F The accumulator preprocessor

The accumulator preprocessor is a powerful tool to simplify the development of large programs. The preprocessor does a source-to-source transformation that adds accumulators to predicates. An *accumulator* provides a means of calculating a value incrementally. For example, the incremental calculation could be the building of a list. Implementing an accumulator consists in adding two arguments to each predicate and chaining the arguments between goals inside each clause. This passes the intermediate values around during the calculation. The preprocessor can also expand single arguments (called "passed arguments" below) which is useful to pass global information to procedures.

To speed up Wild LIFE's start up time, the preprocessor is not loaded by default. Any program using it must first import (*i.e.*, load and open) the preprocessor module with the command `import("accumulators")`.

Example F.1 This example shows how to use the accumulator preprocessor to write a program `main(N,L)` that takes an input N and generates a list L of integers from 1 to N. The accumulator `myacc` is used to accumulate the elements of the list. This example is shown here in its entirety to show how to use the preprocessor and to give a flavor of its the abilities.

```
> import("accumulators")?
*** Loading File "Tools/accumulators.lf"
*** Loading File "Tools/std_expander.lf"
*** Loading File "Tools/acc_declarations.lf"

> acc_info(myacc,X,In,Out,acc_pred=>(Out=[X|In]))?
> pred_info(loop,myacc)?

> loop(0) :-- !?
> loop(N) :-- N+myacc, loop(N-1)?
> main(N,L) :-- loop(N) with myacc([],L)?

> listing(loop,main)?

loop(0, in_myacc=>A, out_myacc=>A) :- !, succeed.
loop(A, in_myacc=>B, out_myacc=>C) :- D=[A|B],
    loop(A-1, in_myacc=>D, out_myacc=>C).

main(A,B) :- loop(A, in_myacc=>[], out_myacc=>B).

> main(10,L)?

*** Yes
L = [1,2,3,4,5,6,7,8,9,10].
```

The declaration `acc_info` declares the accumulator `myacc`. The declaration `pred_info` declares that predicate `loop` uses `myacc`. The predicates `loop` and `main` are defined using queries with root `:--` (*i.e.*, terminated with `?`) instead of definitions with root `:-`. The listing of `loop` and `main` shows how the accumulator `myacc` is added. Complete explanations of all these items are given below.

The accumulator preprocessor does a generalization of the DCG (Definite Clause Grammar) expansion of Prolog. Definite Clause Grammars (DCGs) are the standard example of a preprocessor for a single accumulator, which is in this case a difference list. This is a standard technique, described in Prolog textbooks, for example in Sterling and Shapiro's Art of Prolog [19]. The Wild LIFE preprocessor replaces Prolog terms by ψ-terms.

The generalized technique, called EDCG (Extended Definite Clause Grammar), was developed and implemented for the Aquarius compiler [16, 17]. It has proven extremely useful in the development of large Prolog programs. It has been used by the authors and others to develop various compilers, simulators, analyzers and test generators.

The Wild LIFE preprocessor jointly developed by Dumant and Van Roy provides much extra functionality over the EDCG preprocessor. It is being used in the development of the LIFE compiler.

F.1 Accumulators

F.1.1 Basic examples and syntax

The rules to be expanded are written:

```
Head :-- Body?
```

`Head` is like any clause head; `Body` is like any predicate definition body except that some special symbols may appear in predicate places (see below: accumulation and other features).

The predicates occurring in the rule are expanded according to the `pred_info` declarations attached to them, that tell the preprocessor which arguments have to be added.

```
pred_info(p,castor)?
```

This declaration states that the accumulator `castor` should be expanded when `p` is encountered. The arguments of `pred_info` may also be lists, to relate a set of predicates and a set of accumulators.

◑◑ If you use named features, be careful that there are no conflicts with the features added by the preprocessor. These features always start with `in_` or `out_`.

Example F.2 (adapted from Van Roy)

```
pred_info([p,q],[castor,pollux])?
pred_info(r,castor)?
```

The clause `p :-- p, q, r, s?` is translated into:

```
p(in_castor => A,in_pollux => B,
  out_castor => C,out_pollux => D) :-
```

```
    p(in_castor => A,in_pollux => B,
      out_castor => E,out_pollux => F),
    q(in_castor => E,in_pollux => F,
      out_castor => G,out_pollux => D),
    r(in_castor => G,out_castor => C),
    s.
```

F.1.2 Accumulation

The principal operation performed by an accumulator is to accumulate! The way accumulation is performed for a given accumulator is specified through `acc_info` declarations. These declarations usually contain a predicate (or a sequence of predicates) used for accumulation, that may take three external arguments:

- the two terms implementing the accumulator (`In,Out`);
- the data to be accumulated (`X`).

An `acc_info` declaration contains the name of the accumulator, references to these three terms, and the accumulation predicate:

```
acc_info(AccName,X,In,Out,acc_pred => AccPred)?
```

There are also other optional arguments to `acc_info` declarations which will be introduced later. The syntax for accumulation in a rule is the following:

`X + Acc`: accumulate `X` in accumulator `Acc`.

Example F.3

```
acc_info(fwd,X,In,Out,acc_pred => (Out = [X|In]))?
pred_info(foo,fwd)?

foo :-- 4+fwd, foo?
```

is translated into:

```
foo(in_fwd => A,out_fwd => B) :-
    C = [4|A],
    foo(in_fwd => C,out_fwd => B).
```

The expression in `acc_pred` just replaces the `X + Acc` goal, with the proper arguments instantiated.

F.1.3 Other features

- Initialization.

  An accumulator that has to be expanded and doesn't appear in the head of the clause, is initialized (this happens when `with` is used, see below). Initialization information may be given in the `acc_info` declaration (`in_start` and `out_start` features).

Example F.4

```
acc_info(einstein, in_start=>mc2, out_start=>energy)?
pred_info(t,einstein)?

s :-- t?
```

is translated into:

```
s :- t(in_einstein => mc2,out_einstein => energy).
```

as `einstein` doesn't appear in the head.

If `acc_info` is used several times to define the same accumulator, only the last declaration is taken into account. The system gives a warning.

- `X is Acc` unifies X with the current value of the accumulator Acc.

Example F.5 (with the previous declarations)

```
p :-- p, X is castor, write(X), Y is pollux, write(Y), p?
```

The obtained clause is:

```
p(in_castor => A,in_pollux => B,
  out_castor => C,out_pollux => D) :-
    p(in_castor => A,in_pollux => B,
      out_castor => E,out_pollux => F),
    G = E,
    write(G),
    H = F,
    write(H),
    p(in_castor => E,in_pollux => F,
      out_castor => C,out_pollux => D).
```

It is also possible to write:

- `Acc is X`: same meaning as `X is Acc`
- `Acc1 is Acc2`: unify current value of `Acc1` with current value of `Acc2`

Warning: the expansion fails if none of the arguments of `is` is an accumulator supposed to be expanded.

- `insert(X,Y,Acc)` inserts X and Y in the chain implementing Acc. This primitive is provided as an escape to standard LIFE code. It should not be needed in most cases.

Example F.6 (with the previous declarations)

```
r :-- r, insert(a,b,castor), r?
```

The obtained clause is:

```
r(in_castor => A,out_castor => B) :-
    r(in_castor => A,out_castor => C),
    a = C,
    b = D,
    r(in_castor => D,out_castor => B).
```

- `cond` and disjunctions.

  Expansion is performed inside `cond` and disjunctions.

- Code insertion.

  The preprocessor may be told not to expand a piece of code by inserting it between brackets.

Example F.7 (with the previous declarations)

```
p :-- q , {p}?
```

The obtained clause is:

```
p(in_castor => A,in_pollux => B,
  out_castor => C,out_pollux => D) :-
    q(in_castor => A,in_pollux => B,
      out_castor => C,out_pollux => D),
    p.
```

- Use of cut.

  Cuts may be used anywhere and are never expanded. "`!`" is in fact syntactic sugar for "`{!}`".

- Meta-programming.

  Variables may be used as symbols in the body of the rules:

Example F.8 (with the declarations above)

```
p(X) :-- X?
```

is translated into:

```
p(X,
  in_castor => B,in_pollux => C,
  out_castor => D,out_pollux => E) :-
    interpret_symbols(X,
                      @(castor => B,pollux => C),
                      @(castor => D,pollux => E),
                      @,
                      cname => default_C,
                      gram => false).
```

The features of `interpret_symbols` contain the necessary information to interpret `X` when it gets bound; `interpret_symbols` will residuate until `X` is no longer @. This way, if `X` is instantiated to some symbol `foo`, the first rule will behave exactly like:

```
p :-- foo?
```

A warning is given at expansion time to indicate that variables are expanded using `interpret_symbols`.

`interpret_symbols` is a non-strict predicate.

- Changing the argument names.

  If you want special names to be given to the arguments implementing an accumulator, you may specify them in the `acc_info` declaration:

  ```
  acc_info(box,in_name => "input",out_name => "output")
  ```

  With this declaration, `box` will be expanded using `input` and `output` instead of `in_box` and `out_box`.

F.2 Operations on accumulators

F.2.1 Context of an expansion

The tools we have described above don't give good answers to some practical questions such as:

- How can we specify rules like:

  ```
  head :- body1(in => In,out => Inter),
          body2(in => Inter,out => Out).
  ```

  The problem here is that the accumulator is not expanded in the head of the clause.

- How can we link different accumulators together? There is no reason why different accumulators always have to be isolated from each other.

In fact, the accumulators expanded in the head of the clause are very important in the expansion of the clause, because only these accumulators are linked (the others are just initialized). The `with` operator has been designed to let other sets of accumulators play the same role.

An expansion is characterized by two concepts:

- Its *scope*: the set of terms affected by the expansion.
- Its *context*: the set of accumulators that will be linked together during the expansion.

Until now, the scope of an expansion was always the whole clause, and its context the set of accumulators expanded in the head. `With` enables the programmer to define the scope and the context of an expansion. `With` is defined as an infix operator of precedence 800 and kind xfy, *i.e.*, it binds tighter than the control operators used in clauses ("`,`", "`;`", "`:-`", and so on), but looser than functional expressions.

Example F.9 Let us consider the following program:

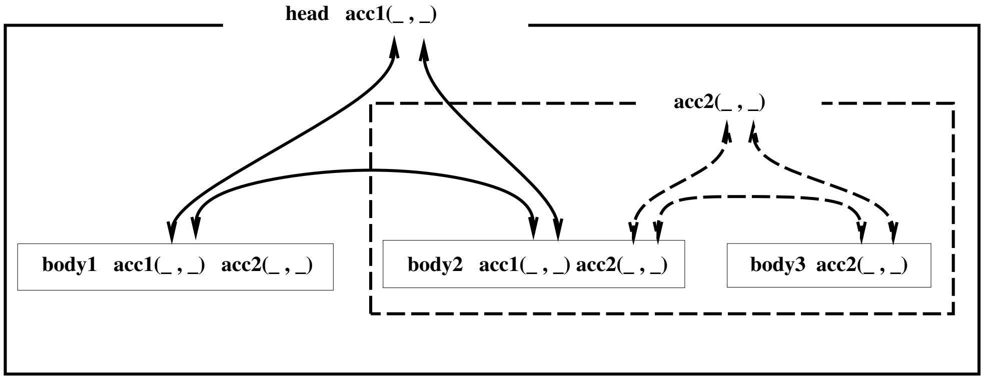
```
pred_info([head,body1,body2],acc1)?
pred_info([body1,body2,body3],acc2)?

head :-- body1, (body2,body3) with acc2.
```

The initial scope of the expansion is the whole clause, with context {`acc1`}: This means that `acc1` will be expanded and linked in `head`,`body1`,`body2`.

`With` defines a new context, {`acc2`}, with scope `(body2,body3)`: this means that `acc2` will be expanded and linked in `body2` and `body3`, but not linked with the arguments of `body1` since `body1` doesn't belong to the scope of this expansion.

This may be represented this way: The scopes are represented by the big rectangles, with the corresponding contexts on their upper edge. Each pair of expanded arguments is represented by a term `acc(_,_)` where the first argument is the input, and the second the output. The arrows represent the unified arguments.



The clause is thus expanded to:

```
head(in_acc1=>A,out_acc1=>B) :-
    body1(in_acc1=>A,out_acc1=>C,in_acc2=>@,out_acc2=>@),
    body2(in_acc1=>C,out_acc1=>B,in_acc2=>@,out_acc2=>D),
    body3(                       in_acc2=>D,out_acc2=>@).
```

When an accumulator appears in the right hand side of a `with` *and* in the parent context, the expansion of the accumulator inside and outside the scope of the `with` are totally independent: arguments appearing inside the scope of the `with` won't be linked with arguments appearing outside.

Example F.10

```
p :-- r, q with castor?
```

is translated into:

```
p(in_castor => A,in_pollux => B,
  out_castor => C,out_pollux => D) :-
    r(in_castor => A,out_castor => C),
    q(in_castor => E,in_pollux => B,
      out_castor => F,out_pollux => D).
```

The arguments implementing the accumulator `castor` in `q` are not linked with the others. `pollux` is regularly expanded.

F.2.2 Operations

The notion of context gives the possibility to perform operations on accumulators. In a context, an accumulator is represented by a pair of arguments `(In,Out)`. In the example above, the castor accumulator is represented by the pair `(A,C)` in the head context, and the pair `(E,F)` in the context defined by the `with` operator.

We define three operations on these pairs of arguments: composition, inversion, equality.

- Composition: =>

  `(A,B) => (C,D)` returns `(A,D)`; `B` and `C` are unified.

- Inversion: `inv`

  `inv((A,B))` returns `(B,A)`

- Equality: =

  `(A,B) = (C,D)` unifies `A` and `C`, and `B` and `D`.

Moreover, direct access to the pairs of arguments is given, which allows the accumulators to be initialized in a convenient way.

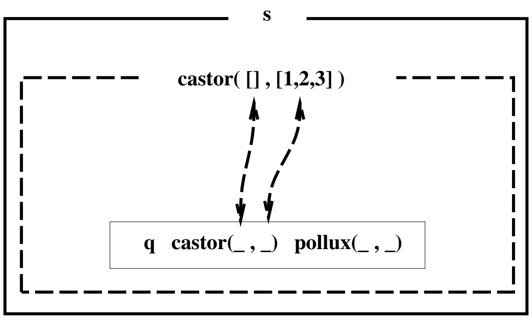
The right hand side of the `with` operator contains a conjunction of constraints to be applied to the new context: either simple constraints meaning: "expand this accumulator locally" or more complex like: "compose these two accumulators", or "make these accumulators equal". What follows is a set of example clauses and their translations.

- `s :-- q with castor([],[1,2,3])?`

  When arguments are given to accumulators in a scope of a `with`, the first one is unified with the input of the accumulator, the second with the output. The above clause is translated into:

  ```
  s :- q(in_castor => [],in_pollux => @,
         out_castor => [1,2,3],out_pollux => @).
  ```

  This may be represented graphically by:

- `s :-- q with castor => pollux?`

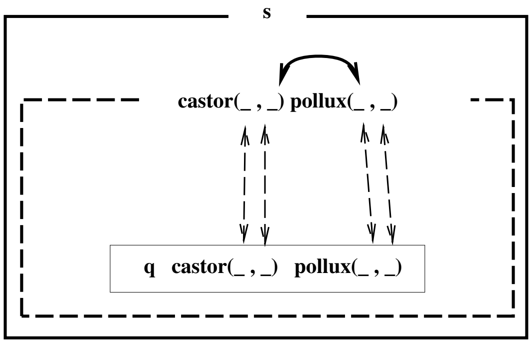
  "=>" is the composition operator. It links the output of its left hand side with the input of its right hand side:

  ```
  s :- q(in_castor => @,out_castor => A,
         in_pollux => A,out_pollux => @).
  ```

  The same clause could have been specified in the following way:

  ```
  s :-- q with (castor(\_,X),pollux(X,\_)).
  ```

Graphically:



The constraints imposed by the right argument are represented by solid arrows, the other links by dashed arrows.

- `s :-- q with inv(castor) => pollux?`

  "`inv`" inverts input and output arguments. In this example, you may notice that both inputs are linked together.

```
s :- q(in_castor => A,in_pollux => A,
       out_castor => @,out_pollux => @).
```

- `s :-- q with castor = pollux?`

  Equality of accumulators means unification of both input and output.

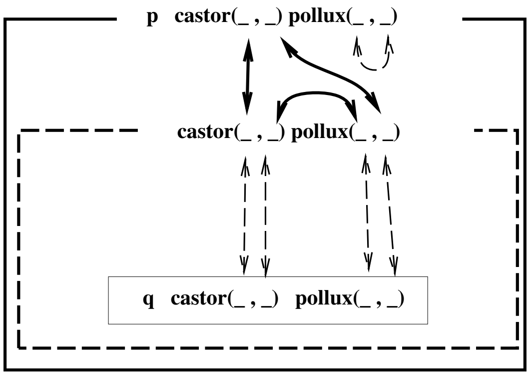
```
s :- q(in_castor => A,in_pollux => A,
       out_castor => B,out_pollux => B).
```

- `p :-- q with glob(castor) = castor => pollux?`

  `glob(castor)` is a reference to the `castor` accumulator appearing in the parent context.

```
p(in_castor => A,in_pollux => B,
  out_castor => C,out_pollux => B) :-
    q(in_castor => A,in_pollux => D,
      out_castor => D,out_pollux => C).
```

Graphically:



- All these things may be used together.

```
p :-- (q, r, s)
     with inv(glob(castor)) =
          castor(begin) => pollux(2 => end)?
```

is translated into:

```
p(in_castor => A: end,in_pollux => B,
  out_castor => C: begin,out_pollux => B) :-
      q(in_castor => C,in_pollux => D,
        out_castor => E,out_pollux => A),
      r(in_castor => E,out_castor => D),
      s.
```

- Initialization may also be performed at the context level.

  The initialization rule given above should be rewritten: An accumulator is initialized if it doesn't appear in the context of the expansion. But it is possible to force initialization, using `init`.

```
acc_info(einstein,in_start=>mc2,out_start=>energy)?
pred_info(t,einstein)?

s :-- t with init(einstein)?
```

is translated into:

```
s :- t(in_einstein => mc2,out_einstein => energy).
```

You may specify which argument has to be initialized by giving a second argument to `init`: `in` or `out`. For example, this clause:

```
s :-- t with init(einstein,in)?
```

is translated into:

```
s :- t(in_einstein => mc2,out_einstein => @).
```

This clause:

```
s :-- t with init(einstein,out)?
```

is translated into:

```
s :- t(in_einstein => @,out_einstein => energy).
```

If the second argument of `init` isn't `in` or `out`, then both are initialized.

F.3 The DCG accumulator

The accumulator preprocessor provides built-in definitions that mimic Prolog's DCG translation. There are a few differences with Prolog DCGs, mainly due to the use of ψ-terms instead of Herbrand terms. The main differences are that the arguments added have features `in_dcg` and `out_dcg` and that declarations given interactively must be terminated with `?`. If `expand_load(true)` is called, then declarations in a file may be terminated with "`.`".

Example F.11 This example shows how to use DCGs in Wild LIFE.

```
> import("accumulators")?
*** Loading File "Tools/accumulators.lf"
*** Loading File "Tools/std_expander.lf"
*** Loading File "Tools/acc_declarations.lf"

> a --> b, c?

> listing(a)?

a(in_dcg => _A, out_dcg => _B) :-
     b(in_dcg => _A, out_dcg => _C),
     c(in_dcg => _C, out_dcg => _B).

*** Yes
```

F.3.1 *Definition*

The DCG accumulator is predefined as follows:

```
acc_info( dcg, Term, Xs, Ys, acc_pred => 'C'(Term,false,Xs,Ys))?
```

The `'C'(Terms,FoldOk,Xs,Ys)` function is used for accumulation. Its arguments have the following meaning:

- `Terms`: the list representing the non-terminals to be recognized;
- `FoldOk` : a boolean telling whether `Terms` may be folded into the program or not;
- `Xs`: the input stream of non-terminals;
- `Ys`: the output stream of non-terminals.

You may replace the `'C'` function by your own version:

```
set_C(my_C)?
```

`'C'` is only used during the expansion of your grammar in LIFE clauses, so you can (and should) reset its value to the default after translation (with the query `reset_C?`) The tokenizer and parser written in LIFE (see Tools directory) redefine `'C'` for their own needs.

F.3.2 DCG syntax

The standard DCG syntax is supported by this preprocessor:

```
Head --> Body?
```

will expand the DCG accumulator in all predicates of `Head` and `Body` even if there is no `pred_info` declaration. The other accumulators will be expanded according to the declarations.

Accumulation in the DCG may be specified either with the above notation `(X + dcg)`, or using the standard list notation of DCGs:

```
foo --> [3], bar?
```

is translated into:

```
foo(in_dcg => [3|A],out_dcg => B) :-
    bar(in_dcg => A,out_dcg => B).
```

F.3.3 Implementation notes

- Folding Terminals.

  The DCG accumulator has been optimized to allow the folding of terminals (*i.e.*, terms to be accumulated in the DCG accumulator).

  A translation of: `foo --> bar, [1]?` could be:

  ```
  foo(in_dcg => A,out_dcg => B) :-
      bar(in_dcg => A,out_dcg => C),
      C = [1|B].
  ```

  The last statement may be folded into the bar predicate, yielding:

  ```
  foo(in_dcg => A,out_dcg => B) :-
      bar(in_dcg => A,out_dcg => [1|B]).
  ```

This translation is more efficient than the previous one, and the meaning of the two clauses are identical, as long as the bar predicate does not contain things like destructive assignments on its `out_dcg` feature.

When code is inserted — for instance a cut — we have to make sure that this folding does not bind variables occurring before the insertion. Consider the following clause:

```
foo --> !, [1]?
```

This first translation of it (with folding):

```
foo(in_dcg => [1|A],out_dcg => A) :-
     !.
```

does not have the same behavior as this one (without folding):

```
foo(in_dcg => A,out_dcg => B) :-
     !,
     A = [1|B].
```

The first translation is not correct w.r.t. the usual meaning of cut. The translator we propose here deals with this in a very simple way: no folding is performed after a cut or a code insertion. `FoldOk` is the variable used in the translator telling whether folding is authorized or not .

- Code Insertion.

  There are two ways of translating a rule like:

  ```
  foo --> {pred}?
  ```

  Either as:

  ```
  foo(in_dcg => A,out_dcg => B) :- pred, A = B.
  ```

  or as:

  ```
  foo(in_dcg => A,out_dcg => A) :- pred.
  ```

  In most cases, those two translations will have exactly the same behavior, but if some side-effect is expected from `pred`, they may differ. This is the case if `pred` is a cut for instance. The translation used here is the second one, namely:

  ```
  foo --> {pred}?
  ```

  is equivalent to:

  ```
  foo --> [], {pred}?
  ```

  The other alternative may be obtained by writing:

  ```
  foo --> {pred}, []?
  ```

F.4 Passed arguments

A passed argument is just an extra feature added to a predicate, according to some pred_info declaration. The use of passed arguments is less interesting since the introduction of global variables in LIFE.

```
pass_info(ball)?
```

declares ball as a passed argument. There are other optional arguments to this declaration that will be explained later.

To declare that a predicate uses this argument, just add ball to the list in pred_info.

pred_info(basket,[ball])?

The expansion will add a ball feature to all occurrences of basket. If it is not present in the head of the definition, it may be initialized: if the pass_info has a start feature, its value will be used to initialize the passed argument.

Example F.12

```
pass_info(kick,start => quick)?
pass_info(ball)?

pred_info(foot,[ball,kick])?
pred_info(basket,ball)?

basket :-- foot, basket?
```

is translated into:

```
basket(ball => A) :-
     foot(ball => A,kick => quick),
     basket(ball => A).
```

All rules related to expansion of disjunctions, cond, cuts, and insertion of code still hold. The is primitive may be also used with passed arguments. Passed arguments may be used in the left hand side of with operators, but no operations may be performed on them. Their value may be initialized by giving them an argument, but not (yet) using init.

pass_info declaration may also specify an "accumulation" predicate. This predicate can of course only take two arguments, the value to accumulate, and the value of the passed argument.

Example F.13

```
pass_info(hash,X,Pass,acc_pred => (Pass.X = true))?
pred_info(store,hash)?

store(X) :-- X+hash?
```

is translated into

```
store(A,hash => B) :-
    B.A = true.
```

F.5 Common problems and debugging

Debugging a program written using the accumulator expander is not an easy task. One of the most common mistakes is to forget the expansion of facts or to type `:-` instead of `:--`. To help the programmer, a directive may be added to the programs so that warnings are given each time a predicate with an associated `pred_info` declaration occurs in a non expanded rule. This directive is

```
check_expansion?
```

It only works with `expand_load(true)`, and for rules ending with a period.

Example F.14 Consider a file with the following declarations:

```
acc_info(acc)?
pred_info(pred,acc)?

pred :-- pred.
pred.
pred :- pred.
other_pred :- pred.
```

A warning will be generated for each of the last three clauses.

Example F.15 This example illustrates a common problem when mixing DCG predicates with non-DCG predicates. The problem occurs when the predicates contain other accumulators in addition to the DCG accumulator. The problem is solved through judicious use of the `with` operator. Consider the following definition of predicates `a` and `c`:

```
> import("accumulators")?
> acc_info(acc)?
> pred_info([a,b,c,d],acc)?

> c :-- r,s,t? % Non-DCG predicate

> a --> b,c,d? % DCG predicate
```

The expansion of `a` adds the two DCG arguments to `b`, `c`, and `d`:

```
> listing(a)?

a( in_acc => _A, in_dcg => _B,
  out_acc => _C,out_dcg => _D) :-
        b( in_acc => _A, in_dcg => _B,
          out_acc => _E,out_dcg => _F),
        c( in_acc => _E, in_dcg => _F,
          out_acc => _G,out_dcg => _H),
        d( in_acc => _G, in_dcg => _H,
          out_acc => _C,out_dcg => _D).
```

The problem is that `c` is not a DCG predicate, hence it does not know what to do with the two DCG arguments! The fix is to insulate `c` from the DCG accumulator by using `with`:

```
> a --> b, c with dcg, d?  % Correct definition
> listing(a)?

a( in_acc => _A, in_dcg => _B,
  out_acc => _C,out_dcg => _D) :-
        b( in_acc => _A, in_dcg => _B,
          out_acc => _E,out_dcg => _F),
        c( in_acc => _E, in_dcg => @,    % c is bypassed
          out_acc => _G,out_dcg => @),
        d( in_acc => _G, in_dcg => _F,
          out_acc => _C,out_dcg => _D).
```

The DCG arguments are now correctly chained between `b` and `d`, bypassing `c`.

F.6 Term expansion

The preprocessor includes term expansion clauses for `:--`, `-->`, `pred_info`, `acc_info` and `pass_info`. This means that if you execute `expand_load(true)`, then all rules loaded from files may be written as definitions instead of as queries, *i.e.*, ending with a period instead of a question mark.