Morphological Analyzer as Syntactic Parser

Gábor Prószéky

MorphoLogic

Németvölgyi út 25, Budapest, H-1126 Hungary

h6109pro@ella.hu

Abstract. We describe how a simple parser can be built on the basis of morphology and a morphological analyzer. Our initial conditions have been the techniques and principles of **Humor**, a reversible, string-based unification tool (Prószéky 1994). Parsing is performed by the same engine as morphological analysis. It is useful when there is not enough space to add a new engine to an existing morphology-based application (e.g. a spell-checker), but you would like to handle sentence-level information, as well (e.g. a grammar checker). The morphological analyzer breaks up words into several parts, all of which stored in the main lexicon. Each part has a feature structure and the validity of the input word is checked by unifying them. The morphological analyzer returns various information about a word including its categorization. In a sentence, the category of each word (or morpheme) is considered a meta-letter, and the sentence itself can be transformed into a meta-word that essentially behaves like a real one. Thus the set of sentences recognized by the parser called **HumorESK** can form a lexicon of meta-words that are processed much the same way as lexicons of real words (morphology). This means that algorithmic parsing step are substituted by lexicon look-up, which, by definition, is performed following the surface order of string elements. Both the *finitizer* that transforms formal grammars into finite lexicons and the run-time parser of the proposed model have running implementations.[1]

1 INTRODUCTION

Lexical entries in a morphology-based system are words. Because of the similarity, syntactic constructions occurring as entries in a meta-lexicon can be called meta-words. Meta-letters, that is, letters of a meta-word are morpho-syntactic categories having an internal structure that describes syntactic behavior of the entry in higher level constructions. The system called **HumorESK** (**Humor** Enhanced with Syntactic Knowledge, where **Humor** stands for High-speed Unification Morphology) to be shown here consists of numerous meta-lexicons. Each of them has a name: the syntactic category it describes. Categories like S', S, NP, VP, etc. are described in separate lexicons. Meta-lexicons form a hierarchy, that is, letters in a meta-lexicon can refer to other (but only lower level) lexicons. Parsing on each level, therefore, can be realized as lexical look-up. Neither backtracking, look-ahead, nor other time-consuming parsing steps are needed in order to get the analysis of a sentence. The only on-line operation is a unifiability check for each possible lexical entry that matches the sentence in question.

[1] This work was partially supported by the Hungarian National Scientific Fund (OTKA).

Grammars are compiled into a multi-level pattern structure. On a lower level, parsing a word results in a meta-letter, that is, part of a meta-word on a higher level. Such structures, for example, NP and VP, are meta-letters coming from lower levels and form a meta-word that can be parsed as a sentence, because of the existence of a rule S → NP VP in the original grammar. A complex sentence grammar can be broken up into non-recursive grammars describing smaller grammatical units on different levels. These grammars are, of course, much simpler than the original one. Recursive transition networks (RTN) can also be made according to similar principles, but their recursive nature cannot be found in our method. In other words: the output symbol of any level does not occur in the actual or lower level dictionaries.

The whole lexicon cascade can be generated from arbitrary grammars written in any usual (for the time being, CF, but in the near future any feature-based) formalism. We call this step *grammar learning*. The software tool we have developed for this reason takes the grammar as input, creates the largest regular subset of the language it describes regarding the string-completion limit of Kornai (1985), then forms a finite pattern structure by depth limit and length limit from the above regular description.

2 PARSING WITH PATTERNS

Parsers are (computational) tools that read and analyze a sentence, and return a wide range of information about it, that is, they recognize

(1) if the input is a valid sentence (according to the rules of the object language),
(2) segment the input sentence as many ways as possible, and
(3) provide some custom information.

The latter custom information can be a simple 'OK' sign indicating that the sentence is well-formed (*grammar checker*), but it can also be the same sentence in another language (*translation tool*), or, in case of a (grammatically) incorrect sentence, it can be a list of suggestions how it may be corrected (*grammar corrector*). In the present implementation we use morpho-syntactic categories as output information on every level (*parser*).

For the input sentence

The dog sings.

the English module of **Humor** returns the following morphological categorization:

The[DET] *dog*[N] *sing*[V]+[3SG] .[END]

Let us now strip off the actual words from the morphological information (from now on we call them morphological codes or *morph-codes*). Writing only the morph-codes, we get

DET N V 3SG END.

The problem is now how we recognize this as a sentence. This sequence must somehow be stored in another lexicon

describing phrases and phrase structures. It is quite clear that in the above string, DET, N, and V are simple symbols that can easily be encoded as single letters like *d, n, v, x* and *e*. Transforming the sequence of morph-codes we get the word *dnvxe*. Earlier we said that the **Humor** engine is lexicon-independent, so if we have another lexicon, we can easily switch to it and instruct **Humor** to analyze the actual word. **Humor** returns something like *dnvxe*[S] where 'S' is now the category of the input word indicating that it is a sentence.

The meta-level, of course, can be split up to further levels. Let us use, for the sake of simplicity, a simple toy grammar of two levels for the nominal phrase and the sentence:

(Level 2) S → NP S, S → S NP, S → V (3SG)
(Level 1) NP → DET NG, NG → ADJ NG, NG → N

Now we feel a need for a tool that generates a set of finite patterns out of this grammar description. We, therefore, developed a tool that finds the largest regular subset of a context free language (regarding a special parameter set) and then uses a recursive generator to produce the finite patterns. |For the above toy grammar a possible lexicon can be the following:

(Level 2): V END, V 3SG END, NP V END. NP V 3SG END, NP V NP END, NP V 3SG NP END, V NP END, V 3SG NP END, ...
(Level 1): DET N, DET ADJ N, DET ADJ ADJ N, ...

If we use letters *v, m, x, n, a* and *d* for V, NP, 3SG, N, ADJ and DET, respectively, we get the following lexicons:

(Level S) *ve, vxe, mve, mvxe, mvme, mvxme, vme, vxme, ...*
(Level NP) *dn, dan, daan, ...*

If the appropriate lexicons are built from the pattern lists for grammars of both levels, the parser is ready to run. The parsing algorithm can be outlined as follows. The parser runs a morphological analysis on each word in the input sentence and encodes the morph-codes into meta-letters. Using our example, *The dog sings* (DET N V 3SG END) the parser will find that the string 'DET N' forms a noun phrase, because *dn* can be found in the NP lexicon. The meta-morphological analysis (a search in the lexicon of the patterns of Level 1) returns *dn[m]*, that is, DET N [NP]. For level 2, the parser exchanges the substring 'DET N' with the meta-letter 'NP'. So the new meta-word is *mve*, that is, 'NP V END' which is accepted by the Level 2 grammar (sentences). In fact, we have another meta-word here, namely, a single *n* (='N') that can also be categorized as a noun phrase (*m*); and this yields *dmve*, that is, 'DET NP V END' which is not accepted by the Level 2 grammar. Giving these two as input to the Level 2 meta-morphological analysis, the system will reject *dmve* 'DET NP V END' but will accept *mve* 'NP V END' by returning *mve*[S], that is, NP V END [S].

It is clear that no backtracking is possible in our run-time system, that is, a meta-word cannot be categorized by a symbol that is a meta-letter of meta-words on the same or lower level. It is an important restriction: category symbols must be meta-letters used only on higher levels. This constraint provides us with another advantage: any set of category symbols (higher level meta-letters or meta-morph-codes) is disjoint from the set of lower level meta-letters (or meta-letters used on the level of morphology), therefore, parsing lexicons can be unified: meta-words (morphological or any set of phrase structure patterns) for all levels can be stored in a single lexicon.

In the explanation of the parsing techniques we have excluded one aspect until this point, and this is *unification*. Without feature structures and unification, however, numerous incorrectly formed sentences are accepted by the parser. If a meta-word is not found, it is rejected and the process goes on to the next meta-word. If the meta-word is found, then it may still be incorrect. This is checked through the *unifiability-checking* of the feature structures of its meta-letters. For instance, in a noun phrase 'DET N', the unifiability of the feature structures assigned to DET and N is checked. If they are not unifiable, the meta-word is rejected and the process goes on to the next meta-word. If they are unifiable, the output is passed on to the next level. The last level is responsible for providing the user with the proper analysis, that is, all the information collected so far.

3 FROM GRAMMARS TO LEXICAL PATTERNS

All infinite structures generated by recursion can be restricted by limiting the *recursion depth*. This means a constraint of the depth of the derivation tree of a sentence in a language. We can also restrict the *direction of branching* in the derivation tree. This means that we could generate (finite) patterns directly from the original (context-free) language imposing various limits on embedding; but these methods can be too weak or too strong and, most of all, irrelevant to the object language. There is, however, a slighter constraint that helps transforming context-free grammars. According to Kornai's hypothesis (Kornai 1985), any string that can be the beginning of a grammatical string can be completed with *k* or less terminal symbols, where *k* is a small integer. This *k* is called the *string completion limit* (SCL). A grammar transformation device can be instructed to discard sentence beginnings that have a minimal SCL larger than specified (by the user). SCL limits center-embedding but allows arbitrary deep right-branching structures (easily defined by right regular grammars). Left branching is also limited, but this limitation is less pronounced than that of center-embedding.

Our special tool, GRAM2LEX, takes a CF grammar as input. As a first step, it reads the grammar and creates the appropriate RTNs from it. Goldberg and Kálmán (1992) describe an algorithm unifying recursive transition networks. We have improved their algorithm. Its implementation is incorporated into the GRAM2LEX tool as a second processing phase. The algorithm creates the largest regular subset of a context-free language that respects the SCL. In terms of finite state automata, SCL is the number of branches in the longest path from a non-accepting state to an accepting one (regarding all such paths). The process results a finite state automaton. In order to get a finite description from the FSA we introduced two independent parameters. The *length of the output string* (in terms of terminal symbols) If the current string reaches the maximum length, the recursion is cut and the process immediately tracks back a level. The *maximum number of passing the same branch* during the generation of an output string can also be specified. In the current implementation, this

maximum is global to a whole output string. There is, however, another approach: this number can be related to the current recursion level, so if a certain iteration occurs at more than one position in a sentence, the maximum length of the iteration is the same at both positions and the actual lengths are independent.

The GRAM2LEX tool takes all the three parameters (the SCL, the maximum string length and the maximum iteration length) as user-defined ones. The set of finite patterns can be compiled into a compressed lexicon with MorphoLogic's lexicon compiler. The GRAM2LEX tool produces a file in the input format required by this compiler.

Levels of the parser are individual processes that communicate with each other. The most important medium is the *internal parsing table* that represents the parsing graph described below. Based on that graph, the process of a particular level is able to execute its main functional modules, namely

- to create the appropriate input to call the morphology engine,
- switch to the phrase pattern lexicon of the current level,
- run the morphology engine and process the output of the morphology engine, and
- if possible, insert new branches into the parsing graph for the next level.

Each level is an independent process communicating with the others (including level 0, the morphological analysis). The medium of communication is the parsing graph of which there is only one copy and is generally accessed by all levels. The parsing process on each level can be decomposed into three layers. All levels have the same functionality; it is only the internal operation of the first layer that differs in the case of the lowest level (morphology) and the highest one (sentences):

- *pre-process* that based on the current structure of the parsing graph (if it exists), produces the set of the possible phrase structures,
- *search* that checks all the elements of the set generated by Layer 1 if they are acceptable by the current level using the **Humor** engine equipped with the current level's parsing lexicon,
- *post-process* that based on the patterns accepted by Layer 2, inserts new nodes and branches into the parsing graph.

The different levels are connected to each other like the layers of a single level. The structure of our present (demonstrational) 0-1-2-level parser for Hungarian is the following:

- *Morphology* (Preprocess Words, Search Morphology Lexicon, Create/Modify Parsing Graph),
- *Noun Phrases* (Create Patterns, Search Level 1 Pattern Lexicon, Modify Parsing Graph),
- *Sentences* (Create Patterns, Search Level 2 Pattern Lexicon, Modify Parsing Graph).

4 IMPLEMENTING THE RUN-TIME PARSER

In the current implementation, the parsing levels are executed sequentially, but they can be made concurrent: during one session, level 0 reads a word from the input sentence, analyzes it and inserts the appropriate nodes and branches into the parsing graph. Further on, the system has a self-driving structure: the level that made changes to the parsing graph sends an indication to the next level which then starts the same processing phase. The changes in the parsing graph are thus spread upwards in the level structure. When the last level (usually the highest) finished updating the graph, it sends a 'ready for next' signal to level 0 which starts the next session.

Termination is controlled by level 0: if it finished analyzing the last word (morpheme) of the sentence, it sends a 'terminate' signal to the next level. Receiving this signal, intermediate levels pass it to the next level after finishing the processing the changes that were made to the parsing graph. The last level (usually the highest) then terminates all levels and passes the parsing graph to the output generator.

Let us see an example:

Patterns:
S: NP VP END
NP: N | N N | DET N | DET ADJ N | DET ADJ ADJ N
VP: V | V 3SG | V NP | BE VING | BE VING ADV | V NP
END: . | !

Input: *Professor Smith is coming home.*

Output:
S → [NP VP END]
NP → [N N]
N → *Professor*[N]
N → *Smith*[PROP]
VP → [BE VING ADV]
BE → *is*[BE]
VING → *come*[V]+*ing*[ING]
ADV → *home*[ADV]
END → .

This is the inherent tagging of the sentence built from the information stored directly in the phrase structure patterns. We have begun, however, the development of another type of tagging where phrases correspond to the source grammars' non-terminal symbols, like this:

```
(S
  (NP
    (N professor)
    (N Smith))
  (VP
    (BE is)
    (VG
      (VING
        (V come)
        (ING ing))
      (ADV home))))
```

The current average speed of this multi-level system (even for dictionaries with 100.000 entries) is around 50 input/sec for each module on a Pentium/75 machine, where input can mean either sentence or phrase or word to be analyzed.

5 USER INTERFACE

The current implementation of the **HumorESK** parser allows the run-time expansion of the user-defined lexicon file. This was achieved by developing a small user interface that performs the following functions:

- Works in both batch and interactive mode.
- Users can review all the different taggings of a sentence.
- Users can view the internal parsing table from which the parser output was generated. This means the review of the analysis of each morpheme and the meta-words generated from them.
- Users can view both the morpho-lexical and the syntactical part of the user-defined lexicons.
- The user can add new entries to the user-defined lexicon file on any level. The changes take effect suddenly, that is, when processing the next sentence or re-parsing the last one.

6 CONCLUSION

We have developed a parser called **HumorESK** that is quite powerful (even in its present format, without feature structures) and has several important features:

1. unified processing method of every linguistic level
2. possible parallel processing of the levels (morphology, phrase levels, sentence level, etc.)
3. morphological, phrasal and syntactic lexicons can be enhanced, even in run-time
4. easy handling of unknown elements (with re-analysis)
5. easy correction of grammatical errors
6. reversible (generation with the 'synthesis by analysis')
7. the same system can be used both for corpus tagging and fine-grained parsing

Feature 1 seems important if there is not enough space to add a new engine to an existing morphology-aseed application (e.g. a spell-checker), but you must handle sentence-level information, as well (e.g. a grammar checker). Real parallelism indicated in 2 has not yet been implemented. Usefulness of attributes 3–6 are going to be proven in practice, because we have just finished the first version of the first Hungarian grammar checker called **Helyesebb**. It uses the spelling corrector and morphological analyzer/generator modules relying on the **Humor** morphological system – the basis of **HumorESK** – that are widely used by tens of thousands of both professional and non-professional end-users (Prószéky 1994, Prószéky et al. 1994). We have results in proving the first part of feature 7, namely corpus tagging. Fine-grained parsing would need the extended use of features. This system – as we mentioned earlier – is under development.

7 REFERENCES

[1] Goldberg, J. and L. Kálmán, 'The First BUG Report', *Proceedings of COLING-92*, Nantes (1992).

[2] Kis, B. 'Parsing Based on Morphology', Unpublished Master's Thesis, Budapest Technical University, 1995.

[3] Kornai, A. 'Natural Languages and the Chomsky Hierarchy', *Proceedings of the 2nd Conf. of the EACL*, Geneva, 1-7. (1985).

[4] Prószéky, G. 'Industrial Applications of Unification Morphology', *Proceedings of ANLP-94*, Stuttgart (1994).

[5] Prószéky, G., M. Pál and L. Tihanyi, '**Humor**-based Applications', *Proceedings of COLING-94*, Kyoto (1994).


```
[12]  A Duna után a Tisza a legnagyobb folyónk.
[1/1]                              0:00:02.91                    HumorESK 2.0 REVIEW

        SS -> [DP Cas DP DP End]
               DP -> [Det N]
                     Det -> a[Article]=A
                       N -> Duna[ProperNoun]
             Cas -> után[PostPosition]
               DP -> [Det N]
                     Det -> a[Article]
                       N -> Tisza[Noun]
               DP -> [Det Adj\ Adj \Adj N PSfx]
                     Det -> a[Article]
                    Adj\ -> leg[Superlative]
                     Adj -> +nagy[Adjective]
                    \Adj -> +obb[Comparative]
                       N -> folyó[Noun]
                    PSfx -> +nk[PersSuffPlurFirst]
             End -> .

First[^HOME]       Last[^END]       [G]o to          Syntax exceptions[Alt+S]
[P]arse again      Accept[^ENTER]   [R]eject           Word exceptions[Alt+W]
Exit[ESC]                                          Internal parsing table[F10]
```

Figure 1. HumorESK-analysis of the sentence
"A Duna után a Tisza a legnagyobb folyónk."
(After the Danube, our biggest river is Tisza.)