Today's Agenda: Input-Output Issues

- Leftovers: The mystery of terminals on the LHS in DCGs
- Macros in Prolog
  - A general concept: hooks
  - A hook into printing: portray/1
  - A hook into reading of programs: term_expansion/2
- Some hints regarding input/output with your projects
  - Changing the output behavior of the top-level
  - A useful extension of print/1: format/2

The mystery of terminals on the LHS of DCGs

What is the significance of the LHS terminal in the following DCG rule?

```
aux, [not] --> [aint].                  v --> [walks].
```

Translation to Prolog:

```
aux(A, B) :- 'C'(A, aint, C),           v(A,B) :- 'C'(A,walks,B).
             'C'(B, not, C).
```

After unfolding of call to 'C'/3:

```
aux([aint|X],[not|X]).                  v([walks|X],X).
```

An example grammar

```
s  --> np, vp.

np --> [john].

vp --> aux, neg, v.

aux, [not] --> [aint].

neg --> [not].

v --> [leaving].
```

The example grammar compiled and a trace

```
s(A, B)  :-  np(A, C),
             vp(C, B).

np(A, B) :- 'C'(A, john, B).

vp(A, B) :-  aux(A, C),
             neg(C, D),
             v(D, B).

aux(A, B) :- 'C'(A, aint, C),
             'C'(B, not, C).

neg(A, B) :- 'C'(A, not, B).

v(A, B) :- 'C'(A, leaving, B).

'C'([A|B],A,B).
```

```
| ?- trace,s(X,[]).
     Call: s(_1,[]) ?
     Call: np(_1,_2) ?
     Call: 'C'(_1,john,_2) ?
     Exit: 'C'([john|_2],john,_2) ?
     Exit: np([john|_2],_2) ?
     Call: vp(_2,[]) ?
     Call: aux(_2,_3) ?
     Call: 'C'(_2,aint,_4) ?
     Exit: 'C'([aint|_4],aint,_4) ?
     Call: 'C'(_3,not,_4) ?
     Exit: 'C'([not|_4],not,_4) ?
     Exit: aux([aint|_4],[not|_4]) ?
     Call: neg([not|_4],_5) ?
     Call: 'C'([not|_4],not,_5) ?
     Exit: 'C'([not|_5],not,_5) ?
     Exit: neg([not|_5],_5) ?
     Call: v(_5,[]) ?
```

```
    Call: 'C'(_5,leaving,[]) ?
    Exit: 'C'([leaving],leaving,[]) ?
    Exit: v([leaving],[]) ?
    Exit: vp([aint,leaving],[]) ?
    Exit: s([john,aint,leaving],[]) ?
X = [john,aint,leaving] ?
```

The idea of a hook

A hook offers the opportunity to execute one or more user-supplied code fragments at some designated place in a program.

Hooks are a programming technique often used to provide flexibility around a common kernel of functionality, cf., e.g., emacs or user interface programming.

Hookable predicates in Prolog are introduced by a program like the following:

```
some_program(A,Z) :-
     some_predicate(A,X),
     give_user_a_chance(X,Y),
     more_stuff(Y,Z).

give_user_a_chance(X,Y) :-
     user_hook(X,Y),
     !.
give_user_a_chance(X,Y) :-
     default_action(X,Y).
```

where the user provides the definition of the hook `user_hook/2`.

A hook into printing: portray/1

- The standard output predicate `print/1` is a hookable predicate. The user-definable hook is called `portray/1`.
- `print/1` is the predicate one should normally call to produce output.
- `print/1` is used by the system for output at the top-level and in the debugger.
- If the argument to `print/1` is a non-variable then a call is made to the user defined predicate `portray/1`. If this succeeds then it is assumed the term has been output. Otherwise `print/1` is called recursively on the components of the term until the term is atomic, at which time it is written via basic `write/1`.

Using portray to print strings

Strings are encoded as lists of character codes:

```
| ?- print("abc").
[97,98,99]
```

Change the way lists of integers are printed:

```
portray([X|Y]) :-
   integer(X),
   format('"~s"',[[X|Y]]).
```

Resulting output:

```
| ?- print("abc").
"abc"
yes

| ?- print([97,98,99]).
"abc"
yes
```

Portray is called recursively for each subterm

```
portray(X) :- nl,write(*),write(X),write(-),nl,fail.

?- print(a(b,c(d,e),f)).

   *a(b,c(d,e),f)-
   a(
   *b-
   b,
   *c(d,e)-
   c(
   *d-
   d,
   *e-
   e),
   *f-
   f)
```

Lists as a special case

When printing a list, print/1 first gives the whole list to portray/1, then each element (instead of each subterm):

```
?- print([a,b,c,d,e,f]).
?- print(.(a,.(b,.(c,.(d,.(e,.(f,[]))))))).

     *[a,b,c,d,e,f]-
     [
     *a-
     a,
     *b-
     b,
     *c-
     c,
     *d-
     d,
     *e-
     e,
     *f-
     f]
```

The parallel compound term example

```
| ?- print(x(a,x(b,x(c,x(d,x(e,x(f,[]))))))).

     *x(a,x(b,x(c,x(d,x(e,x(f,[]))))))-
     x(
     *a-
     a,
     *x(b,x(c,x(d,x(e,x(f,[])))))-
     x(
     *b-
     b,
     *x(c,x(d,x(e,x(f,[]))))-
     x(
     *c-
     c,
     *x(d,x(e,x(f,[])))-
     x(
     *d-
     d,
     *x(e,x(f,[]))-
     x(
     *e-
     e,
     *x(f,[])-
     x(
     *f-
     f,
     *[]-
     []))))))
```

Term expansion

What is term expansion and when does it take place?

- Term expansion is a source-to-source transformation that takes place whenever a file is consulted or compiled.
- Term expansion can be called explicitly: `expand_term(+Term1,?Term2)`

How is the transformation carried out?

- The user-defined hook predicate `term_expansion/2` is called for each clause that is read in. If it fails, the default DCG expansion is applied.
- Different from `portray/1`, the `term_expansion/2` hook is only called for the clause itself, not for its parts.

Note:

- `term_expansion(?-(Query),?-(ExpandedQuery))` can be used to transform queries entered at the terminal in response to the `| ?-` prompt.
- Use `:- multifile user:term_expansion/2.` to avoid overwriting clauses defined in other files or manage term expansion (and portray) dynamically using `add_expansion/1` and `del_expansion/1` code as suggested by O'Keefe.

A simple example for term expansion

```
term_expansion((A :- B),(A :- B, write(B),nl)).

p :- q,
     r.
```

The result can be checked by calling `listing/0`:

```
p :-    q,
        r,
        write((q,r)),
        nl.
```

Using term expansion to translate explicitly

```
translate(InputFile,OutputFile) :-
     see(InputFile),
     tell(OutputFile),
     repeat,
         read(Term),
         expand_term(Term,Expansion),
         (  Expansion == end_of_file
         ;  portray_clause(Expansion),fail
         ),
     !,
     told,
     seen.
```

A simple macro facility using term expansion

```
% Rule
expand_clause((Head :- OldBody)) :-!,
   expand_body(OldBody, NewBody).
% Directive
expand_clause((:- OldBody), (:- NewBody)) :-!,
   expand_body(OldBody, NewBody).
% Query
expand_clause((?- OldBody), (?- NewBody)) :-!,
   expand_body(OldBody, NewBody).
% Fact
expand_clause(OldBody,NewBody) :-
   expand_body(OldBody,NewBody).

% Variable
expand_body(Var, call(Var)) :-
   var(Var), !.
% Conjunction
expand_body((OldA,OldB), Answer) :- !,
   expand_body(OldA, NewA),
   expand_body(OldB, NewB),
   get_rid_of_extra_true(NewA, NewB, Answer).
% Disjunction
expand_body((OldA;OldB), (NewA;NewB)) :- !,
   expand_body(OldA, NewA),
   expand_body(OldB, NewB).
```

```
% Implication
expand_body((OldA->OldB), (NewA->NewB)) :- !,
   expand_body(OldA, NewA),
   expand_body(OldB, NewB).
% Forall
expand_body(forall(OldA,OldB), forall(NewA,NewB)) :- !,
   expand_body(OldA, NewA),
   expand_body(OldB, NewB).
% Negation
expand_body(\+(Old), \+(New)) :- !,
   expand_body(Old, New).
% And finally:  macro application
expand_body(Old, New) :-
   macro(Old, New),
   !.       % FORCE a unique expansion.
expand_body(Old, Old).  % Not a macro.

% remove true conjuncts:
get_rid_of_extra_true(true, X, X) :- !.
get_rid_of_extra_true(X, true, X) :- !.

% Now we're ready to insert it into term_expansion/1
term_expansion(X,Y) :-
   expand_clause(X,Y).
```

```
% Example of macro use: eliminate overhead of
% calling field access predicates

macro(context(Context, A, B, C, D),
      Context=context(A,B,C,D)).
macro(context_a(Context,A),
      Context=context(A,_,_,_)).
macro(context_b(Context,B),
      Context=context(_,B,_,_)).
macro(context_c(Context,C),
      Context=context(_,_,C,_)).
macro(context_d(Context,D),
      Context=context(_,_,_,D)).

% Example:
c :- context(Context,1,2,3,4),
     p(Context).

p(Context):- context_a(Context,A),write(A).
p(Context):- context_b(Context,B),write(B).

% This expands to:
% c :- A=context(Context,1,2,3,4),
%      p(A).
%
% p(X):- X=context(A,B,C,D),write(A).
% p(X):- X=context(B,B,C,D),write(B).
```

More example macros and their use

```
macro(cons(H, T, [H|T]),   true).
macro(head([H|T], H),      true).
macro(tail([H|T], T),      true).
macro(empty([]),           true).
macro(positive(X),         X>0).

append(Prefix, Suffix, Answer) :-
   head(Prefix, Head),
   tail(Prefix, Tail),
   cons(Head, Rest, Answer),
   append(Tail, Suffix, Rest).
append(Prefix, Answer, Answer) :-
   empty(Prefix).

member(Element, List) :-
   head(List, Element).
member(Element, List) :-
   tail(List, Rest),
   member(Element, Rest).

greater(X, Y) :-
   Z is Y-Z,
   positive(Z).
```

Changing the output behavior of the top level

To change the print depth and other properties of the printing of results on the top-level use `prolog_flag/3`.

Example changing `max_depth` to 100:

```
?- prolog_flag(toplevel_print_options,
               _Old,
               [quoted(true),
                numbervars(true),
                portrayed(true),
                max_depth(100)]).
```

To do the same for the debugger, use `debugger_print_options` instead of `toplevel_print_options`.

A useful extension to `print/1`: `format/2`

`format(+Format,+Arguments)` prints `Arguments` according to format `Format`.

- `Format` is an atom, a list of character codes, or special formatting characters.
- `Arguments` is a list of items to be printed.

```
?- format("Hello world!", []).
?- format('Hello world!', []).
```

The character ~ introduces a control sequence, e.g., `~n` for newline

```
?- format('~nHello world!~n', []).
```

is equivalent to

```
?- nl, write('Hello word!'), nl.
```

Control character `~p` prints the next argument in the list:

```
?- format('var 1: ~p, var 2: ~p', [one,two]).

var 1: one, var 2: two
```

Many other control characters are available (see manual).