parsing system.

4 Parser efficiency as part of efficient parser development

For a development of a truly broad-coverage parser, it is critical that grammar changes are constantly verified against a very large set of sentences, and that the time for feedback is minimal. The efficiency of the parsing engine is thus inseparable from efficient grammar development.

Our parsing engine is already quite fast: for example, our English system currently parses Section 21 of Penn Wall Street Journal (WSJ) Treebank (1,671 sentences) in 110 seconds (or about 15 sentences/sec) on a standard machine (993MHz Pentium III with 512MB RAM); this performance is comparable across languages.

Speed improvements are usually performed by non-linguistic developers following standard optimization techniques. We use internal profiling tools to identify performance bottlenecks, and make a special effort to ensure that the G-to-C translator generates efficient C-code. Because the linguistic code is independent of the non-linguistic code of the system, the parsers for any language can immediately benefit from performance improvements made at the system level.

For regression testing, we also have a means to distribute the processing onto multiple CPUs: the processing cluster currently consists of 19 machines with 2 CPUs each (500MHz, 128~512MB RAM), which parses the entire WSJ section of Penn Treebank (49,208 sentences) in 3 minutes and 10 seconds (or 259 sentences/sec), and a one million-sentence Nikkei newspaper corpus of Japanese in about 30 minutes (550+ sentences/sec). In daily grammar development, each grammarian typically works with a regression suite consisting of 10,000 to 30,000 sentences at various levels of analyses; the time required for processing a regression suite is 2 to 6 minutes. In addition, automatic regression testing is run nightly against relevant regression suites using the most recent builds of the system, ensuring that no negative impact is made by any changes introduced during the day[5].

In this section, we have discussed the issue of parser efficiency from the perspective of grammar development. Our processing environment enables immediate feedback to grammar changes over very large corpora, and is thus an essential part of the development environment for a broad-coverage parser.

5 Conclusion

In this paper we have described the tools and methods for a development of large-scale parsing systems. We have argued that constant testing of the grammar against a large regression suite is the central part of the daily grammar development, and that the tools and methods described in this paper are indispensable for maximizing the productivity of linguistic developers. Though the tools are specific to NLPWin, we believe that the general practice of grammar development presented in this paper is of interest to anyone engaged in grammar development under any grammar formalism.

As a cross-linguistic development environment for analysis and generation components, some of

[5] We use standard version control software to manage both linguistic and non-linguistic source code.

the properties of NLPWin discussed in this paper are shared with such projects as ParGram (Butt *et al*., 1999). One of the main differences between ParGram and NLPWin is that the latter has so far been developed and used at one site. As there are more parsers available in many languages, it would be interesting to see if externally developed components can be plugged into NLPWin at the level of LNS. Such research is left for the future as a possible extension to the modularity and cross-linguistic aspect of NLPWin.

Acknowledgements

This paper presents the work that has been designed and implemented by many people in the NLP Group at Microsoft Research, particularly George Heidorn and Karen Jensen.

References

Bies, Ann, Mark Ferguson, Karen Katz, and Robert MacIntyre, 1995. B*racketing Guidelines for Treebank II Style*. Penn Treebank Project, University of Pennsylvania.

Butt, Miriam, Tracy Holloway King, Maria-Eugenia Niño and Frédérique Segond. 1999. *A Grammar Writer's Cookbook*. CSLI Publications, Stanford.

Campbell, Richard, and Hisami Suzuki. 2002. Language-Neutral Representation of Syntactic Structure. In *Proceedings of SCANALU 2002*.

Campbell, Richard, Carmen Lozano, Jessie Pinkham and Martine Smets. 2002. Machine Translation as a Test Bed for Multilingual Analysis. In *Proceedings of the Workshop on Grammar Engineering and Evaluation*, COLING 2002, Taipei.

Heidorn, George. 2000. Intelligent Writing Assistance. In Robert Dale, Hermann Moisl and Harold Somers (eds.), *A Handbook of Natural Language Processing: Techniques and Applications for the Processing of Language as Text*. Marcel Dekker, New York. Chapter 8.

Jensen, Karen, George E. Heidorn and Stephen D. Richardson (eds.). 1993. *Natural Language Processing: The PLNLP Approach*. Kluwer Academic Publishers, Dordorecht.

Appendix

Figure A: Graphical user interface of NLPWin

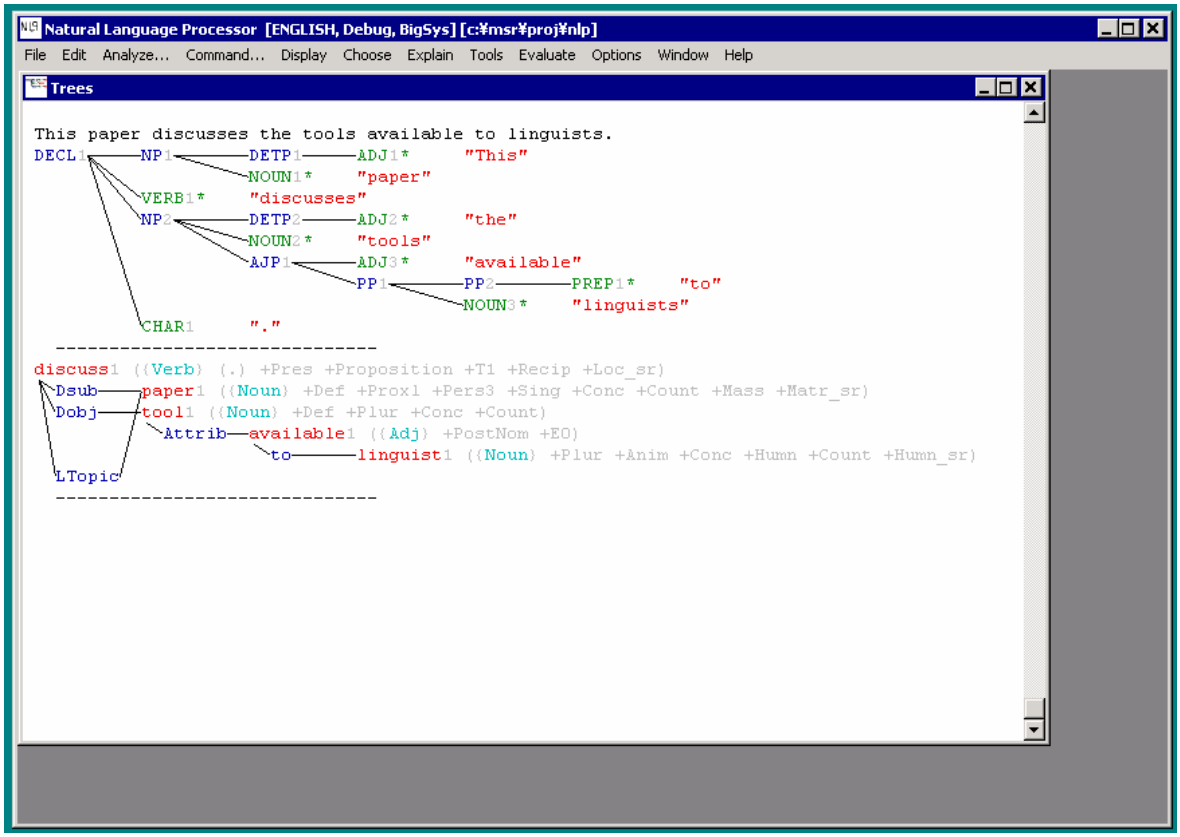
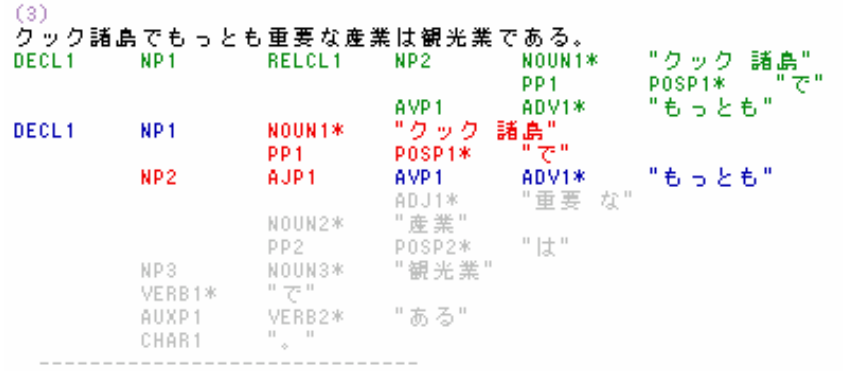


Figure B: Example of a phrase-structure rule[6]

```
AVPwAVPl:
    AVP#1 (^Comma & ^Conjt & ^NoAdv & ^Top & Nodetype(Head)^="IJ" &
          (Nodetype ^in? set{AVPNP AVPVP} | Compr | Intens | Tme | Ntimes) &
          (Advrz -> (^Adv(Lex) & Intens)) &
          (ModalAdvs -> Intens) &
          (Nconj -> (^Conj(Lex) & Lemma^in? set{同})) )
    AVP#2 (^AVPcoord & ^Conjt & ^Kakari & ^Nconj & ^NoAdv & ^Wh &
           Nodetype^in? set{AVPNP AVPVP} & Nodetype(Head)^="IJ" &
          ^tokntest("ADV", Ft(AVP#1), Lt, []) &
          ^tokntest(-1, Ft(AVP#1), Lt(first(Factrecs)), []) &
          (Compr(AVP#1) -> (Advrz | Quant)) &
          (Demo(AVP#1) -> ^J_state_zyoo) &
          (Intens -> Intens(AVP#1)) &
          (Tme -> (Quant | P_every_mai)) &
          (Tme(AVP#1) -> (Intens(AVP#1) | Compr(AVP#1))) &
           Lem^="hoka" &
           Lemma^in? set{早} )
--> AVP { %%AVP#2; Temp=segrec{%%AVP#1; -Quant;
            if (Quant) Nodetype="QUANP";
            if (Comp & ^Wa5(AVP#2)) -Mim; };
          Prmods=Temp++Prmods; Degree=Degree(AVP); -Temp;
          if (Compr(AVP#1) | (Lem(AVP)=="mou" & Quant)) +Comp;
       }
```

Figure C: Example of the difference display with master file



[6] This rule, taken from the NLPWin Japanese grammar, is read as "AVP with AVP to the left", which takes two adjacent nodes, whose categories are both AVP (adverbial phrase), and creates a new node that spans both of the input nodes, also labeled as AVP, whose head is the second AVP of the left-hand side of the rule (indicated by %%AVP#2 in the right-hand side of the rule). A rule can be as small as this one, or can be very large (up to hundreds of lines of code). Each language in NLPWin has about 100 to 150 phrase-structure rules, in 10 to 20 files that are language-specific. LF rules are also written in G and have a similar format, but the files are shared by all languages, as are most rules, to ensure the output of the LF component is consistent across languages.