# Head Corner Parsing for TAG

Gertjan van Noord

Vakgroep Alfa-informatica RUG
Groningen
vannoord@let.rug.nl

**Abstract**

This paper describes a bidirectional head-corner parser for (unification-based versions of) Lexicalized Tree Adjoining Grammars.

Keywords: head-driven parsing, bidirectional parsing, Tree Adjoining Grammar.

## 1 Introduction

Natural language parsing is still inefficient. There are two important reasons for this. Natural language contains many ambiguities which lead to a large search space for parsers. On the other hand the complex structure of natural language is problematic; natural language parsers usually are based on grammars of which the formal power goes beyond the context-free grammars. Both observations imply that efficient algorithms that have been developed for e.g. programming languages, such as LR(1), can not be used as such.

To obtain a parsing algorithm for a class of grammars beyond the context-free grammars, the usual strategy is to generalize an efficient algorithm for context-free grammars to this more powerful class. The problem with this approach is that in the course of the generalization the efficiency may disappear, both in terms of the worst-case and average-case performance.

It might be fruitful therefore to investigate processing techniques which can be motivated from a linguistic point of view. In such a strategy algorithms are developed which are based on the typical properties of the grammars that are actually used. Even if such techniques have a worst-case performance that is comparable to the performance of conventional strategies, it is worthwhile to consider these techniques if they improve upon the average case behavior.

As an example consider head-driven parsing for unification grammars. In such unification grammars the information a constituent contains is usually percolated upward from its *head*. For example, the agreement features of a noun phrase are determined by the head of that noun phrase, the noun. Furthermore a head of a constituent determines what kinds of other constituent it governs. For example, the head of a verb phrase, the verb, determines what kind of objects the verb phrase contains. If the verb is a bi-transitive then the verb phrase might contain two noun phrases. In head-corner parsing this flow of information is mimicked by the parser. Rather than parsing from left to right, the head-corner parser parses from the head outward in both directions. As the parser starts off from the head the information about the constituents to be expected left and right of the head is available in time to reduce search space.

Head-corner parsing has first been suggested as an interesting approach to natural language parsing in Kay (1989). Some results have been obtained both for context-free grammars (Satta and Stock, 1989; Sikkel and op den Akker, 1992), unification grammars (Bouma and van Noord, 1993) and non-concatenative grammatical formalisms (van Noord, 1991; van Noord, 1993).

This paper describes a head-corner parser for Tree Adjoining Grammars. In order to define a head-driven parser for TAGs, it is necessary to define a version of TAGs in which all elementary trees are 'headed' trees. In the next section I define such headed TAGs.

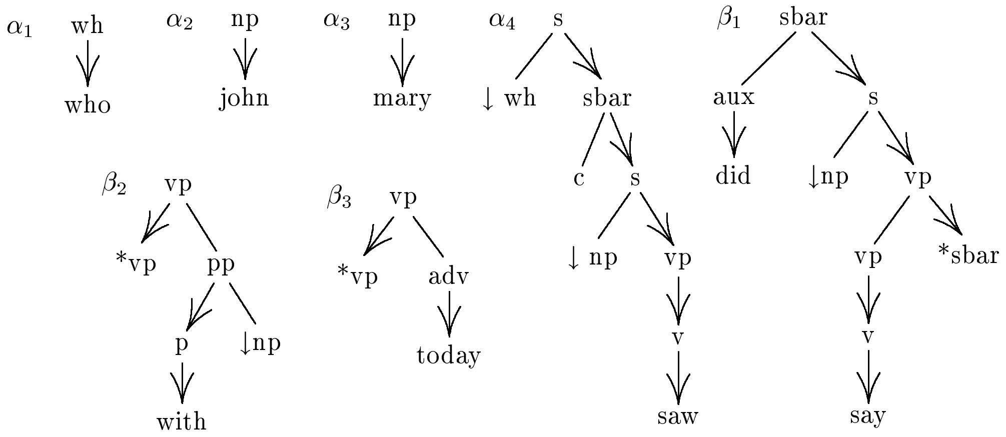


Figure 1: Example of a Headed TAG: $H_1$. The arrows indicate the head of a local tree.

In section 3 I present the head-corner parser for headed LTAGs. The head-corner parser will be presented as a definite clause specification. Such a definite clause program provides an abstract and declarative specification of the parser. It is possible to implement parsers on the basis of this specification.

## 2 Headed TAGs

In many linguistic theories, a 'head' of a construction plays an important role. For example, heads of a construction determine what other parts the construction may have. Furthermore, heads carry the features associated with the construction as a whole (such as case, agreement). The notion 'head' plays an important role in grammatical theories as diverse as Government and Binding, (Xbar theory); Generalized Phrase Structure Grammar (the head-feature convention); and Head-driven Phrase Structure Grammar.

A headed TAG is a TAG in which each elementary tree is a headed tree. As an example of a headed TAG, consider figure 1. For each non-terminal node in a headed tree it is specified which daughter is the head of that tree. In the examples I use arrows to indicate the head-daughter of a local tree. The notion 'head-corner' is defined as the reflexive and transitive closure of the head relation. For example, in $\alpha_4$ the terminal symbol 'saw' is a head-corner of the root node. Similarly, in $\beta_1$, the foot node is a head-corner of the root node.

I define a parser for headed and lexicalized TAGs for which the following two constraints hold:

- the anchor of an initial tree $\alpha$ must be a head-corner of the root node of $\alpha$.
- the foot node of an auxiliary tree $\beta$ must be a head-corner of the root node of $\beta$.

Note that these requirements are satisfied in grammar $H_1$. Clearly, as I have not constrained the notion 'head' in any way, any lexicalized TAG can be transformed into a headed TAG obeying this constraint.

The constraints reflect how I want to use headed TAGs in a head-corner parser. In this parser processing will proceed essentially in an anchor-driven way for initial trees, and in a foot-driven way for auxiliary trees. The notion 'head' generalizes the two cases, and is also meaningful in order to define an order of proceesing for sub-trees of elementary trees that do not contain an anchor or a foot node.

The parser proceeds *anchor-driven* because we want to make use of the fact that elementary trees are lexicalized. The parser proceeds *foot-driven* in order to avoid a problem that otherwise occurs in bidirectional parsers (cf. discussion below).

## 3 The Head-corner Parser for TAG

In this section I present a formulation of the head-corner parser as a set of definite clauses. This provides for an abstract, non-deterministic, characterization of the parser.

Standard techniques can be used to obtain an implementation of the parser on the basis of this specification. For example, one might use SLD resolution with backtracking, or with memo-ization, or Earley-deduction.

Preliminary experiments indicate that using backtracking (as in Prolog) is more efficient than tabular or memo-ization techniques if complex feature-structures are added.

### 3.1 An example

In parsers that proceed from the left the parse predicate typically consists of a category to be parsed, a given start position and an unknown end position. In a bidirectional parser we are given a category, and two pairs of positions: a given pair of indices that indicate the extreme positions between which the category has to be found, and a pair of indices that indicates the exact position of the category once it has been found. Depending on whether we are parsing to the left or to the right, one of the extreme positions matches the actual position.

Suppose that we are given the grammar in figure 1, and the sentence to parse is:

(1) Who did john say mary saw?

The first goal of the parser is to parse a sentence (assume the top-category is `s`) from position 0 to 6 within the extreme positions 0 to 6. In order to parse such a sentence, the head-driven parser proceeds as follows. It *predicts* an initial tree whose root node matches the top category and whose anchor occurs somewhere between position 0 and 6. In the current grammar the initial tree $\alpha_4$ is the only candidate.

The head-corner parser then proceeds by a head-driven bottom-up walk through this initial tree (starting from the head-corner, i.e. the anchor) by trying to recognize the other parts of the initial tree, and by applying adjunctions non-deterministically. In the current example our goal is to *connect* the head-corner `saw` to the top category `s` by climbing up in $\alpha_4$.

During the connect phase there are three possibilities. Firstly, once we arrive at the root of the tree we are done. Secondly, in order to move upward in the tree, we are to recognize the sisters left and right of the current node (the current head). Thirdly, it may also be the case that at the current head node an adjunction occurs. In that case we have to climb from the foot node of the auxiliary tree up to the root node of the auxiliary tree, after which we proceed by climbing up from the node at which this adjunction occurred.

In the current example we proceed by climbing up from `saw` to the dominating `v` and `vp` nodes by two instantiations of the second possibility. As both these nodes are unary branching nothing interesting happens. At this point we want to climb up to the `s` node, but in order to reach that node we have to recognize a substituted `np`. This leads to a recursive call of the parse predicate: we have to parse a `np` within the extreme positions 0 and 5, ending in 5.

In general there are three possibilities to consider in the case of such an embedded parse-goal. In the example the subtree to be recognized simply consists of a substitution node with category `np`. In general such a subtree may be complex. Its head-corner is either a substitution node, a terminal symbol, or an empty element. In each case the parser first recognizes this head-corner (recursively in the first case, and trivially in the second and third case) after which the parser proceeds to connect this embedded head-corner up to the root of the subtree. In the example, the subtree to be recognized consists solely of a substitution node. The head-corner of this subtree therefore is this substitution node. The path from the head-corner upward to the root is the empty path. Therefore the parser proceeds by predicting an initial tree of which the root matches `np`, after which the anchor of that initial tree must be connected to that root. In this case the parser predicts the initial tree $\alpha_3$ and climbs from its anchor to its root without any adjunctions or substitutions. After the recognition of the substitution node np of $\alpha_4$, we now are at the embedded s node of $\alpha_4$ covering positions 4 to 6.

Again we climb up one level, to the sbar node. In order to get there we need to parse an empty c. This provides an example of the second possibility of an embedded parse goal. This succeeds immediately (in general there might be adjunctions at this empty node, but not in this example). Now we are at node sbar of $\alpha_4$. If we were to climb up at this point the tree walk would fail to find a wh category ending in position 4. However, another possibility at the current node is to adjoin auxiliary tree $\beta_1$. If an auxiliary tree $\beta$ is adjoined at a node $\eta$ this implies that we proceed our tree walk from the foot node of $\beta$ up to the root node of $\beta$ after which we return to $\eta$.

In the example we are to climb from the foot node of $\beta_1$ up to its root node. We have covered position 4 to 6 and our extreme positions are 0 and 6. In order to climb up one level we have to parse the subtree rooted by vp between 0 and 4 and ending in 4. We start with the head-corner, the terminal symbol say, after which we have to climb up from this head-corner with positions 3 and 4 up to the vp again. As all this succeeds, we can climb up to the vp node dominating the foot node, now covering positions 3 to 6. Proceeding in this way we parse 'john' as a noun-phrase and consume the terminal symbol 'did' after which we climb up to the root of $\beta_1$, from 1 to 6. Control then returns to the initial tree in which the auxiliary was adjoined. In $\alpha_4$ we are at node sbar from 1 to 6, and now we can climb up successfully because indeed we find a wh from position 0 to 6. The sentence thus is well-formed. This completes the example.

### 3.2 Definite clause specification

To explain the basic ideas behind the head-corner parser for LTAGs I present a definite clause specification of the head-corner parser. Such a definite clause specification can be considered as a *declarative* characterization of the parser. It defines a search space which can then be investigated with different techniques. One can use SLD resolution with backtracking (as in Prolog), possibly with memo-ization and packing, or Earley deduction. A definite clause definition can be seen as an abstract specification of different implementations of such a head-corner parser. It is easy to obtain a worst-case efficiency bound if we view the definite clause specification as an abstract characterization of an Earley deduction system. In that case, the number of possible instantiations of a clause gives an approximation of the upper-bound for the computational resources needed in the worst-case.

The basics of the parser are very simple. A tree to be recognized by the parser is represented as a pair consisting of the head-corner and a list of triples. The list of triples is the chain of nodes from the head-corner up to the root. Each triple consists of a head, a list of trees representing the daughters of this node left of its head, and a list of trees representing the daughters of this node right of its head.

```
t(Term,Chain).    % for tree with terminal symbol as head-corner
e(Cat, Chain).    % for tree with empty string as head-corner
s(Subs,Chain).    % for tree with substitution node as head-corner
```

As an example, consider initial tree $\alpha_4$ of figure 1. The head-corner of this tree is the terminal node saw. The chain of nodes consists of the nodes v, vp, s, sbar and the root node s. This tree therefore is represented as follows:

```
t(saw,[c(v   ,[]          ,[]),
       c(vp  ,[]          ,[]),
       c(s   ,[s(np,[])],[]),
       c(sbar,[e(c ,[])],[]),
       c(s,   [s(wh,[])],[])
      ]).
```

Note that this is a simple example in the sense that each of the non-head daughters of the tree is a non-complex tree. In general however chains appear recursively within chains.

The parse predicate identifies three cases depending on the nature of the head-corner of the tree to be recognized. The head-corner is either a substitution node, a terminal symbol, or an empty element.

In the first case the parser predicts an initial tree whose root node matches the substitution node (the `ini` predicate). The chain of nodes needed to connect the anchor of this initial tree to its root node, and the chain of nodes from the substitution node up to its root node are both traversed via the `hc_na` predicate.

In the second case (head-corner is a terminal symbol) the parser checks whether there is indeed a terminal symbol in a possible position (i.e. between the current extreme positions). The chain of nodes to connect the head-corner to the root is then traversed via the `hc_na` predicate.

In the third case (head-corner is empty element) the parser selects a position where the empty element might occur and continues to traverse the chain via the `hc` predicate.

```
goal(s(Subs,Chain),P0,P,E0,E) :-
    ini(Subs,Chain0,Word,Q0,Q),                % select initial tree
    between(Q0,Q,E0,E),                        % in appropriate position
    hc_na(Chain0,lex(Word),Mid,Q0,Q,R0,R,E0,E), % recognize initial tree
    hc_na(Chain,Mid,_,R0,R,P0,P,E0,E).         % proceed with Chain
goal(l(Word,Chain),P0,P,E0,E) :-
    connects(Word,Q0,Q),                       % select word in string
    between(Q0,Q,E0,E),                        % in appropriate position
    hc_na(Chain,lex(Word),_,Q0,Q,P0,P,E0,E).   % proceed with Chain
goal(e(Cat,Chain),P0,P,E0,E) :-
    between(Q,Q,E0,E),                         % gap in appr. position
    hc(Chain,Cat,_,Q,Q,P0,P,E0,E).             % proceed
```

The predicates `hc` and `hc_na` both define a head-driven bottom-up tree walk. The difference is that in the latter case no adjunctions are possible at the current node. Note that this predicate is used if the current node is a terminal symbol. [1]

The predicate `hc` identifies two cases. Either no adjunction occurs, or an adjunction does occur. In the first case the predicate `hc_na` is called (the predicate `unify_node` is relevant only in the case of feature-structures, cf. below). If an adjunction is possible then an appropriate auxiliary tree is selected. The chain of nodes of this auxiliary tree is traversed first. After that we continue to traverse the current chain of nodes. [2]

The predicate `hc_na` succeeds trivially in case the chain is empty. Otherwise it takes the first triple of the chain, recognizes each of the daughters left and right of the current head and proceeds with a head-driven traversal of the tail of the chain.

```
hc(Chain,Cat,Goal,Q0,Q,P0,P,E0,E) :-
    unify_node(Cat),
    hc_na(Chain,Cat,Goal,Q0,Q,P0,P,E0,E). % proceed without adjunction
hc(Chain,Cat,Goal,Q0,Q,P0,P,E0,E) :-
    aux(Cat,Chain0,_,E0,Q0,Q,E),          % select aux, anchor in E0-Q0 or Q-E
    hc_na(Chain0,_,Mid,Q0,Q,R0,R,E0,E),   % recognize aux, no adjs. at foot
    hc_na(Chain,Mid,Goal,R0,R,P0,P,E0,E). % proceed

hc_na([],G,G,P0,P,P0,P,_,_).              % finish at root
hc_na([t(Cat,L,R)|Chain],_,Goal,Q1,Q2,P0,P,E0,E) :-
    goal_l(L,Q0,Q1,E0),                   % recognize material left of head
```

[1] By using this predicate for the remaining chain of the substitution node we prevent spurious ambiguities that would otherwise occur: adjoining at a node after or before a substitution results in the same derived tree. This parser never adjoins at substitution nodes: the same effect is obtained by adjoining at the root node of the tree to be substituted.

[2] By using the predicate `hc_na` for the chain of the auxiliary tree we generally forbid adjunctions at foot nodes in order to prevent massive undesired ambiguities.

```
    goal_r(R,Q2,Q,E),                      % and right of head
    hc(Chain,Cat,Goal,Q0,Q,P0,P,E0,E).     % proceed with mother
```

To parse a list of trees the predicates `goal_l` and `goal_r` are defined in order to parse to the left or to the right respectively. In order to simplify the first predicate it is assumed that the daughters left of a head are represented in a right-to-left order. Note the use of the indices that represent extreme positions.

```
goal_l([],Q,Q,_).
goal_l([H|T],Q0,Q,E0):- goal(H,Q1,Q,E0,Q), goal_l(T,Q0,Q1,E0).

goal_r([],Q,Q,_).
goal_r([H|T],Q0,Q,E):-  goal(H,Q0,Q1,Q0,E), goal_r(T,Q1,Q,E).
```

If a sentence is grammatical, then, after selecting possible initial and auxiliary trees from the lexicon, the following goal should succeed (where `TopCat` should represent the top category, and `End` the length of the sentence):

```
goal(s(TopCat,[]),0,End,0,End).
```

This completes the definite clause description of the head-corner parser for lexicalized TAGs.

**Feature structures** The head-corner parser straightforwardly deals with feature structures, to be represented here as first-order terms. The details are hidden in the definitions of the predicates `aux` and `unify_node`. The latter predicate simply unifies the bottom and top parts of a node. This predicate is called for some node, therefore, once it is known that no adjunctions are possible at that node. In the predicate `aux` the appropriate unifications are carried out with regard to the foot and root node of the auxiliary tree and the target node of the adjunction.

Note that it is assumed that adjunction constraints are all expressed via constraints on feature structures. Also note that adjunctions at foot nodes are generally disallowed (although it would not be difficult to allow such adjunctions if there were linguistic motivation to do so).

**Derived trees and derivation trees** The head-corner parser can be easily extended to deliver derived trees, and/or derivation trees. It is not difficult to extend the parser with a mechanism to obtain structure sharing in the case of ambiguities (packing). The implementation of the head-corner parser in the appendix indeed uses packing, and produces items on the basis of which derivation trees are recovered.

**Efficiency** It is possible to obtain an efficient parser for LTAGs on the basis of the specification given above. Rather than using 'chains', use a pointer to implement the tree walk. As there is only a polynomial number of different instantiations of each of the predicates, it is possible to obtain a polynomially bounded implementation, for example by using Earley deduction.

In fact, if we remove from all of the clauses the pair of arguments that represent the extreme positions, then it is quite easy to see that an $O(n^6)$ parser can be obtained (note that this does not affect the correctness of the algorithm, as the extreme positions are used as filter only). This can be seen if we regard the definite clauses above as inference rules for an Earley-deduction system. The most complex inference rule—the second clause of `hc`—then has 6 position markers, the other arguments are all bounded by the size of the grammar.

The version that we presented above—including a pair of markers indicating the extreme positions—has an obvious implementation with a $O(n^7)$ worst-case bound (as one of the extreme positions `E0, E` is always the same as one of the resulting positions `P0,P`), because we are parsing either to the left or to the right from a given source, we have at most $c \cdot n^7$ instantiations), but can be turned into an $O(n^6)$ chart parser with some additional effort. This effort consists in representing 'goals' as separate items, along the lines of Sikkel and op den Akker (1993).

If we consider feature-structures then a polynomial worst-case bound can only be obtained by placing some (arbitrary) limit on the size of feature-structures. But in such cases it is far from clear whether chart-based techniques lead to practical benefits. In my experience, a backtracking scheme, possibly enriched with memo-ization of a few well-chosen predicates (in this case the `goal` predicate), is in such cases much more practical.

## 4 Discussion

I presented an abstract specification of a head-corner parser for *headed* and lexicalized TAGs. The important properties of the parser are that processing proceeds in a *bidirectional* fashion in two senses: the string is not derived from left to right, but rather from heads outward both to the left and to the right. Furthermore the parser employs both bottom-up and top-down information. For this reason it fully uses the benefits of lexicalized grammars.

The parser has some similarities with the bidirectional parser of Lavelli and Satta (1991). They present a bidirectional parser in which analyses are also discovered from the anchor of a lexicalized tree outward.

An important difference with their approach lies within the way in which auxiliary trees are handled. In our case, an auxiliary tree is traversed from the (known) foot node, rather than from the anchor of the auxiliary tree. This is reflected by our requirement that the head-corner of an auxiliary tree is its foot node. In the parser of Lavelli and Satta (1991) the parser proceeds from the lexical anchor of an auxiliary tree. This implies that there is some non-determinism in the case of adjunction: the parser does not take into account the possible analysis of a foot node and jumps over a substring possibly dominated by the foot node.

Furthermore, the bidirectional parser of Lavelli and Satta (1991) proceeds bottom-up from the anchor of an elementary tree upward. However, subtrees of an elementary tree that do not lie on this path are recognized in a top-down fashion. In the head-corner parser such subtrees are also processed bottom-up, starting from the head-corner of this subtree.

Finally note that the head-corner parser deals with substitution, whereas Lavelli and Satta (1991) do not consider the possibility of substitution.

## References


Gosse Bouma and Gertjan van Noord. Head-driven parsing for lexicalist grammars: Experimental results. In *Sixth Conference of the European Chapter of the Association for Computational Linguistics*, Utrecht, 1993.

Martin Kay. Head driven parsing. In *Proceedings of Workshop on Parsing Technologies*, Pittsburg, 1989.

Alberto Lavelli and Giorgio Satta. Bidirectional parsing of lexicalized tree adjoining grammar. In *Fifth Conference of the European Chapter of the Association for Computational Linguistics*, Berlin, 1991.

Giorgio Satta and Oliviero Stock. Head-driven bidirectional parsing. a tabular method. In *Proceedings of the Workshop on Parsing Technologies*, pages 43–51, Pittsburg, 1989.

Klaas Sikkel and Rieks op den Akker. Head-corner chart parsing. In *Proceedings Computer Science in the Netherlands (CSN '92)*, Utrecht, 1992.

Klaas Sikkel and Rieks op den Akker. Predictive head-corner chart parsing. In *IWPT 3, Third International Workshop on Parsing Technologies*, pages 267–276, Tilburg/Durbuy, 1993.

Gertjan van Noord. Head corner parsing for discontinuous constituency. In *29th Annual Meeting of the Association for Computational Linguistics*, Berkeley, 1991.

Gertjan van Noord. *Reversibility in Natural Language Processing.* PhD thesis, University of Utrecht, 1993.

## A Head-corner parser for LTAGs

I provide a Prolog implementation of the head-corner parser for Headed, Lexicalized and Feature-based TAGs. The parser returns derivation trees. This parser is the most efficient parser that I was able to implement on the basis of the specifications given above.

This version of the head-corner parser proceeds in three phases. In the first phase the candidate elementary trees are extracted from the lexicon (dictionary look-up). In the second phase the sentence is recognized and items are recorded from which derivation trees (and hence derived trees) can be recovered. In the third phase derivation trees can be recovered (it is also easy to recover derived trees — if so desired).

The second phase implements the definite clause specification given in the text. It uses memo-ization for the goal predicate only. Memo-izing other relations turns out to be much too expensive. Furthermore extra argument positions are used for the administration concerning derivation trees. An extra argument in the `hc` and `hc_na` predicates is used to indicate what words are already 'reserved' by an auxiliary tree, but not yet processed. This is necessary for termination (note that this would not be a problem in a chart-based implementation).

```
parse(TopCat,String,Deriv) :-
    first_phase(String,0,Max),                    % asserts elementary trees
    memo(goal(s(Cat,[]),0,Max,0,Max,top)),       % recognition
    rgoal(s(Cat,[]),0,Max,0,Max,top,[],[Deriv]). % recover derivation trees
```

In the first phase, the predicates `ini`, `ign_ini`, `aux` and `ign_aux` are asserted. These predicates represent appropriate elementary trees from the lexicon. `ini(Cat,Chain,Name,Q0,Q)` is true if there is an initial tree `Name` of which the anchor connects `Q0` to `Q` in the current sentence, of which the root node's top features match with `Cat`, and of which the chain from the anchor up to the root is represented as `Chain`.

Similarly, `aux(Cat,Chain,Name,Q0,Q)` is true if there is an auxiliary tree `Name` of which the anchor connects `Q0` to `Q` in the current sentence, and which is adjoined at node `Cat`, and of which `Chain` is the chain of nodes from the foot node to the root node. Note that the feature unifications of bottom and top between the target node `Cat` and the foot and root nodes of the auxiliary tree should be taken care of here. Note that we do not explicitly have to represent the foot node as we do not allow adjunctions at foot nodes.

During the recognition phase the parser uses a version of these predicates (both prefixed with `ign_` in which semantic information is suppressed. The idea is that most ambiguities also lead to different semantic constraints. Packing would hardly ever be useful if semantic information was used immediately. In this parser the semantic information is only added once derivation trees are recovered. Note that the recognizer therefore may be a bit too liberal in case such semantic constraints are used to filter out certain derivations.

Both the recognition and the recovering phase proceeds in a head-driven fashion. Whereas the recognizer asserts items for computations that are successful, the recovering phase uses these items to recover the derivation trees associated with these computations. In combination with the use of memo-ization this implies that the parser uses *packing* to reduce the search space during recognition in the case of ambiguities. The resulting set of items of the recognition phase thus provide a compact representation of all derivation trees. The third phase recovers each of the different derivation trees on the basis of this compact representation, and checks the associated semantic constraints. During recognition items are asserted of the form `d(Name,Address,SubName)` where `Name` is the name of an elementary tree where at node `Address` the elementary tree `SubName` is adjoined/substituted.

The recognizer uses the meta-logical predicate `memo(Goal)` which is equivalent to `call(Goal)` except that it uses memo-ization to compute the goal.

```
% goal(+Tree,?Begin,?End,+LeftExtreme,+RightExtreme,+ElemTreeName)
goal(s(Cat,Chain),P0,P,E0,E,Up):-
    address(Cat,Add), ign_ini(Cat,Chain0,Name,Q0,Q), between(Q0,Q,E0,E),
    hc_na(Chain0,_,Mid,Q0,Q,R0,R,E0,E,[],[],Name),
    hc_na(Chain,Mid,_,R0,R,P0,P,E0,E,[],[d(Up,Add,Name)],Up).
goal(t(Word/Q0,Chain),P0,P,E0,E,Up):-
    connects(Word,Q0,Q), between(Q0,Q,E0,E), hc_na(Chain,_,_,Q0,Q,P0,P,E0,E,[],[],Up).
goal(e(Cat,Chain),P0,P,E0,E,Up):-
    between(Q,Q,E0,E), hc(Chain,Cat,_,Q,Q,P0,P,E0,E,[],[],Up).

% hc(+Chain,?Head,?Goal,+HeadBegin,+HeadEnd,?GoalBegin,?GoalEnd,
             +LeftEx,+RightEx,+Used,+ItemsToBeAsserted,+ElemTreeName)
hc(Chain,Cat0,Cat,P0,P,Q0,Q,E0,E,U,D,Up) :-
    unify_node(Cat0), hc_na(Chain,Cat0,Cat,P0,P,Q0,Q,E0,E,U,D,Up).
hc(Chain,Cat,Goal,Q1,Q2,P0,P,E0,E,U,Ds,Up) :-
    address(Cat,Add), ign_aux(Cat,Chain0,Name,L0,L), term(L0,L,E0,QL,QR,E,U),
    hc_na(Chain0,_,Mid,Q1,Q2,Q0,Q,E0,E,[L0|U],[],Name),
    hc_na(Chain,Mid,Goal,Q0,Q,P0,P,E0,E,U,[d(Up,Add,Name)|Ds],Up).

hc_na([],Cat,Cat,P0,P,P0,P,_,_,_,Deriv,_) :- assert_deriv(Deriv).
hc_na([c(Mid,L,R)|T],_,Goal,QL,QR,P0,P,E0,E,U,D,Up) :-
    goal_l(L,Q0,QL,E0,Up), goal_r(R,QR,Q,E,Up),
    hc(T,Mid,Goal,Q0,Q,P0,P,E0,E,U,D,Up).

% goal_l(+RevListOfTrees,?Begin,+End,+LeftExtreme,+ElemTreeName)
goal_l([],Q,Q,_,_).
goal_l([H|T],Q0,Q,E0,Up):- memo(goal(H,Q1,Q,E0,Q,Up)), goal_l(T,Q0,Q1,E0,Up).

% goal_r(ListOfTrees,+Begin,?End,+RightExtreme,+ElemTreeName)
goal_r([],Q,Q,_,_).
goal_r([H|T],Q0,Q,E,Up):- memo(goal(H,Q0,Q1,Q0,E,Up)), goal_r(T,Q1,Q,E,Up).

% assert_deriv(+ListOfItemsToBeAsserted)
assert_deriv([]). assert_deriv([H|T]) :- (\+ H -> assertz(H);true), assert_deriv(T).

% term(+Begin,+End,+LeftBegin,+LeftEnd,+RightBegin,+RightEnd,+Used)
term(P0,P,L0,L,R0,R,U):- \+ member(P0,U), (between(P0,P,L0,L);between(P0,P,R0,R)).
```

The predicates of the recovery phase are almost identical to those of the recognition phase. Whereas the latter collects items to be asserted, the former uses these items to guide the search (the predicate `d/3`). Furthermore two extra argument positions are used to build the derivation tree.

A derivation tree is represented with a term `r(Name,Address,Daughters)` where `Name` is the name of an elementary tree, `Address` is the address at which the current tree is adjoined or substituted in the dominating node, and `Daughters` is a list of daughters.

```
rgoal(s(Cat,Chain),P0,P,E0,E,Up,Ds0,D):- address(Cat,Add),
    d(Up,Add,Name), ini(Cat,Chain0,Name,Q0,Q), between(Q0,Q,E0,E),
    rhc_na(Chain0,_,Mid,Q0,Q,R0,R,E0,E,[],Name,[],Dx),
    rhc_na(Chain,Mid,_,R0,R,P0,P,E0,E,[],Up,[r(Name,Add,Dx)|Ds0],D).
rgoal(t(Word/Q0,Chain),P0,P,E0,E,Up,D0,D):- connects(Word,Q0,Q),
    between(Q0,Q,E0,E), rhc_na(Chain,_,_,Q0,Q,P0,P,E0,E,[],Up,D0,D).
rgoal(e(Cat,Chain),P0,P,E0,E,Up,D0,D):-
    between(Q,Q,E0,E), rhc(Chain,Cat,_,Q,Q,P0,P,E0,E,[],Up,D0,D).

rhc(ToDo,Cat0,Cat,P0,P,Q0,Q,E0,E,U,T,D0,D) :-
    unify_node(Cat0), rhc_na(ToDo,Cat0,Cat,P0,P,Q0,Q,E0,E,U,T,D0,D).
rhc(Chain,Cat,Goal,Q1,Q2,P0,P,E0,E,U,Up,Ds,D) :- address(Cat,Add),
    d(Up,Add,Name), aux(Cat,Chain0,Name,L0,L), term(L0,L,E0,QL,QR,E,U),
    rhc_na(Chain0,_,Mid,Q1,Q2,Q0,Q,E0,E,[L0|U],Name,[],AD),
    rhc_na(Chain,Mid,Goal,Q0,Q,P0,P,E0,E,U,Up,[r(Name,Add,AD)|Ds],D).
```

```
rhc_na([],Cat,Cat,P0,P,P0,P,_,_,_,_,D,D).
rhc_na([c(Mid,L,R)|T],_,Goal,QL,QR,P0,P,E0,E,U,Up,D0,D) :-
    rgoal_l(L,Q0,QL,E0,Up,D0,D1), rgoal_r(R,QR,Q,E,Up,D1,D2),
    rhc(T,Mid,Goal,Q0,Q,P0,P,E0,E,U,Up,D2,D).

rgoal_l([],Q,Q,_,_,D,D).
rgoal_l([H|T],Q0,Q,E0,Up,D0,D):- rgoal(H,Q1,Q,E0,Q,Up,D0,D1), rgoal_l(T,Q0,Q1,E0,Up,D1,D).

rgoal_r([],Q,Q,_,_,D,D).
rgoal_r([H|T],Q0,Q,E,Up,D0,D):- rgoal(H,Q0,Q1,Q0,E,Up,D0,D1),rgoal_r(T,Q1,Q,E,Up,D1,D).
```

As an example consider again the sentence 'who did john say mary saw' for the grammar (without feature structures) given above. In that case the first phase of the parser asserts the following clauses. Here it is assumed that nodes are represented as a pair `n(Cat,Address)` where `Cat` is an atomic symbol and `Address` is used for the construction of the derivation tree. As no semantic constraints are used, assume that the predicates `ign_ini` and `ign_aux` are defined exactly the same.

```
aux( n(sbar,_),[c(n(vp,22),[t(say/3,[c(n(v,2211),[],[]),c(n(vp,221),[],[])])],[]),
                c(n(s,2),[s(n(np,21),[])],[]),
                c(n(sbar,0),[t(did/1,[c(n(aux,1),[],[])])],[])],beta_1,3,4).

ini(n(wh,_),[c(n(wh,0),[],[])],alpha_1,0,1). ini(n(np,_),[c(n(np,0),[],[])],alpha_2,2,3).
ini(n(np,_),[c(n(np,0),[],[])],alpha_3,4,5).
ini(n(s,_), [c(n(v,2221),[],[]),             c(n(vp,222),[],[]),
             c(n(s,22),[s(n(np,221),[])],[]), c(n(sbar,2),[e(n(c,21),[])],[]),
             c(n(s,0),[s(n(wh,1),[])],[])],   alpha_4,5,6).
```

In order to complete this example, assume the definitions

```
unify_node(_).   address(n(_,Add),Add).
```

For example, upon the goal:

```
?- parse(node(s,0),[who,did,john,say,mary,saw],Deriv).
```

the parser will instantiate the derivation tree `Deriv` as

```
r(alpha_4,0,[r(alpha_1,1,[]),  r(beta_1,2,[r(alpha_2,21,[])]),  r(alpha_3,221,[])]).
```

This derivation tree is recovered on the basis of the set of items:

```
d(alpha_4,221,alpha_3).  d(beta_1,21,alpha_2).  d(alpha_4,1,alpha_1).
d(alpha_4,2,beta_1).     d(top,0,alpha_4).
```