Lex, a software project for linguists

Draft 1, 2009/03/25.

This paper describes a tool called Lex, developed initially to assist linguists with analysis of the Hebrew old testament and now being extended to other languages. We will report on our use of the corpus linguistics database Emdros and how it could be useful to linguists working in other languages; the other linguistic and practical features of Lex; and our plans for future work, including machine translation.

Introduction

Lex is a tool designed to help linguists to perform detailed analysis of text corpora and to publish their results:

- Browsing the corpus
- Searching for words (simple)
- Searching for grammatical constructs (advanced)
- Parsing (using a database of parser rules)
- Tree drawing (parse trees and manually)
- Transliteration (for exchange with other linguists)
- Logical structure definition (RRG theory)
- Semantic categories or ontologies of verbs (assists in translation and glossing)
- Lookup in other corpora (for glossing)

There is explicit support for data security, user access control, data separation (multiple databases) and controlled publication (granting public access to defined subsets of the data).

Lex could also be seen as a graphical user interface to a corpus database tool called Emdros. It adds many features which are not in Emdros, such as a lexicon, but fundamentally relies on Emdros for much of its work.

Lex is the result of a collaboration of three people over several years:

- Nicolai Winther-Nielsen, PhD, Hebrew old testament and theology scholar and lecturer
- Chris Wilson, MA, computer scientist and software developer
- Ulrik Sandborg-Petersen, PhD, computer scientist, linguist and software developer

Many others have also assisted and contributed ideas, particularly in the Role and Reference Grammar community, where we are especially indebted to Robert van Valin, Dan Everett, Brian Nolan, Ricardo Mairal and Elizabeth Guest.

Lex is now being extended to incorporate support for other languages, and for machine translation between languages.

Requirements

Lex was initially designed and developed as a tool for machine analysis and translation of Biblical Hebrew (BH), based on a database provided by the *Werkgroep Informatica*, the Department of Biblical Studies and Computer Science at the Faculty of Theology, *Vrije Universiteit*, Amsterdam (WIVU). Lex needed to support the following features of this specific database:

- single WIVU BH corpus organised by book, chapter, verse and clause;
- Hebrew surface consonant form, and archaic machine representation of Hebrew characters and inflection;
- complete (but possibly incorrect!) encoded syntactic tagging of clauses, parts of speech, nucleus, arguments and subject and object (not PSA).

In order to provide a complete workbench for machine translation, the entire RRG work flow must be supported and understood by Lex. This means that Lex must understand each step in the processes of conversion from syntax to semantics and back. It must also be possible to automate each step.

The processes of conversion between syntax and semantics have been broken down as follows. This list may not be complete, because Lex is not yet finished, however we are nearly there. Lex should support the following:

- multiple languages, by definition for a machine translation project;
- storage of corpus text in a database, and enable easy searching and retrieval of corpus text, to help linguists to develop and test lexica and parser rule sets;
- multiple corpora in each language, to help linguists to correlate across different corpora and check their work;
- transliteration of local languages, to help users to prepare papers and reports on their work;
- sharing of corpora between different users, while isolating their changes to enable them to work independently;
- construction and testing of morphological rules, to enable first-level morphology transformations and to generalise the lexicon from surface forms to semantic primitives;
- construction and testing of parsing rules (templates), to enable automated parsing of clause structure, for machine and machine-aided translation;
- easy experimentation with new parser rules and examination of their effects on the automated parsing of the corpora, to enable efficient development and refinement of rule sets;
- construction and maintenance of the lexicon, which is crucial for converting the syntax tree into logical structures and back;
- connection of the lexicon to the corpora, to assist with development of the lexicon;
- evaluation of the actual semantics (logical structures) in the corpora;
- parsing of logical structures back into syntax in another language;
- rendering syntax trees as surface text, including reverse morphology.

Some features of the project are not linguistically motivated, but rather by a desire to encourage sharing and collaboration in the community, and to build useful tools for the community. Therefore, we decided that Lex must also support:

- assistance for collaboration between researchers working at remote locations;
- low cost availability and easy installation of the software;

- flexibility for future development through open source code and well-structured code;
- customisation of the software through source code availability;
- reliability through unit and integration testing.

Design Decisions

Possible means to satisfy each requirement are analysed, one is chosen and the reasons for the choice are explained below.

WIVU BH Corpus

The WIVU BH corpus is available in two forms: as multiple ASCII table files in fixed column width format, designed for use by WIVU's original Pascal software; and as an Emdros database which allows efficient searching and retrieval of complex data structures.

We could have used either format, but the Emdros data was significantly easier to access and use. We did have to overcome some hurdles in accessing the data from Java, and developed software libraries to make this easier. We have worked on Emdros to make this process easier, to resolve some bugs and to add missing features.

The corpus is licensed from WIVU, and unfortunately not fully available to the public, but we have permission to display certain parts of it as examples to the public, and full access may be granted on request by WIVU.

Hebrew Encoding

The WIVU corpus provides several encodings of the Biblical Hebrew (BH) text. As it pre-dates the Unicode standard, it does not use Unicode at its foundations. Recent versions have offered a Unicode (UTF-8) encoding of the Hebrew characters, but this was not available when we started work. It is also difficult for Western scholars who don't have a good knowledge of Hebrew to understand work that is based on these characters, and they are difficult to enter on a Western keyboard.

WIVU has always contained two other encodings: consonants only, and a full and very complex encoding of consonants plus vowels and cantillation marks expressed in numbers and punctuation. We used the former to connect the surface text to the lexicon, as a "rough and ready" alternative to developing a complete morphological analysis of BH that would enable us to completely undo the morphological inflection, and link the corpus and the lexicon using base forms of inflected words. This will need to be addressed properly to generalise Lex to other languages and corpora.

We used the full encoding to generate a transliteration of BH into the phonetic alphabet, to help linguists from other backgrounds to understand our work. We hope to publish this transliteration as a modern refinement of the original Anstey system on which it was based.

Complete Syntactic Tagging

The contribution of WIVU in producing their database was not just in entering the Hebrew characters onto a computer. They spent many years conducting important research on the corpus and machine analysis of BH, resulting in a machine-generated and manually checked and corrected database of syntactic structure of the BH corpus.

Access to this database gave us a unique advantage, as for a long time we did not need to develop a parser and we could rely just on the syntactic analysis. However, their analysis pre-dated RRG and uses several concepts which do not exist in RRG, so their work was not entirely suitable for RRG analysis.

Here is an example of their structural analysis of Genesis 1,1 from BH:

type / monad	1	2	3	4	5	6	7	8	9	10	11
book	1 (Genesis)										
chapter	1										
verse	1 (in beginning he created God (et) the heavens and (et) the earth)										
clause	28737										
phrase	PP/Time		VP/Pred	NP/Subj	PP/ObjC						
subphrase					mother				daughter		
word	prep	noun	verb	noun	prep	article	noun	conj	prep	article	noun
surface	*bə-*	*rē?šît*	*bārā? ?*	*ĕlōhîm*	*?ēt*	*ha-*	*ššāmayim*	*wə-*	*?ēt*	*hā-*	*?ārec*

Table 1: Syntactic structures from WIVU database for Genesis 1,1 (key components)

If we compare the tree structure of the WIVU data with the canonical RRG tree for the same clause, we can see that a lot of work is still required to convert the one into the other!

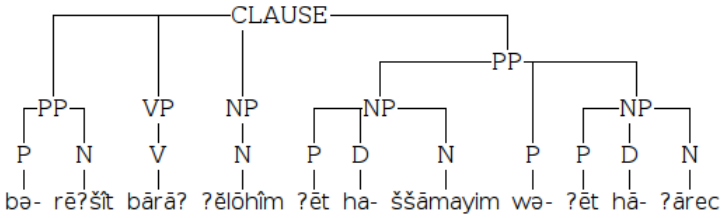


Illustration 1: Syntax tree from WIVU database (key components)

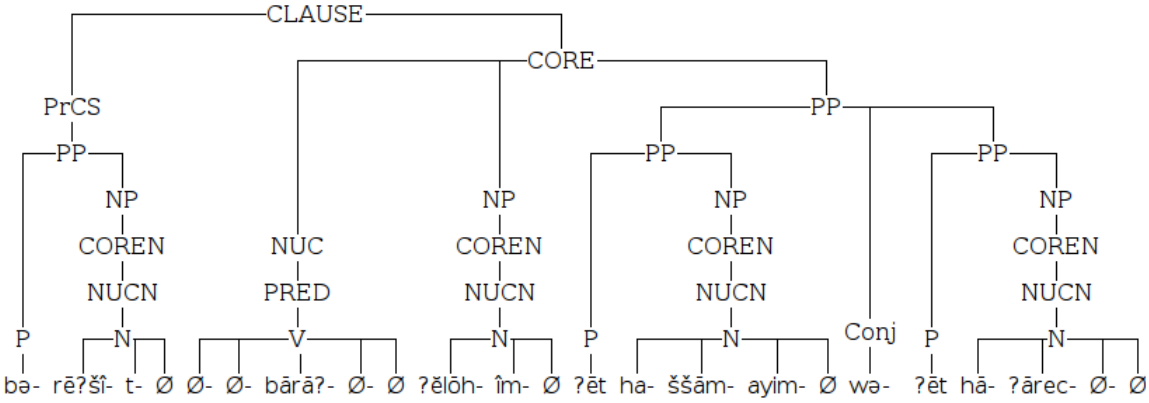


Illustration 2: Syntax tree in RRG form

Until we had a parser, we fed the *phrase* level units to the logical structure generator. Now we have replaced the WIVU syntactic structure with RRG from the word level up. Analysis at the sub-word level, i.e. morphology, is still done using data about *word* structures from the WIVU database, which will be discussed further under *Morphological Rules* below.

Multiple Languages

Storage of characters from any language on a computer requires the definition of a character set, which maps characters (*glyphs* or shapes) to binary codes for storage.

In the past, many disparate character sets were used, some overlapping or conflicting, sometimes with incompatible alternatives for the same language. This collection of characters sets could best be described as an absolute mess, and prevented and complicated the exchange of information between computers with different language settings (character sets) for many years.

Recently, a group of experts has developed a single character set called Unicode, which includes almost every known language and character in the world. The aim was to include every language and character, but some may have been omitted due to lack of knowledge or standardisation. We can expect the omissions to be corrected in subsequent versions of the standard.

The advantages of using Unicode over separate character sets to encode a corpus are many:

- the character set for a corpus need not be decided in advance;
- it is possible to mix multiple languages in a corpus;
- it is easy to exchange information in different languages between computers, and between software programs on the same computer;
- the binary encoding is reasonably efficient and not difficult to implement.

Availability of Unicode fonts used to be a major problem, as several traditional or dead languages had non-Unicode fonts available but no Unicode fonts. With the growing popularity of Unicode, it seems likely that most living languages already have Unicode fonts, and any new fonts developed for new languages are more likely to be Unicode than not.

Unicode encoding may be less efficient, using more disk space, memory and CPU power, than language-specific character sets. This is inevitable due to the flexibility of the Unicode character set, and the choices made by the designers of Unicode. In the light of the low and falling cost of memory and disk space, and the quantity of textual information that are likely to be processed by the software package, the authors do not consider this a significant problem.

No alternative universal character set is known to the authors. The decision therefore comes down to Unicode versus individual language-specific character sets. The advantages of Unicode over those character sets are considered to outweigh the disadvantages.

Luckily, the WIVU database now includes Unicode (UTF-8) encoded versions of the Hebrew surface text, so we were able to display and work with actual Hebrew characters without any extra work in conversion between character sets. In addition, the standard Windows fonts have reasonable Hebrew glyphs, and better results can be obtained by the user installing the free SIL Hebrew fonts.

Storage of Corpus Text

Linguists will often work with a corpus of text in the languages that they study, since it provides a useful subject for research and for testing theories about the language. The corpus is traditionally stored as a flat text file, with characters in sequence, separated by spaces into words. This approach is simple and works well for small corpora, but it is not efficient to search larger corpora in this way, nor to store linguistic structural data alongside the text, nor to exchange revisions to the corpus with other researchers.

Ulrik Sandborg-Petersen, built on the work of Crist-Jan Doedens to develop a database specifically for storing corpora and structural data about their contents. This database is called Emdros, and it offers a number of advantages over traditional relational databases. Some of these will concern us in the next section, but for now it suffices to say that Emdros makes it significantly easier to navigate, browse and modify structured text than either flat files or relational databases.

The extent of this is shown by the fact that two companies which develop bible software, the German Bible Society and Logos, have purchased licenses to use Emdros in their commercial software. Emdros is the only database software known by the authors to be available and designed specifically for storage of corpora and linguistic structural data in any language.

The main disadvantage of Emdros from a user's point of view is that it does not have a graphical user interface, that is a simple graphical tool to allow users to enter and query their data. Thus, working with Emdros is currently significantly harder than working with a flat text file. In order to choose Emdros as the underlying database for this project, it would therefore be necessary to develop a simple user interface to make using Emdros easier to use, while also giving access to the advantages of the powerful structured data storage and query tools that it provides.

The authors expect that at least some linguists will want to work with reasonably large quantities of text, and therefore that the advantages of using Emdros outweigh the disadvantages. They also hope that the user interface to Emdros that will be developed as part of this project will be a useful tool in its own right.

Navigation of the corpus is currently completely tied to the WIVU database structure of Biblical Hebrew, and therefore needs an overhaul before other corpora can be used successfully and navigated easily. A more generic navigation system would allow the administrator to select some object types that would be used for navigation, and the Emdros *feature* that would be displayed in the drop-down box for each object type. It might be useful to apply processing to these features, such as transliteration. The current system is a special case of this generic one.

Below is a screen shot of the current navigation interface, and the object types and attributes used to populate it:

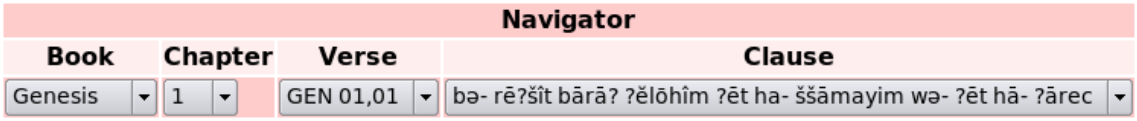


Illustration 3: Lex navigation interface for Biblical Hebrew

Order	Emdros Object	Emdros Feature	Processing
1	book	book (name)	none
2	chapter	chapter (name)	none
3	verse	verse_label	none
4	clause	[word GET many]	concatenate and transliterate

Table 2: Lex navigation interface generalised to Emdros objects and features

Searching of the corpus is currently possible in two ways:

- using a simple word search for Hebrew consonants against a single attribute of the [word] object type in Emdros;
- powerful but complex MQL search which allows any [clause] object to be selected by its features, or the objects or features that it contains.

It would be useful to introduce an intermediate type of search that would allow users to quickly select the object type, feature and value that they were looking for, which would be built into an MQL query. Claus Tøndering has developed an Emdros search engine like this in Java and we may be able to use some of the ideas or code in Lex.

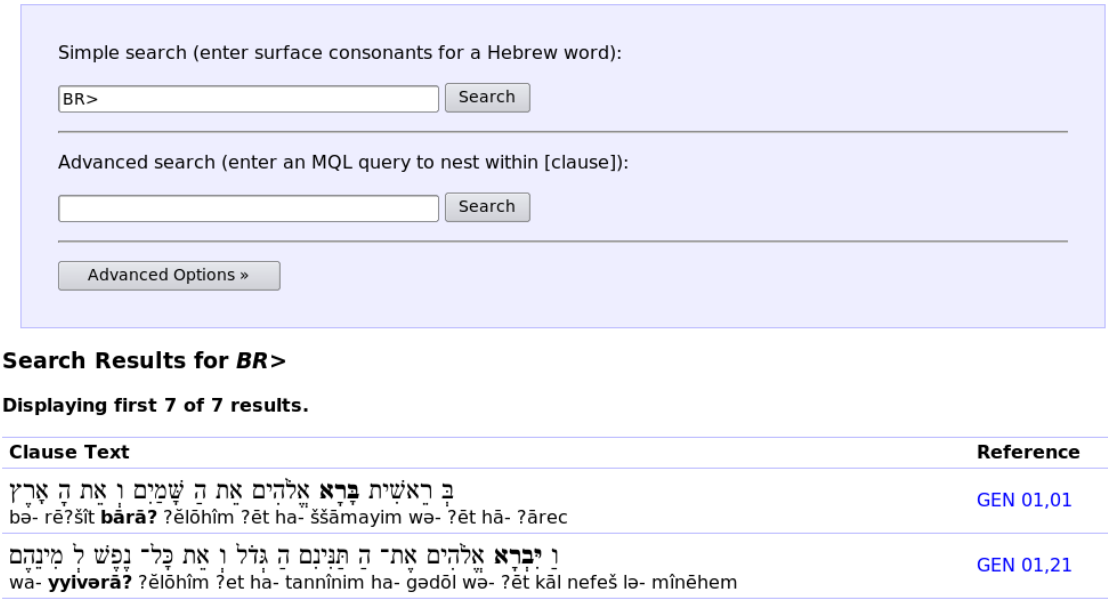


Illustration 4: Search interface for Biblical Hebrew, showing two results

As before, it would be useful here for the administrator to be able to specify the type of object returned (to control the amount of text, e.g. one clause or one paragraph), and the features and processing that would be displayed for each result. Below is an example of the current search interface, showing [clause] objects with the Hebrew and transliterated text, and a location indicator.

Multiple Corpora

Although Lex was originally developed using a single corpus and language, it must now be extended to support many of both, otherwise it will hold little interest for linguists outside of the BH domain.

This requirement is not difficult to achieve using any corpus database, whether flat files, Emdros or a relational database. Emdros has the notion of separate, independent databases within the same engine, which will be used to separate the corpora.

However, Lex also uses a database of its own, which stores the lexicon, parser rules and access permissions for each language, and this database must also be generalised to multiple instances.

The work of actually supporting multiple databases of either type in Lex has not yet been done.

Transliteration

Until recently, Lex was using a custom transliterator based on a specification for Biblical Hebrew by Winther-Nielsen. Of course, this was completely specific to BH. In addition, it was based on the archaic WIVU transliteration, which made it easier for non-Hebrew readers to understand the code, but at the cost of portability or relevance to other languages.

A new transliteration system has been designed and built and is under testing. The rules were developed by Winther-Nielsen and Tøndering, and consist of an ordered list of contextual rewrite rules. An excerpt of the list of rules is included below for illustration.

In database	Unicode	Name	Preceded by	Character	Followed by	Transliterate as
,	\u05bf	Rafe		,		Ignore
.	\u05bc	Dagesh		.		Ignore
/		Nominal indicator		/		Ignore
:	\u05b0	Sheva	^ cons	:		ə
				:	$	Ignore
			cons	:	cons : $	Ignore
			longvow cons	:		ə
			@ accent cons	:		ə
			consND .	:		ə
			consND	:		Ignore
:@	\u05b3	Hataph Qamets		:@		o

Table 3: New Biblical Hebrew transliteration rules (extract)

In the table above, the "in database" column is given in the archaic WIVU transliteration, which has been converted to Unicode (using the second column) for implementation of the transliterator.

Each line is a context-sensitive rule. It matches the wherever it sees "preceded by + character + followed by" (the concatenation of these three columns), and replaces the character in the "character" column with the character in the "transliterate as" column. *Ignore* means that the matched characters should be deleted (replaced by the empty string). The special characters "^" and "$" match the start and the end of a word, respectively, and *cons*, *consND*, *longvow* and *accent* are regular expressions. They would be simple character classes, but sometimes they represent multiple characters in the input. For example, the value of *cons* is the following regular expression:

cons = `(אעבדגהיכלמנפקרסתטוחצז]|שׁ|שׂ])`

For efficient implementation, each rule is converted into two compiled regular expressions, one matching against "preceded by" and the other matching against "character + followed by".

To transliterate a string, we iterate a pointer over the gaps between characters, starting before the first. At each position, we then apply each rule in turn at that position until one matches. When that happens, we consume the characters in the "character" column and output the ones in the "transliterate as" column, and move the pointer forward by the number of characters consumed (at least one position). If no rule matches at a position, that is a transliteration failure, and the original character is output as a debugging aid.

Unlike the previous solution, these rules are perfectly suited to being stored in a database table, and so they are. This makes it possible to completely change the transliteration without any programming, so it is ideal for developing new transliterations.

In addition, a large number of test cases were developed for the rules. There is at least one test case for each rule, sometimes more. These test cases can also be stored in a table in the database. It will be possible to explore the effects of a transliteration rule change by showing the input, expected output and actual output for each test case, highlighting the ones where the actual output does not match the expected one, before making the change. This would facilitate testing new rules and improving the transliteration. It should also be possible to add any number of new test cases.

Finally, in some cases it may be difficult to see why the transliteration did not produce the expected result. In these cases, it would be useful to analyse the transliteration process step by step, to see

which rule matched in each case and what the output was. This could be linked from the test case display, and also from other places where transliterations are shown.

Sharing of Linguistic Data

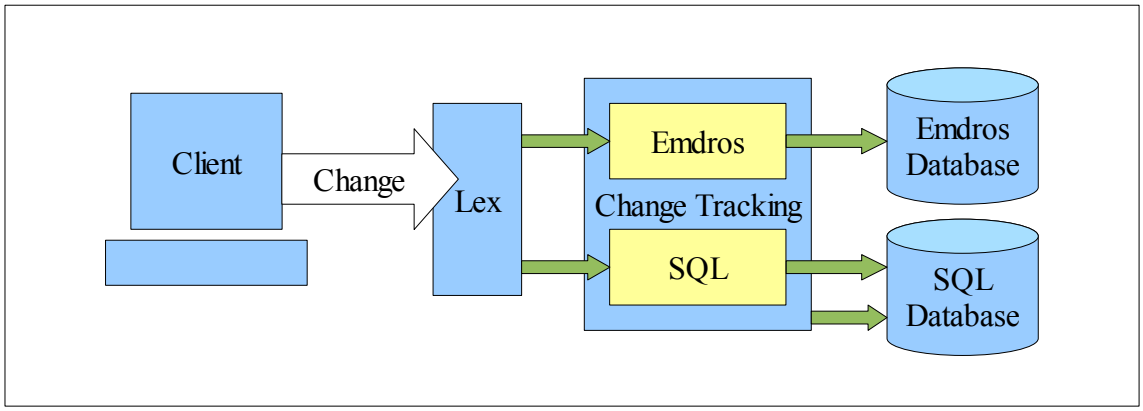
Related to the concept of multiple databases is the idea that several users might be working independently on a given corpus or language. They might then wish to exchange data with each other, such as the following:

- new or modified parser rules;
- new or modified lexicon entries;
- modified structural data in the Emdros database.

Lex should support the import and export of such data. It should be possible to export a subset of the entire corpus, parser rules or lexicon, and to import it on another computer or database, provided that it makes sense to do so (for example, the underlying language or corpus is the same). It should be possible to undo such operations and review changes. It should be possible to apply changes to existing objects rather than creating duplicates.

In addition, where multiple linguists collaborate on a single database, they should each be able to see the changes made by each other, as well as their own change history, and to undo changes if they are discovered to cause problems later.

Lex provides a foundation for import and export as well as change tracking and reversibility through its Database Change Tracking (DCT) layer. This layer sits between Lex itself and the underlying databases of Emdros and SQL. Although Emdros stores its data in a SQL database as well, Lex is not able to see this database or track changes to it directly. Instead, Lex treats the Emdros database as an opaque interface, and tracks changes to Emdros objects rather than the underlying SQL tables that Emdros uses for data storage.



Drawing 1: The Database Change Tracking layer (DCT) records changes to the Emdros and SQL databases in the SQL database

The DCT stores the following information about any change to a SQL database table or Emdros object:

- Logged in user name and IP address (that made the change)
- Current date and time
- The database type (either Emdros or SQL)
- The database name (for future expansion)
- The table name (SQL) or object type (Emdros)
- The command type (create object/row, update object/row, delete object/row)

- The unique ID (SQL primary key or Emdros object ID) of each object affected
- The old and new values of each field (SQL) or attribute (Emdros) changed

This data is stored in the same SQL database that Lex uses for the lexicon and for access control. Of course, the changes to the DCT tables are not themselves tracked. Also, changes to table and object structures (e.g. adding and removing columns or attributes) are not tracked, as these are made by administrators and not by end users of the software.

The storage of this data provides sufficient information to:

- view any changes made by any user to the databases;
- undo any change to the databases;
- export any change to be redone on a different set of databases;
- merge local and remote changes on import, and prompt for conflict resolution.

However, these features are not implemented in the user interface yet.

Morphological Rules

The importance of morphology to machine parsing and translation cannot be understated. Understanding and processing of morphology is required to:

- identify the primary syntactic argument (PSA);
- in the case of head marking structures, create the primary syntactic argument (PSA);
- provide semantic information such as tense that cannot be inferred otherwise;
- generalise the lexicon from surface to underlying semantic forms;
- produce grammatically correct output from machine translation in most languages.

Lex currently uses hard-coded morphology rules for Biblical Hebrew that use the information provided by the WIVU database to assist with morphological analysis. The reason for this is simply that the availability of this data makes the task of morphological analysis significantly easier and more reliable. However, unless it can be generalised, it will greatly increase the amount of work needed to build and maintain a lexicon for other languages in Lex.

One idea for generalisation of the lexicon is using expansion rules. Normal parser rules have at least one *non-terminal* symbol on the right, and just one *terminal* symbol on the left. The terminal symbol is placed above the non-terminals on the parse chart, and subsumes and consumes them, so the tree always becomes narrower as we approach the top, where there is a single root node that subsumes all of the others.

However, in morphological analysis, the opposite occurs. A single word, the normal unit of syntactic analysis, is broken up into multiple syntactic units with different functions. This causes an expansion which, if left unchecked, could carry on forever and result in parsing taking infinite time. Therefore, this is a potentially dangerous strategy. However, it may also be very productive, and provided that the same rule is never applied twice to the same word, infinite loops should be impossible as eventually all rules would be exhausted.

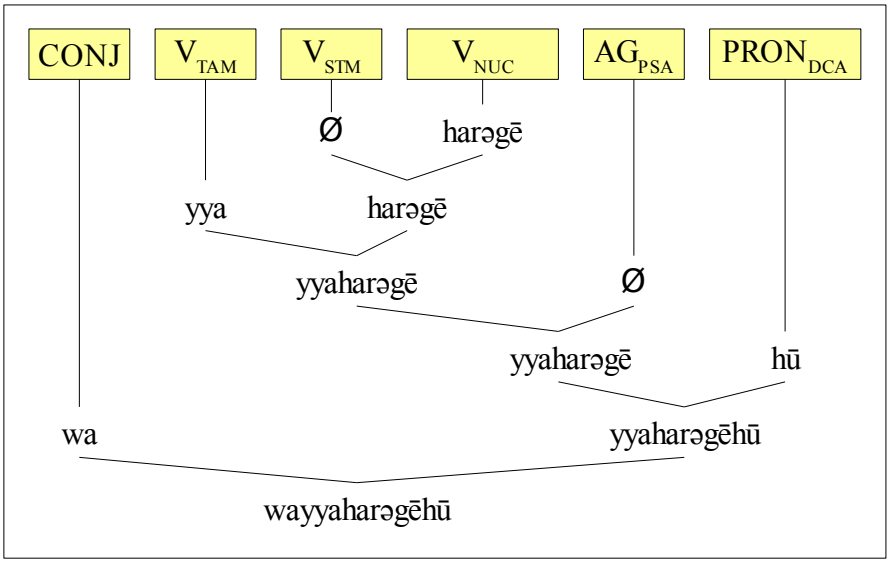
Let us take as an example the Hebrew word וַיַּהַרְגֵֽהוּ (wa-yya-harəgē-hū), “and he killed him”. This single word is a complete sentence in Hebrew. Here is an example of how the correct set of morphemes (according to Winther-Nielsen 2008) may be formed from the surface word, using morphological expansion rules.

Let us formulate some rules that allow us to split the word *wayyaharəgēhū* into six parts:

- the initial conjunction, *wa*, which links this clause to the previous one (CONJ)
- the tense aspect marker, *yya* (V_{TAM})

- the verbal stem, empty in this case, written as Ø (V_{STM})
- the verbal nucleus, *harəgē* (V_{NUC})
- the agreement clitic for the primary syntactic argument (PSA), empty in this case (AG_{PSA})
- the direct core argument head marker for the direct object pronoun, *hū* ($PRON_{DCA}$)

This diagram shows the morphology tree for the word *wayyaharəgēhū*. Above this tree, attached to the six top nodes, are the nodes of the usual syntactic parse tree, concentrating up to a CLAUSE at the top. Unlike that other tree, this one has its root at the bottom and expands upwards.



Drawing 2: Morphology of the Hebrew word wayyaharəgēhū*, "and he killed him"*

Let us write our morphology rules like this, in the opposite shape of traditional parser rules, but in the same sense, as they represent expansion and not concentration of the single OBJECT on the right.

OBJECT1 [surface1] OBJECT2 [surface2] → OBJECT [surface]

This rule means that OBJECT is split into two objects (OBJECT1 and OBJECT2) whose surfaces are *surface1* and *surface2* respectively.

Let us shorten and generalise our rules by making use of the asterisk (*) as a wildcard character, that matches any number of surface characters (on the right-hand side), and [1] as a reference to the characters thus matched (on the left-hand side).

The following five rules work to expand *wayyaharəgēhū* into the six objects shown in the morphology tree:

- CONJ [“wa”] V_{TSNAP} [1] → WORD [“wa*”]
- V_{TSNA} [1] $PRON_{DCA}$ [“hū”] → V_{TSNAP} [“*hū”]
- V_{TSN} [1] AG_{PSA} [Ø] → V_{TSNA} [“*”]
- V_{TAM} [“yya”] V_{SN} [1] → V_{TSN} [“yya*”]
- V_{STM} [Ø] V_{NUC} [1] → V_{SN}[“*”]

These rules are not specific to the verb *harəgē*, and may apply successfully to other compound verbs. They may even apply in cases where they should not, especially the rules on empty AG_{PSA} and V_{STM} objects, which could equally well apply to objects that do have an agreement clitic or a verbal stem, as nothing prevents them from doing so. In such cases, it’s important that the lexicon does not include verbs which still contain the agreement clitic or verbal stem. In other words, it must contain verbal nuclei only. Then, false matches of such rules will produce a “verbal nucleus” that does not exist in the lexicon, and will be discarded as impossible to parse.

Morphological rules as described above are not implemented yet. Only the hard-coded Hebrew morphological analysis is currently available.

Parser Rules

Canonical RRG theory describes a "linking algorithm" that fits surface text into a complete tree in a single step. However, the linking algorithm suffers from some problems in regard to computational linguistics:

- it is not fully specified, and leaves a lot to the linguist's intelligence and knowledge;
- it does not provide a way to specify or control how reordering of words can occur within a template;
- it is unproductive, in the sense that many similar templates sharing many common features will be needed for any given language;
- it does not facilitate the transfer of structure between languages.

The best-known grammars which are easy to process computationally are Noam Chomsky's *formal grammars*, which are widely used in computer programming. Chomsky later applied the same rules of formal languages to his controversial Universal Grammar, an attempt to model human languages in formal terms. RRG has rejected the Universal Grammar approach, and by extension formal grammars, as they are incapable of parsing languages with free word order without the use of transformations. They also tend to over-generate when used as generative grammars, potentially leading to infinite numbers of output templates, and structures which are never used by human speakers.

However, it is possible to solve both of these problems, and all of the limitations of template-based approaches, using a modified rule-based grammar which adds attributes, unification, and two types of permutations to traditional formal grammars:

- **attributes** are properties of symbols (terminal and non-terminal) which convey additional information such as the *tense* and *illocutionary force* of a verb or morpheme, without introducing new symbols that would break the generality of the grammar.
- **unification** allows attributes to propagate up the parse tree that would otherwise stay buried inside it. For example, the *tense* and *illocutionary force* of a verb or morpheme can be propagated up to the CLAUSE, as required by the RRG operator projection. Unification also allows rules to place arbitrary restrictions on how they link to other rules, and hence allows any number of templates of any size and shape to be constructed, without loss of generality.
- **permuting rules**, the first kind of permutation, can find their terminals in any order or permutation, but they must adjoin each other. If the rule CORE → NUC ARG ARG is a permuting rule, then the NUC and ARGs may occur in any order.
- **searching rules**, the second kind of permutation, may find their terminals anywhere within the input, and in any order, without the requirement that they adjoin. Searching rules are required to parse Djirbal, as in the rule NP → DET N, the matching determiner and noun can appear anywhere at all in the input.

Lex has developed and tested a rule-based active chart parser that implements all of these features, and can successfully parse Djirbal and many artificial test cases. The Lex parses is also reasonably fast. Parsing 9 words against 20 rules takes under 4 milliseconds on a four-year-old laptop.

Lex also has a user interface designed to make it easy to add new rules by simply selecting a set of adjoining nodes. This does not yet support the above features, which must be added manually after the rule is created.

Experimenting with Parser Rules

Changing a single rule can have far-reaching consequences on the entire corpus:

- clauses that previously parsed successfully might no longer do so, or vice versa;
- clauses that previously parsed unambiguously might become ambiguous, or vice versa.

There is a need for Lex to be able to display a list of clauses whose parsing is influenced by a rule change, before that rule change is made permanent. This should help to examine the effects of a rule change and adjust the rule, or rethink the strategy, or recheck the affected clauses if necessary. Unfortunately, this is difficult to implement as it could require applying the parser to hundreds of thousands of clauses, which would be extremely slow.

Lex includes a parser debugger which shows all completed edges generated by the parser for a given input. It might be useful to have the option to display incomplete edges as well, or to drill down to certain types of edges.

Lexicon Editor

The lexicon editor in Lex allows entries in the lexicon to be created, modified and deleted. Its most important function is to edit the logical structure (LS) of a lexicon entry, which is only relevant to verbs at present. Adverbs may have logical structures in Lex in future.

The LS can be hand-written, or can be created using the logical structure builder, which is Javascript code that runs in the user's browser and assembles the logical structure using the answers to various questions which correspond to the grammatical tests on pages 93 and 106 of [VVLP 97].

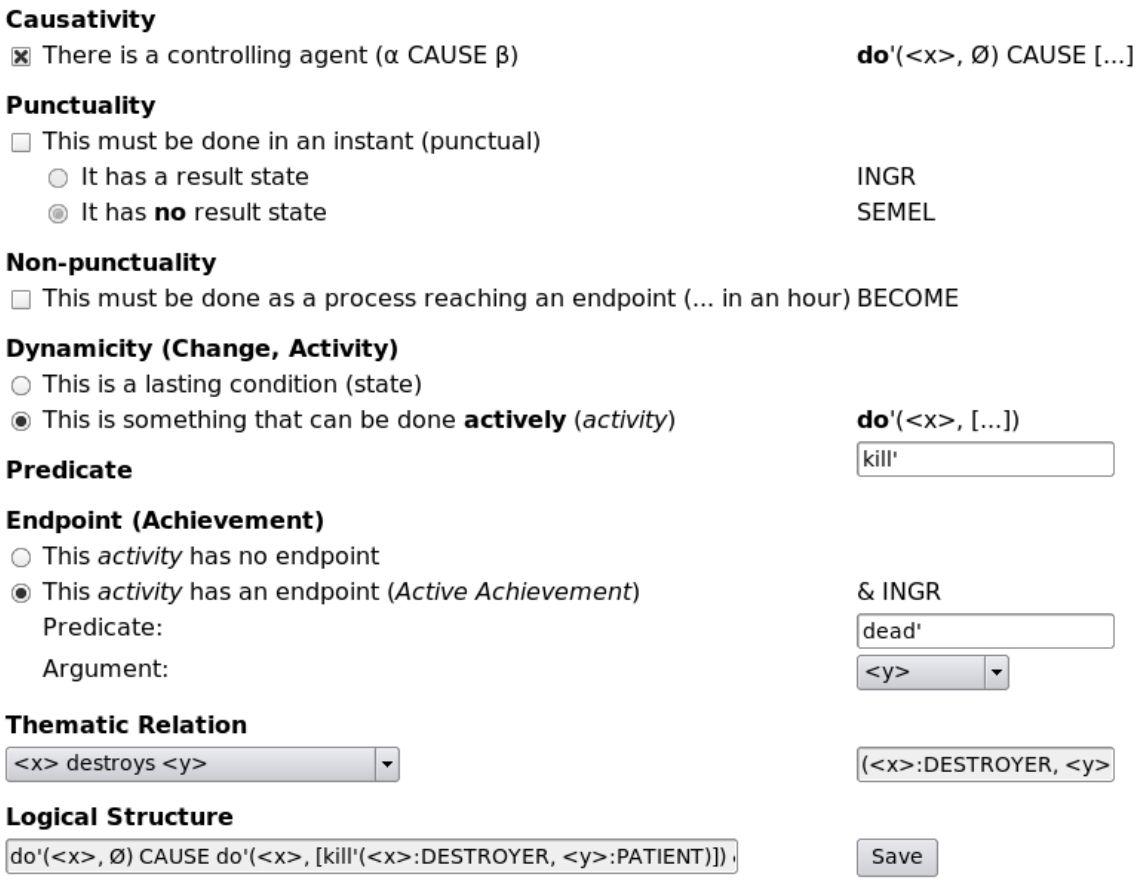


Illustration 5: The Logical Structure Builder for the lexicon in Lex

The LS builder helps to avoid mistakes in the entry of the LS, and also to ensure that structures for similar verbs in different languages are represented in similar ways and with properties that can help to match them more easily and accurately across languages.

Not every possible logical structure can be built using the LS builder. For example, [VVLP 97] defines causative structures as "α CAUSE β, where α, β are LSs of any type" (p. 109). However, using the LS builder above, we cannot build the LS (1) or (2) below:

1. [**do**'(Tom, ø)] CAUSE [BECOME NOT **have**'(prisoner, knife)]

2. [**do**'(man, [**carve**'(man, log)])] CAUSE [BECOME **exist**'(canoe)]

Therefore, the logical structure builder needs more work to make it general enough to represent arbitrary logical structures, or even the more complex examples in VVLP.

Words may be organised into arbitrary tree-structured hierarchies. Any lexicon entry may be specified to have any other as its parent, as long as this does not cause a loop in the tree. Entries with no parent are displayed as roots. As an example, we have developed an ontology based on the one described by Mairal in 2002. It could be very helpful for machine translation to use a common ontology across different languages, as this could help to suggest alternative verbs during translation when an exact match was not found. We hope to hear more from Elizabeth Guest and Brian Nolan on their progress on this front.

One point of note above is the Thematic Relation. Introduced at the request of Winther-Nielsen, this allows to assign more specific *actor* and *undergoer* roles to the various participants. This could be used together with coded attributes on nouns to indicate whether or not they could take various roles with various verbs, for example that a knife is more likely to cut than to be cut, and hence to be the actor rather than the undergoer of the verb *cut*. They can also be used to clarify or document the argument positions in a logical structure where they are not obvious.

The lexicon editor has suffered somewhat from an abundance of features. In Lex, the lexicon includes the following items, which appeared at the time to be of similar substance:

- logical structures for Hebrew verbs
- glosses for Hebrew verbs and other words
- categories from the FLM verbal ontology (Mairal 2004)

There is an overlap between the lexicon and parser rules, which are currently held in separate tables. When the lexicon specifies that a particular word exists in the language, and is a verb, this is equivalent to the following parser rule existing:

VERB → Word ["yyaharəgēhū"]

However, if using morphological analysis as above, it may be preferable to store only verbal nuclei in the database, for example:

VERB → V_{NUC} ["harəgē"]

Neither of these is currently implemented, and the lexicon is only used to look up glosses and logical structures.

Connection of Corpus to Lexicon

Currently, the "lexeme" attribute (from WIVU) of the verb in the clause (as determined by WIVU phrase dependent part of speech) is looked up in the lexicon to find the logical structure. This is clearly specific to the WIVU database.

It should be possible to specify which text to look up in the lexicon by means of an attribute which is passed all the way up the parse tree to the top level. This attribute might originate as the surface attribute of the Word or V_{NUC} below the VERB node, and be copied up the tree by unification. this removes the need for the logical structure resolver to have domain-specific knowledge of RRG.

Evaluation of Logical Structures

This is an area which currently requires domain-specific knowledge of RRG, as described above. The evaluator looks within the displayed clause to find a single Emdros [word] object, whose part of speech is a verb (according to the part_of_speech feature in Emdros, which comes from the WIVU database). If no verb is found, the empty string is used, as clauses with no verb are grammatical in Hebrew and imply equality or shared attributes of the subject and object, like the verb "to be" in English.

The found string (the name of the verb) is looked up in the lexicon (database table) to retrieve its logical structure (LS). The LS may contain any number of *variables*, which are enclosed in angle brackets, for example <*x*> and <*y*> (the names are arbitrary). These variables must be linked to referents in the syntactic structure, in other words to syntactic arguments.

Lex currently uses Emdros objects of type [phrase], with the phrase_type being one of NP, PP or a few other options, as possible arguments. However, since the development of the parser, the arguments could now be extracted from the core, using attributes and unification in parser rules.

The linkage between variables and arguments is specified manually by the user and saved in the Emdros database. If no values have been specified yet, then the subject (coded by WIVU) links by default to the variable *x*, and the direct object to *y*, assuming that the verb is more likely to be active than passive. Ideally that information would come from the parser, by using unification to pass morphological attributes up the parse tree.

Having resolved the values of the variables, we replace them with the (transliterated) text to produce the final, linked logical structure. For example, the general logical structure (1) of the verb *bārā?* (to create) in Hebrew may be linked in a particular case to create structure (2).

1. do'(<x>, [create'(<x>:CREATOR, <y>:CREATION)]) & INGR exist'(<y>)

2. do'(*?ĕlōhîm*, [create'(*?ĕlōhîm*:CREATOR, *?ēt ha- šš̄āmayim wə- ?ēt hā- ?ārec*:CREATION)]) & INGR exist'(*?ēt ha- šš̄āmayim wə- ?ēt hā- ?ārec*)

The values of the arguments are not translated here. In the case of common nouns, they could be looked up from the lexicon as glosses. The layered structure of the noun phrase, particularly conjunctions and adjectives, could be used to generalise further, and avoid the need to have phrases like "*?ēt ha- šš̄āmayim wə- ?ēt hā- ?ārec*" (the heavens and the earth) in the lexicon.

Parsing of Logical Structures

As noted by several authors, the logical structures, if completely implemented, constitute a semantic metalanguage which is independent of the syntax or expression of any human language. The process of converting syntax to semantics is reversible, and therefore can be used for machine translation. The first step is to reverse the generation of the logical structure.

When adverbs are not taken into account, the outer form of the logical structure is entirely determined by the verb. Therefore, in the lexicon for the destination language, we must be able to find a verb with the same logical structure. Here it helps enormously to have consistent logical structures between languages, for example by using the LS builder described above.

Errors in the entry of logical structures, both syntactic errors such as missing brackets and semantic errors such as using the wrong predicate or verb classification, would cause a failure to find an exact match in the target language. In such cases, we may be able to prompt the user with approximate matches, derived from an ontology or by examining the settings entered into the LS builder, that may help them to identify and correct the problem.

In some cases, such as the different classification, and hence meaning, of the verb *to die* in Mandarin and English, an exact match may be impossible (VVLP 97 p.106).

All the information from previous steps, including the mapping from variable names to arguments and the tree structure of the entire clause, is known from the previous steps, which took place as part of the same translation process. Therefore we do not need to rely on actually being able to parse all information from the logical structure itself (as a written string).

The arguments may be decomposed to separate nouns from adjectives which have lexicon entries in the target language, and render them according to the rules of the layered structure of the noun phrase (LSNP) in that language.

If the nouns and adjectives have direct matches or glosses in the lexicon of the target language, these can be used to translate them to native words. Otherwise, if there is enough similarity in the transliteration schemes and they are reversible, then a native phonetic writing of the foreign word is possible, at least as a place holder until a lexicon entry is created. Failing that, only a foreign word can be inserted.

Tøndering points out that Russian has no determiners, and therefore the meaning of many noun phrases is guaranteed to be ambiguous with relation to English, which requires determination. The most obvious solution is to allow rendering noun phrases as indeterminate when no determination was explicitly specified. This may result in some very strange English (1), but does not claim anything that was not in the source language (2) (c.f. Gorbachev):

1. a crisis is caused by a problem with an economic system

2. the crisis is caused by the problem with the economic system

Having completed these steps, we should have a parse tree identical to the original one, but with the nodes below the core, the nucleus and argument nodes, linked to additional information about the choices of translations into the target language.

Rendering Syntax Trees

Before the syntax tree can be written out as text, it may need to be transformed or restructured, to take account of the syntactic differences between languages.

The canonical RRG approach would be to discard the old tree structure entirely and create a new one based on a template. While this works well for intelligent speakers who can see how a template can be adapted to fit their needs, we postulate that mechanically following this route will result in many cases where no suitable template can be found.

As we are using parser rules to generate templates, we may also have the opposite problem. If we are forced to enumerate all possible templates allowed by our rules, we may find that there are an infinite number, or a very large number, and it may take too long to check them all for suitability.

One alternative is to start at the root of the existing syntax tree. If the input parse was ambiguous, then we can use each possible parse tree in turn to generate multiple possible outputs. We start at the root of the tree, comparing the rule used at each level, to generate each node in the parse tree, to the ones in the target language rule set. If there is an exact match, we use it (for now). If not, then we try each possible rule for that node in turn.

For each rule, we try to fill all of its slots using the children of the old node. However, they may not be compatible, due to their type, or due to unification constraints. If we cannot find a place for a child node, we leave it orphaned. If we cannot find a child node to place in a slot, we leave it empty. Then we repeat the process for the child nodes, all the way down to the morphological level of the tree. At this point we proceed to the next filled slot, until all filled slots have been processed.

We may now have a number of orphans or holes left over. We assign as many orphans as possible directly to holes. Then we try adding rules to the left-over holes recursively, stopping and backtracking whenever we have more holes than orphans, or the depth exceeds a fixed limit.

If we still do not succeed in filling all the holes, we can try to strip the top node off each orphan in turn, and try again, until there are no nodes left to strip on any orphans. Stripping the top node may increase the number of orphans, and therefore the number of steps that the hole-filling algorithm can try before giving up.

If the holes in the tree ultimately cannot be filled, then it must be discarded for another alternative. However, it is possible to render a tree with some orphans left over, although it will not convey the full sense of the original input and is therefore not an ideal solution, if one can be found.

This algorithm allows, for example, the moving of argument from the core to the periphery, and vice versa, and the generation or removal of extra nodes required to adapt them to their new position.

We now have to calculate the morphology. As the rules of morphology will vary significantly between languages, there seems to be little point in trying to carry over the structure, so we recursively try every possible combination of morphology rules, provided that each word uses each rule no more than once.

If the transformation and regeneration process was successful, then we should have one or more complete tree from the top node (e.g. Sentence) down to morphemes. If we have multiple possible trees, they can be offered to the user as alternatives, or ranked by preference to disfavour those which discard information from the original, or which use infrequent grammatical constructs in the target language.

Note that this approach has not been tested, and requires further work.

Collaboration Tools

The construction of a lexicon and parser rule set for a single language is a significant undertaking, and even more so when multiple languages are required. Therefore it makes sense to envision the development of the tool as either a massive commercial project or a community collaborative effort.

The author favours community collaborative approaches to science. However the commercial possibilities of an improved machine translation tool should not be ignored, and the copyright ownership of the lexicons and parser rules should be used to appropriately reward their authors when commercial opportunities arise.

Lex was designed as a web-based tool, in order to facilitate remote access and easy maintenance. It is possible to install it on a local computer, for offline use or for security reasons, if desired by the user, but upgrades may be complex and difficult to test remotely.

With the previously mentioned facilities for import and export of data, and sharing of corpora and rules between multiple users with independent databases and access control, Lex as a web service should be a more powerful collaborative tool than any software installed on a user's own computer or local server.

Low Cost Software

It was important for the authors to demonstrate a commitment to openness in science and research, by making the software available free of charge to all. Many academic projects have done the same, both with software, fonts, and the research that they have published. Cost should not be an obstacle in taking part in the project, as the contribution of each participant benefits all.

Open Source Code

In addition to being free of charge, the Lex software is released under an open source license, the GPL. This allows it to be modified by the user, and to be incorporated into other open-source projects. This license also applies to the Emdros database which we use. For those who would like to use the source code in commercial, closed source projects, alternative licensing for Lex and Emdros software can be discussed with the respective authors. This license does not apply to the databases, corpora, lexica and parser rules developed by users of the software, who are free to use them as they wish.

The open source license allows users to contribute to development of Lex, by customising it to meet their own needs and submitting patches for new functionality that can be shared with other users. It also means that Lex is not controlled by any one entity and the right to use and modify it cannot be taken away.

Lex has extensive unit tests for most features, to help ensure reliability across new versions, and these will continue to be developed and improved alongside the code of Lex itself.

Lex uses and depends on other open source software, such as the MySQL database, the Java programming language, and the Apache Tomcat web server. Lex does not require any proprietary software to run.

Summary

We have described the software tool called Lex, its various features that may be of interest to linguists, especially in the RRG community, and our plans for future work based on Lex, including machine translation.