[illegible]

1976

AN INTERPRETER FOR PREDICATE LOGIC PROGRAMS

PART I : Basic Principles

by M. BRUYNOOGHE

Report CW 10, October 1976

Applied Mathematics and Programming Division
KATHOLIEKE UNIVERSITEIT LEUVEN (BELGIUM)

AN INTERPRETER FOR PREDICATE LOGIC PROGRAMS

Part I : Basic Principles

by

Maurice BRUYNOOGHE
aspirant NFWO

Katholieke Universiteit Leuven
Afdeling Toegepaste Wiskunde en Programmatie
Celestijnenlaan 200B
3030 Heverlee
Belgium

TABLE OF CONTENTS

1. Introduction

The interpretation of predicate logic as a programming language is based upon the interpretation of implications B if A_1 and ... and A_n as procedure declarations where :

- B is the procedure name
- $A_1, \ldots, A_n$ is the set of procedure calls A_i constituting the procedure body.

For a procedure or clause we shall use the notation $B \leftarrow A_1, \ldots, A_n$. If it contains the variables $x_1, \ldots, x_k$ then it means : for all $x_1, \ldots, x_k$ B is implied by A_1 and ... and A_n. We call B, $A_1, \ldots, A_n$ the literals of the clause. We restrict ourself to Horn clauses, clauses with at most one literal as consequence.

Besides the usual form we have three special cases :

- an assertion of fact : $B \leftarrow$: a procedure with an empty body.
- a goal statement : $\leftarrow A_1, \ldots, A_n$: it asserts the goal of successfully executing all the procedure calls $A_1, \ldots, A_n$. It is a procedure without a name.
- the null clause : $\square$: a halt statement or satisfied goal statement, a nameless procedure with an empty body.

The reader can find a more elaborate description of the use of predicate logic as programming language in the work of Kowalski : [3], [4].

The goal of this paper is to describe the principles of an interpreter for this language, which intends to be an improvement of the existing PROLOG interpreter implemented at Marseille [1], [5].

2. Some features of PROLOG [1], [5]

2.a. The control tree

The execution of a PROLOG program can be controlled by a tree. We introduce this control tree by a simple example

The program :

```
(1) ← P,Q            (5) Q ← S
(2) P ← R,S          (6) R ←
(3) Q ← R            (7) S ←
(4) Q ← E
```

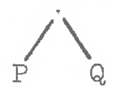


Fig. 1.a.
We start with the goal statement (1). The root of the control tree has a son for each literal in the goal statement.

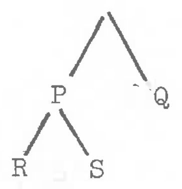


Fig. 1.b.
The leftmost call is executed first.

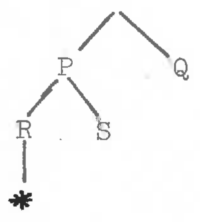


Fig. 1.c.
Procedure R (6) has an empty body, we get a terminal node indicated by a *

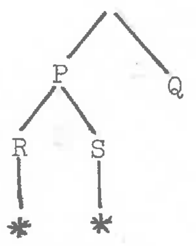


Fig. 1.d.
PROLOG uses a depth first, left to right search strategy, procedure S is selected and also has an empty body.

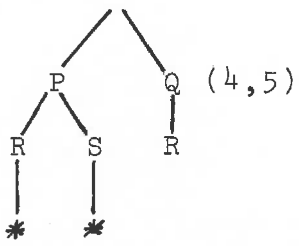


Fig. 1.e.
For executing Q, there are three possible procedures, PROLOG takes the first (3) and notes the others. This is a non-deterministic point.

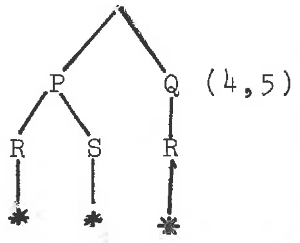


Fig. 1.f.
All goals are successfully executed, a solution is found, now the system backtracks to find other solutions.

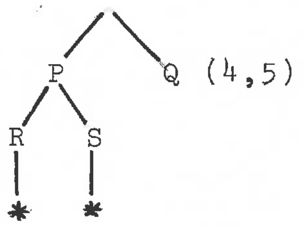


Fig. 1.g.
The control tree is unwond until the last nondeterministic point.

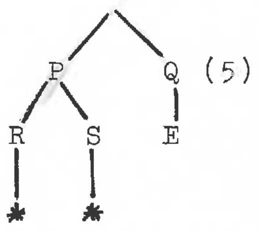


Fig. 1.h.
The system tries another procedure (4) to solve Q.

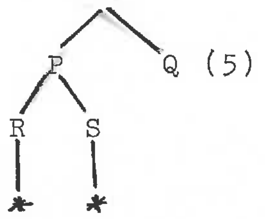


Fig. 1.i.
The system fails to solve E and backtracks again

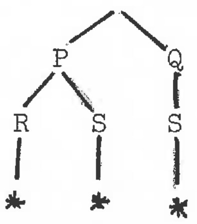


Fig. 1.j.
Using (5) and (7) the system finds a second solution. There are no more non-deterministic points, the execution is terminated.

Remarks

1. PROLOG uses a depth first, left to right search strategy, this means that :
   a. The order of the literals in the clauses is important, it defines the order in which the different calls are executed.
   b. The order of the clauses is important, it defines the order in which the different methods to execute a call are tried.
2. The different methods to execute a call are handled by a backtracking mechanism.

2.b. Variables

Clauses are stored as skeletons in a dictionary. In the control tree, the nodes are pointing to the skeleton of the literal. These literals can contain variables. To represent their value, we can associate with each node an environment describing the variables of the procedure used to execute that call. PROLOG uses a kind of structure sharing [2]: the value of a variable is represented by two pointers, one to a skeleton (a term) in the dictionary and the other to an environment giving the values of the variables occurring in that skeleton.

We give an example :

Program :

```
(1) Append (nil, z, z)←
(2) Append (cons(x,y),z,cons(x,u))← Append (y,z,u)
(3) ← Append (cons(a,nil),cons(b,nil),x)
```

For clarity we write the whole skeleton in place of a pointer, we also give the heading of the procedures used. The control tree is given in Fig. 2.

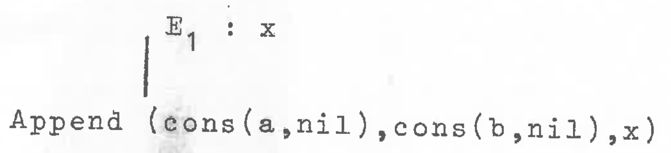
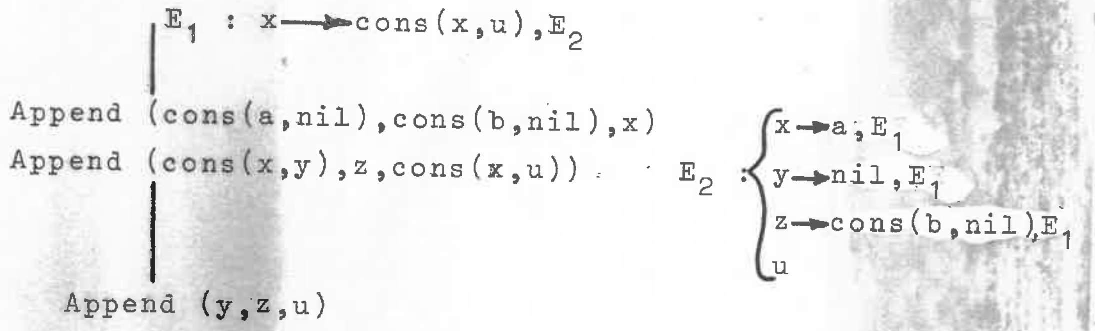


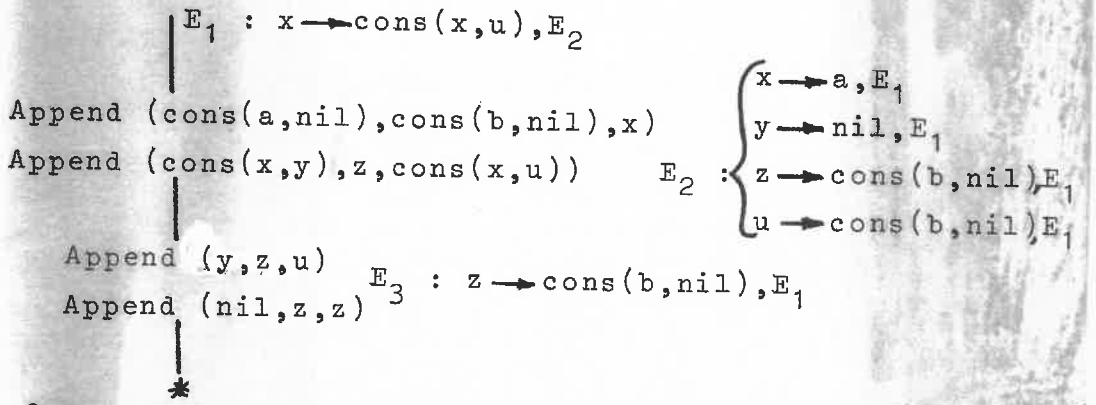
Fig. 2.a. The goal statement has one variable. The value is given in the environment E_1 of the root node. The variable has no value, it is a free variable.



E_1 : $x \rightarrow$ cons(x,u),E_2
Append (cons(a,nil),cons(b,nil),x)
Append (cons(x,y),z,cons(x,u))
E_2 :
$x \rightarrow a, E_1$
$y \rightarrow nil, E_1$
$z \rightarrow$ cons(b,nil),E_1
u
Append (y,z,u)

Fig. 2.b. Procedure invocation is controlled by a <u>pattern matching</u> mechanism (<u>unification</u>) :

(1) does not match, (2) does match with the call. The matching process binds the variables x, y and z of E_2 and also x of the environment E_1. The variable u in E_2 is free.



E_1 : $x \rightarrow$ cons(x,u),E_2
Append (cons(a,nil),cons(b,nil),x)
Append (cons(x,y),z,cons(x,u))
E_2 :
$x \rightarrow a, E_1$
$y \rightarrow nil, E_1$
$z \rightarrow$ cons(b,nil),E_1
$u \rightarrow$ cons(b,nil),E_1
Append (y,z,u)
Append (nil,z,z)
E_3 : $z \rightarrow$ cons(b,nil),E_1
*

Fig. 2.c. Matching is only possible with (1). The final value of x in E_1 is cons(a,cons(b,nil)).

Note that now backtracking is not as simple as unwinding the control tree. Suppose we want to backtrack from the situation in fig. 2.c. into the situation in fig. 2.a. then we have not only to unwind the tree but also to undo the binding of x in the environment E_1.

The main drawback of the PROLOG-interpreter of [1] is that it can only free memory by backtracking. Even if a call is successfully executed in a deterministic way then the interpreter cannot free the memory occupied by that part of the control tree because it is possible that variables are pointing to that part of the control tree. An example is given in fig.3.

Program :

$\leftarrow P(x), Q(x)$
$P(f(x)) \leftarrow R(x)$
$R(f(a)) \leftarrow$

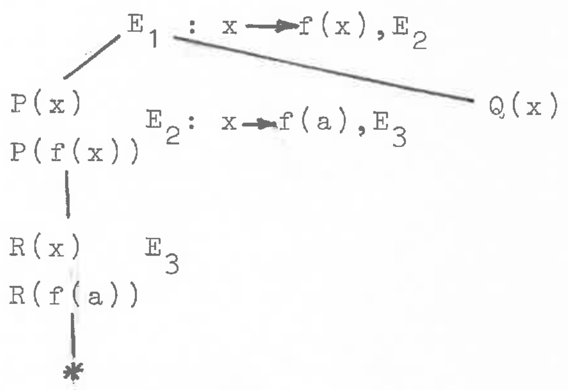


Fig. 3. We cannot free the space occupied by the left subtree of the root because the variable x in E_1 is pointing to the environment E_2 in this left subtree of the control tree.

The basic goal of our interpreter is to free the memory occupied by the parts of the control tree which describe calls completely executed in a deterministic way. For example the left part of the control tree in fig. 3.

2.c. The slash

This is a control feature that distinguishes PROLOG from pure predicate logic. It allows the user to delete all nondeterministic points in a subtree of the control tree.

In our implementation, it will also free the space occupied by that subtree. The interpreter handles the slash as a built-in predicate : the interpreter considers the predicate as defined by an assertion but it also performs some actions (side effects) which cannot be undone by backtracking.
We illustrate its effect on the control tree by an example in fig. 4.

The program :

```
(1)  ← P,Q
(2) P ← R,S
(3) R ← T
(4) R ← S
(5) S ←
(6) S ← T
(7) Q ← R,S,/,U
(8) Q ← U,V
(9) T ←
(10) T ← V
(11) U ←
```

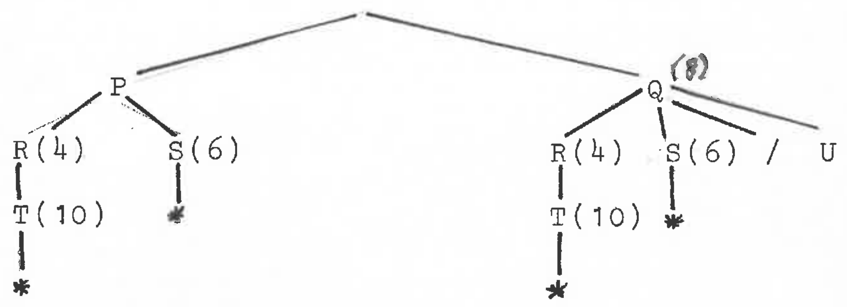


Fig. 4.a. The next call to execute is the slash. Its effect is to delete all nondeterministic points in the subtree with the parent of the slash as root.

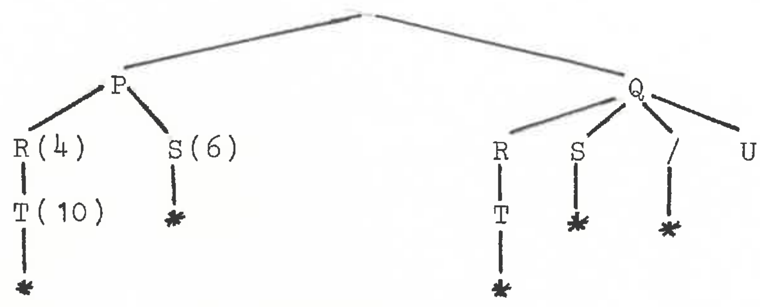


Fig. 4.b. The control tree after the execution of the slash. The next call to execute is U.

2.d. Summary of the features

- Procedure invocation controlled by a pattern matching process (unification).
- Recursivity.
- Backtracking.
- Control features which distinguish PROLOG from pure predicate logic:
  . order of the literals.
  . order of the clauses.
  . the slash.

3. A space-saving implementation

As stated in the previous section, our goal is to free the storage in the control tree occupied by the description of a deterministically executed procedure call. This is not possible when variables are pointing to other environments in the control tree as in fig. 3. We solve the problem by creating the value of the variables in a heap. We demonstrate this by the following program :

```
(1) Append (nil,z,z)←
(2) Append (cons(x,y),z,cons(x,u))← Append (y,z,u)
(3) ← Append (cons(a,nil),cons(b,nil),x)
```

The consecutive control trees are given in fig. 5.

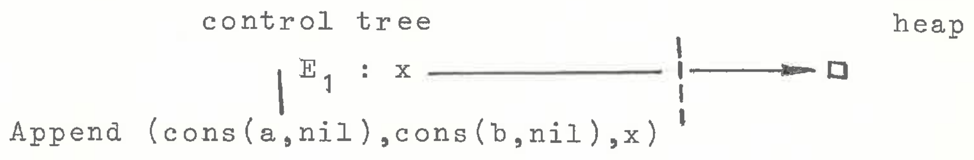


Fig. 5.a. The free variable x is pointing to an empty cell in the heap.

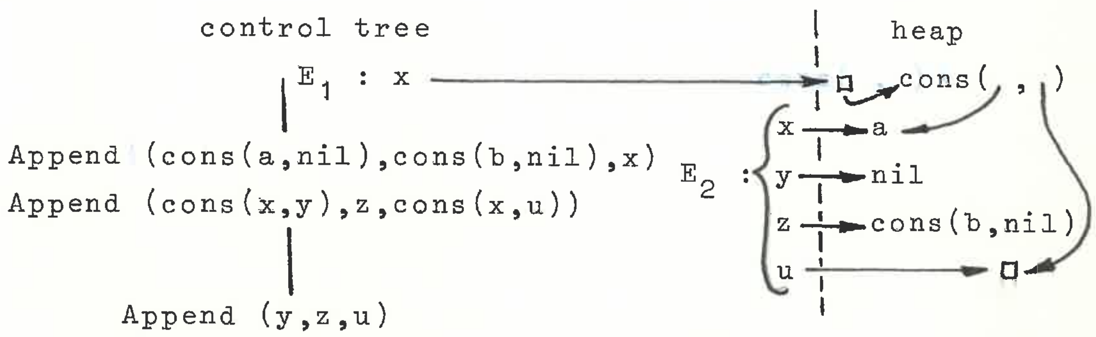


Fig. 5.b. Unifying the first two arguments causes the creation of a,nil,cons(b,nil) in the heap. Unifying the last arguments causes the empty cell representing x in E_1 to be bound to a created structure cons(x,u) where x and u are replaced by their values in E_2.

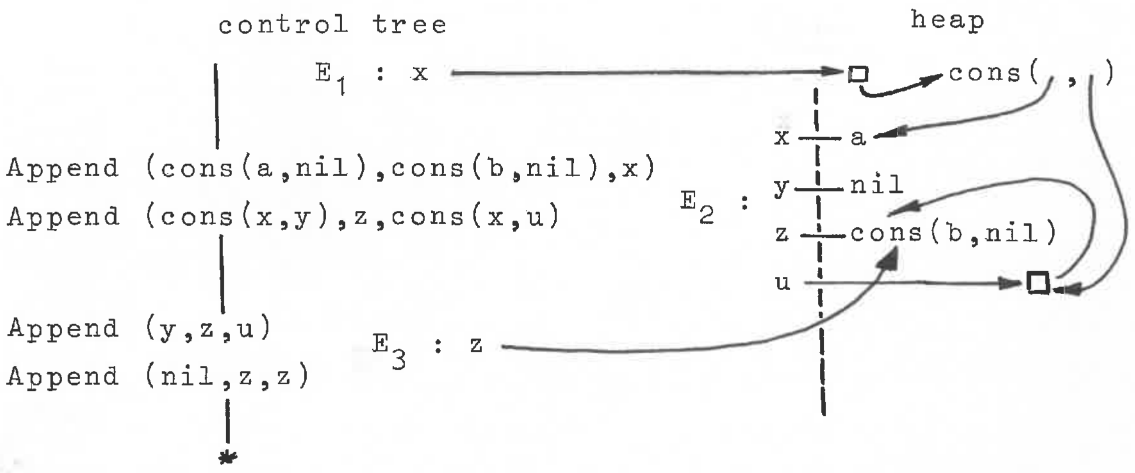


Fig. 5.c. The variables z in E_3 and u in E_2 are both bound to the value of z in E_2. The final value of x in E_1 is cons(a,cons(b,nil)). We can delete all except the root node of the control tree and still have access to this value.

We demonstrate the recuperation of a deterministically terminated subtree in fig. 6.

Program

```
 ←P(x),Q(x)
P(f(x))←R(x)
R(f(a))←
```

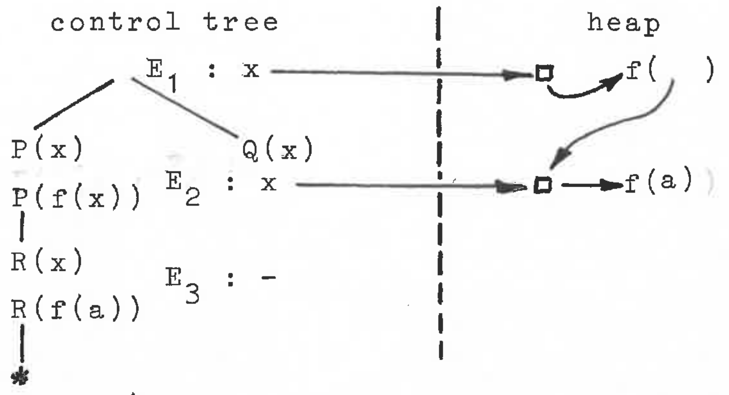


Fig. 6.a. The left subtree is terminated, there are no non-deterministic points in it.

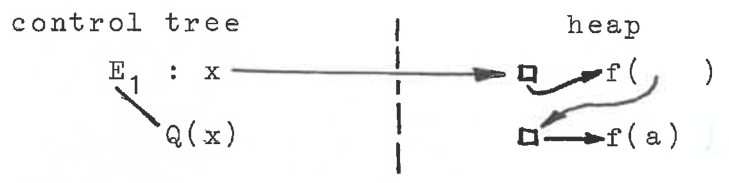


Fig. 6.b. The left subtree is deleted.

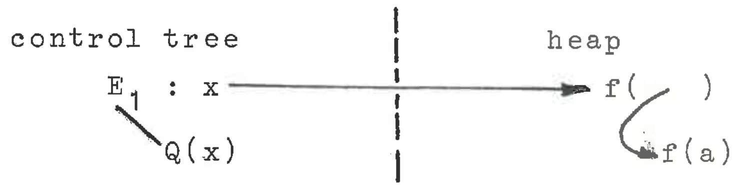


Fig. 6.c. A garbage collector can recuperate parts of the heap and can simplify the access to the structures.

Deleting some parts in the control tree causes some structures in the heap to become inaccessible, they can be recuperated by a garbage collector. The garbage collector can also simplify the access to the structures in the heap. This is shown in fig. 6.c.

Backtracking consists of unwinding the control tree and the assignments in the heap. By assignment we mean the binding of a free variable, represented by an empty cell, to a value. Unwinding the control tree causes some parts of the heap to become inaccessible. When we also unwind all creations in the heap (empty cells and structures) then we easily recuperate these unaccessible parts of the heap and we need the garbage collector less frequently.

Note that at this level of description, we can select any unsolved call in the control tree. It is only in the next section, when we will choose a particular representation of the control tree that we definitely select the depth first, left-to-right search strategy.

4. Representation of the control tree

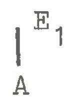
4.a. Deterministic programs

When we use a depth first, left-to-right search strategy, then we can look at the body of a procedure as a sequence of statements, every statement being a procedure call. When we restrict ourselves to deterministic programs then we get a simple Algol-like language. The control tree can easily be implemented as a stack. Every environment on the stack has, in addition to the variables of the procedure, a pointer to the environment of the calling procedure (E) and a pointer to the skeleton of the next call in the calling procedure (SK). The state of the execution is determined by a current environment CE and a current skeleton CSK. As an example, we take the following program :

```
 ← A
A ← B,C
B ←
C ←
```

The different control trees and stacks for the execution of this program are given in fig. 7.

control tree



control stack

$$\boxed{\begin{array}{lll} E_1 : & E & : \text{nil} \\ & SK & : \text{nil} \end{array}}$$

CE : E_1
CSK : A

Fig. 7.a. Initial state.

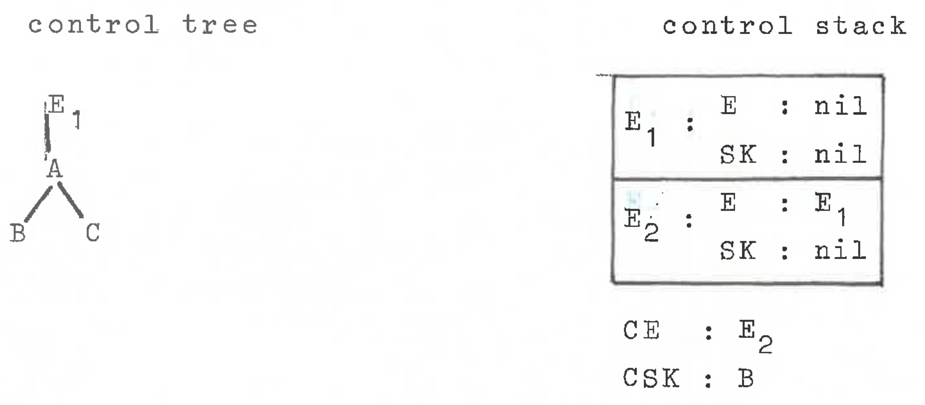


Fig. 7.b. We enter the body of procedure A. SK in E_2 is nil because A is the last call in E_1.

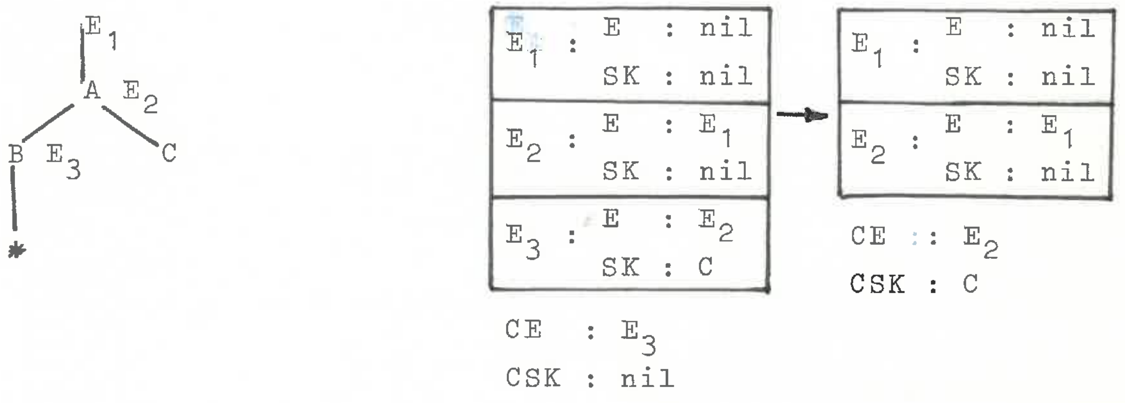


Fig. 7.c. CSK is nil thus in the stack we return to the cal-ling environment E_2.

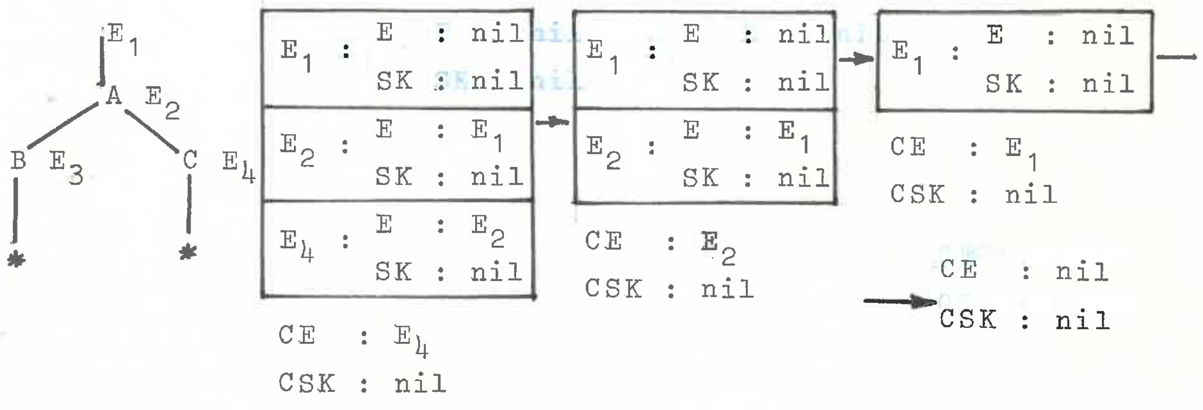
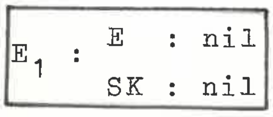


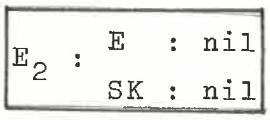
Fig. 7.d. We have a CSK = nil and we terminate with an empty stack.

In fig. 7.d. we see, we do not need the environments E_1 and E_2 because all calls in these environments are executed. We do not need an environment any more when the last call is activated. This gives us a more compact stack where every environment is pointing to the call (and the corresponding environment) we will execute after terminating the current call. The different states of this more compact stack are given in fig. 8.



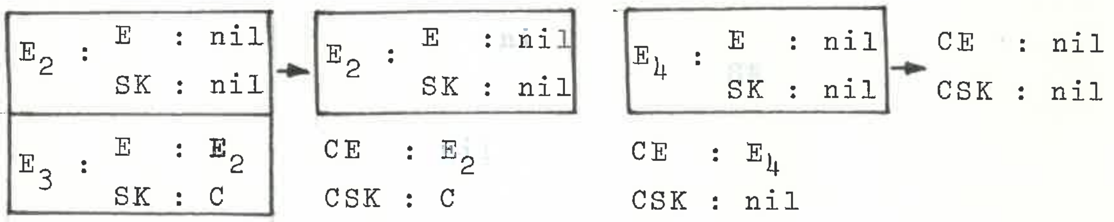
CE : E_1
CSK : A

Fig. 8.a. Initial state.



CE : E_2
CSK : B

Fig. 8.b. The environment E_1 is deleted because A is the last call.



CE : E_3
CSK : nil

CE : E_2
CSK : C

CE : E_4
CSK : nil

Fig. 8.c. We return in the stack when CSK is nil.

Fig. 8.d. C was the last call in E_2 thus E_3 is deleted. CSK is nil and we terminate with the empty stack.

The stack combined with the definitions of the procedures in the dictionary gives us a list of all calls we still have to execute together with the binding of the variables in these calls. The very simple structure of the stack is due to the fact that we always select the first element of that list (and replace it by its body); thus due to the depth first, left-to-right search strategy.

A more elaborate search strategy would be able to select any call in that list. This would ask for a much more complex control structure.

For the control stack we can notice that

- the current environment is always the top-element of the stack.
- each element is pointing (E-pointer) to the previous element of the stack.

This is no longer the case when we extend to nondeterministic programs.

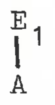
4.b. Nondeterministic programs and backtracking

Backtracking is returning to a past state in the control stack; it means we have to memorize all environments of that state. We reach this goal by not deleting them when the execution of the last call of a procedure is started.

When we have different procedures to match the current call (a nondeterministic point) then we create a backtrackingelement on the top of the stack. In that element we put information about the nondeterministic call : the skeleton, the environment, the next procedure to try. Then we start the execution of the call with the condition that we must not delete any environment of the current stack. This condition can be stated as a very simple rule : do not delete an environment if it is not the top element of the stack.

We illustrate this by an example in fig. 9.

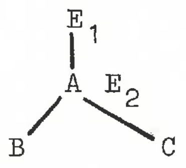
Program : (1) $\leftarrow$A
(2) A$\leftarrow$B,C
(3) A$\leftarrow$C
(4) B$\leftarrow$
(5) B$\leftarrow$C
(6) C$\leftarrow$D
(7) C$\leftarrow$



E_1 :	E	: nil
	SK	: nil

CE : E_1
CSK : A

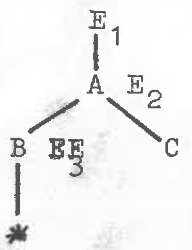
Fig. 9.a. Initial state.



E_1 :	E	: nil
	SK	: nil
B_1 :	E	: E_1
	SK	: A
	PR	: (3)
E_2	E	: nil
	SK	: nil

CE : E_2
CSK : B

Fig. 9.b. There are different procedures to execute the call A, thus we first create a backtrackelement pointing to environment E_1, skeleton A and procedure (3). Then we execute A using (2). Although A is the last call, we do not delete E_1 because it is not on the top of the stack.



E_1 :	E	:	nil
	SK	:	nil
	E	:	E_1
B_1 :	SK	:	A
	PR	:	(3)
E_2 :	E	:	nil
	SK	:	nil
	E	:	E_2
B_2	SK	:	B
	PR	:	(5)
E_3	E	:	E_2
	SK	:	C

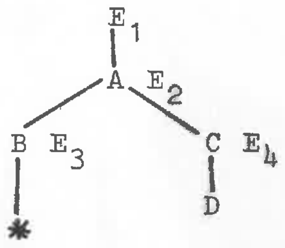
CE : E_3
CSK : nil

$\longrightarrow$

E_1 :	E	:	nil
	SK	:	nil
	E	::	E_1
B_1 :	SK	:	A
	PR	:	(3)
E_2 :	E	:	nil
	SK	:	nil
	E	:	E_2
B_2 :	SK	:	B
	PR	:	(5)

CE : E_2
CSK : C

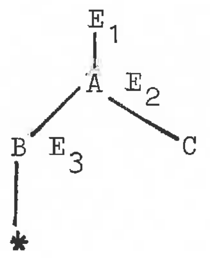
Fig. 9.c. A second backtrackelement is created, E_3 is deleted. Note that the current environment is not on the top of the stack.



E_1 :	E	;	nil
	SK	:	nil
	E	:	E_1
B_1 :	SK	:	A
	PR	:	(3)
E_2 :	E	:	nil
	SK	:	nil
	E	:	E_2
B_2 :	SK	:	B
	PR	:	(5)
	E	:	E_2
B_3 :	SK	:	C
	PR	:	(7)
E_4 :	E	:	nil
	SK	:	nil

CE : E_4
CSK : D

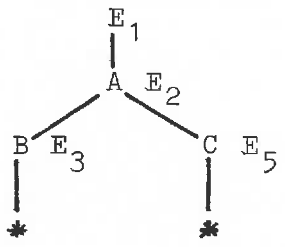
Fig. 9.d. Also the call on C is nondeterministic. Now we fail to execute D and we backtrack.: we unwind the tree to the last backtrackelement. This element tells us to execute C in E_2 using (7).



E_1 :	E	:	nil
	SK	:	nil
	E	:	E_1
B_1 :	SK	:	A
	PR	:	(3)
E_2 :	E	:	nil
	SK	:	nil
	E	:	E_2
B_2 :	SK	:	B
	PR	:	(5)

CE : E_2
CSK : C

Fig. 9.e. The current call is C in E_2, we have to execute it using (7). Note that we have the same situation as in fig. 9.c.



E_1 :	E	:	nil
	SK	:	nil
	E	:	E_1
B_1 :	SK	:	A
	PR	:	(3)
E_2 :	E	:	nil
	SK	:	nil
	E	:	E_2
B_2 :	SK	:	B
	PR	:	(5)
E_5	E	:	nil
	SK	:	nil

CE : E_5
CSK : nil

$\longrightarrow$

E_1 :	E	:	nil
	SK	:	nil
	E	:	E_1
B_1 :	SK	:	A
	PR	:	(3)
E_2	E	:	nil
	SK	:	nil
	E	:	E_2
B_2 :	SK	:	B
	PR	:	(5)

CE : nil
CSK : nil

Fig. 9.f. CE = nil indicates that a solution is found, the stack is not empty and we can backtrack to find other solutions.

As in the deterministic case, the chain of environment elements, starting with the current environment, describes the list of calls we still have to execute. The elements of this chain are the active environments. Contrary to the deterministic case we now also have elements which are not in this chain. We call them passive environments. Their deletion is prevented by the backtrackingelements. Starting from each backtrackingelement we can also find a list of calls. It is the list of calls we had to execute when we arrived at the corresponding nondeterministic point. Backtracking is just reactivating such a chain.

Remember that the environmentelements also include information about the variables of the procedures and the backtrackelements also include information about the unwinding of the heap.

5. <u>The heap</u>

As we can see in fig. 5 and 6, we can distinguish three sorts of elements in the heap :

1. constants and function symbols : a, b, nil, cons. We can put them in a dictionary, together with other information (for instance their number of arguments, ...).
2. structures : cons(a,nil). These will be elements of the heap. We can, for example, choose elements with two fields. A possible representation is given in fig. 10.

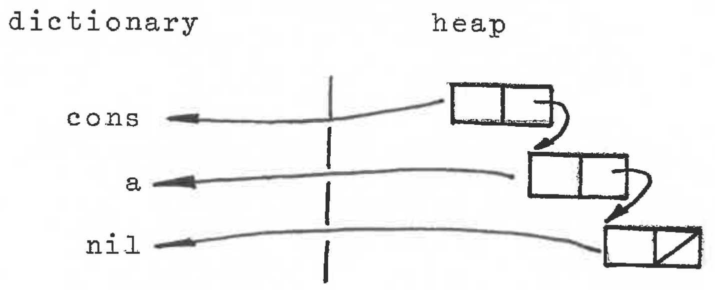


Fig. 10. Representation of cons(a,nil)

3. variables who are initially free, indicated by ◘ in fig. 5 and 6. They will be elements with one field : a pointer to their future value.

The variable elements have two important dates in their lives :

- the creation date : when the empty element is created.
- the assignment date : when the variable takes a value.

For the structure elements the assignment happens at the same time as the creation.

This assignment date is very important for the backtracking. Backtracking is returning to a past state, this means that all assignments made since that moment must be deleted. This can be done by putting all assigned variable elements on a list. During backtracking we unwind the as-signment

During backtracking we unwind this assignmentlist to a point noticed in the backtracking element.

By unwinding the stack we lost for garbage (thus delete) alle created variable and structure elements during the back-tracking. By putting them also on a list, we can easily re-cuperate this memory and we need the garbage collector less frequently.

Acknowledgements

Thanks are due to Philip Roussel and Alain Colmerauer for their help in studying the original PROLOG interpreter; to Yves Willems and Marc Gobin for reading and discussing the paper.

This work was done during my mandate as "aspirant bij het Nationaal Fonds voor Wetenschappelijk Onderzoek". A grant from the French "Centre National de la Recherche Scientifique" in coordination with the Belgian "Ministerie van Nationale Opvoeding en Nederlandse Cultuur, Bestuur voor Internationale Culturele Betrekkingen" allowed me to visit for three months the University of Aix-Marseille.

Bibliography

1. BATTANI, G. and MELONI, H., Interpreteur du langage de programmation PROLOG. Groupe d'Intelligence Artificielle, U.E.R. de Luminy, Université d'Aix-Marseille, 1973.

2. BOYER, R.S. and MOORE, J.S., The sharing of structure in theorem-proving programs. In Machine Intelligence 7, B. Meltzer and D. Michie, Eds., Edinburgh, U. Press, Edinburgh, Scotland, 1972, pp. 101-116.

3. KOWALSKI, R., Predicate logic as programming language. In Information Processing 74, North-Holland Publishing Company, 1974, pp. 569-574.

4. KOWALSKI, R., Logic for problem solving. Memo No 74, Department of Computational Logic, School of Artificial Intelligence, University of Edinburgh, March 1974.

5. ROUSSEL, P., PROLOG Manuel d'utilisation. Groupe d'Intelligence Artificielle, U.E.R. de Luminy, Université d'Aix-Marseille, septembre 1975.