```
% KG

% PROGOL COMPILER MASTER MODULE - MC68000 VERSION

:-op(1000,xfy,:).               % procedure header
:-op(1001,fx,let).              % labels macro definition
:-op(800,xfx,else).             % if check fails
:-op(700,yfx,:=).               % assignment
:-op(700,xfx,\=).               % not equal to
:-op(500,yfx,!).                % indexed by
:-op(500,fx,--).                % preindex
:-op(500,xf,++).                % postindex
:-op(250,fx,@).                 % address of
:-op(100,xfy,.).                % slice application
:-op(100,fx,#).                 % labels constant
:-op(50,yfx,``).                % long integer constructor
:-op(50,yfx,..).                % identifier conactenation

% Progol compiler modules:
%       kg              % this module;
:-[
        kgadmn,         % general administration;
        kgpre,          % pre-processing;
        kggoal,         % rewriting goals;
        kg68,           % transliteration of Progol to 68000 code;
        kg68i,          % 68000 instruction repertoire;
        kg68c           % output of 68000 code;
].
```

```
% KGCOMP

% COMPILING THE PROGOL COMPILER - MC68000 version

:-op(1000,xfy,:).                % procedure header
:-op(1001,fx,let).               % labels macro definition
:-op(800,xfx,else).              % if check fails
:-op(700,yfx,:=).                % assignment
:-op(700,xfx,\=).                % not equal to
:-op(500,yfx,!).                 % indexed by
:-op(500,fx,--).                 % preindex
:-op(500,xf,++).                 % postindex
:-op(250,fx,@).                  % address of
:-op(100,xfy,.).                 % slice application
:-op(100,fx,#).                  % labels constant
:-op(50,yfx,``).                 % long integer constructor
:-op(50,yfx,..).                 % identifier concatenation

% Progol compiler modules:
%       kgcomp                % this module;
:-compile([
        kgadmn,         % general administration;
        kgpre,          % pre-processing;
        kggoal,         % rewriting goals;
        kg68,           % transliteration of Progol to 68000 code;
        kg68i,          % 68000 instruction repertoire;
        kg68c           % output of 68000 code;
]).
```

```
% KGADMN

% GENERAL ADMINISTRATIVE CHORES

:-public progol/1, process/0.

progol(File) :-
   name(File,CC),
   (concatenate(CC0,[46|_],CC) | CC0=CC), !,
   concatenate(CC0,".MAC",CC1),
   name(OutputFile,CC1),
   see(File), tell(OutputFile),
   begin_progol,
   process,
   end_progol,
   seen, told.

begin_progol.

end_progol :-
   write('.end'), nl.

process :-
   repeat,
   read(C),
   process(C),
   C=end_of_file, !.

process(end_of_file) :- !.
process((let(D))) :- !, define(D).
process([C|CC]) :- !,
   (C =:= " " | put(12) ), !,    % optional form feed
   put(";"), put(" "), putcomment([C|CC]).
process(psect(Name,N)) :- !,
   write('   .psect '), write(Name), put(","), write(N), nl, nl.
process(align(N)) :- !,
   write('   .align '), write(N), nl, nl.
process(for(I,L,N,C)) :- !,
  (range(I,1,N),
      process(C),
      fail
  |true).
process(C) :-
   expand(C,C0),
  (C0 = (Pr:D), is_data(D), VAXCode = C0
  |   c_proc(C0,Code0),
      flatten_code(Code0,Code,end),
      map(Code,1,VAXCode)),
   !,
   putinstr(VAXCode), nl.
process(X) :- write('; ** ERROR **'), nl.

is_data(words(_)).
is_data(bytes(_)).
```

```
flatten_code(true,S,S) :- !.
flatten_code((Code1,Code2),S0,S) :- !,
   flatten_code(Code1,S0,S1),
   flatten_code(Code2,S1,S).
flatten_code((P:Code),(P:S0),S) :- !,
   flatten_code(Code,S0,S).
flatten_code(Instr,(Instr,S),S).

%  print_code(end) :-
%     write('   '), write(end), put("."), nl.
%  print_code((Line,Code)) :-
%     write('   '), write(Line), put(","), nl,
%     print_code(Code).
%  print_code((Line:Code)) :-
%     write(Line), put(":"), nl,
%     print_code(Code).

putcomment("") :- nl, nl.
putcomment([C|CC]) :-
   put(C),
   (C =\= 31 | put(";"), put(" ")), !,
   putcomment(CC).

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

range(I,L,N) :- L < N, !,
   M is (L+N)/2, M1 is M+1,
   (range(I,L,M) | range(I,M1,N)).
range(I,I,I).
```

```
% KGPRE

% PRE-PROCESSING

expand(X,X) :- ( var(X); integer(X) ), !.
expand((P:Q),(P:Q1)) :- !, expand(Q,Q1).
expand((P,Q),(P1,Q1)) :- !, expand(P,P1), expand(Q,Q1).
expand((P->Q|R),(P1->Q1|R1)) :- !,
   expand(P,P1), expand(Q,Q1), expand(R,R1).
expand(X,X1) :- recorded(X,X=X0,_), !, expand(X0,X1).
expand(X,X1) :-
   functor(X,F,N),
   functor(X1,F,N),
   expanda(1,N,X,X1).

expanda(I,N,T,T1) :- I =< N, !,
   arg(I,T,X),
   arg(I,T1,X1),
   expand(X,X1),
   I1 is I+1,
   expanda(I1,N,T,T1).
expanda(I,N,_,_) :- I > N.

define((D1,D2)) :- !, define(D1), define(D2).
define(A=B) :- recordz(A,A=B,_).
```

```
% KGGOAL

% COMPILING GOALS

c_proc((Pr:Goals), (Pr:Code) ) :-
   c_goal(Goals,goto(exit),goto(fail),Code).

% c_goal(
%        + Goal, :the goal (or goals) to be executed,
%        + Then, :where to do if the goal succeeds,
%        + Else, :where to do if the goal fails,
%        - Code  :the code for executing the goal.
%        )

c_goal(true,Then,_, Then ) :- !.

c_goal((Pr:Goals),Then,Else, (Pr:Code) ) :- !, c_goal(Goals,Then,Else,Code).

c_goal((G1,G2),Then,Else, (Code1,Code2,Label) ) :- !,
   reroute(Else,Label,Else1),
   c_goal(G1,true,Else1,Code1),
   c_goal(G2,Then,Else,Code2).

c_goal((G1->G2|true),Then,Else, (Code1,Code2,Label) ) :- !,
   reroute(Then,Label,Then1),
   c_goal(G1,true,Then1,Code1),
   c_goal(G2,Then,Else,Code2).

c_goal((G1->true|G2),Then,Else, (Code1,Code2,Label) ) :- !,
   reroute(Then,Label,Then1),
   c_goal(G1,Then1,true,Code1),
   c_goal(G2,Then,Else,Code2).

c_goal((G1->G2|G3),Then,Else, (Code1,Code2,label(L):Code3,Label) ) :-!,
   c_goal(G1,true,goto(label(L)),Code1),
   reroute(Then,Label,Then1),
   c_goal(G2,Then1,Else,Code2),
   c_goal(G3,Then,Else,Code3).

reroute(true,(label(L):true),goto(label(L))).
reroute(goto(L),true,goto(L)).

c_goal(E,Then,true,Code) :-
   negate_goal(E,E1), !,
   Then\==true,
   c_goal(E1,true,Then,Code).

c_goal(E,Then,Else, (Code,Then) ) :-
   evaluable_goal(E), !,
   c_expr(E,Else,Code).

c_goal(call_external(N,P),_,_, call_external(N,P) ) :- !.

c_goal(goto(Pr),_,_, goto(Pr) ) :- !.
c_goal(call(Pr),goto(exit),_, goto(Pr) ) :- !.

c_goal(call(Pr),Then,_, (call(Pr),Then) ) :- !.

c_goal(Pr,Then,Else,Code) :- c_goal(call(Pr),Then,Else,Code).

c_expr(E,_,E) :- non_boolean_goal(E), !.
c_expr(E,goto(L), E else L).

non_boolean_goal(_ := _).

evaluable_goal(_ := _).
evaluable_goal(_ = _).
evaluable_goal(_ \= _).
evaluable_goal(_ < _).
evaluable_goal(_ >= _).
evaluable_goal(_ > _).
evaluable_goal(_ =< _).

negate_goal(X=Y,X\=Y).
negate_goal(X\=Y,X=Y).
negate_goal(X<Y,X>=Y).
negate_goal(X>=Y,X<Y).
negate_goal(X>Y,X=<Y).
negate_goal(X=<Y,X>Y).
```

```
% KG68

% TRANSLATING PRIMITIVE PROGOL INTO 68000 INSTRUCTIONS

map(end,L,end).
map((Pr:Goals),L,(Pr:Code)) :- !, fix_label(Pr,L,L1), map(Goals,L1,Code).
map((E:=E,Goals),L,Code) :- !, map(Goals,L,Code).
map((Goal1,Goal2,Goals),L,(Code,Code1)) :-
   optimise(Goal1,Goal2,Goals,Goals1,Code), !, map(Goals1,L,Code1).
map((Goal,Goals),L,(Code1,Code2)) :- !, map1(Goal,Code1), map(Goals,L,Code2).

fix_label(label(L),L,L1) :- !, L1 is L+1.
fix_label(Pr,L,L).

optimise( E := E1,
          Test else Label,
          Goals,  Goals,                (Instr,
                                        $(BranchOp,Label))
) :-
   comparison(Test,F,E,0), !,
   map1( E := E1, Instr),
   negated_op(F,F1),
   m68branch(F1,BranchOp).

map1( E1 := E,                          $(ArithqOp,#C,X1)
) :-
   arithexpr(E,F,E1,C), quickconstant(C),
   m68arith1(F,T,ArithqOp), !,
   m68operand(E1,T,X1).

map1( E2 := E,                          $(ArithOp,X1,X2)
) :-
   arithexpr(E,F,E2,E1), !,
   m68operand(E1,T,X1),
   m68operand(E2,T,X2),
   m68arith(F,T,ArithOp).

map1( E2 := @E1,                        $(lea,X1,X2)
) :-
   m68operand(E1,T,X1), !,
   m68addrreg(E2,32,X2).

map1( E := 0,                           $(ClearOp,X)
) :- !,
   m68operand(E,T,X),
   m68clear(T,ClearOp).

map1( E2 := E1,                         $(MoveOp,X1,X2)
) :- !,
   m68operand(E1,T,X1),
   m68operand(E2,T,X2), valid_type(T), !,
   m68move(T,MoveOp).

map1( E2 := E1,                         $(MoveOp,X1,X2)
) :- !,
   m68operand(E1,16,X1),
   m68operand(E2,32,X2), !,
   m68move(16,MoveOp).

map1( E2 := E1,                              (,$(MoveOp,X1,X2),
                                               $(ExtOp,X2))
) :- !,
   m68operand(E1,T1,X1),
   m68datareg(E2,T2,X2),
   m68extend(T1,T2,ExtOp), !,
   m68move(T,MoveOp).

map1( Test else Label,                       Code
) :-
   comparison(Test,F,E1,E2), !,
   negated_op(F,F1),
   maptest(F1,E1,E2,Label,Code).

maptest(F,(I,1).E,0,Label,                   ($(btst,#I,X),
                                              $(BranchOp,Label))
) :-
   equality_op(F),
   m68datareg(E,32,X), !,
   m68branch(F,BranchOp).

maptest(F,E1,0,Label,                        ($(TestOp,X1),
                                              $(BranchOp,Label))
) :- !,
   m68operand(E1,T,X1),
   m68test(T,TestOp),
   m68branch(F,BranchOp).

maptest(F,E1,E2,Label,                       ($(TestOp,X1,X2),
                                              $(BranchOp,Label))
) :-
   m68datareg(E1,T,X1),
   m68operand(E2,T,X2),
   m68compare(T,TestOp),
   m68branch(F,BranchOp).

map1( call(Pr),                              $(CallOp,Pr)
) :-
   m68address_type(Pr,T),
   m68call(T,CallOp).

map1( goto(exit),                            $(rts)
) :- !.

map1( goto(Pr),                              $(GotoOp,Pr)
) :-
   m68address_type(Pr,T),
   m68goto(T,GotoOp).
```

```
map1( call_external(N,Pr),              $(pushs,#N,Pr)
).

m68address_type({X},jump).
m68address_type(N,branch).

m68operand({X},32,X) :- !.
m68operand(T.{X},T,X) :- !.
m68operand(N,_,#N).              %  :- is_constant(N), !.

m68reg(E,T,X) :- m68datareg(E,T,X).
m68reg(E,T,X) :- m68addrreg(E,T,X).

m68dtatreg(d0).
m68dtatreg(d1).
m68dtatreg(d2).
m68dtatreg(d3).
m68dtatreg(d4).
m68dtatreg(d5).
m68dtatreg(d6).
m68dtatreg(d7).

m68addrreg(a0).
m68addrreg(a1).
m68addrreg(a2).
m68addrreg(a3).
m68addrreg(a4).
m68addrreg(a5).
m68addrreg(a6).

valid_type(32).
valid_type(16).
valid_type( 8).

quickconstant(N) :- integer(N), 0 < N, N < 9.

% is_constant(label(L)) :- !.
% is_constant(N1``N2) :- !.
% is_constant(N) :- atomic(N).

arithexpr(X+Y,+,X,Y).
arithexpr(X-Y,-,X,Y).
arithexpr(X*Y,*,X,Y).
arithexpr(X/Y,/,X,Y).

comparison(X=Y,=,X,Y).           comparison(X\=Y,\=,X,Y).
comparison(X<Y,<,X,Y).           comparison(X>=Y,>=,X,Y).
comparison(X>Y,>,X,Y).           comparison(X=<Y,=<,X,Y).

negated_op(=,\=).                negated_op(\=,=).
negated_op(<,>=).                negated_op(>=,<).
negated_op(>,=<).                negated_op(=<,>).

equality_op(=).                  equality_op(\=).
```

```
% KG68I

% MC68000 INSTRUCTION REPERTOIRE

m68move( 32, movl).
m68move( 16, movw).
m68move(  8, movb).

m68extend( 16, 32, extl).
m68extend(  8, 16, extw).

m68clear( 32, clrl).
m68clear( 16, clrw).
m68clear(  8, clrb).

m68test( 32, tstl).
m68test( 16, tstw).
m68test(  8, tstb).

m68compare( 32, cmpl).
m68compare( 16, cmpw).
m68compare(  8, cmpb).

m68branch( =, jeq).                     m68branch( \=, jne).
m68branch( <, jlt).                     m68branch( >=, jge).
m68branch( >, jgt).                     m68branch( =<, jle).

m68arithq( +, 32, addql).               m68arith1( -, 32, subql).
m68arithq( +, 16, addqw).               m68arith1( -, 16, subqw).
m68arithq( +,  8, addqb).               m68arith1( -,  8, subqb).

m68arith( +, 32, addl).                 m68arith( -, 32, subl).
m68arith( +, 16, addw).                 m68arith( -, 16, subw).
m68arith( +,  8, addb).                 m68arith( -,  8, subb).

m68goto(branch,jra).                    m68call(branch,jbsr).
m68goto(jump,jra).                      m68call(jump,jbsr).
```

```
% KG68C

% OUTPUT OF MC68000 MACHINE CODE (A68 ASSEMBLER)

putentry(Pr) :-
   write('.entry '), putconst(Pr), put(","), putconst(2'111111111100), nl.

putinstr((entry(Pr):Code)) :- !, putentry(Pr), putinstr(Code).
putinstr((Pr:Code)) :- putconst(Pr), put(":"), nl, putinstr(Code).
putinstr((Instr,Code)) :- putinstr(Instr), putinstr(Code).
putinstr(end).
putinstr($(Op)) :-
   write('   '), write(Op), nl.
putinstr($(Op,X)) :-
   write('   '), write(Op), put(" "), putoperand(X), nl.
putinstr($(Op,X1,X2)) :-
   write('   '), write(Op), put(" "), putoperand(X1), put(","),
                                        putoperand(X2), nl.
putinstr(words(WW)) :- is_list(WW), !, putwords(WW).
putinstr(bytes(BB)) :- is_list(BB), !, putbytes(BB).
putinstr(words(N)) :- write('   .blkl '), putconst(N), nl.
putinstr(bytes(N)) :- write('   .blkb '), putconst(N), nl.

is_list([]).
is_list([_|_]).

putwords([W|WW]) :-
   write('   .long '), putconst(W), nl, putwords(WW).
putwords([]).

putbytes([B|BB]) :-
   write('   .byte '), putconst(B), nl, putbytes(BB).
putbytes([]).

putoperand(#N) :- !, put("#"), putconst(N).
putoperand(D+{R}!{R1}) :- !,
   putreg(R), put("@"), put("("), putconst(D), put(","),
                                  putreg(R1), put(":"), put("l"), put(")").
putoperand({R}) :- !, putreg(R), put("@").
putoperand(--{R}) :- !, putreg(R), put("@"), put("-").
putoperand({R}++) :- !, putreg(R), put("@"), put("+").
putoperand(D+{R}) :- !, putreg(R), put("@"), put("("), putconst(D), put(")").
putoperand(X) :- putconst(X).

putreg(R) :- atom(R), !, write(R).
putreg(R) :- write(R).

putconst(label(N)) :- !, write(N), put("$").
putconst(F..N) :- !, putconst(F), write(N).
putconst(N1``N2) :- !, put("^"), put("X"), puthex(N1), puthex(N2).
putconst(C) :- write(C).

puthex(N) :- puthex(4,N).
puthex(0,N) :- !.
puthex(I,N) :-
   D is (N/\15), (D < 10, !, H is "0"+D | H is "A"+D-10),
   N1 is (N>>4), I1 is I-1,
   puthex(I1,N1), put(H).
```

```
| BENCH68.S : 68000 Engine II Benchmark

        .data
conc:   .word clause,clause,clause,clause
                                                              8
clause: .word glistA1               | conc([                  3
        .word uvarX4                |      X|                 3
        .word uvarX1                |      L1],L2,           15        Save
        .word glistA3               |     [                   4          2
        .word uvalX4                |      X|                 5          2
        .word uvarX3                |      L3])              10          2
        .word exec,conc             | :- conc(L1,L2,L3).     ──          2
                                                             48         ──
        .text                                                            8

        nop                 37      | leavecopymode
exec:   cmpl Htop,aH        38      | H < Htop else heapfull
        jge heapfull        39      |
        movl aP,d0          40      | P := P//procedure(P)
        movw aP@,d0         41      |
        movl d0,aP          42      |
        movw aP@(4),d0      43      | P := P//entrypoint(2,P)
        movl d0,aP          44      |
        movw aP@+,a0        45      | do nextoperation(P)
        jmp a0@             46      |

heapfull: jra xxxxx

        bras wvarX4             32  | incopymode then wvalX4
uvarX4: movl aS@+,d4         9  12  | X4 := nextterm(S)
        movw aP@+,a0        10  13  | do nextoperation(P)
        jmp a0@             11  14  |

wvarX4: movl aH,d4              33  | X4 := tagref(H)
        movl d4,aH@+            34  | nextterm(H) := X4
        movw aP@+,a0            35  | do nextcopyoperation(P)
        jmpa0@(-2)              36  |

        bras wvalX4         28      | incopymode then wvalX4
uvalX4: jra xxxxx                   | ...

wvalX4: movl d4,aH@+        29      | nextterm(H) := X4
        movw aP@+,a0        30      | do nextcopyoperation(P)
        jmpa0@(-2)          31      |

fail:   jra xxxxx
xxxxx:  .long 0

        nop                         | leavecopymode
glistA3: btst #0,d3          1  15  | islistorref(A3) else glistA3f
        jne glistA3f         2  16  |
        btst #1,d3           3  16a | islist(A3) else glistA3r
        jeq glistA3r         4  16b |
        movl d3,aS           5      | S := untaglist(A3)
```

```
            subqw #2,aS        6      |
            movw aP@+,a0       7      | do nextoperation(P)
            jmp a0@            8      |

glistA3f: jra xxxxx

glistA3d: jra xxxxx

glistA3r: movl d3,aR          17      | R := untagref(A3)
            cmpl aR@,aR       18      | isunbound(R) else glistA3d
            ~~jne glistA3d~~      19      |
          / lea aH@(2),a1     20      | binding(R) := taglist(H)
            movl a1,aR@       21      |
            cmpl aR,aH        22      | isinheap(R) else glistA3l
            jle glistA3l      23      |
            cmpl aR,aHB       24      | isbefore(R,HB) else glist A3x
            jle glistA3x      25      |
            movl aR,aTR@+             | nextref(TR) := R
glistA3x: movw aP@+,a0        26      | do nextcopyoperation(P)
            jmp a0@(-2)       27      |

glistA3l: jra xxxxx

| 7 P-instrs, 48 instrs, 56.8 microsec on Plexus (12.5 MHz)
| 0.85 mips, 17.6 klips
```