Chapter III

CUPARSE System

CUPARSE is a software used in this study to implement the dependency grammar proposed for the parsing of the corpus of 50 sentences. It runs on an IBM AT compatible with 640 KB memory.

3.1 Introduction to CUPARSE

CUPARSE uses chart as data representation and the dependency theory approach for linguistic analysis. An input sentence must be a list of segmented words. The output from CUPARSE can be one or more D-trees and conceptual networks depending on whether the sentence is ambiguous.

Following is an overview of CUPARSE, with a description of the concepts of window, rule, link[1], and link order.

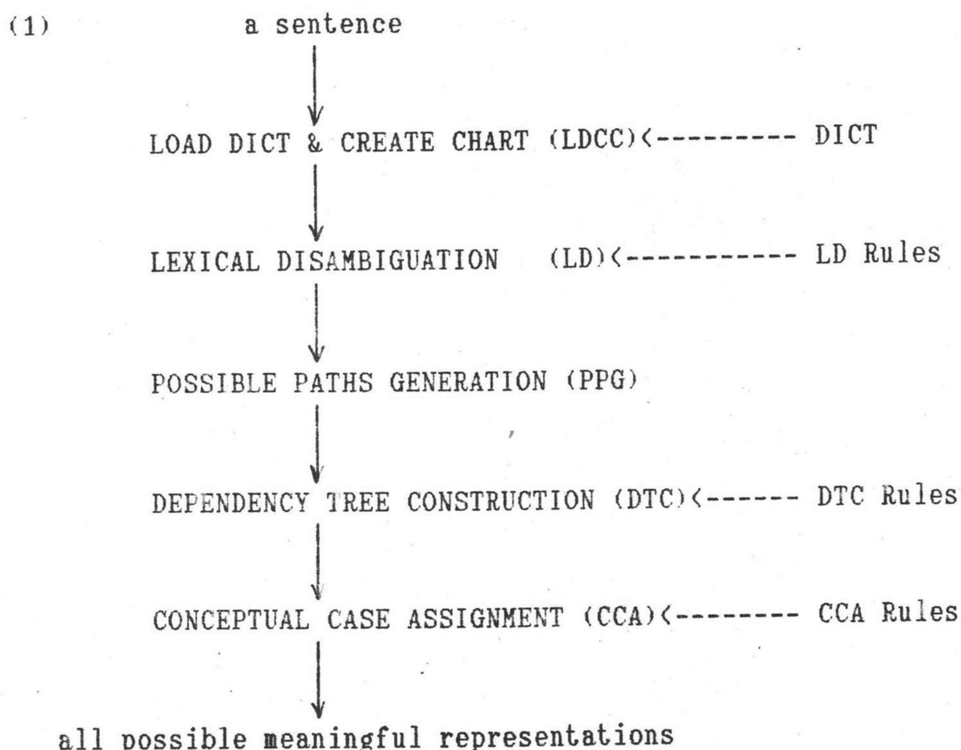
---

1. The concepts of window scanning and rule organization are in part adopted from the analysis phase of the PIVOT Machine Translation system designed by the staff of the NEC laboratory and used as an implementation system in the joint MT research between the Ministry of Sciences, Technology and Energy of Thailand and the Ministry of International Trade and Industry of Japan. The author would like to express his appreciation to NEC for generously sharing these concepts with Thai researchers. CUPARSE is an attempt to try out these concepts in a different system.

3.1.1 Overview of CUPARSE

There are five modules in CUPARSE as follows:

(1)



Following is an overview of each module.

1. Load Dictionary & Create Chart (LDCC)

LDCC is the first module in CUPARSE system. Its function is to load all information of lexemes in the dictionary which correspond to each wordform in an input sentence, and then creates a chart structure. Each lexeme occupies an edge in the chart. Ambiguity results in more than one edge between two vertices in the chart.

2. Lexical Disambiguation (LD)

Since each wordform can be ambiguous, this module aims at reducing the number of ambiguity as much as possible. By using LD Rules, we can decide which edge can be deleted from the chart. However, some ambiguities may still remain. In another word, there may still be more than one edge between two vertices in the chart because category information is not enough for disambiguation in all instances.

3. Possible Path Generation (PPG)

This module creates a path which is the connection of edges from the first vertex through the last vertex by selecting only one edge from each two vertices in the chart. The sequence of lexemes generated will be treated as a set of possible paths. These possible paths will be outputted for the syntactic and semantic analysis in the next module. At this point, it is still possible to have more than one path if ambiguity still remains.

4. Dependency Tree Construction (DTC)

Every possible path outputted by PPG module, will be converted into a D-tree with the use of DTC rules which determine whether or not there is a syntactic relation between each pair of wordforms on the path. Those which successfully pass through DTC rules and are converted into D-trees will be inputted to the next module.

5. Conceptual Case Assignment (CCA)

This module determines whether or not each syntactic relation linking a pair of nodes in a D-tree can be mapped onto a conceptual relation. The status of head and depender may be changed at this stage. A D-tree in which there remain syntactic relations, which cannot be converted into conceptual relations, fails at this stage. The output from this module is one or more conceptual networks of the input sentence. The multiple outputs result from semantic ambiguity.

3.1.2 Windows

In any operation command, we must know the scope of reference for the operation. In some parsers, such as the ATN parser, there is only one focused position as a scope of reference. The system performs operation on one word at a time from the beginning of the sentence to the end. Some parsers have a wider range of reference. PARSIFAL uses three buffers for making a reference of words in an

input sentence. CUPARSE also uses a multiple focused approach known as windows.

3.1.3 Rule file

A rule file is a set of rules used in LD, DTC and CCA modules of the CUPARSE system. Each rule in a rule file is a set of operation commands, composed of condition and action commands. All action commands in a rule will be operated if all condition commands in that rule are satisfied. Condition and action commands are grouped and titled with a condition and an action name respectively. All rules are referred to by their names in a link. The following is the example of rule VpN.

```
(2) <CVpN>                              ==> <Condition name>
    [ intersect(*.MAJCAT,"V");          ==> [ Condition command;
      intersect(+.MINCAT,"PREP");       ==>   Condition command;
      intersect(r.MAJCAT,"N");          ==>   Condition command;
    ]                                   ==> ]
    <AVpN>                              ==> <Action name>
    [ rlink(*,+);                       ==> [ Action command;
      rlink(+,r);                       ==>   Action command;
      add(*.PATT,+.SYNTC);              ==>   Action command;
      copy(*.MAJCAT,"V");               ==>   Action command;
      copy(r.MAJCAT,"N");               ==>   Action command;
      combine(+,r);                     ==>   Action command;
      combine(*,+);                     ==>   Action command;
    ]                                   ==> ]
```

3.1.4 Link file

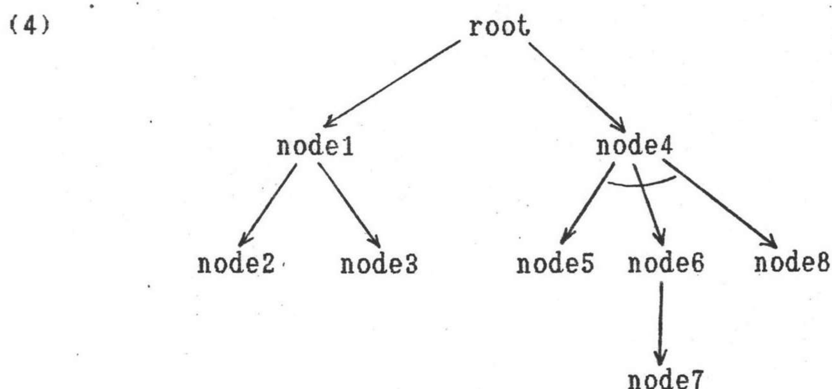
A link file is a set of links used in LD, DTC and CCA modules. A link is one part of process in a module. The whole process of analysis in each module depends on the links put into order. A link is composed of rules branching as a rule tree. The rules are referred by condition names or action names. Condition names begin

with a character "C" while action names begin with a character "A". The following is an example of a link.

```
(3) <LRel>                                        ==> <Lname>
   [{ CR1SubMiss -> Adummy                        ==> [ { node1
      { CRMSubAgt & CR1csAGT -> AR1Agt }          ==>      { node2 }
      { CRMSubObj & CR1csOBJ -> AR1Obj }          ==>      { node3 }
    }                                             ==>    }
    { CR2SubMiss -> @                             ==>    { node4 @
        { CRMSubAgt & CR1csAGT -> AR1Agt }        ==>        { node5 }
        { CRMSubIns ->                            ==>        { node6
          { CR1csINS | CR1dfINS -> AR1Ins} }      ==>          { node7 }}
        { CRMFobObj & CR2csOBJ -> AR2Obj}         ==>        { node8 }
    }]                                            ==>    }]
```

The symbols "|" and "&" used for conditions "OR" and "AND" respectively between condition names. Curly brackets { } enclose each node in the rule tree. The node at the inner boundary is the child node of the node at the nearest outer boundary. For example, { CRMSubAgt & CR1csAGT -> AR1Agt } or { node2 } is the child node of { CR1SubMiss -> Adummy } or { node1 }. The rule tree structure of the link in (3) is organized as follows.

(4)



The head node can have two types of child nodes in a rule tree, inclusive_or child node and exclusive_or child node. The former, which is marked by the symbol @ at the head node, such as the

nodes 5, 6 and 8, is used in case more than one successful child node is allowed. The latter, such as nodes 2 and 3, is used in case only one successful child node is allowed.

The format "condition -> action" is like the conditional statement, "if...then...else", in a computer language. A link, therefore, is like a computer program that is the combination of conditional statements. Three types of conditional statement combinations can be represented in a link as follows:

1. Multi-branch if

This combination corresponds to exclusive_or child nodes combination in a rule tree. For example, (5a) corresponds to (5b) as follows.

```
(5) a) if CR1SubMiss then Adummy
          if CRMSubAgt and CR1csAGT then AR1Agt
          else if CRMSubObj and CR1csOBJ then AR1Obj
          endif
        endif
    b) { CR1SubMiss -> Adummy
          { CRMSubAgt & CR1csAGT -> AR1Agt }
          { CRMSubObj & CR1csOBJ -> AR1Obj }
        }
```

In this case, only one action is performed. If CRMSubAgt and CR1csAgt succeed then AR1Agt operates. But if one of them fails, CRMSubObj and CR1csOBJ will be checked and if they succeed AR1Obj will operate.

2. Sequence of single-branch if.

This combination corresponds to inclusive_or child nodes combination in a rule tree. For example, (6a) corresponds to (6b) as follows.

```
(6) a)  if CR2SubMiss then
            if CRMSubAgt and CR1csAGT then AR1Agt endif
            if CRMSubIns then
                if CR1csINS or CR1dfINS then AR1Ins endif
            endif
            if CRMFobObj and CR2csOBJ then AR2Obj endif
        endif
    b) { CR2SubMiss -> @
            { CRMSubAgt & CR1csAGT -> AR1Agt }
            { CRMSubIns ->
              { CR1csINS | CR1dfINS -> AR1Ins} }
            { CRMFobObj & CR2csOBJ -> AR2Obj}
       }
```

In this case, the conditions "CRMSubAgt and CR1csAGT", "CRMSubIns", and "CRMFobObj and CR2csOBJ" do not affect each other. If any condition succeeds, the action that corresponds to it will operate.

3. Nesting-if

This combination corresponds to the combination of head nodes and child nodes. For example, (7a) corresponds to (7b) as follows.

```
(7) a) if CR1SubMiss then Adummy
           if CRMSubAgt and CR1csAGT then AR1Agt endif
           if CRMSubObj and CR1csOBJ then AR1Obj endif
       endif
    b) { CR1SubMiss -> Adummy
           { CRMSubAgt & CR1csAGT -> AR1Agt }
           { CRMSubObj & CR1csOBJ -> AR1Obj }
       }
```

This case is different from the conditional statement in a computer language in that the actions "Adummy" and "AR1AGT" will

operate only when the conditions "CR1SubMiss" and "CRMSubAgt and CR1csAGT" succeed. If one of these conditions fails, none of the actions will be taken.

These three constructions are the basic structures of the rule tree in CUPARSE. The system uses a rule tree, or a link, as a flow of operations. By searching through the rule tree, the CUPARSE system will find only one successful path from the root node through to the leaf node, and then perform the operations as specified by the actions along that successful path. The successful path means that all conditions of each node in that path succeed. The successful path will be searched as a depth-first left search. The successful path and the actions taken in the link (3) is either one of these following paths searched in order.

```
(8) CR1SubMiss-CRMSubAgt-CR1csAGT          ===> Adummy-AR1Agt
    CR1SubMiss-CRMSubObj-CR1csOBJ          ===> Adummy-AR1Obj
    CR2SubMiss-CRMSubAgt-CR1csAGT-CRMSubIns-CR1csINS-CRMFobObj-CR2csOBJ
                                           ===> AR1Agt-AR1Ins-AR2Obj
    CR2SubMiss-CRMSubAgt-CR1csAGT-CRMSubIns-CR1dfINS-CRMFobObj-CR2csOBJ
                                           ===> AR1Agt-AR1Ins-AR2Obj
    CR2SubMiss-CRMSubAgt-CR1csAGT-CRMSubIns-CR1csINS
                                           ===> AR1Agt-AR1Ins
    CR2SubMiss-CRMSubAgt-CR1csAGT-CRMSubIns-CR1dfINS
                                           ===> AR1Agt-AR1Ins
    CR2SubMiss-CRMSubAgt-CR1csAGT          ===> AR1Agt
```

3.1.5 Link order file

A link order file is a set of link orders. A link order is a sequence of links used in a module. The analysis of each module depends on these links. CUPARSE is designed to have more than one link order so it will analyze the input sentence in every possible link order because each link order is a different alternative for the analysis. This is to account for syntactic ambiguity such as in the

following sentence.

(9) "The robot moves the box in the room."

The ambiguity occurs because the noun phrase "in the room" can modify either "the box" or "moves"; therefore, two link orders are possible. The former link order handles the modification of NP while the latter handles the modification of the verb. Multiple link orders make it possible to obtain duplicate output. The following is an example of a link order.

(10) <Linkorder1>

LRelMissSubFob

LCaseAssign

LCaseAssign1

LCaseFrame

3.2 LDCC Module

LDCC's function is to load all information of the lexeme which correspond to each wordform in the input sentence in the dictionary, and then create a chart structure. A chart is a graph, or a set of vertices, each pair of which is linked by an edge. Lexemes loaded from the dictionary will be arranged as different edges in the chart. It is possible that one wordform may match more than one lexeme in the dictionary, a case of lexical or word ambiguity; therefore, there may be more than one edge between two vertices in a chart.

For example, If the correspondence between wordforms and lexemes in the dictionary is like in (11a), after dictionary loading, the chart structure created will be like (11b).

(11) a) Wordform: W1 Lexemes: L13, L18, L25

Wordform: W2 Lexemes: L4, L5

Wordform: W3 Lexemes: L9, L11

Wordform: W4 Lexemes: L53, L60

b)

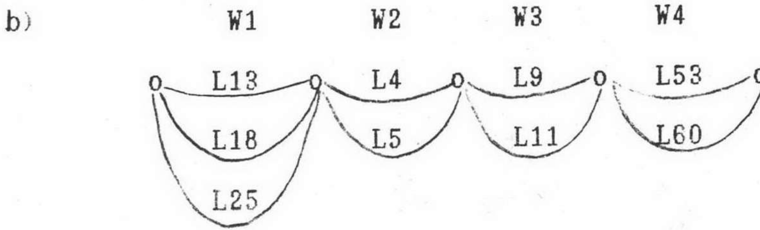


chart structure

3.3 LD Module

LD's task is to reduce the lexical ambiguity or the number of edges between two vertices through the operation of LD rules. Following is a description of windows, commands, control strategy, and type of rules in this module.

3.3.1 Windows

In this module, there are five continuous windows as follows:

1. Focus Window (*): This is the central window. It is used to refer to the focused position of a rule.

2. Next Window (+): This window is used to refer to the right adjacent position of the focused window.

3. Prior Window (-): This window is used to refer to the left adjacent position of the focused window.

4. Right Boundary Window (r): This is the right boundary of the window scope. It is used to refer to the right position next to the + window.

5. Left Boundary Window (l): This is the left boundary of the window scope. It is used to refer to the left position adjacent to the - window.

A window is used to refer to all edges between two vertices (or a segment). In other words, all lexemes between two vertices are referred to by windows during the operation of a rule. In addition, there are six extra windows, A B C D E F, used in the same manner as the five continuous windows. These windows make references by the use of "lsearch" or "rsearch" command.

3.3.2 Commands

The symbol "~" is added to reverse the return value of the command. There are 9 commands as follows:

1. (~) wbind(W);

This command checks whether or not the window specified is bound to any segment. It returns SUCCESS or FAIL.

2. (~) single(W);

This command checks whether or not at the position specified by the window, there is only one lexeme.

3. (~) intersect(arg1, arg2);

This command checks whether or not there is any intersection value between argument 1 and argument 2. Arguments 1 and 2 refer to the segments bound; therefore, the command checks whether there is at least one intersection value between the lexemes specified by argument 1 and the lexemes specified by argument 2 or not. Argument 2 may be constant values specified in the command, or values of the specified window and feature. If the constant value is null, the command will check whether or not argument 1 exists. If there is no such specified feature in the argument 1 or 2, it returns FAIL.

4. (~) subset(arg1, arg2);

This command is like the "intersect" command, but it checks whether argument 2 is a subset of argument 1.

5. (~) equal(arg1, arg2);

This command is like the "intersect" command, but it checks whether or not argument 2 is equal in value to argument 1.

6. cut(W,{F[V],F[V],..});

This command will delete any lexeme or edge at the segment bound, if it has the features and values as specified in the list.

7. select(W,{F[V],F[V],..});

This command is the opposite of the "cut" command. It deletes the edges that do not correspond to the conditions specified.

8. (~) rsearch(W,EW,{condition list... });

This command searches for the first segment that matches all conditions specified in the condition list. It begins searching from the next segment to the right of argument 1 through to the last segment in the chart. It will stop searching when it finds the segment that matches the conditions specified. If it cannot find that segment, it returns FAIL, but if that segment is found, it will bind that segment with the extra window specified by argument 2. The condition command specified in the condition list is "intersect", "subset", or "equal" command.

9. (~) lsearch(W,EW,{ condition list... });

This command is like the "rsearch" command except that it begins searching leftward.

3.3.3 Control strategy

Each link in a link order is applied in sequence. At the initial state, the * window is bound to the first segment in the chart. The engine will find a successful path in a link, and then perform the action commands if the successful path is found. The window scope will be shifted right for one position regardless of whether or not a successful path is obtained. Each link will be stopped when the * window finds no segment to bind.

3.3.4 Types of LD rule

Disambiguation usually involves more than one segment. Three disambiguation methods are used in this module.

1. Deletion rule

There must be at least one lexeme specified by the rule to the left or the right of the focused position.

(12) a) if X then the left must be P,Q,R,...

b) if X then the right must be P,Q,R,...

X is the lexeme at the focused segment. If to the left or right of this segment none of the lexemes specified (P,Q,R,..) exists, then X is not a correct lexeme for this segment. By this rule we can delete lexeme X from the segment.

b. Selection rule

At the focused position which is not ambiguous, we can check what can be to its left or right.

(13) a) if X then the left must be P, Q, R,...

must not be A, B, C,...

b) if X then the right must be P, Q, R,...

must not be A, B, C,...

X is the unambiguous lexeme at the focused position. This rule deletes those lexemes to its left or right, which violate this rule. The lexemes A, B, C,.. and the lexemes other than P, Q, R,.. will be deleted.

c. Probability Rule

Although all segments are often ambiguous, there are frequent patterns which are helpful in the disambiguation process. For example, if (14a) is the ambiguous sequence and (14b) is one of the probability patterns, other lexemes that are not B, P, Y will be deleted by this rule and (14c) will be selected.

(14) a)

segment1	segment2	segment3
A	P	W
B	Q	X
C	R	Y
D		Z

b) if the pattern B P Y is found select this pattern.

c) segment1 segment2 segment3

segment1	segment2	segment3
B	P	Y

3.4 <u>PPG Module</u>

More than one edge may remain in a segment in the output of LD module. This module will create a path by selecting one edge in each segment, from the first segment through to the last segment. The number of possible paths generated is equal to the multiplication of the number of edges of each of the segments. All possible paths are treated as alternatives of analysis. Each path which is a lexeme sequence will be analyzed independently. For example, (15a) is the output from LD and (15b) lists the possible paths generated from PPG.

(15) a)

segment1	segment2	segment3	segment4
A	B	C	D
	E	F	G

b) The possible paths generated

A B C D, A B C G, A B F D,
A B F G, A E C D, A E C G,
A E F D, A E F G,

3.5 <u>DTC Module</u>

DTC is the most important part in CUPARSE. Its main task is to create a D-tree from a lexeme sequence. A D-tree consists of nodes and arcs linking nodes. A node is labeled with a wordform while an arc is labeled with a syntactic case.

In the following example, nodes X and Y are treated as content nodes while node R is treated as a case node. In CUPARSE, the same data structure is used for both types of node.

R

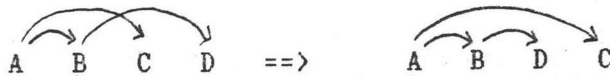
(16) X------->Y is treated as X--->R---->Y.

Each node contains information of the lexeme it represents or case information. New information may be added during the process. It consists not only of syntactic information but also semantic information, which will be used in the CCA part too.

CUPARSE provides a set of operation commands for constructing a D-tree. One lexeme sequence may output more than one D-tree depending on the number of link orders.

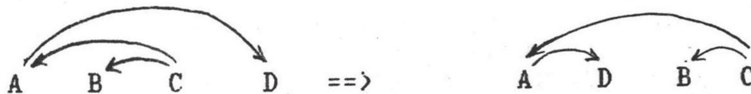
The first step in this module is to make sure that the input sentence is a non-projective sentence (see 2.2.1) because this type of sentence will violate the simple adjacency principle (see 2.2.4) adopted for CUPARSE analysis. If non-projectivity exists, it has to be undone first by moving the critical lexeme to the appropriate position, as in the following examples.

(17) a) b)



There is a crossing dependency between A->C and B->D in (17a). CUPARSE cannot construct the relation between B and D because B is not adjacent to D. This lexeme sequence must be adjusted by moving D to the position adjacent to B as in (17b) before any relation with B is constructed.

(18) a) b)



C is the top node in (18a). A->D is an arc covering the top node. CUPARSE cannot construct the relation between A and D because they are not adjacent to each other. The lexeme sequence must be adjusted by moving D next to A as in (18b) before any relation with A is constructed.

3.5.1 Windows

There are five continuous windows (l - * + r) and six extra windows (A B C D E F) in this module like in LD module. A window is used to refer to a lexeme in the active path. The extra windows are bound to any lexemes by the use of "rsearch" or "lsearch" command.

3.5.2 Active path

The active path is a sequence of lexemes linked as a list. A dependency relation will be constructed only on the lexemes which are on the active path. After the relation is constructed, the depender lexeme will be moved out of the active path and the linking of lexemes on the active path will be changed. For example, if (19a) is an initial state of the active path, and (19b) is a construction product, then the active path will be changed to (19c). At the end of DTC analysis, there should be only one lexeme in the active path as the root node of a D-tree unless that input path syntactically fails.

(19) a) Active path: A B C D E F G H

b) construction: E->F

c) Active path: A B C D E G H

3.5.3 Commands

The commands in this module are like the commands in LD, but the windows specified in this module is used to refer to the lexemes in the active path, not the segments in the chart. There are 15 commands in this module. Six, namely "wbind", "intersect", "subset", "equal", "lsearch", and "rsearch", are the same as those in LD module (see 3.3.2). The following commands are added.

1. add(arg1, arg2);

This is an action command. It adds the values from argument 2 to argument 1. If the specified feature in argument 1 does not exist, the feature will be created and the value from argument 2 will be added to it.

2. copy(arg1, arg2);

This command is like the "add" command except that the existing value in argument 1 is replaced by the value of argument 2.

3. delete(arg1, arg2);

This command deletes all values specified in argument 2 from argument 1. If no such specified feature in argument 1 exits, it returns FAIL.

4. combine(W,W);

This command combines the lexemes specified in arguments 1 and 2. Arguments 1 and 2 must be adjacent windows, such as (*,+), (+, *), etc. Argument 1 is the head node while argument 2 is the depender. Since this command moves the lexeme specified by argument 2 out of the active path, updates the linking in the active path, and automatically changes the binding between windows and lexemes, it must be used after all linking has been made; otherwise it causes an error.

5. move(W, W);

This command moves and connects the lexeme specified by argument 2 to the right of the lexeme specified by argument 1, and updates the linking of the lexemes in the active path. Arguments 1 and 2 may be either the regular window or the extra window. This command moves any lexeme in the active path, and rearranges the sequence of lexemes in the active path. After this command, the window status will change to the initial state in which the * window is bound to the first lexeme in the resulting active path.

6. alloc(N);

This command creates one new node for further use by other commands such as "llink" and "rlink" in dependency construction operation.

7. llink(W| N , W| N);

This command assigns a dependency relation between two nodes

specified by arguments 1 and 2. Argument 1 is the head of argument 2. This command is used when the depender is on the left of the head.

8. rlink(W¦ N , W¦ N);

This command is like "llink" command except that it is used when the depender is on the right of the head.

9. shift(R¦ L¦ E¦ B);

This is a window moving command. There are four possible movements in this command.

a. Shift(R); This command moves the window scope one position to the right.

b. Shift(L); This command moves the window scope one position to the left.

c. Shift(B); This command changes the window scope to the initial state where the * window is bound to the first lexeme in the active path.

d. Shift(E); This command changes the window scope to the ending state where all windows are not bound to any lexeme in the active path.

3.5.4 Control strategy

Each link will be applied in sequence as specified in the link order. At the initial state, the * window is bound to the first lexeme in the active path. The engine will find a successful path in a link. If a successful path is not found, the window scope will shift one position rightward automatically and try to find a successful path in that link again. After a successful path is found, the action commands will be performed and the search for successful path continues. Each link will be applied until the * window is out of the active path.

All operation commands will operate on the active path, and the window scope and the linking of lexemes on the active path is automatically changed after dependency construction. The system does

not shift the window scope to the right when a successful path is found as in the LD module. Shifting is operated only by the use of "shift" command, or in the case that a successful path is not found.

3.6 CCA Module

CCA is the last module in CUPARSE. It assigns a conceptual case and determines conceptual dependency direction to the dependency relation between lexemes in a D-tree. This section describes this module in details.

3.6.1 Windows

Unlike in earlier modules, there are only three continuous windows (- * +) in the reference scope. The movement of the window scope in this module is different from that in LD and DTC modules. In this module, the window scope scans a D-tree in the depth-first left search manner.

1. Focus Window (*): This is the central window. It is used to refer to a case node.

2. Next Window (+): This window is used to refer to the syntactic depender.

3. Prior Window (-): This window is used to refer to the syntactic head.

There are six extra windows (A B C D E F) in this module. They can be bound to any nodes by the use of "nsearch", "csearch", or "child" command.

3.6.2 Commands

The commands in this module are like those in DTC module, but the windows used in this module refer to nodes in a D-tree. There are 10 commands in this module. Six, namely "intersect", "subset", "equal", "add", "copy", and "delete" are the same as those in DTC

module (see 3.5.3). The following commands are added.

1. semlink(W, W);

This command assigns a conceptual dependency direction between two nodes. Argument 1 is the head of argument 2.

2. child(W, W);

This command binds the extra window specified in argument 2 to the node that is the first child node of argument 1.

3. (~) csearch(+| -, W,{condition list... });

This command searches for the case node which is the child node of argument 1 and matches all conditions in the condition list. If that case node is found, it will be bound with the extra window specified by argument 2. Argument 1 can be either the *, or - window.

4. (~) nsearch(+| -, W,{condition list... });

This command is like "csearch" command except that it searches for a concept node which is the grandchild node of argument 1 and matches all condition specified in the condition list.

3.6.3 Control strategy

Each link in this module is applied in the same manner as that described in LD, and DTC modules. The difference is in the scanning of the window scope. At the initial state, the - window is bound to the root node, the * and + windows are bound the first child and grandchild node of the root respectively. If neither of a successful path is found or not, the window scope is shifted to the next deeper level. If no such level exists, the windows will be bound to the sister node at the same level. This depth-first left traversing can be represented as in the following example.

(20)