# Parsing Generalized Phrase Structure Grammar with Dynamic Expansion

Navin Budhiraja
Subrata Mitra
Harish Karnick
Rajeev Sangal

Department of Computer Science and Engineering
Indian Institute of Technology Kanpur
Kanpur 208 016 India

**SUMMARY**

A parser is described here based on the Cocke-Young-Kassami algorithm which uses immediate dominance and linear precedence rules together with various feature inheritance conventions. The meta rules in the grammar are not applied beforehand but only when needed. This ensures that the rule set is kept to a minimum. At the same time, determining what rule to expand by applying which meta-rule is done in an efficient manner using the meta-rule reference table. Since this table is generated during "compilation" stage, its generation does not add to parsing time.

## 1 INTRODUCTION

GPSG as introduced by Gazdar et.al. gives a formalism to parse natural languages assuming they are context free. The phrase structure rules are like the normal CFG rules, except that features are added to the categories. These features are used by Feature Co-occurence Restrictions, Feature specification Defaults, Head Feature Convention, Foot Feature Principle and Control Agreement Principle, during parsing.

The second important feature of GPSG, and towards which this paper is mainly directed, is the metarule. A major problem of a complete natural language grammar is its size, which causes difficulties as far as memory requirements and efficiency of any practical parser are concerned. GPSG tries to overcome this, partly, by keeping the rule set to the minimum. In addition to the minimal set of rules, it has certain metagrammatical structures to generate rules from the previously defined minimal set. Thus the number of rules at any time are the minimum possible, reducing the search time of the parser. In addition, this captures certain linguistic generalisations (e.g. active-passive).

Lastly GPSG goes to the thematic representation directly from the c-structure (in contrast to other formalisms like LFG). The IL formula is built up as parsing proceeds.

Our endeavour is, thus, to build a natural language parser

incorporating all of the above. We describe the parser in the sections that follow.

### 1.1 EXISTING PARERS AND OUR APPROACH

All the implementations of GPSG reported in the literature use a rather straight forward approach of first expanding the entire rule set by using the available metarules, in the process augmenting the set of rules, and finally the normal context free parser is run on this new set of rules.

Thus there are two basic steps involved :-

1. Rule expansion using the available metarules
2. Actual parsing using the expanded set of rules.

It should be noted that in such an implementation one does not need to bother about the metarules after the first stage.

An inherent drawback with this approach is that if the initial set of rules is of sizeable cardinality, then a number of rules may get added to the set, (a large number of these rules may never get used during the actual parse of a sentence), thus not only causing memory storage problems, but also slowing down the system considerably.

The main motivation of this paper is to describe a method for parsing GPSG without initial expansion (i.e. our implementation expands metarules as and when necessary). Further, in our implementation we have assumed in ID-LP format for the rules, thus making them more compact.

Because of the above reasons, it has become necessary to make some changes to an ordinary context-free parsing algorithm to suit our requirements (i.e. to incorporate dynamic expansion and the ID-LP format of rules).

## 2 PARSING ALGORITHM

The essential characteristics of our approach towards a solution of the problem has been listed over the next few pages.

### 2.1 DYNAMIC STRUCTURE OF RULES

As has been mentioned earlier, our implementation gets new rules from old ones as the parsing proceeds. Under such a situation, it becomes necessary to suspend parsing temporarily, only to return to it after a rule of the appropriate type has been generated by expanding sing one or more metarules some appropriate rule from the already available set.

At this stage a decision has to be taken as to whether the rule which was recently derived should be stored for further use or should be discarded. Here the choice should be guided by the

relative gain in time by storing the rule (as opposed to re-expanding) against the storage overhead. The type of sentences to be parsed may also play a role in this decision. For example, it may be worthwhile to store the rule which gets generated during parsing. Equivalently the other approach may be tried.

For this type of implementation, metarules become in important part of parsing. Further justification on this issue is given in the next section.

## 2.2 TABLE BUILDING

We have seen in the previous section that an important aspect of our parser implementation is the generation of rules at an intermediate stage.

A native way to tackle the problem would be to go over the entire set of rules and metarules when a failure occurs, and try each metarule-rule combination to find one which produces a rule of the required type, and then carry on with the parser. But this will obviously be highly inefficient.

To cut down the time of generating rules and trying them out, a table can be constructed to help us select the metarule-rule combination. This is what is done.

In the first stage (called the metarule compilation stage) we go over all the rule-metarule combinations to build up a reference table which can be consulted by the parser (during the second stage) to get the required rules efficiently. Compilation is a one time job and, therefore, does not affect the complexity of the actual parser.

The rule set can be structured for faster access to the relevant rules. In our current implementation we have structured the rule set on the basis of the number of categories (non-terminals) on the right-hand-side (rhs henceforth).

The metarule reference table is built up in the compilation stage as follows :-

1. For each rule r do the following.
2. Store the rule in the appropriate entry of a new table called the RULE TABLE, which is the one that the parser refers to. (For our case store it with all other rules which have the same number of rhs categories).
3. For each metarule m that can be applied on r do
   3.1 apply m on r yielding a new rule s
   3.2 hash the rhs categories of the newly produced rule s to get an index into another table called the META-REFERENCE table.
   3.3 Build up a meta reference entry as follows :-
   (a) Get index (here number of rhs categories) of the input rule r. Let this be I1.

(b) Get the position of the current rule in the rule table corresponding to I1. Let this be I2.
(c) Get the position of the current metarule m in the metarule list. Let this be M1.

3.4 Now append the triplet (I1 I2 M1) to the contents of the meta reference table entry pointed to be the index found in step 3.2 above.

The above takes care of cases where one level of expansion of metarules is sufficient. But in general a rule could be expanded successively more than once by the same or by different metarules before it can be used for parsing. Thus it is necessary to extend the meta-reference entries to handle the problem.

Basically a triplet (I1 I2 M1) as defined above corresponds to a rule which is produced by applying metarule M1 to the I2th. entry in the I1th. sub-structure of the rule table. Let the resultant rule be R1. Now R1 may expand some metarule whose position is M2 to produce a rule R2, and so on, until at some stage we get a rule Rn which is not meta expandable any further. The termination is guaranteed because GPSG is equivalent in power to CFG.

In the compilation stage we must now make entries in the meta-reference table for each of R1... Rn, because any of them may be necessary during parsing. This can be done as follows:-

```
R1 is the rule corresponding to (I1 I2 M1)

For i:=1 to n do
   Ri  is Mi applied on R(i-1),
  Get an index to the meta-reference table using rhs
                             categories of Ri
  To this entry append ((I1 I2 M1) M2...Mi)
```

Here M1,M2...,Mi gives the successive position in the metarule list of the metarules to be applied.

An example will clarify the situation:
consider a metarule of the form
(VP--> W NP)====>(VP--->W (optional(PP[by]))).
where W is any set of categories.
This generates the passive counterparts of active sentences.

Now if we have a rule of the type
0..VP--> V NP NP,
/* The features etc. have been omitted for simplicity * /
after first expansion we shall get two rules, namely:

1..VP--> V NP,
2..VP--> V NP PP.

Further, because of the given structure of the intermediate rules and the metarules under consideration, a second expansion is possible. Consider the rule VP--> V NP PP (rule 2 above). When the above metarule is expanded using this rule, we get the

following pair of rules:

3.. VP--> V PP
4.. VP --> V PP PP

Similarly the rule VP --> V NP (rule 1 above) will generate two rules of the form:

5..VP --> V
6..VP --> V PP

Now since none of the newly generated rules are meta-expandable the process will stop. The meta referencea table entries will be of the following nature :

/* Let us assume that there is just this one metarule in the meta-list, and that the initial rule (rule 0 above) is the only one present in the rule array corresponding to length of rhs three,i.e,

I1 is 3
I2 is 1, and
M1 is 1

*/ (a) Corresponding to rhs <V,NP,PP> and <V,NP> we shall have entries of the type ((3 1 1)),while
(b) Corresponding to any other possible collection of rhs categories, for example <V,PP>, the entry will look like ((3 1 1 ) 1), which incorporates two levels of meta expansion.

A point of importance is that since one expansion of a metarule can produce more than one output rule (e.g. rules (1) & (2) from rule (0) above ) the meta expander must check for category names before returning the generated rule.

For example if meta expansion is called with parameters (V,NP)in the above situation, then only rule (1) should be returned, the other has to be discarded .

Another change could be incorporated regarding the structuring of the set of input rules. One can use a hashing technique similar to the one used for storing meta reference entries. Thus, rules would be stored not by the number of rhs categories, but hashed according to the categories present in the rhs. This would make rule access at parse time much faster and direct because during a bottom-up parse we have to reduce a given set of rhs categories into the corresponding left hand side. This would however mean keeping more entries in the rule table.

## 2.3 THE PARSING ALGORITHM

The parsing algorithm we have used is the well known Cocke-Young-Kassami (CYK) algorithm, with a few modifications. The differences are for the following requirements. We have to :

1) make the algorithm work for an ID/LP grammar,
2) make the algorithm work for grammars not in Chomsky Normal Form (CNF),
3) allow for meta-rule expansion during parsing

We discuss these one by one.

1) In order to handle ID/LP grammars, we have to just look for a rule with the required nonterminals on the right hand side, with no importance attached to the order (except of course, for precedence relations)

2) In order to account for grammars which are not in CNF, we had to increase the nesting of the loops which handle rules of the form

A----> B1 B2

in the CYK algorithm. The loop depth should now be (k-1) in order to handle rules like

A----> B1 B2. ..Bk

(See algorithm extract given below)

3) In order to get new rules from the old, we have to make some additions to the CYK algorithm. A part of the algorithm is given below :-

```
/*
    The algorithm to handle grammars not in CNF and to
    allow for metarule application during parsing is
    shown below. This handles all rules which have k
    nonterminals on the right hand side.
*/
procedure length_k(i,j) ;
begin
    for a1 := 1 to j-k+1 do
      for b1 := 1 to j-a1-k+2 do
        for c1 := 1 to j-a1-b1-k+3 do
              .
              .
              .
          for j1 := 1 to j-a1-b1-c1...-i1-k+j do
              RULESET := RULESET U { new rules obtained by
                                     expanding the metarules
                                     as required by the
                                     parser }
              /* it is in the above line that we get the new
                 set of rules as demanded by the parser */
               CYK(i,j) :=
                          CYK(i,j) U {A | A---->B1B2...Bk
                             is a production, and
                                B1 is in CYK(i,a1),
                                B2 is in CYK(i+a1,b1)
                                .
                                .
                                .
                                Bk is in CYK(i+a1+b1...+j1,
```

```
                                            j-a1-b1...-j1) }
          end;
```

As can be seen from the algorithm extract given above, the meta-rules are expanded here. Once we have the required RHS (B1,B2...Bk), it is hashed to a value in the meta-rule reference table which returns us a triplet of the form (I1 I2 M1) where

I1 stands for the index of the rule-table i.e the table which contains all the rules according to the number of right hand sides they have.

I2 stands for the number of the rule in the I1 entry of the rule index table

M1 stands for the number of the metarule which needs to be used.

We discuss an example to illustrtate the algorithm.

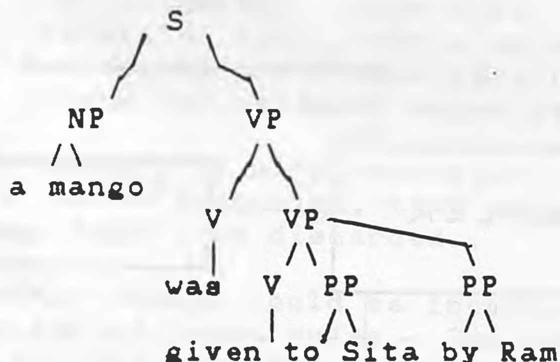
Example: In the parsing of the sentence

A mango was given to Sita by Ram

the rule

VP--> V PP[pform to ] PP[pform by ] --- (1)

is required.




Initially we only have the rule

VP --> V NP PP --- (2)

and the meta rule

VP-->W NP ===>

VP[vform pas] ---> W(PP[pform by]) --- (3)

Suppose (V PP PP) hashes to 20. Also assume that the meta rule (3) is the first meta rule in the meta rule list and rule (2) is the fourth rule in the rule list which contains all rules which

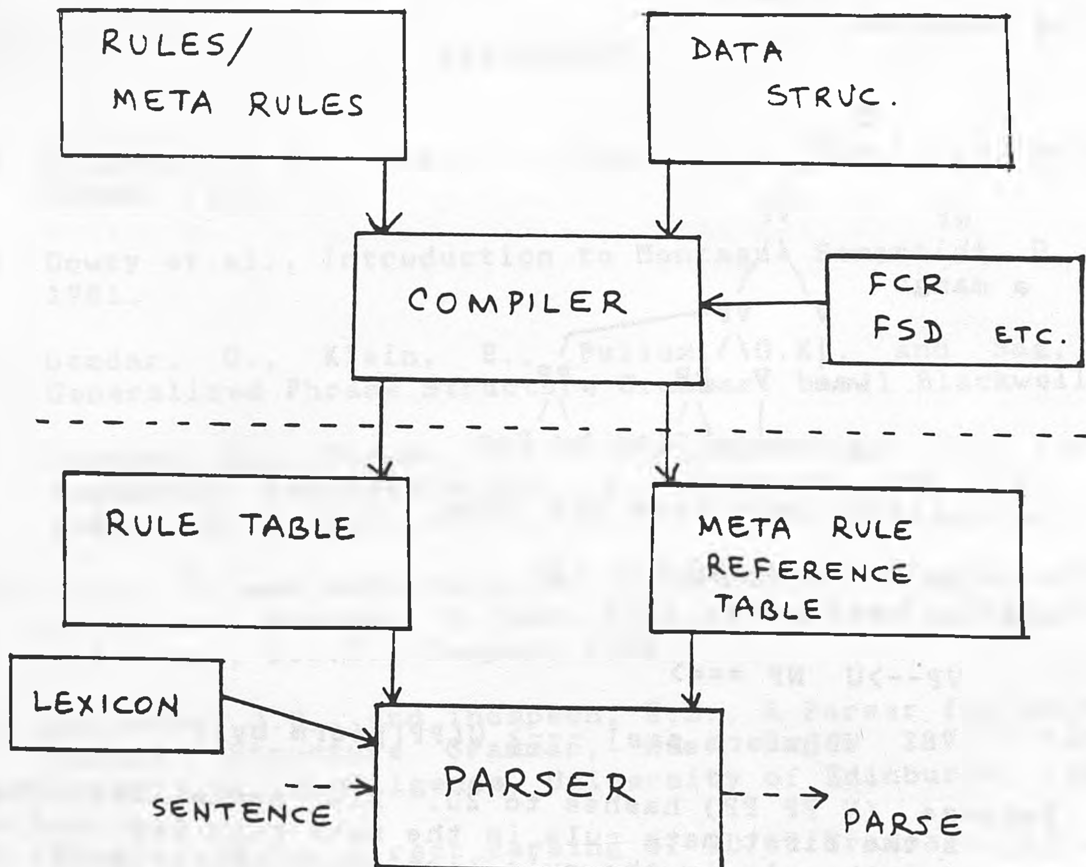
have three right hand sides. Now when the parser sees that it requires a rule containing V PP PP (as in (1)) it makes the following call

```
(return-meta-expanded '(V PP PP))
```

The triplet that is obtained from the table lookup is (3 4 1) which calls the meta rule expander to apply (3) to (2) which returns (1). The parser then continues its normal course after adding the generated rule to the appropriate rule-list.

## 2.4 THE SYSTEM STRUCTURE

The block structure of the compiler and parser is given below with the dotted-line separating the two. The part above the dotted line is done only once when the grammar is "fed" in. First the rules specified by the grammar designer are stored in an appropriate data structure. The compiler then applies the various feature restriction principles to this rule set (similar to the Edinburgh approach described in Philips (86)), makes the feature bindings and then indexes them according to the number of categories on the right hand sides. In addition it also creates the all important Meta-rule reference table. Both these tables are then passed to the parser which then, using the lexicon, works as described before.



## 2.5 HANDLING FEATURE RESTRICTIONS

The heart of GPSG is made up of the set of <feature, feature-value> pairs associated with every syntactic category. GPSG introduces some rule and conventions to associate values with these features in required manner.

Some of these restrictions should cause values to be given to features during actual parsing of a sentence, while others should pass up the tree certain feature values which get instantiated at parse time to ensure a valid parse.

Our implementation handles such problems at the compilation stage by considering fully expanded categories, where feature values corresponding to a particular feature which is as yet uninstantiated are bound to a unique variable, and the variable is shared among all instances of the same feature in the rule, which have to be bound together. This approach is similar to the Edinburgh parser.

Later, during the actual parse, if any variable gets bound to a value, then all other instances of the same variable in the rule also get the same value. Any mismatch leads to rejection.

For example, in the rule

A---> B1, B2...Bk

the variable valued features in A get bound to their values as instantiated in B1,B2...Bk. We are assuming that the RHS of a rule is fully instantiated during the parse i.e once a category is added to the CYK table, no more features are added to it. This approach has forced us to use multiple entries in the lexicon.

For example, the entry for 'the' contains two entries, one each for singular and plural respectively.

## 3 SEMANTICS

The IL formula for the input sentence is built up as the parsing proceeds. Each node in the parse tree being built contains the IL formula of the node. Using the type information and the Semantic Interpretation Schema, the IL formula of the mother is built up from the IL formulae of its children. Finally the node S (the start symbol) contains the IL formula of the input sentence. After parsing finishes, transformations as required by GPSG (e.g. the passive-active transformation, paraphrases etc.) are applied to the IL formula of the root.

## 4 CONCLUSIONS

The parser described here uses immediate dominance and linear precedence rules together with various feature inheritance conventions. The meta rules in the grammar are not applied

beforehand but only when needed. This ensures that the rule set is kept to a minimum. At the same time, determining what rule to expand by applying which meta-rule is done in an efficient manner using the meta-rule reference table.

The Cocke-Young-Kassami algorithm has been modified to work on the context free grammar without converting it to Chomsky Normal form. Conversion would lead to an increase in number of rules, and would also affect the dominance relationships. The modified algorithm continues to be a polynomial time algorithm on the length of the input sentence.

The implementation of the parser has been tested with a small grammar and with a small number of meta rules. To get performance figures, it needs to be tested more extensively. Experiments can also be conducted regarding when the generated rules should be stored for future use and when they should be discarded.

Our parser, at the moment, does not have the Kleene Closure facility to handle conjunctive/disjunctive sentences. It is a simple matter, however, to add this.

**ACKNOWLEDGEMENT**

Insightful Suggestions by Vineet Chaitanya and B.N. Patnaik thro--ughout the course of this work are gratefully acknowledged.

## REFERENCES

[1] Allwood et.al., Logic in Linguistics, Cambridge University Press, 1977.

[2] Dowty et.al., Introduction to Montague Semantics, D. Reidel, 1981.

[3] Gazdar, G., Klein, E., Pullum, G.K., and Sag, I.A., Generalized Phrase Structure Grammar, basil Blackwell, 1985.

[4] Gazdar, G., Phrase Structure Grammar, in The Nature of Syntactic Representation, P. Jacobson and G.K. Pullum (eds.), D. Reidel, 1982.

[5] Mitra, S. and Budhiraja, N., A Parser for Generalized Phrase Structure Grammar, B.Tech. thesis, Dept. of Computer Sc. and Engg., I.I.T., Kanpur, 1988.

[6] Phillips, J.D., and Thompson, H.S., A Parser for Generalized Phrase Structure Grammar, Res. Paper 289, Dept. of Artificial Intelligence, University of Edinburgh, 1986.

[7] Shieber, S.M., Direct Parsing of ID/LP Grammars, Linguistics and Philosophy, 7,2.