AN EXTENDED LISP SYSTEM FOR COMPLEX CONTROL STRUCTURES PROGRAMMING

C. Montangero (+) G. Pacini (+)

Nota Interna B73-1

(Gennaio 1973)

Consiglio Nazionale delle Ricerche

ISTITUTO DI ELABORAZIONE DELLA INFORMAZIONE

P I S A

AN EXTENDED LISP SYSTEM FOR COMPLEX CONTROL STRUCTURES PROGRAMMING

C. Montangero[+] G. Pacini[+]

Nota Interna B73-1

(Cennaio 1973)

[+] Computer Science Department, University of PISA, PISA, and Istituto di Elaborazione dell'Informazione, del C.N.R. PISA.

Abstract.

The paper describes the way we intend to extend a LISP system, in order to supply it with a small number of elementary primitives allowing flexible programming of a large and graded range of complex control regimes. The extended system will be used for the development of problem-solving systems.

A model of the basic system is outlined first; a first extention permits to easily program control regimes such as coordinate cooperating processes. A further extention allows the retention of the computation state, so that non-deterministic programming is straightfoward. Finally a number of techniques to be exploited in the model implementation are outlined.

Key Words and Phrases.

Control-structures, programming primitives, computation-state retention, garbage collection.

1. Introduction

Recent research developments in various field of AI (as automatic theorem proving, plans generation for robots, question answering, automatic program synthesis and verification), have made clear that more powerful programming systems than traditional languages are highly desirable [4-9, 11, 12] .

One of the major limitations of languages such as ALGOL and LISP is that they operate in a strict last-in-first-out discipline. As a matter of fact there is plain evidence nowadays that more general control regimes (as backtracking, parallel processes, cooperating processes) are highly desirable in order to face that kind of problems. Indeed such control regimes provide natural ways to translate in programs the common-sense reasoning about those problem-domains.

Several languages, that supply one (or more) of the control regimes that we mentioned above, have been developed: we recall, among others, PLANNER [8] , QA4 [10] , POPLER [6] . Such languages (problem-solving languages) have been implemented either directly in machine code or in a list-processing language (typically LISP) or anyway in a language that does not provide facilities for the implementation of the control regimes that the problem-solving language is intended to supply: that is, the basic language does not provide any primitive which allows direct programming of the control regime.

The purpose of this paper is to describe the way we are extending our LISP system [1] in order to supply the language with a small number of primitives which will allow flexible programming of control regimes. Therefore we do not intend to develop a system with powerful primitives which realize specific control structures; we want instead to develop a system endowed with a few elementary primitives which allow a large and graded range of complex control regimes.

We think that the system which we describe will be highly suitable to support the construction of problem-solving systems, to be used for researches in the fields that we mentioned at the beginning.

In the next section we briefly outline a model of the basic system and of a first extention, that permits to program control regimes such as coordinate cooperating processes, that share their free variables. Section 3 extends the model, in order to allow the retention of the state of the computation; therefore non deterministic programming is possible. The last two sections outline a number of techniques that we think to use in order to implement the model. The system will provide also a simple interrupt mechanism (not described here), that allows easy programming of simulated parallel processes.

Although we make explicit reference to a LISP system, the model seems to be highly general and therefore easily adaptable to other languages.

2. A first extention

We introduce first a few, very general primitives which seem to be a good basis for a number of control structures; for instance, it is easy to program coordinate cooperating processes (coroutines), that transfer control to each other, and share free variables. However, the functions that we describe here are still limited in their ability to cope with control regimes which need to save the state of the computation: for instance non-deterministic programming in its generality. These limitations are removed in the next section.

We will refer hereafter to a LISP system, which is essentially an "eval" interpreter which evaluates forms, that is lists made up of a function definition and its arguments. The system employs an association list (A-list) in order to deal with variable

bindings, and a control list (C-list) to manage the control (*). Whenever the interpreter is recursively called, in order to evaluate a form F a new control element (CE) is appended to the control list; next association elements (AE) are appended to the A-list: they correspond to the bindings between the λ-variables of the function occurring in F and its arguments.

Each AE is made up of three fields: 1) variable name, 2) value, 3) pointer to the preceding AE in the A-list. Each control element is made up of several fields:

1) a pointer (CEP) to the preceding element in the C-list;

2) a pointer (ALP) to the A-list to be used in the evaluation of free variables; this pointer is updated in order to keep trace of the bindings of the function occurring in F;

3) a field used to store the continuation point (C-point) in the activation of the interpreter corresponding to the CE;

4) other informations which are of no interest here.

For each recursive activation of eval there is a corresponding continuation point in the calling activation. This C-point can be stored in the CE corresponding to the caller (say A) as well as in the CE of the called activation (say B), provided that the organization is merely recursive. It has been pointed out [3] that if the organization is more general, it is not convenient to store the C-point in the called activation, since the return into A will not necessarily occur from B. If, on the contrary, the C-point is stored in the CE of the caller, there is no problem: whatever activation which returns in A will be

---

(*) Actually the current version of the system has two push-down lists for these purposes. Nevertheless we think that the model, that we describe here, is more apt to support the extention that we are introducing.

able to continue correctly. Obviously, while the pointers ALP and CEP are set when the CE is allocated, the C-point is set only when a recursive call of the interpreter occurs. Then, when a form has been evaluated, the interpreter goes down one element in the C-list, following the CEP: interpretation will then continue according to the C-point.

Up to inessential details, standard LISP systems operate according to the previous schema: the pointers in the CE are usually set, at allocation time, so that the CEP points to the calling CE and ALP is equal to the ALP of the calling CE. The possibilities of programming the two pointers provided by LISP systems are limited, especially with regard to the CEP. Our aim is precisely to introduce some primitives that allow a flexible programmability of the two pointers. This requirement leads to a more complex structure of the C-list: actually in our extended model, while each CE points to only one other, every CE may be pointed by two (or more) others. Another consequence of control programmability is that there must be functions that evaluate to control element pointers and others that operate on such pointers: then the most general approach is to introduce data types for these pointers (*) on the same level of the other data types the system provides. As a matter of fact, we think that almost nothing is lost in generality if only one data type, namely for control element pointers (CEP), is introduced: indeed every CEP specifies a unique A-list, by means of its ALP.

The fundamental tool, that we introduce in order to allow to program the control regime, is an extention of the basic evaluator "eval": it is called eswitch, for eval switch (indeed it allows to depart from the standard LIFO discipline). Its format is the following one:

eswitch[fr; cep1; cep2].

(*) Indeed we suppose that A-list and C-list are not in the free-storage and are not homogeneous with the list structures contained there.

This function evaluates the form fr, using the A-list specified by the ALP of the control element pointed by cep1. When the evaluation is over, return occurs into the CE pointed by cep2. Actually this is done allocating a CE and setting its CEP to cep2 and its ALP equal to the ALP of cep1.

In order to make explicit references to CE's possible, we introduce the following functions:

controlset [fr]

which is an identity function that returnes the value of the form fr; it is introduced for its use in connection with the next function

controlnth [n]

which returns as a value the CEP to the CE of the n-th control-set which is found scanning the C-list. Finally we need a predicate which tests if its argument is a CEP:

cepp [arg]

evaluates to nil if arg is not a CEP, to arg itself otherwise.

The following example will use another primitive that seems pratically useful in many situations. It allows to push onto the A-list, new bindings: since it evaluates its argument it is suitable to push onto the A-list variables whose names are not known at programming time. More precisely the function is

pusha [x]

where x is a list of couples of the format ((var1 v1) (var2 v2)... ..(varn vn)). This function pushes on the A-list of its caller the bindings (var1.v1), (var2.v2) ... and so on.

We will program a coroutines control regime defining two functions: the first one (called coroutines) is able to activate the coroutine system; the second one (resume) to transfer control from one to another.

coroutines[((cor1 def1) ... (corm defm));
arg1; arg2; ... argn]

corl ... corm are the names of the coroutines, to be used in order to transfer the control from one to another; defl ... defm are the definitions and argl ... argn the arguments to be passed to the coroutine which will be activated first, namely corl.

resume [mess; cor]

evaluates the form mess and reactives the coroutine cor. The message (the value of form mess) is returned as the value of the last resume performed in the coroutine cor; if cor had not been yet activated, it is activated with the message as argument list. The coroutine system is exited as soon as a coroutine returns. In the following definitions the numbers at the beginning of the line refer to comments.

Example I (please skip underlined characters in the example for the moment, together with comments whose number is underlined. They will be explained in Sec.5).

```
   COROUTINES ≡
1.   (Nλ COROUTINES
2.      (PROG ((CACTIVE (CAAAR COROUTINES))
3.            (ARG(EVLIS(CDR COROUTINES))))
4.      (PUSH(CAR COROUTINES))
        (RETURN(RCONTROLSET
                (PROG()
5.                (SETQ COROUTINES(CONTROLNTH 1))
6.                (ESWITCH (CONS(EVAL CACTIVE)ARG)
                           COROUTINES COROUTINES))
13.             (QUOTE GLOBAL) ))))

   RESUME ≡
     (Nλ RESUME
7.      (RCONTROLSET
          (PROG ((MESSAGE(EVAL(CAR RESUME))))
8.          (SET CACTIVE(CONTROLNTH 1))
9.          (SETQ CACTIVE(CADR RESUME))
10.         (COND((CEPP(SETQ RESUME(EVAL CACTIVE)))
11.               (CONTROLSET
```

```
                    (ESWITCH(QUOTE MESSAGE)
                            (CONTROLNTH 1) RESUME)))
12.                 (T (ESWITCH (CONS RESUME MESSAGE)
                            COROUTINES COROUTINES))))
13.     (QUOTE GLOBAL) ))
```

Comments

1. According to the BBN-LISP standards[2], we assume that (1) missing arguments of a function are set to nil, (2) prog local variables can be initialized: e.g. (PROG((X FR)).... means that X is set to the value of FR; (3) Nλ X .... means that the unevaluated list of arguments is bound to X.
2. The variable cactive is used to store the name of the active coroutine. Initially is set to cor1.
3. The variable arg is set to the list of the evaluated arguments of cor1.
4. The bindings (cor1. def1)...(corm. defm) are pushed onto the A-list. When a coroutine cori has been activated, the variable cori is used to store the pointer to the CE to be reentered whenever a resume [m; cori] occurs.
5. The value of the variable coroutine is the pointer to the CE to which will return the coroutine that actually returns. This is achieved starting the evaluation of all coroutines using this pointer as first CEP.
6. Activation of the coroutine cor1. Notice that the arguments of eswitch are evaluated, like those of eval.
7. This controlset fixes the CE to reenter in order to resume the coroutine that is being abandoned.
8. This statement stores under the name of the coroutine to be abandoned the CEP needed to (eventually) resume it (see comment 4).
9. Updates the active coroutine indicator.
10. The value associated to the name of the coroutine to be resumed is stored in the variable resume, and then tested:

11. If it is a CEP the corresponding coroutine is reactivated.

12. Otherwise it is activated: note that in this case the value of resume is the definition of the coroutine.

13. The option global in the rcontrolset allows to release (i.e. to eliminate from the heading set) all CE's dependent from the rcontrolset CE.

Finally we note that the definition of the two functions coroutines and resume is such that it is possible, inside a coroutine, to activate again the whole system: i.e. it can be used recursively.

3. Contexts

The estention that we describe in Sec. 2 has many general features: as it is shown by Example I, it allows programming rather complex control regimes. Anyway it can be easily seen that it must be extended further. Let us suppose, for instance, that we are interested only in the value resulting from an evaluation, which on the contrary has a number of side-effects, for instance it alters free variables. Furthermore we suppose that choice among several possibilities has to be done (backtracking). In order to be able to follow another possibility, when a failure occurs, it is necessary to restore the state of computation at the moment of the first choice. This section will introduce a mechanism which allow to retain the state of the computation at any moment. In this way it is possible to deal straightforwardly with the foresaid problems.

We define a model in which the programmer operates as if it can ask for a copy of the situation that he intends to retain. Actually the system is able to simulate the creation of several copies of the system itself: each copy can then evolve indipendently.

In the following discussion we will refer to the schema of Fig. 1. Each block corresponds to a memory area:

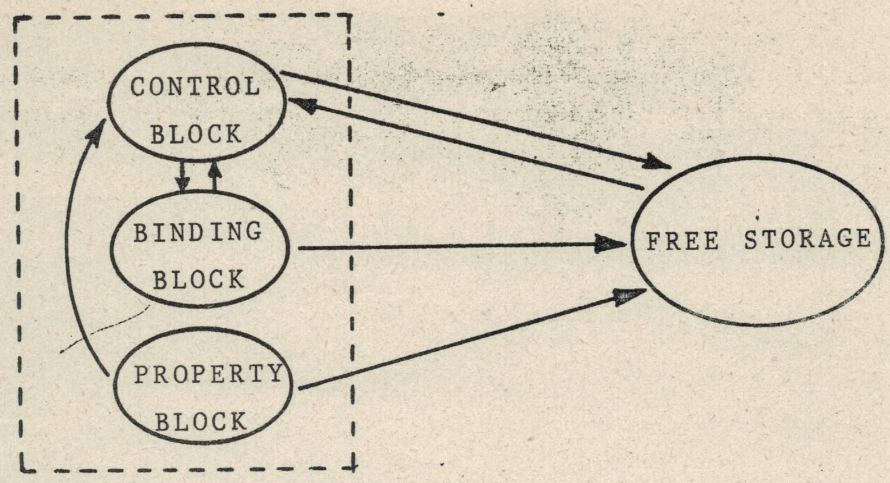


Fig. 3.1

- the control block contains the set of CE's;
- the binding block contains the set of AE's;
- the property block can be thought as a table whose entries are value of properties; it is accessed specifying a couple (atom property);
- the free storage contains all list-structures (obviously except A-lists and C-lists).

In Fig. 3.1 arrows specify the inter-block references that are allowed.

We call context (*) the control, binding and property blocks altogether; context state is the contents of the context.

Whenever the programmer wants to proceed in the computation in a context A, and to let himself the possibility to come back to a previous situation, he will be able to ask for the creation of a new context B, whose initial state is set equal to the state of the context A when B is requested.

The free storage is considered as a common pool where all values are kept indipendently from contexts. Therefore equal pointers in different contexts identify the same value. Alternatively the

(*) We use a term introduced by QA4 [10], since contexts, as they are defined here, work similarly to QA4's backtracking contexts.

free storage could be part of contexts: in this case equal pointers in different contexts might identify different values. This solution seems to offer few advantages and on the contrary introduces some difficulties in the implementation. As a matter of fact, the solution that we adopted requires only a careful use of the functions rplaca and rplacd, since they work as universal (in all contexts) modifiers.

We introduce two new primitives, and generalize the function eswitch, which will have five arguments, in order to deal with contexts. Namely, the function

context [ ]

returns the name of the context in which it is evaluated (as we did for control element pointers, we introduce a new data type for context names). The function

newcontext [cxt]

creates a new context whose state is set to the state of context cxt; it returns the name of the newly created context.

The general form of the function eswitch is:

eswitch [fr, cep1, cep2, cxt1, cxt2]

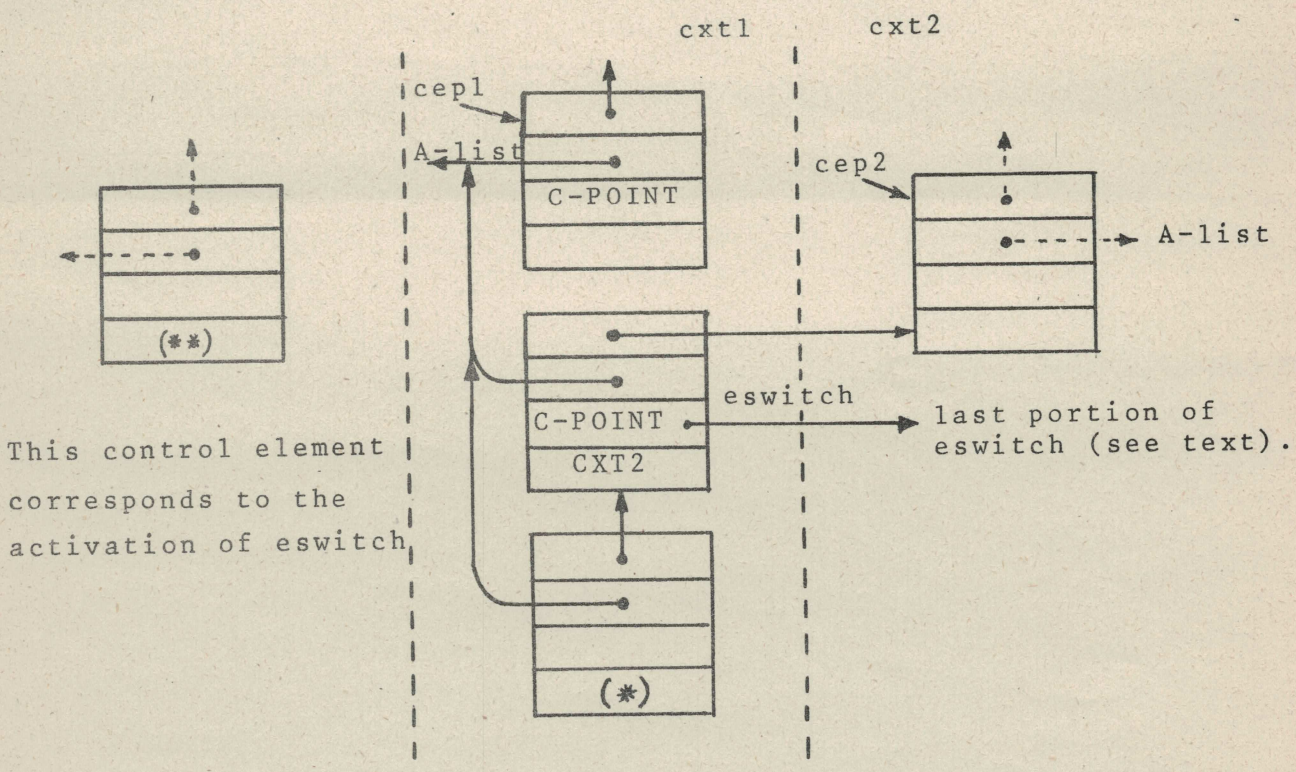
fr is evaluated in the context cxt1, using the A-list associated with cep1 (in cxt1) and returns the value to the CE cep2 in the context cxt2.

It is worthwhile noting that the introduction of contexts does not modify the structure of CE's, which do not need to contain any information about contexts - apart from the special case of eswitch -. It is only needed an active context indicator: the evaluator assumes that CEP's and ALP's point to elements in the active context. The function eswitch updates the active context indicator to cxt1, allocates a new CE in the same context, setting the CEP to cep2 and the ALP to that of the control element cep1 (obviously in cxt1 - see Fig.3.2). This newly allocated CE contains also the context name cxt2: the C-point is set so that reentering in the elements results in a jump to a piece of code (namely the last portion of eswitch) that updates again the active context indicator, to cxt2, and returns.

As a first example of the use of the primitives that manage contexts, let us consider the case in which the application of a function has undesired side-effects. We will define a function

value [fn; arg1; arg2; ... argn]

which applies fn to its arguments in a context which is different from the one where the value is required. This way side-effects are ineffectual.



(*) This CE corresponds to the activation of eval to evaluate the form fr.

(**) This CE corresponds to the activation of eval which evaluates the arguments and activates the code of eswitch: note that this may happen in a context different from both cxt1 and cxt2. This element will never be reentered.

Fig. 3.2

Example II (see remarks to example I)

```
VALUE ≡
  (Nλ  VALUE
1.     (RCONTROLSET
2.      (DESWITCH (CONS(CAR VALUE)
                       (EVLIS(CDR VALUE)))
                  (CONTROLNTH 1)          (CONTROLNTH 1)
3.                (NEWCONTEXT(CONTEXT))   (CONTEXT)
                  (QUOTE GLOBAL) )))
```

Comments

1. The CE of this controlset will be deleted from the heading set.
2. All contexts that may have been created during the application of fn are deleted from the living context list.
3. Recall that the arguments of eswitch are evaluated before the activation of the code of eswitch itself.

A simple use of the context mechanism requires - apart from the creation of new contexts and form evaluation in one context or another - that the transfer of informations between contexts can be done in a straightforward manner. A tipical case is the modification of the value of a variable in a context, from inside another. This operation is easily programmable with the primitives that we introduced so far. Nevertheless it is practically convenient to generalize the functions set, cset, put, get etc, so that the context (and the control element), where they operate, can be specified. For instance

set[var, v, cep, cxt]

set var to the value v in the A-list identified by cep in the context cxt (*); analogously

cset[var, v, cxt]

associates the value v to the property APVAL of the atom var in the context cxt (*). Put and get can be extended analogously.

(*) cep = nil and cxt = nil indicate respectively the present CE and the present context (if cxt ≠ nil also cep must be different from nil).

There are values (for instance a backtracking list which contains the choice points not yet exhausted) which are not related to any particular context but seem to be associated with the global state of the system. Then it is not convenient to force contexts to evolve indipendently, i.e. to impose that any change in a context never effects other contexts. Moreover global variables can be introduced in a natural way, that is based on the hierarchy of context generation. More precisely we say that a context x is a son of context y if and only if y has been created by the execution of newcontext [y]; the set of the descendant of y is defined recursively as it follow: y is a descendant of y; every son of a descendant of y is a descendant of y. In other words, contexts can be organized in a natural way in a tree (context tree) that reflects the hierarchy of generation: then the set of the descendants of x is the subtree whose radix is x itself. These concepts are used to generalize further the functions set, cset, etc, so that their operations depend on a new argument which specifies if the change has to be done in a single context or in all its descendants. For instance

set/setq [x; v; mod; cep; cxt]

work as defined above if mod = nil; if mod ≠ nil the assignment occurs in all the descendants of cxt; similarly

cset/csetq [x; v; mod; cxt]

will assigne v as the value of the property APVAL of x in all the descendants of cxt, whenever mod ≠ nil. Put and get can be extended analogously. Finally we assume that cxt = 0 denotes the radix of the context tree, so that universal variables can be easily dealt with.

Example III (see the remark about the previous examples).

This example deals with backtracking; we suppose that the choice point corresponds to a function whose unique argument is selected from a set. More precisely, the function

backarg [fn; arg1; arg2; ... argn]

applies fn to arg1; if a failure occurs, fn is applied to arg2 and so on. The context mechanism allows to save easily the context state in which the function must be applied.

```
BACKARG =        (N\ BACKARG
1.     (RCONTROLSET
         (PROG((ARG(CDR BACKARG)))
2.       A (RCONTROLSET
             (PROG(CXT)
3.            (CSETQ BACKLIST
                     (APPEND BACKLIST (CONS(CONTROLNTH 1)
                                           (CONTEXT)))

                     (QUOTE GLOBAL) 0)
                   (ESWITCH(CONS(CAR BACKARG) (EVAL(CAR ARG))
                     (CONTROLNTH 2)      (CONTROLNTH 2)
                     (SETQ CXT(NEWCONTEXT(CONTEXT)) CXT)))
4.         B (COND((NULL(SETQ ARG(CDR ARG))) (FAIL)))
           (GO A))))

5.   FAIL [mess] = (\ MESSAGE
       (CONTROLSET
        (PROG((BACKPOINT(CAR BACKLIST)))
6.       (CSETQ BACKLIST(CDR BACKLIST) (QUOTE GLOBAL) 0)
7.8.     (ESWITCHD (QUOTE MESSAGE)
                   (CONTROLNTH 1) (CAR BACKPOINT)
                   (CONTEXT)      (CDR BACKPOINT)))))
```

Comments

1. This controlset fixes the CE from which starts - each time in a new context - each application of fn to an argument.
2. This controlset fixes the CE to be reentered in case of failure.
3. A new element (indicating a new choice) is appended to the choice point list (backlist); backlist is a universal variable. In case of a failure (see the definition of the function fail) the re-entering in the controlset in statement A results in a jump to the statement B.
4. Backarg fails if the list of arguments is exhausted; otherwise a new attempt is done.
5. This is a possible definition of a failure function. It reenters in the first element of an associated fail list (backlist), returning mess as a value.

6. Updating the choice point list.

7. The CE pointed by the first choice point is reentered.

8. The context in which the failure occurred is deleted from the ACL.

4. Context implementation

There are two main points to be noted in context implementation: (1) only one control block, one binding block and one property block are actually used; (2) the elements in these three blocks are never duplicated. Consequently some fields of elements in these blocks should contain a different value, depending upon the active context; actually this is realized through an "indirect addressing": the field actually points to a list (redefined value list-RVL) which contains all the values in the different contexts. Namely the binding and property blocks may have pointers to RVL in correspondence to values of variables or properties; the control block may have pointers to RVL in correspondence to C-Points and similar informations. RVL are in the free storage and contain dotted pairs (V.C), where V denotes the value in the context C. We will call (V.C) the pair corresponding to C in the RVL. Fig. 4.1 shows the situation for AE which defines the value of x as V_o in context C_o and as V_1 in C_1.

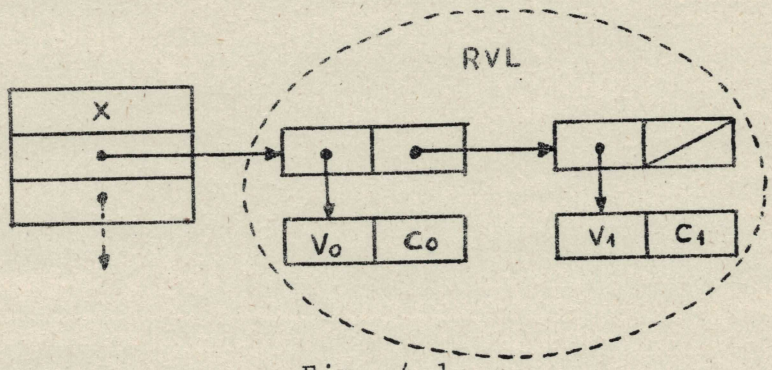


Fig. 4.1

When the function newcontext is executed no memory is allocated. It is allocated, on the contrary, when assignments in the new context are done. In order to assigne a value to x in the context C the RVL pointed by x is searched for the pair corresponding to C: if it is found it is modified, otherwise a new pair is added to the RVL.

It can be easily seen that, in order to look for a value in a context C, we must look for a pair corresponding to C itself or (if missing) to its nearest ancestor in the context tree.

The crucial point is the way RVL's are searched and updated: we describe an organization which seems simple and efficient enough, and requires a relatively small amount of memory. Each RVL has as many pairs as the number of contexts in which the corresponding information has been redefined. Let

$$((v_n \cdot cxt_n)\ (v_{n-1} \cdot cxt_{n-1}) \ \ldots\ (v_1 \cdot cxt_1))$$

be an RVL: the ordering of the sequence $cxt_1\ cxt_2 \ldots cxt_n$ corresponds to an examination in preorder of the context tree. For instance, let the context tree be that of Fig. 4.2: the RVL

$$((v_6 \cdot C_6)\ (v_5 \cdot C_5)\ (v_3 \cdot C_3)\ (v_o \cdot C_o))$$

corresponds to a value defined in C_o, and then redefined in the contexts C_3, C_5 and C_6.

The ordering and the search of RVL's are based on a context table, which associate each context C with two numbers, r(C) and s(C):

- r(C) is the number of nodes that preceed C in the examination in preorder of the context tree;
- s(C) is equal to r(C) plus the number of descendants of C minus 1. (recall that C is descendant of itself).

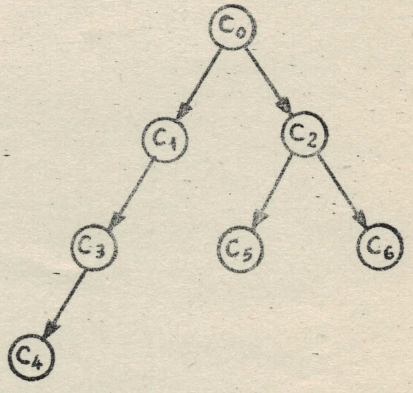


Fig. 4.2

The context table for the context tree of Fig. 4.2 is given in Fig. 4.3. The context table allows to detect easily if a context C_s is a descendant of C_2: indeed this happens only if

$$r(C_2) \leq r(C_1) \leq s(C_2). \qquad (1)$$

The following algorithm allows the search of a value in a context C:

1) scan the RVL until a pair $(v_1 \cdot C_1)$ such that

	r	s
C_o	0	6
C_1	1	3
C_2	4	6
C_3	2	3
C_4	3	3
C_5	5	5
C_6	6	6

Fig. 4.3

$r(C_1) \leq r(C)$ is found;

2) scan the RVL from $(v_1 \cdot C_1)$ (included) until a pair $(v_2 \cdot C_2)$ such that $s(C_2) \geq r(C)$.

This way the ancestor of C with greatest r is found, i.e. the nearest ancestor of C. Note that the complexity of the search in a RVL is a function of the size of the RVL, and not of that of the context tree. Indeed a number of tests, which is always less than the number of pairs of the RVL, is required. As a counterpart it is necessary to update the context table whenever a context is created (or deleted - see next section). It is interesting to note that this updating can be done simply scanning the table. When a context e (son of context f) is created, the following operations are needed: s(f):= s(f)+1; r(e), s(e):= s(f); increment by 1 all r's and s's greater or equal to r(e), except from r(e), s(e) and s(f). Similar rules are used when a context is deleted.

In order to be efficiently used the context table must be rapidly accessible; this can be achieved if context names are numbers:then the quantities r and s can be stored in an array indexed by context names.

In order to assign a value in a context C: scan the RVL until a pair, corresponding to C', is found, such that r(C') r(C). If r(C')=r(C), i.e. C'=C, the pair is consequently modified; otherwise insert a new pair (corresponding to C), just before the pair corresponding to C'. Moreover, in order to assure the correct behaviour, it may be that other pairs must be inserted or deleted (according to the assignment modality):

- if the assignment is global all pairs corresponding to descendants of C must be deleted from the RVL;
- otherwise, the pairs corresponding to the sons of C (with value equal to the value in C before of the updating) must be inserted in RVL whenever missing. In the first case the operation can be easily carried out using relation (1), in the other case the list of C's sons in needed.

5. Memory management and garbage collection

Because of its generality, the organization that we describe introduces heavy problems with respect to memory management: for instance garbage collection seems to be particularly involved and time consuming.

These difficulties are due essentially to the introduction of contexts; they can be partially overcome if the programmer is given the possibility to intervene on memory management. This choice is, in our opinion, quite coherent with the general philosophy of the system. Indeed a context is a unit whose retention depends on the program logic more than on the existance of a pointer pointing to it: then it is natural that the programmer has the ability to delete the contexts that he consider useless. Then some primitives that allow to abandon explicitely a context (or parts of it), are introduced. They do not require any knowledge of the actual implementation: nevertheless they allow an elegant control of the memory exploitment.

5.1 Context deletion

When a new context is created, it becomes alive and its name is appended to a list of alive context (ACL). The system assures the retention of all the information reachable in any alive context. In order to drop contexts from the ACL we introduce three functions:

delete [cxt1, cxt2,..., cxtn; mod]

drops from the ACL the contexts cxt1....cxtn (and all their descendants if mod ≠ nil);

deswitch [fr; cep1; cep2; cxt1; cxt2; mod]

works like eswitch; as a side-effect the active context at the end of the evaluation of fr (normally, but not necessarily, cxt1) is dropped from the ACL: this occurs immediately before reentering the CE cep2 in context cxt2;

eswitch [fr; cep1; cep2; cxt1; cxt2; mod]

also works like eswitch; before it starts the evaluation of fr in cxt1, the active context at the moment of activation of eswitch itself is dropped from the ACL.

The reader is kindly requested to reconsider the examples of Section 3, where these functions are used.

5.2 CE release

A set of CE's, called heading set (HS) is defined in each context. A CE is added to the HS when it is returned as a value of a controlnth.

The system guarantees the retention of all the CE's that belong to the C-lists headed by the elements of the HS (and of all A-lists pointed by ALP in the retained CE's). We define the following relation among CE's: CE1 depends on CE2 if and only if CE2 is in the C-list headed by CE1. Then this dependance relation is similar to the descendance relation for contexts. The following functions allow to delete CE's from the HS, then releasing them.

release [ce; mod]

deletes ce from the HS (if mod≠nil it deletes also all CE's depending on ce);

rcontrolset [fr; mod]

is like controlset; when it returns, the corresponding CE is deleted from the HS (mod as above);

reswitch fr, cep1, cep2, cxt1, cxt2, mod

is like eswitch; when fr has been evaluated it deletes from the HS of context cxt1 the CE cep1 (mod as above).

Examples in Sections 2 and 3 should be reconsidered in order to see how these functions are used.

5.3 Garbage collection

A sketch of the techniques that we will use for garbage collection (g.c.) is given in this section. The system has a partial g.c., limited to control and binding blocks, and a global g.c. that takes care of the free storage also. Partial g.c. is possible because of the existance of the alive context list and of heading sets.

Control and binding blocks are managed through free lists. When one of these is exausted, memory is collected by a g.c. with the following starting points: for the control block, the union of the HS's of alive contexts; for the binding block the set of the ALP's of the retained CE's. It is necessary to point out that the management of these blocks is not completely devoted to the partial g.c. It is often possible to reappend to the free lists CE and AE's returning from a CE: moreover the system works almost as an interpreter using a stack, as long as the primitives defined in this paper are not used.

The most natural technique for free storage g.c., traces and marks anything which is reachable from the active context (apart from reachable but no longer alive contexts). The main difficulty of this technique is that contexts should be traced when their names are found,

so that the g.c. must scan several times the control and binding blocks, and the RVL's.

We think to use a g.c. that marks anything that can be traced in the alive contexts, so that the control and binding blocks and RVL's can be swept once. This way, alive contexts which are not reachable from the active (at g.c. time) context, may be retained. However, at a little cost, the g.c. can build a list of alive unreachable contexts (AUCL). Such contexts will be dropped from the ACL: then the next g.c. will collect the memory they use. In order to build the AUCL, lists in the free storage that contain context names are dealt with in a special way: the paths that end in a alive-context name are traced every time, in order to allow to construct a context reference table. This table associates each alive context to the alive contexts that can be reached from it. The list of alive unreachable contexts (from the active context at g.c. time) can be easily derived from the examination of the context reference table.

References

1. Asirelli, P;, Montangero, C., Pacini, G., Implementazione del nucleo di un sistema LISP orientato verso la elaborazione di modelli semantici. Nota Tecnica I.E.I. C72-5, Pisa (November 1972).
2. Bobrow, D.G., Murphy, D.L., and Teitelman, W., BBN-Lisp reference manual. Bolt Beranek and Newman, April 1969.
3. Bobrow, D.G., and Wegbreit, B., A model and stack implementation of multiple environments. BBN Rep. No. 2334, March 1972.
4. Bobrow, D.G., Requirements for Advances Programming Systems for List Processing. Comm. ACM 15 (1972), 618-627.
5. Cheatham , T.E.,Jr., and Wegbreit, B., A laboratory for the study of Automatic Programming. Proc. AFIPS 1972 FJCC.
6. Davies,D.J.M., POPLER 1.5: A revised PLANNER-like system in POP2. Unpublished notes. School of Artificial Intelligence, Theoretical Psycology Unit Edinburg, June 1972.
7. Fischer,D., Control Structures for Programming Languages. Ph.D.Th. Carnegie Mellon University (1970).
8. Hewitt, C., Description and theoretical analysis (using schemata) of PLANNER: a language for proving theorems and manipulating models in robots.Ph.D.Th. MIT, Feb. 1971.
9. Rulifson, J.F., Dieksen, J.A.,and Waldinger, R.J., A language for writing problem solving programs. Proc. of IFIP Congress 71, Ljubljana (1971), pp. 201-205.

10. Rulifson, J.F., Derksen, J.A., and Waldinger, R.J., QA4: A procedural calculus for intuitive reasoning. Stanford Research Institute, Technical note 73 (November 1972).

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

12. Winograd, T., Procedures as a representation for data in a computer program for understanding natural languages. Ph.D. Thesis, MIT, 1970, Project MAC TR-84, MIT, Cambridge, Mass. Feb. 1971.

AN EXTENDED LISP SYSTEM FOR COMPLEX CONTROL STRUCTURE PROGRAMMING

C. Montangero and G. Pacini

Errata corrige

pag. 6

ERRATA

```
3.   (ARG (EVLIS(CDR COROUTINES) )))
```

CORRIGE

```
3.   (ARG (CDR COROUTINES)))
```

pag. 7

ERRATA

3. The variable arg is set to the list of the evaluated arguments of

CORRIGE

3. The variable arg is set to the list of the arguments of

pag. 12

ERRATA

```
2.   (DESWITCH (CONS(CAR VALUE)
                (EVLIS(CDR VALUE)))
```

CORRIGE

```
2.   (DESWITCH VALUE
```

pag. 14 (line 10)

ERRATA

```
     (ESWITCH (CONS (CAR BACKARG) (EVAL(CAR ARG))
```

CORRIGE

```
     (ESWITCH (LIST (CAR BACKARG) (CAR ARG) )
```

pag. 14 (line 7,8)

ERRATA

```
     (APPEND BACKLIST (CONS(CONTROLNTH 1)
                            (CONTEXT)))
```

CORRIGE

```
     (CONS (CONS (CONTROLNTH 1)(CONTEXT))
           BACKLIST)
```