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
4700 Braga-Portugal
jj@di.uminho.pt

July 11, 1995

Abstract

In this article 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 that is linked with a morphological analyzer and with whatever we want, 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, LaTeXlike things, dates, quotes, ...)

An auxiliary tool that adapts NLlex generated code to be called from Prolog is under development and has shown to be a valid tool to build Prolog NL processing programs.

Keys: natural language understanding, natural language processing

1 Introduction

The design of a Natural Language Processor is hard because:

- it depends heavily on the existence of a dictionary
- That dictionary must have "the right structure" (must return the right attributes in the right format)
- generally speaking, even if you had 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+.

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

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

An NLlex program 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 and yacc notation and concepts.

In section 2 is done the presentation of 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 the 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. 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.

2.2 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 is easy).

Structure of NLlex input

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

%%     /*----------------zone of reg. exp - actions as in lex */
<RE>        {C code}
...
%word  /*---------------------------- zone of word processing */
<word RE+>  {C code}
...
<<EOW>>     {C code}
%undef /*------------------- zone of undefine word processing */
<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(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),
- 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[1]. Jspell was designed using many code of ispell(!!) and is based on a external dictionary and affix rule files.

Each dictionary entry has root, classification and morphological rules names (flags). The affix file contains codification character details, suffix rules and prefix rules. Each rule 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 (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, 0 or more values can be obtain. 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 Example 1 - independent use

In this section, an 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 [3].

```
%{
int pr=1;
%}
%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=pn,LC*}       { printf("(pn-l)");}  /* proper name in lowercase */
{*CAT=pn*}          { printf("(pn)");}
{*CAT=prep*}        { printf("(p)");}
{*CAT=v,T=i*}       { ;}          /* ignore imperative tense analysis */
{*CAT=v,ROOT=ser*}  { printf("(ser)");}  /* special verb (ser=to be)  */
{**}                { printf("[%s-%s]",LETT,CAT);}

<<EOW>>             { if(pr){printf("%s ",nltext);}else{pr=1;}}

%undef  /* guesser part */

{*CAT=v*}           { printf("(v?-%s)",ROOT);}
{*CAT=n*}           { printf("(n?-%s)",ROOT);}
{*CAT=adv*}         { printf("(adv?-%s)",ROOT);}
{*nltext=jj*}       { printf("(jj!)");pr=0;}
{*UC*}              { printf("(pn?)");}     /* Uppercase -> prop.noun? */
```

```
{*AUC*}              { printf("(abrev?)");}  /* all uppercase -> abrev? */
{**}                 { printf("(%s-?)",LETT);}

<<EOW>>              { if(pr){printf("%s ",nltext);}else{pr=1;}}
%%
```

3.2 Building a ELR parser - using NLyacc and NLlex

Extended LR(ELR) parsing [5, 6, 2, 4] was designed to deal in a determinisc way with ambiguity of grammars and lexica.

NLyacc is a parser generator with a syntax that is a superset of yacc but implementing ELR.

3.2.1 Example

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 invocation of yyset - to return multiple values.

Parsing ambiguities produce multi-parsing trees. In this case no code was written to process attributes (other then UNDEF-str) to make the code easy to read. In order to make the example easy to read, no control actions were inserted in NLyacc. Control actions would enable the pruning of situations that would not fit the agreement constrains.

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

extern YYSEMTYPE yysval;
extern YYSTYPE yylval;
%}

%initnlex port

%feat CAT G N TR P T SEM
%%
[0-9]+  {yyset(INTE); return 0; }
[A-Z]\. {yyset(ABR);  return 0; }
...
<<EOF>> {yyset(EOF); 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 classe returns 2 */
                        yyset(ADJ);}  /* values!!                      */
...
<<EOW>>               { return 0 ;}
%undef
{**}                  { fprintf(stderr,"(Undefined %s) \n", yytext);
                         yylval.str=strdup(yytext);
                         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 | PIND | 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(); }
```

3.3 Example 3 - interface to Prolog

In this example the connection of 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 file "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_string(+string)` to set current string 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 return 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.

A DCG variant to deal with this nllex.pl is available.

3.3.1 NLlex example to be loaded

```
%{
#include <string.h>
#define s1 s->sem[s->n_sol]
#define c1 s->cat[s->n_sol]
#define nlnext (s->n_sol++)

#define YY_DECL yylex(State *s)
%}

%initnlex port

%feat     CAT G N TR P T SEM I Prep Art FSEM
%%
%word
{*CAT=nc*}           { sprintf(s1,"%s.",ROOT);
                       sprintf(c1,"nc(%s).",G);
                       nlnext;}
{*CAT=v,ROOT=ser*}   { sprintf(s1,"%s.",ROOT);
                       sprintf(c1,"v(ser,%s).",T);
                       nlnext;}
{**}                 { sprintf(s1,"%s.",ROOT);
                       sprintf(c1,"%s.",CAT);
                       nlnext;}
<<EOW>>              { printf(".\n"); return 0 ;}
```

```
%undef

{**}                    { sprintf(s1,"undef(%s).",nllex);
                          sprintf(c1,"X.");
                          nlnext;}
<<EOW>>                 { printf(".\n"); return 0 ;}

%%
```

3.3.2 Prolog part (allways the same)

Obviously this file is not to be seen by the user.

```
:- module(nllex,[lex/3,lex2/3,set_string/1]).
:- use_module(charsio).

%lex(+Palavra,-Categoria,-Semantica)
lex(Palavra,Categoria,Semantica) :-
        set_string(Palavra),
        init_lex(Ref),
        repeat,
        get_lex(Ref,Cat,Sem,Flag),
        (Flag=1 -> (
            name(Sem,Seml),
            read_from_chars(Seml,Semantica),
            name(Cat,Catl),
            read_from_chars(Catl,Categoria))
         ; !, fail).

foreign_file('nllex.o',[init_lex,get_lex,set_string]).

foreign(set_string,c,set_string(+string)).
foreign(init_lex,c,init_lex(-address('State'))).
foreign(get_lex,c,get_lex(+address('State'),-string,-string,[-integer])).

:-load_foreign_files('nllex.o',['nllex.a']).
```

4 Conclusions and future work

Any NLP work needs a morphological/lexical analyser. For corpus based NLP, dictionaries, morphological rules are "heavy" modules.

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

NLlex has shown to be a useful tool to adapte morphological analizers 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. A better interface to Prolog and Perl is under development.

References

[1] 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.

[2] 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.

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

[4] Hiroaki Saito Masayuki Ishii, Kazuhisa Ohta. An efficient parser generator for natural language. *COLING*, 1994.

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

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