LISPP: AN INTERACTIVE EXTENDED LISP SYSTEM

Carlo Montangero Giuliano Pacini

Nota Interna B73-7

Aprile 1973

Istituto Scienze dell'Informazione
Università di Pisa

Consiglio Nazionale delle Ricerche

ISTITUTO DI ELABORAZIONE DELLA INFORMAZIONE

PISA

LISPP: AN INTERACTIVE EXTENDED LISP SYSTEM

Carlo Montangero Giuliano Pacini

Nota Interna B73-7

Aprile 1973

Istituto Scienze dell'Informazione
Università di Pisa

Presentato a SEAS Spring Technical Meeting 1973 on Interactive Computing, Rimini, Aprile 1973.

Abstract

The paper describes some characteristics of an interactive extended LISP system (LISPP). The system is presently under development, it is written in PL/I and is intended to be an experimental tool for the design and implementation of problem-solving languages which are employed in Artificial Intelligence research. LISPP is oriented towards non-deterministic programming, through its generalize control structure and "context mechanism", which allow saving the state of the system at any time, then continuing the computation and, if necessary, reintegrating any previously saved state.

The generalized control structure and the context mechanism are described, together with their implementation; finally it is stressed how these features lead to a very natural and flexible interaction between the user and the system in writing and debugging programs: indeed the system provides a frame in which an attempt (e.g. re-definition of functions, new test assignments) can be done, retaining the old situation for alternative attempts.

1 - Introduction

The paper describes the main characteristics of an interactive extended LISP system. The system is presently under development, it is written in PL/I and is intended to be an experimental tool for the design and implementation of problem-solving languages which are employed in Artificial Intelligence [1,2] . By problem-solving languages we intend languages as PLANNER [3], QA4 [4], POPLER [5] which have been developed in the last years to facilitate the construction of the large programs that are needed in application of AI as:

- question answering systems
- natural language comprehension
- automatic theorem-proving
- planning actions for robots
- interactive or automatic demonstration of the correctness and termination of programs and program schemas
- automatic program synthesis.

The late results in these fields can be found in the proceedings of recent congresses [6,7]; moreover several survey papers exist in the literature [8,9,10].

The target of these problem-solving languages is to provide a natural vehicle to translate into programs the common-sense reasoning about the problems that we listed above. A complete discussion of these languages can be found in [8]: we simply point out that these languages provide, to some extent, more complex control regimes than the simple LIFO organization, adopted in languages as LISP and ALGOL. These control regimes include backtracking (non deterministic programming), parallel processes, cooperating processes.

The purpose of this paper is to outline the way we are extending our LISP system in order to supply the language with a small number of primitives that allow flexible programming of complex control regimes. The system is extended basically through the introduction of a generalized control structure and a "context mechanism", which allow the programmer to save the state of the system at any time, to continue the computation and, if it is necessary, to reintegrate any previously saved state. This necessity arises mainly in non-deterministic programming, which is the first application we foresee for the system.

It must be pointed out that we do not intend to develop a language with powerful primitives which directly provide specific control structures: on the contrary, LISPP is intended to be a system endowed with a few elementary primitives allowing to program a large and graded range of complex control regimes. In fact, LISPP will be a basic nucleus for the development of problem-solving languages: providing a frame in which various control structures can be easily implemented and their feasibility easily tested.

The experimental aim of the system lets foresee a feed-back on the structure of the nucleus itself; it seems possible that the actual experimentation will suggest to introduce new facilities or to adapt old ones. This experimental nature of the system motivated our decision to write the system in a high level language, namely PL/I. This choice should make the system handy and easily modifiable. These characteristics seem in this first phase more important than efficiency itself.

The generalized control structure and the context mechanism have been introduced into the system because of the motivations that we mentioned above: however, as it is shown in the last section, these features lead also to a very natural and flexible interaction between

the user and the system in writing and debugging programs. Indeed, the system provides a frame in which it is possible to attempt corrective actions (e.g. redifinition of functions, new test assignments) retaining the old situation for alternative attempts.

2. The generalized control structure.

Most of the older programming languages, as FORTRAN, ALGOL, LISP, APL, SNOBOL have a control structure which is based on a strict LIFO discipline for procedure invocation and return. In these languages, to invoke a procedure, it is enough to specify the name and the arguments of the procedure. In fact, more information is needed to execute the procedure correctly. Namely:

(1) the free variable link (FVL), i.e. the information where to get the values of free variables;

(2) the return link (RL), i.e. where to return. These need not appear explicitely in the procedure invocation, because they are defined implicitely according to the rules of the language. The rule for the return is common to all the languages mentioned above: the return always occurs in the calling procedure. On the contrary, the rule which defines the free variable link depends on the language: e.g. in LISP it is dinamically defined by the calling procedure, while in ALGOL it is statically defined by the structure of the program.

By generalized control structure we mean an organization that allows to specify both the FVL and the RL in the procedure invocation.

The following simple model allow us to discuss the main characteristics of such an organization. A memory block (activation module) is allocated for each activation of a procedure. The activation module consists of a control element and an association area. The control element has a fixed size, and it is used to store:

1) the return link

2) the free variable link

3) the procedure definition (1)

4) the continuation point.

1 and 2 are pointers to activation modules and, just as 3, are defined at activation time; the continuation point, on the contrary, is defined, as we will see more precisely, only when another invocation occurs. The association area contains the values of local variables and parameters: its size obviously depends on the procedure.

When the procedure A calls B, the continuation point in A is stored in the module of A itself; a module for B is allocated and the links are defined as it is specified in the invocation. When a return occurs, the continuation point is taken from the module pointed by the return link.

In a purely recursive organization the continuation point can be stored in the module of the calling as well as in that of the called procedure (and the last solution is generally used). On the contrary, in this generalized control structure, the continuation point must be stored in the calling module, to allow reentering a procedure from any other.

In such an organization, moreover, the programmer must have the possibility to handle pointers to activation modules.

Fig. 1 shows an example of a control regime that can be easily programmed in the generalized control structure: A activates B which in turn may generate a calling chain until the procedure R activates B' with the links pointing to A. B' activates in turn other procedures,

---

(1) We think of an interpretative system: hence the necessity of this information.

until an R' is called with RL pointing to R, so that R is reentered. Now R generates a new calling chain until a procedure S reenters the activation module Q' invoking a procedure T with RL pointing to Q'. This is essentially a coroutine control regime: there are two coordinate cooperating processes (B,.....R,S,.. and B',....Q',.....); the control jumps from one to the other.

LISPP is intended to realize a generalized control structure, as it is outline above: nevertheless we limit the possibility to specify the FVL and RL to a few functions. This choice, that allows to avoid an excessive overload for the interpreter, is not an actual limitation to the generality of the control structure, because the functions for which it is possible to specify RL and FVL include the two basic components of the interpreter (namely eval and apply). For instance aswitch (i.e. apply switch: to point out that it allows to depart from the standard LIFO discipline) is a generalization of the function apply and takes 4 arguments

aswitch [fn; args; fvl; rl] ;

the activation of aswitch results in the allocation of an activation module, in which the links are set to the last two arguments of aswitch itself, and in the application of fn to the list of arguments args: everything works as if fvl and rl were specified in the invocation of fn.

As to the pointers to activation modules, they are accessible essentially by the use of a function

controlnth [n]

where n is an integer; this function return the pointer to the nth module back in the chain of return links.

3. The context mechanism.

The generalized control structure that we described has many general features, and allows programming rather complex control regimes: for instance coordinate cooperating processes (coroutines) can be easily programmed [2] .

However, this structure is still limited, because it is not able to cope, in a natural way, with control regimes that need to save the state of the computation. This necessity is peculiar to systems oriented towards non-deterministic programming, as it is the case with problem-solving systems.

For instance, to program a backtracking control regime (which is a widely used tecnique for problem-solving) it is necessary to fix the state of the computation, whenever a choice is done, so that the fixed situation can be easily restored if the last choice has proved unsatisfactory and then a new choice has to be done.

In this section we briefly outline a model of a "context mechanism" in which the programmer operates as if he can ask for a copy of the situation that he wants to retain. More precisely we assume that the system has a storage pool, organized in a free-list: activation modules are obtained from the free list and returned to it either by garbage collection or by explicit release by the programmer. We call context this storage pool and context state its contents, i.e. the set of all allocated activation modules. In the model the programmer is able to ask for the allocation of a new storage pool, (a new context) initialized to the contents of an already existing one: moreover, he can (later) specify in which context the computation must proceed; naturally tha state of the other contexts will not be altered.

In order to deal with contexts, we introduce the function

context [ ]

which returns the name of the context in which it is executed, and

the function

newcontext [cxt]

that creates a new context whose state is set to that of context cxt, and returns the name of the newly created context. Actually newcontext simulates the allocation of a new storage pool in which the contents of the storage pool named cxt is copied.

The functions that permit to change context are those that allow the specification of the FVL and RL: it is then necessary to specify, in the function invocation, also the contexts in which the activation modules pointed by the FVL and RL are to be considered. For instance, the function aswitch has two more arguments:

aswitch [fn; args; fvl; rl; cxtl; cxt2]

meaning that fn must be applied to the list of arguments args; the application occurs in the context cxtl, using fvl as the free variable link: the return occurs in the activation module pointed by rl in the context cxt2.

As a simple example of the use of contexts, let us consider the case in which we are interested only in the value resulting from the invocation of a function foo, which on the contrary has a number of undesired side-effects, for instance it alters some free variables. Using aswitch we can apply foo to its arguments in a context different from that where the value is required, so that side-effects are ineffectual.

```
.............
aswitch [quote [foo], evlis [quote [arg1; ..... argn]] ;
        controlnth [2]; controlnth [2];
        newcontext [context[ ]]; context[ ]]
.............
```

As to the value everything works as if the invocation

foo [arg1; ... argn]

had been executed, but the application of foo occurs in the newly created context. Both FVL and RL point to the activation module calling aswitch, but in different contexts.

With regard to the implementation, two main points must be noticed:

(1) only one storage pool is actually used

(2) the activation modules are never duplicated.

Consequently some fields in activation modules should contain different values depending on the context in which the system is working; actually that is realized through an "indirect addressing": the field points to the list ("Redefined value list" : RVL) of all the different values corresponding to the different contexts. Every RVL is a list of pairs (v.C) where v denotes the value in the context C. For instance, fig.2 shows the situation in which the variable X has value v_o and v_1 in context C_o and C_1 respectively.
The RVL's are not in the storage pool, but in the free storage, as all the other lists on which the system operates.
When the function newcontext is executed no memory is allocated. Memory is allocated, on the contrary, when assignments in the new context are done. To assign a value to a variable in a context C the corresponding RVL is searched: if it contains a pair:

(v.C)

the value v is modified, otherwise a new pair is added.

This organization requires an efficient technique to search and update the RVL's; finally it is necessary to point out that the context mechanism results in an overload for the garbage collector [2].

4. Interaction and Debugging.

To aid program construction and debugging, LISPP is designed for use in interactive on-line fashion. Entering the system, the control is given to a function "EXECUTIVE", which is essentially an infinite loop, that reads an expression from the teletype, evaluates it and prints back the value. The evaluation is carried out by interpreting the expression.

The system provides a very simple but extremely flexible error-handling and debugging technique, which is similar to that used in BBN-Lisp [11] and ECL [12]. A code and a message are associated with each possible error: whenever an error occurs the system looks for a procedure associated with the error code; if such a procedure exists, it is activated, with FVL and RL pointing the activation module of the function where the error occurred. The continuation point of the suspended module is forced to the starting point of eval (i.e. of the interpreter): reentering the suspended module results in a new attempt to execute the procedure in which the error occurred, after that the function associated with the error performed its corrective actions. Obviously the error-handling function may re-enter not the suspended module, but any other module whose pointer it can retrieve, exploiting the generalized control structure of the system. Obviously the error-handling procedures are under the programmer's control, which can define and modify them, also during the execution.

If, on the contrary, no specific procedure is associated with the error code, the error message is printed; then the system assumes that the "EXECUTIVE" has to be activated: therefore the programmer is provided with the whole power of the system, in a "break" condition that temporarily suspends the execution of the program.

Then the programmer can either decide "a priori" the corrective actions to be performed in case of errors, or he can delay this decision until (and if) the error occurs.

This error-handling technique, together with the possibility of inserting break points in a program, provides a particularly convenient way to detect and fix up both logical and syntactic errors. A break-point is simply an explicit call of the EXECUTIVE: then, whenever the execution is suspended, either because of an error or because of a break-point, the programmer can define and execute corrrective actions, however complex - from the usual assignment of a variable, to the executions of programs written at break time, to the editing of the program itself - and finally he can resume the computation from the interruption point.

It is interesting to notice that neither a special debugging state nor a debugging language is needed, since the system itself provides a suitable frame for this task: this is due mainly to its interpretative organization, its generalized control structure and to the homogeneity of data and programs, which are indeed internally represented as lists, i.e. the basic data type the system deals with.

In testing programs that work on complex data structure, a program, that works incorrectly, may alter the system state in a hardly reversible way: typically an action, which was intended to repair an error, may, on the contrary, compromise any further corrective action. The problem has been already considered [9]: the proposed solution is based on an "undoing" function which is available in a special "test mode". In such a mode the system saves enough information to allow to undo the changes to the data structures occurring during the computation.

The context mechanism provides an alternative to the undoing technique in a straightforward manner. Indeed the context mechanism makes possible to experiment corrective actions in a different context from the one in which the error occurred. If the corrective action proves unsatisfactory, nothing is lost since the programmer can restore the initial situation and try a different action. Restoring the initial situation is extremely simple: it is enough to reenter the context in which the error occurred. In some sense this technique consists in "not doing" the changes until they prove satisfactory. In this case, it is enough to continue the computation in the context where the corrective action was successful.

Therefore, the generalized control structure and the context mechanism, which have been introduced to allow the programming of complex and unsual control regimes, seem to increase the interaction capabilities of the system, providing a very natural and flexible way of testing and debugging programs.

REFERENCES

1. Asirelli, P., Montangero, C. and Pacini, G. Implementazione del nucleo di un sistema LISP orientato verso la elaborazione di modelli semantici. Nota tecnica I.E.I. C72-5, Pisa (Novembre 1972).

2. Montangero, C. and Pacini, G. An extended LISP system for complex control-structure programming. Internal report B73-1, I.E.I., Pisa (January 1973).

3. Hewitt, C. Description and theoretical analysis (using schemata) of PLANNER: a language for proving theorems and manipulating models in a robot. Ph.D.Th., M.I.T. (February 1971).

4. Rulifson, J.F., Derksen, J.A. and Waldinger, R.J. A language for writing problem-solving programs. Proc. of IFIP Congress 71 Ljubljana (1971), pp. 201-205.

5. Davies, D.J.M., POPLER 1.5: A revised PLANNER-like system in POP2. Unpublished notes. School of Artificial Intelligence, Theoretical Psycology Unit, Edinburgh (June 1972).

6. Proceedings of 2nd Internat. Conf. on Artificial Intelligence, The British Computer Society, London, 1971.

7. Proceedings of an ACM Conference on proving assertions about programs, Las Cruces, New Mexico, (January 1972).

8. Aiello, M., Aiello Carlucci, L., Albano, A. and Montangero, C. Semantic concepts in Problem Solving, Planner languages and Question Answering. Monografia I.E.I., Pisa, 1973.

9. Bobrow, D.G. Requirements for Advanced Programming Systems for List Processing. Comm. ACM. 15, 7 (July 1972) pp. 618-627.

10. Elpas, B. and al. An assessment of Techniques for proving Program Correcteness, ACM Computing Surveys, Vol. 4, n.2 (June 1972), pp. 97-147.

11. Bobrow, D.G., Murphy, D.L., and Teitelman, W. BBN-Lisp reference manual. Bolt Beraneck and Newman, (April 1969).

12. Wegbreit, B. The ECL programming system. Proc. 1971 FJCC, Vol. 39, AFIPS Press; Montvale, N.J., pp. 253-262.

ACKNOWLEDGMENTS

We would like to acknowledge the members of the Non Numerical Information Processing Group at I.E.I., Pisa, for helpful discussions.

We are indebted with C.N.U.C.E., Pisa, for supporting us programming the system.

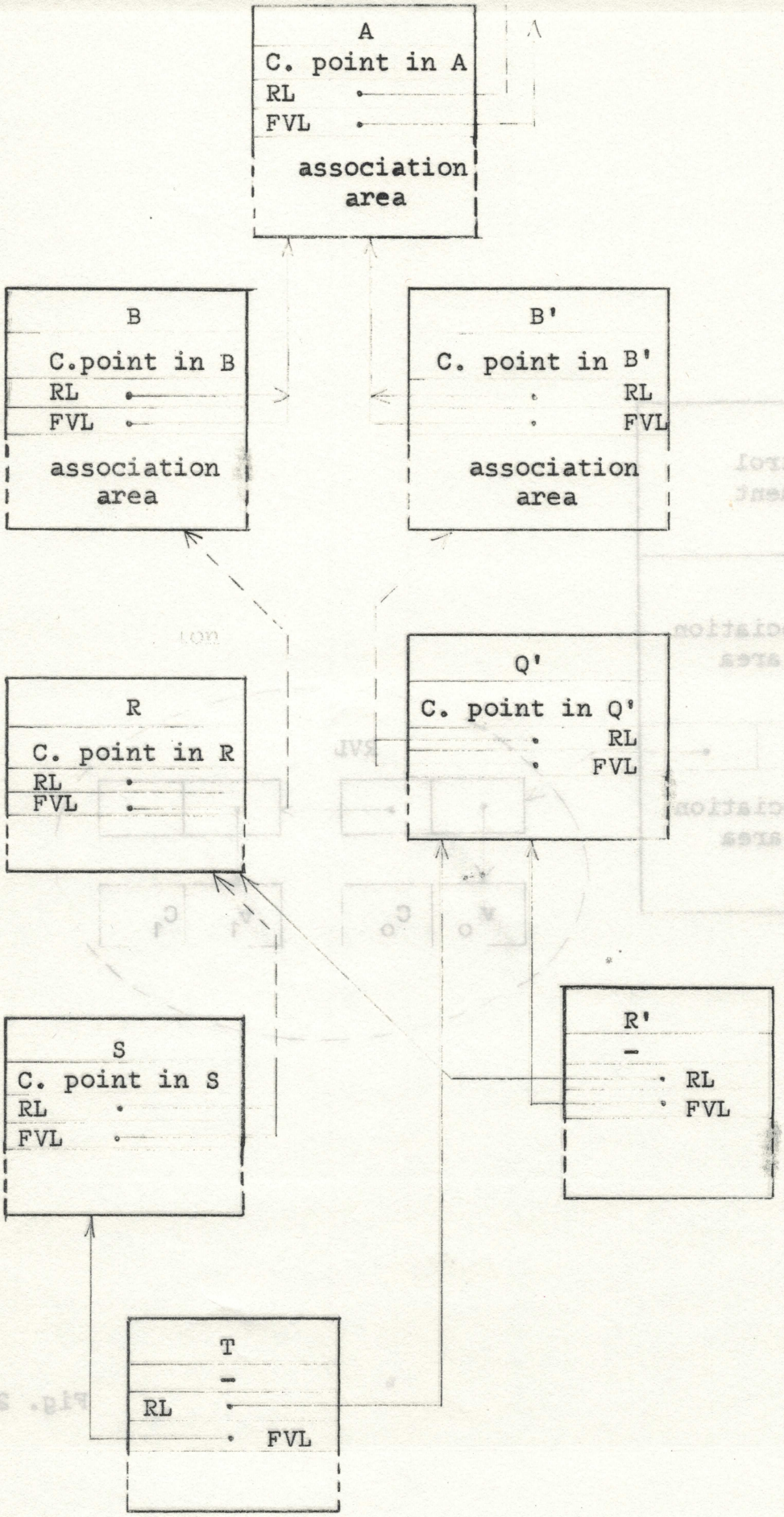
A
C. point in A
RL
FVL
association area
B
C.point in B
RL
FVL
association area
B'
C. point in B'
RL
FVL
association area
Q'
C. point in Q'
RL
FVL
R
C. point in R
RL
FVL
R'
RL
FVL
S
C. point in S
RL
FVL
T
RL
FVL

Fig. 1

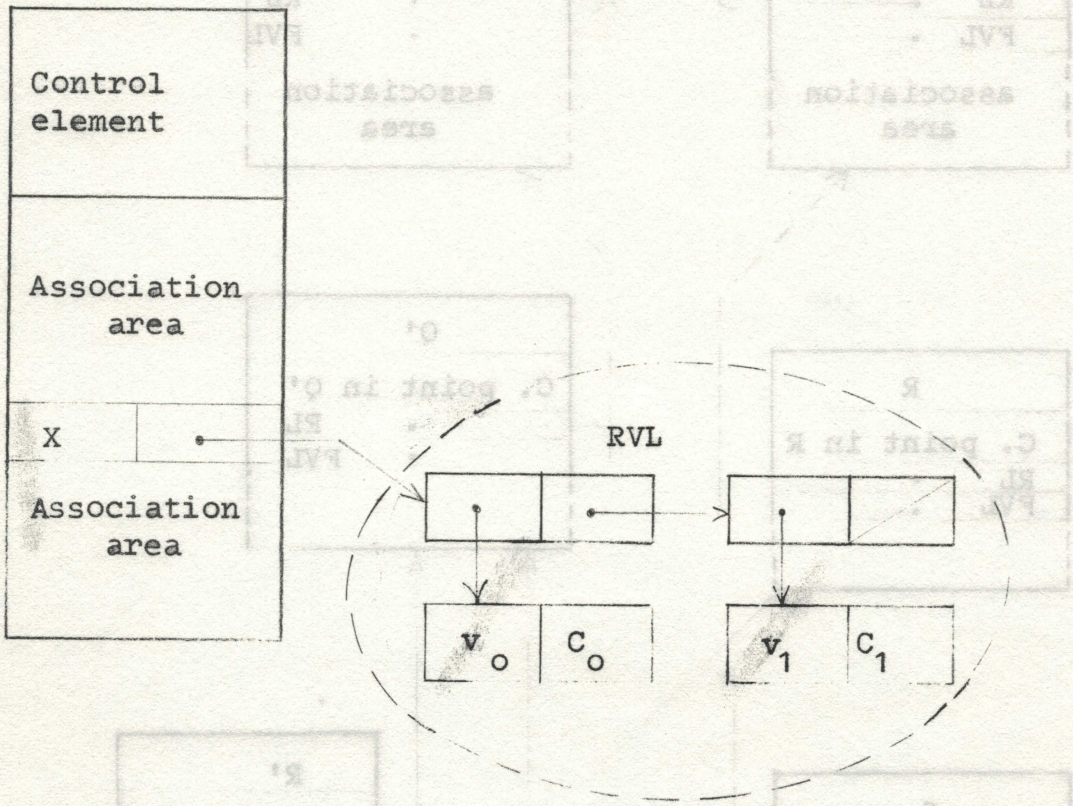
Control element
Association area
X
Association area
RVL
v_o
c_o
v_1
c_1

Fig. 2