Dealing with Free Order and Non-Language Markers in a Top-Down-Left-First Algorithm

Analiza szyku swobodnego i znaczników pozajęzykowych w algorytmie zstępującym z lewej do prawej

Krzysztof Jassem

Adam Mickiewicz University
Faculty of Mathematics and Computer Science
ul. Matejki 48-49
60-769 Poznań
e-mail: jassem@amu.edu.pl

ABSTRACT

The first part of the paper presents an algorithm of parsing a Polish elementary sentence with a Top-Down-Left-First algorithm. The algorithm aims at optimising the efficiency of parsing free-order sentences by analysing some phrases in a deterministic way. An improvement in relation to the algorithm presented in [3] consists in the availability of parsing verbal complements occurring outside the verbal phrase. The second part of the paper suggests a formalism for describing re-write rules for phrases which can contain non-language markers. An algorithm for compiling such rules into Horn clauses interpretable by the Prolog engine is sketched.

STRESZCZENIE

W pierwszej części pracy zaprezentowano algorytm parsowania polskiego zdania elementarnego przy pomocy algorytmu zstępującego z lewej do prawej. Algorytm ma na celu optymalizację złożoności obliczeniowej parsowania zdań z szykiem swobodnym poprzez analizę niektórych fraz w sposób deterministyczny. Postęp w stosunku do algorytmu prezentowanego w pracy [3] polega na możliwości parsowania dopełnień czasowników, które występują poza frazą czasownikową. W drugiej części pracy zaproponowano formalizm opisu reguł zastępowania dla fraz, które mogą zawierać znaczniki pozajęzykowe. Naszkicowano algorytm, który kompiluje takie reguły do klauzul Horna, które mogą być interpretowane przez maszynę prologową.

1. Free order of Polish sentences

In this section we shall discuss the problem of parsing simple Polish sentences with free order of components. We shall focus on the classical

structure of an elementary sentence: s -> np, vp, where both phrases may be completed by modifiers or complements, occurring in various orders, not necessarily inside the phrase.

Beneath a review of existing solutions for Polish is given (for the sake of readability the cited rules are simplified). The comparison is made on the assumption that a Top-Down-Left-First (TDLF) algorithm (used by the Prolog engine) is applied to each description.

1.1. S. Szpakowicz, (1986) [1]

An elementary sentence is described with the rule:

s -> np, vp

(in this paper the conversed order of components is not discussed).

The verbal phrase consists of a verb and a set of (at most three) verb requirements – these elements are set in an arbitrary order:

vp -> {verb, req1, req2, req3}

(braces denote all possible permutations of elements).

Characteristics of the approach

Number of possible non-subject complements

The number is equal to three which seems to be sufficient for most NLP applications.

Order of complements

There are no limitations to the free order of complements, which is ill-advised from the point of view of parsing efficiency.

Parts of speech which may be complemented

The description allows complements (called requirements) only for verbs.

Non-continuos phrases

The description does not allow of parsing verb complements if they occur outside the verbal phrase.

Efficiency of parsing

A TDLF algorithm working on the rules would be extremely inefficient – in order to parse complements preceding the constitutive verb, all admissible types of complements would have to be verified, with expensive backtracking after encountering the verb in the input string.

1.2. Świdziński (1992) [2]

An elementary sentence is described with the rule:

s -> {vp, req1, req2, req3}

An elementary sentence is realised by a verbal phrase with at most three complements. The subject is treated as one of the complements.

Characteristics of the approach

Number of possible non-subject complements

The number is equal to two, which does not seem to be sufficient for all NLP applications. For example, in order to correctly translate the sentence: *Tłumaczę teksty z języka polskiego na język angielski – ang.: I translate texts from Polish into English*, one needs to allow the verb *tłumaczę* to have three complements excluding the subject.

Order of complements

There are no limitations to the free order of complements, which is ill-advised from the point of view of parsing efficiency.

Parts of speech which may be complemented

The description allows prepositional phrases to modify parts of speech other than verbs, i.e. nouns, adjectives and adverbs. However, no attempt is made to solve structural ambiguities which may arise as a result of parsing.

Non-continuos phrases

The approach makes it possible to parse verb complements, also if they occur outside the verbal phrase.

Efficiency of parsing

The same problems apply as pointed out above for Szpakowicz (1.1.2).

1.3. Jassem (1997) [3]

The algorithm of parsing given in [3] takes the description of Szpakowicz as its starting point. However, the TDLF parsing algorithm is improved by deterministic analysis of verb complements preceding the verb.

Characteristics of the approach

Number of possible non-subject complements

The number is equal to three which seems to be sufficient for all NLP applications.

Order of complements

Free order is limited by information stored in the dictionary. This makes the algorithm check only for orders of complements which are likely to occur in a Polish sentence.

Parts of speech which may be complemented

The description allows complements only for verbs.

Non-continuos phrases

The description does not allow of parsing verb complements if they occur outside the verbal phrase.

Efficiency of parsing

The deterministic analysis of complements preceding verbs improves efficiency largely – in fact, an improvement of such type seems necessary to make the algorithm work in close-to-real time.

1.4. The present algorithm

The algorithm presented here aims at overcoming the drawbacks of the approaches mentioned above, i.e.

- the number of non-subject complements should be equal to three,
- free order of complements should be limited by information stored in the dictionary,
- complements should be allowed for all parts of speech that may actually be complemented in Polish. Moreover, the algorithm should enable structural disambiguation,
- verb complements should be parsed also if they occur outside the phrase,
- the algorithm should work in close-to-real time.

The below description of the algorithm concerns parsing an elementary sentence of the form: S -> np, vp.

The steps of the algorithm will be exemplified by parsing the sentence:

Mojego *(my)* ***szefa*** *(boss)* ***z*** *(from)* ***Niemiec*** *(Germany)* ***import*** *(import)* ***do*** *(to?, into?)* ***Polski*** *(Polska)* ***na pewno*** *(surely)* ***zadowoli*** *(will satisfy).*

The algorithm presented here aims at parsing the sentence in order to translate it into English. The output sentence should have the following form:

The import from Germany to Poland will surely satisfy my boss.

(Let us notice that the PP: *z Niemiec* may be 'on surface' interpreted as modifying either of: NP *mojego szefa*, noun *import*, verb: *zadowoli*. The algorithm will attach the PP to the noun *import* according to the information extracted from the dictionary).

Steps of the algorithm for parsing a Polish sentence with free order of components

The algorithm assumes that the subject may be preceded by the following types of phrases only:

- possible complement of a verb or a noun, i.e.:
  -- a nominal phrase in case other than nominative
  -- a prepositional phrase
  -- an adverbial phrase treated as a possible complement here (e.g. an adverb derived from an adjective)
- other adverbial phrases (e.g. adverbs of time).

1. Parse the sentence starting from the left side checking only for the types of phrases listed above or a subject.

Store possible complements in the complements list (CL).
Store adverbial phrases in the adverbs list (AL)

Until a subject is found. Denote the rest of the sentence (including subject) as Rest1.

The subject of the sentence is recognised as the first nominal phrase built around a nominative form of a noun or a pronoun. This may lead to confusion, as some nouns have forms in other cases identical to that of nominative and the algorithm will „lose some time" for checking a false hypothesis before failing and backtracking.

The first step results in the following mappings.

CL = [[mojego, szefa], [z, Niemiec]]

AL = []

Rest1 = [import, do, Polski, na pewno, zadowoli]

2. Parse the subject

2. 1. Check in the dictionary for possible complements of the constitutive noun.

For example, in the bilingual dictionary of the system POLENG, the complementation field for the word 'import' is

`{z G -> from NP, do G -> into NP}`

which means that the word 'import' may be complemented by one or both PP's: preposition 'z' followed by a nominal phrase in genitive or/and preposition 'do' followed by a nominal phrase in genitive; the PP's may occur in arbitrary order (arbitrary order is marked by braces).

2.2. Take the sublist [CL1] of CL which realises the complementation. Denote the rest of CL as CL2

CL1 = [[z, Niemiec]] (*realisation of '`z G -> from NP`')*

CL2 = [[mojego, szefa]]

2.3. Check for not realised complements behind the noun.

The complement '`do G -> into NP`' is realised by the phrase 'do Polski'.

2.4. Denote the rest of the sentence as Rest2. Pass the lists AL and CL2 to the analysis of a verbal phrase.

3. Parse Rest2 as the verbal phrase.

3.1. Find the constitutive verb of the phrase

3.2 Parse the sequence of words preceding the verb storing possible complements in CL3 and adverbial phrases in AL1.

CL3 = [],

AL1 = [[na pewno]]

3.3 Append pending lists of complements and adverbs respectively, giving CL4, AL2.

CL4 =[[mojemu, szefowi]],

AL2 = [[na pewno]]

3.4. Continue with the algorithm for parsing verbal phrases described in [3].

2. Dealing with non-language markers

Some texts contain non-language symbols inserted inside and a parsing algorithm should be able to parse such texts. An example may be a text in a HTML format where HTML markers intrude into plain texts. On the reasonable assumption that HTML markers embrace lexical phrases (e.g. in the sentence *Mojego szefa z Niemiec import do Polski na pewno zadowoli*, markers are supposed to embrace nominal phrases: *mojego szefa*, *import*, *Niemiec*, *Polski*, *z Niemiec import do Polski*, or the whole sentence, rather than random parts of the sentence like *import do*, *Polski na pewno*), it is easy to expand a PSG grammar into grammar which parses expressions including markers: it will do if each re-write rule is replaced by two rules: one for parsing a structure without markers, the other for parsing a structure embraced by a pair of markers (if a structure is marked with a left-hand marker only, then it may be interpreted as a special case of o pair of markers with the right-hand marker empty).

Another approach is not to double the rules in a source code: instead, it is possible to assume a formalism of coding rules with double interpretation – below referred to as **marker rules** - and design a compiler which translates each marker rule into two appropriate rules.

The formalism presented here consists in encoding marker rules by denoting the symbol of replacement with '->' (in contrast to DCG rules, where the symbol of replacement is denoted by '-->'). However, automatic replacement of **all** DCG rules by marker rules would be inexpedient as far as efficiency of parsing is concerned. It seems preferable to limit marker rules to expanding only those symbols of grammar which represent phrases likely to be embraced by markers. As a result, we need the compiler to be capable of dealing with three types of rules which may occur in a source file:

- marker rules ('->')
- DCG clauses ('-->'),
- standard Horn clauses (':-').

2.1. Gazdar-Mellish algorithm for compiling DCG rules

In [4] the authors give the algorithm for compiling DCG rules into Horn clauses with difference lists:

```
translate_clause(LHS_in --> RHS_in,
                 LHS_out:- RHS_out):-
(1)     LHS_in =.. LHS_list,
(2)     append(LHS_list, [In, Out], LHS_list1),
(3)     LHS_out =..[LHS_list1],
(4)     add_vars(RHS_in, In, Out, RHS_out), !.
```

Here is an explanation of predicates used in the algorithm:
(1) convert the term *LHS_in* into the list *LHS_list*, the first member of the list

being the name of the term *LHS_in* (standard Prolog procedure)
(2) append variables standing for difference lists to *LHS_list*
(3) convert the list back to the term, giving the compiled form of the left-hand side of the rule
(4) add variables standing for difference lists to all predicates of the right side of the rule
The predicate *add_vars* is described in detail in [4]. It needs small adjustments in order to deal with rules including alternatives and cuts.

2.2. Algorithm for compiling files including rules which parse expressions containing HTML markers

For each clause of the file:
If a clause is a Horn clause
Then assert it to the database
If a clause is a DCG clause
Then
Translate it into a Horn clause (Gazdar – Mellish algorithm)
Assert it to the database
If a clause is a marker rule
Then
Translate it into two Horn clauses according to the Marker Rules Compile Algorithm given below
Assert the clauses to the database

Marker Rules Compile Algorithm

It is assumed that a marker rule consists of a left-hand non-terminal symbol, character '->', and a right-hand side. The algorithm translates the rule into two Prolog rules: one for parsing the bare expression (without markers), the other for parsing the expression starting with a left marker.

Example

(The problem of trace parameter is ignored in the example).
A rule

```
np -> adj, noun
```

should be translated into two rules:

```
np--> adj, noun.
np --> left_marker(M), adj, noun, right_marker(M).
```

which re-written into Horn clauses will take the form :

```
np(In, Out) :- adj(In, Out1), noun(Out1, Out).
np(In, Out):- left_marker(In, Out1), adj(Out1, Out2), noun(Out2,
                Out3), right_marker(Out3, Out).
```

A problem of over-propagation of markers must be overcome. Suppose that the following rules are given in the grammar:

```
Grammar 1
(1) np_rel -> noun, rel_clause.
(2) np_rel -> noun.
(3) noun -> [dog].
```

The compilation would result in the following set of rules:

```
(1)  np_rel --> noun, rel_clause.
(2)  np_rel --> l_marker, noun, rel_clause, r_marker.
(3)  np_rel --> noun.
(4)  np_rel --> l_marker, noun, r_marker.
(5)  noun -> [dog].
(6)  noun --> l_marker, [dog], r_marker.
```

which result in two paths in which the np_rel "LM dog RM" (LM and RM stand for some left and right markers respectively) may be developed (one by rules (3) and (6) and the other by rules (4) and (5)).

A method for avoiding multiplication of success paths is that the compiling algorithm should recognise non-expanding rules (like rule (3)) and translate them into one rule only – the one not including markers.

A non-expanding rule may not be given explicitly and the algorithm should still recognise it in order not to replace it with both rules. For example, the rules 1) and 2) of Grammar 1 may be re-written by means of alternative and a trace parameter.

```
Grammar 2
np_rel(T) -> noun(T1),
             ( rel_clause(T2),
               {T = noun_rel_cl(T1, T2)}
             ;
               {T = T1}
             ).
```

A suggested solution is that parts of right-hand sides which stand for non-expanding rules (above: *T = T1*) should be appropriately marked (for example by double braces), in order to warn the algorithm of a possibility of over-propagation of markers.

If in the above rule the term *{T = T1}* were replaced by *{{T = T1}}*, the algorithm would be expected to translate the rule into the following Horn clauses:

```
np_rel(T, In, Out) :-
   noun(T1, In, Out1),
```

```
   (  rel_clause(T2, Out1, Out),
      T = noun_rel_clause(T1, T2)
   ;  T = T1,
      Out1 = Out
   ).
np_rel(T, In, Out) :-
   left_markers(MarkerList, In, Out1),
   noun(T1, Out1, Out2),
   rel_clause(T2, Out2, Out3),
   right_markers(MarkerList, Out3, Out),
   T3 = noun_rel_clause(T1, T2),
   modify_trace(T3, MarkerList, T).
```

Note that the parameter T must store the information on the type of the marker(s) that have been parsed. This is secured by the predicate *modify_trace*.

The algorithm of compiling marker rules is realised by the predicate *translate_marker_rule* given below (for the sake of simplicity the problem of compiling clauses with 'double braces' is omitted here). The first argument of the predicate is the input marker rule, the second argument is the output rule which allows of parsing a phrase without markers, the third parameter is the output rule which allows of parsing a phrase which starts with a marker.

```
translate_marker_rule(LHS_in -> RHS_in,
                                  (LHS_out:- RHS1),
                                  (LHS_out:- RHS2)):-

(1) LHS_in =.. LHS_list,
(2) last_element(LHS_list, TraceIn),
(3) lists_differ_by_last(LHS_list, LHS_in1, TraceOut),
(4) append(LHS_in1, [In, Out], LHS_out1),
(5) LHS_out =.. LHS_out1
(6) add_vars(RHS_in, In, Out, RHS1_out), !,
(7)  combine(RHS1_out, TraceOut = TraceIn, RHS1),
(8) add_markers(RHS_in, MarkerList, In, Out,
                RHS2_out),!,
(9) combine(RHS2_out, modify_trace(TraceIn, MarkerList,
               TraceOut), RHS2).
```

Here is the explanation of the subsequent sub-predicates:

(1) convert the term *LHS_in* into a list *LHS_list* (e.g. *np_rel(T)* is converted into *[np_rel, T]*),

(2) extract the last (trace) element (*TraceIn*) from the list *LHS_list*,

(3) make the list *LHS_in* be almost identical to *LHS_list* with the last element replaced by *TraceOut*
(4) append difference lists *In, Out* to *LHS_in1*, giving the final form of the left_hand side - represented in a list,
(5) convert "back" the list form of the left side into the term form,
(6) add variables standing for difference lists to the right-hand predicates of the rule, giving RHS1_out (as in G-M algorithm)
(7) make no changes to the trace parameter (as RHS1 corresponds to the right side of the rule without markers)
(8) to the right side of the rule, add the predicates which parse left and right markers; in the same predicate add variables denoting appropriate difference lists to the right-hand symbols
(9) to the right-hand side add the subgoal *modify_trace* (in order to make the trace parameter store the information on having parsed the markers)

2.3. Techniques for parsing punctuation marks

We hope that similar techniques may be used for parsing expressions containing punctuation characters like comas, colons, quotes or hyphens. Research has been started to define, on the basis of a text corpus, regularities of appearance of those characters in Polish texts.

3. Conclusion

The solutions described above have been applied to the translation algorithm of the system POLENG. The ideas seem general enough for use in any system which parses Polish expressions with a TDLF algorithm.

BIBLIOGRAPHY

[1] Szpakowicz S., *Formalny opis składniowy zdań polskich*, Wydawnictwa Uniwersytetu Warszawskiego, 1986

[2] Świdziński M., *Gramatyka formalna języka polskiego*, Wydawnictwa Uniwersytetu Warszawskiego, 1992

[3] Jassem K., *POLENG - a Machine Translation System Based on an Electronic Dictionary*, in: "Speech and Language Technology. Volume 1", Poznań 1997

[4] Gazdar, Melish, *Natural Language Processing in Prolog: an Introduction to Computational Linguistics*, 1989