ICOT Technical Report: TR-410

TR-410

# Macro Processing in Prolog

by
S. Kondoh (Mitsubishi) and T. Chikayama

July, 1988

ICOT

Mita Kokusai Bldg. 21F
4-28 Mita 1-Chome
Minato-ku Tokyo 108 Japan

(03) 456-3191~5
Telex ICOT J32964

Institute for New Generation Computer Technology

# MACRO PROCESSING IN PROLOG

SEI-ICH KONDOH
Information Systems and Electronics Development Laboratory,
Mitsubishi Electric Corporation,
Kamakura-shi, Kanagawa, JAPAN

TAKASHI CHIKAYAMA
Institute for New Generation Computer Technology (ICOT),
Minato-ku, Tokyo, JAPAN

**ABSTRACT**
Macro expansion capability is introduced to Prolog. Macros are defined in the same way as Prolog clauses are defined, and are expanded anywhere in a program, including both heads and bodies of clauses, and also by clauses themselves. The proposed macro mechanism not only replaces the original macro in place, but also inserts goals and auxiliary clauses. Various extensions to Prolog can be easily implemented using the macro feature. The syntax of macro definition and its expansions along with their rules are presented. Typical examples are also presented.

## 1 Introduction

Macro expansion is a very powerful tool of program abstraction. Abstraction using macros is very similar to that provided by subroutines. Macros differ from subroutines in that inline expansion is explicit for macros, while inline expansion may be used as an optimization technique for subroutines in smart compilers. In some cases, implicit optimization without bothering the programmer is desirable. Yet, in other cases, the programmer may want to specify the expansion explicitly, if he has a little more intelligence than the optimizing compiler.

Macros are frequently used in the two major language families, namely, assembly languages and LISP languages. Although the notions of macro expansion in the two language families look quite similar, there is a crucial difference between them.

Macros in assembly languages are written in a specially devised macro definition language, a meta-language for the assembly language. Though the syntax of the expansion language usually resembles that of the assembly language itself, it is a different language indeed. When macro expansion capability is introduced to an assembly language, the language has to be augmented with a totally new set of instructions (a conditional assembly instruction, for example), and programmers using the language are also asked to learn those new features in order to fully utilize the newly introduced capability. Macros in language C are the same as those of assembly languages in this sense.

The situation is utterly different with LISP. Macros are expanded using LISP itself. In other words, LISP itself is chosen to be the meta-language. Thus, all that is needed to introduce macro expansion capability to a LISP system is only to implement the expansion capability itself. The only extra that programmers have to learn is how to access arguments to macros within macro definitions, how to convey the expanded result to the system, etc. All the essential part of the macro expansion can be written in LISP itself. This is made possible by the fact that programs can be easily handled as data in LISP.

Prolog has the same characteristic as LISP; programs can be nicely processed as data by themselves. Details of expansion can be written in Prolog itself, and, without any doubt, it is much easier to write programs which generate parameterized patterns in Prolog than in LISP, thanks to its powerful unification and backtracking mechanisms.

## 2 Motivation

In Prolog, arguments of body goals are treated as data and are *never evaluated* except for some special built-in predicates. Functional notation, however, is sometimes very convenient.

**[Example 1]** Mathematical operation in arguments

```
    p(X,Y) :- q(X-1,Y).
```

In this example the predicate q should be given an argument one less than X which should be computed as follows:

```
p(X,Y) :- X1 is X-1, q(X1,Y).
```

As a functional notation in the head argument is considered to be an output, subtracting 1 from X should be done in the tail of its body.

```
p(X,X-1) :- integer(X).
===>  p(X,X1) :- integer(X), X1 is X-1.
```

If both patterns can be specified using a Prolog clause itself as follows, it is very convenient.

```
X-Y => Z when Z is X-Y.
```

Such a mechanism can be achieved if an interpreter is invoked at runtime whenever the specified patterns appear. Unfortunately, interpretive execution is usually 20 - 50 times slower than compiled execution. Thus, the interpretive method is not practical. In this paper, we show a mechanism which implements these functions using macro expansion at compilation time, not runtime.

## 3 Requirements

The simplest macro expansion rules are those which replace some pattern appearing as an argument of a head or a body goal.

**[Example 2]** Radix conversion and character code conversion

```
p(16#"A",X) :- q(X,ascii#"A").
    ===> p(10,X) :- q(X,65).
```

In Lisp, as arguments of function calls are evaluated, macro-expanded results are also evaluated. In Prolog, data are explicitly distinguished from executed goals and arguments are *never evaluated*. Thus functional notation for arithmetic operations like in **[Example 1]** cannot be treated well by in-place replacement. The macro expansion mechanism must be able to insert additional goals.

When the inserted goal requires certain conditional or recursive behavior at runtime, some new *clauses* are also added to the program.

**[Example 3]** Replacement of a mapcar macro with a list which consists of absolute values of all elements of the given list.

```
p(List) :- q( mapcar(abs,List) ).

===>    p(List) :- $(List,Result), q(Result).
        $([],[]) .
        $([X|Rest], [Y|Tail]) :-
                        abs(X,Y), $(Rest, Tail).
```

In order to convert each element in the list whose length is not known at compilation time, a new predicate including a loop structure, "$" in the above case, and a goal which calls it is generated.

In the above examples only arguments of goals are considered to be macros. The following patterns are also considered.

(1) When an argument is a structure, its elements, and if some of them are structures, their elements recursively, are expanded.

**[Example 4]** Elements of a list are considered to be macros.

```
p(X,Y) :- q([X+Y,X-Y]).
===>  p(X,Y) :-
           add(X,Y,Z), subtract(X,Y,W), q([Z,W]).
```

(2) Expanded result of a macro may include another macro as its subterm which, in turn, will be expanded, including the cases where the expanded pattern itself is another macro.

(3) Goals in the clause body are expanded. Their expanded patterns are also goals.

**[Example 5]**

```
p(L,X) :- every(L,q), r(L,X).
===>    p(L,X) :- $(L), r(L,X).
        $([]).
        $([H|T]) :- q(H), $(T).
```

(4) Clauses are expanded. Their expanded patterns will be a set of clauses. DCG (Definite Clause Grammars) of DEC-10 Prolog [1] can be implemented using this function.

These requirements are summarized in Table 1. Data, goals, and clauses are considered to be macros.

Another requirement for the macro expansion is that expansion conditions and expanded results should be easily defined. Prolog has the following advantages.

Table 1: Invocation Pattern and its Results

| *Invocation* | *Result* | | |
|---|---|---|---|
| | Data | Goals | Clauses |
| Data | Replacement | Insertion | Insertion |
| Goals | - | Replacement | Insertion |
| Clauses | - | - | Replacement |

(1) Programs can be naturally processed as data.

(2) Unification mechanism can make expansion conditions concise.

(3) Backtracking mechanism can be utilized for specifying complicated macro definitions easily.

Thanks to these mechanisms, especially (2) and (3), Prolog is much more suitable for macro programming than LISP.

## 4 Macro Definition and Examples

### 4.1 Macro Definition

The macro definition has the following syntax. It consists of one Prolog clause.

```
<pattern> "=>" <expansion>
    [ <runtime condition> ]
    [ ":-" <expansion-time condition> ] "."
```

`<Pattern>` is the macro to be expanded. It will be replaced with `<expansion>`. `<Pattern>` and `<expansion>` may include logical variables. A sub-term in a macro, which appears at the position corresponding to a variable in `<pattern>` will be included in the expanded result at the position where the same variable appears in `<expansion>`. Following are examples of the simplest macro definitions without runtime nor expansion-time conditions:

**[Example 6]**

```
(D1)    one  =>  1.
(D2)    ++X  =>  X + one.
```

In this case, every atom "`one`" which appears in the source program is replaced with integer 1 and every compound term "`++X`" is replaced with another compound term "`X+one`".

`<Runtime condition>` has the following format:

```
[ "when" <goals> ]
[ "where" <goals> ]
[ "with" <list of clauses> ]
```

These conditions are also included in the expanded result. How they are expanded will be described in the following sections.

`<Expansion-time condition>` is also a condition associated with the macro expansion, but it is examined when a macro is expanded, rather than being included in the expansion. It is for checking whether the candidate pattern should really be expanded. It is also used for generating patterns in `<expansion>` and/or `<runtime condition>`. When it fails, its macro is not expanded even if the unification with `<pattern>` was successful.

Following are examples of macro definitions:

[Example 7]

```
(D3)      T^I => E  when arg(I,T,E).
(D4)      X+Y => Z  :-
                    integer(X), integer(Y), add(X,Y,Z).
(D5)      X+Y => Z  when  add(X,Y,Z).
```

The definition `(D3)` is for accessing structure arguments in a functional format. Two definitions in `(D4)` and `(D5)` are meant to allow functional description of arithmetical expressions. `(D4)` specifies partial evaluation at macro expansion time. If both operands of an addition are integer like "`1+2`" in the source program, it is evaluated at macro expansion time, not runtime, without any semantic difference. In this way an optimization to specify partial evaluation at macro expansion time is easily defined using `<expansion-time condition>`. When both `(D4)` and `(D5)` are given, (D5) will be used only if expansion by `(D4)` fails.

## 4.2 Macro Expansion in a Clause Body

Macros appearing in a body of a clause, including cases where goals themselves are macros, are expanded as follows:

(1) The candidate pattern is unified with the `<pattern>` part of the macro definition.

(2) The expansion-time condition, if any, is executed in the same way as in normal Prolog execution. If unification (1) or execution (2) fails, next macro definition will become an alternative.

(3) The predicate call including the macro is replaced by a logical conjunction of the following 3 parts:

   1) the runtime conditions preceded by "`when`",
   2) the original predicate call with the concerning pattern substituted by the expansion,
   3) the runtime conditions preceded by "`where`".

(4) The clauses in the list preceded by "`with`" are asserted.

Macros are expanded when the term containing the macro is compiled, consulted or asserted. Even if a pattern looking like a macro is generated during program execution, macro expansion does not take place.

Following are examples of macro expansions occurring in predicate calls:

[Example 8]

```
(B1)  f(M+2)     ===>    add(M,2,Z), f(Z)           (D5)
(B2)  g(++N)     ===>    g(N+one)                   (D2)
                 ===>    add(N,one,Z), g(Z)         (D5)
                 ===>    add(N,1,Z), g(Z)           (D1)
```

Macro expansions are tried only in a top-down manner. Once a term is examined and is not recognized as a macro, it will never be treated as a macro even if a later macro expansion of its sub-terms has made the parent term unifiable with a certain macro pattern.

## 4.3 Macro Expansion in a Clause Head

Macros appearing in the head of a clause are expanded in a slightly different manner:

(1) The candidate pattern is unified with the `<pattern>` part of the macro definition (the same as when expanded within the body).

(2) The expansion-time condition, if any, is executed in the same way as in normal Prolog execution. When unification (1) or execution (2) fails, next macro definition will become an alternative (same).

(3) This step is different. The clause head including the macro is replaced by the original head with concerning pattern substituted by the expansion specified in the macro definition. The body, i.e., the right-hand side of the clause, is replaced by a logical conjunction of the following in this order.

   1) the runtime condition preceded by "`where`",
   2) the original body,
   3) the runtime condition preceded by "`when`".

(4) The clauses in the list preceded by "`with`" are asserted (same).

Following are examples of macro expansions occurring in clause heads.

[Example 9]

```
(D6)  op(::,xfx,200).
      X::Type => X where Term
             :- Term =.. [Type,X].
(C1)  add1(N::integer,++N).
      ===>  add1(N,++N) :- integer(N).                (D6)
      ===>  add1(N,N+one) :- integer(N).              (D2)
      ===>  add1(N,M) :- integer(N), add(N,one,M).   (D5)
      ===>  add1(N,M) :- integer(N), add(N,1,M).     (D1)
```

Extensions like macros and functional notation in [3] can be easily implemented using these macro definitions.

## 4.4 Insertion of Clauses

Newly generated clauses are also asserted using "`with`" definition of `<runtime condition>`.

[**Example 10**] Function is applied to each element of the given list, and the macro is replaced with the list of results like mapcar function of LISP.

```
(D7) mapcar(Function,List) => Result
       when $(List,Result)
       with [ ( $([],[]) ),
              ( $([X|Rest],[Y|Tail]) :-
                      Call, $(Rest,Tail) ) ]
        :- Call =.. [Function,X,Y].


(C2) p(List) :- q( mapcar(abs,List) ).
        ===>    p(List) :- $(List,Result), q(Result).
                $([],[]).
                $([X|Rest],[Y|Tail]) :-
                                abs(X,Y), $(Rest,Tail).
```

In the above example predicate names (**$**) will conflict when there are several mapcar macros. This problem is solved by a mechanism which creates a new unique atom.

## 4.5 Macro Expansion of a Clause

Clauses are also treated as a candidate pattern. A clause is replaced with another clause by the expansion and some new clauses may also be inserted.

[**Example 11**] Generation of clauses of a given predicate with one argument which is an element of the given list. One clause is generated for each of the list element.

```
(D8) op(<-,xfx,1100).
     ( Functor <- List ) => void
                             % void means a null clause.
          with Clauses
          :- create_clauses(List,Functor,Clauses).
     create_clauses([],_,[]) :- !.
     create_clauses([One|Rest],Functor,[Clause|Tail]) :-
          Clause =.. [Functor,One]
          create_clauses(Rest,Functor,Tail).
```

```
(C3) atomic_type <- [ atom, integer, floating_point ].
          ===>   atomic_type(atom).
                 atomic_type(integer).
                 atomic_type(floating_point).
```

### 4.6 Controlling Macro Expansion

When a pattern unifiable with some macro pattern should be treated *as it is* rather than being expanded, it is quoted using the operator "‘" or "‘‘". For example, when a description "`X+1`" should be treated as a compound term whose functor is '+', not as the result of adding `1` to `X`, it is described as "`‘(X+1)`". The term prefixed by "‘" or "‘‘" is never expanded even if it is a macro. The difference between them is that "‘" is effective only for the top-level of the pattern, and "‘‘" is effective also for all levels inside the pattern.

**[Example 12]**

```
(B3)      ... :- ..., p((A+B)+(C+D)), ...
             ===>    ... :- ...,
                          add(A,B,X),
                          add(C,D,Y),
                          add(X,Y,Z),p(Z), ...
(B4)      ... :- ..., p(‘((A+B)+(C+D))), ...
             ===>    ... :- ...,
                          add(A,B,X),
                          add(C,D,Y),
                          p(X+Y), ...
(B5)      ... :- ..., p(‘‘((A+B)+(C+D))), ...
             ===>    ... :- ...,
                          p(((A+B)+(C+D))), ...
```

## 5 Comparison

The functionality provided by the macro expansion mechanism described in the previous sections can be achieved if an interpreter is invoked at runtime whenever specified patterns appear. For example `mapcar` function and **every** can be realized as follows, using meta-level "call" mechanism which essentially invokes the interpreter.[1]

[1] The *univ* operation is avoided here because it is much slower than the combination of `functor/3` and `arg/3`.

Table 2: Evaluation of Inline Expansion

| *Machine* | Function | Interpreter | Inline | ratio |
|---|---|---|---|---|
| DEC10-Prolog on DEC2060 | `mapcar` | 1681msec | 82msec | 20.5 |
| | `every` | 555msec | 37msec | 15.0 |
| Quintus Prolog on VAX8700 | `mapcar` | 480msec | 40msec | 12.0 |
| | `every` | 417msec | 11msec | 37.9 |

```
mapcar([],[],_) :- !.
mapcar([X0|Rest],[X|Tail],F) :-
    functor(Term,F,2),
    arg(1,Term,X0),
    arg(2,Term,X),
    call(Term),
    mapcar(Rest,Tail,F).

every([],_) :- !.
every([X|Tail],F) :-
    functor(Term,F,1),
    arg(1,Term,X),
    call(Term),
    every(Tail,F).
```

In Table 2 execution times of the above examples are shown compared with the cases when inline expansion like **[Example 5]** and **[Example 10]** is done. They are measured with executing the following goal:

```
..., mapcar(List,Output,abs), ...

abs(X,Y) :- X >= 0, !, Y = X.
abs(X,Y) :- Y is - X.

..., every(List,integer), ...
```

"`List`" is a list which consists of 1000 elements like [1,-1,1,-1, .... ]. We used DEC10-Prolog on DEC2060 and Quintus Prolog on VAX8700. When inline expansion is used, the execution is more than 10 times faster than when an interpreter is invoked.

Of course, such an inline expansion can be done only when the function is explicitly given in the source programs. It is easily specified to do inline expansion only when function is explicitly mentioned in the source program and to invoke an interpreter dynamically otherwise, as follows:

```
(D9) every(List,Function) => $(List)
       with [ $([]),
            ( $([One|Tail]) :-
                    Call, $(Tail) ) ]
       :- atom(Function), !, Call =.. [Function,One] .
     every(List,Function) => $(List,Function)
       with [ $([],_),
            ( $([One|Tail],Function) :-
                    functor(Term,Function,1),
                    arg(1,Term,One),
                    call(Term),
                    $(Tail,Function) ) ] .
```

If a macro expansion mechanism can be used, we do not have to describe frequently the inline expanded patterns which are almost the same and need not to be worried about naming expanded predicates no longer. Thus, the writability and readability are improved without lowering of efficiency.

## 6 Implementation

We implemented these macro expansion mechanisms for the language ESP, Extended Self-contained Prolog [2]. ESP is an object-oriented logic programming language based on Prolog and is the system and application program description language for the sequential inference machines PSI and PSI-II [4]. Its macro expansion mechanism has been practically used in many application programs. ESP language has the following standard macros:

(1) Constant values
    e.g. radix conversion, character code conversion

(2) Arithmetic and relational operations

(3) Object-oriented features
    e.g. method call, slot access

In ESP, macro definitions are described in a module called *macro bank*, which is treated as an object class. Macros defined in a macro bank are effective in the *class*[2] definitions where the name of the macro bank is mentioned. The inheritance mechanism allows the sub-macro bank to inherit macro definitions. Normally, macro banks inherit the standard macro bank.

[**Example 13**] Macro bank and class

In ESP ";" is used to end a clause instead of ".".

```
(M1)    macro_bank m1 has
                T^I => E  when  arg(I,T,E);
                X+Y => Z :- integer(X), integer(Y),
                                add(X,Y,Z);
                X+Y => Z when add(X,Y,Z);
        end.

(E1)    class c1 with_macro m1 has
                :p(Obj,V,N,V^N+1) ;
        end.
      ===>
        class c1 has
                :p(Obj,V,N,Elem1) :-
                        arg(N,V,Elem),
                        add(Elem,1,Elem1);
       end.
```

## 7 Remaining Problems

The macro expansion mechanism replaces macros with specified expansion patterns automatically. The writability and readability are improved as the result. However, in some cases, expanded patterns may be greatly different from the corresponding original patterns in the source program. During the debugging phase, users may have difficulties in relating the source image to the traced result.

Following two solutions are considered.

(1) To display programs before macro expansion and expand them at runtime whenever a macro appears. A

[2]ESP programs consist of a set of axiom databases called *classes*.

major problem of this method is the large execution time overhead for macro expansion.

(2) To maintain both original and expanded patterns, display the original one when tracing and use the expanded one for execution. There still remain two problems. First, two patterns must be maintained only for tracing. Second, it is hard to treat macros in the head argument that requires insertion of runtime conditions to the tail of the body.

## 8 Conclusion

Introduction of macro expansion capability to Prolog has been proposed. It has the following features.

(1) Macros are expanded using Prolog itself, that is, Prolog is used as the meta language for the macro processing.

(2) Powerful mechanisms of Prolog, that are, unification and backtracking make macro definitions concise and readable.

(3) The macro mechanism not only replaces the original pattern but also inserts newly generated goals and clauses. By this, the macro expansion feature is used for both data and control abstraction.

DCG (Definite Clause Grammars) of DEC-10 Prolog [1] can be easily implemented using these macros. Many of other special Prolog-like systems, such as, Prolog based on fuzzy theory [5], Prolog including extension [3], and so on, can be also implemented easily. If many desired features can be easily implemented using one general mechanism, there should be no reason to implement each of the features independently.

## References

[1] D. L. Bowen, L. Byrd, F. C. N. Pereira, L. M. Pereira, and D. H. D. Warren. *DECsystem-10 Prolog User's Manual.* November 1983.

[2] T. Chikayama. *ESP Reference Manual.* ICOT Technical Report TR-044, ICOT, 1984.

[3] P. R. Eggert and D. V. Schorre. Logic Enhancement: a Method for Extending Logic Programming Languages. In *Conference Record of the 1982 ACM Simposium on LISP and Functional Programming*, 1982.

[4] H. Nakashima and K. Nakajima. Hardware Architecture of the Sequential Inference Machine:PSI-II. In *Proceedings of IEEE Symposium on Logic Programming*, 1987.

[5] I. P. Orci and M. Karlson. PROLOG/S Programming Language Based on Possibilistic Logic. In *Proceedints of Fourth Japanese-Swedish Workshop on Fifth Generation Computer Systems*, July 1986.