ICOT Technical Report: TR- 008

Chart Parsing in Concurrent Prolog

by

Hideki Hirakawa

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

Technical Report of ICOT Research Center : TR-008 May, 1983

CHART PARSING IN CONCURRENT PROLOG

by

Hideki Hirakawa

ICOT Research Center
Institute for New Generation Computer Technology

Mita Kokusai Bldg. 21F, 1-4-28 Mita,
Minato-ku, Tokyo, 108
Tel:03-456-3195, Telex:ICOT J32964

CHART PARSING IN CONCURRENT PROLOG

H.HIRAKAWA (ICOT 2nd lab.)

1. Introduction

There have been developed various parsing systems based on logic programming language Prolog(for example, DCG [Pereira 80] and BUP [Matsumoto 83]). In these systems, grammar rules are translated into Prolog programs which parses input sentences. Since the control mechanism of Prolog is based on backtracking, all the parsing systems developed on Prolog are backtrack based. This type of parsing methods have an inefficiency that many identical computations along the different parsing paths are duplicated. On the other hand, parallel parsing strategies can avoid this problem of duplication by backtracking by keeping all information about parsing processes in some memory areas. These methods are called bookkeeping parsing methods. In this memo, one of the bookkeeping parsers (CHART PARSER) is introduced and the experiment on PCP (Parser in Concurrent Prolog) is described. PCP implements the algorithm equivalent to chart parsing, using multiple process forks and message passing mechanism of Concurrent Prolog.

2. Active Chart Parser

In this section, 'Chart Parser' [Kay 67], which is a high efficient parser for Context Free Grammar, is introduced. See Chap. 3 of [Winograd 83] for the details.

2.1 Chart Parsing Features

(1) Parsing strategy
The Chart Parser applies rules in Top Down and Pseudo Parallel ways.

(2) Uses Chart as Bookkeeping
A Chart Parsing is one of the bookkeeping parsing methods. A chart is a data structure which keeps all the information about what has been tried and what is yet to be tried.

(3) Grammar With Left Recursive Rules
Since the chart keeps track of parsing process, infinite rule application can be avoided by checking the chart.

(4) No Computational Duplication
The backtracking based parsing strategy duplicates many computations that are the same along separate paths, while the parallel strategy makes it possible to combine common subcalculations by the use of bookkeeping (Chart).

2.2 Structure of Chart

The chart is a bookkeeping data structure which contains vertices and edges. A vertex is a number which specifies the position on an input sentence. For example, numbers 1 through

o in the following figure are vertices.

```
1  john  2  hits  3  ball  4  with  5  bat  6
```

An edge represents partial a constituent and consists of a starting vertex, an ending vertex, a label and a remainder. The surface word sequence which corresponds to the edge is represented by starting and ending vertices. A label is a left hand side symbol of a CFG rule (A label is sometimes a lexical category). The remainder is a part of a sequence of symbols in a righthand side of a CFG rule (possibly a null sequence). The edge can be represented in the following form.

```
[STARTING_VERTEX LABEL --> cat .. cat
                    ENDING_VERTEX REMAINDER]
```

Some examples are shown:

```
      1  john  2  hits  3  ball  4  with  5  bat  6
      |       | |                 |
      ----NP--- ------- VP--------
```

(a) [1 NP --> 1 NOUN]
(b) [1 NP --> NOUN 2]
(c) [2 VP --> VERB NP 4 PP]

Edge (a) represents a null string (the starting and the ending vertices are the same), and its remainder is NOUN. Edge (b) represents the string 'john' and its remainder is null. Edge (c) represents string 'hits ball' and its remainder is PP. An edge which has a null remainder such as (b) is called a terminated edge. The formal description of the chart is shown below. CHART has EDGES and PENDING EDGES. The former is a storage for active edges and the latter for pending edges.

```
CHART  ::=  VERTICES          (a sequence of vertices)
            EDGES             (a set of edges)
            PENDING EDGES     (a set of pending edges)


EDGE   ::=  STARTING VERTEX   (a vertices in the chart)
            ENDING VERTEX     (a vertex in the chart)
            LABEL             (a symbol)
            REMAINDER         (a sequence of symbols,
                               possibly empty)

VERTEX ::=  INCOMING EDGES    (The set of edges
                               in the chart having
                               this vertex as their
                               ending vertex)
            OUTGOING EDGES    (The set of edges in the
                               chart having this vertex
                               as their starting vertex)
```

2.3 Parsing Algoritnm

The chart parser applies grammar rules in a topdown parallel way. If given rules are

a. S --> NP VP
b. NP --> NOUN
c. NP --> DET NOUN

then the first rule, (a) is tried, and then (b) and (c) are applied simultaneously. The applied rule (a) is stored in PENDING EDGE and the newly applied rules are stored in EDGE in the chart. In this derivation, left recursion check is done and the rule already applied in the same context will never be derived. When a derivation meets a lexical category, that of the input word is checked whether its lexical category is identical to that of the rule. If identical, the derivation is tried on category following that lexical category in the rule. If there's no succeeding category, the edge is terminated. If a terminated edge is detected, the parser checks all pending edges and sets appropriate edges into active edge in the chart. In this way the parser continues parsing until there's no edge in EDGE.

3. Parser in Concurrent Prolog

PCP is a parser based on multiple process forks and the message passing mechanism supported by Concurrent Prolog [Shapiro 83], and provides the parsing algorithm equivalent to the Chart Parsing. Consequently, PCP provides the same characteristics as the Chart Parser does,

(1) Top Down and Pseudo Parallel (Concurrent)
(2) Using multiple processes and shared variables as Bookkeeping
(3) Grammar with Left Recursive Rules
(4) No Computational Duplication

3.1 Parsing Strategy

In PCP, a basic parsing component is a process corresponding to an edge, and there is no explicit bookkeeping data structure as the chart. Each process can communicate with each other in term of a communication channel. These process are consists of four elements as shown next;

```
process(State, Edge, Channel1, Channel2 <,C3>)
```

State is 'active' or 'wait'. Edge keeps the same information described in the Chapter2 (i.e. STARTING VERTEX, ENDING VERTEX, LABEL, REMAINDER). Channel1 is communication a channel for the terminated edges. Channel2 is a channel for the tried edges, which is necessary for rule derivation loop checking. C3 is used for keeping the head of the Channel1 (D-list).

The following CP program shows the behavior of the above process.
(The whole program is listed in Appendix 2)

```
(c1) process(active,Edge,C,C2) :-
       dict(Edge,NewEdge) |
          process(active,NewEdge,C,C2).

(c2) process(active,Edge,C,C2) :-
       non_terminalp(Edge) |
          process(wait,Edge,C?,C2,C) //
process_fork(Edge,C,C2).

(c3) process(active,Edge,C,C2) :-
       terminate(Edge) |
          broadcast(Edge,C).

(c4) process(active,Edge,C,C2).

(c5) process(wait,Edge,[Terminated_Edge|C],C2,C3) :-
       reactivate(Edge,Terminated_Edge,NewE1) |
          process(wait,Edge,C?,C2,C3) //
          process(active,NewE1,C3,C2).

(c6) process(wait,Edge,[_|C],C2,C3) :-
       true |
          process(wait,Edge,C?,C2,C3).
```

(c1)-(c4) specify the behavior of an active process and (c5) and (c6) specify that of a wait process. In the above program, "|","//" and "?" are specific symbols peculiar to Concurrent Prolog. "|" is a guard bar which separates a guarded sequence from goal sequence. Once a guard sequence is processed successfully, then the choice points for another clauses are removed. In this sense "|" works like 'cut' in Prolog. "//" is a parallel-AND symbol which logically denotes AND. The goals connected by parallel AND must be executed in parallel. "?" attached to a variable specifies that the variable must not be unified with a non-variable term (Read only annotation). The read only annotation can generally be used to the variables shared by concurrent processes in order to restrict the direction of data flow. The process which annotates the shared variable can not instantiate the variable and wait for the variable to become instantiated by the other process which does not annotate it.

Active processes are categorized into four cases. In the first case of (c1), the head element of the REMAINDER in the Edge is lexical category. For example, the edge has the following form:

```
        [1 NP --> 1 DET NOUN]
  where
        vertex 1 is      1 the 2 man ....
        'the' has a lexical category DET
```

In this case, 'dict' produces 'NewEdge' with updated 'RMAINDER' and 'ENDING EDGE', i.e. [1 NP --> DET 2 NOUN]. And a new active process is created.

In the second case of (c2), the head element of the 'REMAINDER' is a grammetical category. The edge has following form, for example:

```
    C1 3 --> 1 NP VP3
  where a grammar rule "NP --> DET NOUN"
  and "NP --> NOUN" exist
```

In this case, a guard clause 'non_terminalp' succeeds. Then the current process become a wait process which pends until some message is sent through channel1. This pending mechanism is specified by the read only annotation. 'process_fork' picks up all the grammar rules whose lefthand side is the head element of the REMAINDER, and creates the edges corresponding to the rules. Then each edge is checked by scanning C2 whether it already exist or not. If it is a new edge, then it is registered into C2 and a new active process with that edge is produced. If it has been already registered in C2, there's no need to continue processing with this edge because the old one will produce or has been produced the same effect as this one. This mechanism inhibits an infinite rule application loop and provides no duplication of the same subcalculations. A more precise explanation is given in Chap 3.2 in connection with the augmentation of CPC.

In the third case of (c3), the edge is a terminated edge. Guard clause 'terminate' checks whetner the REMAINDER of the edge is null or not. If it is null, then it's a terminated edge. 'broadcast' sends this edge to all the other processes, which is accomplished by attaching the edge to the tail of communication channel1 (D-list C).

The fourth case (c4) is other than (c1) to (c3). This case is an inappropriate one. For example,

```
      C1 NP --> 1 NOUN3
  where
      1 the 2 boy .....
```

In this case the process terminates (or vanishes).

(c5) and (c6) define the behavior of a wait process. Since the channel1 of wait processes are all read only variables, then (c5) and (c6) wait until a message (edge) is sent through the channel1 by other processes. In the case of (c5), a guard 'reactivate' checks that the message 'Terminated_Edge', can reactivate this process or not. The reactivate conditions are

(1) The LABEL of the terminated_edge is the same as the head element of the REMAINDER of the waiting process.

(2) The STARTING VERTEX of the terminated_edge is the same as the ENDING VERTEX of the waiting process.

If 'reactivate' succeeds, the current wait process produces a new active process. This new process has the edge with STARTING VERTEX and LABEL that are the same as those of the current wait process, ENDING VERTEX the same as that of the terminated edge, the REMAINDER the same as that of the wait process except for the top element.

Example

edge of the wait process	:	[1 S --> 1 NP VP]
the terminated edge	:	[1 NP --> DET NOUN 2]
edge of new process	:	[1 S --> NP 2 VP]

Note that, after the creation of new process, the old waiting process still exists.

In the case of (c6), the wait process neglects the message and continues waiting.

To show the parsing process, a simple example is given. The example sentence is "John hits ball with bat." The input sentence, grammar rules and dictionary are:

(i) 1 john 2 hits 3 ball 4 with 5 bat6

(g1)	s --> np, vp.	(d1)	dict(n, john).
(g2)	np --> noun.	(d2)	dict(n, ball).
(g3)	vp --> v.	(d3)	dict(n, bat).
(g4)	vp --> v, np, pp.	(d4)	dict(v, hits).
(g5)	pp --> p, np.	(d5)	dict(p, with).

Parsing begins with the following active process.

(p1) process(active, [1 s --> 1 np vp], C1, C2)

The head of the remainder 'np vp' is nonterminal, then program (c2) is applied and produces the following concurrent processes.

(p2) process(wait, [1 s --> 1 np vp], C1?, C2)
(p3) process(active, [1 np --> 1 noun], C1, C2)

(p2) has a read only variable C1, then this process waits. (p3) has a remainder 'noun' and ending vertex 1 which points 'john', then 'dict' clause of (c1) succeeds and the next process is produced.

(p4) process(active, [1 np --> noun 2], C1, C2)

(p4) has a null remainder, so (c3) is applied and 'broadcast' sets the terminated edge to C1. (c3) has no 'process' clause in the body, then (p4) and consequently (p3) terminates successfully. In the other words, (p3) and (p4) vanish.

Now the message '[1 np --> noun 2]' is sent to process (p2) through C1. Waiting of (p2)is released, and both (c5) and (c6) are applied. In this case, (p2) waits for a message in the form of '[1 np ..]', (c5) is selected. (c5) newly produces next process as follows:

(p5) process(wait, [1 s --> 1 np vp], C1?, C2)
(p6) process(active, [1 s --> np 2 vp], C1, C2)

(p5) is a dead copy of old process (p2). New process (p6) creates the following processes according to (c2).

(p7) process(wait, [1 s --> np 2 vp], C1?, C2)
(p8) process(active, [2 vp --> 2 v], C1, C2)
(p9) process(active, [2 vp --> 2 v np pp], C1, C2)

Then (p8) creates the following processes.

(p10) process(active, [2 vp --> v 3], C1, C2)

And (p10) has a terminated edge, then broadcasts the message '[2 vp --> v 3]'. This message reactivates (p7). (p7) creates a copy of itself and the following process.

(p11) process(active, [1 s --> np vp 3], C1, C2)

This edge is a terminated one, so this broadcasts '[1 s --> np vp 3]' and vanishes. This edge has root 's' but is not a correct parsing one, because it only corresponds to 'john hits ball'. In this way, active process (p9) creates the following processes.

(p12) process(active, [2 vp --> v 3 np pp], C1, C2)
(p13) process(active, [3 np --> 3 noun], C1, C2)
(p14) process(active, [3 vp --> v np 4 pp], C1, C2)
(p15) process(active, [4 pp --> 4 p np], C1, C2)
(p16) process(active, [4 pp --> p 5 np], C1, C2)
(p17) process(active, [5 np --> 5 noun], C1, C2)

(p17) sends message '[5 np --> noun 6]' and this terminates (p16). Then (p16) sends message '[4 pp --> p np 6]', terminating (p14). (p14) sends '[3 vp --> v np pp 6]' and this terminates (p7). (p7) sends the following edge.

[1 s --> np vp 6]

This message means that the sentence 'john hits ball with bat' is correctly analyzed as 's'.

In PCP the parsing continues as long as some active processes exist. If no active process is detected, computation terminates with a deadlock. At this point all possible analysis has been tried. In this sense, the whole algorithm is complete. The proof of completeness is omitted.

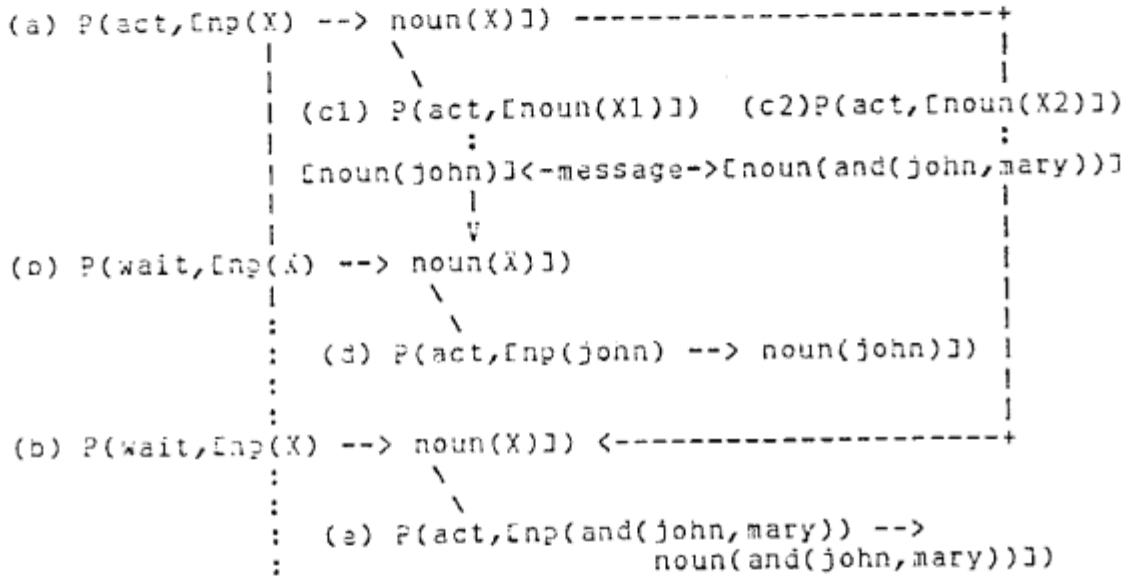
3.2 Augmentation

The program described in 3.2 is a very fundamental one that can treat only simple CFG rules. In a practical parsing system, some additional mechanisms are required to provide a structure building facility, semantic checking facility and so on. It is easy to augment CPC to handle CFG with arguments. Augmentation is to change the LABEL and the elements of REMAINDER from an

atom to a structure. The edge is represented as follows:

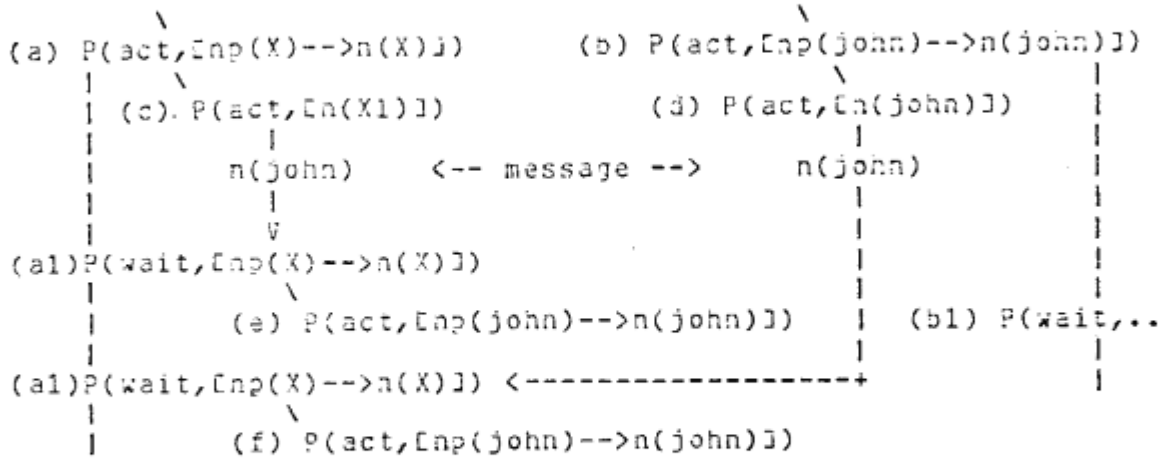
```
[1 s(s(NP,VP)) --> 1 np(NP) vp(VP)]
```

's(NP,VP)' and 'NP','VP' are arguments for grammar category 's','np' and 'vp'. The capital letters specify those words to be variables. In handling variables in the grammar rule, the OR-parallel problem must be solved. In the parser, the problem is avoided by copying edges when one process forks to multiple processes. And the re-binding of variables is done in the 'reactivate' processing. A next figure shows the variable manipulation processes.



To simplify the explanation, VERTICES and communication channels are abridged in this figure. 'P' specifies a process. Process (a) creates (b), (c1) and (c2). Process (b) is a dead copy of (a), except that it is a waiting process. Process (c1),(c2) are OR parallel daughter processes of (a) with new variables 'X1' and 'X2' instead of 'X'. Process (c1) finally broadcasts message '[noun(john)]' where 'X1' is bound to 'john'. This message reactivates (b). (b) creates (d) by means of creating the copy of the edge (i.e. [np(Y) --> noun(Y)]) and unifying the message 'noun(john)' and 'noun(Y)'. Process (b) is still waiting for a new message after the production of (d), and creates (e) by receiving message '[noun(and(john,mary))]'.

As shown above, the calculation of the daughter is executed independently of its parent, and the variable binding between them is done through the unification of the message. Since a wait process receives all the messages that are sent from all other processes, the process must recognize the correct message. In the case of a grammar with variables, the check of the unification between the message and the head element of REMAINDER is insufficient. Consider the following example:



In this example, there are similar active processes, (a) and (b). These must not be shared because there is no guarantee that both processes will produce the same results. (a) and (b) create (c) and (d). Consequently, the same two messages of 'n(john)' are broadcasted and wait process (a1) creates (e) and (f). To avoid this duplication, a message identifier is introduced. The message identifier is an integer to distinguish each message. Each process has two message identifiers; a parent identifier and a daughter one. A parent identifier is an identifier which the process attaches to message when the edge terminates. A daughter identifier is used to check the messages by the wait process. In the above example, two messages of 'n(john)' have different identifiers so the duplication is avoided.

4. Conclusion

The parser in Concurrent Prolog is described. It provides a top-down pseudo parallel and bookkeeping parsing mechanism. This means that the parsing efficiency is better than that of backtracking based parsing method. As for its generative power, PCP can be augmented to handle the grammar written in DCG with left recursive rules and empty derivation rules.

ACKNOWLEDGEMENT

I would like to finish by thanking the member of ICOT, in particular A.Takeuchi for his help in advising me about Concurrent Prolog and PCP, K.Sakai and R.Onai for the discussion about PCP. Also thanks to Y.Okada for her looking over this memo.

REFERENCE

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

[Matsumoto 83] Matsumoto,Y., Tanaka,H., Hirakawa,H., Miyoshi,H.
Yasukawa,H., Mukai,K. and Yokoi,T.:
"BUP: A Bottom Up Parser embedded in Prolog",

[Kay 67] Kay,M:
"Experiments with a powerful parser".
Proc. 2nd Int. COLING,(Aug. 1967)

[Winograd 83] Winograd,T:
"Language as a Cognitive Process",
Addison-Wesley Pub.,(1983)

[Shapiro 83] Shapiro,E,Y:
"A Subset of Concurrent Prolog and Its Interpreter",
ICOT Technical Report TR-003,(1983)

[Shapiro 83] Shapiro,E,Y and Takeuchi,A:
"Object Oriented Programming in Concurrent Prolog",
New Generation Computing, Vol. 1,No. 1,(1983)

APPENDIX A

A simple example of parsing is given.

The definition of grammar

```
gr(s(S(NP,VP)),          [np(NP),vp(VP)]).
gr(vp(vp(V)),            [v(V)]).
gr(vp(vp(V,NP)),         [v(V),np(NP)]).
gr(vp(vp(V,NP,PPS)),     [v(V),np(NP),pps(PPS)]).
gr(np(np(N)),            [n(N)]).
gr(np(np(DET,N)),        [det(DET),n(N)]).
gr(np(np(N,PP)),         [np(N),pp(PP)]).
gr(pp(pp(P,N)),          [p(P),np(N)]).
gr(pps(pps(PP,PPS)),     [pp(PP),pps(PPS)]).
gr(pps(pps(PP)),         [pp(PP)]).
gr(start(S),s(S)).
```

The definition of dictionary

```
dictionary(n(room),      [room|X],X).
dictionary(n(bat),       [bat|X],X).
dictionary(n(john),      [john|X],X).
dictionary(n(ball),      [ball|X],X).
dictionary(v(walks),     [walks|X],X).
dictionary(v(hits),      [hits|X],X).
dictionary(p(with),      [with|X],X).
dictionary(p(in),        [in|X],X).
```

The following output shows the parsing process. An input sentence is "John hits ball with bat". Messages passed through a communication channel are specified by 'Message ='. The killed processes are shown by 'process killed'.

```
| ?- test.    /* john hits ball with bat --- ambiguous */

process killed  [pp(pp(X,Y)),[p(X),np(Y)],[hits,ball,with,bat]]
process killed  [np(np(X,Y)),[det(X),n(Y)],[john,hits,ball,with,bat]]

Message = [np(np(john)),[n(john)],[],[john,hits,ball,with,bat],
                [hits,ball,with,bat]]

process killed  [pp(pp(X,Y)),[p(X),np(Y)],[]]
process killed  [np(np(X,Y)),[det(X),n(Y)],[bat]]
process killed  [np(np(X,Y)),[det(X),n(Y)],[ball,with,bat]]

Message = [vp(vp(hits)),[v(hits)],[],[hits,ball,with,bat],
                [ball,with,bat]]
Message = [np(np(ball)),[n(ball)],[],[ball,with,bat],[with,bat]]
Message = [np(np(bat)),[n(bat)],[],[bat],[]]
Message = [s(s(np(john),vp(hits))),[np(np(john)),vp(vp(hits))],[],
                [john,hits,ball,with,bat],[ball,with,bat]]

Message = [vp(vp(hits,np(ball))),[v(hits),np(np(ball))],[],
                [hits,ball,with,bat],[with,bat]]

Message = [pp(pp(with,np(bat))),[p(with),np(np(bat))],[],[with,bat],[]]
```

```
Message = [vp(vp(hits,np(ball),pp(with,np(bat)))),
                [v(hits),np(np(ball)),pp(pp(with,np(bat)))],[],
                [hits,ball,with,bat],[]]

Message = [start(s(np(john),vp(hits))),[s(s(np(john),vp(hits)))],[],
        [john,hits,ball,with,bat],[ball,with,bat]]

Message = [s(s(np(john),vp(hits,np(ball),pp(with,np(bat))))),
                [np(np(john)),vp(vp(hits,np(ball),pp(with,np(bat))))],
                [],[john,hits,ball,with,bat],[]]

Message = [s(s(np(john),vp(hits,np(ball)))),
                [np(np(john)),vp(vp(hits,np(ball)))] ,[],
                [john,hits,ball,with,bat],[with,bat]]

Message = [np(np(np(ball),pp(with,np(bat)))),
                [np(np(ball)),pp(pp(with,np(bat)))] ,
                [],[ball,with,bat],[]]

Message = [start(s(np(john),vp(hits,np(ball)))),
                [s(s(np(john),vp(hits,np(ball))))],
                [],[john,hits,ball,with,bat],[with,bat]]

Message = [start(s(np(john),vp(hits,np(ball),pp(with,np(bat))))),
                [s(s(np(john),vp(hits,np(ball),pp(with, ))))],
                [],[john,hits,ball,with,bat],[]]

Message = [vp(vp(hits,np(np(ball),pp(with,np(bat))))),
                [v(hits),np(np(np(ball),pp(with,np(bat))))],
                [],[hits,ball,with,bat],[]]

Message = [s(s(np(john),vp(hits,np(np(ball),pp(with, ))))),
                [np(np(john)),vp(vp(hits,np(np(ball),pp(with, ))))],
                [],[john,hits,ball,with,bat],[]]

Message = [start(s(np(john),vp(hits,np(np(ball),pp(with, ))))),
                [s(s(np(john),vp(hits,np( , ))))],
                [],[john,hits,ball,with,bat],[]]

 Computation deadlock
Step = 9
yes
```

There are four messages which have parsing trees (the first element of message) with root 'start'. The first two parsing trees correspond to partial parsing tree for input sentences ('john hits' and 'john hits ball'). The last two trees represent different parsing results for the input sentence. The ambiguity of the modification of a prepositional phrase provides two parsing results.

The following examples show the use of an empty derivation rule which specifies the abbreviation of a relational preposition. The messages are sorted.

TEST FOR EMPTY DERIVATION

```
| ?- do([john,hits,mary,hits,lucy]).

Message = [np,[n,srel],[],[john,hits,mary,hits,lucy],[hits,lucy]]
Message = [np,[n,srel],[],[john,hits,mary,hits,lucy],[]]
Message = [np,[n,srel],[],[mary,hits,lucy],[]]
Message = [np,[n],[],[john,hits,mary,hits,lucy],[hits,mary,hits,lucy]]
Message = [np,[n],[],[lucy],[]]
Message = [np,[n],[],[mary,hits,lucy],[hits,lucy]]
Message = [rpro,[],[],[hits,lucy],[hits,lucy]]
Message = [rpro,[],[],[hits,mary,hits,lucy],[hits,mary,hits,lucy]]
Message = [rpro,[],[],[],[]]
Message = [s,[np,vp],[],[john,hits,mary,hits,lucy],[hits,lucy]]
Message = [s,[np,vp],[],[john,hits,mary,hits,lucy],[]]
Message = [s,[np,vp],[],[john,hits,mary,hits,lucy],[]]
Message = [srel,[rpro,vp],[],[hits,lucy],[]]
Message = [srel,[rpro,vp],[],[hits,mary,hits,lucy],[hits,lucy]]
Message = [srel,[rpro,vp],[],[hits,mary,hits,lucy],[]]
Message = [start,[s],[],[john,hits,mary,hits,lucy],[hits,lucy]]
Message = [start,[s],[],[john,hits,mary,hits,lucy],[]]
Message = [start,[s],[],[john,hits,mary,hits,lucy],[]]
Message = [vp,[v,np],[],[hits,lucy],[]]
Message = [vp,[v,np],[],[hits,mary,hits,lucy],[hits,lucy]]
Message = [vp,[v,np],[],[hits,mary,hits,lucy],[]]
process killed  [rpro,[rp],[rp],[hits,lucy],[hits,lucy]]
process killed
[rpro,[rp],[rp],[hits,mary,hits,lucy],[hits,mary,hits,lucy]]
process killed  [rpro,[rp],[rp],[],[]]
process killed  [vp,[v,np],[v,np],[],[]]


| ?- do([john,who,hits,mary,hits,lucy]).

Message = [np,[n,srel],[],[john,who,hits,mary,hits,lucy],[hits,lucy]]
Message = [np,[n,srel],[],[john,who,hits,mary,hits,lucy],[]]
Message = [np,[n,srel],[],[mary,hits,lucy],[]]
Message =
[np,[n],[],[john,who,hits,mary,hits,lucy],[who,hits,mary,hits,lucy]]
Message = [np,[n],[],[lucy],[]]
Message = [np,[n],[],[mary,hits,lucy],[hits,lucy]]
Message =
[rpro,[rp],[],[who,hits,mary,hits,lucy],[hits,mary,hits,lucy]]
Message = [rpro,[],[],[hits,lucy],[hits,lucy]]
Message =
[rpro,[],[],[who,hits,mary,hits,lucy],[who,hits,mary,hits,lucy]]
Message = [rpro,[],[],[],[]]
Message = [s,[np,vp],[],[john,who,hits,mary,hits,lucy],[]]
Message = [srel,[rpro,vp],[],[hits,lucy],[]]
Message = [srel,[rpro,vp],[],[who,hits,mary,hits,lucy],[hits,lucy]]
Message = [srel,[rpro,vp],[],[who,hits,mary,hits,lucy],[]]
Message = [start,[s],[],[john,who,hits,mary,hits,lucy],[]]
Message = [vp,[v,np],[],[hits,lucy],[]]
Message = [vp,[v,np],[],[hits,mary,hits,lucy],[hits,lucy]]
Message = [vp,[v,np],[],[hits,mary,hits,lucy],[]]
process killed  [rpro,[rp],[rp],[hits,lucy],[hits,lucy]]
process killed  [rpro,[rp],[rp],[],[]]
process killed
```

```
[vp,[v,np],[v,np],[who,hits,mary,hits,lucy],[who,hits,mary,hit,lucy]]
process killed  [vp,[v,np],[v,np],[],[]]
```

APPENDIX B

```
%
%     PARSER IN CONCURRENT PROLOG
%
%                                          H.Hirakawa (2nd lab. ICOT)
%
%           Programmed on one rainy day in March 1983
%           Revised on  Friday, 13 May 1983
%
%    This program implements chart parsing using the communication
%    mechanism in Concurrent Prolog.
%
%
%        1. data structure
%
%           Edge  :  [NonTerminal, Derivation, CurDerivation, S, S0]
%
%           Edge represents a status of one process. Arguments in a
%           Edges are
%             NonTerminal : Left hand side of CFG rule
%             Derivation  : Right hand side of CFG rule
%             CurDeri..   : derivation part to be analysed
%             S, S0       : Differential list of input words
%
%          ex.
%              [np, [det,adj,noun], [noun],
[the,happy,boy,..],[boy,..]]
%
%           this edge represents that 'np' derives 'det,adj,noun';
%           'the,happy' is analysed as 'det,adj'; next step is to
%           analyze 'boy...' as 'noun.
%
%        2. process
%
%           process(State,Edge,Channel1,Channel2)
%
%           State is 'active' or 'wait'.  If 'wait', then process waits
%           until some messages(data) are transfered through 'Channel'.
%           'ChannelN' is a common communication pass among all
%           processes.
%
%
%
%    VERSION SPECIFICATION
%
%        This version can't handle variables
%


process(active,Edge,C,C2) :- dict(Edge,NewEdge) |
                                process(active,NewEdge,C,C2).

process(active,Edge,C,C2) :- nonterminalp(Edge) |
                process(wait,Edge,C3,C2,C) // processfork(Edge,C,C2).
```

```
process(active,Edge,C,C2) :- terminate(Edge) | broadcast(Edge,C).

process(active,Edge,C,C2) :- call((write('process killed  '),
                                   write(Edge),nl)).

process(wait,Edge,[TerminatedEdge|C],C2,C3) :-
                  reactivate(Edge,TerminatedEdge,NewEdge) |
                       process(wait,Edge,C?,C2,C3) //
                       process(active,NewEdge,C3,C2).

process(wait,Edge,[|C],C2,C3) :- true | process(wait,Edge,C?,C2,C3).

reactivate(Edge,Terminated,New) :-
          call((Edge=[T,Non1,[N|NonRest],S,S0],
                Terminated=[N,,,S0,S2],
                New=[T,Non1,NonRest,S,S2])) | true.

dict([Head,D,[CurHead|Rest],S0,S],[Head,D,Rest,S0,S1]) :-
                dictionary(CurHead,S,S1).

nonterminalp([H,,[CurHead|],,]) :-
                gr(CurHead,) | true.

terminate(Edge) :-
                Edge=[,,[],,].

broadcast(Edge,Channel) :- call((var(Channel) , Channel=[Edge|]))
                                | true.
broadcast(Edge,Channel) :- Channel=[|C] | broadcast(Edge,C).

processfork(Edge,C,C2) :- call(tried(Edge,C2)) | true.
processfork([Head,ConstDList1,ConstDlist2,,S],C,C2) :-
          call(( ConstDlist2=[NonTerm|],
                register([Head,ConstDList1,ConstDlist2,,S],C2),
                rules(NonTerm,RuleList))) |
                forks(RuleList,S,C,C2).

tried(Edge,C2) :- var(C2), !, fail.
tried(Edge1,[Edge2|]) :- Edge1=[,,[Gtop|],S0,S1],
                Edge2=[,,[Gtop|],S0,S1], !.
tried(Edge,[|C2]) :- tried(Edge,C2).

register(Edge,[Edge|C2]) :- !.
register(Edge,[|C2]) :- register(Edge,C2).

rules(NonTerm,RuleList) :-
                setof0([NonTerm,X],gr(NonTerm,X),RuleList).

forks([],,,).
forks([Rule|Rest],S0,C,C2) :-
          call(( trace(fork,(Rule + Rest)),
                Rule=[,Consts],
                append(Rule,[Consts,S0,S0],NewEdge))) |
                process(active,NewEdge,C,C2) // forks(Rest,S0,C,C2).
```

GRAMMAR AND DICTIONARY

```
%

gr(s, [np,vp]).

gr(vp,[v,np,pp]).
gr(vp,[v]).
gr(vp,[v,np]).

gr(np,[n]).
gr(np,[n,pp]).
gr(np,[n,srel]).

gr(pp,[p,np]).

gr(srel,[rpro,vp]).

gr(rpro,[]).
gr(rpro,[rp]).

gr(start,s).

dictionary(n,[john|X],X).
dictionary(n,[lucy|X],X).
dictionary(n,[mary|X],X).
dictionary(n,[room|X],X).
dictionary(n,[bat|X],X).

dictionary(v,[walks|X],X).
dictionary(v,[hits|X],X).

dictionary(rp,[who|X],X).

dictionary(p,[with|X],X).
dictionary(p,[in|X],X).

parse(S) :- true |
                process(active,
                        [start,[s],[s],S,S],
                        C,C2) // writemsg(C?).

do(S) :- solve(parse(S),Res,N),nl,nl,write(' Computation '),write(Res),
        nl,write('Step = '),write(N).
do1(S) :- solve(parse(S)).

doo :- do([john,walks]).

test1 :- true |
                process(active,
                        [s,[np,vp],[np,vp],[john,walks],[]],
                        C,C2) // writemsg(C?).
test :- true |
                process(active,
                        [s,[np,vp],[np,vp],[john,walks],[john,walks]],
                        C,C2) // writemsg(C?).
writemsg([X|Rest]) :- call((write('Message = '),write(X),nl))
                        | writemsg(Rest?).
```