TR-252

# A PARSING SYSTEM BASED ON LOGIC PROGRAMMING

by

Y. Matsumoto and R Sugimura

April, 1987



ICOT

Mita Kokusai Bldg. 21F
4-28 Mita 1-Chome
Minato-ku Tokyo 108 Japan

(03) 456-3191~5
Telex ICOT J32964

Institute for New Generation Computer Technology

# A PARSING SYSTEM BASED ON LOGIC PROGRAMMING

Yuji Matsumoto and Ryoichi Sugimura
Institute for New Generation Computer Technology
1-4-28 Mita, Minato-ku,Tokyo, 108, Japan

## ABSTRACT

The paper presents a practical parsing system based on logic programming. A restricted Definite Clause Grammar is assumed as grammar description and the grammar is translated into a parsing program written in Prolog. The system employs a bottom-up parsing strategy with top-down prediction. The major advantages of our system are that the system works in a bottom-up manner so that the left-recursive rules do not cause difficulties, the parsing process does not involve backtracking, and there is no duplicated construction of same syntactic structures. Experiments are shown to estimate the efficiency of the system.

## I INTRODUCTION

This paper presents a practical parsing system based on logic programming. Although the key algorithm of the system originates from the authors' idea on parallel parsing [Matsumoto 86], it provides a quite efficient parsing environment even in a sequential implementation. We also present a grammar description method which is actually a subset of Definite Clause Grammar (DCG) formalism [Pereira 80]. The restriction given to DCG guarantees that the parsing system operates efficiently. We do not think this restriction is too severe for grammar writers.

The current parsing system is called SAX, while the parallel implementation is called PAX. In both of them, all the grammatical symbols such as noun phrases and verb phrases as well as lexical symbols are defined as predicates of Prolog or of the parallel logic programming language the system is implemented in. In this sense it resembles DCGs translated into Prolog programs. The major advantages of our system are that it works in a bottom-up manner so that the left-recursive rules do not cause difficulties, the parsing process does not involve backtracking, and there is no duplicated construction of syntactic structures. Our previous bottom-up parsing system, BUP [Matsumoto 83], has similar charasteristics and we have almost equal performance from both of them when they are executed by the Prolog interpreter. However, SAX is nearly one order of magnitude more efficient when they are both compiled. This is because BUP keeps partial parsing results by side-effect whereas they are represented as processes in SAX.

As described above, the basic algorithm is based on our parallel parsing method. The current system is specialized for sequential implementation. The next section describes the basic algorithm of our parsing system and shows how grammar rules are translated into a Prolog program. Section 3 explains how grammar rules are written and examines the performance of the system by a sample English grammar and shows that we have achieved sufficient efficiency in parse time.

## II OVERALL DESCRIPTION OF THE SYSTEM

This section briefly describes the organization of SAX. It is convenient to describe first the basic algorithm for context free grammars. Suppose we have a context free grammar shown in (1). In the right-hand side of the grammar rules, the symbol '*id*' followed by a figure stands for an identifier that indicates a particular position in a particular grammar rule and is not a grammatical symbol. When these rules are seen as grammar rules (or more precisely as DCG rules) they should be neglected. As a matter of fact, users need not specify these identifiers. They are automatically assigned by the SAX translator, which generates a parser from the grammar. We ommit the lexical part in the following grammar rules.

```
(1)  sentence --> np, id1 vp.
     np --> det, id2 noun.
     np --> np, id3 coconj, id4 np.
     noun --> noun, id5 rel_clause.
     noun --> noun, id6 pp.
     rel_clause --> [that], id7 vp.
     pp --> prep, id8 np.
     vp --> verb.
     vp --> verb, id9 np.
     vp --> vp, id10 coconj, id11 vp.
```

The parsing process operates from left to right and from bottom to top, ie, from surface words to more well-formed tree structures. Suppose a noun phrase has just been found, there are two kinds of processes that must be performed according to the grammar rules. The first is to start parsing by using new grammar rules. The other is to augment already constructed incomplete tree structures to more complete ones. As mentioned in the introduction, all grammatical symbols are defined as predicates of the Prolog program. Therefore, the discovery of a noun phrase corresponds to a call of the definition of np. Since the parsing process proceeds from left to right and bottom to top, a call of np produces identifiers id1 and id3 which indicate that the parsing process has successfully proceeded up to these points of the grammar rules. It is defined as a Prolog clause (2).

```
(2)  np1(X,[id1(X),id3(X)|Yt],Yt).
```

The second and third arguments of the clause represent a set of these two identifiers by a difference list. The first argument of this clause is a list of identifiers produced by the words or grammatical symbols just preceding the noun phrase in the given input sentence. The clause produces these identifiers without regard to its contents. It will be, however, modified when top-down prediction is made use of. This clause corresponds to the first job for a noun phrase mentioned above and the meaning is that by finding a noun phrase these two rules are possibly used to build up new parsing tree structures. For example, if id1 is received by a verb phrase this means that a sentence is found. Similarly, the second job for the noun phrase is to build up more complete tree structures using partially constructed trees and is defined by the Prolog clauses shown in (3).

```
(3)  np2([],X,X).
     np2([id4(X)|Xt],Y,Yt) :-
          np(X,Y,Y1), !,np2(Xt,Y1,Yt).
     np2([id8(X)|Xt],Y,Yt) :-
          pp(X,Y,Y1), !,np2(Xt,Y1,Yt).
     np2([id9(X)|Xt],Y,Yt) :-
          vp(X,Y,Y1), !,np2(Xt,Y1,Yt).
     np2([_|Xt],Y,Yt) :- np2(Xt,Y,Yt).
```

The second clause of (3) says that it can construct a noun phrase if it receives id4, which is only produced by a coconj(coordinating conjunction) that has already received a noun phrase. The third and fourth clause correspond to the other occurrences of np in the grammar rules. The first clause specifies that it produces an empty difference list when it receives an empty list. The last clause is necessary to discard the identifiers irrelevant to a noun phrase.

The definition (2) is for the occurrences of noun phrases as the left-most element in right-hand side of grammar rules, and the definition (3) is for the other occurrences of noun phrases in right-hand side of grammar rules. We call them type-one occurrence and type-two occurrence, respectively. The complete definition of a noun phrase is just a union of these definitions as shown in (4).

```
(4)  np(X,Y,Yt) :- np1(X,Y,Y1),np2(X,Y1,Yt).
```

If a grammatical symbol appears in grammar rules only as either of type-one and type-two occurrence, a clause like (4) is not necessary for that grammatical symbol. coconj is an example and is defined like (5). It has merely type-two occurrences.

```
(5)  coconj([],X,X).
     coconj([id3(X)|Xt],[id4(X)|Y],Yt) :- !,
          coconj(Xt,Y,Yt).
     coconj([id10(X)|Xt],[id11(X)|Y],Yt) :- !,
          coconj(Xt,Y,Yt).
     coconj([_|Xt],Y,Yt) :- coconj(Xt,Y,Yt).
```

Lexicon is defined in a quite straightforward manner. All words can also be defined conceptually as Prolog predicates, though it is not practical when we have thousands of lexical elements. For the present, words are defined as (6), which says that 'the' is a determiner.

```
(6)  the(X,Y,Yt) :- det(X,Y,Yt).
```

Parsing of a sentence is done by calling the definitions of the words that comprises the sentence. For instance, (7) is the initial call for parsing 'The man walks.'

```
(7)  the([begin],X0,[]),man(X0,X1,[]),
          walks(X1,X2,[]),fin(X2).
```

In order to specify that the grammatical symbol the parser is looking for is a sentence, the following definition must be added to the type-two definition of sentence.

```
(8)  sentence2([begin|Xt],[end|Y],Yt) :- !,
          sentence2(Xt,Y,Yt).
```

begin is a special identifier to indicate the beginning of a sentence, and end is produced only when a sentence is found from the head of the input sentence. It is now very clear that the parsing of a sentence succeeds when the identifier end is produced by the last word of the input sentence. fin is the predicate to recognize end.

This summerizes briefly the basic algorithm of the system. One more thing we have to mention is top-down prediction to reduce the search space of the parsing process. Type-one clauses like (2) produce identifiers without regard to the contents of the list they receive. Since the received list consists of the contexts just before the word or grammatical symbol that calls the type-one clause, the parsing space can be reduced by referring to the contents. The list it receives is a list of identifiers and each identifier has its own expecting grammatical symbol. For instance, id1 is expecting a verb phrase, id2 is expecting a noun, and so on. The production of an identifier by a type-one clause means the use of the grammar rule which the identifier belongs to. As for the clause (2), id1 corresponds to the use of the first grammar rule in (1). If sentence is not expected or if any grammatical symbol that can be a root of a tree with a sentence as its left-most leaf is not expected by any of the elements in the list it receives, it is useless to produce id1. Top-down prediction can be realized as a filtering process in our parsing system. A filtering process is assigned to each identifier produced by a type-one clause, and it filters out all the unnecessary elements from the list (the context that preceeds it). If all the elements in the received list are filtered out, the current identifier need not be produced. (9) is the new definition of (2) that incorporates such a filter. (10) and (11) give the auxiliary predicates. tp_filter filters out all the unnecessary elements from X and produces New_X consisting of identifiers that are at least necessary. id_pair returns the grammatical symbol that the identifier is expecting. link checks whether the given two grammatical symbol can be related as parent and the left-most son of a tree. Definitions of these clauses are generated by the translator automatically. tp_out returns an empty difference list if New_X is empty and produces the identifier if New_X contains at least one element. id3 does not have a filter because the head of the original grammar rule it belongs to is np, the same grammatical symbol as itself.

```
(9)  np1(X,[id3(X)|Y],Yt) :-
       tp_filter(X,sentence,New_X),
       tp_out(New_X,[id1(New_X)|Xt],Xt,Y,Yt).

(10) tp_filter([],_,[]).
     tp_filter([Id|Ids],Term,[Id|New_ids]) :-
       functor(Id,Idntf,_),
       id_pair(Idntf,Top_cat),
       link(Term,Top_cat),!,
       tp_filter(Ids,Term,New_ids).
     tp_filter([_|Ids],Term,New_ids) :- !,
       tp_filter(Ids,Term,New_ids).

(11) tp_out([],_,_,Y,Y).
     tp_out([_|_],Y,Yt,Y,Yt).
```

## III SYSTEM AND PERFORMANCE

Careful readers may have noticed that there are some difficulties in implementing full DCGs in our parsing algorithm. Actually, ambiguities in grammar rules that are handled by backtracking in DCGs are expanded into processes, and they are solved in a single environment (this means there is no restoration of environments required by backtracking). DCG formalism is a context free grammar augmented by arguments in grammatical symbols and extra-conditions (expressed by Prolog programs). Since arguments in grammatical symbols of DCGs are represented as arguments of predicates also in SAX, they are treated in the same way. The simplest way to treat extra-conditions is to put them in the corresponding place in the transformed Prolog clauses. Unfortunately, they may then have different meanings for two reasons. Firstly, extra-conditions are evaluated only once in our system. In other words, only the first successful substitution to variables is computed. Secondly, since ambiguities spawn as many processes, the same variables that should be in different environments are inevitably treated in a single environment. This requires copying or renaming the variables, and that would cost much in both time and space.

To cope with these problems, we put some restrictions on the extra-conditions. That is, the extra-conditions evaluated dynamically in the parsing process must be deterministic and substitution to variables in the body of the grammar rules is prohibited. The second condition insists that only the variables in the heads of grammar rules are allowed to be instantiated. Thus, the flow of data must be from bottom to top. The form of a grammar rule of our system is defined like (12).

```
(12) a0 --> a1,{ extra_1 },... ,an,{ extra_n }
                & { delayed_extra }.
```

In this rule, 'ai' is a grammatical symbols possibly with arguments, and `extra_i` is an extra condition (a sequence of Prolog goals) evaluated dynamically. `delayed_extra` is also an extra-condition. It is, however, not evaluated dynamically. & is the special symbol to separate such an extra-condition from the syntactic description. As in DCGs, extra-conditions are written between braces ( { and } ) in grammar rules. In the actual implementation, extra-conditions separated by & are pushed into a stack-like data structure and they are evaluated after the successful termination of the parsing process. Such a parsing program is, of course, generated by the SAX translator. The user can also specify the strategy to evaluate these delayed extra-conditions, either in a top-down manner or a bottom-up and left to right manner. To give the flavour of the translation of such a grammar description into Prolog programs, (13) and (14) show the transformed clauses corresponding to a1 and an of (12), in which a11 is the type-one clause for a1, and an2 is the type-two clause for an. Of course, these definitions are more complicated if there are other occurrences of these nonterminal symbols in grammar rules. One extra argument is added to both grammar symbols and identifiers for carrying non-dynamic extra-conditions. `extra_1` is used as the condition to decide whether to produce the identifier. `EX` is (a set of) non-dynamic extra-conditions included in the grammar rule that has constructed a1. It is passed to the next process through the identifier. (14) succeeds only when `extra_n` is evaluated successfully. `delayed_extra` is passed to the head of the grammar rule forming a tree structure consisting of extra-conditions. Although the leaves are aligned in reverse order, they are evaluated according to the user's instruction.

```
(13) a11(X,EX,Y,Yt) :-
       tp_filter(X,a0,New_X),
     ( extra_1,
         tp_out(New_X,[id1(New_X,EX)|Xt],
                Xt,Y,Yt) ; Y = Yt ), !.

(14) an2([idn(X,EX1)|Tail],EXn,Y,Yt) :-
       extra_n, !,
       a0(X,[delayed_extra,EXn|EX1],Y,Y1),
       an2(Tail,LC,Y1,Yt).
```

The restriction to DCG formalism may seem a strong limitation in describing natural language grammars. The authors, however, do not think that this causes difficulties in writing grammars. In one view, it is a separation of the test procedures for checking grammaticality and the procedures for generation of meaning structures for the input sentence. The extra-condition `extra`'s work as a test procedure to determine whether to produce an identifier or a new process. For example, `extra_1` suppresses the production of the identifier put at the place between a1 and a2 if its evaluation fails. We recommend that users write test programs for reducing the parsing space as dynamic extra-conditions and write programs for constructing meaning structures that do not affect the syntactic well-formedness as non-dynamic extra-conditions.

The system has been tested by an English grammar with about 200 grammar rules and about 500 lexical entries. The grammar is based on that of Diagram [Robinson 82] with some modification. Most of the sample sentences are collected from the abstract of Robinson's paper and are listed in the Appendix. The time required to obtain all the parse trees are listed in Table 1. This experiment does not involve the morphological analysis, and inflections are treated by grammar rules. Comma is also defined as a lexical entry. The morphological analysis part of the system is

now under development. It will be implemented as a preprocess of the parser. It also employs a similar model to the parsing algorithm, which will be reported elsewhere. For the expreriment, we used Quintus Prolog on VAX 11/785 and ESP on PSI Machine, the Prolog Machine developed at ICOT. The speed of PSI is about 30 kiloLIPS (Logical Inferences per Second). The system is currently used as the syntactic analysis part of our Japanese discourse understanding system DUALS.

## IV CONCLUSIONS

This paper briefly introduced our parsing system based on logic programming. Let us summarize the main charasteristics of the system. The system employs a bottom-up parsing strategy so that left-recursive rules do not cause problems. The group of efficient parsing algorithms called tableau methods, such as Earley's algorithm [Earley 70] or Chart Parsing [Kay 80] use side-effects to keep intermediate parse trees. Our system creates processes (predicate calls) when intermediate parse trees are constructed. It gives the same effect to the parser without using any side-effects. The translated Prolog program is deterministic and it never backtracks. In particular, the definition of type-two clauses is a tail recursive program. These are the reasons of the efficiency of our system when it is compiled.

The translator from a grammar of the restricted DCG introduced in the preceeding section to a Prolog program has been developed. The parser automatically produces parse trees consisting of non-dynamic extra-conditions. If there is more than one parse tree, they are evaluated after renaming the variables since some parse trees may share logical variables.

Several projects are underway at ICOT concerning the system. A large Japanese grammar is under development, which will eventually run on the system. A morphological analysis part, together with automatic segmentation part for Japanese language are also under development. Efforts to extend the range of grammar formalism are also being made. As for syntactic description, we have already proved that even Gapping Grammars [Dahl 84a] [Dahl 84b] can be implemented in our framework [Matsumoto 87]. Some of our members are comparing our system with other general parsing systems.

## APPENDIX : Sample Sentences

1. He explain s the example with rule s. (8 words)

2. It is not tie ed to a particular domain of application s. (12 words)

3. Diagram analyze s all of the basic kind s of phrase s and sentence s and many quite complex one s. (21 words)

4. The annotation s provide important information for other part s of the system that interpret the expression in the context of a dialogue. (23 words)

5. Procedure s can also assign score s to an analysis, rate ing some application s of a rule as probable or as unlikely. (24 words)

6. This paper present s an explanatory overview of a large and complex grammar, Diagram, that is use ed in a computer for interpret ing English dialogue. (28 words)

7. Its procedure s allow phrase s to inherit attribute s from their constituent s and to acquire attribute s from the large er phrase s in which they themselves are constituent s. (32 words)

8. Consequently, when these attribute s are use ed to set context sensitive constraint s on the acceptance of an analysis, the contextual constraint s can be impose ed by condition s on constituency. (34 words)

**Table 1** : Parse Time for Sample Sentences

| Sentence Number | Number of Words | Number of Parse Trees | Parse Time* (msec) | Parse Time** (msec) |
|---|---|---|---|---|
| 1 | 8 | 2 | 160 | 60 |
| 2 | 12 | 1 | 290 | 126 |
| 3 | 21 | 3 | 540 | 239 |
| 4 | 23 | 7 | 1100 | 508 |
| 5 | 24 | 4 | 690 | 332 |
| 6 | 28 | 1 | 870 | 394 |
| 7 | 32 | 12 | 1530 | 741 |
| 8 | 34 | 4 | 820 | 375 |

* Time measured by compiled Quintus-Prolog on VAX 11/785
** Time measured by compiled ESP on PSI

## REFERENCES


[Dahl 84a] V. Dahl and H. Abramson, "On Gapping Grammars," Proc. 2nd International Conference on Logic Programming, Uppsala, Sweden, pp.77-88, 1984.

[Dahl 84b] V. Dahl, "More on Gapping Grammars," Proc. the International Conference on Fifth Generation Computer Systems, Tokyo, Japan, pp.669-677, 1984.

[Earley 70] J. Earley, "An Efficient Context-Free Parsing Algorithm," C.ACM, Vol.13, No.2, pp.94-102, 1970.

[Kay 80] M. Kay, "Algorithm Schemata and Data Structures in Syntactic Processing," Technical Report CSL-80-12, Xerox PARC, Oct. 1980.

[Matsumoto 83] Y. Matsumoto, et al., "BUP: A Bottom-Up Parser Embedded in Prolog," New Generation Computing, Vol.1, No.2, pp.145-158, 1983.

[Matsumoto 86] Y. Matsumoto, "A Parallel Parsing System for Natural Language Analysis," Proc. 3rd International Conference on Logic Programming, pp.396-409, London, 1986.

[Matsumoto 87] Y. Matsumoto, "Parsing Gapping Grammars in Parallel," to be published as ICOT Technical Report, 1987.

[Pereira 80] F. Pereira and D. Warren, "Definite Clause Grammars for Language Analysis -- A Survey of the Formalism and a Comparison with Augmented Transition Networks," Artificial Intelligence, Vol 13, pp.231-278, 1980.

[Robinson 82] J.J. Robinson, "Diagram: A Grammar for Dialogues," C.ACM, Vol.25, No.1, pp.27-47, 1982.