1

Tabulation for multi-purpose partial parsing

Vitor Jorge Rocio (vjr@di.fct.unl.pt)* and Gabriel Pereira Lopes (gpl@di.fct.unl.pt)
CENTRIA - Departamento de Informática
Faculdade de Ciências e Tecnologia
Universidade Nova de Lisboa
Quinta da Torre
2825 Monte de Caparica
PORTUGAL

Eric de la Clergerie (eric.clergerie@inria.fr)
INRIA - Rocquencourt - BP 105
78153 LE CHESNAY
FRANCE

Abstract. Efficient partial parsing systems (chunkers) are urgently required by various natural language application areas as these parsers always produce partially parsed text even when the text does not fully fit existing lexica and grammars.

Availability of partially parsed corpora is absolutely necessary for extracting various kinds of information that may then be fed into those systems, increasing their processing power.

In this paper, we propose an efficient partial parsing scheme based on chart parsing that is flexible enough to support both normal parsing tasks and diagnosis in previously obtained partial parses of possible causes (kinds of faults) that led to those partial parses instead of complete parses.

Through the use of the built-in tabulation capabilites of the DyALog system, we implemented a partial parser that runs as fast as the best non-deterministic parsers. In this paper we ellaborate on the implementation of two different grammar formalisms: Definite Clause Grammars (DCG) extended with head declarations and Bound Movement Grammars (BMG).

Keywords: Constituent movement, Head-driven parsing, Partial parsing, Tabulation

1. Introduction and motivation

It is almost impossible to completely parse every sentence in real text from electronically available sources given the current state of the parsing technologies. This is due to several unavoidable factors: errors in the input text, unknown words, insufficient or erroneous knowledge in both the grammar and the lexicon, errors introduced during pre-

* VJR POR OS PROJECTOS
* VJR POR AS BOLSAS

parsing phases by tokenizers, part-of-speech (POS) taggers, heuristic proper name identifiers, etc.

However, unless fully parsed text is specifically required, there is a lot of information that can be extracted from partially parsed text, namely the syntactic structures for segments of the input text, other than complete sentences, and this information can then be used for automatically learning subcategorization frames (Roth and Carroll, 1996; Collins, 1997; Carroll et al., 1998), for extracting information about gender and number of nouns and adjectives that did not exist in the lexicon but that were POS-tagged (Marques and Lopes, 1996a)and for defining variable length windows to be used on word sense disambiguation, on automatic thesaurus construction (Grefenstette, 1994), on pp-attachment, relative clause attachment, adjective phrase attachment (Ratnaparkhi, 1998; Yeh and Vilain, 1998; Collins and Brooks, 1995). Most work on these matters use fixed length windows ($\pm n$ words, with $n = 5, 7, 10$) but their efficiency would be improved if the window length is fixed regarding the number of phrases (not words) to the right or to the left of the word under study.

Another approach to chunking starts with human validated parsed corpora and learns how to chunk new texts from those examples (Daelemans et al., 1999; Ramshaw and Marcus, 1995). Although at first these example-based approaches might seem quite appealing and attractive they embody a reasoning loop. In order to learn a grammar they require parsed text for being trained and so the bottleneck is placed at the production of correctly parsed text which is not exactly a problem once we have grammars that were produced and ameliorated along the years.

The use of indexed partially parsed text collections (together with the indexed raw text) brings new insights to information retrieval tasks by improving precision, by normalizing the text bases and queries, by enabling further disambiguation of word senses, by allowing ellipsis and anaphora resolution in those collections and bringing up the power of using multiple knowledge sources other than the morphological information analysis that is generally used for performing these kinds of tasks.

The creation of partial treebanks is an important application of partial parsing. For languages, such as Portuguese, for which these resources are scarce or difficult to access, this work can serve as a bootstrapping process for automatically acquiring and/or improving lexical and grammatical knowledge mainly by using statistical induction techniques.

Besides the need for partial treebanks, it is necessary that the parsing process can contribute to detect and overcome the causes that lead

to partial parses. The so-called robust parsers try to overcome incomplete/erroneous information by using error anticipation and constraint relaxation techniques. These methods, though avoiding the failure of the parsing process, do not attempt to identify the real causes for strict partial parsability and do not use that information for learning from previous experience. On the other hand, partial parses provide information that help to pinpoint the real causes for partial parsability and to find corrections for those failures. According to our approach, the use of the same chart parsing machinery (the one we are going to present in this paper) both for partial parsing and for diagnosing parsing faults and proposing fault corrections, enables the construction of multiple instances of a partial chart parser, that just differs from each other on their agenda initialization policies. These parsers will act at different stages of the parsing problem resolution. The agenda is initialized either with actual lexical information for normal partial parsing or with alternative information, corresponding to possible corrections (properly marked with fault modes in order to allow their incorporation and propagation through charts while parsing), for fault diagnosis (Lopes and Rocio, 1999; Lopes et al., 1999; Lopes and Balsa, 1998; Balsa et al., 1995). Alternative agenda information leads to alternative partial parses. A partial parse will only be considered for further fault finding and fault repair if and only if it is better (it has lower granularity, see definition 2 in section 2) than the parses previously obtained by alternative diagnosis hypotheses. As a consequence one obtains a declaration of the fault modes involved, and at the same time a parse is obtained taking that fault declaration into account.

The partial parsing machinery that we present in this paper follows the chart parsing philosophy, by using the built-in tabulation of DyALog (Clergerie and Lang, 1994), a logic programming environment, with an execution model that enables structure sharing and storing of partial results. The grammars are clearly separated from the parsing processes (which are hidden in DyALog) and the use of tabulation allows the machinery to run as fast as other non-deterministic parsers (Abney, 1996; Hobbs, 1997). The partial parsers presented in this paper use two declarative grammar formalisms: DCG extended with head declarations for describing the syntax of phrases where there is no linguistic material moved away from its regular position and Bound-Movement Grammar (Lopes et al., 1996) for describing various kinds of linguistic movement and enabling the binding of moved material to their traces (questions, verb fronted sentences, prepositional phrase fronted sentences, relative clauses, clitics movement). These two formalisms are linguistically adequate for describing natural language syntax in two levels. The head declarations suggest a head-driven, bi-directional analysis of text

(left-to-right and right-to-left). A mixed search strategy (top-down and bottom-up) is also implicit in the grammar formalism so that the most efficient parser can be built automatically, without losing grammatical declarativity.

Bound-Movement Grammar (BMG, for short) involves movement operators that were incorporated in DyALog. Thus, a BMG can be directly compiled, and the corresponding parser is also generated automatically. This parser acts on the chunked text obtained by the previous bi-directional head-driven DCG based parser.

Section 2 of this paper defines some fundamental notions on partial parsing used throughout the paper. In section 3 the parsing architecture is presented. An introduction to the DyALog system is made in section 4. Sections 5 and 6 introduce the two grammar formalisms and detail the implementations of the corresponding parsers in DyALog. Experiments with the partial parsers are reported in section 7. We discuss our approach and compare it with related work in section 8 Finally, in section 9 conclusions are drawn and future work in the area will be discussed.

2. Definitions

The notions related to partial parsing used in this paper are formally defined in this section. We assume here a rather general notion of grammar. The important aspect is the existence of a derivation relation, denoted as

$$S_0 \Rightarrow^* S_1 ... S_n$$

where $S_0, S_1 ... S_n$ are grammar symbols (terminal or non-terminal). All other traditional grammar features (initial symbol, set of terminals, form of production rules) are undefined, leaving room for the use of a rather diversified set of formalisms. Two instances of grammar formalism are defined in sections 5 and 6, corresponding to two levels of syntactic analysis acting in tandem.

The notion of partial parse is central to our work and is formally defined as:

Definition 1 - A *partial parse* P over an input string $w_1 ... w_n$, according to a grammar G is a sequence of k tuples of the form $< p_i, p_{i+1}, C >$, $i = 1...k$, where $p_1 = 0$, $p_{k+1} = n$, $p_i < p_{i+1}$ for each i, and C is a non-terminal from G deriving the input substring stretching from position

p_i to position p_{i+1}, i.e., $C \Rightarrow^* w_{p_i+1}...w_{p_{i+1}}$ (each token w_p is located between positions $p-1$ and p).

In order to select partial parses for further consideration in a fault finding process, we need a measure of granularity.

Definition 2 - The *granularity* **g** of a partial parse P is the ratio k/n, where k is the number of triples in P and n is the number of tokens in the input string.

We are now able to define a partial parser.

Definition 3 - A *partial parser* is a function **F** that maps a non-empty set **P** of partial parses over an input string $w_1...w_n$ into another non-empty set of partial parses over the same input string, on the condition that for each $P \in \mathbf{P}$ there is a $Q \in \mathbf{F(Q)}$ such that $\mathbf{g}(Q) \leq \mathbf{g}(P)$.

From definition 3, we notice that, in order to produce partial parses from an input string, we need at least one partial parse! This apparently endless recursion is easily solved by pre-parsing (POS-tagging) the input string, i.e., by assigning part-of-speech tags to each token in the input string. As we can see from definitions 1 and 2, the result is already an upper bound granular partial parse.

Finally, we define a subsumption relation between partial parses, since the output of the partial parser can contain redundancies in the form of partial parses subsumed by other partial parses.

Definition 4 - A partial parse P is *subsumed* by another partial parse Q (both P and Q are partial parses over $w_1...w_n$ according to a grammar G) iff for each triple $< p_i, p_{i+1}, C_1 > \in P$ there is a triple $< q_j, q_{j+1}, C_2 > \in Q$ such that $p_i \geq q_j$, $p_{i+1} \leq q_{j+1}$ and $C_2 \Rightarrow^* XC_1Y$, where X,Y are strings of symbols from G.

3. Parsing architecture

The partial parser implemented in the scope of the work presented in this paper is divided into four levels of a cascaded architecture. The lower level (level 0) is a neural-net part-of-speech tagger (Marques and Lopes, 1996b; Marques, 2000) that assigns syntactic categories to words in raw input text, thus producing a first partial parse. Each of the subsequent levels (1, 2 and 3) picks up the partial parse produced

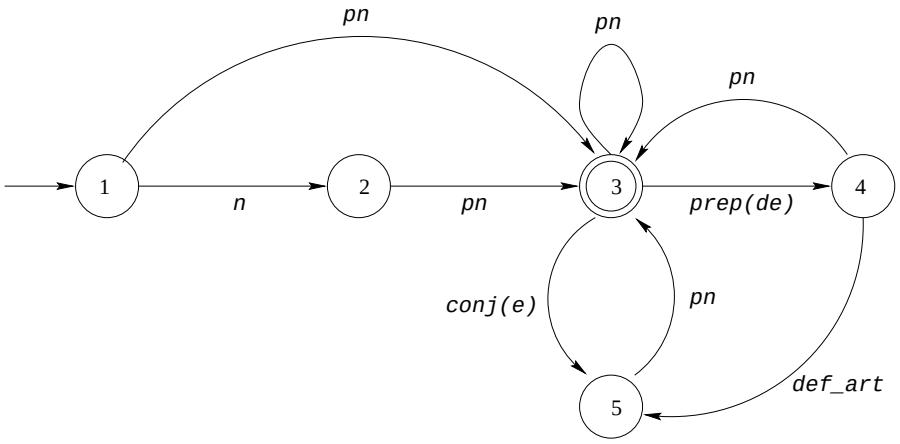

Figure 1. FSA for compound proper names

by the previous level and produces another partial parse with lower granularity (or equal, in the worst case).

Level 1 is a pre-processor based on finite-state automata. Its purpose is to identify sequences of words described by very specific, local grammars: numbers written in full, dates, compound proper names, temporal/spatial adverbial clauses and complex prepositions, adverbs and conjunctions. We show in figure 1 the automata for compound proper names in Portuguese as an example of the mechanisms used at this level[1].

Level 2 is a partial chart parser using a DCG extended with head declarations, and identifies sentence constituents (noun phrases, verb phrases, prepositional phrases, adjective phrases and adverbial phrases) involving no movement. Level 3 deals with constituents moved out of the canonical order, and uses the Bound Movement Grammar (BMG) formalism. Both levels 2 and 3 are described in more detail in sections 5 and 6. These two levels work in tandem.

4. A brief description of DyALog

Levels 2 and 3 of the parsing architecture described in the previous section are implemented in DyALog, a logic programming environment featuring tabulation and structure-sharing. These features are achieved through an execution model based on logic pushdown automata (LPDA).

4.1. Logic Pushdown Automata (LPDA)

LPDA are non-deterministic pushdown automata with logic terms as stack symbols. A classic PDA consists of a finite state memory and a pushdown memory (whose state determines the state of the automaton), together with a finite set of transitions, defining the possible successive state changes. The finite state memory is absent from LPDA since it can always be encoded in the pushdown stack, without loss of generality.

There are three kinds of transitions in LPDA, corresponding to different kinds of operations on the pushdown stack: $PUSH$, POP and $SWAP$. $PUSH$ is the classic primitive of stack manipulation for inserting a symbol on the top of the stack. POP is *not* the classic POP primitive for stacks. Instead, it replaces the two stack top symbols by another symbol. $SWAP$ simply replaces the top symbol with another symbol. The following notation for these transitions is used in this paper:

$$PUSH : B \mapsto CB$$
$$POP : BD \mapsto C$$
$$SWAP : B \mapsto C$$

The $\mapsto$ operator denotes a transition between the states specified on its left- and right-hand sides. B, C and D are any stack symbols and the sequences shown represent the top of the stack (leftmost corresponds to topmost).

A transition is applicable to a given state (represented by the stack) if the symbol(s) on the top of the stack unify with the symbol(s) on the left of the $\mapsto$ operator. In this case, the stack is subject to the specified operation ($PUSH$, POP or $SWAP$) and the substitution resulting from the unification is applied to the whole stack, producing the new state of the automaton.

LPDA are the basic model of logic program execution in DyALog. There are two essential procedures that need to be performed in order to run a logic program: the compilation of a logic program into a set of LPDA transitions and the implementation of the resulting non-deterministic LPDA.

More details on the internal workings of DyALog can be found on (Clergerie and Lang, 1994). The compilation scheme for BMG is presented in this paper on subsection 6.2.4. The LPDA implementation uses tabulation techniques and allows the use of different search strategies through modulation, which is described on the following subsection.

4.2. Modulation

In DyALog, a predicate is solved by pushing a **call** atom into the LPDA stack and waiting for a corresponding **return** atom to be popped. Different search strategies are achieved by distributing (modulating) the predicate information (functor and arguments) between these two atoms. For instance, to obtain a pure top-down search strategy, all predicate information must be conveyed in the **call** atom. On the other hand, to obtain a bottom-up strategy, none of the predicate information should be conveyed in the **call** atom, but only matched against the **return** atom. Finer strategies can also be defined.

DyALog provides a directive `dcg_mode`/4 to specify the modulation relative to a set of DCG (or BMG) non-terminals (and mirrors the declaration `mode`/2 used for Prolog predicates).

Both non-terminal names and respective arguments can be modulated with the following scheme:

```
:-dcg_mode(NT, NT_Info, Left_Pos, Right_Pos).
```

`NT` is a non-terminal or a list of non-terminals, `NT_Info` specifies the type of modulation for the non-terminal functor and arguments, and `Left_Pos` and `Right_Pos` specify the type of modulation for the left and right positions[2] in the input. A plus sign (`+`) in each of `NT_Info`, `Left_Pos` and `Right_Pos` indicates that the respective information is conveyed in the **call** atom, while a minus sign (`-`) indicates the opposite: the info should only be matched against the **return** atom. In other words, `+` indicates top-down prediction, while `-` indicates bottom-up propagation. The second argument of the `dcg_mode`/4 directive, `NT_Info`, may separately specify modulations for both the non-terminal name and respective arguments, through a notation that mirrors the functor-argument structure of the non-terminal. For instance, a value of $+(+,-)$ for `NT_Info` with respect to a non-terminal nt with two arguments specifies a `+` modulation for the name nt, a `+` modulation for the first argument and a `-` modulation for the second argument.

As a more thorough example, the directive

$:-$`dcg_mode`$([nt_1/2, nt_2/2], +(+,-), +, -).$

produces the call and return atoms in table I (assuming nt_1 and nt_2 non-terminals with 2 arguments each, arg_{11} and arg_{12} for nt_1, and arg_{21} and arg_{22} for nt_2):

This particular modulation for the left and right positions (respectively `+` and `-`) assumes that parsing is to be done from left to right,

Table I. Call and Return atoms for nt_1 and nt_2

DCG non-terminal	$nt_1(arg_{11}, arg_{12})$	$nt_2(arg_{11}, arg_{12})$
Expanded	$nt_1(L, R, arg_{11}, arg_{12})$	$nt_2(L, R, arg_{11}, arg_{12})$
Call	$dcg\_call\_nt_1\_2(L, arg_{11})$	$dcg\_call\_nt_2\_2(L, arg_{21})$
Return	$dcg\_return(R, arg_{12})$	$dcg\_return(R, arg_{22})$

`s --> np, vp.`	`head(s, vp).`	(a)
`np --> det, n(Type), n_args(Type).`	`head(np, n).`	(b)
`vp --> v(Type), v_args(Type).`	`head(vp, v).`	(c)
`pp --> prep, np.`	`head(pp, np).`	(d)
`n_args(0) --> [].`		(e)
`n_args(1) --> pp.`		(f)
`v_args(0) --> [].`		(g)
`v_args(1) --> np.`		(h)
`v_args(2) --> np, pp.`		(i)
`v_args(3) --> pp.`		(j)

Figure 2. Example of a grammar for level 2

since only the left position is known at call time. When parsing right-to-left, the modulation for the left and right positions must be `-` and `+`, respectively, because now only the right position is known at call time. If neither the left nor the right positions are known, then modulation of both parameters must be `-`. The possibility of specifying different modulations for different predicates is especially useful in our mixed bi-directional parsing strategy, as we will see in the following section.

5. Partial parsing with DCG extended with head declarations

Level 2, in our parsing architecture, identifies sentence constituents, using a DCG grammar extended with head declarations. We first present the grammar notation, and then detail the parser implementation.

5.1. Definition and example of DCG extended with head declarations

The grammar for level 2, which describes the structures of phrases in a sentence, is a DCG (Pereira and Warren, 1980), extended with head declarations, which specify head constituents in phrases.

Figure 2 shows an example of a reduced grammar used at level 2 [3]. Declaration (b), for instance, states that a noun is the head of a noun phrase[4].

The head declarations allow the parsing procedure to follow a bidirectional strategy; those constituents to the right of the head are analyzed left-to-right, those to the left are analyzed right-to-left.

This horizontal bidirectionality is combined with a vertical bidirectionality that reflects the co-existence of rules that have an associated head declaration and rules that do not have it. Rules that have an associated head declaration are triggered by the head category in a bottom-up way; those that don't have an associated head declaration are triggered top-down by the category on the left-hand side.

Careful coding of the grammar and head declarations allows one to achieve an adequate compromise between bottom-up propagation and top-down prediction, while avoiding parsing incompleteness (Ritchie, 1999)

As a main guideline for coding the head declarations, the category on the right-hand side (RHS) of a rule whose features determine specific information on the other constituents on the RHS of that rule should be chosen as head. The heads declared in the grammar of figure 2 are a good example of the application of this principle. For instance, by choosing the noun as the head of the noun phrase on rule (b), we allow noun arguments (`n_args`) to be selected at once on the basis of `Type`, provided by the noun lexical information.

On the other hand, the analysis of categories such as `n_args` and `v_args` in the example in figure 2 depends strongly on the `Type` information (subcategorization), which is conveyed in a top-down way by the category on the left-hand side of the respective rule. So, no head must be declared for these categories and they are processed top-down. Each of the numbers on the argument position of `n_args` and `v_args` denote a subcategorization class.

DCGs allow constraints in the form of Prolog goals and so does our formalism. However, due to bidirectionality, it is more difficult to determine when constraints should be checked. For this purpose we added two infix operators that state explicitly when a constraint is intended to be checked. These operators are `pre` and `post`. The `pre` operator is used to force the constraint to be checked before a

given literal. The `post`, forces the constraint to occur after the literal it affects. For instance,

NT `post` {Const}

means that Const must be checked only after NT has been identified and is independent of the direction (left-to-right or right-to-left) the parser is following at the moment. The same applies to

NT `pre` {Const}

except that in this case the checking of the constraint takes place before the identification of the non-terminal NT.

The grammar specified as described above is translated into DyALog bidirectional grammar notation and is then compiled into LPDA transitions. The translation process is described in detail in the following subsection.

5.2. Translating DCG with head declarations into DyALog

5.2.1. *DyALog bi-directional grammar notation*

DyALog automatically builds a parser from a DCG specification. The problem is that DCGs do not say which constituents on the right-hand side of rules should be analyzed first. By default, parsing starts always from the left hand side constituent of a rule. To overcome this limitation, DyALog DCG notation was extended with two operators (`+>` and `<+`) that may be used instead of commas (,) to separate the constituents in the right hand side of a rule. They indicate the order in which constituents must be analysed. For instance, the rule

```
a --> b <+ c +> d +> e
```

states that constituent `c` must be analysed before `b` and `d`, and that `d` must be analysed before `e`.

This is a procedural rather than a declarative notation, so we built a translator in order to produce a grammar using these operators from a specification in our DCG extended with head declarations. For the example in figure 2, the result of the translation is the DyALog grammar in figure 3.

```
s --> np <+ vp.
np --> det <+ n(Type) +> n_args(Type).
vp --> v(Type) +> v_args(Type).
pp --> prep <+ np.
n_args(0) --> [].
n_args(1) --> pp.
v_args(0) --> [].
v_args(1) --> np.
v_args(2) --> np +> pp.
v_args(3) --> pp.
```

Figure 3. Translation of the grammar on figure 2

5.2.2. *Translator implementation details*
The translator works in two passes: the first does most of the work, converting the declarative grammar notation into an internal representation; the second produces the final DyALog program, from the internal representation, resolving references that were not possible to determine in the first pass.

The first pass is used for:

- determining, for each grammar rule, which are the constituents on its right-hand side that should be analyzed from right to left (at the left of a head) and which ones should be analyzed from left to right (to the right of the head). For instance, for rule (b) of the grammar in figure 2, `det` must be analyzed from right to left and `n_args` from left to right.

- for rules not containing syntactic heads in their right-hand side, the previous operation is unaltered, in the sense that all literals on the right-hand side are assumed as being to the left of a (non-existent) head. These rules are worked further on the second pass. Example - rules (e) to (j) in the grammar of figure 2.

- marking, for each rule, its left-hand side non-terminal and its head (if present) as outputting categories, and establishing a subsumption relation between them (the left-hand side subsumes the head). This relation will be used to avoid the output of partial parses subsumed by others. In the example grammar of 2, `s` subsumes `vp`, `np` subsumes `n`, `vp` subsumes `v` and `pp` subsumes `np`.

- identifying pre-terminal categories. To match pre-terminals with the previously POS-tagged input text, dummy terminal rules for

each pre-terminal in the grammar must be added. For instance, the rules

```
det --> [det].
n(Type) --> [n(Type)].
v(Type) --> [v(Type)].
prep --> [prep].
```

must be added to the grammar in figure 2.

- dealing with `pre` and `post` constraints, placing them in the correct order, depending on the direction that part of the rule must be analyzed (left-to-right or right-to-left).
- signaling syntactic errors:
  - rules with 2 or more heads.
  - non-terminals that are defined with both top-down and bottom-up rules.
  - incompleteness of the bidirectional notation (Ritchie, 1999) (this may be just a warning).

The second pass does the following:

- generate two versions of headless non-terminals (except, of course, pre-terminals), for left-to-right and right-to-left usage. The rules for each version will differ only on the order of analysis of the constituents. For instance, rule (i) in the grammar of figure 2 is replaced by the following two rules:

```
v_args_left(2) --> np +> pp.
v_args_right(2) --> np <+ pp.
```

Of course, only `v_args_left` is needed, because, in the grammar, `v_args` always appears to the right of the head. The translated grammar in figure 3 reflects this, but in the real implementation we don't bother to eliminate the unused new non-terminal, because it does not affect parsing performance.

- change references to non-terminals split with the previous operation, in the right-hand side of the grammar rules to the left or

right version, whichever the direction the parser should follow at that point. For instace, this generates, for rule (b) of the grammar on figure 2, the translated rule

```
np --> det <+ n(Type) +> n_args_right(Type).
```

– transfer the definitions of predicates called in DCG constraints, to the output file, with no alteration. Not being strictly part of the grammar, these predicates must nevertheless be copied to the output, so that the parser can use them.

– determine non-terminal modulation and produce the respective `dcg_mode` declarations, to make the parser follow the head-first bottom-up and top-down strategy defined by the grammar. For our example grammar of figure 2, the following directives are needed:

```
:-dcg_mode([s/0,np/0,vp/0,pp/0], -, -, -).
:-dcg_mode([det/0,n/1,v/1,prep/0], -, -, -).
:-dcg_mode([n_args_left/1,v_args_left/1], +(+), +, -).
:-dcg_mode([n_args_right/1,v_args_right/1], +(+), -, +).
```

This last directive is not really needed, for the same reason mentioned above. It does not harm, though. `n_args` and `v_args` are analyzed top-down, so the respective modulation is `+`. As for the other non-terminals, they are analysed bottom-up from their heads, so the corresponding modulation is `-`. Left and right positions are modulated according to the horizontal direction of parsing. For the bottom-up non-terminals, left and right positions are both modulated `-`, since neither of them is known *a priori*.

6. Partial parsing with Bound-Movement Grammar

6.1. Bound-Movement Grammar definition

BMG is the formalism used in level 3 of our parsing architecture. It is an extension of Extraposition Grammars (XG) (Pereira, 1981), used to describe movement of constituents out of their canonical position. XG implies the use of a stack where moved constituents are stored until they are anchored at their canonical position (i.e., at the moment its

trace is found). Being a declarative formalism, XG hides this procedural aspect behind the following rule syntax:

$$NT1 \ldots NT2 \longrightarrow RHS$$

This rule states that NT2 is a moved constituent (described by the sequence of constituents RHS - often NT2 and RHS are the same) that needs to be anchored later in the deep structure of the sentence.

However, the use of just one movement stack is restrictive, because there are several independent types of movement: in interrogative sentences, relative clauses, topicalization, clitics, clefting, etc. BMG extends the XG formalism by using several stacks for the various kinds of movement. The three dots (...) operator is replaced with `int`, `rel` or `slash`, according to the type of movement involved (interrogative sentences, relative clauses or topicalization, respectively - we will not deal with clitics and clefting in this paper).

The other innovation in BMG is the possibility of using barriers called island operators that may prefix constituents in the RHS of a rule. Island operators prevent movement, of all or some types, in and out of the affected constituent. For instance, the `isl_slash` operator in the BMG rule

```
s ⟶ isl_slash np, vp
```

prevents the movement of topicalized material into the np, i.e., no topicalized material can find its trace inside that noun phrase. Simultaneously, no topicalized material recognized during the parse of this noun phrase can find its trace outside that noun phrase. Island operators can be composed, resulting in a non-terminal prefixed by two or more island operators.

Figure 4 shows an example of a BMG illustrating the concepts just introduced. Remember that this grammar just becomes active after the text has been chunked by level 2 of the parsing architecture we propose. The directives in the first lines of the grammar specification define BMG parameters and will be explained in the following sub-section. Rules (a), (b) and (c) expand a sentence into a noun phrase followed by a verb phrase, or into some topicalized or interrogative phrase (`wh`) followed by a sentence. Obviously, the moved material must find its trace inside the second sentence constituents. Rules (d) and (e) define two types of topicalized material, to be pushed into the `slash` stack, for later anchoring. Through rule (f), an interrogative noun phrase is pushed into the `quest` stack. An interrogative prepositional phrase is

```
:-bmg_stacks([slash,rel,quest]).
:-bmg_pushable([np/0,pp/0],[quest,rel]).
:-bmg_pushable([v/0,pp/0],[slash]).

s --> isl_slash np, vp.                                  (a)
s --> topicalized_ph, s.                                 (b)
s --> wh, s.                                             (c)

topicalized_ph slash v(Type) --> isl v(Type).            (d)
topicalized_ph slash pp --> isl pp.                      (e)

wh quest np --> isl int_np.                              (f)
wh quest pp --> isl int_pp.                              (g)

np --> isl np, isl rel_c.                                (h)

rel_c --> [].                                            (i)
rel_c --> rel_head, s.                                   (j)

rel_head rel np --> isl rel_np.                          (k)
rel_head rel pp --> isl rel_pp.                          (l)

vp --> isl_rel isl_quest v(Type), v_args(Type).          (m)

v_args(0) --> [].                                        (n)
v_args(1) --> np.                                        (o)
v_args(2) --> np, pp.                                    (p)
v_args(3) --> pp.                                        (q)

pp --> isl prep, isl np.                                 (r)
```

Figure 4. A small Bound Movement Grammar

pushed in a similar way in rule (g). Rule (h) attaches a relative clause to an already found noun phrase. Relative clauses are defined in rules (i) and (j) as empty or as a sentence preceded by a relative clause head. Typically, the relative clause head is to be pushed into the `rel` stack, according to the rules (k) and (l) and anchored inside the sentence that forms the rest of the relative clause. Rules (m) to (r) define verb phrases, verb arguments and prepositional phrases in the same way as in figure 2.

Island operators are employed to prevent moved material in and out of the affected phrases. Rule (a), for instance, prevents material from the `slash` stack to find its trace in the shielded noun phrase, and, conversely, it prevents that linguistic material inside the noun phrase to be pushed into the `slash` stack. Phrases prefixed with the `isl` operator are not allowed to import or export moved material from/to any stack. A composition of operators `isl_rel` and `isl_quest` is used in rule (m), so that no material from the `rel` and `quest` stacks can be used as a trace of a verb.

To illustrate the use of BMG, we present an example (Example (1)) of the application of the grammar on figure 4 over a Portuguese sentence:

Entre as obras sobressai um romance
Among the works stands out a novel
A novel stands out among the works

. The use of topicalization in this sentence is adequately described by the BMG grammar. After the pre-processing phases (levels 0 and 1 - see section 3), we obtain the following categories for the words in the sentence (we assume that verb *sobressair* (to stand out) is classified in the lexicon has having subcategorization type 3 and nouns *obras* (works) and *romance* (novel) as having type 0):

Entre (`prep`) *as* (`det`) *obras* (`n(0)`) *sobressai* (`v(3)`) *um* (`det`) *romance* (`n(0)`)

After the parser on level 2 is applied, we obtain the following chunked sentence:

Entre as obras (`pp`) *sobressai* (`v(3)`) *um romance* (`np`)

Because of the moved material, the parser on level 2 is unable to parse the whole sentence. That is a job for level 3, that, by using the

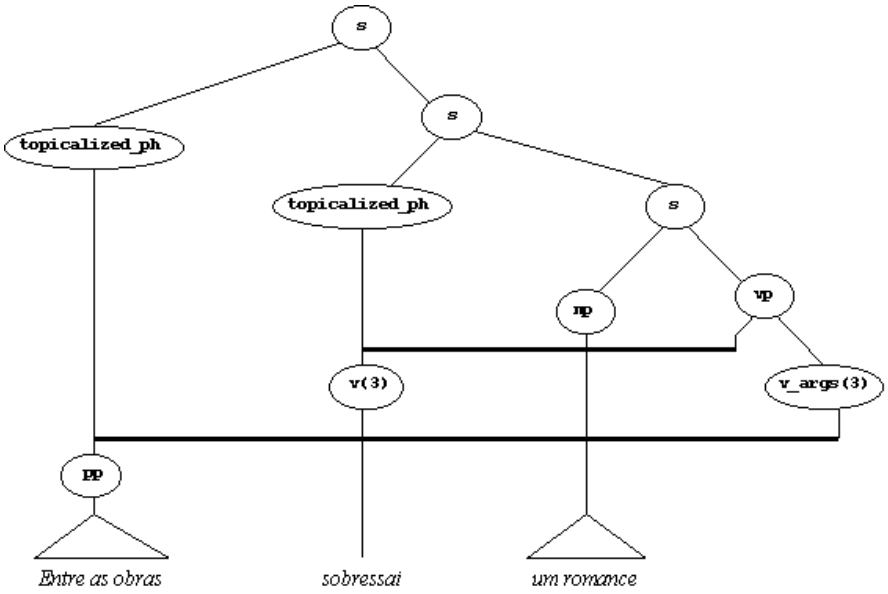

Figure 5. Parsing tree generated by the BMG parser

BMG, will completely and correctly parse the sentence, outputting the tree on figure 5. To emphasize movement issues, we excluded the subtrees for the chunks identified on level 2. The bold black lines represent links from moved constituents to their respective traces.

6.2. BMG compiler implementation

6.2.1. *Parametrizing BMG*

BMG required the implementation of new DyALog directives to declare and define the set of movement stacks, island operators, and the constituents that may be pushed and popped into/from these stacks. The DyALog compiler has then been slightly extended to take into account the usage of the movement stacks.

The BMG in figure 4 will help us to introduce these different directives and operators.

`bmg_stacks`/1 is a directive used to introduce the set of constituent stacks, here named "`slash`", "`rel`" and "`quest`".

Introducing a stack also implicitly defines an infix push operator with the same name used to push a constituent on this stack when reading some non-terminal (i.e., *int* `quest` *np* states that *np* should be pushed on `quest` whenever a non-terminal *int* is recognized).

Several other prefix operators are also defined to set some island constraints. These operators are

- `isl` island constraint for all stacks
- `isl_<name>` for every declared stack `<name>`.

`bmg_isl`/2 is a directive that may be used to name compound island constraint operators for convenience. For instance,

$:-$`bmg_isl(isl_relquest,[rel,quest])`

declares an island constraint operator `isl_relquest` for stacks `rel` and `quest` that can be used instead of `isl_rel isl_quest`.

`bmg_pushable`/2 is a directive used to specify the kind of constituents that may appear on a given stack. For instance,

$:-$`bmg_pushable`$([v/0, pp/0], [slash])$

specifies that only atoms built on predicates $v/0$ and $pp/0$ may appear on stack `slash`.

6.2.2. *Expansion*

In traditionnal Prolog implementation, DCG clauses are expanded into Horn clauses by adding two extra arguments to non-terminals and terminals denoting left and right positions in the input string.

For instance, the DCG clause

```
s --> np,vp
```

gives rise to the Prolog clause

```
s(P0,P2) :- np(P0,P1),vp(P1,P2)
```

DyALog works similarly, except that rule expansion takes place during the compilation process into LPDA (there is no creation of new clauses).

A similar expansion mechanism is used for BMG non-terminals and terminals. Besides the left and right positions, we add, for each constituent stack $\mathcal{S}$, two additionnal arguments yielding the value of $\mathcal{S}$ just left and right of the (non-)terminal. Here again, in DyALog, this expansion takes place during the compilation process.

Table II. Expansion of a BMG non-terminal.

BMG non-terminal	nt
Expanded	$nt(L, R, SL, SR, RL, RR, QL, QR)$
Call	$dcg\_call\_nt\_0(L, SL, RL, QL)$
Return	$dcg\_return(R, SR, RR, QR)$

Therefore, assuming movement stacks $\{\texttt{slash}, \texttt{rel}, \texttt{quest}\}$, a BMG non-terminal nt expands into

$$nt(LP, RP, LSlash, RSlash, LRel, RRel, LQuest, RQuest).$$

Given some (terminal or non-terminal) BMG constituent A, we note $A[\vec{L}; \vec{R}]$ the expanded value of A with the tuple $\vec{L} = (lp, ls_1, \ldots, ls_n)$ (resp. $\vec{R} = (rp, rs_1, \ldots, rs_n)$) holding the left (resp. right) values of position and movement stacks.

6.2.3. *BMG modulation*
A BMG non-terminal inherits any defined DCG modulation (relative to its standard arguments as well as to its left and right position arguments). Although modulation could theoretically be extended to handle left and right stack arguments, this is not yet implemented. The default choice is to use left (resp. right) stack arguments in call (resp. return) atoms (see table II).

We note $\mathcal{C}A[\vec{L}; \vec{R}]$ (resp. $\mathcal{R}A[\vec{L}; \vec{R}]$) the call (resp. return) atom associated to a BMG non-terminal A with current left (resp. right) position and stack values tuple $\vec{L}$ (resp. $\vec{R}$).

6.2.4. *Compilation*
We compile a grammar clause by clause. A BMG clause is denoted

$$\gamma_k : A_{k.0} - - > A_{k.1}, \ldots, A_{k.n_k}.$$

We introduce a set of predicates $\nabla_{k.i}$ that denote intermediary computation points during the refutation of clause γ_k. Their arguments are used to store the values of the variables of γ_k that are pertinent for the rest of the refutation or for publishing.

Actually, these predicates are used to express more easily the different steps of the resolution strategy but are not strictly needed: they can be avoided by using a heavier continuation style notation for the transitions, that we will not detail here.

The following resolution strategy extends the Prolog "modulated Call/Return strategy" used in DyALog (Clergerie and Lang, 1994). Besides the traditional steps **[Call, Return, Select, Publish]**, there is a **Scan** step to read terminals (also present in DCG) and a **Discharge** step to use constituents already pushed into any movement stack. We consider in this strategy that modulation may apply to the movement stacks (even if it is not yet implemented).

The **Call** step is used to push a **call** atom into the LPDA stack when a non-terminal comes in line to be analyzed. **Return** proceeds to the next non-terminal when the analysis of the current one has been completed. **Select** selects a clause to recognize a non-terminal and **Publish** produces a **return** atom after recognizing a whole clause. In the case of BMG these steps now take into account movement and island operators.

We use the notation introduced in section 4.1 for LPDA transitions and make the following abbreviations: $C_{k.i}[\vec{L};\vec{R}] = \mathcal{C}A_{k.i}[\vec{L};\vec{R}]$ and $R_{k.i}[\vec{L};\vec{R}] = \mathcal{R}A_{k.i}[\vec{L};\vec{R}]$. $\vec{A}^j$ refers to the element in position j of vector $\vec{A}$. In the case of the $\vec{L}$ and $\vec{R}$ vectors above, $j > 0$ refers to the movement stacks used, while $j = 0$ refers to the input string. Also, the set of defined movement stacks is denoted as $\{\mathbf{s_1}, \ldots, \mathbf{s_n}\}$. Note that the symbols we introduce to denote predicates and arguments, except $\nabla_{k.i}$, are not logical terms themselves. They are second-order symbols corresponding to the actual predicates and arguments in the program, only used for the purpose of the resolution steps exposition.

Scan Reads a terminal from the input. $A_{k.i+1}$ is a terminal. No action on any stack.

$$SWAP : \nabla_{k.i}[\vec{A};\vec{B}] \mapsto \nabla_{k.i+1}[\vec{B};\vec{C}]$$

where $\vec{A}^0 = [A_{k.i+1}|\vec{B}^0]$ and $\vec{A}^j = \vec{B}^j$ for $j > 0$.

Discharge Discharging a pushed non-terminal in place of non-terminal $A_{k.i+1}$ from stack u (if $A_{k.i+1}$ belongs to the set of pushable non-terminals on stack u).

$$SWAP : \nabla_{k.i}[\vec{A};\vec{B}] \mapsto \nabla_{k.i+1}[\vec{B};\vec{C}]$$

where $\vec{A}^u = [A_{k.i+1}|\vec{B}^u]$ and $\vec{A}^j = \vec{B}^j$ for $j \neq u$.

Call Calling a non-terminal $A_{k.i+1}$ prefixed with island constraints that prevent movement in the stacks belonging to the subset **Is** $\subseteq \{s_1, \ldots, s_n\}$.

$$PUSH : \nabla_{k.i}[\vec{A};\vec{B}] \mapsto C_{k.i+1}[\vec{C};\vec{D}]\nabla_{k.i}[\vec{A};\vec{B}]$$

where $\vec{C}^j = \vec{A}^j$ and $\vec{D}^j = \vec{B}^j$ if $j \notin$ **Is** and $\vec{C}^j = \vec{D}^j = []$ and $\vec{A}^j = \vec{B}^j$ otherwise.

Return Return from a recognized non-terminal $A_{k.i+1}$ prefixed with island constraints that prevented movement in the stacks belonging to the subset **Is** $\subseteq \{s_1, \ldots, s_n\}$.

$$POP : R_{k.i+1}[\vec{C};\vec{D}]\nabla_{k.i}[\vec{A};\vec{B}] \mapsto \nabla_{k.i+1}[\vec{B};\vec{E}]$$

where $\vec{C}^j = \vec{A}^j$ and $\vec{D}^j = \vec{B}^j$ if $j \notin$ **Is** and $\vec{C}^j = \vec{D}^j = []$ and $\vec{A}^j = \vec{B}^j$ otherwise.

Select Selecting clause γ_l to recognize a non-terminal and (if a movement operator is present) push a non-terminal N on stack u.

$$SWAP : \ C_{l.0}[\vec{A};\vec{B}] \mapsto \nabla_{l.0}[\vec{C};\vec{D}]$$

where $\vec{C}^j = \vec{A}^j$ and $\vec{D}^j = \vec{B}^j$ if $j \neq u$ and $\vec{C}^u = \vec{D}^u = []$ and $\vec{B}^u = [N|\vec{A}^u]$.

Publish Publishing some information about a recognized rule γ_l that (if a movement operator is present) pushes a non-terminal N on stack u.

$$SWAP : \ \nabla_{l.n_l}[\vec{C};\vec{D}] \mapsto R_{l.0}[\vec{A};\vec{B}]$$

where $\vec{C}^j = \vec{A}^j$ and $\vec{D}^j = \vec{B}^j$ if $j \neq u$ and $\vec{C}^u = \vec{D}^u = []$ and $\vec{B}^u = [N|\vec{A}^u]$.

To illustrate the operation of the resolution strategy just presented, we again pick example (1). We consider that it is already chunked (see section 6) as a `pp-v(3)-np` sequence. So these categories are now terminal symbols (and may be subject to a **Scan** step). Table III shows the sequence of resolution steps needed to correctly parse the sentence using the grammar on figure 4.

Table III. Resolution steps needed to parse the sequence `pp-v(3)-np`.

Resolution step	LPDA stack	`slash` stack
	$\mathcal{C}$`s`	
Select (rule 2)	$\nabla_{2.0}$	
Call	$\mathcal{C}$`topicalized_ph` $\nabla_{2.0}$	
Select (rule 5)	$\nabla_{5.0}$ $\nabla_{2.0}$	
Scan (`pp`)	$\nabla_{5.1}$ $\nabla_{2.0}$	
Publish	$\mathcal{R}$`topicalized_ph` $\nabla_{2.0}$	`pp`
Return	$\nabla_{2.1}$	`pp`
Call	$\mathcal{C}$`s` $\nabla_{2.1}$	`pp`
Select (rule 2)	$\nabla_{2.0}$ $\nabla_{2.1}$	`pp`
Call	$\mathcal{C}$`topicalized_ph` $\nabla_{2.0}$ $\nabla_{2.1}$	`pp`
Select (rule 4)	$\nabla_{4.0}$ $\nabla_{2.0}$ $\nabla_{2.1}$	`pp`
Scan (`v(3)`)	$\nabla_{4.1}$ $\nabla_{2.0}$ $\nabla_{2.1}$	`pp`
Publish	$\mathcal{R}$`topicalized_ph` $\nabla_{2.0}$ $\nabla_{2.1}$	`v(3) pp`
Return	$\nabla_{2.1}$ $\nabla_{2.1}$	`v(3) pp`
Call	$\mathcal{C}$`s` $\nabla_{2.1}$ $\nabla_{2.1}$	`v(3) pp`
Select (rule 1)	$\nabla_{1.0}$ $\nabla_{2.1}$ $\nabla_{2.1}$	`v(3) pp`
Scan (`np`)	$\nabla_{1.1}$ $\nabla_{2.1}$ $\nabla_{2.1}$	`v(3) pp`
Call	$\mathcal{C}$`vp` $\nabla_{1.1}$ $\nabla_{2.1}$ $\nabla_{2.1}$	`v(3) pp`
Select (rule 13)	$\nabla_{13.0}$ $\nabla_{1.1}$ $\nabla_{2.1}$ $\nabla_{2.1}$	`v(3) pp`
Discharge	$\nabla_{13.1}$ $\nabla_{1.1}$ $\nabla_{2.1}$ $\nabla_{2.1}$	`pp`
Call	$\mathcal{C}$`v_args(3)` $\nabla_{13.1}$ $\nabla_{1.1}$ $\nabla_{2.1}$ $\nabla_{2.1}$	`pp`
Select (rule 17)	$\nabla_{17.0}$ $\nabla_{13.1}$ $\nabla_{1.1}$ $\nabla_{2.1}$ $\nabla_{2.1}$	`pp`
Discharge	$\nabla_{17.1}$ $\nabla_{13.1}$ $\nabla_{1.1}$ $\nabla_{2.1}$ $\nabla_{2.1}$	
Publish	$\mathcal{R}$`v_args(3)` $\nabla_{13.1}$ $\nabla_{1.1}$ $\nabla_{2.1}$ $\nabla_{2.1}$	
Return	$\nabla_{13.2}$ $\nabla_{1.1}$ $\nabla_{2.1}$ $\nabla_{2.1}$	
Publish	$\mathcal{R}$`vp` $\nabla_{1.1}$ $\nabla_{2.1}$ $\nabla_{2.1}$	
Return	$\nabla_{1.2}$ $\nabla_{2.1}$ $\nabla_{2.1}$	
Publish	$\mathcal{R}$`s` $\nabla_{2.1}$ $\nabla_{2.1}$	
Return	$\nabla_{2.2}$ $\nabla_{2.1}$	
Publish	$\mathcal{R}$`s` $\nabla_{2.1}$	
Return	$\nabla_{2.2}$	
Publish	$\mathcal{R}$`s`	

7. Experimental results

In order to test the partial parsing machinery described in this paper, we randomly selected 30 previously POS-tagged texts from the LUSA news corpus (40 million words) and partially parsed them with our system. The texts had on average 1765.9 words each.

7.1. Comparison between standard chart parsing and DyALog-based parsing

The first experiment was made to assess the performance improvement by using DyALog with its built-in tabulation. Due to the imprecision in measuring small time intervals, we discarded 1 and 2-word sentences from the sample[5]. Thus, we used 556 sentences, with a total of circa 50,000 words, for this comparison. The equipment used was a Pentium PC at 200 MHz with 64Mb of RAM running Red Hat Linux, version 5.2. The results were 1.66 words per second when a standard chart parser was used, and 254 words per second when DyALog was used. The reason for such a dramatic improvement has to do with the complexity of dealing with chart structures in a standard chart parser. DyALog does a much more efficient management of those structures.

7.2. Comparison among different parsing strategies

The performance of different parsing strategies was measured not in terms of time, but in terms of generated chart edges. This is due to higher precision and unvariability of this measure, and its independence of the actual implementation. So, we used the chart parser in Prolog for this experiment, since we had better control over the chart structure.

The strategies tested were standard bottom-up (left-to-right) and a mixed strategy (top-down and bottom-up) in two variants, left-to-right and bidirectional (left-to-right and right-to-left, head-first). This last strategy uses the full potential of DCG extended with head declarations.

The input sample and equipment used were the same as for the previous subsection. The results obtained are quantified in table IV, in terms of generated chart edges per input word.

As can be seen from these numbers, the head-first approach mildly improves the left-first strategy. This is mainly because in Portuguese, the constituents to the left of heads (determiners, prepositions, ...) are often syntactically independent of their respective head. However, for languages in which the head appears mostly to the right of its phrase, it could mean a significant improvement. There are also advantages in using the head-first approach for lexical fault diagnosis, since the most

Table IV. Performance results for different parsing strategies

Strict bottom-up left-first	21.07 edges per word
Mixed top-down and bottom-up left-first	18.54 edges per word
Mixed top-down and bottom-up head-first	18.23 edges per word

informative lexical items are heads. When the information associated to heads is wrong or incomplete (which is more probable than on other categories), the use of this strategy simplifies the agenda initialization, since head categories alone trigger the rules they are used in, and no other categories need to be put in the initial agenda.

We can see from the results on table IV that performance is improved by using a mixed strategy (being left-first or head-first). In fact, a correct distribution of information between top-down prediction and bottom-up propagation eliminates many of the ambiguity during parsing.

8. Discussion of the pros and cons of our approach and comparison with related work

The partial parsing approach presented in this paper is purely symbolic. Most partial parsers (chunkers) developed today use example based learning techniques (Daelemans et al., 1999; Skut and Brants, 1998b; Skut and Brants, 1998a) in order to achieve greater performance and accuracy with a minimum effort on grammar coding. However, the languages they work on (mostly English) have already a large amount of computational resources, especially treebanks, available, which provide training data for the statistical models used.

The absence of such computational resources for Portuguese except for raw text poses a greater challenge when building parsing systems. The partial parsing machinery we presented in this paper provides a basis for a symbolic parsing system that does not need to use manually annotated corpora. Moreover, the use of our level 3, based on a Bound-Movement Grammar enables us to deal with recursion and with linguistic movement of constituents which is not at all a problem handled by currently available chunking techniques.

High performance could only be achieved by cascading successive levels of parsing and by using a background tabulation machinery, like DyALog.

Cascaded parsing is used in (Abney, 1996; Hobbs, 1997). These works, however, over-simplify each of the cascading levels to finite-state

automata. Through the use of built-in tabulation (specifically, by using DyALog), we proposed to use the full potentiality of DCG and BMG grammars without loss of efficiency. In fact, the average speed of our system (4 ms per word = 250 words per second) rivals with the class of parsers that Abney calls "skimming parsers", with a range of speeds of 20-50 words per second.

Tabulation has been used for parsing since the Earley algorithm (Earley, 1970) was proposed. Only recently the technique was generalized for logic programming, with advantages for parsing, since the management of the tabulation data structures is built-in and efficiently performed.

The execution model of DyALog allows us to define different parsing strategies, giving us an additional advantage in relation to other tabulation systems (e.g. XSB (Sagonas et al., 1997)), where such strategies cannot be directly defined.

The head-driven mixed strategy followed in parsing level 2 extends head-corner chart parsing strategies (Sikkel and Akker, 1996) by allowing the grammar writer to specify heads in the right hand side of rules only where it makes sense. The idea is to guide the parsing process with as much information as possible. In some rules, this information is located in the heads. In others, the information comes top-down, and defining a head for these rules only increases non-determinism where there is the possibility to avoid it.

On the other hand, the risk of grammar incompleteness (as pinpointed by (Ritchie, 1999)) is present. We assume it and argue that there is always the risk of making mistakes in grammar writing, even for standard top-down CFG or DCG, and this mistake in particular is no worse. The conditions for grammar completeness proposed by (Ritchie, 1999) are reasonable, at least for Natural Language, which follows well-behaved patterns, described by linguistic theories such as X-bar. Generally, there is no need to violate those completeness conditions. As an extra help, (Ritchie, 1999) also suggests an algorithm to automatically decide if a grammar satisfies a sufficient condition for completeness.

As for BMG, they require little effort to be implemented into a logic programming environment already equipped with DCG. Making BMG available in DyALog was a simple matter, through a simple extension of the resolution strategy used.

9. Conclusions, current and future related work

We presented in this paper a flexible partial parsing system based on tabulation techniques. Due to the use of DyALog, a logic programming environment with built-in tabulation, we were able to dramatically improve parsing performance.

The choice of DyALog instead of other similar systems has to do with its execution model, which is especially adequate for parsing tasks in the sense that it is easily adaptable to various grammar formalisms and search strategies.

In particular, it allowed us to implement DCG extended with head declarations and BMG. Many search strategies are possible, and the head-driven bi-directional strategy of the DCG extended with head declarations was easily achieved through modulation.

The work with and on DyALog will continue, with the extension of modulation for the movement stacks and with extensions for fault finding support.

Fault finding representes one of the major uses of the parsing machinery described in this paper. It consists of incrementally re-parsing a sentence on the basis of alternative, hypothetical initial information. If this improves parsing results, then the hypotheses made are considered for further evaluation and eventually become part of the linguistic knowledge. This work will continue, by building new partial parsers, specialized on the diagnosis of various kinds of faults. Recently, by pursuing some experiments on the diagnosis of lack of verb subcategorization information in the lexicon, we obtained some promising results for some selected verbs (*sobressair*, *consentir* and *respirar*) that, on some cases confirm the results obtained by statistically based methods (Marques, 2000), and, on other cases, provide new information, not obtainable by those methods. For instance, by using BMG, we were able to gather evidence that the noun phrase that frequently follows the verb *sobressair* is its subject and not one of its objects, as (Marques, 2000) wrongly acquires, in an automatic way.

The use of a society of diagnosing agents (Lopes and Balsa, 1998; Lopes et al., 1999) together with cross-validation techniques will truly allow a parsing system to evolve by learning new lexical and grammatical information that reduces granularity and improves parsing precision.

The choice of granularity as a measure for filtering partial parses for further consideration is a conscient one, and has to do with the nature of the partial parses obtained at this stage. We prefer to decide correctly on the basis of available (lexical, syntactical) information than to ambiguously output partial parses based on uncertain information, even

if they are less granular. Each of the levels of the cascaded structure follows this principle, always trying to reduce granularity, as possible.

Notes

[1] Meanwhile, due to recent work in our research group on statistical extraction of multi-word lexical units (Silva et al., 1999) some changes may be introduced at this level. However, we will not detail that process in this paper.

[2] These correspond to the extra arguments resulting from standard DCG expansion into Horn clauses.

[3] The real grammar we use at this level has 124 rules.

[4] The head of a phrase is not necessarily unique, when there is more than one rule expanding that phrase. Each of those rules may have associated a different head category. However, the head must be uniquely identified in each rule.

[5] Often, these small sentences result from tokenizing and/or POS-tagging errors

References

Abney, S.: 1996, 'Partial parsing via finite-state cascades'. In: J. Carroll (ed.): *Proceedings of Workshop on Robust Parsing at Eighth Summer School in Logic, Language and Information*. pp. 8–15.

Balsa, J., V. Dahl, and J. Lopes: 1995, 'Datalog Grammars for Abductive Syntactic Error Diagnosis and Repair'. In: *Proceedings of the Fifth International Workshop on Natural Language Understanding and Logic Programming, Lisbon*. pp. 111–125.

Carroll, J., G. Minnen, and T. Briscoe: 1998, 'Can Subcategorization Probabilities Help a Statistical Parser?'. In: *Proceedings of the 6th ACL/SIGDAT Workshop on Very Large Corpora*. Montreal, Canada, pp. 118–126.

Clergerie, E. d. l. and B. Lang: 1994, 'LPDA: Another Look at Tabulation in Logic Programming'. In: V. Hentenryck (ed.): *Proceedings of the 11th International Conference on Logic Programming (ICLP'94)*. pp. 470–486.

Collins, M.: 1997, 'Three Generative, Lexicalised Models for Statistical Parsing'. In: *Proceedings of the European Chapter of the Annual Meeting of ACL*.

Collins, M. and J. Brooks: 1995, 'Prepositional Phrase Attachment through a Backed-Off Model'. In: *Proceedings of the Third Workshop on Very Large Corpora*.

Daelemans, W., S. Buchholz, and J. Veenstra: 1999, 'Memory-based shallow parsing'. In: *Proceedings of the EACL'99 Workshop on Computational Natural Language Learning, Bergen, Norway*. pp. 53–60.

Earley, J.: 1970, 'An Efficient Context-Free Parsing Algorithm'. *Communications of the ACM* **13**(2), 94–102.

Grefenstette, G.: 1994, *Explorations in Automatic Thesaurus Discovery*. Kluwer Academic Publishers. PhD Thesis, University of Pittsburgh.

Hobbs, J. R. e. a.: 1997, 'FASTUS: A Cascaded Finite-State Transducer for Extracting Information from Natural-Language Text'. In: E. Roche and Y. Schabes (eds.): *Finite-State Language Processing*. Bradford Books, pp. 383–406.

Lopes, J., V. Rocio, R. Viccari, and E. Padilha: 1996, 'Bound Movement Grammar for Natural Language Parsing'. In: *Proceedings Second Workshop on Computational Processing of Writen and Spoken Portuguese, Curitiba, Brazil, October 21-22, 1996*. pp. 11–19.

Lopes, J. G. P. and J. Balsa: 1998, 'Overcoming incomplete information in NLP systems - verb subcategorization'. In: F. Giunchiglia (ed.): *Proceedings of Artificial Intelligence: Methods Systems and Applications, 8th International Conference, AIMSA'98, Sozopol, Bulgaria, Proceedings. Lecture Notes on Artificial Intelligence 1480*. pp. 331–340.

Lopes, J. G. P. and V. J. Rocio: 1999, 'An infra-structure for diagnosing causes for partially parsed natural language input'. In: *Proceedings of the 6th International Symposium on Social Communication*. Santiago de Cuba, pp. 550–554.

Lopes, J. G. P., V. J. Rocio, and J. a. Balsa: 1999, 'Superando a incompletude da informação lexical'. In: P. Marrafa and M. A. Mota (eds.): *Linguística Computacional: Investigação Fundamental e Aplicações*. in Portuguese.

Marques, N.: 2000, 'Uma metodologia para a modelação estatística da subcategorização verbal'. Ph.D. thesis, FCT - Universidade Nova de Lisboa.

Marques, N. and J. G. P. Lopes: 1996a, 'Using Neural Nets for Portuguese Part-of-Speech Tagging'. In: *Proceedings of the Fifth International Conference on the Cognitive Science of Natural Language Understanding*.

Marques, N. M. C. and J. G. P. Lopes: 1996b, 'Using Neural Nets for Portuguese Part-of-Speech Tagging'. In: *Proceedings of the Fifth International Conference on The Cognitive Science of Natural Language Processing*. Dublin City University, Ireland.

Pereira, F.: 1981, 'Extraposition Grammars'. *American Journal of Computational Linguistics* **7**(4).

Pereira, F. C. N. and D. H. D. Warren: 1980, 'Definite Clause Grammars for Language Analysis - A Survey of the Formalism and a Comparison with Augmented Transition Networks'. *Artificial Intelligence* **13**, 231–278.

Ramshaw, L. and M. Marcus: 1995, 'Text chunking using transformation-based learning'. In: *Proceedings of the 3rd Workshop on Very Large Corpora, Cambridge, MA, USA*. pp. 62–64.

Ratnaparkhi, A.: 1998, 'Statistical Models for Unsupervised Prepositional Phrase Attachment'. In: *Proceedings of COLING-ACL'98: 36th Annual Meeting of the Association for Computational Linguistics and 17th International Conference on Computational Linguistics, Montreal, Canada*.

Ritchie, G.: 1999, 'Completeness Conditions for Mixed Strategy Bidirectional Parsing'. *Computational Linguistics* **25**, 457–486.

Roth, M. and J. Carroll: 1996, 'Valence induction with a head-lexicalized PCFG'. Technical report, draft at http://www.ims.uni-stuttgart.de/ mats.

Sagonas, K. F., T. Swift, D. S. Warren, J. Freire, and P. Rao: 1997, 'The XSB Programmer's Manual, Version 1.7.1'. Technical report, State University of New York at Stone Brook.

Sikkel, K. and R. O. D. Akker: 1996, 'Predictive Head-Corner Chart Parsing'. In: H. Bunt and M. Tomita (eds.): *Recent Advances in Parsing Technology*. Kluwer, pp. 169–182.

Silva, J., G. Dias, S. Guillore, and G. Lopes: 1999, 'Using LocalMaxs Algorithm for the Extraction of Contiguous and Non-contiguous Multiword Lexical Units'. In: P. Barahona (ed.): *Proceedings of the 9th Portuguese Conference on Artificial Intelligence (EPIA'99), Evora, Portugal. Lecture Notes in Artificial Intelligence, vol. 1695*. pp. 113–132.

Skut, W. and T. Brants: 1998a, 'Chunk Tagger'. In: *Proceedings of the ESSLLI-98 Workshop on Automated Acquisition of Syntax and Parsing*. Saarbrucken, Germany.

Skut, W. and T. Brants: 1998b, 'A Maximum-Entropy Partial Parser for Unrestricted Text'. In: *Proceedings of the 6th Workshop on Very Large Corpora*. Montréal, Québec.

Yeh, A. S. and M. B. Vilain: 1998, 'Some Properties of Preposition and Subordinate Conjunction Attachments'. In: *Proceedings of COLING-ACL'98: 36th Annual Meeting of the Association for Computational Linguistics and 17th International Conference on Computational Linguistics, Montreal, Canada*. pp. 1436–1442.