ICOT Technical Report: TR-146

TR-146

A Parallel Parsing System for Natural Language Analysis

by
Yuji Matsumoto

November, 1985

© 1985, ICOT

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 Parallel Parsing System for Natural Language Analysis

Yuji Matsumoto

Institute for New Generation Computer Technology
Tokyo, 108, Japan

(20 August 1985)
(revised 23 November 1985)

1. Introduction

Recent trend in computational linguistics puts a great attention on context-free grammars for natural language grammar formalism. Although it is an arguable topic whether natural languages could be described by context-free grammars, they are supported by recent prominent works in linguistics [Pullum 82],[Gazdar 82],[Kaplan 82].

DCG (Definite Clause Grammar) formalism, proposed by Pereira and Warren [Pereira 80], is based on Horn clause logic and can be seen as a formalism in a context-free grammar reinforced by unification between arguments in nonterminal symbols and attached Prolog procedures. It is powerful enough to cope with most of the grammar theories employing context-free grammar forms for describing their syntactic structure.

In this paper we assume DCG formalism as the user language for specifying grammars, and present an efficient and general parsing system which operates in parallel logic programming languages like Parlog [Clark 84] or GHC [Ueda 85]. Current system has been tested in Parlog interpreter [Gregory 84] and the same program incidentally runs in Prolog as well with a slight modification.

We stick to nondeterministic parsing method throughout in this paper, though there is another trend in parsing natural languages concerning deterministic methods, eg. [Marcus 80]. This is because we adopted the DCG formalism for describing grammars and we like to make the description of grammar rules as independent as possible of each other and of the procedural semantics of the parsing system. However, there could be some restrictions on the forms of grammar rules or on the way of expressing extra conditions because we employ the bottom-up strategy for our parsing algorithm and make use of parallel logic programming languages.

The algorithm of our system is based on the concurrent process model of parallel logic programming languages originated from Relational Language [Clark 81]. As is seen in a later section, what our parsing method does is virtually equivalent to what most of the efficient parsing algorithms so-called tableau methods do. The most important feature of our method is that the grammar rules and the dictionary are completely compiled into the logic programming language and the system does not need any program which interprets the grammar and the dictionary. Furthermore, the tableau itself is also compiled in the program. More precisely, an item that is normally kept in a tableau is represented as either an process or an item in a stream and no item in the streams is created duplicatedly.

The next section describes the basic algorithm which our system is based on. A naive way is shown for transforming grammar rules into a Prolog program which reflects our idea clearly.

The following two sections consider the way to make the system more efficient and extend it so that it becomes capable to work in parallel logic programming languages.

In Section 5 we compare our system with other parsing algorithm for context-free grammars in order to estimate the computational complexity. We mainly take up Chart parsing [Kay 80] and point out the similarity between them.

Lastly we discuss how to incorporate context dependent information in the system. Extra conditions included in DCG rules are not difficult to be incorporated in the system if Prolog is accessible to the parallel logic language or all the extra conditions are written in the parallel language itself. However, we need a special consideration of the arguments in nonterminal symbols in grammar rules. A brief discussion on this problem is given in this section.

2. Basic Algorithm

In this section and following several sections we deal with not DCG rules but bare context-free grammar rules. This simplification helps to convey the essential idea more easily. We also inhibit grammars from containing empty production rules and cyclic set of rules. The first restriction comes from the fact that our algorithm is basically a bottom-up parsing. The latter is in order to make it sure that no process runs into an infinite loop. Though a cyclic set of rules is a special case of a set of left recursive rules, our system allows to contain them, which possibly cause infinite loops in the case of some top-down parsing algorithms.

Our basic algorithm is the left-corner parsing [Aho 72], in which phrases are constructed from bottom to top and from left to right. Whenever a phrase of a certain nonterminal grammar category is obtained, it checks whether that phrase is usable to make up a larger and more complete structure extending the previously constructed partial structures or to make it as the left-most element of a new tree structure. These processes proceed until every possibility is tried. We have given this procedural semantics to DCGs by transforming each DCG rule into a certain form of Prolog clause. This system is called the BUP system [Matsumoto 83,84].

In this section we show another way of achieving the same effect on a grammar written in the DCG formalism. To keep the history of the parsing processes of a left-corner parsing, it is necessary to remember not only the sequence of rules used so far but also the internal positions in these rules until which the parsing process has finished. This is simply done if each position in the right-hand side of every grammar rule has an identifier to distinguish itself from others. Suppose each nonterminal or terminal symbol is defined as a predicate in Prolog and has two arguments, one for the data coming from its left in the sentence and the other for the data to pass to the right. The data that move around between the grammar symbols are sequences of identifiers that give the history of parsing process. This consideration leads to the following transformation of the context-free grammar rule (1) into Prolog clauses (2). In the right-hand side of the grammar rule, the symbol 'id' followed by a figure stands for identifiers.

```
(1)  a --> b,  id1  c, id2  d.

(2)  b(X,[id1|X]).
     c([id1|X],[id2|X]).
     d([id2|X],Y) :- a(X,Y).
```

Lists are used to represent the history of the parsing process and each nonterminal symbol of the grammar is defined as a Prolog predicate. If a nonterminal symbol receives 'id1' as the top element of the list, this means that nonterminal symbol 'b' has already been found and the corresponding part in rule (1) has been used in the parsing process. The sequence of identifiers can be expressed by defining identifiers as functors, that is, (2) is alternatively written like (2)'. We use this notation rather than lists because of its better readability.

```
(2)' b(X,id1(X)).
     c(id1(X),id2(X)).
     d(id2(X),Y) :- a(X,Y).
```

A simpler rule like (3) is transformed into the Prolog clause (4).

```
(3)  vp --> verb.

(4)  verb(X,Y) :- vp(X,Y).
```

This shows that nonterminal symbol 'verb' can be immediately reduced to the nonterminal 'vp'. Note that nonterminal and terminal symbols need not be treated in different ways.

The following grammar (5) written in DCG formalism is transformed into the Prolog program (6). Again, the figures preceded by the symbol 'id' are identifiers for particular positions in the grammar rules.

```
(5)  s --> np,  id1  vp.
     np --> det,  id2  noun.
     np --> det,  id3  noun,  id4  relc.
     relc --> [that],  id5  s.
     vp --> verb.
     vp --> verb,  id6  np.
     det --> [the].
     noun --> [man].
     noun --> [woman].
     verb --> [loves].
     verb --> [walks].

(6)  np(X,id1(X)).
     vp(id1(X),Y) :- s(X,Y).
     det(X,id2(X)).
     noun(id2(X),Y) :- np(X,Y).
     det(X,id3(X)).
     noun(id3(X),id4(X)).
     relc(id4(X),Y) :- np(X,Y).
     that(X,id5(X)).
     s(id5(X),Y) :- relc(X,Y).
     verb(X,Y) :- vp(X,Y).
     verb(X,id6(X)).
     np(id6(X),Y) :- vp(X,Y).
     the(X,Y) :- det(X,Y).
     man(X,Y) :- noun(X,Y).
     woman(X,Y) :- noun(X,Y).
     loves(X,Y) :- verb(X,Y).
     walks(X,Y) :- verb(X,Y).
```

Now the Prolog clauses in (6) give a left-corner backtracking parsing program for the context-free grammar (5). In order to parse a input sentence, for example, "the man walks", only we have to do is simply to call the following Prolog goals.

```
(7) the(begin,X),man(X,Y),walks(Y,end).
```

The terms 'begin' and 'end' in these goals are constants for identifying the beginning and terminal positions in the sentence. Note that one variable is shared by each consecutive pair of goals, which is used to pass the history of the parsing process from left to right. To complete the parsing program, another Prolog clause (8) is required, which checks if a sentence structure is constructed by using up the whole words in the sentence.

```
(8) s(X,Y) :- X==begin,Y==end.
```

Figure 1 shows the parsing process starting from the Prolog goals (7). In the figure, a single arrow shows that a goal is reduced to another goal(s) through the matching process with the head of a clause. A double arrow indicates the assignment of values to the arguments of the goal by the unification. This Prolog program goes through every possible analysis for the given sequence of words utilizing the backtracking mechanism of Prolog.

3. Parallel Parsing

The parsing program shown in the preceding section is now examined to be more efficient and suitable for parallel execution. A close look at the Prolog program reveals that there are two kinds of clauses in it. We refer to them as type-one clauses and type-two clauses. Structurally a clause in type-one has a variable for its first argument of the head and a clause in type-two has a structured data for its first argument of the head. During the execution of the program, type-one clauses start to construct a new tree structure regarding itself as the left-corner constituent of a tree structure. The first clause in (6) is an example of this type. It puts an identifier 'id1' on the top of the sequence it receives. A type-two clause matches the first element of the received sequence against the element in its first argument. If they are equal to each other it makes either a new Prolog call or a new data structure. The second clause in (6) makes a new call corresponding to a nonterminal 's' if the first element of the sequence is 'id1'. This identifier is removed from the sequence on calling the new Prolog call, since it is no longer necessary to keep this particular fragment of history. The sixth clause in (6) works in a different way. If the first element in the received sequence is 'id3' it puts another identifier 'id4' on the top of the sequence after removing the previous identifier. This corresponds to the process of extending the partially constructed tree structure. In either case each type-two clause works on a particular identifier.

This algorithm is actually not so efficient. Its time complexity is exponential with respect to the length of the sentence since many repetitive computations occur in the parsing process. They are caused by the definition of type-one clauses in that they put an identifier on the received data of the history without regard to its content. Therefore, the computation hereafter becomes independent of the history up to that moment. In order to avoid the repetition caused by type-one clauses it is natural to make up a set of all histories and to apply the type_one clauses to the set. So, a type-one clause puts its proper identifier on the top of the set that represents the whole history so far. A type-two clause receives a set of histories and examines if each of them is possibly extended by itself.

This modification fits in with parallel logic programming languages. Type-one clauses and type-two clauses belonging to the same nonterminal symbol can run independently, that is, in parallel. Both of them receives the same data. In a parallel logic programming languages like Parlog or Guarded Horn Clauses it is common to represent a set as a list and treat it as a data stream. If we assume a list structure for the set of histories, all the type-one clauses of the same nonterminal symbol can be bundled together to make up such a stream and we can define each type-two clause of the same nonterminal symbol as a distinct process that works on the stream and modifies it accordingly.

The following modification of the Prolog clauses into Parlog clauses materializes this idea. We can define equivalent clauses also in GHC. We employ Parlog syntax in this paper because of its readability. Parlog clause (10) is given from Prolog clauses of (9). Clauses of (12) are Parlog definition of type-two clauses shown in (11).

```
(9)  det(X,id2(X)).
     det(X,id3(X)).

(10) mode det1(?,^).
     det1(X,[id2(X),id3(X)]).

(11) noun(id2(X),Y) :- np(X,Y).
     noun(id3(X),id4(X)).

(12) mode noun2(?,^).
     noun2([],[]).
     noun2([id2(X)|T],Y) :- |
          np(X,Y2),
          noun2(T,Y1),
          merge(Y1,Y2,Y).
     noun2([id3(X)|T],[id4(X)|Y]) :- |
          noun2(T,Y);
     noun2([_|X],Y) :- |
          noun2(X,Y).
```

Clauses defined in Parlog assume that the received data, i.e., the data given to its first argument is a list (or a stream) of histories though the first argument is expressed by the same variable name as in Prolog clauses. As can be seen in the program the stream is a list of streams each of which is headed by an identifier. Generally speaking, type-one clauses are to collect all possible left-corner branches starting at their own position. There is a special case for type-one clauses where some of the original type-one clauses have a variable for their second argument. Such a clause is to be derived from a grammar rule the right-hand side of which consists of only one symbol. For example, type-one clauses in Prolog like (13) are transformed into a Parlog clause (14).

```
(13) verb(X,Y) :- vp(X,Y).
     verb(X,id6(X)).
```

```
(14) mode verb1(?,^).
     verb1(X,[id6(X)|Y]) :- |
          vp(X,Y).
```

When there are more than one such a clause, their outputs are merged together.

Type-two clauses are defined as a set of or-processes, each of which specializes in an individual identifier that can extend partially completed tree structures in the stream data. The first clause in (12) is for the case when the stream is empty. The second clause accepts the identifier 'id2'. In this case, this means the completion of a noun phrase and it produces a goal for 'np' along with another call for 'noun2' for the remaining data in the stream. The third clause deals with the case where the noun is used to modify a partially completed tree into a larger but still incomplete tree structure. The last definition in (12) is referred to only when all other processes have failed to utilize the first element in the stream. Note that this clause is separated from the preceding clauses by a colon not by a full stop. This is a Parlog convention which indicates that this clause should be executed only when all the preceding clauses failed. This clause throws away the first element in the stream and calls itself with the remaining data.

Each nonterminal symbol is now defined by one type-one process and one type-two process as follows.

```
(15) mode noun(?,^).
     noun(X,Y) :- |
          noun1(X,Y1),
          noun2(X,Y2),
          merge(Y1,Y2,Y).
```

Although the use of merge processes guarantees to earn higher parallelism, merging is not a cheap operation and it can be avoided by using a data structure called different lists. A different list consists of a pair of lists and represents a list as the difference of these two lists. The programs shown in this section also run in Prolog with a simple modification. The usage of different lists in Prolog makes the program more efficient since we can do away with merge operations. The following definitions from (16) to (19) written in Prolog are the equivalents of the definitions (10), (12), (14) and (15). This modification is also available in the Parlog program but restricts the order of execution. Hereafter, different lists are used in the Prolog definition. Note that the operational semantics of Prolog guarantees that the streams are completely constructed when they are passed to the next processes. In the following, the first argument in a definition of a nonterminal symbol works as an input stream and the second and third arguments represent a different list that works as the output stream.

```
(16) det1(X,[id2(X),id3(X)|Yt],Yt).

(17) noun2([],Y,Y) :- !.
     noun2([id2(X)|T],Y,Yt) :- !,
          np(X,Y,Y1),
          noun2(T,Y1,Yt).
     noun2([id3(X)|T],[id4(X)|Y],Yt) :- !,
          noun2(T,Y,Yt).
     noun2([_|T],Y,Yt) :- !,
          noun2(T,Y,Yt).

(18) verb1(X,[id6(X)|Y],Yt) :- !,
          vp(X,Y,Yt).

(19) noun(X,Y,Yt) :- !,
          noun1(X,Y,Y1),
          noun2(X,Z,Yt).
```

If we bring in a syntax sugar, we can get rid of all the nuisance difference lists. As a matter of fact, the DCG syntax can be used to express the program more simply. The clauses from (16)' to (19)' represent exactly the same clauses from (16) to (19).

```
(16)' det1(X) --> !,
          [id2(X),id3(X)].

(17)' noun2([]) --> !.
       noun2([id2(X)|T]) :- !,
          np(X),
          noun2(T).
     noun2([id3(X)|T]) --> !,
          [id4(X)],
          noun2(T).
     noun2([_|T]) --> !,
          noun2(T).

(18) verb1(X) --> !,
          [id6(X)],
          vp(X).

(19) noun(X) --> !,
          noun1(X),
          noun2(X).
```

The modifications of the program in this section necessitates some alterations to the definition of nonterminal symbol 's'. One way of doing it is change the initial goals to (20) and to add the clause shown in (21) or (22) to the definition of 's2' (type-two clauses for nonterminal symbol 's'). The former is to be added to the Parlog program and the latter to the program with difference lists.

```
(20) the([begin],X),man(X,Y),walks(Y,Z),fin(Z).

(21) s2(X,[end]) :-
          X==[begin] | true.

(22) s2([begin]) --> !,
          [end].
```

The definition for 's2' given here produces the term 'end' that indicates the completion of a sentence structure. The process 'fin' is a user defined predicate and is positioned at the end of the initial goals, which receives the stream produced by the last word in the sentence and checks if the stream contains the term 'end'. The usage and the definition of this process depend on the user's intention of using this parsing system.

4. Top-down Prediction

A straightforward improvement in the efficiency is achieved by employing a notion referred to as 'oracle' in LINGOL [Pratt 75] or as 'reachability' in Chart Parsing [Kay 80]. We call this notion as top-down prediction since this naming seems to be suitable for coupling with the bottom-up strategy.

The second grammar rule of the grammar shown in (5) means that a possible structure of a noun phrase is a determiner followed by a noun. Suppose a determiner is found in the parsing process. Making use of this rule corresponds to sending the identifier associated with this rule. There are two things to take note of. The first is that the usage of this rule gives rise to an expectation of a noun at the position just after the determiner. This gives a positive reason to encourage to build up a noun structure at that place. In this case a noun is referred to as a top-down prediction at that position in the sentence. Top-down predictions are obtained dynamically during the parsing process. The second thing to pay attention is that the usage of a particular grammar rule should be guaranteed by at least one top-down prediction. In our current case, the left-hand side of the grammar rule is a noun phrase. In order for this grammar rule to have a guarantee of being a part of a larger constituent, at least one of the top-down predictions at that position must be a noun phrase or must be something which can have a noun phrase as its left-most descendant. If there is no such top-down prediction, there is no place for the noun phrase which will eventually be constructed by this grammar rule. Top-down prediction is very useful in avoiding bottom-up searches which have no place to go. Whether a nonterminal symbol has a possibility of being a left_most descendant of another nonterminal symbol is precomputable from the grammar rules. The left-most element on the right-hand side of a grammar rule is possibly a left-most descendant of the nonterminal symbol on the left-hand side of the grammar. The relation we are talking about is defined as the transitive and reflexive closure of the set of such pairs. We refer to this relation as the link relation and say that a nonterminal symbol links to another if they satisfy this relation.

There are several ways to incorporate this idea in our parsing program. We introduce one idea which we are employing currently. A stream given to nonterminal symbols is a stream of streams each of which is headed by an identifier. An identifier is a unique symbol for a particular position in a grammar rule, and is waiting for a unique nonterminal symbol that follows the identifier in the grammar rule. This nonterminal symbol is exactly what is expected as the top-down prediction at the corresponding position in the sentence. The mapping from identifiers to nonterminal symbols is of course obtainable when context-free grammar rules are translated into the parsing program. The special identifiers 'begin' and 'end' predict 's' and nothing, respectively. The top-down prediction is easily made use of by modifying the type-one clauses. In the original program shown in Section 2, a type-one clause corresponds to the left-most element on the right-hand side of a grammar rule and produces an identifier. This identifier is blocked if the nonterminal symbol on the left-hand side of the original grammar rule is not predicted by the identifier it received. In the parallel definition of a type-one clause, an identifier is blocked only in the case that the corresponding nonterminal symbol is not predicted by any head identifiers in the input stream. In other words, the identifier is passed to the output stream if at least one head identifier in the input stream is known to predict the nonterminal symbol on the left-hand side of the original grammar rule concerned.

In our current implementation, the process of the top-down prediction is realized by a process of filtering. This process take the input stream to the type-one clause and throws all the unusable elements away. The definition of type-one clause (10) now becomes (23).

```
(23) det1(X,Y) :-
         tp_check(X,np,New_X),
         tp_output(New_X,[id2(New_X),id3(New_X)],Y).
```

In this definition, 'tp_check' is the filtering process which allows to pass only the elements in 'X' whose head identifier predicts a noun phrase. 'New_X' is the output of this filter. The process 'tp_output' returns the data at the second argument to the third argument if 'New_X' produces at least one element. Otherwise, it returns empty to the third argument. Note that these two processes run in parallel, that is, two identifiers 'id2' and 'id3' are passed to the next process even though the stream in their argument is still incomplete. Also note that the filtering process is shared by two output identifiers since both of the corresponding grammar rules have 'np' on their left-hand side. When there are more than one filtering process, hence, more than one 'tp-output', their outputs are merged.

5. Comparison with Chart Parsing

In order to estimate the time and space complexity of our parsing method, we compare it with Chart Parsing [Kay 80].

Chart Parsing consists of processes constructing a data structure called a chart, which is a so-called well-formed substring table and is conceptually depicted as a directed graph. Each element in a chart is a term representing a partially or perfectly constructed tree structure. A partially constructed tree is expressed by a term with some empty slots and a perfectly constructed tree is expressed by a term that represents a tree structure without any empty element. They are referred to as an active edge and an inactive edge when they are represented by a directed graph. Chart Parsing is actually not a parsing algorithm but an algorithm schema as its author says. The process of constructing terms in a chart (or edges in a directed graph) varies according to the control given to the schema. We will not explain the schema in detail. We will point out the similarity between our parsing method and Chart Parsing with a bottom-up control. When a bottom-up strategy is given to the Chart Parsing, it operates with two rules. One rule is to start building up new tree structures from a perfectly constructed tree (ie an inactive edge). When an inactive edge is obtained, this means that a complete nonterminal symbol is made up. Referring to the grammar rules, this rule produces all the partially complete terms (ie active edges) that have the current nonterminal symbol as its left-most element in the right-hand side. The second rule is for filling an empty place in a partially complete term with a complete term.

There is a very close one-to-one correspondence between a bottom-up Chart Parsing and our parsing algorithm. In our parsing algorithm the first and the second rules of Chart Parsing correspond to processes done by type-one clauses and type-two clauses, respectively. Inactive edges are represented by calls of nonterminal symbols in the program and a set of active edges are represented by the data structure being passed to nonterminal symbols.

The important differences between them are that our algorithm is compiled into a Parlog or Prolog program and that partial results need not be kept in something like a well-formed substring table. In Chart Parsing, adjacency of two edges are checked by the location and the length associated with terms. A chart keeps them together with corresponding terms. Our algorithm, however, does not require them since the original words in a given sentence are connected by shared variables and the data representing active edges are passed through these variables, each of which indicates a specific position in the original sentence. The definition of clauses ensures that a tree structure is never created repeatedly unless some grammar rules are defined duplicatedly. Figure 2 shows the chart created by bottom-up Chart Parsing method using the grammar rule (23) [Kay 80]. In this table, a term represents a tree , locus shows the position of the term in the sentence and length is the number of words in the term. A question mark in a term are the undefined part in the term.

```
(23)  s --> np, vp.
      np --> a, n.
      np --> prp, n.
      vp --> v, a.
      vp --> v, av.
      a --> [failing].
      a --> [hard].
      n --> [students].
      v --> [looked].
      av --> [hard].
```

The main part of the Parlog program obtained from these grammar rules are shown in (24). (24) consists only of the essential clauses. Some more clauses should be added to complete the program. Top-down predictions are not used in this program.

```
(24)  np1(X,[id1(X)]).
      a1(X,[id2(X)]).
      prp1(X,[id3(X)]).
      v1(X,[id4(X),id5(X)]).

      vp2([id1(X)|T],Y) :-
          s(X,Y1),
          vp2(T,Y2), merge(Y1,Y2,Y).
      n2([id2(X)|T],Y) :-
          np(X,Y1),
          n2(T,Y2), merge(Y1,Y2,Y).
      n2([id3(X)|T],Y) :-
          np(X,Y1),
          n2(T,Y2), merge(Y1,Y2,Y).
      a2([id4(X)|X1],Y) :-
          vp(X,Y1),
          a2(T,Y2), merge(Y1,Y2,Y).
      av2([id5(X)|T],Y) :-
          vp(X,Y1),
          av2(T,Y2), merge(Y1,Y2,Y).

      failing(X,Y) :-
          a(X,Y1), prp(X,Y2), merge(Y1,Y2,Y).
      hard(X,Y) :-
```

```
        a(X,Y1), av(X,Y2), merge(Y1,Y2,Y).
    students(X,Y) :-
        n(X,Y).
    looked(X,Y) :-
        v(X,Y).
```

(25) is the initial calls of Parlog program for parsing the same sentence. Figure 3 shows the execution of this program.

```
(25) failing([begin],D1),students(D1,D2),looked(D2,D3),hard(D3,D4),fin(D4).
```

In this figure, arrows mean calls of new processes or creation of data. Double arrows indicate the flow of data structure. The numbers are associated with arrows to show the one-to-one correspondence between processes in Chart Parsing and in our algorithm. Arrows that do not have a number correspond to the creation of type-one and type-two literals from nonterminal symbols. Note that the analysis starting from each word in the sentence does not depend on the analysis of other places. It is affected by other process only when it refers to the data passed from other processes. For example, the analysis starting from 'looked' can proceed even if 'D2' is not instantiated. The analysis starting from 'hard' is suspended on calling 'av2' or 'a2' if 'D3' is still an uninstantiated variable. In Figure 3, the vertical axis more or less shows the time. Processes on the same level could be done in parallel. From this figure we can see that the time complexity of our algorithm is to be proportional to the height of the analysis tree, which is in worst case equivalent to the length of the given sentence.

The space complexity of our algorithm is not worse than that of Chart Parsing since no data structure is duplicated. Furthermore, if variables in our program are shared as in Prolog, it requires less space than the chart. Subterms must be copied to register a newly constructed term in the chart whereas they need not be copied in our algorithm because of the shared variables. In the case of parallel execution, the time complexity depends on the treatment of logical variables in the parallel logic languages.

6. Conclusions

This paper briefly described the idea of our parallel parsing system based on parallel logic programming languages. Our specification of parsing program runs not only on Parlog but on many of other parallel logic programming languages like Concurrent Prolog and Guarded Horn Clauses that derive from the Relational Language [Clark 81]. As is described in Section 3, Prolog implementation of our parsing system is also practically useful. As for the parallel implementation we have to wait for an efficient realization of a parallel logic programming language.

There are some problems with our parsing system that must be mentioned here. We did not explain the full treatment of DCG rules. When DCG rules have arguments in nonterminal symbols and Prolog programs in their right-hand side as extra conditions, they must be coupled with the Parlog or Prolog program.

Extra conditions are not difficult to handle in our program though some restrictions are inevitably placed on the property of extra conditions. After the transformation there is in general one clause for each nonterminal symbol in the right-hand side of any DCG rule. Extra conditions in a right-hand side of a DCG rule is put in the guard part of the clause produced for the nonterminal symbol preceding these extra conditions. According to our bottom-up procedural semantics, extra conditions preceding the first nonterminal symbol in the right-hand side of a rule does not have any significant meaning. Users are advised not to write extra conditions there or they are put at the guard part of the clause for the first nonterminal symbol of the rule even when a user writes some extra conditions at the beginning of the rule. In the programs shown in preceding sections, type-one clauses with the same nonterminal symbol are bundled up into one clause. However, type-one clauses having different extra conditions cannot be bundled up simply. Such clauses are defined separately and their outputs are merged later. Another restriction is that the operational semantics of the derived program prohibits extra conditions from having nondeterminacy. It is because extra conditions are treated as guards.

Another problem arises when nonterminal symbols have arguments and they contain uninstantiated variables. Variables in an argument may be passed to more than one place in the stream data. The data structures moving through a shared variable are something like tree structures. We have to note that substructures in different branches in the data structure are in different environments. Although they can share the value of a variable, an instantiation of one variable must not affect the value of the same variable in different environments. The easiest way to avoid this difficulty is to copy the value of a variable any time the variable is passed to more than one environment. However, such a simple treatment may cause a space explosion. More moderate and safe way is to copy the value of a variable that is included in the output of a type-two clause any time the guard part of the clause finishes successfully. The treatment of this problem is crucial for our system to be practical. A development of a parsing system in Prolog based on this idea is now under way.

Acknowledgments

This work was mainly done while the author was staying with the Logic Programming Group at Imperial College of Science and Technology. The author wishes to express his thanks to Professor Robert Kowalski for giving him the opportunity to work in his group. Discussions with the members of his group and of the Natural Language Group led by Dr Jim Cunnigham were very valuable. Thanks are also due to the British Council that supported his staying in the College.

$$\begin{tikzcd}
\texttt{the(begin,S0),} \arrow[d, dashed] & \texttt{man(S1,S1),} \arrow[d, Rightarrow, dashed] & \texttt{walks(S1,end).} \arrow[d, Rightarrow, dashed] \\
\texttt{det(begin,S0)} \arrow[d, dashed, "\begin{array}{l}\texttt{X=begin}\\\texttt{S0=id2(begin)}\end{array}"] & \texttt{man(id2(begin),S1)} \arrow[d, dashed] & \texttt{walks(id1(begin),end)} \arrow[d, dashed] \\
\emptyset & \texttt{noun(id2(begin),S1)} \arrow[d, dashed] & \texttt{verb(id1(begin),end)} \arrow[d, dashed] \\
 & \texttt{np(begin,S1)} \arrow[d, dashed, "\begin{array}{l}\texttt{X=begin}\\\texttt{S1=id1(begin)}\end{array}"] & \texttt{vp(id1(begin),end)} \arrow[d, dashed] \\
 & \emptyset & \texttt{s(begin,end)} \arrow[d, dashed] \\
 & & \texttt{begin==begin,end==end}
\end{tikzcd}$$

Figure 1. Sample Parsing

#	Locus	Length	Term
1	0	1	[failing]a
2	0	1	[failing]prp
3	1	1	[students]n
4	2	1	[looked]v
5	3	1	[hard]a
6	3	1	[hard]av
7	0	1	[[failing]a [?]n]np
8	0	1	[[failing]prp [?]n]np
9	0	2	[[failing]a [students]n]np
10	0	2	[[failing]prp [students]n]np
11	0	2	[[[failing]a [students]n]np [?]vp]s
12	0	2	[[[failing]prp [students]n]np [?]vp]s
13	2	1	[[looked]v [?]a]vp
14	2	1	[[looked]v [?]av]vp
15	2	2	[[looked]v [hard]a]vp
16	2	2	[[looked]v [hard]av]vp
17	0	4	[[[failing]a [students]n]np [[looked]v [hard]a]vp]s
18	0	4	[[[failing]prp [students]n]np [[looked]v [hard]a]vp]s
19	0	4	[[[failing]a [students]n]np [[looked]v [hard]av]]s
20	0	4	[[[failing]prp [students]n]np [[looked]v [hard]av]vp]s

Figure 2 The Chart

```
    |---> failing         students            looked         hard
    |     |     |            |                   |           |   |
[begin]   |1    |2           |3                  |4          |5  |6
          V     V            V                   V           V   V
          a    prp           n                   v          av   a---|
          |     |            |                   |           |   |   |
          |     |            |                   |           |   |   |
          V     V            V                   V           V   V   |
         a1   prp1     D1==> n2--|              v1   D3==> av2  a2  a1
          |    |             |   |               |           |   |   |
          |    |8 [          |9  |10             |  [        |   |   |   [
         7|    |->id3(begin) V   |             13|->id4(D2)  |15 |   |->id2(D3)
          |                 np   V             14   id5(D2)  |   |16         ]
          |------>id2(begin) |   np                       ]  V   |
                          ]  V   |    [              D3     vp   V
                    D1      np1--+---->id1(begin)            |  vp------|
                                 |  11                       |          |
                                 V                           |          |
                                np1-->id1(begin)             V          |
                                    12         ]     D2==> vp2          V
                                         D2               |  |   D2==>vp2
                                                         17|  |        | |
                                                           |  |18   19| |
                                                           V  |        V |20
                                                           s  |        s |
                                                              V          V
                                                              s          s
```

Figure 3. Parallel Parsing Example

References

[Aho 72] Aho, A. V. and Ullman, J. D.,
'The Theory of Parsing, Translation, and Compiling,
Volume 1: Parsing,' Prentice-Hall, 1972.

[Clark 81] Clark, K. L. and Gregory, S.,
"A Relational Language for Parallel Programming,"
Imperial College Research Report DOC 81/16, July 1981.

[Clark 84] Clark, K. and Gregory, S.,
"PARLOG: Parallel Programming in Logic,"
Research Report DOC 84/4, Imperial College, April 1984.

[Gazdar 82] Gazdar, G.,
"Phrase Structure Grammar,"
in 'The Nature of Syntactic Representation'
P. Jacobson and G. K. Pullum (eds.), SLL 15, D.Reidel,
pp.131-186, 1982.

[Gregory 84] Gregory, S.,
"How to Use Parlog (C-Prolog Version),"
Department of Computing, Imperial College, Oct. 1984.

[Kaplan 82] Kaplan, R. M. and Bresnan, J.,
"Lexical-Functional Grammar: A Formal System for Grammatical
Representation,"
Chap.4 of 'The Mental Representation of Grammatical Relations'
J. Bresnan (ed.), MIT Press, pp.173-281, 1982.

[Kay 80] Kay, M.,
"Algorithm Schemata and Data Structures in Syntactic Processing,"
XEROX Palo Alto Research Center, CSL-80-12, Oct. 1980.

[Marcus 80] Kay, M.,
'A Theory of Syntactic Recognition for Natural Language,'
The MIT Press, 1980.

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

[Matsumoto 84] Matsumoto, Y., M. Kiyono and H. Tanaka,
"Facilities of the BUP Parsing System,"
Proceedings of Natural Language Understanding and Logic
Programming, Rennes, 1984.

[Matsumoto 85] Matsumoto, Y.,
"On Parallel Parsing," Electrotechnical Laboratory Research
Memorandom, ETL-RM-85-34E, 1985.

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

[Pratt 75] Pratt, V.R.,
"LINGOL -- A Progress Report,"
Proc. the 4th IJCAI, pp.422-428, 1975.

[Pullum 82] Pullum, G.K. and G. Gazdar,
"Natural Languages and Context-free Languages,"
Linguistic and Philosophy 4, pp.471-504, 1982.