YaLG – extending DCG for NLP

José João Dias de Almeida
Universidade do Minho
Department of Computer Science
Campus de Gualtar
4700 Braga-Portugal
jj@di.uminho.pt

Abstract

Logical grammars are known to be powerful methods to specify syntax and semantics of language.

In a PROLOG environment, DCGs are very popular and known to be simple and easy to learn and teach. When natural language is involved, DCGs are suitable for toy problems but difficult to scale up over big dictionaries, complex morphology or big texts to analyze.

In this paper YaLG [1](yet another logic grammar) is presented that enables the possibility of using DCG like grammars with external lexical analysis for natural language processing (NLP).

YaLG is based on a set of modules that:

- perform morphological analysis with external real size dictionaries and rules
- give the possibility of scanning external files
- give control over non-word elements of text
- enable backtracking over multiple analysis

Some difficulties on the interface PROLOG-C, namely large input, backtracking over the input and multi-analysis are discussed.

1 Introduction

Design a Natural Language Processor is a hard problem because, even for very simple tasks the behavior of the processor depends heavily on the existence of a dictionary. Furthermore, the dictionary organization is a crucial point in any NLP application (dictionary modules must return the right attributes in the right format).

Generally speaking, even when some tools are available, it is not a trivial task to adjust them to cooperate and to adapt them to new environments.

In this paper we describe YaLG designed to extend DCG[8] with a powerful lexical level in order to have:

- external control and definition of file scanning details,
- external control over word morphological analysis with external dictionaries and rules

[1] the original name NYAGNLPLG(not yet another good natural language processing logic grammar) was reject because of pronunciation problem

- non-word elements processing,
- undefined words processing,
- complex attributes,

The scanning and the morphological parts are done with NLlex tool generator[4] which is meant to adapt the scanner, the morphological-analyzer and the parser to particular problems and corpora conventions. In this way we expect to improve reusability.

The main difficulties found with this approach are:

- interface between NLlex and PROLOG
  note: this items are discussed in more detail in section 3-Prolog-C interface.
  - backtracking over input and multi-analysis
  - complex attribute returning
  - file/string analysis commutations
  - necessity of processing very long files which is incompatible with DCG list approach
- lexical level:
  - to be able to build linguistics resources (dictionaries, morphological rules, and morphological processors) in a reusable way

In section 2 the system will be described. Section 3 presents the PROLOG-C interface. In section 4 the lexical-analyzer modules will be described in detail and in section 5 conclusions and future work will be drawn.

In this text some knowledge about logic grammars [1], PROLOG[7], DCG[8] and LEX[?] will be assumed.

2 Overview of YaLG

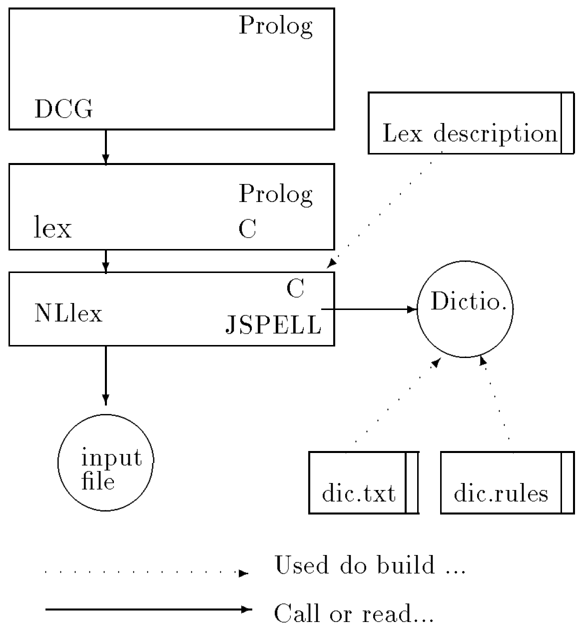
2.1 System Architecture

The implementations were carried out in Sicstus PROLOG[6] but are easily portable to other PROLOG with a good C interface.

The main modules present in the system are:

- the user definable PROLOG + YaLG DCG code
- the lexical interface (with parts in C and parts in PROLOG)
- the NLlex scanner and morphological analyzer (written in C) part of this module is generated by the NLlex tool based on the "lex description" provided by the user.

  This module imports a dictionary containing and is able to scan input file for new symbols.



2.2 System at work

In order to have a working NLP YaLG program the user :

- must have (or write) a dictionary (classified word list(dic.txt) + morphological rules(dic.rules)). This resource can be reusable for (from) many other applications
- must write a (reusable) view of the dictionary and attributes in order to feat is grammar (used to generate the NLlex module)
- must write a grammar

The main differences between YaLG [2] and DCG will be seen at lexical level and at input representation level.

As an extension to the terminal symbols of DCG
`[terminalSymbol]`
YaLG offers a new syntactical element (correspondent to DCG's pseudo-terminals)
`[Cat-Sem]`
meaning any word from category Cat and semantics Sem.

The DCG example:

```
s(...)  --> np(...), vp(...).
vp(...) --> v(S,iv,Agr).
vp(...) --> v(S,tv,Agr), np(...).
vp(...) --> v(S,be,Agr),....
v(chase,tv,present(3,s)) --> [chases].
...
?- s(...,[the,dog,chases,the cat],[]).
```

[2] YaLG are based on GPC grammars[2] which are DCG extended with higher-order grammar specification environment with automatic semantic rewriting. For the purpose of this paper none of this properties are relevant.

would be replaced for:

```
s(...)  ---> np(...), vp(...).
vp(...) ---> [v(iv,Agr)-S].
vp(...) ---> [v(tv,Agr)-S], np(...).
vp(...) ---> [v(be,Agr)-S],....
...
?- set_file(text1).
?- s(...,0,Y).
```

Obviously the words (radicals) must exist in the dictionary. The lex-description file should adapt the attribute notations:

```
 1 %initnlex english -p personalDic
 2 %feat CAT G N T Tr
 3 ...
 4 %word
 5 {*CAT=v,ROOT=be*}  {
 6    sprintf(cat,"v(be,%s(%s,%s)).",T,P,N);
 7    nlnext;}
 8 {*CAT=v*}    {
 9    sprintf(sem,"%s.",ROOT)
10    sprintf(cat,"v(%s,%s(%s,%s)).",Tr,T,P,N);
11    nlnext;}
12 {**}         {
13    sprintf(sem,"%s.",ROOT)
14    sprintf(cat,"'%s'.",CAT);
15    nlnext;}
16 <<EOW>>              { return yytext ;}
17 ...
```

Some comments about the example (see details in section 4):
line 1 - declares the dictionaries to be used
line 2 - defines the relevant features
line 5..7 - yields treatment of verb to be
line 9..11 - declare general verbs
line 9 - indicates that the semantic of the verb comes from the radical
line 10 - builds the category term
line 11 - save this analysis
line 12..15 - by default semantic comes from the radical and cat comes from the CAT feature.

3 Prolog-C interface

This section is concerned with problems of the interface between C (NLlex) and PROLOG. We will begin with the description of the module interface and then focus on internal interface problems and decisions.

In the development of this interface the main concern was focus on the expressive power and not (yet) in the efficiency.

3.1 Low level module interface

The module NLlex exports the following relevant functions:

```
lex(+Word,-Category,-Semantic)
analyzes a given word

lex2(-Word,-Category,-Semantic)
read and analyzes word from current
string or file

set_string(+String)
sets current input to be String

set_file(+Filename)
sets current input to be this file
```

3.2 Backtracking over multi-analysis

It is crucial to have access to all the solutions returned by the lexical-analyzer. The most natural way of doing this in the PROLOG side is to get one at the time and to be able of get a new solution by backtracking.

The PROLOG notion of backtracking is unknown in the C environment.

It is possible to establish a maximum for the number of analysis of each word, without cutting system coverance.

The current version evaluates a stack of solutions (in C)each time a new word is analyzed (function init_lex). Internally backtracking triggers a pop on the solution stack (function get_lex). The glue is done with a PROLOG predicate (lex or lex2).

3.3 Input representation and related backtracking

In CDG traditional implementation all nonterminal symbols(NT) are translate in a PROLOG predicate with two more attributes: list at input and list at the output.

When we are processing a file, one natural solution would be to read the file to a list and apply the NT predicate to it. However this strategy is incompatible with medium/large files processing.

Furthermore, the input related backtracking can be almost infinite in theory. Fortunately it is possible to write NL grammars which do some commitment (for example in the end of a paragraph). This way we can parse input with limited backtracking length. Warning: It is crucial to the performance the way the grammar is written (this problem is common to PROLOG in general).

In the current version:

- a circular finite list of K elements is used for input buffering (K is a parameter that can be defined by the user).
- this list is kept on the internal database and K elements are available for backtracking
- this circular list is build on demand by the nllex scanner, from external file or string
- list references are done by an integer. Difference lists correspond to a pair of integer.

The right choice of a K value proved to be crucial in system performance. For example in a text of 2500 words (like this paper) if all the text is buffered (k=2500) the time is 3 times longer then if just the last 30 works were kept (k=30).

3.4 Syntactic and semantic attributes

The result of the lexical analysis is return as a pair of Semantic and Syntax term. This terms can be complex in term-depth size and in term composition.

For example the syntactic category of the Portuguese word "gosta" (loves) could be

`v(prep(de),present(3,s))`

in order to say that we have a present tense, third person of the singular of a verb asking for the preposition "de".

The problem is to define an easy way of building complex PROLOG terms from a C environment. The present solution is to build a string (typically using sprintf C command) and translate it to a PROLOG term using `read_from_chars` function using `chario` module.

4 Lexical Analysis

In a simple way we can say that NLlex is a composition of LEX with a morphological analyzer (JSPELL).

4.1 NLlex

In this section we'll give a brief introduction to NLlex. A more detailed presentation can de found in [4].

The main goal of NLlex is to be able to control:

- actions to be done when we find words with certain morphological properties. This properties are defined by attribute-value pairs.
- the non textual elements of text like :
  - keywords
  - special symbols
  - markup elements
  - date, sport results, numbers with units, ...
- heuristics for dealing with undefined words with rules using:
  - capitalization features
  - unofficial application of morphological rules to words
  - context

In the design care was taken not to build a new syntax but trying to follow (whenever possible) the existing lex (flex) syntax.

4.2 Description

In the present version (NLlex0.4) flex code is generated and morphological analysis is done by jspell (the replacement by any other morphological analyzer that returns attribute values pairs strait-forward).

Structure of NLlex input

```
/*------------ general definitions */
[optional C code (as in lex)]
%initnlex <dicname>
%features <feat-list>

/*------------regul. expre-actions */
%%
<RE>        {C code}
...

/*-----------------word processing */
%word
<word RE+>  {C code}
...
<<EOW>>     {C code}

/*----------undef. word processing */
%undef
<word RE+>  {C code}
...
<<EOW>>     {C code}
%%
[C-code]
```

NLlex syntax extends the lex syntax in

- `%initnlex options` – to define the dictionary and correspondent morphological option,
- `%features feat-list` – to define list of features to be used; each attribute-value will be available in the word conditions and in the word actions,
- `%word` new zone – to define conditions over words and actions to be done when they are on. A condition can use a limited constraint expression over:
  - attributes declared in `%feature`. Example:`CAT=prep`
  - the root of the word. Example: `ROOT=ser`
  - the letter-case type. Example: `UC` (the word is uppercase)
  - the word in itself. Example: `nltext=jj`
- `<<EOW>>` – (optional) action to be done after analyzing every classification of a word
- `%undef` new zone – (optional) to define conditions over undefined words (guessing is performed with the morphological rules, and actions are associated with the different guesses)
- `{P}{S}{W}` – predefined regular expressions for paragraph, white space and words (their values can be used in the lex RE),
- features – the names of the features can also be used in actions as features string values. The same is valid for ROOT, LETT (letter case type) and nltext (the main word).

Jspell

The morphological analysis is done by Jspell[3]. Jspell was designed using many code of ispell(!!) and is based on an external dictionary and affix rule files.

Each dictionary entry has a root, classification and morphological rules names (flags). The affix file contains codification character details, suffix rules and prefix rules. Each rule may redefine the word classifications.

Special mechanism is provided to deal with irregular verbs and words in order to store their "roots" and features correctly.

Jspell can be used as a command (to find non existent words), as a pipe (receiving "questions" dealing with words to analyze and returning words in a predefined format), as an interactive spell checker(!) or as a C library (which is the mode used in this system).

As a library, this morphological analyzer and guesser has, for instance, functions to:

- get words from a buffer
- analyze morphologically a word with possibility of calculation of near-misses and words derived from unofficial application of rules
- inserting words in a personal dictionary
- saving the personal dictionary
- changing a word in a buffer

As the result of word analysis, 0 or more values can be obtained. Each value has:

- root of the word
- set of attribute-value pairs containing information derived from the dictionary and affix-file.

5 Conclusions and future work

YaLG shows that it is possible to share linguistic resources and code even between rather different environments.

The present version is using a Portuguese dictionary (built for a spell checker!) with around 40,000 radicals.

We did not provide any results about the efficiency of the system. Performance evaluation and further improvement will be given in a future paper,

however experiments showed that grammars for specific details, are easy to write and useful for various information extraction and processing.

Expanding the system to incorporate the nllex api with word insertion/substitution is under development.

References

[1] Harvey Abramson and Veronica Dahl. *Logic Grammars*. Symbolic Computation - Artificial Intelligence. Springer-Verlag, 1989.

[2] J. Joao Almeida. Gpc – a tool for higher-order grammar specification. In Carlos Martin Vide, editor, *Actas del X Congreso de Lenguajes Naturales e Leanguajes Formales,Sevilla*, 1994.

[3] J. Joao Almeida and Ulisses Pinto. Jspell – um módulo para análise léxica genérica de linguagem natural. In *Actas do Congresso da Associação Portuguesa de Linguística, Évora*, 1994.

[4] J.J. Almeida. Nllex – a tool to generate lexical analysers for natural language. Technical Report UM-DI-95.04, Universidade do Minho, Departamento de Informatica, 1995.

[5] Nicoletta Calzolari. Computational lexicons. reader of 5.LLI, Faculdade de Letras, Universidade de Lisboa, 1993.

[6] M.Carlsoon, J. Widen, J. Andersson, A. Andersson, K. Boortz, H. Nilsson, and T. Sjoland. Sicstus prolog user'a manual. Technical Report T91:11B, SICS, 1991.

[7] R.A. O'Keefe. *The Craft of Prolog*. Logic Programming. The MIT Press, 1990.

[8] F. C. N. Pereira and D. H. D. Warren. Definite clause grammars for language analysis: A survey of the formalism and comparison with augmented transistion networks. *Artificial Intelligence*, 13(3):231–278, 1980.