Parser and interlinear text

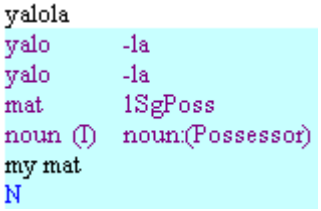
Andy Black, Gary Simons, and Ken Zook

November 3, 2007

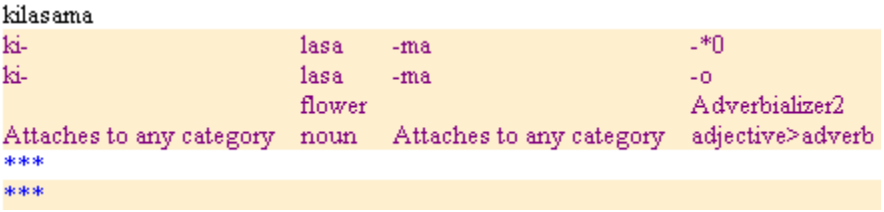
Contents

1 Introduction

When you interlinearize text manually, ,Flex proposes an analysis whenever it can for words based on previous analyses you made for the same word. It uses some statistics to try to find the most likely analysis from your previous work. If the analysis is based on a previous analysis that you approved, the analysis is shown with a light blue background.

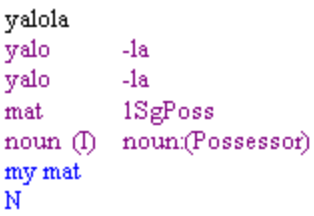


When the parser is run, it goes through the wordform inventory and adds analyses that are currently possible, based on your lexicon and grammar. If there are no user-approved analyses for a wordform, but there is a parser approved analysis, then that is shown with a light orange background.



If there are no analyses on a wordform, Flex displays the wordform with asterisks on all of the following lines:

After a user confirms an analysis by moving the focus box from a word to another word, the approved analysis has a white background.



To move the focus box without confirming the change, use Ctrl+Shift+Arrow or Ctrl+Shift+N.

2 Parser overview[1]

The basic requirements for the parser in Flex are that it should

- be fully integrated into lexicon management and interlinear text analysis so that users do not have to learn one more program
- use an underlying model of morphology that is already familiar to linguists
- be as easy to get started as Shoebox
- result in a human-readable grammar sketch as well as a machine-interpretable parser, and
- reuse existing SIL parsing software where practical.

For a good introduction to the morphology model, in Flex, go to Help…Resources…Introduction to Parsing.

2.1 Integrated approach

The parser is integrated into the Language Explorer component of FieldWorks. It uses information that was entered into the lexicon to make suggested analyses for wordforms when interlinearizing text. Conversely, as new morphemes are encountered in analyzing texts, they are entered into the lexicon. The integration of components is based on the use of a single underlying database. That is, the data used by all tools are stored in a common relational database which is fully normalized.

2.2 A Conceptual-Modeling approach

The second requirement was to use notions and notations already familiar to linguists. By contrast, earlier SIL parsers used arcane notations and notions. To overcome this, we based our model of morphology on the concepts taught in the textbook most commonly used by our training schools.[2] We built those concepts into a conceptual model of the objects in the problem domain, their attributes, and the relationships between them. Using this approach, the task of describing a language is that of creating records in the database to document the instances of the conceptual objects that are discovered in the language. See BlackSimonsFLExPArser.doc for more details.

[1] This information on the parser is a summary of a paper written by Andy Black and Gary Simons. See BlackSimonsFLExParser.doc for the full paper.

[2] Bickford, J. Albert. 1998. *Tools for analyzing the world's languages: Morphology and syntax*. Dallas: Summer Institute of Linguistics. x, 400 pages.

2.3 A Stealth-to-Wealth approach

The third requirement is that the parser should be as easy to get started with as the Shoebox parser. At the same time, users need to be able to do a complete job of modeling all aspects of the morphology of a given language. That is, the idea is to enable a researcher to start quickly and yet be able to finish well. We call this notion, "stealth-to-wealth."

This approach allows linguistically-aware users to begin at their own level and use the tool to help them successively improve their analysis and the resulting model of it. That is, users gradually tell the system what they know about the grammar, receiving as a reward increasingly automated analysis of text.

"Stealth" refers to the parser hiding behind the interlinearizer and lexicon and parsing with no user setup at all. As users manually perform word analysis in the interlinearizer tool, the system creates minimal lexical entries behind the scenes. The parser then uses this information to offer up possible parses on new words encountered in texts. In the early stages, the parser knows just allomorph forms and the only constraint on where they may occur is given by their classification as prefix, root, suffix, and the like. At first this gives satisfying results as the parser is able to propose correct analyses using roots and affixes already encountered in the text. But as more allomorphs are added to the lexicon, the parser begins coming up with many incorrect parses as it finds more and more combinations of allomorph strings that will cover the wordform. At some point, users become frustrated enough by the incorrect parses to want to get rid of them.[3] In the process, a description of the morphology gradually (and stealthily) unfolds.

"Wealth" refers to the end result of a powerful, linguistically satisfying, highly productive system for both processing texts and describing language. It allows users at all levels of linguistic understanding to describe the grammar of a language to the level they are comfortable with. It provides a migration path from simple initial observations up to detailed, linguistically satisfying descriptions that take advantage of all the concepts supported in the conceptual model of morphology.

2.4 A Descriptive approach

The fourth requirement was that users must be able to generate not only the information needed for the parser, but also a grammatical writeup of the description. We do this by autogenerating a grammatical sketch as well as the files needed by the parser. In both cases, the generation is from the modeled knowledge stored in the database.

2.5 An Eclectic approach

After developing the conceptual model for the system, we realized that current SIL parsers could not implement it. But we also realized that we did not have the resources to implement a new parser from scratch. Thus, we set the goal of reusing existing SIL parsing code as much as possible.

[3] We have prototyped help pages on "How do I get rid of incorrect parses?" which will describe how to use the features of the morphology model listed in the preceding section to constrain away incorrect parses.

The best morphological parser was AMPLE--however, it had a sequential right-branching bias. This made it impossible, in general, to correctly infer the word category of a form involving both prefixes and suffixes or to correctly percolate features within the derivation. We also had PC-PATR, a unification-based syntactic parser. It could address AMPLE limitations if we merely treated the sequence of morphemes produced by AMPLE as the tokens to be input to a word grammar for PC-PATR.

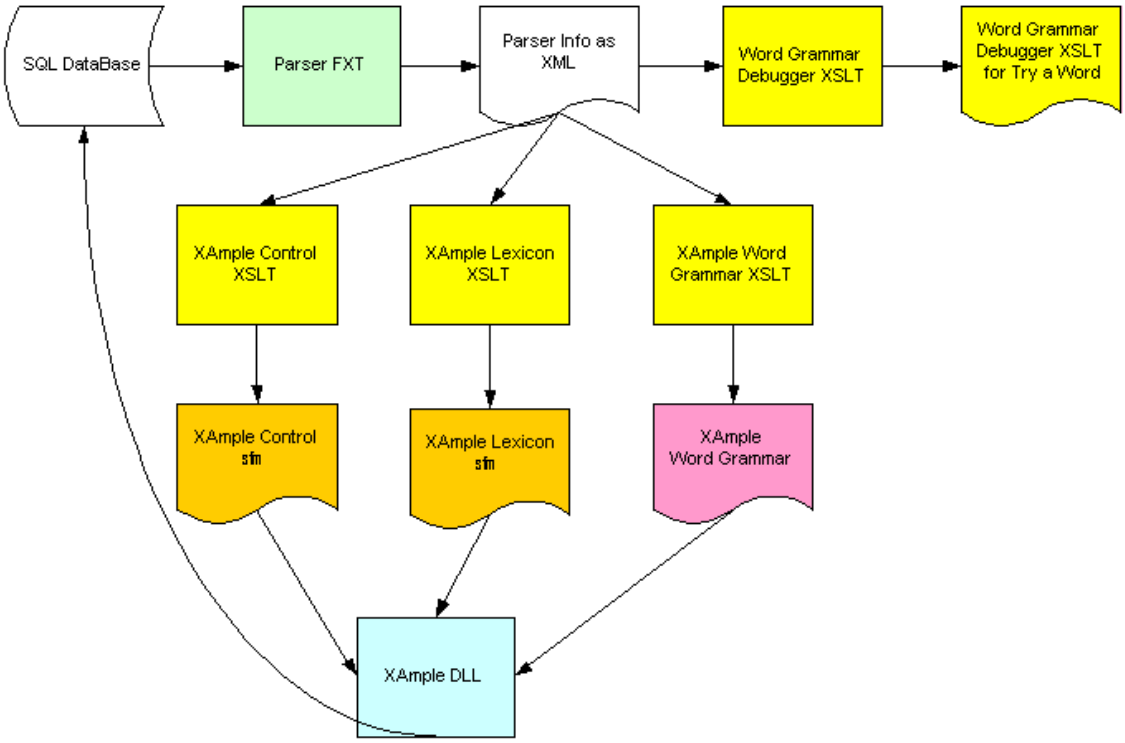
Therefore, we chose to merge our AMPLE morphological parser with PC-PATR to produce an XAMPLE version.[4] When we generate the source files for XAMPLE, in addition to the lexicon and analysis control files traditionally required by AMPLE, we create a word grammar file for the embedded PC-PATR parser. After the AMPLE component breaks the wordform into a possible sequence of allomorphs, this preliminary result is passed to the word grammar. This does the lion's share of the work, including the following:

- Handling inflectional templates
- Controlling inflection classes
- Constraining and percolating inflectional features
- Constraining exception "features"
- Constraining stem compounding
- Constraining derivation
- Constraining clitic attachment
- Allowing for unspecified or underspecified affixes or roots
- Percolating morphosyntactic features
- Allowing derivational affixes to override morphosyntactic features of the stem
- Constraining derivation outside of inflection appropriately
- Constraining derivational and inflectional circumfixes

When the parser runs, it exports the lexicon and grammar to external files, and then calls XAmple to parse a wordform from the wordform inventory. XAmple uses the lexicon and grammar files to parse the word, and returns the analyses to Flex. It then updates statistics on existing analyses or adds new analyses as needed.

This diagram illustrates the flow of information while the parser is running:

[4] The "X" in XAMPLE originally stood for "eXperimental," but now we like to think of it as meaning "eXtended." See http://www.sil.org/computing/catalog/show_software.asp?id=1 and http://www.sil.org/pcpatr/manual/pcpatr.html.



Here is what happens (all files are located in the "Local Settings/Temp" directory):

1. Flex extracts the needed data items from the database and converts them to XML via the Parser FXT file (the result is in *database-name*ParserFxtResult.xml).
2. Flex creates the following files used by XAmple via XSLT transforms:
    1. An XAmple analysis control file (*database-name*adctl.txt)
    2. An XAmple lexicon file (*database-name*lex.txt)
    3. An XAmple word grammar file (*database-name*gram.txt)
    4. Along the way, we also create these files:
        1. *database-name*XAmpleWordGrammarDebugger.xsl: a transform used by the word grammar debugger part of Try a Word
        2. *database-name*gafawsData.xml: an XML file that reflects the structure of the inflectional templates
        3. If is fed into the GAFAWS tool (Grimes Algorithm For Analyzing Words), that produces OUT*database-name*gafawsData.xml: we use this to determine inflectional affix orderclass values in the XAmple lexicon file
3. These three files (plus cd.tab located in Language Explorer\Configuration\Grammar) are fed into the XAmple DLL to initialize it.
4. A wordform is then passed to the XAmple DLL which returns an XML formatted form of the result.
5. This result is given to the parser filer which updates or creates analyses in the wordform.
6. Another thing Flex does for efficiency purposes is to calculate a checksum on the XML returned by the XAmple DLL. This is stored on the wordform. When Flex

parses a wordform, it checks the value of the new checksum against the old checksum. If they are different, it files the parse. If they are the same, it skips the parser filing process to save time.

This diagram summaries the process with the Try a Word process which can show the actual steps the parser takes: