NLlex – a tool to generate lexical analyzers for natural language

José João Dias de Almeida
Department of Computer Science
University of Minho
Campus de Gualtar
4710 Braga-Portugal
jj@di.uminho.pt

Abstract

In this paper we present a natural language lexical analysis program generator (NLlex) that looks like Unix lex extended with morphological analysis and other Natural Language (NL) elements.

NLlex generates a C program which is linked with a morphological analyzer and with other modules, in order to produce a NL processor.

As a particular case, NLlex can generate modules to work:

- as a lexico-morphological analyzer (to be called from yacc, NLyacc, btyacc or any modules that need it)
- as a simple lexical processor tool

NLlex can also deal with ,and be tuned to, the so frequently seen non textual elements (markup elements, LaTeX like things, dates, quotes, ...)

An interface between NLlex and Prolog have been developed and a Perl interface is under development.

1 Introduction

Our main goal is to build a modular reusable set of procedures for NL analysis.

Systems for performing the complex tasks of morphological and lexical analysis are at the core of all NL applications.

In this paper we present a NLlex tool that can be seen as a Unix lex for NL processing (NLP).

The design of a Natural Language Processor is hard because:

- it depends on the existence of a dictionary (building a dictionary takes a long time)
- the dictionary must have "the right structure" (must return the right attributes in the right format)

- in general, if we have some tools, it is not so easy to teach them to speak with each other. Generally, tools are not easily adaptable.

NLlex is meant to adapt the scanner, the morphological-analyzer and the parser to particular problems and corpora conventions. This way we expect easier reusability.

In order to deal with formal languages, operating systems like Unix offer a number of tools based on Regular Expressions (RE).

When we are processing NL it is natural to extend RE (RE+) in order to include some word details like:

- features (Cat, Gender, ...)
- uppercase/lowercase
- defined on the dictionary (or not)

and many other things.

After that it would be nice to have all the (Unix) tools tuned to those RE+.

NLlex has primitives to deal with lexical ambiguity and undefined words.

An NLlex module consists on:

- a header with dictionary definitions, feature definitions, ...
- a block of pairs of RE+ and C actions to be done when RE+ is found, including undefined words strategies. In the C actions associated with pairs containing words, the correspondent attribute values are available.
- other C code.

In this text we will assume that the reader is familiar with the lex notation and concepts.

In section 2 we will show the design goals and the NLlex description. A small note about the morphological analyzer (jspell) used by NLlex is presented in order to give a better picture of the tool.

In section 3 some examples are presented; the first one is a stand-alone use of NLlex, a second one shows the connection with a NLyacc parser and the third shows its use with Prolog.

2 NLlex = lex + morphological analysis

2.1 Main goals of design

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

- actions to be done when we find words with certain morphological properties. These properties are defined by attribute-value pairs.
- actions to be done with the non textual elements of text like :
  - keywords

- special symbols
- markup elements
- date, sport results, numbers with units, ...

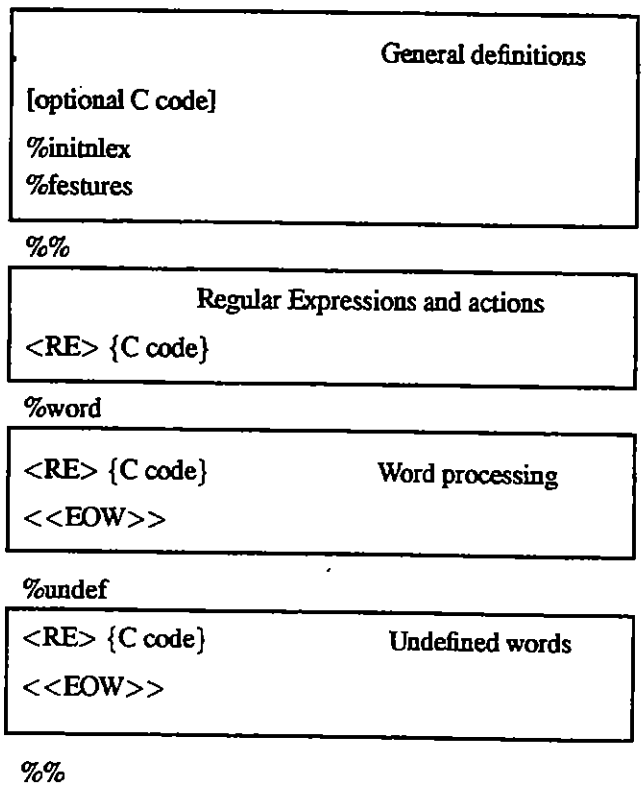
- heuristics for dealing with undefined words with rules using:
  - capitalization features
  - unofficial application of morphological rules to words
  - context

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

2.2 Description

In the present version (NLlex0.7) flex code is generated and morphological analysis is done by jspell (the replacement by any other morphological analyzer that returns attribute values pairs is easy).

Structure of NLlex input



NLlex syntax extends the lex syntax in:

- `%initnlex options` - to define the dictionary and corresponding 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 these are true. A condition can use a limited constraint expression over:
  - attributes declared in `%feature` (`CAT=prep`)
  - the root of the word (`ROOT=ser`)
  - the letter-case type (`UC` (the word is uppercase))
  - the word in itself (`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 (example 1))
- 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**[2]. **Jspell** was implemented reusing many lines of the **ispell** spell checker and is based on an external dictionary and affix rule files.

Each dictionary entry maps a word to its classification and to the set of morphological rules that can be applied. Each rule is identified by a flag. The affix file contains codification character details, suffix and prefix rule definitions. Each rule may redefine the word classification features.

A 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 interactive spell checker or as a C library (this mode is used here).

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, zero 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.

3 Examples

3.1 Independent use

In this section, a NLlex program is presented that prints the number of the paragraph and the form of every occurrences of the verb to write ("escrever" in Portuguese).

```
1   int p=1;
2   %initnlex port                    /* select Portuguese dictionary */
3   %%
4   {P}                 {p++;}        /* increment paragraph counter */
5   %word
6   {*ROOT=escrever*}   {printf("%d-%s\n",p,nltext);}
7   %%
```

When the generated program is applied to a text with 2 occurrence of "escrever" in paragraph 12 and 34 the following output would be produced:

```
12-escrevi
34-escrevendo
```

If line 6 was:

```
6   {*ROOT=escrever,N=s,P=3*} { ... }
```

just the verbal forms in the third person of the singular of any tense would be selected.

A more complex example is discussed in Appendix A.

3.2 NLlex used as lexical analyzer

When NLlex works as a lexical analyzer the tasks to be executed are similar to the traditional use of lex when it works with yacc: when a symbol is found, its type and attributes are returned. Therefore NLlex rules look like:

```
NLlex-Pattern        { return ...}
```

When lexical ambiguity is possible, the function `yyset` is used in order to return multiple values (see Appendix B).

3.3 Interface to Prolog

NLlex generated code can be connected to Prolog. This connection was developed with Sicstus Prolog but can also be easily adapted to other Prolog systems. In the current version there is a Prolog module "nllex.pl" that offers (between other predicates):

- `lex(+word,-cat,-sem)` to get category and semantics from word with backtracking capabilities to get all the word analyses
- `set_file(+string)` to set current input to be scanned by `lex2` predicate
- `set_string(+string)` to set current input to be scanned by `lex2` predicate
- `lex2(-word,-cat,-sem)` to get next word, category, and semantics

Predicate `lex/3` calls the function generated by NLlex in order to get a list of analysis that will be returned one at a time. Each analysis has a semantic part (traditionally a function of the word radical) and a category part (traditionally a term agglutinating the relevant features). NLlex just builds the `sem` and `cat` term strings stack and nllex.pl builds the correspondent terms. The module nllex.pl also takes care of backtraking details using Prolog's internal database.

A more detailed explanation of this module and DCG variant to deal with NLlex is available and it is called YaLG [1].

4 Conclusions and future work

Every NLP application needs a morphological/lexical analyzer. For corpus based NLP, dictionaries, and morphological rules are "heavy" modules.

Different tools often need different views of the words and specific processing/translation of some details.

Different corpora use different formats, markup languages, and non-text element processing.

NLlex has shown to be a useful tool to adapt morphological analyzers to taggers, NLyacc parsers, yacc parsers, prolog-DCGs, and others.

Work has began in extending NLlex to deal with probabilities and some automatic tag resolution.

References

[1] J. Joao Almeida. Yalg - extending dcg for natural language processing. In Carlos Martin Vide, editor, *Actas del XI Congreso de Lenguajes Naturales e Leanguajes Formales,Sevilla*, 1995.

[2] J.João Almeida and Ulisses Pinto. Manual de utilizador do JSpell. Manual, Universidade do Minho, Julho 1994.

[3] Ted Briscoe and John Carroll. Generalised probabilistic lr parsing of natural language (corpora) with unification-based grammars. Technical Report Number 224, University of Cambridge Computer Laboratory, 1991.

[4] Jean-Pierre Chanod and Pasi Tapanainen. Tagging french - comparing a statistical and a constraint-based method. In *EACL-95*, 1995.

[5] Hiroaki Saito MasayukiÎshii, KazuhisaÕhta. An efficient parser generator for natural language. *COLING*, 1994.

[6] M. Tomita. An efficient context-free parsing algorithm. *Computational Linguistics*, 13:31–46, 1987.

[7] M. Tomita, editor. *Current Issues in Parsing Technology*. Kluwer Academic Publishers, Norwell, MA, 1991.

A A Naive tagger

In this appendix, a naive tagger is meant to show how adaptation of classes and non textual elements can be processed in NLlex. In a real tagger, NLlex would work as scanner, morphological analyzer and guesser and could be linked with some statistical or constraint-based disambiguation module [4].

```
%initnlex port
%feat CAT G N TR T P /* features from dict */

%%
[0-9]+                {printf("(number)%s ",yytext);}
[A-Z]\.               {printf("(abrev %s) ",yytext);} /* a name abrevi.*/
\$[^$]\$              {printf("(math)<%s> ", yytext);}/* math in LaTeX */
{P}                   {printf("(parag)");}            /* a paragraph   */

%word
{*CAT=n,G=m*}         {printf("(nm)");}
{*CAT=n,G=f*}         {printf("(nf)"); }
{*CAT=v,T=i*}         {;}          /* ignore imperative tense analysis */
{*CAT=v,ROOT=ser*}    {printf("(ser)");}  /* special verb (ser=to be)  */
{**}                  {printf("(%s)",CAT);}      /* by default tag=CAT */

<<EOW>>               {printf("%s ",nltext);}

%undef  /* guesser part */

{*CAT=v*}             {printf("(v?-%s)",ROOT);}
{*CAT=n*}             {printf("(n?-%s)",ROOT);}
{*UC*}                {printf("(pn?)");}     /* Uppercase -> prop.noun? */
{*AUC*}               {printf("(abrev?)");}  /* all uppercase -> abrev? */
{**}                  {printf("(%s-?)",LETT);}
```

```
<<EOW>>              {printf("%s ",nltext);}
%%
```

B ELR parser using NLyacc and NLlex

Extended LR(ELR) parsing [6, 7, 3, 5] was designed to deal in a deterministic way with ambiguity of grammars and lexica.

NLyacc[5] is a parser generator with a syntax that is a superset of yacc implementing ELR

In this example a simple NL parser is shown. Lexical analysis is done by a module generated from NLlex and parsing is done by a module generated by NLyacc. Lexical ambiguities are processed by NLlex by multiple calls to `yyset` in order to return multiple values.

In this example, the NLlex has some extra tasks:

- to translate the features' patterns over the morphological analysis to the right terminal symbol class
- to deal with some non text elements
- to deal with lexical ambiguity

Parsing ambiguities produces multi-parsing trees. To keep it simple, no code was written to process attributes (other then UNDEF-str) and no control actions were inserted in NLyacc. Control actions would enable the pruning of situations that would not fit the agreement constraints.

```
---(gram.nl)-----------------------------------------------------
%{
#include <string.h>
#define yyset(x) yysetvalue(x) /* NLyacc way of process. multiple  */
                               /* lexical values                  */

extern YYSEMTYPE yysval;
extern YYSTYPE yylval;
%}
%initnlex port

%feat CAT G N ...
%%
[0-9]+  {yyset(INTE); return 0; }
[A-Z]\. {yyset(ABR);  return 0; }
...
{P}     {yyset(EOF); return 0; }     /* paragraph returns EOF      */
\"{W}\" {yyset(PN);  return 0; }     /* a quoted word is like a PN!*/

%word
{*CAT=n*}              { yyset(NOUN);}
{*CAT=v,T=inf*}        { yyset(INF);}  /* verb in the infinitive    */
```

```
|*CAT=v,T=ppa*}        { yyset(PPA);}  /* verb in the past part.     */
|*CAT=v,ROOT=ter*}    { yyset(TER);}  /* auxiliar verb (TER=HAVE)  */
|*CAT=v*}             { yyset(VERB);} /* a verb in the other cases */
...
|*CAT=a_nc*}          { yyset(NOUN);  /* this type of CAT returns 2 */
                       yyset(ADJ);}  /* values!!                   */
...
<<EOW>>              { return 0 ;}
%undef
|**}                 { fprintf(stderr,"(Undefined %s) \n", nltext);
                        yylval.str=strdup(nltext);
                        yyset(UNDEF);}
<<EOW>>              {  return 0 ;}
%%
```

```
---(gram.y )-----------------------------------------------------
typedef  union {char * str;} t  ;
#define YYSTYPE t

%type <str> UNDEF

%token NOUN VERB DET PUNCT UNDEF PPA ADJ INTE ABR PN TER INF
%%
SS : S PUNCT  { puts("S "); }
   | NP PUNCT { puts("NP ");} ;

S : NP VP ;

NP : NP1 | DET op_inte NP1 ;

op_inte : | INTE;

NP1 : NOUN  | c_PN | ADJ NP1 | INF oNP ;

c_PN : ABR c_PN
     | PN c_PN
     | UNDEF {printf("<<%s/CAT=pn>>/n", $1);} c_PN
     | PN ;

VP : VERB oNP | TER PPA oNP ;

oNP : | NP ;

%%
#include "gram.c"   /* generated by NLlex */
main()
{   yyinitialize();
    yyparse();
    yyterminate(); }
```