Extended DCG notation:
A Tool for Applicative Programming in Prolog

Peter Van Roy
vanroy@ernie.berkeley.edu

Computer Science Division
University of California
Berkeley, CA 94720

ABSTRACT

This report describes a preprocessor that simplifies purely applicative programming in Prolog. The preprocessor generalizes Prolog's Definite Clause Grammar (DCG) notation to allow programming with multiple accumulators. It has been an indispensable tool in the development of an optimizing Prolog compiler. Its use is transparent in versions of Prolog that conform to the Edinburgh standard. This report contains a description of the preprocessor, a user manual, a large example program, and the source code of the preprocessor. This information is also available by anonymous ftp to arpa.berkeley.edu.

July 24, 1990

Extended DCG notation: A Tool for Applicative Programming in Prolog

Peter Van Roy
vanroy@ernie.berkeley.edu

Computer Science Division
University of California
Berkeley, CA 94720

1. Introduction

It is desirable to program in a purely applicative style, i.e. within the pure logical subset of Prolog. In that case a predicate's meaning depends only on its definition, and not on any outside information. This has two important advantages. First, it greatly simplifies verifying correctness. Simple inspection is often sufficient. Second, since all information is passed locally, it makes the program more amenable to parallel execution. However, in practice the number of arguments of predicates written in this style is large, which makes writing and maintaining them difficult. Two ways of getting around this problem are (1) to encapsulate information in compound structures which are passed in single arguments, and (2) to use global instead of local information. Both of these techniques are commonly used in imperative languages such as C, but neither is a satisfying way to program in Prolog, for the following reasons:

- Because Prolog is a single-assignment language, modifying encapsulated information requires a time-consuming copy of the entire structure. Sophisticated optimizations could make this efficient, but compilers implementing them do not yet exist.
- Using global information destroys the advantages of programming in an applicative style, such as the ease of mathematical analysis and the suitability for parallel execution.

A third approach with neither of the above disadvantages is extending Prolog to allow an arbitrary number of arguments without increasing the size of the source code. The extended Prolog is translated into standard Prolog by a preprocessor. This report describes an extension to Prolog's Definite Clause Grammar notation that implements this idea.

2. Definite Clause Grammar (DCG) notation

DCG notation was developed as the result of research in natural language parsing and understanding [Pereira & Warren 1980]. It allows the specification of a class of attributed unification grammars with semantic actions. These grammars are strictly more powerful than context-free grammars. Prologs that conform to the Edinburgh standard [Clocksin & Mellish 1981] provide a built-in preprocessor that translates clauses written in DCG notation into standard Prolog.

An important Prolog programming technique is the accumulator [Sterling & Shapiro 1986]. The DCG notation implements a single implicit accumulator. For example, the DCG clause:

```
term(S) --> factor(A), [+], factor(B), {S is A+B}.
```

is translated internally into the Prolog clause:

```
term(S,X1,X4) :- factor(A,X1,X2), X2=[+|X3], factor(B,X3,X4), S is A+B.
```

Each predicate is given two additional arguments. Chaining together these arguments implements the accumulator.

3. Extending the DCG notation

The DCG notation is a concise and clear way to express the use of a single accumulator. However, in the development of large Prolog programs I have found it useful to carry more than one accumulator. If written explicitly, each accumulator requires two additional arguments, and these arguments must be chained together. This requires the invention of many arbitrary variable names, and the chance of introducing errors is large. Modifying or extending this code, for example to add another accumulator, is tedious.

One way to solve this problem is to extend the DCG notation. The extension described here allows for an unlimited number of named accumulators, and handles all the tedium of parameter passing. Each accumulator requires a single Prolog fact as its declaration. The bulk of the program source does not depend on the number of accumulators, so maintaining and extending it is simplified. For single accumulators the notation defaults to the standard DCG notation.

Other extensions to the DCG notation have been proposed, for example Extraposition Grammars [Pereira 1981] and Definite Clause Translation Grammars [Abramson 1984]. The motivation for these extensions is natural-language analysis, and they are not directly useful as aids in program construction.

4. An example

To illustrate the extended notation, consider the following Prolog predicate which converts infix expressions containing identifiers, integers, and addition (+) into machine code for a simple stack machine, and also calculates the size of the code:

```
expr_code(A+B, S1, S4, C1, C4) :-
     expr_code(A, S1, S2, C1, C2),
     expr_code(B, S2, S3, C2, C3),
     C3=[plus|C4],       /* Explicitly accumulate 'plus' */
     S4 is S3+1.         /* Explicitly add 1 to the size */
expr_code(I, S1, S2, C1, C2) :-
     atomic(I),
     C1=[push(I)|C2],
     S2 is S1+1.
```

This predicate has two accumulators: the machine code and its size. A sample call is `expr_code(a+3+b,0,Size,Code,[])`, which returns the result:

```
Size = 5
Code = [push(a),push(3),plus,push(b),plus]
```

With DCG notation it is possible to hide the code accumulator, although the size is still calculated explicitly:

```
expr_code(A+B, S1, S4) -->
     expr_code(A, S1, S2),
     expr_code(B, S2, S3),
     [plus],          /* Accumulate 'plus' in a hidden accumulator */
     {S4 is S3+1}.    /* Explicitly add 1 to the size */
expr_code(I, S1, S2) -->
     {atomic(I)},
     [push(I)],
     {S2 is S1+1}.
```

The extended notation hides both accumulators:

```
expr_code(A+B) -->>
     expr_code(A),
     expr_code(B),
     [plus]:code,     /* Accumulate 'plus' in the code accumulator */
     [1]:size.        /* Accumulate 1 in the size accumulator */
expr_code(I) -->>
     {atomic(I)},
     [push(I)]:code,
     [1]:size.
```

The translation of this version is identical to the original definition. The preprocessor needs the following declarations:

```
acc_info(code, T, Out, In, (Out=[T|In]))./* Accumulator declarations */
acc_info(size, T, In, Out, (Out is In+T)).

pred_info(expr_code, 1, [size,code]).    /* Predicate declaration */
```

For each accumulator this declares the accumulating function, and for each predicate this declares the name, arity (number of arguments), and accumulators it uses. The order of the `In` and `Out` arguments determines whether accumulation proceeds in the forward direction (see `size`) or in the reverse direction (see `code`). Choosing the proper direction is important if the accumulating function requires some of its arguments to be instantiated.

5. Concluding remarks

An extension to Prolog's DCG notation that implements an unlimited number of named accumulators was developed to simplify purely applicative Prolog programming. A preprocessor for C-Prolog and Quintus Prolog is available by anonymous ftp to arpa.berkeley.edu or by contacting the author. Comments and suggestions for improvements are welcome.

This research was partially sponsored by the Defense Advanced Research Projects Agency (DoD) and monitored by Space & Naval Warfare Systems Command under Contract No. N00014-88-K-0579.

6. References

[Abramson 1984]

H. Abramson, "Definite Clause Translation Grammars," *Proc. 1984 International Symposium on Logic Programming*, 1984, pp 233-240.

[Clocksin & Mellish 1981]

W.F. Clocksin and C.S. Mellish, "Programming in Prolog," *Springer-Verlag*, 1981.

[Pereira 1981]

F. Pereira, "Extraposition Grammars," *American Journal of Computational Linguistics*, 1981, vol. 7, no. 4, pp 243-255.

[Pereira & Warren 1980]

F. Pereira and D.H.D. Warren, "Definite Clause Grammars for Language Analysis—A Survey of the Formalism and a Comparison with Augmented Transition Networks," *Journal of Artificial Intelligence*, 1980, vol. 13, no. 3, pp 231-278.

[Sterling & Shapiro 1986]

L. Sterling and E. Shapiro, "The Art of Prolog," *MIT Press*, 1986.

Extended DCG notation: A Tool for Applicative Programming in Prolog

User Manual

Peter Van Roy
vanroy@ernie.berkeley.edu

Computer Science Division
University of California
Berkeley, CA 94720

1. Introduction

This manual describes a preprocessor for Prolog that adds an arbitrary number of arguments to a predicate without increasing the size of the source code. The hidden arguments are of two kinds:

(1) Accumulators, useful for results that are calculated incrementally in many predicates. An accumulator expands into two additional arguments per predicate.

(2) Passed arguments, used to pass global information to many predicates. A passed argument expands into a single additional argument per predicate.

The preprocessor has been tested under C-Prolog and Quintus Prolog. It is being used by the author in program development, and is believed to be relatively bug-free. However, it is still being refined and extended. The most recent version is available by anonymous ftp to arpa.berkeley.edu or by contacting the author. Please let me know if you find any bugs. Comments and suggestions for improvements are welcome.

2. Using the preprocessor

The preprocessor is implemented in the file `accumulator.pl`. It must be consulted or compiled before the programs that use it. In Prologs that conform to the Edinburgh standard, such as C-Prolog or Quintus Prolog, the user-defined predicate `term_expansion/2` is called when consulting or compiling each clause that is read. With this hook the use of the preprocessor is transparent.

Clauses to be expanded are of the form `(Head-->>Body)` where `Head` and `Body` are the head and body of the clause. The head is always expanded with all of its hidden arguments. Table 1 summarizes the expansion rules for body goals. In the table, `Goal` denotes any goal in a clause body, `Acc` denotes an accumulator, `Pass` denotes a passed argument, and `Arg` denotes either an accumulator or a passed argument. Hidden arguments of body goals that are not in the head have default values which can be overridden. For compatibility with DCG notation the accumulator `dcg` is available by default. If-then-else is not handled in this version.

The preprocessor assumes the existence of a database of information about the hidden parameters and the predicates to be expanded. Three relations are recognized: a declaration for each predicate, each accumulator, and each passed argument. These relations can be put at the beginning of each file (in which case their scope is the file) or stored

Table 1 — Expansion rules for the preprocessor

Body goal	Action
`{Goal}`	Don't expand any hidden arguments of `Goal`.
`Goal`	Expand all of the hidden parameters of `Goal` that are also in the head. Those hidden parameters not in the head are given default values.
`Goal:L`	If `Goal` has no hidden arguments then force the expansion of all arguments in `L` in the order given. If `Goal` has hidden arguments then expand all of them, using the contents of `L` to override the expansion. `L` is either a term of the form `Acc`, `Acc(Left,Right)`, `Pass`, `Pass(Value)`, or a list of such terms. When present, the arguments `Left`, `Right`, and `Value` override the default values of arguments not in the head.
`List:Acc`	Accumulate a list of terms in the accumulator `Acc`.
`List`	Accumulate a list of terms in the accumulator `dcg`.
`X/Arg`	Unify `X` with the left term for the accumulator or passed argument `Arg`.
`Acc/X`	Unify `X` with the right term for accumulator `Acc`.
`X/Acc/Y`	Unify `X` with the left and `Y` with the right term for the accumulator `Acc`.
`insert(X,Y):Acc`	Insert the arguments `X` and `Y` into the chain implementing the accumulator `Acc`. This is useful when the value of the accumulator changes radically because `X` and `Y` may be the arguments of an arbitrary relation.
`insert(X,Y)`	Insert the arguments `X` and `Y` into the chain implementing the accumulator `dcg`. This inserts the difference list `X-Y` into the accumulated list.

in a separate file that is consulted first (in which case their scope is the whole program).

A short example gives a flavor of what the preprocessor does:

```
% Declare the accumulator 'castor':
acc_info(castor, _, _, _, true).

% Declare the passed argument 'pollux':
pass_info(pollux).

% Declare three predicates using these hidden arguments:
pred_info(p, 1, [castor,pollux]).
pred_info(q, 1, [castor,pollux]).
pred_info(r, 1, [castor,pollux]).

% The program:
p(X) -->> Y is X+1, q(Y), r(Y).
```

This example declares one accumulator, one passed argument, and three predicates using them. The program consists of a single clause. The preprocessor is used as follows: (bold-face denotes user input)

```
% cprolog
```

```
C-Prolog version 1.5
| ?- ['accumulator.pl'].
accumulator.pl consulted 9780 bytes 1.7 sec.

yes
| ?- ['example.pl'].
example.pl consulted 668 bytes 0.25 sec.

yes
| ?-
```

Now the predicate `p(X)` has been expanded. We can see what it looks like with the `listing` command:

```
| ?- listing(p).

p(X, S1, S3, P) :- Y is X+1, q(Y, S1, S2, P), r(Y, S2, S3, P).
```

(Variable names have been changed for clarity.) The arguments `S1`, `S2`, and `S3`, which implement the accumulator `castor`, are chained together. The argument `P` implements the passed argument. It is added as an extra argument to each predicate.

In object-oriented terminology the declarations of hidden parameters correspond to classes with a single method defined for each. Declarations of predicates specify the inheritance of the predicate from multiple classes, namely each hidden parameter.

3. Declarations

3.1. Declaration of the predicates

Predicates are declared with facts of the following form:

```
pred_info(Name, Arity, List)
```

The predicate `Name/Arity` has the hidden parameters given in `List`. The parameters are added in the order given by `List` and their names must be atoms.

3.2. Declaration of the accumulators

Accumulators are declared with facts in one of two forms. The short form is:

```
acc_info(Acc, Term, Left, Right, Joiner)
```

The long form is:

```
acc_info(Acc, Term, Left, Right, Joiner, LStart, RStart)
```

In most cases the short form gives sufficient information. It declares the accumulator `Acc`, which must be an atom, along with the accumulating function, `Joiner`, and its arguments `Term`, the term to be accumulated, and `Left` & `Right`, the variables used in chaining.

The long form of `acc_info` is useful in more complex programs. It contains two additional arguments, `LStart` and `RStart`, that are used to give default starting values for an accumulator occurring in a body goal that does not occur in the head. The starting values are given to the unused accumulator to ensure that it will execute correctly even though its value is not used. Care is needed to give correct values for `LStart` and `RStart`. For DCG-like list accumulation both may remain unbound.

Two conventions are used for the two variables used in chaining depending on which direction the accumulation is done. For forward accumulation, `Left` is the input and `Right` is the output. For reverse accumulation, `Right` is the input and `Left` is the output.

To see how these declarations work, consider the following program:

```
% Example illustrating the difference between
% forward and reverse accumulation:

% Declare the accumulators:
acc_info(fwd, T, In, Out, Out=[T|In]).    % Forward accumulator.
acc_info(rev, T, Out, In, Out=[T|In]).    % Reverse accumulator.

% Declare the predicates using them:
pred_info(flist, 1, [fwd]).
pred_info(rlist, 1, [rev]).

% flist(N, [], List) creates the list [1, 2, ..., N]
flist(0) -->> [].
flist(N) -->> N>0, [N]:fwd, N1 is N-1, flist(N1).

% rlist(N, List, []) creates the list [N, ..., 2, 1]
rlist(0) -->> [].
rlist(N) -->> N>0, [N]:rev, N1 is N-1, rlist(N1).
```

This defines two accumulators `fwd` and `rev` that both accumulate lists, but in different directions. The joiner of both accumulators is the unification `Out=[T|In]`, which adds `T` to the head of the list `In` and creates the list `Out`. In accumulator `fwd` the output `Out` is the left argument and the input `In` is the right argument. This builds the list in ascending order. Switching the arguments, as in the accumulator `rev`, builds the list in reverse. A sample execution gives these results:

```
| ?- flist(10, [], List).

List = [1,2,3,4,5,6,7,8,9,10]

yes
| ?- rlist(10, List, []).

List = [10,9,8,7,6,5,4,3,2,1]

yes
| ?-
```

If the joining function is not reversible then the accumulator can only be used in one direction. For example, the accumulator `add` with declaration:

```
acc_info(add, I, In, Out, Out is I+In).
```

It can only be used as a forward accumulator. Attempting to use it in reverse results in an error because the argument `In` of the joiner is uninstantiated. The reason for this is that the predicate `is/2` is not pure logic: it requires the expression in its right-hand side to be ground.

3.3. Declaration of the passed arguments

Passed arguments are declared as facts in one of two forms. The short form is:

```
pass_info(Pass)
```

The long form is:

```
pass_info(Pass, PStart)
```

In most cases the short form is sufficient. It declares a passed argument `Pass`, that must be an atom. The long form also contains the starting value `PStart` that is used to give a default value for a passed argument in a body goal that does not occur in the head. Most of the time this situation does not occur.

4. Tips and techniques

Usually there will be one clause of `pred_info` for each predicate in the program. If the program becomes very large, the number of clauses of `pred_info` grows accordingly and can become difficult to keep consistent. In that case it is useful to remember that a single `pred_info` clause can summarize many facts. For example, the following declaration:

```
pred_info(_, _, List).
```

gives all predicates the hidden parameters in `List`. This keeps programming simple regardless of the number of hidden parameters.

5. Acknowledgements

This research was partially sponsored by the Defense Advanced Research Projects Agency (DoD) and monitored by Space & Naval Warfare Systems Command under Contract No. N00014-88-K-0579.

6. References

[Abramson 1984]

H. Abramson, "Definite Clause Translation Grammars," *Proc. 1984 International Symposium on Logic Programming*, 1984, pp 233-240.

[Clocksin & Mellish 1981]

W.F. Clocksin and C.S. Mellish, "Programming in Prolog," *Springer-Verlag*, 1981.

[O'Keefe 1988]

R. A. O'Keefe, "Practical Prolog for Real Programmers," *Tutorial 8, Fifth International Conference Symposium on Logic Programming, Aug. 1988.*

[Pereira 1981]

F. Pereira, "Extraposition Grammars," *American Journal of Computational Linguistics*, 1981, vol. 7, no. 4, pp 243-255.

[Pereira & Shieber 1987]

F. Pereira and S. Shieber, "Prolog and Natural-Language Analysis," *CSLI Lecture Notes*, 1987, no. 10.

[Pereira & Warren 1980]

F. Pereira and D.H.D. Warren, "Definite Clause Grammars for Language Analysis—A Survey of the Formalism and a Comparison with Augmented Transition

Networks," *Journal of Artificial Intelligence*, 1980, vol. 13, no. 3, pp 231-278.

[Sterling & Shapiro 1986]

L. Sterling and E. Shapiro, "The Art of Prolog," *MIT Press*, 1986.

An example using the extended DCG preprocessor

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Copyright (C) 1989 Regents of the University of California.
% All rights reserved.  This program may be freely used and modified for
% non-commercial purposes provided this copyright notice is kept unchanged.
% Written by Peter Van Roy
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% The compilation of unification into a low-level intermediate language.
% For an explanation of what this program does see "An Intermediate Language
% to Support Prolog's Unification", Proceedings of the 1989 North American
% Conference on Logic Programming, MIT Press, vol. 2, pp. 1148-1164.

% Some sample executions:
% | ?- u(X, a, [X], Out).
% | ?- u(X, [Y,Z], [X,Y,Z], Out).
% | ?- u(X, t(1,2,s(3)), [], Out).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Declarations:

% Accumulators:
acc_info(code,   T, Out, In, (Out=[T|In])).   % Generated code.
acc_info(vars,   V, In, Out, incl(V,In,Out)). % Set of initialized variables.
acc_info(offset, I, In, Out, (Out is I+In)).  % Offset into writemode term.
acc_info(size,   I, In, Out, (Out is I+In)).  % Size of a term.

% Predicates:
pred_info(u,          2, [       vars     ]).
pred_info(init_var,   3, [            code]).
pred_info(Name,   Arity, [       vars,code]) :- member(Name/Arity,
        [unify/2, uninit/2, init/4, unify_var/2, unify_block/4, make_slots/5,
         unify_writemode/4, unify_readmode/3, unify_args/6, unify_arg/6]).
pred_info(block,      2, [offset,vars,code]).
pred_info(block_args, 5, [offset,vars,code]).
pred_info(size,       1, [size]).
pred_info(size_args,  3, [size]).

member(X, [X|_]).
member(X, [_|L]) :- member(X, L).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% u(X, T, In, Out)
% Unify variable X with term T and write the result:
u(X, T) -->> unify(X, T):code(Code,[]), write_code(Code, 4).

% unify(X, T, In, Out, Code, Link)
% Unify the variable X with the term T, given the set In of
% variables initialized before the unification.
% Return the intermediate code generated in the accumulator 'code'
```

```
% and the set Out of variables initialized after the unification.
unify(X, T) -->> In/vars, \+in(X, In), !, uninit(X, T).
unify(X, T) -->> In/vars,   in(X, In), !,   init(X, T, nonlast, _).

%**** Uninit assumes X has not yet been initialized:
uninit(X, T) -->> compound(T), !, [move(Tag^h, X)]:code,
        termtag(T, Tag), unify_block(nonlast, T, _, _), [X]:vars.
uninit(X, T) -->>   atomic(T), !, [move(tatm^T, X)]:code, [X]:vars.
uninit(X, T) -->>      var(T), !, unify_var(X, T).

%**** Init assumes X has already been initialized:
init(X, T, Last, LLbls) -->> nonvar(T), !,
        termtag(T,Tag),
        [deref(X), switch(Tag,X,[trail(X)|Write],Read,fail)]:code,
        insert(In,Out):vars,
        {unify_writemode(X, T, Last, LLbls, In, _, Write, [])},
        {unify_readmode(X, T, LLbls, In, Out, Read, [])}.
init(X, T,    _,     _) -->> var(T), !, unify_var(X, T).

%**** Unifying two variables together:
unify_var(X, Y) -->> In/vars, in(X, In),   in(Y, In), !, [unify(X,Y,fail)]:code.
unify_var(X, Y) -->> In/vars, in(X, In), \+in(Y, In), !, [move(X,Y)]:code,
        [Y]:vars.
unify_var(X, Y) -->> In/vars, \+in(X, In), in(Y, In), !, [move(Y,X)]:code,
        [X]:vars.
unify_var(X, Y) -->> In/vars, \+in(X, In), \+in(Y, In), !,
        [move(tvar^h,X), move(tvar^h,Y), add(1,h), move(Y,[h-1])]:code,
        [X,Y]:vars.

%**** Unify_readmode assumes X is a dereferenced nonvariable
% at run-time and T is a nonvariable at compile-time.
unify_readmode(X, T, LLbls) -->> structure(T), !,
        functor(T, F, N), [equal([X],tatm^(F/N),fail)]:code,
        unify_args(1, N, T, 0, X, LLbls).
unify_readmode(X, T, LLbls) -->> cons(T), !,
        unify_args(1, 2, T, -1, X, LLbls).
unify_readmode(X, T,     _) -->> atomic(T), !,
        [equal(X,tatm^T,fail)]:code.

unify_args(I, N, _, _, _,          _) -->> I>N, !.
unify_args(I, N, T, D, X, [_|LLbls]) -->> I=N, !,
        unify_arg(I, T, D, X, last, LLbls).
unify_args(I, N, T, D, X,     LLbls) -->> I<N, !,
        unify_arg(I, T, D, X, nonlast, _),
        I1 is I+1, unify_args(I1, N, T, D, X, LLbls).

unify_arg(I, T, D, X, Last, LLbls) -->>
        [move([X+ID],Y)]:code,
        ID is I+D, arg(I, T, A),
        [Y]:vars,
        init(Y, A, Last, LLbls).

%**** Unify_writemode assumes X is a dereferenced unbound
% variable at run-time and T is a nonvariable at compile-time.
```

```
unify_writemode(X, T, Last, LLbls) -->> compound(T), !,
        [move(Tag^h,[X])]:code,
        termtag(T, Tag),
        unify_block(Last, T, _, LLbls).
unify_writemode(X, T,    _,     _) -->> atomic(T), !,
        [move(tatm^T,[X])]:code.

%**** Generate a minimal sequence of moves to create T on the heap:
unify_block(   last, T, Size,   [Lbl|_]) -->> !,
        size(T):size(0,Size),
        [add(Size,h),jump(Lbl)]:code.
unify_block(nonlast, T, Size, [_|LLbls]) -->> !,
        size(T):size(0,Size), Offset is -Size,
        [add(Size,h)]:code,
        block(T, LLbls):offset(Offset,0).

block(T, LLbls) -->> structure(T), !, D/offset,
        [move(tatm^(F/N),[h+D])]:code,
        functor(T, F, N),
        [N]:offset, [1]:offset,
        D1 is D+1,
        make_slots(1, N, T, D1, Offsets),
        block_args(1, N, T, Offsets, LLbls).
block(T, LLbls) -->> cons(T), !, D/offset,
        [2]:offset,
        make_slots(1, 2, T, D, Offsets),
        block_args(1, 2, T, Offsets, LLbls).
block(T,     []) -->> atomic(T), !.
block(T,     []) -->> var(T), !.

block_args(I, N, _,  [],            []) -->> I>N, !.
block_args(I, N, T, [D],   [Lbl|LLbls]) -->> I=N, !, D/offset,
        [label(Lbl)]:code,
        arg(I, T, A), block(A, LLbls).
block_args(I, N, T, [D|Offsets], LLbls) -->> I<N, !, D/offset,
        arg(I, T, A), block(A, _), I1 is I+1,
        block_args(I1, N, T, Offsets, LLbls).

make_slots(I, N, _, _,            []) -->> I>N, !.
make_slots(I, N, T, D, [Off|Offsets]) -->> I=<N, !,
        arg(I, T, A),
        In/vars, init_var(A, D, In),
        make_word(A, Off, Word),
        [move(Word,[h+D])]:code,
        [A]:vars,
        D1 is D+1, I1 is I+1,
        make_slots(I1, N, T, D1, Offsets).

% Initialize first-time variables in write mode:
init_var(V, I, In) -->> var(V), \+in(V, In), !, [move(tvar^(h+I),V)]:code.
init_var(V, _, In) -->> var(V),   in(V, In), !.
init_var(V, _,  _) -->> nonvar(V), !.

make_word(C, Off, Tag^(h+Off)) :- compound(C), !, termtag(C, Tag).
```

```
make_word(V,    _, V)           :- var(V), !.
make_word(A,    _, tatm^A)      :- atomic(A), !.

% Calculate the size of T on the heap:
size(T) -->> structure(T), !,
        functor(T, _, N), [1]:size, [N]:size, size_args(1, N, T).
size(T) -->> cons(T), !,
        [2]:size, size_args(1, 2, T).
size(T) -->> atomic(T), !.
size(T) -->> var(T), !.

size_args(I, N, _) -->>  I>N, !.
size_args(I, N, T) -->> I=<N, !,
        arg(I, T, A), size(A), I1 is I+1, size_args(I1, N, T).

%**** Utility routines:

in(A, [B|S]) :- compare(Order, A, B), in_2(Order, A, S).

in_2(=, _, _).
in_2(>, A, S) :- in(A, S).

incl(A, S1, S) :- var(A), !, incl_2(S1, A, S).
incl(A,  S, S) :- nonvar(A).

incl_2([], A, [A]).
incl_2([B|S1], A, S) :- compare(Order, A, B), incl_3(Order, A, B, S1, S).

incl_3(<, A, B, S1, [A,B|S1]).
incl_3(=, _, B, S1,   [B|S1]).
incl_3(>, A, B, S1,    [B|S]) :- incl_2(S1, A, S).

compound(X)  :- nonvar(X), \+atomic(X).
cons(X)      :- compound(X),   X=[_|_].
structure(X) :- compound(X), \+X=[_|_].

termtag(T, tstr) :- structure(T).
termtag(T, tlst) :- cons(T).
termtag(T, tatm) :- atomic(T).
termtag(T, tvar) :- var(T).

write_code([], _).
write_code([I|L], N) :- write_code(I, L, N).

write_code(switch(Tag,V,Wbr,Rbr,Fail), L, N) :- !, N1 is N+4,
        tab(N), write('switch('), write(V), write(') {'), nl,
        tab(N), write(tvar), write(':'), nl, write_code(Wbr, N1),
        tab(N), write(Tag), write(':'), nl, write_code(Rbr, N1),
        tab(N), write('else: '), write(Fail), nl,
        tab(N), write('}'), nl, write_code(L, N).
write_code(label(Lbl), L, N) :- !, N1 is N-4,
        tab(N1), write(Lbl), write(':'), nl, write_code(L, N).
write_code(Instr, L, N)' :-
        tab(N), write(Instr), nl, write_code(L, N).
```

Source code of the extended DCG preprocessor

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Copyright (C) 1989 Peter Van Roy and Regents of the University of California.
% All rights reserved.  This program may be freely used and modified for
% non-commercial purposes provided this copyright notice is kept unchanged.
% Written by Peter Van Roy
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Multiple hidden parameters: an extension to Prolog's DCG notation.
% Version: July 16, 1989

:- op(1200, xfx, ['-->>']).    % Same as ':-'.
:- op( 850, xfx, [':']).       % Slightly tighter than ',' and '\+'.

% The predicate term_expansion/2 implements the extended translation.
% If loaded into Prolog along with the appropriate acc_info, pass_info,
% and pred_info facts it will be used automatically when consulting programs.

term_expansion((H-->>B), (TH:-FTB)) :-
        functor(H, Na, Ar),
        '_has_hidden'(H, HList),
        '_new_goal'(H, HList, HArity, TH),
        '_create_acc_pass'(HList, HArity, TH, Acc, Pass),
        '_flat_conj'(B, FB),
        '_expand_body'(FB, TB, Na/Ar, HList, Acc, Pass),
        '_flat_conj'(TB, FTB), !.

'_expand_body'(true, true, _, _, Acc, _) :- '_finish_acc'(Acc).
'_expand_body'((G,B), (TG,TB), NaAr, HList, Acc, Pass) :-
        '_expand_goal'(G, TG, NaAr, HList, Acc, NewAcc, Pass),
        '_expand_body'(B, TB, NaAr, HList, NewAcc, Pass).

% Expand a single goal:
'_expand_goal'({G}, G, _, _, Acc, Acc, _) :- !.
'_expand_goal'(insert(X,Y), LeftA=X, _, _, Acc, NewAcc, _) :-
        '_replace_acc'(dcg, LeftA, RightA, Y, RightA, Acc, NewAcc), !.
'_expand_goal'(insert(X,Y):A, LeftA=X, _, _, Acc, NewAcc, _) :-
        '_replace_acc'(A, LeftA, RightA, Y, RightA, Acc, NewAcc), !.
% Force hidden arguments in L to be appended to G:
'_expand_goal'((G:A), TG, _, HList, Acc, NewAcc, Pass) :-
        \+'_list'(G),
        '_has_hidden'(G, []), !,
        '_make_list'(A, AList),
        '_new_goal'(G, AList, GArity, TG),
        '_use_acc_pass'(AList, GArity, TG, Acc, NewAcc, Pass).
% Use G's regular hidden arguments & override defaults for those arguments
% not in the head:
'_expand_goal'((G:A), TG, _, HList, Acc, NewAcc, Pass) :-
        \+'_list'(G),
        '_has_hidden'(G, GList), GList\==[], !,
        '_make_list'(A, L),
        '_new_goal'(G, GList, GArity, TG),
```

```
        '_replace_defaults'(GList, NGList, L),
        '_use_acc_pass'(NGList, GArity, TG, Acc, NewAcc, Pass).
'_expand_goal'((L:A), Joiner, NaAr, _, Acc, NewAcc, _) :-
        '_list'(L), !,
        '_joiner'(L, A, NaAr, Joiner, Acc, NewAcc).
'_expand_goal'(L, Joiner, NaAr, _, Acc, NewAcc, _) :-
        '_list'(L), !,
        '_joiner'(L, dcg, NaAr, Joiner, Acc, NewAcc).
'_expand_goal'((X/A), true, _, _, Acc, Acc, _) :-
        var(X), nonvar(A),
        '_member'(acc(A,X,_), Acc), !.
'_expand_goal'((X/A), true, _, _, Acc, Acc, Pass) :-
        var(X), nonvar(A),
        '_member'(pass(A,X), Pass), !.
'_expand_goal'((A/X), true, _, _, Acc, Acc, _) :-
        var(X), nonvar(A),
        '_member'(acc(A,_,X), Acc), !.
'_expand_goal'((X/A/Y), true, _, _, Acc, Acc, _) :-
        var(X), var(Y), nonvar(A),
        '_member'(acc(A,X,Y), Acc), !.
'_expand_goal'((X/Y), true, NaAr, _, Acc, Acc, _) :-
        write('*** Warning: in '),write(NaAr),write(' the term '),write(X/Y),
        write(' uses a non-existent hidden parameter.'),nl.
% Defaulty cases:
'_expand_goal'(G, TG, HList, _, Acc, NewAcc, Pass) :-
        '_has_hidden'(G, GList), !,
        '_new_goal'(G, GList, GArity, TG),
        '_use_acc_pass'(GList, GArity, TG, Acc, NewAcc, Pass).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Operations on the Acc and Pass data structures:

% Create the Acc and Pass data structures:
% Acc contains terms of the form acc(A,LeftA,RightA) where A is the name of an
% accumulator, and RightA and LeftA are the accumulating parameters.
% Pass contains terms of the form pass(A,Arg) where A is the name of a passed
% argument, and Arg is the argument.
'_create_acc_pass'([], _, _, [], []).
'_create_acc_pass'([A|AList], Index, TGoal, [acc(A,LeftA,RightA)|Acc], Pass) :-
        '_is_acc'(A), !,
        Index1 is Index+1,
        arg(Index1, TGoal, LeftA),
        Index2 is Index+2,
        arg(Index2, TGoal, RightA),
        '_create_acc_pass'(AList, Index2, TGoal, Acc, Pass).
'_create_acc_pass'([A|AList], Index, TGoal, Acc, [pass(A,Arg)|Pass]) :-
        '_is_pass'(A), !,
        Index1 is Index+1,
        arg(Index1, TGoal, Arg),
        '_create_acc_pass'(AList, Index1, TGoal, Acc, Pass).
'_create_acc_pass'([A|AList], Index, TGoal, Acc, Pass) :-
        \+'_is_acc'(A),
        \+'_is_pass'(A),
```

```
        write('*** Error: '),write(A),
        write(' is not a hidden parameter.'),nl.

% Use the Acc and Pass data structures to create the arguments of a body goal:
% Add the hidden parameters named in GList to the goal.
'_use_acc_pass'([], _, _, Acc, Acc, _).
% 1a. The accumulator A is used in the head:
'_use_acc_pass'([A|GList], Index, TGoal, Acc, NewAcc, Pass) :-
        '_replace_acc'(A, LeftA, RightA, MidA, RightA, Acc, MidAcc), !,
        Index1 is Index+1,
        arg(Index1, TGoal, LeftA),
        Index2 is Index+2,
        arg(Index2, TGoal, MidA),
        '_use_acc_pass'(GList, Index2, TGoal, MidAcc, NewAcc, Pass).
% 1b. The accumulator A is not used in the head:
'_use_acc_pass'([A|GList], Index, TGoal, Acc, NewAcc, Pass) :-
        '_acc_info'(A, LStart, RStart), !,
        Index1 is Index+1,
        arg(Index1, TGoal, LStart),
        Index2 is Index+2,
        arg(Index2, TGoal, RStart),
        '_use_acc_pass'(GList, Index2, TGoal, Acc, NewAcc, Pass).
% 2a. The passed argument A is used in the head:
'_use_acc_pass'([A|GList], Index, TGoal, Acc, NewAcc, Pass) :-
        '_is_pass'(A),
        '_member'(pass(A,Arg), Pass), !,
        Index1 is Index+1,
        arg(Index1, TGoal, Arg),
        '_use_acc_pass'(GList, Index1, TGoal, Acc, NewAcc, Pass).
% 2b. The passed argument A is not used in the head:
'_use_acc_pass'([A|GList], Index, TGoal, Acc, NewAcc, Pass) :-
        '_pass_info'(A, AStart), !,
        Index1 is Index+1,
        arg(Index1, TGoal, AStart),
        '_use_acc_pass'(GList, Index1, TGoal, Acc, NewAcc, Pass).
% 3. Defaulty case when A does not exist:
'_use_acc_pass'([A|GList], Index, TGoal, Acc, Acc, Pass) :-
        write('*** Error: the hidden parameter '),write(A),
        write(' does not exist.'),nl.

% Finish the Acc data structure:
% Link its Left and Right accumulation variables together in pairs:
'_finish_acc'([]).
'_finish_acc'([acc(_,Link,Link)|Acc]) :- '_finish_acc'(Acc).

% Replace elements in the Acc data structure:
% Succeeds iff replacement is successful.
'_replace_acc'(A, L1, R1, L2, R2, Acc, NewAcc) :-
        '_member'(acc(A,L1,R1), Acc), !,
        '_replace'(acc(A,_,_), acc(A,L2,R2), Acc, NewAcc).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Specialized utilities:
```

```
% Given a goal Goal and a list of hidden parameters GList
% create a new goal TGoal with the correct number of arguments.
% Also return the arity of the original goal.
'_new_goal'(Goal, GList, GArity, TGoal) :-
        functor(Goal, Name, GArity),
        '_number_args'(GList, GArity, TArity),
        functor(TGoal, Name, TArity),
        '_match'(1, GArity, Goal, TGoal).

% Add the number of arguments needed for the hidden parameters:
'_number_args'([], N, N).
'_number_args'([A|List], N, M) :-
        '_is_acc'(A), !,
        N2 is N+2,
        '_number_args'(List, N2, M).
'_number_args'([A|List], N, M) :-
        '_is_pass'(A), !,
        N1 is N+1,
        '_number_args'(List, N1, M).

% Give a list of G's hidden parameters:
'_has_hidden'(G, GList) :-
        functor(G, GName, GArity),
        pred_info(GName, GArity, GList).
'_has_hidden'(G, []) :-
        functor(G, GName, GArity),
        \+pred_info(GName, GArity, _).

% Succeeds if A is an accumulator:
'_is_acc'(A)  :- atomic(A), !, '_acc_info'(A, _, _, _, _, _, _).
'_is_acc'(A)  :- functor(A, N, 2), !, '_acc_info'(N, _, _, _, _, _, _).

% Succeeds if A is a passed argument:
'_is_pass'(A) :- atomic(A), !, '_pass_info'(A, _).
'_is_pass'(A) :- functor(A, N, 1), !, '_pass_info'(N, _).

% Get initial values for the accumulator:
'_acc_info'(AccParams, LStart, RStart) :-
        functor(AccParams, Acc, 2),
        '_is_acc'(Acc), !,
        arg(1, AccParams, LStart),
        arg(2, AccParams, RStart).
'_acc_info'(Acc, LStart, RStart) :-
        '_acc_info'(Acc, _, _, _, _, LStart, RStart).

% Isolate the internal database from the user database:
'_acc_info'(Acc, Term, Left, Right, Joiner, LStart, RStart) :-
        acc_info(Acc, Term, Left, Right, Joiner, LStart, RStart).
'_acc_info'(Acc, Term, Left, Right, Joiner, _, _) :-
        acc_info(Acc, Term, Left, Right, Joiner).
'_acc_info'(dcg, Term, Left, Right, Left=[Term|Right], _, []).

% Get initial value for the passed argument:
```

```
% Also, isolate the internal database from the user database.
'_pass_info'(PassParam, PStart) :-
        functor(PassParam, Pass, 1),
        '_is_pass'(Pass), !,
        arg(1, PassParam, PStart).
'_pass_info'(Pass, PStart) :-
        pass_info(Pass, PStart).
'_pass_info'(Pass, _) :-
        pass_info(Pass).

% Calculate the joiner for an accumulator A:
'_joiner'([], _, _, true, Acc, Acc).
'_joiner'([Term|List], A, NaAr, (Joiner,LJoiner), Acc, NewAcc) :-
        '_replace_acc'(A, LeftA, RightA, MidA, RightA, Acc, MidAcc),
        '_acc_info'(A, Term, LeftA, MidA, Joiner, _, _), !,
        '_joiner'(List, A, NaAr, LJoiner, MidAcc, NewAcc).
% Defaulty case:
'_joiner'([Term|List], A, NaAr, Joiner, Acc, NewAcc) :-
        write('*** Warning: in '),write(NaAr),
        write(' the accumulator '),write(A),
        write(' does not exist.'),nl,
        '_joiner'(List, A, NaAr, Joiner, Acc, NewAcc).

% Replace hidden parameters with ones containing initial values:
'_replace_defaults'([], [], _).
'_replace_defaults'([A|GList], [NA|NGList], AList) :-
        '_replace_default'(A, NA, AList),
        '_replace_defaults'(GList, NGList, AList).

'_replace_default'(A, NewA, AList) :-  % New initial values for accumulator.
        functor(NewA, A, 2),
        '_member'(NewA, AList), !.
'_replace_default'(A, NewA, AList) :-  % New initial values for passed argument.
        functor(NewA, A, 1),
        '_member'(NewA, AList), !.
'_replace_default'(A, NewA, _) :-      % Use default initial values.
        A=NewA.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Generic utilities:

% Match arguments L, L+1, ..., H of the predicates P and Q:
'_match'(L, H, _, _) :- L>H, !.
'_match'(L, H, P, Q) :- L=<H, !,
        arg(L, P, A),
        arg(L, Q, A),
        L1 is L+1,
        '_match'(L1, H, P, Q).

% Flatten a conjunction and terminate it with 'true':
'_flat_conj'(Conj, FConj) :- '_flat_conj'(Conj, FConj, true).

'_flat_conj'(true, X, X).
```

```
'_flat_conj'((A,B), X1, X3) :-
        '_flat_conj'(A, X1, X2),
        '_flat_conj'(B, X2, X3).
'_flat_conj'(G, (G,X), X) :-
        \+G=true,
        \+G=(_,_).

'_member'(X, [X|_]).
'_member'(X, [_|L]) :- '_member'(X, L).

'_list'(L) :- nonvar(L), L=[_|_], !.
'_list'(L) :- L==[], !.

'_append'([], L, L).
'_append'([X|L1], L2, [X|L3]) :- '_append'(L1, L2, L3).

'_make_list'(A, [A]) :- \+'_list'(A), !.
'_make_list'(L,   L) :-   '_list'(L), !.

% replace(Elem, RepElem, List, RepList)
'_replace'(_, _, [], []).
'_replace'(A, B, [A|L], [B|R]) :- !,
        '_replace'(A, B, L, R).
'_replace'(A, B, [C|L], [C|R]) :-
        \+C=A, !,
        '_replace'(A, B, L, R).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```