Chart Parsing in Prolog

Neil K. SIMPKINS
Department of Computer Science and Applied Mathematics,
University of Aston in Birmingham,
Birmingham, B4 7ET, United Kingdom.

Peter HANCOX*
School of Computer Science,
University of Birmingham,
P. O. Box 363, Birmingham, B15 2TT, United Kingdom.

Received 17 October 1987
Revised manuscript received 19 March 1990

Abstract Several differing approaches to parsing using Prolog are discussed and their characteristics outlined, in particular Definite Clause Grammar (DCG), the Bottom-Up Parser (BUP) and the Active Chart Parser. Attention is paid to the conflict that arises between the simplicity and efficiency of the parsing algorithm when using a grammar specified as a linguistic, rather than computationally efficient, description of a sublanguage. A simple and efficient parsing algorithm called 'Word Incorporation' is described. Its efficient implementation in Prolog and extensions for handing literals, the Kleene star operator and gaps in grammar rules are described using experience gained with the unification-based formalism, Lexical Functional Grammar (LFG).

Keywords: Chart Parsing, Literals, Kleene Star, Gaps, Lexical Functional Grammar.

§1 Introduction

Prolog is a declarative programming language which has an established popularity with the Natural Language Processing (NLP) community. Many previous NLP systems have simply used the Top-down Depth-first Backtracking (TDB) search strategy of Prolog directly.\textsuperscript{1,2} This idea has also been applied to more recent grammar formalisms such as Lexical Functional Grammar.
These grammars have been designed by linguists to be formalisms suitable for describing language concepts at a high level. In contrast to this some previous grammars were designed with an accent on implementation and computational efficiency.

A linguistic grammar however may prove more efficient in practice. The formalism itself may be more difficult to implement and less computationally efficient but if it has a greater power of linguistic description then the final system should be more efficient. Linguistic grammars draw on a linguistic vocabulary and are thus much clearer and at a higher level than computational grammars. Linguistic grammars, such as LFG, have a notation formal enough to be implemented but say nothing about parsing, nor in general do they directly impose any constraints on the parsing strategy used.

Prolog is a powerful, concise and efficient programming language but we do not see Prolog's search strategy as directly applicable to the problem of parsing natural language. We have found simplistic parsing strategies such as TDB to be inefficient and too restrictive when applied to the linguistically designed grammars such as LFG. We require a parser that can support the notational mechanisms of our formalism (non-deterministic rewrite context-free grammar rules, literals in rules, the Kleene star operator) and also other general linguistic mechanisms (lexical ambiguity, gaps) in an efficient and perspicuous manner.

§2 Prolog's TDB Search Strategy

Prolog (TDB) is the most simple parsing strategy to implement as the grammar rules themselves serve as the Prolog program, as is the case for example with Definite Clause Grammars (DCG). Extra arguments are added to the grammar rules to carry structure building and contextual information. When writing more complex grammars of a non-deterministic nature or that allow lexical ambiguity, the parser may frequently backtrack over previous work. It is possible to restructure a DCG so that backtracking is eliminated but this restructuring requires an understanding of TDB, (perhaps inventing unnatural grammatical categories) and often produces unclear grammatical descriptions. Non-determinism in the lexicon cannot be eliminated. A top-down execution of rules also does not allow left-recursive rules (np → np, mod) to be used.

The inefficiencies inherent in TDB led to the development of a Bottom-Up Parsing approach (BUP). This still uses Prolog's TDB to execute grammar rules directly but alters the order of goals in the (Prolog) grammar rules to achieve bottom-up parsing. The work done during parsing is saved by recording successful and unsuccessful goals in the Prolog database using the Prolog assert predicate. This information can then be used, after backtracking has occurred, to avoid redoing goals which previously completed or that failed. A BUP strategy also allows a greater freedom in writing grammar rules. Left-recursive grammar rules can be used which are a natural description of several common
linguistic constructs. Fully recursive rules \((N \rightarrow N)\) cannot be used, but such rules do not seem to be a desirable linguistic facility. BUP produces the various different parses of a phrase in series. That is to say that the goals pursued during parsing are taken as being a part of a single target parse.

The BUP strategy has two main weaknesses. The most important of these is the loss of clarity and great expense incurred by recording information, during parsing, in the Prolog database. Also BUP cannot easily accommodate 'e' productions where some non-terminal expands to a gap (or trace) in the constituent structure \((np \rightarrow e)\). This type of rule is used by many formalisms as the basic mechanism for uncovering underlying phrase structure in constructs involving movement (interrogatives, relatives etc.). Outside the Prolog user community other, more efficient, parsing algorithms have been developed. These have mostly been written in Lisp but in some cases may be efficiently and naturally adapted to Prolog. Of these algorithms, the Active Chart Parser has attracted the most attention recently.

§3 The Chart

The Chart is a graph data structure which can support a range of parsing algorithms (top-down or bottom-up and breadth or depth first). The Chart acts as book-keeping storage for all the information produced during parsing. Graph nodes are called vertices which can be linked by arcs called 'edges'. Each edge represents a grammar rule (or dictionary entry) and contains a minimum of:

- starting and ending vertices numbers;
- a category (rule Left-Hand Side (LHS) or word category);
- a remainder (initially the rule Right-Hand Side (RHS) or nil for a dictionary item).

Dictionary items (grammar terminals) may be viewed as rules with an empty RHS, and a LHS equal to the item's dictionary category. Edges may carry any required additional information to produce the required output. In our

Distinguished grammar category: s
Grammar rule: \(s \rightarrow np, vp\).
Pending 'active edge' \([1, 1, s, [np, vp]]\)
Complete ‘base’ edges: \([1, 2, noun, []], [2, 3, verb, []], \ldots\]

![Chart Diagram](image)

Fig. 1 Initial Chart State.
implementation edges also carry a functional description (f-structure) of the phrase parsed.

An initialisation step (Fig. 1) creates the initial edges and vertices. Each input word is assigned a starting and ending vertex number and forms a complete edge (having no remainder) in the ‘base’ of the Chart. An active edge representing each grammar rule with LHS which is the distinguished grammar category is proposed at (that is having starting and ending vertices equal to) the first vertex. These active (incomplete) edges are placed on the list of ‘pending’ edges to be entered later onto the Chart; that is, the edges are ‘proposed’ (Fig. 1).

The Chart algorithm selects a pending edge and enters this onto the Chart. This process has two distinct phases, combining the edge with other edges on the Chart and invoking more grammar rules. Each of these activities may create new active or complete edges which are added to the pending edges.

A complete edge which is entered onto the Chart may extend (match) an active edge on the Chart (bottom-up following successful top-down expansion), if the active edge’s end is equal to the complete edge’s start and the complete edge’s category is the first part (or all of) the active edge’s remainder. An active edge which is entered may be extended (bottom-up) by a complete edge on the Chart if these match in the same way. If the complete edge matches the last part of the active edge’s remainder then a new complete edge is created; if some remainder is left then a new active edge is created. The new edge created (active or complete) spans the vertices of both its component edges and is put on the pending list of edges.

An edge, with category ‘C’, invokes (top-down) new rules (edges) at its starting vertex if there is a grammar rule that has the same category ‘C’ as its LHS. Before any edge is added to the pending list the Chart must be searched to ensure that the edge has not been proposed before (this prevents infinite recursion). This requires searching all edges on the Chart (active and complete) and all of the pending edges, for an edge with the same category and starting vertex. The complete algorithm cycles until some complete edge with the required category and/or starting and ending vertices is produced.

The Chart contains common goals (substrings) of different parses so that these need only be produced once. These need not be asserted into the Prolog database but can be carried through the algorithm’s cycles as Prolog arguments in the algorithm’s Prolog rules. The algorithm is obviously cyclic, not a Prolog-like (top-down) search; no backtracking occurs at all. The algorithm can be varied by changing the selection of an edge from the pending list to be entered onto the Chart. For example if edges with highest vertices (ending) are selected, the different parses will tend to be developed in series. By selecting those with lowest ending vertices, the different parses can be developed in parallel. Selecting completed edges promotes bottom-up combination whilst selecting those only with a remainder tends to result in top-down expansion.
Whatever the selection criteria an attempt is made to use each edge both for bottom-up combination and top-down expansion.

The main weakness of the Chart is its low efficiency, mainly caused by a high overhead when proposing new edges (checking an edge has not previously been proposed). It is a simple matter to index edges by their starting or ending vertex so that the outgoing or incoming set of edges to a vertex (edgeset) can be quickly accessed, but the Chart algorithm requires access to Chart edges by start and end and pending edges by start. Thus edge check cannot be efficiently supported in Prolog. The check must be performed on all the edges produced (top-down or bottom-up). As the Chart carries a complete parsing history (Fig. 2) the number of edges to be checked may be very large. The Chart is thus also very expensive in space. This is really the cost of the flexibility (non-determinism) in selecting the next pending edge to combine with the Chart. Pending edges, which have yet to be entered, may span any vertices, including those behind the current parsing front (Fig. 2), thus creating gaps in the parsing history stored on the Chart. When an edge from behind the parsing front is entered onto the Chart the new edges created may have been created before. To avoid the possibility of looping, the entire parsing history must be checked to prevent re-proposing an edge.

![Diagram of Chart Parsing Tree](chart-parsing-tree.png)

**Fig. 2** Outline of Chart Parsing Tree.

Another weakness of the Chart algorithm is the use of every edge to both combine bottom-up and expand top-down. This generates a large number of edges many of which (we have observed especially those generated top-down) may never be used and increases the edge checking problem when proposing new edges.
Chart parsing would seem an obvious candidate for parallel processing and there have been several studies. Following on the work with BUP, Matsumoto\textsuperscript{10} extended the work using Parlog. As with BUP, the parsing strategy remained bottom-up initially without the top-down prediction typical of the Chart algorithm. There was a close correspondence between purely bottom-up chart parsing and Matsumoto's algorithm, and it proved possible to add top-down prediction, giving the similar improvement gained by the usual Chart algorithm. There are associated costs, with the grammar having to be analysed at compile-time to provide information for top-down prediction and (more importantly) the input string has to be complete at parsing, rather than allowing for parsing-on-the-fly. Other work has also used Parlog with a port being undertaken to a transputer-based system\textsuperscript{11}.

§4 Word Incorporation

The Word Incorporation (WI) algorithm\textsuperscript{8} can be thought of as a specialization of the Chart. At a cost of some flexibility the Chart data structure itself and the processing algorithm can be greatly simplified.

The Chart algorithm is restricted to be solely bottom-up depth-first (left-to-right or right-to-left). This means that parsing proceeds along the input (base of completed edges) in an orderly fashion and there are no pending edges entered behind the parsing front. The Chart holds only active edges and the 'pending' list solely consists of completed edges. To achieve depth-first parsing, the completed (pending incorporation) edge list is handled on a First-In-First-Out (FIFO) basis. Each complete (base) edge is combined (bottom-up) with all matching active edges: these being the active edges with an end vertex equal to the complete edge's start and a first remainder category which is that of the complete edge. A complete edge is also used to invoke new rules which have a matching RHS: these being all rules with a first RHS category equal to that of

![Fig. 3 Outline of Word Incorporation Parsing Tree.](image-url)
the completed edge.

New active edges are placed on the Chart and new complete edges on the pending list. Newly completed edges are also tested to see if the final target edge has been reached. Completed edges once entered onto the Chart can be dispensed with. Thus only information on the current active ‘parsing front’ is maintained (Fig. 3). This does not mean that parsing goals need be pursued more than once, as each completed goal is used (bottom-up) to its full potential when added to the Chart and is then expended. As the algorithm proceeds along the base in an orderly fashion, processing never takes place at any vertex more than once so that clearly no check to see if an edge has been proposed before is required. The resultant decrease in the number of edges held in the Chart and the reduction in processing produce a highly efficient parsing algorithm.

It is characteristic of the WI algorithm that the development of different parses is done in parallel. The invocation speed of new rules can be improved by indexing rules by their first RHS category.

Phillips’ implementation of the Word Incorporation parser was undertaken in Lisp.\textsuperscript{8} It has been subsequently used in a larger-scale project (again implemented in Lisp).\textsuperscript{9} The account below represents (to the extent of the authors’ knowledge) the only practical implementation and extension of the algorithm in Prolog.

§5 Implementation

The implementation is described in four parts. First the efficient implementation of the basic Word Incorporation parser is described. Its extension to cover the parsing of literals, the Kleene star operator and gaps is described.

To allow fast access to an edgeseat (those with the same ending or starting vertex) the Chart edges can be held in a one dimensional array. An understanding of the intricacies of the array is not necessary for a general understanding of the work, but necessary for a thorough presentation. Therefore the details are relegated to the Appendix.

The parser we describe makes certain assumptions about dictionary lookup and the grammar. The dictionary is accessed by a procedure \textit{lookup}, which given a word returns a list of dictionary for the word:

\begin{verbatim}
?- lookup(orange, Entries).
Entries = [noun(orange_fruit), adj(orange_colour)].
\end{verbatim}

The grammar is reformed before use so that rules can be accessed by the first category of their RHS. We divide rules into those whose RHS consists of a single category (these rules are completed by single category) and those with some remainder of their RHS after the category (these rules will produce active edges after the category is found). The rules:

\[
\text{np} \longrightarrow \text{n}.
\]
might take the form: 'first RHS category’ rules (‘list of rules with their remain-
ders’, ‘list of rules completed’):

\[
\begin{align*}
n \quad & \text{rules} & \{ [[np, (n, v, np)]]], [[np, (n)]] \}. \\
det \quad & \text{rules} & \{ [[np, (det, n)]]], [np, (det, adj, n)]]], [[]] \}. \\
\end{align*}
\]

The original rules are retracted from the Prolog database and the new ‘rules’
predicates (‘rules’ is defined as an infix operator) asserted. This allows for
instance, all the ‘n’ rules to be accessed:

\[
\begin{align*}
n \quad & \text{rules} & \text{(Started, Completed)}. \\
\end{align*}
\]

As an example of WI we will consider parsing ‘John saw the tall man’ and
building a familiar derivation tree or constituent structure (c-structure in LFG).
The simple context-free grammar we will use is:

\[
\begin{align*}
:- \text{op}(1000, xfy, [‘\=>'\]). \\
:- \text{dynamic ‘\=>'\}/2. \% a dynamic declaration is required in} \\
\quad & \text{\% Quintus for any predicate which is changed.} \\
\end{align*}
\]

\[
\begin{align*}
s(s(np(NP), vp(VP))) & \Rightarrow np(NP), vp(VP). \\
v(v(V), np(NP))) & \Rightarrow v(V), np(NP). \\
v(v(V)) & \Rightarrow v(V). \\
np(np(n(N))) & \Rightarrow n(N). \\
np(np(det(D), n(N))) & \Rightarrow det(D), n(N). \\
np(np(det(D), adj(A), n(N))) & \Rightarrow det(D), adj(A), n(N). \\
\end{align*}
\]

This grammar is converted so as to be indexed by first RHS category:

\[
\begin{align*}
:- \text{use}\_\text{module(library(findall))}. \% \text{findall is a Quintus library} \\
\quad & \text{\% module which must be loaded.} \\
:- \text{op}(100, xfx, [\text{rules}]). \\
index\_by\_rhs\_cat :- \\
\quad & \text{more\_rules(Cat),} \\
\quad & \text{findall([Lhs, (Cat, Remainder)],} \\
\quad & \text{retract([Lhs =\Rightarrow Cat, Remainder])}, \\
\quad & \text{\text{Started),}} \\
\quad & \text{findall([Lhs, Cat],} \\
\quad & \text{retract([Lhs =\Rightarrow Cat]),} \\
\quad & \text{\text{Completed),} \\
\quad & \text{asserta\(\text{Cat rules(Started, Completed))},} \\
\quad & \text{index\_by\_rhs\_cat} \\
\text{index\_by\_rhs\_cat} & \% \text{finished, no more rules.} \\
\end{align*}
\]
more_rules(Cat) :-
    (\_ \implies Cat, \_).
more_rules(Cat) :-
    (\_ \implies Cat).

This will produce a new grammar:

\[
\begin{align*}
n(A) & \quad \text{rules}([], [[np(np(n(B))), n(B)]]). \\
det(A) & \quad \text{rules}([[np(np(det(B), n(C))), (det(B) ,; n(C))], \\
& \quad \quad [np(np(det(D), adj(E), n(F))], \\
& \quad \quad \quad (det(D) ,; adj(E) ,; n(F))]], \\
& \quad \quad []). \\
v(A) & \quad \text{rules}([[vp(vp(v(B), np(C))), (v(B) ,; np(C))]], \\
& \quad \quad [[vp(vp(v(D))), v(D)]]. \\
np(A) & \quad \text{rules}([[s(s(np(B), vp(C))), (np(B) ,; vp(C))]], \\
& \quad \quad []). 
\end{align*}
\]

The various dictionary entries for each word can be similarly indexed by the word they define. For simplicity we will define 'lookup' as the dictionary predicate itself and only hold the grammatical category of each entry for structure building. In LFG we hold feature and function specifications (f-structures):

\[
\begin{align*}
\text{lookup(john, [n(john)]).} \\
\text{lookup(saw, [v(see_past), adj(saw_action)]).} \\
\text{lookup(the, [det(the)]).} \\
\text{lookup(tall, [adj(tall)]).} \\
\text{lookup(man, [n(man)]).} 
\end{align*}
\]

The BUP system also has a single dictionary predicate which performs morphological analysis and asserts all the different definitions of a word into the Prolog database so that these can be used after backtracking without accessing the dictionary again. With WI we can simply create a separate base edge (complete) for each word definition and develop these in a parallel fashion (FIFO).

The top level predicate of the parser first creates the initial base of (pending) completed edges, constructs a target edge and then 'elevates' the base (bottom-up) toward the target edge. We have defined the target as an edge with category 's' (the grammar's distinguished symbol) which spans the entire input(vertices):

\[
\begin{align*}
\text{constant(distinguished(s(_))).} & \quad \% \text{declare 's' as distinguished.} \\
\text{parse(Words, Cat):-} \\
& \quad \text{initialise(Words, Base, 1, Lv),} \\
& \quad \text{new_array Active size Lv elements [],} \\
& \quad \text{constant(distinguished(Cat)),}
\end{align*}
\]
elevate(Base, Active, [I, Lv, Cat]).
initialise([], [], Lv, Lv).
initialise([Word|Rest], Base, Sv, Lv):-
lookup(Word, Entries),
Ev is Sv + I,
make(Base, Entries, Sv, Ev, Rest_base),
initialise(Rest, Rest_base, Ev, Lv).
make(Base, [], _, _, Base).
make([Sv, Ev, Cat]|Rest_base], [Cat|Rest], Sv, Ev, Rest_basedl):-
make(Rest_base, Rest, Sv, Ev, Rest_basedl).

Elevation of the base to produce the target edge is done by entering each base edge in turn onto the Chart of active edges. We require all those active edges incident to the complete edge's starting vertex. Active edges (edgesets) are thus held in array elements according to their ending vertex. The elevation cycle never backtracks: global information (base and active edges) is carried throughout the cyclic Prolog rules as Prolog arguments.

elevate([Sv|Rest_edge]|Base], Active, Target):-
Sv elem_of Active has_value Edge_set,
enter([Sv|Rest_edge], Edge_set, Target, Active, Base).

Entering an edge onto the Chart requires deciding if it matches any incident active edge (we try all of these) and then use the edge to invoke new grammar rules. Also we check to see if any new completed edge is the target edge. The five cases (Prolog rules) for dealing with the edge being entered are thus:

1. having tried to match the complete edge with all incident active edges, now use the complete edge to invoke grammar rules:

enter(Edge, [], Target, Active, Base):-
invo kes(Edge, Active, Activel, Base, Base1),
elevate(Base1, Activel, Target).
invo kes([Sv, Ev, Cat], Active, Activel, Base, Base1):-
Cat rules(Started, Completed), % get the indexed rules.
Ev elem_of Active has_value Old_active,
now_active([Sv, Ev, Cat], Started, Old_active, New_active),
Ev of Active := New_active in Activel, % new may be []
now_base([Sv, Ev, Cat], Completed, Base, Base1).
invo kes(_, Active, Active, Base, Base). % no rules invoked.

% add new active edges to their active edgeset in the array
now_active(_, [], Active, Active).
now_active([Sv, Ev, Info], [[Head, (Info, Remainder)]|Rest],
Old_active, [[Sv, Ev, Head, Remainder]|Rest_new]):-
now_active([Sv, Ev, Info], Rest, Old_active, Rest_new).
% add new complete edges to the front of the base (LIFO) list.
now_base(_, [], Base, Base).
now_base([Sv, Ev, Info], [[Head, Info]|Rest], Base, Base1):-
    now_base([Sv, Ev Info], Rest, [[Sv, Ev, Head]|Base], Base1).

(2) The complete edge matches the next active edge from the edgeset, producing a new active edge which is then added to its edgeset.

enter([Sv, Ev, Cat], [[Sv1, Sv, Head, (Cat, Rem)]|Rest],
      Target, Active, Base):-
    Ev elem_of Active has_value Old_active,
    Ev of Active := [[Sv1, Sv, Head, Rem]|Old_active] in Active1,
    enter([Sv, Ev, Cat], Rest, Target, Active1, Base).

(3) The complete edge matches the next active edge producing a complete edge which is the target edge.

enter([Sv, Ev, Cat], [[Sv1, Sv, Head, Cat]|__], [Sv1, Ev, Head], __).

(4) The complete edge completes an active edge which is then added to the base list (LIFO).

enter([Sv, Ev, Cat], [[Sv1, Sv, Head, Cat]|Rest], Target, Active, Base):-
    enter([Sv, Ev, Cat], Rest, Target, Active, [[Sv1, Ev, Head]|Base]).

(5) The complete edge does not match the first active edge so try the next active edge.

enter(Edge, [__|Rest], Target, Active, Base):-
    enter(Edge, Rest, Target, Active, Base).

The top level Prolog rule to load all files, pre-process the grammar and parse 'John saw the tall man' can be declared:

start :-
    consult([arrays, grammar, lookup, pre_process_rules, parser]),
    index_by_cat,
    parse([john, saw, the, tall, man], Cat),
    Cat =.. [s|Struct],% separate the structural argument
    write(Struct), nl.

The structure produced is:
\[ s( np(np(n(john))), \]
\[ \text{vp(vp(v(see\_past)), np(det(the), adj(tall), n(man)))))}. \]

§6 WI Parsing of Literals, the Kleene Star Operator and Gaps

The previous section described the implementation of the basic Word Incorporation parser. This section describes extensions necessary to make the algorithm readily usable for parsing with linguistic grammars, drawing on our experience with LFG.

6.1 Literals

Many grammar formalisms with a context free grammar basis allow words (literals) to appear in rules as terminals (e.g. \( s \rightarrow np \text{ 'to' vpl} \)). It is a simple matter to extend the WI algorithm to allow this. The grammar must first also be indexed by literals in just the same way as categories.

We only allow 'single item' literals so that a literal will always be a single item (edge) in the base of edges. Of course several contiguous literals may be used in a grammar rule. Essentially edges are matched, as we have described, by vertex and grammatical category. We now also want edges to match by vertex and the literal represented (by a base edge).

We could match literals in rules to any complete (base) edge including those representing more than one literal but this would require that every complete and active edge be maintained with information about the string it spans. Instead we confine literals in rules to be one base edge and introduce a new type of base edge for matching with these (Fig. 4). This reduces the overhead of dealing with literals to entering an extra (literal) complete edge at each successive vertex, and of course invoking grammar rules which have a RHS starting with the literal. Given the previous example there is no difference

![Fig. 4 Initial Complete Edges (category and word types).](image-url)
between a matching a base word type edge with a rule (RHS) word and matching a completed edge structure with a rule (RHS) structure. No changes are thus needed to the code.

Not all words in the base need generate word type edges. We can constrain the generation of word type edges to just those words which are mentioned in the grammar. During grammar pre-processing the literals in rules can be noted and an extra dictionary entry for the word generated (with a special category 'lit'). These dictionary entries can then be used when constructing the base to generate word type complete edges.

6.2 The Kleene Star

The Kleene star operator is another linguistic notational device common to a number of grammar formalisms. The Kleene star operator (*) is used to denote an optional repetition of an item in the context-free grammar:

\[ \text{vp} \rightarrow \text{v, np, pp}^* \]

The ‘pp*’ thus represents zero or more repetitions of ‘pp’. This notational device is particularly convenient for describing free adjunct and adjective repetitions:

- the ball
- the {\textit{big}} ball
- the {\textit{big blue}} ball
- the house
- the house by the river
- the house by the river in the field

We will restrict this notion so that the star represents one or more repetitions of some category. This means that the rule above must be represented:

\[ \text{vp} \rightarrow \text{v, np} \]
\[ \text{vp} \rightarrow \text{v, np, pp}^* \]

This greatly simplifies parsing (and avoids the problem of dealing with rules such as pp \rightarrow pp*) but if the single rule definition were required we could pre-process the grammar to produce multiple rules.

Now an active edge: [vp, (pp*, np)] matches a complete edge: [pp] to produce two new edges: [vp, (pp*, np)] and [vp, (np)]. Also an edge such as: [vp, (pp)] matches with: [pp] to produce: [vp, (pp*)] and [vp].

For any category marked with a Kleene star we must perhaps produce multiple structures: pp*(pp) may lead to a structure:

\[ \text{pp}([\text{pp}(\text{PP1}), \text{pp}(\text{PP2}) \ldots \text{pp}(\text{PPn})]) \]

In Prolog we cannot refer to a structure such as pp(Struct) by Cat(Struct) as this is not legal syntax. If we suitably define an infix operator 'struct' then we can use
this to separate a category from its structure for more convenient matching:

\[
\begin{align*}
\text{:- op( 1000, xfy, ['==>']).} \\
\text{:- op( 100, xf, ['*']).} \\
\text{:- op( 120, xfx, ['*']).} \\
\text{:- op( 110, xfx, [struct]).} \\
\text{:- op( 110, fx, [struct]).}
\end{align*}
\]

Now a category with structure argument such as: pp(PP) is written: pp struct PP. This will then match with Cat struct S. We also will want to produce edges representing the alternative expansions of pp*:

\[
\begin{align*}
\text{pp*} & \rightarrow \text{pp, pp*} \\
\text{pp*} & \rightarrow [] \quad \text{(i.e. nil)}
\end{align*}
\]

That is, we want to take a single structure and produce two new structures using the two alternatives above. One which may accept another (more) pp structures and the other representing a completed pp* structure. These will be generated from a single pp* structure which is also present in the rule's LHS structure. As a Prolog variable can have at most one value we must produce a copy of the LHS structure and pp* structure produced so far, with different variables in corresponding positions. This can be done in most Prolog dialects and is achieved in Quintus using the primitive \textit{copy_term} predicate illustrated below.

\[
\begin{align*}
\text{?- copy_term(name(A, of(A, B), C, B), New_copy).} \\
\text{New_copy = name(D, of(D, E), F, D).}
\end{align*}
\]

If such a primitive were not available, the array of active edges could be kept in the database (using \textit{assert}). Matching against variables stored in the Prolog database provides a set of new variables automatically. If variables are asserted into the database:

\[
\text{?- assert(variables(A, B, C)).}
\]

Then this clause may be matched and the new variables instantiated without instantiating the variables in the database:

\[
\begin{align*}
\text{?- variables(D, E, F), D = a, E = b, F = c, listing(variables/3).} \\
\text{variables(A, B, C).}
\end{align*}
\]

We will illustrate use of the Kleene star in parsing ‘Peter walked by the store into the church’. The grammar to be used is:

\[
\begin{align*}
\text{:- dynamic '==>'/2.} \\
\text{s struct (s(np(NP), vp(VP))) = ==>} \quad \text{np struct NP,} \\
\text{vp struct VP.} \\
\text{vp struct (vp(v(V), pp*(PP))) = ==>} \quad \text{v struct V,} \\
\text{pp* struct PP.}
\end{align*}
\]
np struct (np(pn(PN)))) =⇒ pn struct PN.
np struct (np(det(D), n(N))) =⇒ det struct D,
n struct N.
pp struct (pp(p(P), np(NP))) =⇒ p struct P,
np struct NP.

These rules are pre-processed in the same manner as before. The initialisation stage is very much the same as before, alterations are required only to cope with the new method of structure building:

constant (distinguished(s)). % declare 's' as distinguished symbol.
parse(Words, s) :-
  initialise(Words, Base, I, Lv),
  new_array Active size Lv elements [],
  constant(distinguished(Cat)),
  elevate(Base, Active, [I, Lv, Cat struct S]).

initialise([], [], Lv, Lv).
initialise([Word|Rest], Base, Sv, Lv) :-
  lookup(Word, Entries),
  Ev is Sv + 1,
  make(Base, Entries, Sv, Ev, Rest_base),
  initialise(Rest, Rest_base, Ev, Lv).

make(Base, [], _, _, Base).
make([[Sv, Ev, Cat struct S]Rest_base], [Cat struct S|Rest],
      Sv, Ev, Rest_base1) :-
  make(Rest_base, Rest, Sv, Ev, Rest_base1).

elevate([[Sv|Rest_edge]Base], Active, Target) :-
  Sv elem_of Active has_value Edge_set,
  enter([[Sv|Rest_edge], Edge_set, Target, Active, Base).

Alterations to the parser are required to account for repetitions, signified by Kleene stars positioned after grammatical categories in the grammar rules. A completed base edge may match an active edge to produce a new complete or active edge as before. In addition to this a complete edge may match a rule with a remainder marked with a Kleene star. In these cases two new edges are produced. One where the Kleene repetition is assumed to be complete (this is the same as matching an unmarked edge) and one which contains the structure of the base edge but is the same as the original active edge (allowing more repetitions of the marked category). Complete base edges match active edges in three different ways: extension to produce a new active edge; completion to produce a new complete edge; and completion to produce a target edge. There are thus three new rules required for enter:
% as before: no more active edges to match, try invoking new rules.
enter(Edge, [], Target, Active, Base):-
  invokes(Edge, Active, Active1, Base, Base1),
  elevate(Base1, Active1, Target).

% 1) edge matches next active edge, which has Kleene star, producing two active edges
% one without the Kleene star's marked category one with the category still in place
% for further repetitions.
enter([Sv, Ev, Cat struct S],
      [[Sv1, Sv, Head, (Cat* struct ([S | S1]), Rem)]|Rest],
      Target, Active, Base):-
  copy_term((Head, S1), (Head1, [])),
  Ev elem_of Active has_value Old_active,
  Ev of Active := [[Sv1, Sv, Head, (Cat* struct S1 Rem)],
                   [Sv1, Ev, Head1, Rem]]Old_active in Active1,
  enter([Sv, Ev, Cat struct S], Rest, Target, Active1, Base).

% as before: edge matches next active edge to form new active edge.
enter([Sv, Ev, Cat struct S], [[Sv1, Sv, Head, (Cat struct S, Rem)]|Rest],
      Target, Active, Base):-
  Ev elem_of Active has_value Old_active,
  Ev of Active := [[Sv1, Sv, Head, Rem]]Old_active in Active1,
  enter([Sv, Ev, Cat struct S], Rest, Target, Active1, Base).

% 2) edge matches rules with last remainder marked with a Kleene star.
% if this is the last Kleene repetition the rule will be complete and the edge produced
% may be the target edge.
enter([Sv, Ev, Cat struct S],
      [[Sv1, Sv, Head, Cat struct S]|_], [Sv1, Ev, Head], _,_).

% 3) edge matches the last part of a rule which is marked with a Kleene star producing
% one complete edge (no more repetitions), but which is added to the base, and one
% rule the same as the original but incorporating the base edge's structure allowing
% for more repetitions.
enter([Sv, Ev, Cat struct S],
      [[Sv1, Sv, Head, Cat* struct ([S | S1])]|Rest], Target, Active, Base):-
  copy_term((Head, S1), (Head1, [])),
  Ev elem_of Active has_value Old_active,
  Ev of Active :=
      [[Sv1, Ev, Head, Cat* struct S1]|Old_active] in Active1,
  enter([Sv1, Ev, Cat struct S], Rest, Target, Active1,
        [[Sv1, Ev, Head1]|Base]).

% as before: edge completes an active edge to produce the target edge.
enter([Sv, Ev, Cat struct S],
      [[Sv1, Sv, Head, Cat* struct ([S])]|_], [Sv, Ev, Head], _,_).

% as before: edge completes an active edge, add new complete edge to base (LIFO).
Chart Parsing in Prolog

\[\text{enter}([\text{Sv}, \text{Ev}, \text{Cat struct S}], [\text{[Sv1, Sv, Head, Cat struct S]} | \text{Rest}]),\]

\[\text{Target, Active, Base}) : -\]

\[\text{enter}([\text{Sv}, \text{Ev}, \text{Cat struct S}], \text{Rest},\]

\[\text{Target, Active, [Sv1, Ev, Head]} | \text{Base}]).\]

\% as before: edge does not match first active edge, try the next.
\text{enter(Edge, [\_ | \text{Rest}], Target, Active, Base)} : -
\text{enter(Edge, Rest, Target, Active, Base)}.

For the sake of completeness we include the remainder of the parser which invokes new grammar rules, as before, after active edges have been extended:

\[\text{invokes}([\text{Sv}, \text{Ev}, \text{Cat struct S}], \text{Active, Active1, Base, Base1}) : -\]

\[(\text{Cat struct _}) \text{rules(Started, Completed),}\]
\[\text{Ev elem_of Active has_value Old_active,}\]
\[\text{now_active([Sv, Ev, Cat struct S], Started, Old_active, New_active),}\]
\[\text{Ev of Active := New_active in Active1, \% new may be [\]}\]
\[\text{now_base([Sv, Ev, Cat struct S], Completed, Base, Base1}).\]

\text{invokes(_, Active, Active, Base, Base). \% no rules invoked by edge.}\]
\text{now_active(_, [\_], Active, Active).}\]

\[\text{now_active([Sv, Ev, Cat struct S],}\]
\[\text{[[Head, (Cat* struct[S|S1], Rem)] | Rest],}\]
\[\text{Old_active, [[Sv, Ev, Head, (Cat* struct S1, Rem)],}\]
\[\text{[Sv, Ev, Head1, Rem]} | \text{Rest_new)]) : -}\]
\[\text{copy_term((Head, S1), (Head1, []))},\]
\[\text{now_active([Sv, Ev, Cat struct S], Rest, Old_active, Rest_new}).\]

\[\text{now_active([Sv, Ev, Cat struct S],}\]
\[\text{[[Head, (Cat struct S, Rem)] | Rest],}\]
\[\text{Old_active, [[Sv, Ev, Head, Rem]} | \text{Rest_new)]) : -}\]
\[\text{now_base([Sv, Ev, Cat struct S], Rest,}\]
\[\text{[[Sv, Ev, Head]} | \text{Base}], Base1}).\]

\[\text{now_base(_, [\_], Base, Base).}\]
\[\text{now_base([Sv, Ev, Cat struct S],}\]
\[\text{[[Head, Cat struct S]} | \text{Rest], Base, Base1}) : -}\]
\[\text{now_base([Sv, Ev, Cat struct S], Rest,}\]
\[\text{[[Sv, Ev, Head]} | \text{Base}], Base1}).\]

After pre-processing the grammar the parse can be made:

\[\text{parse([peter, walked, by, the, store, into, the, church], Structure),}\]
\[\text{pretty_print(Structure), nl.}\]

The output structure (pretty-printed here) clearly shows how \text{pp^*} was expanded into a list of \text{pps}:

\[\text{Structure =}\]
\[\text{s( np(np(pn(man_peter))),}\]
\[\text{vp( vp(v(walk_action),}}\]
pp[
  pp( p(past_action),
       np(np(det(the), n(large_shop)))
  )
  pp( p(enter_into),
       np(np(det(the), n(church)))
  )
]

The structure produced by the prepositional phrase (pp) with Kleene star is italicized in the complete structure.

6.3 Gaps

Many context-free grammar formalisms allow the use of gaps or 'e' productions (often called 'traces'). These are usually associated with some movement in the surface structure of a phrase to uncover the underlying phrase structure (or vv). Here the concern is with extraposition, although it is recognized that the use of Dahl's Gapping Grammar\textsuperscript{12,13} (for instance) can be extended to create simpler, higher-level grammar formalisms or to cover free word order. The method we have adopted for dealing with gaps requires that a gap ('e') be viewed as a dictionary or lexical item rather than a terminal in the grammar. A single entry for gaps can be imagined which has category 'e' and a
nil lexical realization. In most formalisms a gap corresponds to some other constituent to which the gap is related by a formalism dependent mechanism.

In LFG controller/controllee correspondences are used as a mechanism by which the f-structure of some constituent is passed to a gap hypothesized by the context-free grammar. Other constraints are also imposed by the formalism in order to select the correct positioning of gaps. We shall use a mechanism similar to that of LFG but simplified to some extent for ease of understanding. This will not detract from our aim of illustrating how to parse gaps bottom-up in other formalisms.

We will deal with gaps simply by hypothesizing an initial gap at each vertex. Gaps are generated during the initialisation phase where each gap edge has category 'e', no remainder and the same starting and ending vertex (Fig. 5). The category 'e' is used in the context-free grammar to denote a gap which can match an 'e' edge in the base. Firstly we need to introduce some notion of the well-formedness of an analysis so that the correct hypothesis of gaps can be identified. For this we will rely on the prediction of a 'correct' set of surface functions, in the correct order to satisfy the subcategorization or government requirements of a verb 'template'. The sentence to be parsed is 'What will Peter bring to the party?'; a WH-Fronted interrogative (question). The underlying structure of this is closely related to that of the corresponding declarative 'Peter will bring <X> to the party.' where '<X>' is the element under question (whatever Peter will bring to the party). The functional decomposition of this phrase is illustrated in Fig. 6.

![Fig. 6 Functional Structure of Declarative Form.](image)

In order to uncover this underlying structure movement of the WH-front is required as shown in Fig. 7. The components undergoing movement may be indexed to signify that they have undergone movement and co-index source and gap (trace) if required.

![Fig. 7 Movement of WH-Front.](image)
To support movement we obviously need some method of passing the information from a source category through the grammar rules to the destination gap. This we will simply do using a list to hold moved elements. The WH-front word ‘what’ will thus deposit its features or representation onto this list and a corresponding gap will take these off the list. The lexical entries (lookup) now have additional ‘movement list’ arguments (input list & output list) prefixed for clarity by an operator ‘moved’:

\[
\text{:- op(90, xfx, [struct]).}
\]
\[
\text{:- op(80, xfx, [moved]).} \quad \% \text{the new operator}
\]
\[
\text{lookup(what, [\text{det struct I moved (Mvd, [[1]] Mvd) }]).}
\]
\[
\text{lookup(will, [ v struct [\text{Comp, [\_np, will (Comp)]] moved (Mvd, Mvd) ]].}
\]
\[
\text{lookup(peter, [ p struct peter moved (Mvd, Mvd) ]).}
\]
\[
\text{lookup(bring, [ v struct [[\text{Subj, bring_to (Subj, Obj To_obj)}], Obj, To_obj] moved (Mvd, Mvd) ]).}
\]
\[
\text{lookup(to, [ p struct to moved (Mvd, Mvd) ]).}
\]
\[
\text{lookup(the, [ det struct the moved (Mvd, Mvd) ]).}
\]
\[
\text{lookup(party, [ n struct party moved (Mvd, Mvd) ]).}
\]

Moved elements will be indexed to signify the fact that they have undergone movement. For simplicity we will store the index in the database but for greater efficiency this should be passed around as a Prolog argument:

\[
\text{index(1) :-}
\]
\[
\text{ retract(ind(i)),}
\]
\[
\text{ i is 1 + 1}
\]
\[
\text{ asserta(ind(1))).}
\]
\[
\text{index(1) :- asserta(ind(1))).}
\]

The grammar is much the same as before, and is pre-processed as before, but also has ‘movement list’ arguments:

\[
\text{:- op(1000, xfy, ['="=>'])).}
\]
\[
\text{:- dynamic '"="=>'/2.}
\]
\[
\text{ s struct VP moved (M, M1) } \Rightarrow \text{ np struct NP moved (M, M2),}
\]
\[
\text{ vp struct [NP, VP] moved (M2, M1).}
\]
np struct DET moved (M, M1) \implies det struct DET moved (M, M1).

np struct NP moved (M, M1) \implies pn struct NP moved (M, M1).

np struct [DET, N] moved (M, M1) \implies det struct DET moved (M, M2),
\hspace*{1cm} n struct N moved (M2, M1).

np struct E moved (M, M1) \implies e struct E moved (M, M1).

sl struct VP moved (M, M1) \implies
\hspace*{1cm} np struct NP moved (M, M2),
\hspace*{1cm} vp struct [subj = NP, VP] moved (M2, M1).

vp struct V moved (M, M1) \implies
\hspace*{1cm} v struct [V, obj = NP, to_obj = PP] moved (M, M2),
\hspace*{1cm} np struct NP moved (M2, M3),
\hspace*{1cm} pp struct PP moved (M3, M1).

vp struct V moved (M, M1) \implies
\hspace*{1cm} v struct [Sl, V] moved (M, M2),
\hspace*{1cm} sl struct Sl moved (M2, M1).

np struct [P, NP] moved (M, M1) \implies
\hspace*{1cm} p struct P moved (M, M2),
\hspace*{1cm} np struct NP moved (M2, M1).

An 'e' edge is put into the initial Chart base at every vertex. The procedure 'make' is called once for each base item ('N' times). If we want to hypothesize gaps at every vertex then an extra gap edge (i.e. N + 1 gap edges) must be added. This is simple enough to do but as an initial gap is linguistically unlikely we need only hypothesize gaps from vertex 2 to N:

\begin{verbatim}
make(_, [[Ev, Ev, Info]|Rest], [], _, Ev, Rest) :-
gap_entry(Ev).

make(Word, [[Sv, Ev, Info]|Rest_base], [Info|Rest], Sv, Ev, Rest_base1) :-
make(Word, Rest_base, Rest, Sv, Ev, Rest_base1).
\end{verbatim}

The predicate 'gap_entry' defines the structure and movement lists of a gap. The structure of a gap, according to the crossing limit (Ref. 6), p. 262) (in English) is quite simply the first structure taken from the movement list:

gap_entry(e struct Mvd_struct moved ([Mvd_struct|Rest], Rest)).

In other languages a gap structure might be taken from further down the movement list.

The parser initialises the base in the same way as before and then calls the procedure 'elevate' to work up from this base to produce the target edge. The target edge is now:
[1, Lv, Cat struct S moved (M, M)].

That is an edge spanning the entire base (1 to Lv as before) with structure ‘S’ where the movement list is unchanged (has no remaining elements). The parser creates edges and invokes new rules as before but now also unifies the movement lists to pass on the list of moved elements:

% (1) no more active edges to match, try invoking new rules.
enter(Edge, [], Target, Active, Base):-
    invokes(Edge, Active, Active1, Base, Base1),
    elevate(Base1, Active1, Target).

% (2) edge matches next active edge to form new active edge.
enter([[Sv, Ev, Cat struct S moved (M, M1)]],
    [[Sv1, Sv, Head, (Cat struct S1 moved (M2, M3), Rem)][Rest]],
    Target, Active, Base):-
    copy_term((S, M, M1, Head, S1, Rem, M2, M3),
              (S2, M4, M5, Head, S2, Rem1, M4, M5)),
    Ev elem_of Active has_value Old_active,
    Ev of Active := [[Sv1, Sv, Head1, Rem][Old_active] in Active1],
    enter([[Sv, Ev, Cat struct S moved (M, M1)]],
          Rest, Target, Active1, Base).

% (3) edge completes an active edge to produce the target edge
enter([[Sv, Ev, Cat], [[Sv1, Sv, Head, Cat][], [Sv1, Ev, Head], __ __]).

% (4) edge completes an active edge, add new edge to base (LIFO).
enter([[Sv1, Sv, Head, Cat struct S moved (M, M1)]],
    [[Sv1, Sv, Head, Cat struct S1 moved (M2, M3)][Rest]],
    Target, Active, Base):-
    copy_term((S, M, M1, Head, S1, M2, M3), (S2, M4, M5, Head1, S2, M4, M5)),
    enter([[Sv, Ev, Cat struct S moved (M, M1)]], Rest, Target, Active,
           [[Sv1, Ev, Head1][Base]]).

% (5) edge does not match first active edge, try the next.
enter(Edge, _, [Rest], Target, Active, Base):-
    enter(Edge, Rest, Target, Active, Base).

invokes([[Sv, Ev, Cat struct S moved (M, M1)]],
        Active, Active1, Base, Base1):-
    Cat rules (Started, Completed),
    Ev elem_of Active has_value Old_active,
    now_active([[Sv, Ev, Cat struct S moved (M, M1)]],
                Started, Old_active, New_active),
    Ev of Active := New_active in Active1, % new may be []
    now_base([[Sv, Ev, Cat struct S moved (M, M1)]],
              Completed, Base, Base1).

invokes(_, Active, Active, Base, Base). % no rules invoked by edge.
now_active(_, [], Active, Active) :- !.
now_active([Sv, Ev, Cat struct S moved (M, M1)],
    [[Head, (Cat struct S moved (M, M1), Remainder)]|Rest],
    Old_active, [[Sv, Ev, Head, Remainder]|Rest_new]) :- !,
    now_active([Sv, Ev, Cat struct S moved (M, M1)], Rest,
    Old_active, Rest_new).
now_active(Edge, [First|Rest], Active, Active1) :-
    now_active(Edge, Rest, Active, Active1).
now_base(_, [], Base, Base) :- !.
now_base([Sv, Ev, Cat struct S moved (M, M1)],
    [[Head, Cat struct S moved (M, M1)]|Rest], Base, Base1) :- !,
    now_base([Sv, Ev, Cat struct S moved (M, M1)], Rest,
    [[Sv, Ev, Head]|Base], Base1).
now_base(Edge, [First|Rest], Base, Base1) :-
    now_base(Edge, Rest, Base, Base1).

The output illustrates how a structure removed from surface has been created by movement of the WH-front element to a position where it functions as direct object.

?- parse([what, will, peter, bring, to, the, party], Semantic),
    write(Semantic).
will(bring_to(subj = peter, obj = [1], to_obj = [to, [the, party]]).

To improve efficiency we might restrict gap hypothesis to positions in the base where they may occur. This might be done using neighbouring categories. For example gaps may not occur between a determiner and a noun as these will be expected to form a composite noun phrase with no intervening gaps. The upward percolation of hypothesized gaps can be limited in LFG by the use of bounded nodes. These are categories in the grammar past which elements in the movement list cannot be passed. Both of these measures are partly linguistically rather than purely efficiency motivated measures.

§7 Conclusion

The flexibility of the Chart algorithm allows for experimental "fine tuning" of the parsing algorithm (bottom-up and top-down balancing). The Chart algorithm also allows an arbitrary selection of the edge to be entered onto the Chart, thus the degree to which different parses are pursued and hence the degree of parallel development can be varied. The W1 algorithm fits more naturally into Prolog and is more efficient both in time and in space.

We do not view the fixed development of parses in parallel as a weakness. Generally the parser cannot tell which parses will complete successfully so that all possibly parses should be developed. Developing parses in series was, of course, the great weakness of the original direct Prolog (TDB) parsers. It is also most frequently required that all successful parses be produced and not just the
first.

The WI algorithm was implemented in Quintus Prolog on a Sun 3 workstation and was used as the parser for a LFG system. It was found after initial tests that the algorithm was approximately five times faster than a similar implementation of the basic Chart algorithm.

Acknowledgements

We gratefully acknowledge financial support from the Science and Engineering Research Council (SERC) and the Speech Research Unit, RSRE (Malvern), and help and advice from Peter Greenfield of the University of Birmingham.

References

Appendix

The Prolog functor/argument structure is used to support simple one dimensional arrays (limited to 255 elements in Quintus Prolog). Quintus supports simple Prolog modules which require a declaration of exported predicates and their arity:

```prolog
:- module(arrays, [new_array/1, elem_of/2, ':='/2]).
% module name.
% exports.
% return a new array structure.
% get the value of an element.
% assign the value of an element.
```

Prolog operators are declared to support the exported predicates and provide some syntactic niceties:

```prolog
:- op( 110, xfx, ['=', changed_to]).
:- op( 100, fx, [new_array]).
:- op( 100, xfx, [elem_of]).
:- op( 95, xfx, [elements, made, has_value, of]).
:- op( 90, xfx, [size, in]).
```

The predicates to support arrays are then declared. We use the functor/arg format because this allows direct use of the Prolog primitives 'arg' and 'functor' to get array elements and build new arrays.

```prolog
new_array - A size + S elements + V.
create a new array 'A' of size 'S' where each element is 
initialised to have value 'V'.  */

new_array A size S elements V :-
new_array A size S, % create the array.
A elements V. % initialise the elements.

new_array - A size + S.
create an array 'A' of size 'S' (each element a different 
Prolog variable).  */

new_array A size S :-
functor(A, a, S), % 'a' is a dummy functor for arrays.

+A elements + V.
make each element of array 'A' have value 'V'.  */

A elements V :-
a =. [a|Elements], % convert the array to a list.
Elements made V. % make each list member 'V'.

+L made + V.
make each (Prolog variable) member of list 'L' equal to 'V'.  */

[] made _ :- !.
[V|Rest] made V :-
Rest made V.
```
/* +Nth elem_of +A has_value -V. 
find the value 'V' of the 'Nth' element of array 'A'. */
Nth elem_of A has_value V :-
    arg(Nth, A, V).

/**************************
change 'Nth' element of array 'A' to be 'V' in array 'A1'. */
Nth of A := V in A1 :-
    A =.. [a|Elements],
    Nth of Elements changed_to V in New_elements,
    A1 =.. [a|New_elements].

/**************************
* Nth of +L changed_to V in +L1. 
changed the Nth member of list 'L' to 'V' in 'L'. */
1 of [..|Rest] changed_to V in [V|Rest] :- !.
Nth of [First|Rest] changed_to V in [First|New_rest] :-
    Nth1 is Nth - 1,
    Nth1 of Rest changed_to V in New_rest.