```
% COMPILING PROLOG INTO ENGINE BYTE-CODE        (ENG-COMPILER.PL)

:-op(700,xfx,=#).

:-public cook/1.

cook(File) :-
   name(File,Name),
   concatenate(Name,".PL", Name_PL),  name(Input,Name_PL),   see(Input),
   concatenate(Name,".ENG",Name_ENG), name(Output,Name_ENG), tell(Output),
   repeat,
   read(C),
 ( C = end_of_file, !;
      compile(C,D),
      encode(D,E),
      emit_code(E), nl, nl,
      fail ),
   seen, told.

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

test :-
   repeat,
   read(C),
   compile(C,D),
   write_list(D), nl,
   encode(D,E),
   emit_code(E), nl, nl,
   fail.

write_list([]).
write_list([X|L]) :- write(X), nl, write_list(L).

compile(Clause,Instructions) :-
   preprocess(Clause,Clause1),
   trans_clause(Clause1,Symbols,[]),
   number_variables(Symbols,0,N,Saga),          % This part
   complete_saga(0,N,Saga),                     % completes the code
   allocate_registers(Saga),                    % for variable occurrences.
   generate(Symbols,Instructions).

preprocess(Clause,Clause).

optimise(Instructions,Instructions).


% TRANSLATING A CLAUSE INTO A SYMBOL LIST

trans_clause((Head :- Body)) --> !,
   trans_head(Head),
   [succeed],
   trans_body(Body),
   [exit].
trans_clause(Head) -->
   trans_head(Head),
   [proceed].

trans_head(Head) --> {functor(Head,P,N)},
   [pred(head,P/N)],
   trans_args(N,Head,head).

trans_body((Goal1,Goal2)) --> !,
   trans_body(Goal2),
   trans_body(Goal1).
trans_body(Goal) --> {functor(Goal,P,N)},
   trans_args(N,Goal,body),
   [pred(body,P/N)].

trans_args(0,_,_) --> !, [].
trans_args(N,Args,body) --> !, {arg(N,Args,Arg), N1 is N-1},
   trans_arg(Arg,body),
   trans_args(N1,Args,body).
trans_args(N,Args,Context) --> {arg(N,Args,Arg), N1 is N-1},
   trans_args(N1,Args,Context),
   trans_arg(Arg,Context).

trans_arg(Var,Context) --> {var(Var)}, !,
   [var(Context,Var,State,Occ,Perishability)].
trans_arg(Const,Context) --> {atomic(Const)}, !,
   [const(Context,Const)].
trans_arg(Struct,Context) --> {functor(Struct,F,N), root(Context,Context0)},
   [functor(Context,F/N)],
   trans_args(N,Struct,struct(Context0)),
   [resume(Context)].

root(struct(Context),Context) :- !.
root(Context,Context).


% NUMBERING THE VARIABLES IN A CLAUSE

number_variables([],N,N,[]).
number_variables([S|SS],I,N,[S|SS1]) :- var_symbol(S,Var), !,
   number_variable(Var,I,I1),
   number_variables(SS,I1,N,SS1).
number_variables([S|SS],I,N,SS1) :-
   number_variables(SS,I,N,SS1).

number_variable(num(I,_),I,I1) :- !, I1 is I+1.
number_variable(_,I,I).


% COMPLETING THE VARIABLE OCCURRENCES

complete_saga(I,I,Bio) :- !, complete_bio(Bio,undef,_).
complete_saga(L,N,Saga) :- M is (L+N)/2, M1 is M+1,
   split(Saga,M,Saga1,Saga2),
   complete_saga(L,M,Saga1),
   complete_saga(M1,N,Saga2).

split([],_,[],[]).
split([S|SS],M,[S|SS1],SS2) :- var_number(S,I), I =< M, !,
   split(SS,M,SS1,SS2).
split([S|SS],M,SS1,[S|SS2]) :-
   split(SS,M,SS1,SS2).

complete_bio([],_,none).
complete_bio([var(Context0,_,State0,Occ,Perishability)|SS],State0,Context1) :-
   root(Context0,Context1),
   affect(Context0,State0,State),
   occurrence(State0,SS,Occ),
   complete_bio(SS,State,Context),
   perishability(Context0,State,Context,Perishability).

perishability(head,local,body,perishable) :- !.
perishability(_,_,_,ok).

affect(struct(_),_,global) :- !.
affect(_,undef,local) :- !.
affect(_,State,State).

occurrence(undef,[],void) :- !.
occurrence(undef,_,first) :- !.
occurrence(_,[],last) :- !.
occurrence(_,_,middle).

% ALLOCATING VARIABLES TO REGISTERS

allocate_registers(Saga) :-
   reverse(Saga,[],Saga1),
   fix_regs(Saga1,1).

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

fix_regs([],_).
fix_regs([S|SS],Free) :-
   occ_and_reg(S,Occ,R),
   fix_reg(Occ,R,Free,Free1),
   fix_regs(SS,Free1).

fix_reg(last,R,Free,Free1) :- get_reg(Free,R,Free1).
fix_reg(first,R,Free,Free1) :- put_reg(Free,R,Free1).
fix_reg(void,0,Free,Free).
fix_reg(middle,_,Free,Free).

get_reg([R|Free],R,Free) :- !.
get_reg(R,R,R1) :- R1 is R+1.

put_reg(Free,R,[R|Free]).


% COMPONENTS OF A VARIABLE OCCURRENCE

occ_and_reg(var(_,num(_,R),_,Occ,_),Occ,R).

var_number(var(_,num(I,_),_,_,_),I).

var_symbol(var(_,Var,_,_,_),Var).


% GENERATING INSTRUCTIONS

generate([],[]).
generate([pred(body,Pr),exit], [execute + hp(Pr)]) :- !.
% generate([S|SS], Instrs) :- noop(S), !, generate(SS,Instrs).
```

```
generate([S|SS], [Instr|Instrs]) :-
   compile_symbol(S,Instr), generate(SS,Instrs).

compile_symbol(var(Context,num(_,R),State,Occ,Perishability), Instr) :-
   compile_var(Context,Occ,State,Perishability,R, Instr).
compile_symbol(const(Context,Const), Instr) :-
   compile_const(Context,Const, Instr).
compile_symbol(functor(Context,Fn), Instr) :-
   compile_functor(Context,Fn, Instr).
compile_symbol(resume(Context), Instr) :-
   compile_resume(Context, Instr).
compile_symbol(pred(Context,Pr), Instr) :-
   compile_pred(Context,Pr, Instr).
compile_symbol(succeed, succeed).
compile_symbol(proceed, proceed).
compile_symbol(exit, exit).

compile_var(head,void,_,_,R,            pop_void                ) :- !.
compile_var(head,first,_,perishable,R,  pop_perishable_var/R    ) :- !.
compile_var(head,first,_,ok,R,          pop_var/R               ) :- !.
compile_var(head,_,_,perishable,R,      pop_perishable_val/R    ) :- !.
compile_var(head,_,_,ok,R,              pop_val/R               ) :- !.
compile_var(body,void,_,_,R,            push_void               ) :- !.
compile_var(body,first,_,_,R,           push_var/R              ) :- !.
compile_var(body,_,_,_,R,               push_val/R              ) :- !.
compile_var(struct(_),void,_,_,R,       unify_void              ) :- !.
compile_var(struct(_),first,_,_,R,      unify_var/R             ) :- !.
compile_var(struct(_),_,global,_,R,     unify_global_val/R      ) :- !.
compile_var(struct(_),_,local,_,R,      unify_val/R             ) :- !.

compile_const(head,C,                   pop_const + wa(C)       ) :- !.
compile_const(body,C,                   push_const + wa(C)      ) :- !.
compile_const(struct(_),C,              unify_const + wa(C)     ) :- !.

compile_functor(head,F,                 pop_struct + hf(F)      ) :- !.
compile_functor(body,F,                 push_struct + hf(F)     ) :- !.
compile_functor(struct(_),F,            unify_struct + hf(F)    ) :- !.

compile_resume(head,                    resume_head             ) :- !.
compile_resume(body,                    resume_body             ) :- !.
compile_resume(struct(head),            resume                  ) :- !.
compile_resume(struct(body),            resume_copy             ) :- !.

compile_pred(head,Pr,                   wc(Pr)                  ) :- !.
compile_pred(body,Pr,                   push_pred + hp(Pr)      ) :- !.


% ENCODING INSTRUCTIONS

encode([],[]).
encode([Instr|Instrs],Code) :-
   encode_instr(Instr,Code,Code1),
   encode(Instrs,Code1).

encode_instr(wc(Pr),[wc(Pr)|Code],Code) :- !.
encode_instr(Op+Arg,[Opcode,Arg|Code],Code) :- !, Op =# Opcode.
encode_instr(Op/R,[Opcode|Code],Code) :- !, Op =# Opcode0,
   Opcode is Opcode0+R-1.
encode_instr(Op,[Opcode|Code],Code) :- Op =# Opcode.
```

```
emit_code([]).
emit_code([X|L]) :- emit_item(X), emit_code(L).

emit_item(wc(P/N)) :- !,
   write('WC '), write(N), put(" "), write(P), nl.
emit_item(hp(P/N)) :- !, nl,
   write('HP '), write(N), put(" "), write(P), nl.
emit_item(hf(F/N)) :- !, nl,
   write('HF '), write(N), put(" "), write(F), nl.
emit_item(wa(C)) :- !, nl,
   write('WA '), write(C), nl.
emit_item(I) :- put(" "), write(I).

proceed                 =#         0.
succeed                 =#         1.
resume_head             =#         2.
resume_body             =#         3.
resume                  =#         4.
resume_copy             =#         5.
pop_struct              =#         6.
unify_struct            =#         7.
pop_const               =#         8.
unify_const             =#         9.
pop_void                =#        10.
unify_void              =#        11.

pop_var                 =#       144.
pop_perishable_var      =#       160.
pop_val                 =#       176.
pop_perishable_val      =#       192.
unify_var               =#       208.
unify_val               =#       224.
unify_global_val        =#       240.


exit                    =#         0.
execute                 =#         1.
push_pred               =#         2.
push_struct             =#         3.
push_const              =#         4.
push_void               =#         5.

push_var                =#       224.
push_val                =#       240.
```