Leo: an Architecture for Sharing Resources for Unification-Based Grammars

Jason Baldridge[*], **John Dowding**[†], **Susana Early**[†]

[*]Division of Informatics, University of Edinburgh
2 Buccleuch Place, Edinburgh, United Kingdom, EH8 9LW
jmb@cogsci.ed.ac.uk

[†]Research Institute for Advanced Computer Science
Mail Stop T27A-2, NASA Ames Research Center, Moffett Field, CA 94035-1000
{jdowding,searly}@mail.arc.nasa.gov

Abstract

Many mature systems for parsing unification-based grammars have been developed over the last two decades. They incorporate a variety of design decisions both in implementation and in the representations they use for grammatical information. The Leo project aims to provide an architecture for automating the sharing of grammatical resources among various systems so that one system can take advantage of specialized algorithms and tools that are implemented for the representations used by another. The project furthermore seeks to learn about best practice in the design of these representations and encode their principles in a new XML-based format. This paper describes initial work toward creating the Leo architecture and tools that convert between different representations.

1. Introduction

Over the years, many parsing systems involving broad coverage grammars and lexicons have been built by teams of highly skilled linguists and computer scientists. Requiring many years of development, such resources come at substantial cost. Unfortunately, these systems have used different and non-compatible grammatical representations; barring a few exceptions, these grammars and lexicons were not sharable, and the algorithms developed in one system could not be used with grammars developed in others.

Within the last two decades, there has been increasing interest in grammars developed in declarative representations and in formalisms which make significant use of complex feature structures and unification. These systems include Definite Clause Grammars (DCGs) (Pereira and Warren, 1980), PATR (Shieber, 1984), Alvey Natural Language Tools (AVNLT) (Boguraev et al., 1988), the Core Language Engine (CLE) (Alshawi, 1992), Gemini (Dowding et al., 1993), the Linguistic Knowledge Builder (LKB) (Copestake, 2001), XTAG (Doran et al., 2002), and Grok (Baldridge and Bierner, 2002). While the representations used by these systems have much in common, they also each have significant individual differences.

The Leo project aims to facilitate the sharing of grammatical and lexical resources developed under one system with the algorithmic components (parsers, generators, etc.) developed under another. One example where this would be quite desirable is the production of a language model for speech recognition based on a given grammar, such as that done for Gemini grammars and the Nuance speech recognizer (Rayner et al., 2000b). Excellent tools exist to provide a context-free approximation of Gemini grammars to improve speech recognition performance and ensure the speech recognizer only produces strings that the full syntactic grammar can handle. If grammars of non-Gemini systems can be converted into Gemini grammars, their users can utilize this complex algorithm.

Sharing of grammatical resources is also motivated by a desire to evaluate and compare systems on comparable data. For example, John Carrol's (1994) experiments comparing AVNLT parsers with the CLE parser on the same grammar required converting the AVNLT grammar to a common DCG-like representation. Similarly, sharing grammars across systems can allow duplication of scientific results. When Kiefer and Krieger (p.c.) wanted to duplicate the results in (Dowding et al., 2001) and compare them with (Kiefer and Krieger, 2000), they had to port the grammar used in that paper by hand from the Gemini notation into the Type Description Language (TDL) notation (Krieger and Shäfer, 1994).

We have implemented procedures to automatically convert grammars represented in the Gemini's Typed Unification Grammar (TUG) representation into an XML-based grammatical representation (and back), and from this XML format into the TDL notation used in the LKB. Thus, we can demonstrate grammars originally written in Gemini being used for parsing and interpretation by the LKB.

In addition to these practical goals, Leo is also motivated by the greater research issue of how to best represent grammars. We expect that by examining these systems and formalisms in the level of detail required to convert between formalisms, much will be learned about what are the best generalizations across systems, what is core to each approach, and what is peripheral.

The structure of the paper is as follows. §2 outlines the overall architecture of the Leo system. §3 describes the components of Leo, which are now publically available under an open source license. §4 highlights some of the details of Gemini's TUG representation and LKB's TDL representation and shows how the conversion is performed between them. §5 outlines the considerable future work remaining to be done, and §6 concludes. An appendix is also included that provides an index to the many terms and abbreviations used throughout the paper.

2. The Leo Architecture

Any architecture attempting to extract the commonalities of various systems and abstract away from their idiosyncratic differences must provide some means of standardization of the data structures involved and the relationships predicated between them. Many data structures are quite common in most systems and are easily translated from one to the other. For example, consider how features, which in their simplest form are comprised of an attribute and an atomic value, are represented in Gemini, the LKB, and Grok:

(1) `vform=finite` (Gemini)

(2) `VFORM finite` (LKB)

(3) `<f a="vform" v="finite"/>` (Grok)

Such structures are trivial to translate. However, the situation gets more complicated very quickly: both Gemini and the LKB permit the values of features to be recursive, whereas Grok does not. Gemini relies on recursive feature structures to handle gap-threading, which Grok eschews in favor of a pure categorial grammar analysis that does not employ gaps. In the LKB, recursive (typed) feature structures are the representation for virtually *all* grammatical information, whereas Gemini uses syntactic categories that have recursive feature structures (of less global impact) attached to them. Feature values in Gemini and Grok can contain boolean values, whereas the LKB supports them only as a special case. These differences all lead to serious divergences in the way that similar data structures are used to define different kinds of grammars.

When we consider the relationships predicated between different data structures in a grammar, the situation becomes yet more complex. LKB grammars are defined as a type hierarchy in which inheritance plays a major role. Grok permits the declaration of one category type (e.g., the category for transitive verbs) to inherit the specifications of other types (e.g., the intransitive category), but grammars nonetheless can be defined entirely without such inheritance and there is no use of inheritance outside this context. Gemini does not utilize inheritance of any kind. Both the LKB and Grok are lexicalized, whereas Gemini is not.

Finally, there are declarations which are of a system specific nature. Some systems reference programming language specific data structures, such as Gemini's references to Prolog list types "`[]`" and "`[_|_]`". Systems also support meta-declarations which provide instructions to the algorithms in order to improve performance or indicate where certain types of information will be found in a given grammar. For example, an LKB grammar is generally accompanied by a series of Lisp statements that tell the LKB how to tie orthography to lexical entries, how to display the output of a parse, etc.

Ideally, the goals of Leo would be met by the creation of a universal core representation into which all unification-based grammars could be mapped. We thus envision a star shaped architecture such as that depicted in Figure 1 in which conversion tools are written to translate grammar resources to and from each of the various systems and the core representation.

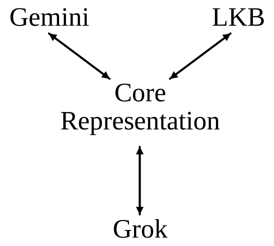

Figure 1: Star architecture with core representation.

For a number of reasons, XML is a natural choice as the representation language for such a standard. First, it is an emerging standard for data representation that allows structured representations of typed objects. XML allows a clean separation between the underlying structure, the DTD or schema that provides the semantics for that structure, and the data itself. Secondly, XML is not intrinsically tied to any particular natural language or programming language. Third, using standard tools available for a wide range of programming languages, the validity of an XML document can be verified with respect to a DTD or a schema, thus reducing the programming burden for utilities which manipulate and use the data held in the document. Finally, standardization of data representations through XML facilitates the creation of system independent visualization tools.

Despite the obvious appeal of such a core representation, it may not be attainable, nor is it presently practicable as there is still much to learn about the landscape of different systems and their representations. Nonetheless, a given representation will share more traits with some representations than it will with others. This leads naturally to a strategy by which we move toward the creation of the core representation *incrementally* by building a network of XML grammar representations (XGRs). This groups similar systems and thus shortens the conversion "distance" between many of the systems. The XGRs thus act as local versions of the ideal core representation. Even though a universal core representation is lacking in this architecture, it nonetheless provides an XML core layer that places system specific representations on the periphery and facilitates the creation of translation highways between different XGRs in that layer. This architecture is intended to grow rather organically, taking advantage of commonality wherever possible, without mandating design decisions upon others or leaving them out due to incompatibility.

For example, Combinatory Categorial Grammar (Steedman, 2000) (CCG) and Tree-Adjoining Grammar (Joshi, 1988) (TAG) are both mildly context-sensitive formalisms (often lexicalized in implementations). While they do have quite different properties, they perhaps share more in common than they do with many other frameworks. For example, both are lexicalized, both use a small set of essentially invariant (and similar) rules, both have an extended domain of locality via categories or elementary trees, and both associate non-recursive feature structures with syntactic categories. If this commonality can be captured by, for example, a Mildly Context-sensitive Grammar XGR (MCSG-XGR), then implementations of CCG and TAG such as Grok and XTAG need only provide translations to that, from which point translators to and from the MCSG-XGR can be used to create TUG or TDL grammars, as depicted in Figure 2.

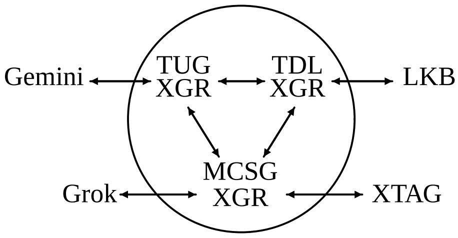

Figure 2: Networked architecture.

The networked architecture outlined above demonstrates how systems like the LKB and Grok can take advantage of the algorithm for converting Gemini grammars to language models for speech recognition. For the LKB, a translator would have to be provided between TDL specifications and TUG-XGR. The already existing utility for turning TUG-XGR into the Prolog terms of Gemini's XGR would then handle the next conversion, after which Gemini could operate on the resulting converted grammar. In the case of Grok, given that the network can translate between MCSG-XGR and TUG-XGR specifications, the Grok developers only need to provide the means to produce MCSG-XGR from CCG-XGR. The rest of the work is then carried out within the network. Although these converted grammars may suffer some deterioration at each step depending on the quality of the translations, the procedure would at least allow the Gemini algorithms to be used where they could not be otherwise. (In fact, the limited expressive power of these language models inherently forces many aspects of the grammar to deteriorate no matter how robust the conversion is). The resulting language model is then used at the speech recognition stage, after which the original grammar would still be used for actual parsing.

This example highlights the redundancy available in the architecture. It might be far easier to develop a translator from Grok's representation to MCSG-XGR than from Grok to TUG-XGR, so Grok grammars could go through three levels of conversion to reach Gemini. However, if the deterioration of the grammar was too high through this path, a translator could be written, perhaps at greater development expense, to convert directly to TUG-XGR. The decision is then one of the cost/benefit variety: a quick solution might get much of the way toward the desired result and be better than nothing at all, but the more elaborate and accurate solution would nonetheless be an option.

It may be that a satisfactory core representation can never be defined, but we believe it is important to keep that ultimate goal in mind. As the networked architecture is developed, the points of commonality will be extracted as far as possible in order to create archetypical representations which do not necessarily stem from any particular system. The XGR for a given system's data representations will thus be defined by importing common structures in conjunction with system specific declarations, thereby reducing the complexity of tools for converting resources from that system to a form processable by others. Once these archetypical representations have matured and stabilized, they can be utilized to create an approximation of the core representation.

Once some approximation of the core representation has stabilized, algorithms such as the one which creates language models from Gemini grammars can be implemented based off that XGR, completely bypassing issues such as dependence on Prolog and Prolog data structures and creating a more direct line from other systems to the representation used by the algorithm. This is precisely one of the main purposes for developing this architecture and is what we mean by the sharing of linguistic resources. It is not intended to be just the sharing of data, but also the algorithms for manipulating that data.

Given that the architecture is meant to grow through the involvement of different groups targeting either Leo's or each other's representations, it is crucial that the core facilities provided by the architecture be freely accessible and modifiable by these groups to improve the speed of Leo's uptake and its convergence on better representations. We thus felt that it was imperative for Leo to be an open source project which encourages code reuse and modification.

3. Resources

The implementation of the architecture described in the previous section is in its initial phases of development and several resources are already available. A project space has been set up on the Sourceforge site for open source development (`http://sf.net`), providing Leo with a wide range of project utilities, including web space, developer and anonymous CVS, release management, and discussion forums. Leo is associated with the OpenNLP project (`http://opennlp.sf.net`), which endeavors to improve the interaction of different open source projects for natural language processing at both the level of code and and the level of communication and collaboration.

A number of facilities have been written to commence the networked architecture and provide examples of its use. These include the following:

- an XML schema acting as a TUG-XGR
- a converter for transforming TUG grammar instance into HTML for improved visualization
- Java classes for working with Prolog terms and a parser which builds Java Prolog objects from Prolog statements
- Java classes for representing TUG grammars
- Java classes for working with TDL and a parser which builds Java TDL objects from TDL statements
- a converter between Gemini grammars and the TUG-XGR
- easy validation of a TUG grammar instance against the TUG-XGR
- a converter from the TUG-XGR to TDL

- a robust and flexible build system and scripts for easily invoking the various algorithms

Together, these facilities lay the foundations and act as examples for building Leo's network of XGRs and converting between them. Some are still incomplete as they have thus far been used to test the feasibility of the overall approach and the efficacy of the technologies chosen for implementation.

Most of the algorithmic components have been thus far implemented in Java for platform independence and because of the availability of excellent tools, such as Xerces and JDOM, for working with XML in Java. However, there is no particular barrier to eventually including algorithms written in different languages. Also, the visualization utility for tranforming TUG grammars into HTML is implemented as a XML style sheet which is invoked by an XSLT processor, thereby avoiding any programming language whatsoever.

Leo is presently bifurcated into two sub-projects. The first is a simple Java package for handling Prolog terms, and the second is Regulus, which holds the XML declarations for algorithms for working with specific formalisms and converting between them. It is envisioned that other sub-projects will be added for further functionality, such as parsers and user interfaces which act upon Leo's data structures natively and otherwise enhance the utility of the XGRs defined in Leo. The following sections provide further detail on the existing components.

3.1. The TUG XML Grammar Representation

The XML Grammar Representation for TUG grammars is provided in Leo as the XML schema `tug.xsd`. At present, this schema is mainly a transparent translation of the declarations used in Gemini grammars. However, as more schemas are defined for other formalisms, the commonalities can be extracted out and the schemas for different grammars will instead import the XML element declarations which they share with others. Nonetheless, the present schema provides an example and demonstrates many of the advantages of using XML and using schemas instead of DTDs to define validity. In addition to permitting all of the validation capabilities of DTDs, schemas allow a much stronger declaration of typing than is possible with DTDs. They permit a limited amount of polymorphism of XML elements, which might be particularly useful if the schema for a given formalism is actually built through the importation of common structures which are then slightly customized for that formalism. They furthermore provide the means to use regular expressions to enforce certain formats for textual elements, a simple example of which is that for Prolog variables:

```
<xsd:simpleType name="varString">
 <xsd:restriction base="xsd:string">
   <xsd:pattern value="[A-Z_][\w_]*"/>
 </xsd:restriction>
</xsd:simpleType>
```

Another capability of schemas is that uniqueness of attributes can be declared with local scope, whereas DTDs only permit uniqueness globally. While not used at present, this could be well suited for ensuring that variables are properly scoped in grammar elements, e.g. syntactic rules.

The advantage of these capabilities is that it simplifies the work needed when producing utilities to work with grammars in XML format. They can be automatically validated against the appropriate schema using the `validateTug` script provided with Leo, thereby permitting early detection of a large class of potential errors. This will be particularly useful when writing converters from grammars of one XGR to another — many problems in the conversion will be detectable through validation before even attempting to run them on a system that actually uses the target grammar.

A further advantage of schemas over DTDs is that document fragments can be more easily validated since there is no assumption of a document type element as there is with DTDs. Thus, an entire TUG grammar might be contained within one XML document and be checked against `tug.xsd`, but also a sub-part of the grammar, such as the lexicon, can be checked against the same schema in isolation. This will certainly be important since most systems split their grammars over multiple files (lexicon, rules, etc.) for the purpose of modularity, and this capability will facilitate validation of grammars stored in such a manner.

3.2. Visualization

Another advantage of specifying grammars in XML is the availability of related XML technologies. In an effort to further improve the readability of the XML instance, it may be transformed into a number of formats with XSLT. Currently, Leo has an XSLT style sheet which transforms a valid XML grammar based on the TUG-XGR and creates HTML pages via an XSLT processor. In this format, the representation is still system independent and the data is condensed, navigable via hypertext links, and more human-readable. Since an XSLT style sheet operates on identifying patterns in an XML instance, the present style sheet is quite specific to the TUG-XGR. However, as more schemas are defined for different systems and formalisms, this style sheet may be easily extended or modified to accommodate different patterns in various XML instances. Multiple style sheets would allow the user to customize the organization of the data in a familiar format or to output to different formats for printing or viewing via a web browser.

3.3. Java package for working with Prolog

The `opennlp.leo.prolog` package defines Java representations of Prolog terms and provides the means to parse Prolog statements and turn them into Java objects. Presently, these objects hold data and facilitate output in both Prolog and XML form. In the future, these will be extended to permit operations such as unification. Currently, this package does not cover all of Prolog, but was built to cover at least the constructs which appear in Gemini grammars. Though it is not yet complete, having a general Prolog-to-Java parsing utility has greatly facilitated the gradual extension of the coverage of Gemini constructs in the TUG-XGR and testing the conversion of Gemini to TUG-XGR on multiple grammars. This package has the interesting side potential of becoming a useful Java tool for working with Prolog.

3.4. Java TUG classes

The `opennlp.leo.regulus.tug` package defines Java representations of elements defined in the TUG-XGR. These classes can be used to build TUG grammars and output them in either Prolog or XML form. A Java representation of a TUG grammar can be created from either a Gemini grammar or an XML document conforming to the TUG-XGR, though we eventually expect these two specifications to diverge.

3.5. Java TDL classes

In a fashion similar to the previous two packages, the `opennlp.leo.regulus.tdl` package defines Java representations of TDL elements and provides the means to parse TDL declarations and turn them into Java objects. In addition to holding the data and facilitating output in TDL format, several ease-of-use functions have been built into this package to permit grammars to be created via Java declarations. In this respect, the package is similar to the JDOM package, which facilitates manipulation of XML documents within the Java programming environment. One of the most important features is that it enables one to hide much – though not all – of the tedium involved in generating a TDL grammar within a Java program. The need for this functionality arose during the creation of the TUG-XGR ⇒ TDL converter, but it will be useful for any other such conversion tasks which target TDL.

3.6. Accessing Leo

The homepage for Leo can be accessed at:

`http://leonlp.sf.net`.

Leo is licensed under the Lesser GNU Public License (LGPL), which permits anyone to obtain the source code to the system and in turn requires that any modifications which are made to the system itself be made available in return. In this respect, it is the same as the basic GNU Public License (GPL); however, it is different in that it permits usage of Leo software within a piece of proprietary software. This licensing scheme will be attractive to entities who work with linguistic software with restrictive licensing conditions, but wish to take part in the Leo architecture. While not requiring that users of Leo open up their own code (as the GPL does), the LGPL still ensures that any improvements to Leo itself will be returned to the community.

Leo utilizes a number of other open source tools, including JDOM, Xerces, Saxon, Jakarta Ant, the ANTLR parser generator, and the OpenNLP core package. Details for these tools are available in the documentation held in the main source code tree for Leo.

4. Converting between Grammar Representations

In this section, we describe in some detail the conversion from Gemini format into the TUG-XGR, and in turn into TDL format for use in LKB. We begin with a description of the format of Gemini and TDL grammars.

4.1. Gemini's Typed Unification Grammars

Gemini uses a notation that is based on Prolog term syntax, and similar to that used in the Core Language Engine (Alshawi, 1992). The terminology that is used is that grammar rules contain one or more *categories*, which are defined as an atomic *major category* symbol (e.g., nouns, verbs, etc.), and an unordered list of *feature* assignments (e.g., agreement, verb forms, etc.). Each *feature* is defined to contain values from a specific *value set* which define the underlying types. Some features can be defined to take category-values, so fully recursive complex feature-structures are supported. These definitions are illustrated by the following examples:

```
syn(s_np_vp, basic,
    [s:[],
     np:[num=Num],
     vp:[num=Num]]).

def_category s with_features [].
def_category np with_features [num]
     enable_lexicon.
def_category vp with_features [num].

syn def_feature num with_value_set
     num_types.

num_types def_value_set [sing,plur].
vforms def_value_set
  [[base,imperative,finite,ing,en,to]];
          enable_boolean_ops.
```

In this grammar fragment, there is one syntactic grammar rule whose name is `s_np_vp` and is in the `basic` rule group[1]. This rule contains 3 categories, `s:[]`, `np:[num=Num]`, and `vp:[num=Num]`, whose major category symbols are `s`, `np`, and `vp`, respectively. The `def_category` expressions define `s`, `np`, and `vp` as categories, and enumerate their possible features. The category `np` is further defined to be a possible lexical category with the `enable_lexicon` flag. The `def_feature` expression defines `num` to be a syntactic feature whose possible values are given in the set defined by `num_types`, which is further defined in the first `def_value_set` expression to contain the two values `sing` and `plur`. A second value set `vforms` is defined in the next line specifying that the value set can contain boolean combinations of values combined with disjunction and negation using the techniques of (Mellish, 1988).

4.2. The LKB and the Type Description Language

Rather than using categories with (optional) attached feature structures as the primary data structure, the LKB uses typed feature structures in a type hierarchy to represent

[1] Gemini allows syntactic and semantic grammar rules to be partitioned into independent rule groups, to allow independent sets of rules to manipulated in a modular way. However, this partitioning has never been put to use in an application and is redundant with other techniques available in Gemini to facilitate partioning the grammar.

all grammatical information. The hierarchy has a unique root element, usually called *top*. A sub-hierarchy for syntactic categories can be declared as follows:

```
cat := *top*
s := cat
vp := cat
np := cat
```

This only declares atomic subtyping, but attributes and values can be specified as well:

```
synsem-struc := *top* & [ SYN cat ]
```

The above declaration makes synsem-struc a subtype of *top* and specifies it to have a feature named SYN whose value must be of the cat type. To define a rule similar to the s_np_vp given above for Gemini, we can create a subtype of synsem-struc as follows:

```
s_np_vp_rule := synsem-struc &
  [ SYN s,
    ARGS [ FIRST  [ SYN np,
                    NUM #Num ],
           SECOND [ SYN vp,
                    NUM #Num ] ] ].
```

It is with this example that we see the recursivity of typed feature structures. It also shows how variables are declared in TDL, e.g. #Num. Notice that the value s for the attribute SYN is indeed a subtype of cat as required by the definitions of synsem-struc.

This manner of declaring grammatical information is very clean in that grammar writers only ever have to work with one kind of structure. It is also quite general and can be as readily used for parsing categorial grammars (Villavicencio, 2000) as it can for Head-Driven Phrase Structure Grammar (Pollard and Sag, 1994) (HPSG). Most grammars include extra Lisp code and special type definitions to inform the LKB as to where to find important elements such as orthography, rules, node names for parse trees, and semantic representations. The LKB is thus an agnostic system which liberates its users from theoretical constraints, but this freedom does come at a certain cost since so much choice is left up to the grammar writer. Fortunately, there is extensive documentation for the system; for more details, see (Copestake, 2001).

4.3. Conversion of Gemini grammars to the LKB

The process of converting a Gemini grammar into one useable by the LKB involves two stages. First, the script gemini2tug is invoked, which invokes Java algorithms that parse the Gemini grammar, turn its declarations into Java representations of Prolog terms, and then interprets their contents to create a Java object representing a TUG grammar. This object is then marshalled into XML format and saved as an XML document conforming to the schema tug.xsd. Using the validateTug script, this XML view of the TUG grammar can be validated against the schema to check for errors in the conversion.

Next, the tug2tdl script is run on the TUG grammar to invoke the procedure for converting it into TDL format for use with the LKB. The basic strategy is to create a type hierarchy that is modeled after many conventional hierarchies utilized in LKB grammars, but which is augmented by extensions needed to convert TUG specific types and values. Apart from the obvious concern surrounding the production of expressions in correct TDL format, a number of other issues, both trivial and non-trivial, arose during the conversion.

The LKB requires some supporting Lisp declarations in order to make a type hierarchy intelligible to it for parsing. Script and settings files that give the LKB this information are thus automatically included in the output of the conversion. Also, the type definitions for supplying parse node information to the LKB are created automatically during the conversion so that categories in a parse tree are properly displayed.

Another trivial matter to resolve was the fact that some TUG elements of different kinds have the same name. For example, n as in the bare noun category and n as in *yes/no*. The solution was to filter all different kinds such that the TDL types they produce are unique. Thus, TUG categories like vp and np are given types of the form c*vp and c*np, and valuesets such as number_types are v*number_types.

The type hierarchy provided a natural solution to the manner in which TUG grammars may associate a single syntax rule with multiple semantic rules. A tier was created in the hierarchy that contained only the syntactic information for a rule, and the types of this tier are then extended by multiple subtypes containing different semantic information.

Another fairly simple issue was that string literals can be used in a TUG rule. For example, consider a Gemini rule which can be used for sentences such as *who has the most field goals*:

```
syn(query_superlative_most, basic,
    [query:[],
     who, have_verb:[],
     the, most, cumulative_stat:[]]).
```

In the TDL translation, it was necessary to treat these uses of strings as categories by creating the *word* subtype of cat, adding subtypes of *word* such as w*the and w*most, and finally adding lexical entries which anchor those categories, such as the entry for *the* given below:

```
l*the := lexeme &
    [ ORTH <! "the" !>,
      SYN w*the ].
```

Feature specifications are included as subtypes of cat which specify the relevant feature's attribute and the type of its value. Categories which use those features then become subtypes, via multiple inheritance, of each of the feature types which the TUG grammar declares for them:

```
f*agr := cat & [ AGR v*agr_vals ].
f*vform := cat & [ VFORM v*vforms ].
c*vp := f*agr & f*vform
```

A more difficult problem is that boolean values are supported in Gemini, but not in the LKB. To solve this, a special utility was written to build a sub-hierarchy for disjunctive and conjunctive values. To avoid expanding all possible combinations, the grammar is probed for the boolean combinations which are actually used and the hierarchy is constructed with only those values.

Another more challenging issue is that TUG values can be simple atoms and complex Prolog terms. A sub-hierarchy defining Prolog terms was developed and complex values use sub-types of the `prolog-value` sub-type of `value`, which declares a feature `PVAL` of type `prolog-term`. This sub-hierarchy is particularly crucial for permitting the Prolog-based semantics of a TUG grammar to be expressed.

TUG values can also be syntactic elements, such as those used for gap threading. The use of such elements creates sub-types of the `category-value` sub-type of `value`, which declares a feature `CAT` of type `cat`. However, though these types can be represented, the converter does not yet handle gap-threading. The gap problem is interesting because TUG is not lexicalized and therefore can simply include an empty rule which introduces the gap. The LKB, however, is lexicalized and cannot utilize such rules. The most apparent solution would be to apply the gap rules to the rest of the rules to produce a further set of rules with the gaps already instantiated. It should be straightforward to do this once unification has been implemented for the `opennlp.leo.prolog` package so that the appropriate rules can be selected for such expansion.

Due to development time limitations, another issue which has not been handled is morphological information. Both Gemini and the LKB handle morphology in particular ways which requires some care in the translation, but there is no barrier in principle to handling morphology.

The utility can presently convert at least a medium-sized TUG grammar such as NASA's grammar for the Personal Satellite Assistant (PSA) demonstration (Rayner et al., 2000a). Preliminary testing indicates that the number of parses found by the converted grammar are the same as those found by the Gemini parser for most sentences. LKB parse times for sentences such as *measure the temperature and pressure in the crew hatch and lower deck and go to the storage lockers* are comparable to Gemini parse times – under a quarter of a second. A full logical form is produced by the parse which is represented by a TDL term that corresponds transparently to the Prolog terms built by the Gemini parser.

The Gemini-to-LKB pathway thus stands as the first example of the application of a series of grammar conversion utilities according to the Leo architecture. Currently, the utility converts directly to TDL since the TDL-XGR is not fully developed. Thus, all error checking has so far been done by loading the converted grammar into the LKB. However, once the TDL-XGR is completed, it will be a simple matter to output to XML and validate the grammar against the schema before even attempting to load the grammar in the LKB.

Because of the generality of TDL, it is expected that it will be considerably more difficult to develop a generic tool to convert TDL grammars to the TUG-XGR. It might be necessary to provide converters between some standardized usage of TDL to TUG-XGR, and then provide converters to that standard within the scope of TDL.

5. Future Work

The obvious next step for Leo is to work on conversion utilities to support other formalisms and systems and thereby refining the architecture and creating an approximation of the gold standard XGR. Though Leo stems from mostly practical goals, it is also motivated by the greater research issue of how to best represent grammars. We expect that by examining these systems and formalisms in the level of detail required to build translators, much will be learned about what are the best generalizations across systems, what is core to each approach, and what is peripheral. We furthermore expect that insight gained from working with various representations will lead to improved modularity in grammar declarations. For example, various representations could be developed for different ways of encoding semantic information. Then, semantic information could be paired with syntax in a modular way that allows a given system to more easily support different encodings.

At some stage, Leo may also include a common set of tools and test beds for evaluating grammars which hail from diverse backgrounds. Because different formalisms and implementations use different data structures and operations for combining them (and hence will exhibit a range of computational power), the translation between any two representations might lead to deterioration. An important subtask of the Leo project will thus be to provide some measure of the deterioration, if any, caused by translation to and from different XGRs. This in turn would provide an excellent formal basis for the comparison of various frameworks with respect to their coverage given comparable (via translation) grammatical resources.

The architecture described here thus provides a way of empirically evaluating different frameworks and studying to what extent a given grammar in a given formalism utilizes the formal power it provides. Thus, by translating an HPSG grammar to a context-free grammar, we may be able to measure how much, if any, grammatical coverage is lost in the process. The validity of the translation utilities themselves can be checked by translating grammars through loops which return to the original format and checking for errors and loss of coverage.

Finally, we aim to provide parsing, generation, and language modeling components which directly utilize Leo's own XGR after it has stabilized. Leo will then act as a highly functional open source NLP system which sits at the center of a number of utilities for converting grammar resources to its core representation and which itself operates on a representation created to accommodate many perspectives on the encoding of grammatical information.

6. Conclusion

We have described an architecture which aims to tackle the diversity of grammar representations and enable systems to share resources more easily. Though we do not yet propose an explicit standard representation, the architecture initiates a strategy for bringing systems and representations together in stages so that we can eventually converge on that goal.

Initial components for implementing this architecture have been created to demonstrate the utility of the technologies we have chosen and the feasibility of the approach. The value of providing XGRs has been shown via the tools for validation of grammar instances with XML schemas and their transformation into HTML for improved visualization. Most importantly, the conversion path for turning a Gemini grammar into one useable with the LKB produces a robust type hierarchy that appears to have similar performance to the original grammar under Gemini.

7. Acknowledgements

The majority of the research reported in this paper was performed at RIACS under NASA Cooperative Agreement Number NCC 2-1006.

8. References

H. Alshawi, editor. 1992. *The Core Language Engine*. MIT Press, Cambridge, Massachusetts.

J. Baldridge and G. Bierner. 2002. The Grok homepage. http://grok.sourceforge.net.

B. Boguraev, J. Carroll, E. Briscoe, and C. Grover. 1988. Software support for practical grammar development. In *Proceedings of the 12th Conference on Computational Linguistics (COLING)*, pages 54–58, Budapest, Hungary.

J. Carroll. 1994. Relating complexity to practical performance in parsing with wide-coverage unification grammars. In *Proceedings of the 32nd Annual Meeting of the Association for Computational Linguistics*, pages 287–294, Las Cruces, New Mexico.

A. Copestake, 2001. *The LKB System*. http://www-csli.stanford.edu/˜aac/lkb.html.

C. Doran, B. A. Hockey, A. Sarkar, B. Srinivas, and F. Xei. 2002. Evolution of the xtag system. In A. Abeille and O. Rambow, editors, *Tree Adjoining Grammars*, pages 371–404, Stanford, CA. CSLI Publications.

J. Dowding, J.M. Gawron, D. Appelt, L. Cherny, R. Moore, and D. Moran. 1993. Gemini: A natural language system for spoken language understanding. In *Proceedings of the Thirty-First Annual meeting of the Association of Computational Linguistics*, Columbus, OH.

J. Dowding, B. A. Hockey, C. Culy, and J. M. Gawron. 2001. Practical issues in compiling typed unification grammars for speech recognition. In *Proceedings of the Thirty-Ninth Annual Meeting of the Association for Computational Linguistics*.

Aravind Joshi. 1988. Tree Adjoining Grammars. In David Dowty, Lauri Karttunen, and Arnold Zwicky, editors, *Natural Language Parsing*, pages 206–250. Cambridge University Press, Cambridge.

B. Kiefer and H. Krieger. 2000. A context-free approximation of Head-Driven Phrase Structure Grammar. In *Proceedings of the 6th International Workshop on Parsing Technologies*, pages 135–146.

H. Krieger and U. Shäfer. 1994. TDL: A type description language for constraint-based grammars. In *Proceedings of the 15th International Conference on Computational Linguistics*, pages 893–899.

C. Mellish. 1988. Implementing Systemic Classification by Unification. *Computational Linguistics*, 14(1):40–51.

F. Pereira and D. Warren. 1980. Definite clause grammars for language analysis – a survey of the formalism and a comparison with augmented transition networks. *Artificial Intelligence*, 13:231–278.

C. Pollard and I. Sag. 1994. *Head-Driven Phrase Structure Grammar*. University of Chicago Press.

M. Rayner, B.A. Hockey, and F. James. 2000a. A compact architecture for dialogue management based on scripts and meta-outputs. In *Proceedings of Applied Natural Language Processing*.

M. Rayner, B.A. Hockey, F. James, E. Bratt, S. Goldwater, and J.M. Gawron. 2000b. Compiling language models from a linguistically motivated unification grammar. In *Proceedings of COLING 2000*.

S. Shieber. 1984. The design of a computer language for linguistic information. In *Proceedings of Coling84, 10th International Conference on Computational Linguistics*, Stanford University, Stanford, California.

Mark Steedman. 2000. *The Syntactic Process*. The MIT Press, Cambridge Mass.

A. Villavicencio. 2000. The use of default unification in a system of lexical types. In Detmar Meurers, Shuly Wintner, and Erhard Hinrichs, editors, *Linguistic Theory and Grammar Implementation*, pages 81–96. ESSLLI.

A Index of Terms and Abbreviations

Due to the large number of abbreviations and terms contained in this paper, we summarize them here.

CCG Combinatory Categorial Grammar

CVS Concurrent Version System

DTD Document Type Definition, a format for specifying the semantics of XML documents.

GPL GNU Public License

HPSG Head-driven Phrase Structure Grammar

LGPL Lesser GNU Public License

LKB Linguistic Knowledge Builder

MCSG Mildly Context-Sensitive Grammar, includes TAG and CCG.

TDL Type Description Language

TAG Tree Adjoining Grammar

TUG Typed Unification Grammar

XGR XML Grammar Representation

XML eXtensible Markup Language

XSL eXtensible Style Sheet Language

XSLT XSL Transformation, an extension of XSL.