TM-0311

A Collection of KLI Programs

– Part 1 –

by
S. Takagi

November, 1987

ICOT
Mita Kokusai Bldg. 21F
4-28 Mita 1-Chome
Minato-ku Tokyo 108 Japan

(03) 456-2511
Telex ICOT J32964

Abstract

This memorandum is a collection of KL1-GHC programs. The purpose of the programs is:

1: To study how to write KL1 programs
   (i.e. how to write AND-parallel Prolog programs)
2: To study how to add pragmas to control goal distribution
3: To obtain information on the behavior of KL1 programs

All the source programs as well as brief documentation for each are included in this memorandum.

We hope that this memorandum will help researchers outside ICOT to try this kind of programming.

1. Introduction

At ICOT, research on a parallel inference machine (PIM) is in progress. For proto-research on the PIM, the multi-PSI (MPSI) system has been built from several personal sequential inference machines (PSIs).

An AND-parallel type Prolog based language called KL1 has been introduced for the multi-PSI. A six-PE model of the multi-PSI has already been built. We have to write an operating system on this multi-PSI system.

To start writing an operating system, we need to study how to write KL1 programs, and how to write pragmas to distribute the goals to PEs. This is the motivation for this collection of programs. These programs can give us some useful information on the behavior of AND-parallel Prolog programs.

So, the purpose of these programs is:

1: To study how to write KL1 programs
   (i.e. how to write AND-parallel Prolog programs)
2: To study how to add pragmas to control goal distribution
3: To obtain information on the behavior of KL1 programs

These programs are running on several different implementations of the KL1-GHC system. These KL1-GHC implementations have different built-in functions, so that the programs need to be modified to run different implementations. Only a few programs have been tested on the real MPSI system.

Some measurement has been done for these programs, but the result is not included here. The result will be reported in other papers and memos.

Any comments or discussions are welcome.
Please send E-mail to

Internet: takagi@icot.jp
UUCP: {enea,inria,kddlab,mit-eddie,ukc}!icot!takagi

2. Programs

Complete source programs as well as brief documentation for each program are listed in this memorandum. They are stored on the DEC-2065 at ICOT to make copies available to researchers.

For each program, read the documentation.

The programs are:

- bestpath_layered
- bestpath_monit_complexnet_taki
- bestpath_shortcircuit_ichiyoshi
- bestpath_termdet_ichiyoshi
- grid_furuichi
- goodpath_layered
- labeling_sugino_579
- life_sugino_580
- life_sugino_580new
- maxflow1-ex1
- maxflow2-ex2-3
- pascal_sugino_591
- queenf_sugino
- rucs_ezaki
- tep_koshimura
- trs1_takagi

3. Comments

Problems are solved in different ways. Each of these programs behaves differently. The result is very informative for writing pragmas, and also for improving the basic algorithm for the programs.

The most important result is that the layered-stream algorithm is probably not very good for the MPSI although the algorithm has significantly improved the execution speed for some sequential Prolog programs.

4. Acknowledgments

I wish to thank the KL1-TG group members for writing these programs and documentation although they are not used to writing in English. I would also like to thank Dr. F. Pereira of SRI, Mr. K. Sakai of ICOT, and Mr. Okumura of ICOT for suggesting these programs.

5. References

K. Furukawa, S. Kunifuji, A. Takeuchi, and K. Ueda:
The Conceptual Specification of the Kernel Language Version 1,
ICOT TR-054, March 1984, 41p

Furukawa, Takeuchi, Miyazaki, Ueda, and Tanaka:
Kernel Language Version 1, ICOT, May 1985, 36p (in Japanese only)

J. Tanaka, K. Ueda, T. Miyazaki, A. Takeuchi, Y. Matsumoto, and K. Furukawa:
Guarded Horn Clauses and Experiences with Parallel Logic Programming,
Proc. FJCC 1986, ACM and IEEE Computer Society, Nov. 2-6 1986, pp. 948-954

K. Ueda:
Guarded Horn Clauses, ICOT TR-103, 1985 and 1986
Also in Proc. Logic Programming '85, E. Wada (ed.),
Lecture Notes in Computer Science 221, Springer-Verlag, 1986, pp. 168-179
To appear in Concurrent Prolog: Collected Papers, Vol. 1,
E. Y. Shapiro (ed.), The MIT Press, 1987

K. Ueda:
Guarded Horn Clauses: A Parallel Logic Programming Language with the
Concept of a Guard, ICOT TR-208, 1986 and 1987
Also to appear in Programming of Future Generation Computers,
M. Nivat and K. Fuchi (eds.), North-Holland, 1987

K. Ueda:
Guarded Horn Clauses, Doctoral Thesis, Information Engineering Course,
Faculty of Engineering, Univ. of Tokyo, 1986

K. Ueda:
Introduction to Guarded Horn Clauses, ICOT TR-209, 1986

% <MPSI.BENCH>BESTPATH_LAYERED.DOC.2, 4-Aug-87 11:56:00, Edit by OKUMURA

(0) Date: 1987-Jul-27, written by A. Okumura
Modified: 4-aug-87 by S. Takagi

(1) Program name: bestpath_layered
(best path problem using the layered-stream method)

(2) Author: A. Okumura, ICOT 1st Lab.

(3) Runs on: GHC system on DECsystem-20

(4) Description of the problem:

A best path search problem on a network.

The network consists of nodes and arcs. Each arc connects two nodes. A node is connected to at least one arc. Each arc has a non-negative cost. A path is a route from a given start node to a given end node through arbitrary arcs. The cost of a path is defined as the total cost of the arcs forming the path.

This problem is to find one of the minimum cost paths between the start and the end node.

(5) Algorithm:

Node names and edge costs are propagated in sequences along edges. If a node receives a sequence which includes the name of the node itself, that sequence is eliminated. During the check, the total cost of the sequence is calculated. It is also propagated to other checking processes. If the calculated cost is higher than the cost of another sequence, the sequence is eliminated. If the goal node receives a sequence from the start node which does not include the name of the goal node, it is a candidate to be output. After the goal node receives the last input, the candidate sequence which has the lowest cost is output as an answer.

(6) Process structure:

One process is generated for each node. A node process prepares a pair of its node name and a layered stream as its output. The stream holds good paths from the start node to that node. The layered stream is made from the input from its neighbors by eliminating non-good paths. Non-good paths are paths that contain the current node name (i.e. some loop is in the path) or that have a higher cost than a path already output.

(7) Pragma: Not set yet

(8) Program:

The top-level predicate is bestPath/3.
    This predicate defines the configuration of the graph,
    generates a process for each node of the graph,
    and sets up layered streams along the edges.

Predicates like graphN/i are the subsidual predicate of bestPath/3.
    They divide the graph into subgraphs for compilation.

node/8 produces the primary output bindings for each node.
    The first clause of node/8 is for the starting node.
    The second clause is for the goal node.

filter/7 is for filtering out paths that include loops.

minMerge/4 the output which costs more than the previous one.

lastFilter/6 outputs the path with the lowest cost.

(9) Source file: US2:<MPSI.BENCH>BESTPATH_LAYERED.GHC
    This document: US2:<MPSI.BENCH>BESTPATH_LAYERED.DOC

(10) Examples:

Invocation:
        bestPath(Start,Goal,Path).
                where Start is the starting node,
                Goal is the goal node,
                and all the best paths between Start and Goal are obtained
                as the third argument Path.

For example, run bestPath(n0,n11,Path). The following result is obtained.

        Path = 13-[[n0,n1,n2,n3,n4,n5,n11]]

(11) Evaluation data: Refer to the data in PSM-O-A-KL1-006.

```
% <OKUMURA.LAYERED-STREAM>BPATH3.2.1, 27-Mar-87 13:07:30, Edit by OKUMURA

bestPath(S,G,Path) :- true |
        node(Min,Fin,S,G,n0,[2-N1,1-N6],N0,Path),
        node(Min,Fin,S,G,n1,[2-N0,3-N2,2-N7],N1,Path),
        node(Min,Fin,S,G,n6,[1-N0,1-N7,2-N12],N6,Path),
        node(Min,Fin,S,G,n7,[2-N1,1-N6,3-N13],N7,Path),
        node(Min,Fin,S,G,n12,[2-N6,3-N13,4-N18],N12,Path),
        node(Min,Fin,S,G,n13,[3-N7,3-N12,5-N14,3-N19],N13,Path),
        graph1(Min,Fin,S,G,Path,N1,N2,N12,N13,N14,N18,N19).

graph1(Min,Fin,S,G,Path,N1,N2,N12,N13,N14,N18,N19) :- true |
        node(Min,Fin,S,G,n2,[3-N1,4-N3,4-N3],N2,Path),
        node(Min,Fin,S,G,n3,[4-N2,2-N4,4-N9],N3,Path),
        node(Min,Fin,S,G,n8,[2-N2,3-N9,4-N14],N8,Path),
        node(Min,Fin,S,G,n9,[4-N3,3-N8,6-N10,8-N15],N9,Path),
        node(Min,Fin,S,G,n14,[4-N8,5-N13,7-N15],N14,Path),
        node(Min,Fin,S,G,n15,[8-N9,7-N14,9-N21],N15,Path),
        graph2(Min,Fin,S,G,Path,N3,N4,N9,N10,N12,N13,N15,N18,N19,N21).

graph2(Min,Fin,S,G,Path,N3,N4,N9,N10,N12,N13,N15,N18,N19,N21) :- true |
        node(Min,Fin,S,G,n4,[2-N3,1-N5,3-N10],N4,Path),
        node(Min,Fin,S,G,n5,[1-N4,1-N11],N5,Path),
        node(Min,Fin,S,G,n10,[3-N4,6-N9,2-N11,6-N16],N10,Path),
        node(Min,Fin,S,G,n11,[1-N5,2-N10,3-N17],N11,Path),
        node(Min,Fin,S,G,n16,[6-N10,4-N17,7-N22],N16,Path),
        node(Min,Fin,S,G,n17,[3-N11,4-N16,4-N23],N17,Path),
        graph3(Min,Fin,S,G,Path,N12,N13,N15,N16,N17,N18,N19,N21,N22,N23).

graph3(Min,Fin,S,G,Path,N12,N13,N15,N16,N17,N18,N19,N21,N22,N23) :- true |
        node(Min,Fin,S,G,n18,[4-N12,2-N19,3-N24],N18,Path),
        node(Min,Fin,S,G,n19,[3-N13,2-N18,4-N20],N19,Path),
        node(Min,Fin,S,G,n24,[3-N18,2-N25,2-N30],N24,Path),
        node(Min,Fin,S,G,n25,[2-N24,5-N26,2-N31],N25,Path),
        node(Min,Fin,S,G,n30,[2-N24,1-N31],N30,Path),
        node(Min,Fin,S,G,n31,[2-N25,1-N30,1-N32],N31,Path),
        graph4(Min,Fin,S,G,Path,N15,N16,N17,N19,N20,N21,N22,N23,N25,N26,N31,N32)

graph4(Min,Fin,S,G,Path,N15,N16,N17,N19,N20,N21,N22,N23,N25,N26,N31,N32) :- true
        node(Min,Fin,S,G,n20,[4-N19,7-N21,6-N26],N20,Path),
        node(Min,Fin,S,G,n21,[9-N15,7-N20,8-N27],N21,Path),
        node(Min,Fin,S,G,n26,[6-N20,5-N25,3-N32],N26,Path),
        node(Min,Fin,S,G,n27,[8-N21,3-N28,4-N33],N27,Path),
        node(Min,Fin,S,G,n32,[3-N26,1-N31,3-N33],N32,Path),
        node(Min,Fin,S,G,n33,[4-N27,3-N32,2-N34],N33,Path),
        graph5(Min,Fin,S,G,Path,N16,N17,N22,N23,N27,N28,N33,N34).

graph5(Min,Fin,S,G,Path,N16,N17,N22,N23,N27,N28,N33,N34) :- true |
        node(Min,Fin,S,G,n22,[7-N16,1-N23,5-N28],N22,Path),
        node(Min,Fin,S,G,n23,[4-N17,1-N22,2-N29],N23,Path),
        node(Min,Fin,S,G,n28,[5-N22,3-N27,2-N29,2-N34],N28,Path),
        node(Min,Fin,S,G,n29,[2-N23,2-N28,1-N35],N29,Path),
        node(Min,Fin,S,G,n34,[2-N28,2-N33,2-N35],N34,Path),
        node(Min,Fin,S,G,n35,[1-N29,2-N34],N35,Path).

node(Min,Fin,_,G,G,In,Out,Path) :- true |
        Out = nil, lastFilter(Min,Min,0-[G],nil,In,Path), announce(Path,Fin).
node(Min,_,S,_,S,_,Out,_) :- true | Out = S*begin.
node(Min0,Fin,S,G,N,In,Out,_) :- S \= N, G \= N |
        filter(Fin,N,Min,Min1,0,In,In1), Out = N*In1,
        minMerge(Fin,Min0,Min1,Min).

announce(_-_,Out) :- true | Out = fin.

filter(fin,_,_,_,_,_,_) :- true | true.
filter(Fin,Node,[Min1,Min2|Mins],Min,Cost,In,Out) :- Min1 >= Cost |
```

(0) Date: 1987-Jun-3, written by K. Taki
Modified: 31-jul-87 by S. Takagi

(1) Program name: bestpath

Bestpath program using a single monitor process

(2) Author: K. Taki, ICOT 4th Lab.

(3) Runs on: GHC and GHC3 on DEC-2065

(4) Description of the problem:

A best path search problem on a network.
A network is constructed from nodes and arcs. Each arc connects two nodes. A node is connected to at least one arc. Each arc has a non-negative cost. A path is a route from a given start node to a given end node through arbitrary arcs. The cost of a path is defined as the total cost of arcs forming the path.
The problem is to find the minimum cost path when the start and end nodes are given.

(5) Algorithm:

In this path search algorithm, each growing path is an activity.
It is called a process here.

A process has a state. The state contains path information and the cost of the growing path. The process has traversed the path and the cost is the sum of all the arc costs in the growing path.

The process forks traversing arcs and expands its path information.

There is another activity called a monitor process. The monitor holds the current best path information at any time. When a new best path is found, the monitor broadcasts the new minimum cost to all the processes.

i. Normal process activity for expanding the path:
When a process arrives at a node,
the process checks whether the node is the end node or not, and
whether a broadcast message arrives or not.
If neither is the case,
the process checks whether the node has been visited the first time.
If not, the process terminates at once.
If that node has been visited the first time,
the process records the node number as its path information.
The process forks by n according to the number of connecting arcs,
and newly forked processes travel along each connecting arc.

A new forked process adds the arc cost to its cost information,
and the forked process visits the next node.

ii. When a process arrives at the end node:
When a process arrives at the end node and no broadcast message has arrived (it means that a new path has been completed).
The process reports the state information to the monitor process,
then terminates itself.

```
        filter(Fin,Node,[Min2|Mins],Min,Cost,In,Out).
filter(Fin,_,[Min1|_],Min,Cost,_,Out) :- Min1 < Cost | Min = [], Out = [].
filter(Fin,Node,Mins,Min,Cost,[C-Node*In1|Ins],Out) :- true |
        filter(Fin,Node,Mins,Min,Cost,Ins,Out).
filter(Fin,Node,Mins,Min,Cost,[C-nil|Ins],Out) :- true |
        filter(Fin,Node,Mins,Min,Cost,Ins,Out).
filter(Fin,Node,Mins,Min,C0,[C-N*In1|Ins],Out) :- N \= Node |
        Cost := C0+C,
        filter(Fin,Node,Mins,M1,Cost,In1,Out1),
        filter(Fin,Node,Mins,M2,C0,  Ins,Outs),
        minMerge(Fin,M1,M2,Min),
        Out = [C-N*Out1|Outs].
filter(Fin,_,_,Min,Cost,begin,Out) :- true | Min = [Cost], Out = begin.
filter(Fin,_,_,Min,_,[],Out) :- true | Min = [], Out = [].


lastFilter([Min1,Min2|Mins],Min,Cost-Stack,Prev,In,Out) :- Min1 >= Cost |
        lastFilter([Min2|Mins],Min,Cost-Stack,Prev,In,Out).
lastFilter([Min1|_],Min,Cost-_,Prev,_,Out) :- Min1 < Cost |
        Out = Prev, Min = [].
lastFilter(_,Min,Cost-Stack,nil,begin,Out) :- true |
        Out = Cost-[Stack], Min = [Cost].
lastFilter(_,Min,Cost-Stack,Cost-Path0,begin,Out) :- true |
        Out = Cost-[Stack|Path0], Min = [].
lastFilter(_,Min,Cost-_,Cost0-Path0,begin,Out) :- Cost > Cost0 |
        Out = Cost0-Path0, Min = [].
lastFilter(_,Min,Cost-Stack,Cost0-_,begin,Out) :- Cost < Cost0 |
        Out = Cost-[Stack], Min = [Cost].
lastFilter(_,Min,_,Prev,[],Out) :- true | Out = Prev, Min = [].
lastFilter(Mins,Min,Stack,Prev,[_-nil|Ins],Out) :- true |
        lastFilter(Mins,Min,Stack,Prev,Ins,Out).
lastFilter(Mins,Min,Cost-Stack,Prev,[C-N*In1|Ins],Out) :- true |
        Cost1 := Cost+C,
        lastFilter(Mins,M1,Cost1-[N|Stack],Prev,In1,Mid),
        lastFilter(Mins,M2,Cost-Stack,Mid,Ins,Out),
        minMerge(_,M1,M2,Min).

minMerge(fin,_,_,_) :- true | true.
minMerge(Fin,[M|A],B,Out) :- true | Out = [M|Outs], minM1(Fin,M,A,B,Outs).
minMerge(Fin,A,[M|B],Out) :- true | Out = [M|Outs], minM1(Fin,M,A,B,Outs).
minMerge(Fin,[],In,Out) :- true | Out = In.
minMerge(Fin,In,[],Out) :- true | Out = In.

minM1(fin,_,_,_,_) :- true | true.
minM1(Fin,M0,[M|A],B,Out) :- M0 > M | Out = [M|Outs], minM1(Fin,M,A,B,Outs).
minM1(Fin,M0,A,[M|B],Out) :- M0 > M | Out = [M|Outs], minM1(Fin,M,A,B,Outs).
minM1(Fin,M0,[M|A],B,Out) :- M0 =< M | minM1(Fin,M0,A,B,Out).
minM1(Fin,M0,A,[M|B],Out) :- M0 =< M | minM1(Fin,M0,A,B,Out).
minM1(Fin,M0,[],In,Out) :- true | minM2(Fin,M0,In,Out).
minM1(Fin,M0,In,[],Out) :- true | minM2(Fin,M0,In,Out).

minM2(fin,_,_,_) :- true | true.
minM2(Fin,M0,[M|In],Out) :- M0 > M | Out = [M|Outs], minM2(Fin,M,In,Outs).
minM2(Fin,M0,[M|In],Out) :- M0 =< M | minM2(Fin,M0,In,Out).
minM2(Fin,_,[],Out) :- true | Out = [].
```

iii. Monitor process activity:
When the monitor process receives the reported state information,
it compares the reported cost with its own recorded cost (minimum cost).
If the reported cost is higher,
the monitor process abandons the reported information.
If the reported cost is equal to the recorded cost,
the monitor adds the reported path information to its recorded
(best) path information.
If the reported cost is lower than the recorded cost,
the monitor changes the whole record with the reported information,
and broadcasts the reported cost to all the live processes.
(This means that the reported cost is the minimum detected
by the monitor.)

iv. When a broadcast message arrives at a process:
When a broadcast message (newly reported minimum cost) arrives
at a process, the process compares the broadcast cost
with its own cost.
If the broadcast cost is less than or equal to its own cost,
the process terminates itself at once.
(This means that there is a better path.)
If the broadcast cost is higher than its own cost,
the process continues to operate as long as its own cost
does not exceed the broadcast cost.
If its own cost exceeds the broadcast cost, the process terminates.

v. Termination:
When all the processes except the monitor terminate, the information
recorded in the monitor process is the best path information.

(6) Process structure:

There are one monitor process and many path processes.
Path processes fork according to the network graph.
Report streams of the forked path processes are merged
and connected to the monitor process.
The broadcast stream of the monitor process is shared by all path
processes.

(7) Pragma: Not supported

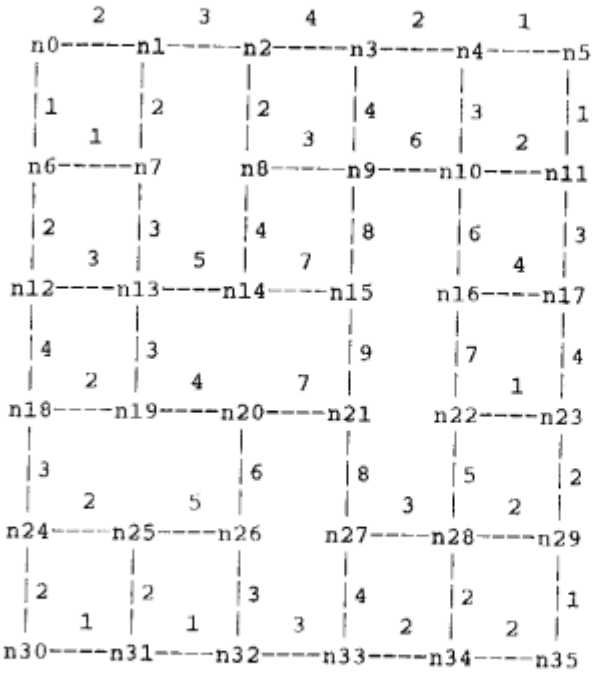
(8) Program:

bestpath/3:	Top level
evaluator/5:	Monitor process
path/6:	Path process
p/7:	This predicate checks loops. If the path has a loop, that path is terminated at once. If it does not, the connecting arc list (edge list) is selected and path/6 is recursively called.
edge/2:	Network data

(9) Source file: US2:<MPSI.BENCH>BESTPATH_MONIT_COMPLEXNET_TAKI.GHC
This document: US2:<MPSI.BENCH>BESTPATH_MONIT_COMPLEXNET_TAKI.DOC

(10) Examples:

Network:



Start the program:

```
%----------------------------------------------------------
%%% Example on GHC3 system

yes
| ?- ghc bestpath(n0,n8,Ans).

259 reductions and 54 suspensions in 34 cycles and 449 msec (576 rps)
The maximum number of reducible goals is 16/28 in the 25th cycle.
The maximum length of the queue is 29.

Ans = 7*[[n8,n2,n1,n0]]

yes
| ?- ghc bestpath(n15,n22,Ans).

4115 reductions and 666 suspensions in 86 cycles and 6700 msec (614 rps)
The maximum number of reducible goals is 114/207 in the 49th cycle.
The maximum length of the queue is 208.

Ans = 24*[[n22,n23,n17,n11,n5,n4,n3,n9,n15],[n22,n23,n17,n11,n10,n9,n15]]

%%%  This example shows that there are two best paths with the same cost.
```

(11) Evaluation data:

Measurement of bestpath problem in GHC3 '87.4.2 TAKI

route	suspensions reductions	cycles	msec	rps	max reducible goals at nth-cycle*1	*2 queue leng.
Taki-monitor-method						
n0-n7	57	13	17	98	581	9/11at11	11
n0-n14	576	113	51	876	657	31/43at25	48
n0-n21	4991	773	108	7935	628	144/260at55	260
n0-n28	23386	2868	135	38175	612	537/891at89	899
n0-n35	31681	4049	153	56952	336	1158/2146at91	2146
n4-n32	15899	2006	108	25732	617	391/720at64	720
n8-n26	6424	1038	120	10455	614	146/269at43	269
n13-n28	8334	1153	107	13815	603	173/299at74	314
Ichiyoshi-method (bestpath_termdet_ichiyoshi.ghc)						
n4-all	4592	858	393	23026	199	42/72at68	162
n8-all	4493	803	389	22183	202	42/135at87	164
n13-all	4778	1008	393	23525	203	44/158at112	160

*1 A/BatC: A = maximum number of reducible goals
B = number of goals (queue length) when A is measured
C = cycle number when A is measured

*2 queue leng.: maximum length of queue

```
%-------- Bestpath problem --------
%       ( Taki-monitor-method )

bestpath(Start,Goal,Best_path) :- true |
        edge(Start,Next),
        path(Next,Goal,0*[Start],Path,Best_cost,10000),
        evaluator(Path,Best_cost,10000,[],Best_path).

evaluator([Total_cost*Path|Next],Best_cost,Cost_work,Path_work,Best_path) :-
        Total_cost < Cost_work |
        Best_cost = [Total_cost|NN],
        evaluator(Next,NN,Total_cost,[Path],Best_path).
evaluator([Total_cost*Path|Next],Best_cost,Cost_work,Path_work,Best_path) :-
        Total_cost = Cost_work |
        evaluator(Next,Best_cost,Cost_work,[Path|Path_work],Best_path).
evaluator([Total_cost*Path|Next],Best_cost,Cost_work,Path_work,Best_path) :-
        Total_cost > Cost_work |
        evaluator(Next,Best_cost,Cost_work,Path_work,Best_path).
evaluator([],Best_cost,Cost_work,Path_work,Best_path) :-  true |
        Best_cost = [],
        Best_path = Cost_work*Path_work.

% prior ;
path(Next,Goal,HIS,Path,[A_best_cost|Cost_next],Best_cost_work) :-
        true | path(Next,Goal,HIS,Path,Cost_next,A_best_cost).
% end prior ;

path(_,_,Acc_cost*_,P0,_,B_work) :-
        Acc_cost >= B_work | P0 = [].
path([Node*Cost|Next],Goal,Acc_cost*History,P0,B_cost,B_work) :-
        Acc_cost < B_work, Goal \= Node |
        p(History,Goal,Node,*(Acc_cost,History,Cost),P1,B_cost,B_work),
        path(Next,Goal,Acc_cost*History,P2,B_cost,B_work),
        merge(P1,P2,P0).
path([],_,_,P0,_,_) :- true | P0 = [].
path([Goal*Cost|Next],Goal,Acc_cost*History,P0,B_cost,B_work) :-
        Acc_cost < B_work |
        Total_cost := Acc_cost + Cost,
        P0 = [Total_cost*[Goal|History]|P1],
        path(Next,Goal,Acc_cost*History,P1,B_cost,B_work).

p([N1|Ns],Goal,N,HIS,P0,B_cost,B_work) :- N \= N1 |
        p(Ns,Goal,N,HIS,P0,B_cost,B_work).
p([],Goal,N,*(Acc_cost,History,Cost),P0,B_cost,B_work) :- true |
        edge(N,Next_next),
        New_acc_cost := Acc_cost + Cost,
        path(Next_next,Goal,New_acc_cost*[N|History],P0,B_cost,B_work).
p([N|_],_,N,_,P0,_,_) :- true | P0 = [].

edge(n0,Nodes) :- true | Nodes = [n1*2,n6*1].
edge(n1,Nodes) :- true | Nodes = [n0*2,n2*3,n7*2].
edge(n2,Nodes) :- true | Nodes = [n1*3,n3*4,n8*2].
edge(n3,Nodes) :- true | Nodes = [n2*4,n4*2,n9*4].
edge(n4,Nodes) :- true | Nodes = [n3*2,n5*1,n10*3].
edge(n5,Nodes) :- true | Nodes = [n4*1,n11*1].
edge(n6,Nodes) :- true | Nodes = [n0*1,n7*1,n12*2].
edge(n7,Nodes) :- true | Nodes = [n1*2,n6*1,n13*3].
edge(n8,Nodes) :- true | Nodes = [n2*2,n9*3,n14*4].
edge(n9,Nodes) :- true | Nodes = [n3*4,n8*3,n10*6,n15*8].
edge(n10,Nodes) :- true | Nodes = [n4*3,n9*6,n11*2,n16*6].
edge(n11,Nodes) :- true | Nodes = [n5*1,n10*2,n17*3].
edge(n12,Nodes) :- true | Nodes = [n6*2,n13*3,n18*4].
edge(n13,Nodes) :- true | Nodes = [n7*3,n12*3,n14*5,n19*3].
edge(n14,Nodes) :- true | Nodes = [n8*4,n13*5,n15*7].
edge(n15,Nodes) :- true | Nodes = [n9*8,n14*7,n21*9].
edge(n16,Nodes) :- true | Nodes = [n10*6,n17*4,n22*7].
```

```
edge(n17,Nodes) :- true | Nodes = [n11*3,n16*4,n23*4].
edge(n18,Nodes) :- true | Nodes = [n12*4,n19*2,n24*3].
edge(n19,Nodes) :- true | Nodes = [n13*3,n18*2,n20*4].
edge(n20,Nodes) :- true | Nodes = [n19*4,n21*7,n26*6].
edge(n21,Nodes) :- true | Nodes = [n15*9,n20*7,n27*8].
edge(n22,Nodes) :- true | Nodes = [n16*7,n23*1,n28*5].
edge(n23,Nodes) :- true | Nodes = [n17*4,n22*1,n29*2].
edge(n24,Nodes) :- true | Nodes = [n18*3,n25*2,n30*2].
edge(n25,Nodes) :- true | Nodes = [n24*2,n26*5,n31*2].
edge(n26,Nodes) :- true | Nodes = [n20*6,n25*5,n32*3].
edge(n27,Nodes) :- true | Nodes = [n21*8,n28*3,n33*4].
edge(n28,Nodes) :- true | Nodes = [n22*5,n27*3,n29*2,n34*2].
edge(n29,Nodes) :- true | Nodes = [n23*2,n28*2,n35*1].
edge(n30,Nodes) :- true | Nodes = [n24*2,n31*1].
edge(n31,Nodes) :- true | Nodes = [n25*2,n30*1,n32*1].
edge(n32,Nodes) :- true | Nodes = [n26*3,n31*1,n33*3].
edge(n33,Nodes) :- true | Nodes = [n27*4,n32*3,n34*2].
edge(n34,Nodes) :- true | Nodes = [n28*2,n33*2,n35*2].
edge(n35,Nodes) :- true | Nodes = [n29*1,n34*2].


merge([A|X],Y,Z) :- A \= [] | Z = [A|W], merge(Y,X,W).
merge(X,[A|Y],Z) :- A \= [] | Z = [A|W], merge(Y,X,W).
merge([],Y,Z)    :- true    | Z = Y.
merge(X,[],Z)    :- true    | Z = X.
```

/* US2:<MPSI.BENCH>BESTPATH_SHORTCIRCUIT_ICHIYOSHI.GHC */

(0) Date: 1987-Jul-2, written by N. Ichiyoshi

(1) Program name: bestpath
(bestpath program with the shortcircuit method as termination detection)

(2) Author: N. Ichiyoshi

(3) Runs on:
Ueda FGHC compiler
Pseudo multi-PSI

(4) Description of the problem:

Given a network with a non-negative cost assigned to each edge, and a start node, find for each node the path with the minimum accumulated cost from the starting node.

(5) Algorithm:

The nodes in the network send a path and cost information packet cp(Cost,Path) to the adjacent nodes. At all time, each node keeps:
(1) C (the minimum cost found so far from the starting node), and
(2) P (a path that realizes cost C).

It maintains the above information as follows. When a node receives a new path and cost information packet cp(Cost,Path) from any one of its neighboring nodes, it does the following. If Cost >= C, then the node simply ignores Cost and Path. Otherwise, it updates C and P to Cost and Path, and sends cost and path information packets to its neighboring nodes. Node (n) sends a path and cost information cp(Cost+EC,[n|Path]) where EC is the cost of the edge from n to n' and [n|Path] is Path extended by appending n to the top.

The above procedure is repeated until there are no path and cost information packets in the network. The finiteness of the network guarantees that this state is reached in a finite time. (There are only a finite number of non-circular paths in the network, and circular paths are guaranteed to be discarded because of the algorithm and the non-negative costs of the edges.)

This program detects this state by the shortcircuit technique. Shortcircuit switches are attached to all packets. The shortcircuit switch is closed when the packet is discarded. It splits into serially connected switches when the packet spawns child packets. The circuit attached to the initial packet is closed when all descendant circuits are closed, i.e. when all descendant packets are discarded.

(6) Process structure:

One node in the network has one corresponding node process. A node process is connected to its neighboring node processes by one input and one output stream.

(7) Pragma: Not attached

(8) Program:

```
(main part)
        node/8: Node process
        send_cp/6:      Subroutine for sending path/cost information packets
                to neighboring nodes
        closeOuts/1:    Closes a stream
        bp/4:           Top level goal
(network generation)
        augment_edges/2: Augments edge information to include both-way streams
        del/3:          Deletes an element from a list
        gen_nodes/5:    Node generation routine
        gen_node/8:     Makes a node process out of node information
(utility predicates)
        merge_all/2:    n-way merger
        merge/3:        Standard merger
        length/2,length/3: Gives the list length
        write_result/2: Writes out stream elements
        bp_sne/3:       Runs the program with the start node, set of nodes,
                        and set of edges given
        bp_ne/2:        Runs the program with the set of nodes
                        and set of edges given
(test program)
        bp_ex1_n/1:     First example
        bp_ex2_n/1:     Second example
        bp_ex3_s/1:     Third example
```

(9) Source file: US2:<MPSI.BENCH>BESTPATH_SHORTCIRCUIT_ICHIYOSHI.GHC
        247 lines (111 lines of which are test data)

(10) Examples:

```
Invocation:
(To run example 1 with start node a)
        :- bp_ex1_n([a,b,c,d,e,f,g,h]).
(To run example 2 with start node a)
        :- bp_ex2_n([a,b,c,d,e,f,g,h,i,j,k,l]).
(To run example 3 with start node n0)
        :- bp_ex3_s(n0).
```

(11) Evaluation data: Not recorded

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
%       Best Path Problem
%
%       N. Ichiyoshi (according to Ohki-san's idea of using short circuit)
%       April 1987
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%       node process              %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% node(In,Outs,Cout,C,P, End,R0-R,Cs,N)
%
%       In      : input stream (including the control stream)
%       Outs    : output streams
%       C       : minimum cost so far
%       P       : path with cost C
%       End     : termination flag
%       R0-R    : result d-list
%       Cs      : costs of outgoing edges (constant)
%       N       : node identifier (constant)
%
node([cp(Cost,Path,T0-T)|In],Outs,C,P, End,R0-R, Cs,N) :-
    Cost >= C |
    T0 = T,
    node(In,Outs,C,P, End,R0-R,Cs,N).
node([cp(Cost,Path,T0-T)|In],Outs,C,P, End,R0-R,Cs,N) :-
    Cost < C |
    send_cp(Outs,Cs,Cost,[N|Path], T0-T, NewOuts),
    node(In,NewOuts,Cost,Path, End,R0-R,Cs,N).
node(_,Outs,C,P, end,R0-R,_,N) :- true |
    R0 = [N-C-P|R],
    closeOuts(Outs).

send_cp([],_,_,_, T0-T, NewOuts) :- true | T0 = T, NewOuts = [].
send_cp([Out|Outs],[C|Cs],Cost,NewPath, T0-T, NewOuts) :-
    NewCost := Cost+C |
    Out = [cp(NewCost,NewPath,T0-T1)|Out1],
    NewOuts = [Out1|NewOuts1],
    send_cp(Outs,Cs,Cost,NewPath, T1-T, NewOuts1).

closeOuts([]) :- true | true.
closeOuts([Out|Outs]) :- true | Out = [], closeOuts(Outs).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%       Process generation
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%
% bp(S,Ns,Es, R)
%
%       S       : start node
%       Ns      : list of nodes (a node has a constant value)
%       Es      : list of edges (an edge is of form e(Node1-Node2,Cost))
%       R       : result (list of nodes with the minimum cost from start node
%                 and a path with the minimum cost)
%
bp(S,Ns,Es, R) :- true |
    augment_edges(Es, AEs),
    del(S,Ns, Ms),
    gen_nodes([S|Ms],AEs,End,R-[],S).

augment_edges([], AEs) :- true | AEs = [].
augment_edges([e(N1-N2,C)|Es], AEs) :- true |
```

```
    AEs = [e(N1-N2,C,_-_)|AEs1],
    augment_edges(Es, AEs1).

del(N,[M|Ns], Ms) :- N = M | Ms = Ns.
del(N,[M|Ns], Ms) :- N \= M | Ms = [M|Ms1], del(N,Ns, Ms1).
del(N,[], Ms) :- true | Ms = [].

gen_nodes([N|Ns],AEs,End,R0-R,S) :- true |
    gen_node(N,AEs,[],[],[],End,R0-R1,S),
    gen_nodes(Ns,AEs,End,R1-R,S).
gen_nodes([],_,_,R0-R,_) :- true | R0 = R.

gen_node(N,[e(N-_,Cost,Self-Other)|AEs],Ins,Outs,Costs,End,RR,S) :- true |
    Ins1 = [Self|Ins],
    Outs1 = [Other|Outs],
    Costs1 = [Cost|Costs],
    gen_node(N,AEs,Ins1,Outs1,Costs1,End,RR,S).
gen_node(N,[e(_-N,Cost,Other-Self)|AEs],Ins,Outs,Costs,End,RR,S) :- true |
    Ins1 = [Self|Ins],
    Outs1 = [Other|Outs],
    Costs1 = [Cost|Costs],
    gen_node(N,AEs,Ins1,Outs1,Costs1,End,RR,S).
gen_node(N,[e(O1-O2,_,_)|AEs],Ins,Outs,Costs,End,RR,S) :-
    N \= O1, N \= O2 |
    gen_node(N,AEs,Ins,Outs,Costs,End,RR,S).
gen_node(N,[],Ins,Outs,Costs,End,RR,S) :- N \= S |
    length(Outs, EC),
    merge_all(Ins, In),
    node(In,Outs,99999,?, End,RR,Costs,N).
gen_node(N,[],Ins,Outs,Costs,End,RR,S) :- N = S |
    length(Outs, EC),
    merge_all([[cp(0,[],End-end)]|Ins], In),
    node(In,Outs,99999,?, End,RR,Costs,N).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%       Utility
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

merge_all([], Out) :- true | Out = [].
merge_all([In|Ins], Out) :- true |
    merge(In,Out1, Out),
    merge_all(Ins, Out1).

merge([],Ys, Zs) :- true | Zs = Ys.
merge(Xs,[], Zs) :- true | Zs = Xs.
merge([X|Xs],Ys, Zs) :- true | Zs = [X|Zs1], merge(Xs,Ys, Zs1).
merge(Xs,[Y|Ys], Zs) :- true | Zs = [Y|Zs1], merge(Xs,Ys, Zs1).

length(L, N) :- true | length(L, 0,N).

length([], K,N) :- true | N = K.
length([_|L], K,N) :- K1 := K+1 | length(L, K1,N).

write_result([], O) :- true | O = [].
write_result([X|Xs], O) :- true |
    O = [write(X),nl|NewO], write_result(Xs, NewO).

%
% Bp_sne is given the start node, list of nodes and list of edges.
% Bp_ne is given list of nodes and list of edges. (The start node becomes
%  the first node in the list of nodes.)
%
bp_sne(Start,Nodes,Edges) :- true |
    bp(Start,Nodes,Edges, Result),
    write_result(Result, O),
    outstream(O).
```

```
bp_ne(Nodes,Edges) :- Nodes = [Start|_] |
    bp_sne(Start,Nodes,Edges).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%       Three examples
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%
%             ----a----
%           /     |     \
%       10 /    10|      \ 1
%         /       |  10   \
%        f        g-------b
%        |        |\      |
%        |        | \ 1   |
%        |        |  \    |
%      10|       1|   h   |1
%        |        |  / \  |
%         \       | /10 \ |
%          \      |/   10\|
%           \-----e---d---c
%                   1   1
%
bp_ex1_n(Nodes) :-
    true |
    bp_ne(Nodes,
      [e(a-f,10),e(a-g,10),e(a-b,1),
       e(b-g,10),e(b-c,1),
       e(c-h,10),e(c-d,1),
       e(d-e,1),
       e(e-h,10),e(e-f,10),e(e-g,1),
       e(g-h,1)
      ]).

%
%                  a
%                 / \
%              10/   \ 1
%               /     \
%              b       c
%             / \     / \
%          3/   \5  8/   \6
%           /     \ /     \
%          d-------e---f-----g
%           \  1  / 2 \  1  /
%          12\   /8   9\   /15
%             \ /       \ /
%              h         i
%             / \       / \
%          11/  18\   /10  \14
%           /      \ /     \
%          j--------k-------l
%                6       3
%
bp_ex2_n(Nodes) :-
    true |
    bp_ne(Nodes,
      [e(a-b,10),e(a-c,1),
       e(b-d,3),e(b-e,5),
       e(c-f,8),e(c-g,6),
       e(d-e,1),e(d-h,12),
       e(e-f,2),e(e-h,8),
       e(f-g,1),e(f-i,9),
       e(g-i,15),
       e(h-j,11),e(h-k,18),
```

```
        e(i-k,10),e(i-l,14),
        e(j-k,6),
        e(k-l,3)
       ]).

bp_ex3_s(Start) :-
    true |
    bp_sne(Start,
      [n0,n1,n2,n3,n4,n5,n6,n7,n8,n9,
       n10,n11,n12,n13,n14,n15,n16,n17,n18,n19,
       n20,n21,n22,n23,n24,n25,n26,n27,n28,n29,
       n30,n31,n32,n33,n34,n35
      ],
      [e(n0-n1,2),e(n0-n6,1),
       e(n1-n2,3),e(n1-n7,2),
       e(n2-n3,4),e(n2-n8,2),
       e(n3-n4,2),e(n3-n9,4),
       e(n4-n5,1),e(n4-n10,3),
       e(n5-n11,1),
       e(n6-n7,1),e(n6-n12,2),
       e(n7-n13,3),
       e(n8-n9,3),e(n8-n14,4),
       e(n9-n10,6),e(n9-n15,8),
       e(n10-n11,2),e(n10-n16,6),
       e(n11-n17,3),
       e(n12-n13,3),e(n12-n18,4),
       e(n13-n14,5),e(n13-n19,3),
       e(n14-n15,7),
       e(n15-n21,8),
       e(n16-n17,4),e(n16-n22,7),
       e(n17-n23,4),
       e(n18-n19,2),e(n18-n24,3),
       e(n19-n20,4),
       e(n20-n21,7),e(n20-n26,6),
       e(n21-n27,8),
       e(n22-n23,1),e(n22-n28,5),
       e(n23-n29,2),
       e(n24-n25,2),e(n24-n30,2),
       e(n25-n26,5),e(n25-n31,2),
       e(n26-n32,3),
       e(n27-n28,3),e(n27-n33,4),
       e(n28-n29,2),e(n28-n34,2),
       e(n29-n35,1),
       e(n30-n31,1),
       e(n31-n32,1),
       e(n32-n33,3),
       e(n33-n34,2),
       e(n34-n35,2)
      ]).
```

/* US2:<MPSI.BENCH>BESTPATH_TERMDET_ICHIYOSHI.DOC */

(0) Date: 1987-Jul-9, written by N. Ichiyoshi

(1) Program name: bestpath
(bestpath program with the message counting scheme
as termination detection)

(2) Author: N. Ichiyoshi

(3) Runs on:
Ueda FGHC compiler
Pseudo multi-PSI

(4) Description of the problem:

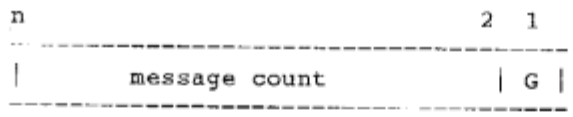
Given a network with a non-negative cost assigned to each edge, and a start node, find for each node the path with the minimum accumulated cost from the starting node.

(5) Algorithm:

First, read BESTPATH_SHORTCIRCUIT_ICHIYOSHI.DOC.
The basic algorithm is the same as this.
The difference is the detection of the finishing state.

This program detects the finishing state by the message counting scheme.
For a general description of the algorithm, see US2:<MPSI.BENCH>TERMDET.DOC.

This program represents the message count and the generation information in one integer (36 bits on DEC-20 and 32 bits on PSI).
The correspondence is as follows:



(6) Process structure:

One node in the network has one corresponding node process. A node process is connected to its neighboring node processes by one input and one output stream. There is one monitor process which continuously checks whether the computation has finished by sending the termination checking token.

(7) Pragma:

US2:<MPSI.BENCH>BESTPATH_TERMDET_ICHIYOSHI.KL1 (for Pseudo MPSI) has pragmas attached. The network is divided into a few segments and PEs are assigned to each of them.

(8) Program:

```
(main part)
        node/10:        Node process
        monitor/3:      Monitor process
        send_cp/6:      Subroutine for sending path/cost information packets
                to neighboring nodes
        closeOuts/1:    Closes a stream
        bp/4:           Top level goal
(network generation)
        augment_edges/2: Augments edge information to include both-way streams
        del/3:          Deletes an element from a list
        gen_nodes/4:    Node generation routine
        gen_node/7:     Makes a node process out of node information
(utility predicates)
        merge_all/2:    n-way merger
        merge/3:        Standard merger
        length/2,length/3: Gives the list length
        write_result/2: Writes out stream elements
        bp_snc/3:       Runs the program with the start node, set of nodes,
                        and set of edges given
        bp_ne/2:        Runs the program with the set of nodes,
                        and set of edges given
(test program)
        bp_ex1_n/1:     First example
        bp_ex2_n/1:     Second example
        bp_ex3_s/1:     Third example
```

(9) Source file:

US2:<MPSI.BENCH>BESTPATH_TERMDET_ICHIYOSHI.GHC (for Ueda compiler)
301 lines (111 lines of which are test data)
US2:<MPSI.BENCH>BESTPATH_TERMDET_ICHIYOSHI.KL1 (for Pseudo MPSI)
239 lines (50 lines of which are test data)

(10) Examples:

```
Invocation:
(To run example 1 with start node a)
        :- bp_ex1_n([a,b,c,d,e,f,g,h]).
(To run example 2 with start node a)
        :- bp_ex2_n([a,b,c,d,e,f,g,h,i,j,k,l]).
(To run example 3 with start node n0)
        :- bp_ex3_s(n0).
```

(11) Evaluation data: Not recorded

A Termination Checking Algorithm for Distributed Computation

Nobuyuki Ichiyoshi
June 1987

< Abstract >

An algorithm for checking termination of distributed computation is described, and a formal proof of its correctness is given. Each participating processor is assumed to keep a message count, and a special token for termination check visits all the processors and see if they are all idle and also sums up the messages counts to ensure the non-existence of outstanding messages in the network.

1. Introduction

2. Model of Computation

Def. 1.

A computation C is an 8-tuple (P,T,M,idle,sp,rp,st,rt) where

(1) a finite set of processors P
(2) a time space T (which can be identified as a subset of R)
(3) a predicate idle over P x T
(4) a set of inter-processor messages M
(5) four mappings

```
sp: M -> P
rp: M -> P
st: M -> T
rt: M -> T
```

Notation.

- Elements of P are called processors and are denoted by p1, p2, etc.
- Elements of M are called messages and are denoted by m1, m2, etc.
- Elements of T are denoted by t1, t2, etc.

(
intended meaning:

idle(p,t) ... Processor p is idle at time t.
sp(m) is the processor which sends the message m.
rp(m) is the processor which receives the message m.
st(m) the time at which the message is sent.
rt(m) the time at which the message is received.

implicit assumptions:

Every message is one to one (one sender, one receiver).
Every message is received in a finite time.
)

In the following discussion, an arbitrary computation
C = (P,T,M,idle,sp,rp,st,rt) is understood.

Axioms (Assumptions, Requirements, or whatever).

(1) (all m:M)(st(m) =< rt(m))
(2) (all t1,t2:T)([m:M | t1 =< st(m) < t2} is a finite set)
(3) (exists t:T)(all m:M)(st(m) >= t)
(4) (all p:P)(all t1,t2:T)(
(idle(p,t1) and (t1 =< t2) and
~(exists m:M)(rp(m) = p and t1 =< rt(m) < t2))
=> idle(p,t2)
)
(5) idle(p,t) => ~(exists m:M)(sp(m) = p and st(m) = t)

(
English translation:

(1) A Message is received after it is sent.
(2) Only a finite number of messages are sent in a finite interval of time.
(3) There is a point in time before which no message exists.
(4) An idle processor remains idle unless it receives a message.
(5) An idle processor does not send messages.
)

Cor. 1.

(1) (every t:T)({m:M | st(m) < t} is a finite set} (by Ax. 2 & 3)

Def. 2.

(1) IDLE(t) <=> (every p:P)(idle(p,t))
(2) IDLE*(t) <=> (every t':T)(t =< t' => IDLE(t'))
(3) IDLE^M(t) <-> IDLE(t) and (every m:M)(rt(m) < t)

(
English translation:

(1) IDLE(t) ... All processors are idle at t.
(2) IDLE*(t) ... All processors are idle at t and remain so.
(3) IDLE^M(t) ... All processors are idle at t and there are no outstanding messages.

IDLE*(t) or IDLE^M(t) means that the distributed computation terminated some time before t.
)

Cor. 2.

(1) (t1 =< t2) and IDLE^M(t1) => IDLE^M(t2) (by Ax. 4)
(2) IDLE^M(t) => IDLE*(t) (by Cor. 2.1)
(3) IDLE*(t) => (exists t':T)(IDLE^M(t')) (by Cor. 1.1)

3. Termination Checking Algorithm

4. Correctness of Algorithm

Notations.

(1) $mc(p,t) = \#\{m:M \mid sp(m) = p,\ st(m) < t\} - \#\{m:M \mid rp(m) = p,\ rt(m) < t\}$

(2) For $k:P\to T$,
$k\_ = \min k(P)$, $k^{\sim} = \max k(P)$.

(3) $Mo\to o(k) = \{m:M \mid st(m) < k(sp(m)),\ rt(m) < k(rp(m))\}$
$Mo\to n(k) = \{m:M \mid st(m) < k(sp(m)),\ rt(m) \geq k(rp(m))\}$
$Mn\to o(k) = \{m:M \mid st(m) \geq k(sp(m)),\ rt(m) < k(rp(m))\}$
$Mn\to n(k) = \{m:M \mid st(m) \geq k(sp(m)),\ rt(m) \geq k(rp(m))\}$

(4) $Mc(k) = \sum_{p:P} mc(p,k(p))$

(5) $Idle(k) \Leftrightarrow (\text{every } p:P)(\ idle(p,k(p))\ )$

(6) $Tc(k) \Leftrightarrow Idle(k)$ and $Mn\to o(k)=0$ and $Mc(k)=0$

(
intended meaning:

mc ... message count at processor p
(number of messages sent - number of messages received)

k(p) ... the time at which termination checking token passes p.

Mo->o(k), etc. ...
A message belongs to Mo->o(k) iff it is sent by a processor which has already received the token to a processor which has already received the token (i.e. it is a message from "past to past"), etc.

Mc(k) ... sum of message counts as observed by the token.

Idle(k) ... Every processor is idle at the time the token visits it.

Tc(k) ... Our termination checking criterion

Note that Tc(k) is "observable" by the token if
(i) every processor maintains the message count, and
(ii) there is a mechanism to determine that a received message was sent by a processor which the token had already visited.
)

Cor. 3.

(1) $Mc(k) = \#Mo\to n(k) - \#Mn\to o(k)$

Proposition. 1.

(1) $IDLE^{\sim}M(k\_) \Rightarrow Tc(k)$

(2) $Tc(k) \Rightarrow IDLE(k^{\sim})$

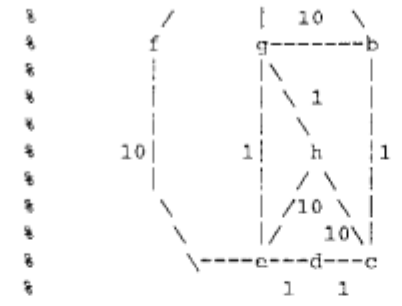
Proof:

Ask Ichiyoshi. [q.e.d.]

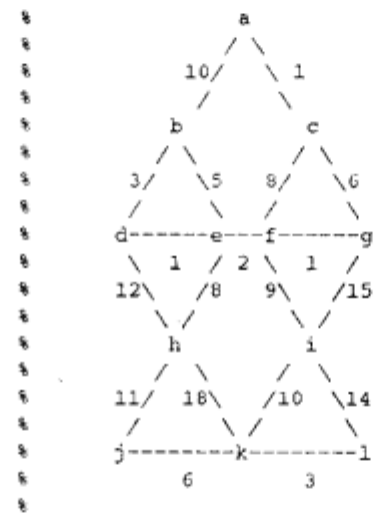
```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
%       Best Path Problem
%
%       N. Ichiyoshi
%       March 1987
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%       node process            %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% node(In,Outs,Cout,C,P,CPC,G, EC,Cs,N)
%
%       In      : input stream (including the control stream)
%       Outs    : output streams
%       Cout    : output control stream
%       C       : minimum cost so far
%       P       : path with cost C
%       CPC     : cp packet count
%       G       : generation
%       EC      : number of outgoing edges (constant)
%       Cs      : costs of outgoing edges (constant)
%       N       : node identifier (constant)
%
%       ( What G represents:                                                )
%       (       G = 2*G1 + G0, where G0 = 0 or 1.                           )
%       (       ( i) G0 is a 1-bit generation information which is          )
%       (            updated every time a new termination detection         )
%       (            packet visits the node.                                )
%       (       (ii) G1 = 0 iff the node has not received a message         )
%       (            (cp packet), since the last visit of a termination     )
%       (            detection token, which has a different generation      )
%       (            than that of the node. On receiving a message with     )
%       (            generation Gen, the node updates G1 to NewG1 as        )
%       (            according to the following formula:                    )
%       (                NewG1 = G1 or (G1 xor Gen).                        )
%       (            (G1 xor Gen = 1 if G1 \= Gen and 0 if G1 = Gen.)       )
%
node([cp(Cost,Path,Gen)|In],Outs,Cout,C,P,CPC,G, EC,Cs,N) :-
    Cost >= C,
    NewG := G+((G+Gen) mod 2)*2,
    NewCPC := CPC-1 |
    node(In,Outs,Cout,C,P,NewCPC,NewG, EC,Cs,N).
node([cp(Cost,Path,Gen)|In],Outs,Cout,C,P,CPC,G, EC,Cs,N) :-
    Cost < C,
    NewG := G+((G-Gen) mod 2)*2,
    NewCPC := CPC+EC-1
    send_cp(Outs,Cs,Cost,[N|Path],NewG, NewOuts),
    node(In,NewOuts,Cout,Cost,Path,NewCPC,NewG, EC,Cs,N).
node([donep(GCPC,NOG)|In],Outs,Cout,C,P,CPC,G, EC,Cs,N) :-
    NewNOG := NOG+(G/2)*2,
    NewG := NOG mod 2,
    NewGCPC := GCPC+CPC |
    Cout = [donep(NewGCPC,NewNOG)|Cout1],
    node(In,Outs,Cout1,C,P,CPC,NewG, EC,Cs,N).
node([done(NCs)|In],Outs,Cout,C,P,CPC,G, EC,Cs,N) :-
    true |
    Cout = [done([N-C-P|NCs])],
    closeOuts(Outs).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%       monitor process         %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
```

```
% monitor(Cin,Cout, R)
%
%       Cin     : input control stream
%       Cout    : output control stream
%       R       : result
%
monitor([donep(GCPC,NOG)|In],Cout, R) :-
    GCPC =\= -1,
    NewNOG := (NOG+1) mod 2 |
    Cout = [donep(0,NewNOG)|Cout1],
    monitor(In,Cout1, R).
monitor([donep(GCPC,NOG)|In],Cout, R) :-
    NOG/2 =\= 0,
    NewNOG := (NOG+1) mod 2 |
    Cout = [donep(0,NewNOG)|Cout1],
    monitor(In,Cout1, R).
monitor([donep(GCPC,NOG)|In],Cout, R) :-
    GCPC =:= -1,
    NOG/2 =:= 0 |
    Cout = [done([])|Cout1],
    monitor(In,Cout1, R).
monitor([done(NCs)|In],Cout, R) :-
    true |
    Cout = [],
    R = NCs.

send_cp([],_,_,_,_, NewOuts) :- true | NewOuts = [].
send_cp([Out|Outs],[C|Cs],Cost,NewPath,NewG, NewOuts) :-
    NewCost := Cost+C |
    Out = [cp(NewCost,NewPath,NewG)|Out1],
    NewOuts = [Out1|NewOuts1],
    send_cp(Outs,Cs,Cost,NewPath,NewG, NewOuts1).

closeOuts([]) :- true | true.
closeOuts([Out|Outs]) :- true | Out = [], closeOuts(Outs).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%       Process generation
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%
% bp(S,Ns,Es, R)
%
%       S       : start node
%       Ns      : list of nodes (a node has a constant value)
%       Es      : list of edges (an edge is of form e(Node1-Node2,Cost))
%       R       : result (list of nodes with the minimum cost from start node
%                 and a path with the minimum cost)
%
bp(S,Ns,Es, R) :  true |
    augment_edges(Es, AEs),
    del(S,Ns, Ms),
    gen_nodes([S|Ms],AEs,[cp(0,[],0)|C0],C),
    monitor([donep(99999,0)|C],C0, R).

augment_edges([], AEs) :- true | AEs = [].
augment_edges([e(N1-N2,C)|Es], AEs) :- true |
    AEs = [e(N1-N2,C,_-_)|AEs1],
    augment_edges(Es, AEs1).

del(N,[M|Ns], Ms) :- N = M | Ms = Ns.
del(N,[M|Ns], Ms) :- N \= M | Ms = [M|Ms1], del(N,Ns, Ms1).
del(N,[], Ms) :- true | Ms = [].

gen_nodes([N|Ns],AEs,C0,C) :- true |
    gen_node(N,AEs,[],[],[],C0,C1),
```

```
    gen_nodes(Ns,AEs,C1,C).
gen_nodes([],_,C0,C) :- true | C0 = C.

gen_node(N,[e(N-_,Cost,Self-Other)|AEs],Ins,Outs,Costs,Cin,Cout) :- true |
    Ins1 = [Self|Ins],
    Outs1 = [Other|Outs],
    Costs1 = [Cost|Costs],
    gen_node(N,AEs,Ins1,Outs1,Costs1,Cin,Cout).
gen_node(N,[e(_-N,Cost,Other-Self)|AEs],Ins,Outs,Costs,Cin,Cout) :- true |
    Ins1 = [Self|Ins],
    Outs1 = [Other|Outs],
    Costs1 = [Cost|Costs],
    gen_node(N,AEs,Ins1,Outs1,Costs1,Cin,Cout).
gen_node(N,[e(O1-O2,_,_)|AEs],Ins,Outs,Costs,Cin,Cout) :-
    N \= O1, N \= O2 |
    gen_node(N,AEs,Ins,Outs,Costs,Cin,Cout).
gen_node(N,[],Ins,Outs,Costs,Cin,Cout) :-
    true |
    length(Outs, EC),
    merge_all([Cin|Ins], In),
    node(In,Outs,Cout,99999,?,0,0, EC,Costs,N).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%       Utility
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

merge_all([], Out) :- true | Out = [].
merge_all([In|Ins], Out) :- true |
    merge(In,Out1, Out),
    merge_all(Ins, Out1).

merge([],Ys, Zs) :- true | Zs = Ys.
merge(Xs,[], Zs) :- true | Zs = Xs.
merge([X|Xs],Ys, Zs) :- true | Zs = [X|Zs1], merge(Xs,Ys, Zs1).
merge(Xs,[Y|Ys], Zs) :- true | Zs = [Y|Zs1], merge(Xs,Ys, Zs1).

length(L, N) :- true | length(L, 0,N).

length([], K,N) :- true | N = K.
length([_|L], K,N) :- K1 := K+1 | length(L, K1,N).

write_result([], O) :- true | O = [].
write_result([X|Xs], O) :- true |
    O = [write(X),nl|NewO], write_result(Xs, NewO).

%
% Bp_sne is given the start node, list of nodes and list of edges.
% Bp_ne is given list of nodes and list of edges. (The start node becomes
%  the first node in the list of nodes.)
%
bp_sne(Start,Nodes,Edges) :- true |
    bp(Start,Nodes,Edges, Result),
    write_result(Result, O),
    outstream(O).

bp_ne(Nodes,Edges) :- Nodes = [Start|_] |
    bp_sne(Start,Nodes,Edges).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%       Three examples
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%
%             ----a-----
%           /     |     \
%       10 /    10|      \ 1
```



```
bp_ex1_n(Nodes) :-
    true |
    bp_ne(Nodes,
      [e(a-f,10),e(a-g,10),e(a-b,1),
       e(b-g,10),e(b-c,1),
       e(c-h,10),e(c-d,1),
       e(d-e,1),
       e(e-h,10),e(e-f,10),e(e-g,1),
       e(g-h,1)
      ]).
```



```
bp_ex2_n(Nodes) :-
    true |
    bp_ne(Nodes,
      [e(a-b,10),e(a-c,1),
       e(b-d,3),e(b-e,5),
       e(c-f,8),e(c-g,6),
       e(d-e,1),e(d-h,12),
       e(e-f,2),e(e-h,8),
       e(f-g,1),e(f-i,9),
       e(g-i,15),
       e(h-j,11),e(h-k,18),
       e(i-k,10),e(i-l,14),
       e(j-k,6),
       e(k-l,3)
      ]).

bp_ex3_s(Start) :-
    true |
    bp_sne(Start,
      [n0,n1,n2,n3,n4,n5,n6,n7,n8,n9,
       n10,n11,n12,n13,n14,n15,n16,n17,n18,n19,
       n20,n21,n22,n23,n24,n25,n26,n27,n28,n29,
       n30,n31,n32,n33,n34,n35
```

```
],
[e(n0-n1,2),e(n0-n6,1),
 e(n1-n2,3),e(n1-n7,2),
 e(n2-n3,4),e(n2-n8,2),
 e(n3-n4,2),e(n3-n9,4),
 e(n4-n5,1),e(n4-n10,3),
 e(n5-n11,1),
 e(n6-n7,1),e(n6-n12,2),
 e(n7-n13,3),
 e(n8-n9,3),e(n8-n14,4),
 e(n9-n10,6),e(n9-n15,8),
 e(n10-n11,2),e(n10-n16,6),
 e(n11-n17,3),
 e(n12-n13,3),e(n12-n18,4),
 e(n13-n14,5),e(n13-n19,3),
 e(n14-n15,7),
 e(n15-n21,8),
 e(n16-n17,4),e(n16-n22,7),
 e(n17-n23,4),
 e(n18-n19,2),e(n18-n24,3),
 e(n19-n20,4),
 e(n20-n21,7),e(n20-n26,6),
 e(n21-n27,8),
 e(n22-n23,1),e(n22-n28,5),
 e(n23-n29,2),
 e(n24-n25,2),e(n24-n30,2),
 e(n25-n26,5),e(n25-n31,2),
 e(n26-n32,3),
 e(n27-n28,3),e(n27-n33,4),
 e(n28-n29,2),e(n28-n34,2),
 e(n29-n35,1),
 e(n30-n31,1),
 e(n31-n32,1),
 e(n32-n33,3),
 e(n33-n34,2),
 e(n34-n35,2)
]).
```

(0) Date: 1987-Jul-2, written by Masa Furuichi
Modified: 30-jul-87 by S. Takagi

(1) Program name: GRID (Grid Problem)

(2) Author: Masa Furuichi, Mitsubishi Electric Co. IEL Lab.

(3) Runs on: GHC Compiler System on DEC2065

(4) Description of the problem:

This is a prototype grid problem. Consider the grid shown in figure 1. The number in each cell is an ID, and is used to explain the problem.

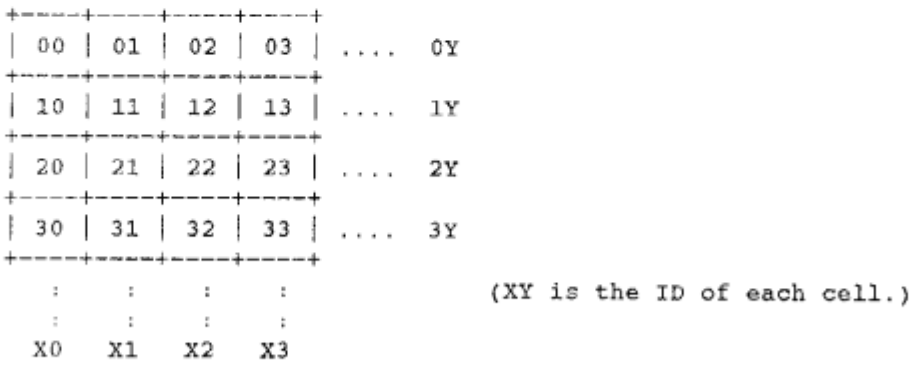


Figure 1

The essential characteristics of this class of grid problems are as follows:

1. The computation involves a cellular space.
   That is, there is a grid that divides the space into cells.

2. Each cell has a state characterized by an integer value.

3. There is a neighborhood function
   that defines the set of neighbors for a given cell.

4. Time is thought of as a discrete sequence such as t(1), t(2) ...
   There is a transition function that defines the state of a cell at time t(i+1) in terms of the state of the cell and its neighbors at time t(i).

In this program, a particularly simple case is chosen. The state of each cell, XY, is represented by integer value v(XY). However, the non-zero value of v(XY) is given only to the boundary cell, and 0 is given to the internal cells. The neighbors of a cell are the four cells that share the boundary faces of the cell. The transition function is defined as follows:

The states of the boundary cells keep constant values forever.

The state of the internal cells at time t(i+1) is the average of the states of the four neighboring cells at time t(i).

In this sample program, the following simple initial state is given in the program, but any kind of state can be executed by modifying the program.

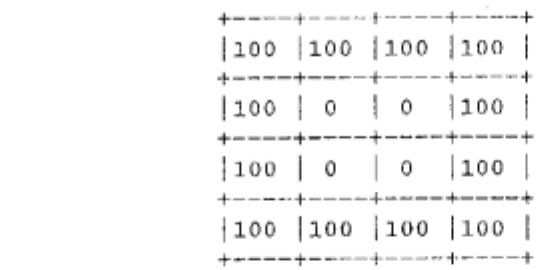


Figure 2: Initial State of Cells

(5) Algorithm:

The initial state of each cell is given as a list such as:

```
[[100,100,100,100],
 [100,  0,  0,100],
 [100,  0,  0,100],
 [100,100,100,100]]
```

The process forks for each cell, and each process is suspended at time t(i+1) until the state of the four neighbors' states at time t(i) is instantiated.

When the process is forked for each cell, it contains a stream which keeps the value of itself of each time t(i). It contains the four streams, and they are connected to the streams of another process. These four streams are used at time t(i+1) to check whether the states of the four neighbors at time t(i) are instantiated.

If the states of four neighbors at time t(i-1) are instantiated, the process is activated and calculates the state at time t(i), then it is suspended until the four states at time t(i) are instantiated.

(6) Process Structure:

Logically, each process which calculates the state value of time t(i) can be run on the different processors, but this requires N**2 processors when N**2 is the size of a grid. However, this program can currently run only on GHC system on DEC2065. All the processes are forked to different processors.

(7) Pragma: None

(8) Program:

The program is composed of the following three parts:

[A] Generate Grid Process

Generate a grid process for each cell, and connect the streams to the grid process of the four neighbors.

go/2:	Top level predicate to start this program
start/3:	Set up the parameter and call gen_grid/5
gen_grid/5:	Generate a stream for the cell and four streams for the neighbors, and connect them. The connection of streams for the boundary cells is done in this predicate, and for the internal cells in gen_each_grid/7.
gen_each_grid/7:	Connect the streams of the cell and the four neighbors, then fork process grid/7.

[B] State Calculation

Suspend at time t(i+1) until the four neighbors' states at time t(i) are instantiated. When the states are instantiated, the process is activated and calculates the state at time t(i+1), then it is suspended again.

grid/7:	The main process which generates the state value at time t(i) until i is the specified limit number.
calculate/9:	Calculate the state value at time t(i+1)
value/3:	Suspend until one neighbor's state is instantiated.

[C] Output

To display the state value at time t(1), t(2), t(n)..., this program uses some terminal control functions of the VT100 terminal. It has cosmetic value only.

(9) Source file: US2:<MPSI.BENCH>GRID_FURUICHI.GHC
This document: US2:<MPSI.BENCH>GRID_FURUICHI.DOC

(10) Examples:

Invoke the GHC system on the DEC2065:

```
@ GHC<CR>
```

Compile the program:

```
?- ghccompile('GRID.GHC').<CR>  % Note that the file must be
                                % renamed before it is compiled.
```

Invoke the program:

```
?- ghc go(1,50).<CR>            % The first argument of go/2
                                % is the initial state
                                % pattern.  1,2 and 3 are provided.
                                % The second argument of go/2
                                % is the iteration number.
```

The state of each cell at time t(i) is displayed
on the terminal until i is 50.

The following initial state patterns are provided.

1.
```
[[100,100,100,100],
 [100,  0,  0,100],
 [100,  0,  0,100],
 [100,100,100,100]]
```

2.
```
[[100, 90, 80, 70, 60, 50],
 [ 90,  0,  0,  0,  0, 40],
 [ 80,  0,  0,  0,  0, 30],
 [ 70,  0,  0,  0,  0, 20],
 [ 60,  0,  0,  0,  0, 10],
 [ 50, 40, 30, 20, 10,  0]]
```

3.
```
[[100,100,100,100,100],
 [100,  0,  0,  0,100],
 [100,  0,  0,  0,100],
 [100,  0,  0,  0,100],
 [100,100,100,100,100]]
```

(11) Evaluation data: Not recorded yet

```
:- public go/2.

go(1,It):- prolog((ttyput(27),write('[2J'),ttyput(27),write('[H'))),
        InitList =      [[100,100,100,100],
                         [100,  0,  0,100],
                         [100,  0,  0,100],
                         [100,100,100,100]],
        start(It,InitList,Result),
        write(Result,1,20).

start(It,InitList,Result):- true,
        gen_grid(InitList,[],Result,It,1).

%%%%%%%%%%%%%%%%%%%%
gen_grid([Row|Next],S0,S,It,1):- true,          % North Boundary
        north_boundary(Row,S1,It),
        S=[S1|S2],gen_grid(Next,S1,S2,It,2).
gen_grid([Row|[]],S0,S,It,Ro):- Ro\=1,                  % South boundary
        south_boundary(Row,S0,S1,It,1), S=[S1].
gen_grid([Row|Next],S0,S,It,Ro):- Next\=[],Ro\=1,Ro1:=Ro+1,
        gen_each_grid(Row,S0,[],S1,It,Ro,1),S=[S1|S2],
        gen_grid(Next,S1,S2,It,Ro1).

%%%%%%%%%%%%%%%%%%%%
gen_each_grid([InSt|Rest],[(_,_,_,_,_)|NeiRowRest],[],S0,It,Ro,C):-C1:=C+1,
        G=[InSt|_],S1=(G,G,G,G,G),S0=[S1|S2],   % West Boundary
        grid(1,It,G, G,G,G,G),
        gen_each_grid(Rest,NeiRowRest,S1,S2,It,Ro,C1).
gen_each_grid([InSt|[]],                        % East Boundary
        [(Gu,Nu,Eu,Su,Wu)|NeiRowRest],          % Ue
        ( Gl,Nl,El,Sl,Wl),S0,It,Ro,C):-C1:=C+1, % Hidari
        G=[InSt|_],     S1=(G,N,E,S,W),S0=[S1],
        grid(1,It,G, N,E,S,W),
        N=G,E=G,S=G,W=G,El=G.
gen_each_grid([InSt|Rest],                      % Other(Colum=2)
        [(Gu,Nu,Eu,Su,Wu)|NeiRowRest],          % Ue
        ( Gl,Nl,El,Sl,Wl),S0,It,Ro,2):-Rest\=[],Ro\=2,  % Hidari
        G=[InSt|_],     S1=(G,N,E,S,W),S0=[S1|S2],
        N=Gu,W=Gl,Su=G, %El=G,
        grid(1,It,G, N,E,S,W),
        gen_each_grid(Rest,NeiRowRest,S1,S2,It,Ro,3).
gen_each_grid([InSt|Rest],                      % Other(Row=2)
        [(Gu,Nu,Eu,Su,Wu)|NeiRowRest],          % Ue
        ( Gl,Nl,El,Sl,Wl),S0,It,2,C):-C1:=C+1,Rest\=[],C\=2, % Hidari
        G=[InSt|_],     S1=(G,N,E,S,W),S0=[S1|S2],
        N=Gu,W=Gl, El=G, % Su=G,
        grid(1,It,G, N,E,S,W),
        gen_each_grid(Rest,NeiRowRest,S1,S2,It,2,C1).
gen_each_grid([InSt|Rest],                      % Other(Col=Row=2)
        [(Gu,Nu,Eu,Su,Wu)|NeiRowRest],          % Ue
        ( Gl,Nl,El,Sl,Wl),S0,It,2,2):-Rest\=[], % Hidari
        G=[InSt|_],     S1=(G,N,E,S,W),S0=[S1|S2],
        N=Gu,W=Gl, % El=G, % Su=G,
        grid(1,It,G, N,E,S,W),
        gen_each_grid(Rest,NeiRowRest,S1,S2,It,2,3).
gen_each_grid([InSt|Rest],                      % Other(Col\=Row\=2)
        [(Gu,Nu,Eu,Su,Wu)|NeiRowRest],          % Ue
        ( Gl,Nl,El,Sl,Wl),S0,It,Ro,C):-Ro\=2,C\=2,C1:=C+1,Rest\=[],% Hidari
        G=[InSt|_],     S1=(G,N,E,S,W),S0=[S1|S2],
        N=Gu,W=Gl,El=G,Su=G,
        grid(1,It,G, N,E,S,W),
        gen_each_grid(Rest,NeiRowRest,S1,S2,It,Ro,C1).


%%%%%%%%%%%%%%%%%%%%
north_boundary([],S0,It):- true.
```

```
        S0=[].
north_boundary([InSt|Rest],S0,It):-true|
        G=[InSt|_],S1=(G,G,G,G,G),S0=[S1|S2],
        grid(1,It,G, G,G,G,G),
        north_boundary(Rest,S2,It).
%%%%%%%%%%%%%%%%%%%%
south_boundary([InSt|[]],[(Gu,Nu,Eu,Su,Wu)|NeiRowRest],S0,It,_):- true|
        G=[InSt|_],S1=(G,G,G,G,G),S0=[S1],
        grid(1,It,G, G,G,G,G).
south_boundary([InSt|Rest],[(Gu,Nu,Eu,Su,Wu)|NeiRowRest],S0,It,1):- true|
        G=[InSt|_],S1=(G,G,G,G,G),S0=[S1|S2],
        grid(1,It,G, G,G,G,G),
        south_boundary(Rest,NeiRowRest,S2,It,2).
south_boundary([InSt|Rest],[(Gu,Nu,Eu,Su,Wu)|NeiRowRest],S0,It,Ro):-
        Ro\=1,Rest\=[],Ro2:=Ro+1|
        G=[InSt|_],S1=(G,G,G,G,G),S0=[S1|S2],
        Su=G,
        grid(1,It,G, G,G,G,G),
        south_boundary(Rest,NeiRowRest,S2,It,Ro2).
%%%%%%%%%%%%%%%%%%%%
grid(Time,Iteration,[_|T],N,W,S,E):-            % Stop condition.
        Time =:= Iteration | T = [].
grid(Time,Iteration,[_|T],N,W,S,E):- Time=\=Iteration|
        calculate(N,W,S,E, NT,WT,ST,ET, State), % Calculate the state value.
        T = [State|_],
        Time1 := Time+1,
        grid(Time1,Iteration,T, NT,WT,ST,ET).

calculate(N,W,S,E, NT,WT,ST,ET, Value):- true |
        value(N,NT,V1),                         % Suspend until the neiborhood
        value(W,WT,V2),                         % states are bound.
        value(S,ST,V3),                         %         ''
        value(E,ET,V4),                         %         ''
        Value := (V1+V2+V3+V4)/4.               % Neiborhood function.

value([H|T],OUT,Value):- true |
        Value = H, OUT = T.

write([H|Rest],X,Y):- X1:=X+1|
        write_row(H,X,Y),write(Rest,X1,Y).
write([],X,Y):- true|true.

write_row([(G,_,_,_,_)|RestGrid],X,Y):- wait(G),Y1:=Y+4|
        write_grid(G,X,Y),write_row(RestGrid,X,Y1).
write_row([],X,Y):-  true|true.

write_grid([G|Rest],X,Y):-
        wait(G)|
        display(X,Y,G),
        write_grid(Rest,X,Y).
write_grid([],X,Y):- true|true.

display(X,Y,G):-
        prolog((ttyput(27),write('['),write(X),write(';'),write(Y),write('H'),
                write(G),
                ttyput(27),write('['),write(15),write(';'),write(1),write('H'),
                write((X,Y))  ))|true.


%%%%%%%%%%%%%%%%%
%  SAMPLE DATA %
%%%%%%%%%%%%%%%%%

go(2,It):- prolog((ttyput(27),write('[2J'),ttyput(27),write('[H')))|
        InitList =      [[100, 90, 80, 70, 60, 50],
                         [ 90,  0,  0,  0,  0, 40],
```

```
                        [ 80,  0,  0,  0,  0, 30],
                        [ 70,  0,  0,  0,  0, 20],
                        [ 60,  0,  0,  0,  0, 10],
                        [ 50, 40, 30, 20, 10,  0]],
        start(It,InitList,Result),
        write(Result,1,10).
go(3,It):- prolog((ttyput(27),write('[2J'),ttyput(27),write('[H'))),
        InitList =      [[100,100,100,100,100],
                        [100,  0,  0,  0,100],
                        [100,  0,  0,  0,100],
                        [100,  0,  0,  0,100],
                        [100,100,100,100,100]],
        start(It,InitList,Result),
        write(Result,1,10).
```

% <MPSI.BENCH>GOODPATH_LAYERED.DOC.5, 4-Aug-87 11:38:47, Edit by OKUMURA

(0) Date: 16-Jul-87, written by K. Wada
    Modified: 27-Jul-87 by A. Okumura
    Modified: 4-Aug-87 by S. Takagi

(1) Program name: goodpath_layered
                  (good path problem using the layered-stream method)

(2) Author: A. Okumura, ICOT 1st Lab.

(3) Runs on: GHC system on DECsystem-20

(4) Description of the problem:

The good path problem is to find all acyclic paths from the start node to the goal node in a given graph.

(5) Algorithm:

Node names are propagated in sequences along edges. When a node receives a sequence which includes the name of the node, the sequence is eliminated. If the goal node receives a sequence from the start node which does not include the name of the goal node, the sequence is output as an answer.

(6) Process structure:

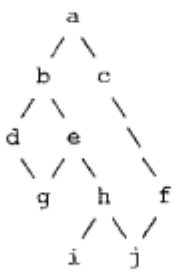
One process is generated for each node. A node process prepares a pair consisting of its node name and a layered stream as an output. The stream holds good paths from the start node to that node. The layered stream is made from the input from its neighbors by eliminating paths which contain the current node name.

(7) Pragma: Not set yet

(8) Program:

Top-level predicate is goodPath/3.
    This predicate defines the configuration of the graph,
    generates a process for each node of the graph,
    and sets up layered streams along the edges.
node/6 produces the primary output bindings for each node.
    The first clause of node/6 is for the starting node.
    The second clause is for the goal node.
filter/3 is for filtering out paths that include loops.

The following graph is defined in this program:



(9) Source file: US2:<MPSI.BENCH>GOODPATH_LAYERED.GHC
    This document: US2:<MPSI.BENCH>GOODPATH_LAYERED.DOC

(10) Examples:

Invocation:

```
goodPath(Start,Goal,Path).
        where Start is the starting node,
        Goal is the goal node,
        and all good paths between Start and Goal are obtained
        as the third argument, Path.
```

For example, run goodPath(a,h,Path). The following result is obtained.

```
Path = h*[e*[b*[a*begin,d*[g*[]]],
            g*[d*[b*[a*begin]] ] ],
          i*[],
          j*[f*[c*[a*begin]]]       ].
```

It means that good paths between nodes a and h are :

```
a-b-e-h,
a-b-d-g-e-h,
a-c-f j h.
```

Note: The output of the value of Path is not formatted as shown above.
The formatting has been done by hand for legibility.

(11) Evaluation data: Refer to the data in PSM-O-A-KL1-005.

```
%  goodpath problem using layered stream

goodPath(Start, Goal, Path) :- true |
        node(Start, Goal, a, [B,C],   A, Path),
        node(Start, Goal, b, [A,E,D], B, Path),
        node(Start, Goal, c, [A,F],   C, Path),
        node(Start, Goal, d, [B,G],   D, Path),
        node(Start, Goal, e, [B,G,H], E, Path),
        node(Start, Goal, f, [C,J],   F, Path),
        node(Start, Goal, g, [D,E],   G, Path),
        node(Start, Goal, h, [E,I,J], H, Path),
        node(Start, Goal, i, [H],     I, Path),
        node(Start, Goal, j, [F,H],   J, Path).

node(Start, _, Start, _, Out, _) :- true | Out = Start*begin.
node(_, Goal, Goal, In, Out, Path) :- true |
        Out = [], Path = Goal*In.
node(Start, Goal, Node, In, Out, _) :- Start \= Node, Goal \= Node |
        Out = Node*In1, filter(In, Node, In1).

filter(begin, _, Out) :- true | Out = begin.
filter([], _, Out) :- true | Out = [].
filter([Node*_|In], Node, Out) :- true | filter(In, Node, Out).
filter([[]|In], Node, Out) :- true | filter(In, Node, Out).
filter([N*Ns|In], Node,Out) :- Node \= N |
        Out = [N*Ns1|Out1], filter(Ns, Node, Ns1), filter(In, Node, Out1).
```

(0) Date: 1987-Jul-10, written by E. Sugino
Modified: 29-Jul-87 by S. Takagi
Modified: 12-Aug-87 by E. Sugino
Modified: 13-Aug-87 by S. Takagi

(1) Program name: waltz

(2) Author: E. Sugino, ICOT 4th lab.

(3) Runs on: GHC1, GHC2 and GHC3 on DEC2065

(4) Description of the problem:

The labeling problem is to analyze a line drawing.
All lines are classified into three types.

a) A border line which has a direction.
The way to determine a direction is:
when you walk along a line looking at an object, your forward direction is decided as one direction. One direction is going out from the node, the other is coming into the node.
These are described as 'out' and 'in' in this program.
b) Convex line which is described as 'p' ('+')
c) Concave line which is described as 'm' ('-')

Generally speaking, nodes (junctions) of an object are classified into 9 types. Only four types:

'arrow' /|\ , 'l' \/ , 'fork' Y , and 't' -|

are enough to classify objects on the assumption of:

a) No cracks or shadows;
b) All of the vertices are made with only three planes;
c) The properties of junctions of any line drawing are not changed when the viewpoint is moved.

(5) Algorithm:

Each node has several possible candidates for the type of junctions.
Each node sends the possible types of its own hands to the neighbors.
Candidates for the types of nodes will be restricted to fewer types by the neighbors' types. When candidates for the types of a node are restricted to fewer types, the node sends the restricted possible types of its own hands to the neighbors. Finally, any node will be restricted to only one type (if the line drawing is correct).

(6) Process structure:

Each node is a process. They are connected to each other.

(7) Pragma: None

(8) Program:

waltz1, waltz2, waltz3 and waltz4 are sample programs.
The line drawings in these programs are seen in
  'Preira Si ga teiansita 5tu no mondai nituite' (PSM-O-A-KL1-003).

waltz/3 needs the information of the junctions and the borders.
    The first argument is a list for information of the junctions,
    any element of which is the form:
        p(type of the node,
          left node number,
          back node number,
          right node number,
          node number)
    The second argument is a list of information of the borders,
    which consists of border node numbers.

Hands of junctions are connected by conv in ana.
conv1 gives the information that
    if a junction is a border,
    the types of its hands must be in or out.
Junctions are released as processes by waltz_go.
p1 is the first process
    which sends the information of its type to the neighbors.
p is a process for a node.
    When its node type is restricted to only one,
    the node decides its type.
p waits for information from neighbors.
    When a message has come, p checks whether its node type
    candidates can be restricted by the information.
    Then p replies by sending its new candidates to the neighbors
    if the candidates are restricted.

(9) Source file: us2:<mpsi.bench>labeling_sugino_579.ghc
    This document: us2:<mpsi.bench>labeling_sugino_579.doc

(10) Example:

Sample programs are invoked as follows.

```
        ?- ghc waltz1.          % Simple Cube (fig. 4-8 in PSM-I-A-KL1-003)

        ?- ghc waltz2.          % (fig. 4-9 in PSM-I-A-KL1-003)

        ?- ghc waltz3.          % (fig. 4-6.b in PSM-I-A-KL1-003)

        ?- ghc waltz4.          % (fig. 4-6.a in PSM-I-A-KL1-003)
```

If you want to try another line drawing, use waltz/3.
See waltz1 or others.

(11) Evaluation data: Not recorded

```
waltz1 :- true | waltz([p(a,1,7,2,1),p(l,3,0,2,2),p(a,3,8,4,3),p(f,7,9,8,4),
                        p(l,5,0,4,5),p(a,5,9,6,6),p(l,1,0,6,7)],
                       [1,2,3,4,5,6],
                       Result),
        wr(Result).
waltz2 :- true |
        waltz([p(a,1,9,2,1),p(l,3,0,2,2),p(t,4,3,14,3),
             p(l,5,0,4,4),p(a,5,15,6,5),p(l,7,0,6,6),
             p(a,7,17,8,7),p(l,1,0,8,8),p(f,9,17,10,9),
             p(a,16,11,10,10),p(l,12,0,11,11),p(a,12,13,14,12),
             p(f,13,16,15,13)],
        [1,2,3,4,5,6,7,8],
            Result),
        wr(Result).
waltz3 :- true |
        waltz([p(a,6,7,1,1),p(l,2,0,1,2),p(a,2,8,3,3),
             p(l,4,0,3,4),p(a,4,9,5,5),p(l,6,0,5,6),
             p(f,7,15,10,7),p(a,10,16,11,8),p(f,11,12,8,9),
             p(a,12,17,13,10),p(f,14,13,9,11),p(a,14,18,15,12),
             p(f,18,17,16,13)],
        [1,2,3,4,5,6],
            Result),
        wr(Result).
waltz4 :- true |
        waltz([p(a,8,9,1,1),p(l,2,0,1,2),p(a,2,12,3,3),p(l,4,0,3,4),
               p(a,4,13,5,5),p(f,5,14,6,6),p(a,6,15,7,7),p(l,8,0,7,8),
               p(f,9,15,10,9),p(a,10,14,11,10),p(f,11,13,12,11)],
               [1,2,3,4,5,6,7,8],
               Result),
        wr(Result).

waltz1(Result) :- true |
        waltz([p(a,1,7,2,1),p(l,3,0,2,2),p(a,3,8,4,3),p(f,7,9,8,4),
                p(l,5,0,4,5),p(a,5,9,6,6),p(l,1,0,6,7)],
               [1,2,3,4,5,6],
               Result).
waltz2(Result) :- true |
        waltz([p(a,1,9,2,1),p(l,3,0,2,2),p(t,4,3,14,3),
             p(l,5,0,4,4),p(a,5,15,6,5),p(l,7,0,6,6),
             p(a,7,17,8,7),p(l,1,0,8,8),p(f,9,17,10,9),
             p(a,16,11,10,10),p(l,12,0,11,11),p(a,12,13,14,12),
             p(f,13,16,15,13)],
        [1,2,3,4,5,6,7,8],
            Result).
waltz3(Result) :- true |
        waltz([p(a,6,7,1,1),p(l,2,0,1,2),p(a,2,8,3,3),
             p(l,4,0,3,4),p(a,4,9,5,5),p(l,6,0,5,6),
             p(f,7,15,10,7),p(a,10,16,11,8),p(f,11,12,8,9),
             p(a,12,17,13,10),p(f,14,13,9,11),p(a,14,18,15,12),
             p(f,18,17,16,13)],
        [1,2,3,4,5,6],
            Result).

ana([p(Type,L,D,R,N)|Rest],Result,Kyoukai,Vars,NVar) :- true |
        conv(L,LL,Kyoukai,Vars,V0),
        conv(D,DD,Kyoukai,V0,V1),
        conv(R,RR,Kyoukai,V1,V2),
        Result = [p(Type,LL,DD,RR,N) | Resultt],
        ana(Rest,Resultt,Kyoukai,V2,NVar).
ana([],Result,Kyoukai,Vars,V0) :- true | Result =[],V0=[].

conv(0,V,K,L,NV) :- true | V = _-_,NV=L.
conv(N,V,K,[N-Var|L],NV) :- N > 0 | V=Var,NV=L.
conv(N,V,K,[M-Var|L],NV) :- N > 0,N =\= M |
                conv(N,V,K,L,NV1),NV=[M-Var|NV1].
```

```
conv(N,V,K,[],NV) :- N > 0 |
                conv1(K,N,V,NV).

conv1([A|B],A,V,NV) :- true | V = [[in,out]|X]-Y,NV=[A-(Y-X)].
conv1([A|B],C,V,NV) :- A =\= C | conv1(B,C,V,NV).
conv1([],C,V,NV) :- true | V = A-B,NV=[C-(B-A)].

wr([]) :- true | true.
wr([A-T-X|B]) :- wait(X),prolog((write(A-T-X),nl)) | wr(B).

waltz(List,K,Result) :- true |
                ana(List,L,K,[],_),
                waltz_go(L,Result).

waltz_go([ p(Type,L,D,R,N) | Rest],Out) :- true |
                junctions(Type,Types),
                p1(L,D,R,Types,Result),
                Out = [N-Type-Result | Ot],
                waltz_go(Rest,Ot).
waltz_go([],O) :- true | O=[].


p(L,D,R,[],Type) :- true |  Type =[(?,?,?)].
p(L-LL,D-DD,R-RR,[(L1,D1,R1)],Type) :- true |
                Type=[(L1,D1,R1)],LL=[L1],DD=[D1],RR=[R1].
p(L,D,[R|Rt]-RR,[T1,T2|T3],Type) :- prolog(atom(R)) |
                filter(r,[T1,T2|T3],R,New),
                ack(r,[T1,T2|T3],New,RR,RRt),
                p(L,D,Rt-RRt,New,Type).
p([L|Lt]-LL,D,R,[T1,T2|T3],Type) :- prolog(atom(L)) |
                filter(l,[T1,T2|T3],L,New),
                ack(l,[T1,T2|T3],New,LL,LLt),
                p(Lt-LLt,D,R,New,Type).
p(L,[D|Dt]-DD,R,[T1,T2|T3],Type) :- prolog(atom(D)) |
                filter(d,[T1,T2|T3],D,New),
                ack(d,[T1,T2|T3],New,DD,DDt),
                p(L,Dt-DDt,R,New,Type).

p([[L1|L2]|Lt]-LL,D,R,Types,Type) :- true |
                filter(l,Types,[L1|L2],New),
                ack(l,Types,New,LL,LLt),
                p(Lt-LLt,D,R,New,Type).
p(L,[[D1|D2]|Dt]-DD,R,Types,Type) :- true |
                filter(d,Types,[D1|D2],New),
                ack(d,Types,New,DD,DDt),
                p(L,Dt-DDt,R,New,Type).
p(L,D,[[R1|R2]|Rt]-RR,Types,Type) :- true |
                filter(r,Types,[R1|R2],New),
                ack(r,Types,New,RR,RRt),
                p(L,D,Rt-RRt,New,Type).

p1(L,D,R,(Left,Down,Right),Type) :- true |
                send(Left,L,LL),
                send(Down,D,DD),
                send(Right,R,RR),
                p(LL,DD,RR,(Left,Down,Right),Type).

send(Hand,X-To,New) :- true | send1(Hand,To1),To=[To1|T],New = X-T.
send1([[P|List]|B],To) :- true | To=[P|T],send1(B,T).
send1([],To) :- true | To = [].

ack(P,Old,List,Send,St) :- true |
                same(Old,List,X),ack_s(X,P,List,Send,St).
ack_s(y,P,List,Send,St) :- true | Send=[L|St],ack1(P,List,L,[]).
ack_s(n,P,List,Send,St) :- true | Send = St.
```

```
same([A|B],[AA|BB],X) :- true  | same(B,BB,X).
same([],[],X) :- true | X=y.
same([],A,X) :- A \= [] | X=n.
same(A,[],X) :- A \= [] | X=n.
same(A,B,X) :- A \= [_|_] | X = n.
same(B,A,X) :- A \= [_|_] | X = n.

ack1(l,[(L1,_,_)|LL],Send,M) :- true |
        filtack(L1,M,M1,Send,S1),ack1(l,LL,S1,M1).
ack1(d,[(_,L1,_)|LL],Send,M) :- true |
        filtack(L1,M,M1,Send,S1),ack1(d,LL,S1,M1).
ack1(r,[(_,_,L1)|LL],Send,M) :- true |
        filtack(L1,M,M1,Send,S1),ack1(r,LL,S1,M1).
ack1(_,[],Se,_) :- true | Se=[].

filtack(A,[A|B],X,Send,St) :- true | Send = St,X=[A|B].
filtack(A,[C|B],X,Send,St) :- A \= C | filtack(A,B,X1,Send,St),X=[C|X1].
filtack(A,[],X,Send,St) :- true | Send=[A|St],X=[A].


junctions(l,Candidates) :-
        L1=(out,nil,in),L2=(in,nil,out),L3=(p,nil,out),
        L4=(in,nil,p),L5=(m,nil,in),L6=(out,nil,m) |
        Candidates =
                ([[out,[L1,L6]],[in,[L2,L4]],[p,[L3]],[m,[L5]]],
                 [],
                 [[in,[L1,L5]],[out,[L2,L3]],[p,[L4]],[m,[L6]]]).
junctions(f,Candidates) :-
        F1=(p,p,p),F2=(m,m,m),F3=(in,m,out),F4=(m,out,in),F5=(out,in,m) |
        Candidates =
                ([[p,[F1]],[m,[F2,F4]],[in,[F3]],[out,[F5]]],
                 [[p,[F1]],[m,[F2,F3]],[out,[F4]],[in,[F5]]],
                 [[p,[F1]],[m,[F2,F5]],[out,[F3]],[in,[F4]]]).
junctions(t,Candidates) :-
        T1=(out,p,in),T2=(out,m,in),T3=(out,in,in),T4=(out,out,in) |
        Candidates =
                ([[out,[T1,T2,T3,T4]]],
                 [[p,[T1]],[m,[T2]],[in,[T3]],[out,[T4]]],
                 [[in,[T1,T2,T3,T4]]]).
junctions(a,Candidates) :-
        A1=(in,p,out),A2=(m,p,m),A3=(p,m,p) |
        Candidates =
                ([[in,[A1]],[m,[A2]],[p,[A3]]],
                 [[p,[A1,A2]],[m,[A3]]],
                 [[out,[A1]],[m,[A2]],[p,[A3]]]).

% filter([l,d,r],Input,[in,out,p,m],Out)

filter(l,(L,_,_),T,F) :- true | filter1(T,L,F).
filter(d,(_,D,_),T,F) :- true | filter1(T,D,F).
filter(r,(_,_,R),T,F) :- true | filter1(T,R,F).
filter(P,[A|B],T,F) :- true | filter(P,T,A,F,Ft),filter(P,B,T,Ft).
filter(P,[],T,F) :- true | F=[].

% filter([l,d,r],[in,out,p,m],Type,Out,Out_tail)
% filter( List  ,     ..     , .. , . ,  ..    )

filter(l,T,(L,D,R),F,Ft) :- prolog(atom(T)) | filter2(T,L,(L,D,R),F,Ft).
filter(d,T,(L,D,R),F,Ft) :- prolog(atom(T)) | filter2(T,D,(L,D,R),F,Ft).
filter(r,T,(L,D,R),F,Ft) :- prolog(atom(T)) | filter2(T,R,(L,D,R),F,Ft).
filter(P,[A],Type,F,Ft) :- true | filter(P,A,Type,F,Ft).
filter(P,[A,B|C],Type,F,Ft) :- true |
                filter(P,A,Type,F,Ft1),
                filter(P,[B|C],Type,Ft1,Ft).

filter1(p  ,[[p  ,L]|_],F) :- true | F=L.
```

```
filter1(m  ,[[m  ,L]|_],F) :- true | F=L.
filter1(in ,[[out,L]|_],F) :- true | F=L.
filter1(out,[[in ,L]|_],F) :- true | F=L.
filter1(p  ,[[T|_]|Rest],F) :- T \= p | filter1(p,Rest,F).
filter1(m  ,[[T|_]|Rest],F) :- T \= m | filter1(m,Rest,F).
filter1(in ,[[T|_]|Rest],F) :- T \= out | filter1(in,Rest,F).
filter1(out,[[T|_]|Rest],F) :- T \= in  | filter1(out,Rest,F).
filter1(_,  [],          F) :- true | F=[].
filter1([A|B],Types,F) :- true | filter1(A,Types,F1),filter1(B,Types,F2),
                                 append(F1,F2,F).
filter1([],_,F) :- true | F=[].

append([],X,Y) :- true | X=Y.
append([X|Y],L,Z) :- true | Z = [X|ZZ],append(Y,L,ZZ).

filter2(p,p,Type,F,Ft) :- true | F = [Type|Ft].
filter2(m,m,Type,F,Ft) :- true | F = [Type|Ft].
filter2(in,out,Type,F,Ft) :- true | F = [Type|Ft].
filter2(out,in,Type,F,Ft) :- true | F = [Type|Ft].
filter2(p,X,Type,F,Ft) :- X \= p | F = Ft.
filter2(m,X,Type,F,Ft) :- X \= m | F = Ft.
filter2(in,X,Type,F,Ft) :- X \= out | F = Ft.
filter2(out,X,Type,F,Ft) :- X \= in | F = Ft.
```

(0) Date: 1987-Jul-18, written by E. Sugino
Modified: 29-Jul-87 by S. Takagi
Modified: 12-Aug-87 by E. Sugino
Modified: 13-Aug-87 by S. Takagi

(1) Program name: life

(2) Author: E.Sugino, ICOT 4th lab.

(3) Runs on: GHC1 on DEC2065

(4) Description of the problem: The life game is a popular game.

(5) Algorithm:

Any node waits for the state of all neighbors.
A node changes its state as follows:
When a node is alive and there are 2 or 3 live nodes
around the node, the node is alive at the next time period.
When a node was dead and there were 3 live nodes around it,
the node becomes live at the next time period.
In other cases, the node becomes dead.

(6) Process structure:

Each node is a process. They are connected to each other.

(7) Pragma: None

(8) Program:

life_game/3: A predicate to invoke a life game pattern.
The first argument (M) is the number of cycles in the pattern.
The second argument is a list of integer 0 or 1, which
makes a pattern with M cycles. The last argument (N)
is a life cycle, and the pattern of the N-th time
period displayed.
wr/3: Writes a pattern.

The following predicates make a mesh.
make_mesh, make_node, mg, temp_node, con,
parts of node (clauses in which the 11th argument is wait)

The node processes are nodes of the pattern.

The rest of the nodes make themselves the nodes of the next time
period. They are over at the life cycle, and a token, result(S,St),
is sent through the control stream of the nodes to obtain the last
pattern.

(9) Source file: US2:<MPSI.BENCH>LIFE_SUGINO_580.GHC
This document: US2:<MPSI.BENCH>LIFE_SUGINO_580.DOC

(10) Example:

A sample program glider:

```
        ?- ghc life_test(1).      % A glider: Original pattern
                                  % at the home position
        ?- ghc life_test(5).      % Every 4 cycles, the glider appears
                                  % at another position.
        ?- ghc life_test(9).

        ....................

        ?- ghc life_test(21).
```

If you want to try another pattern, use life_game/2.
See life_test/1.

(11) Evaluation data: Not recorded

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%          LIFE GAME  Program     1987.2.24. by E.Sugino                    %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

life_game(X,List,Max) :- true | make_mesh(X,List,Max,Result),
        wr(1,X,Result).

%%%%%%%%%%%%%%%%%%%%%%%%%%%
%         Writer            %
%%%%%%%%%%%%%%%%%%%%%%%%%%%
wr(N,M,[A|B]) :- N =< M,prolog((tab(5),write(A))),N1 := N + 1 | wr(N1,M,B).
wr(N,M,[A|B]) :- N > M,prolog((nl,tab(5),write(A))) | wr(2,M,B).
wr(_,_,[]) :- prolog(nl) | true.

%%%%%%%%%%%%%%%%%%%%%%%%%%%
%        Mesh maker         %
%%%%%%%%%%%%%%%%%%%%%%%%%%%
make_mesh(Max,[Status|Next],Maxcycle,Result) :- true |
        make_node((_,_,E,SE,S,SW,W,_),Status,Maxcycle,Contr1-C1),
        con(E,W),
        make_mesh(2,Max,(SE,S,SW),Next,C1-C2,Maxcycle),
        temp_node(C2,Contr),
        mg(Contr,[result([end|Result],Result)],Contr1).

make_mesh(K,Max,(NW,N,NE),[Status|Status_list],C1-C3,Maxcycle) :-
        K =< Max, K1 := K + 1 |
        con(E,W),con(N,N1),con(NW,NW1),con(NE,NE1),
        make_node((N1,NE1,E,SE,S,SW,W,NW1),Status,Maxcycle,C1-C2),
        make_mesh(K1,Max,(SE,S,SW),Status_list,C2-C3,Maxcycle).
make_mesh(K,Max,(NW,N,NE),Status,C1-C2,Maxcycle) :- K > Max |
        C2=[end_line(Status,(NW,N,NE))|C1].

%%%%%%%%%%%%%%%%%%%%%%%%%%%
%        Node maker         %
%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% node(Nr,NEr,Er,SEr,Sr,SWr,Wr,NWr,Contr,Status,Cycle,Maxcycle,Work)
%
%         Nr,...,NWr  : Receive_stream - Send_streame
%         Contr       : Controle stream   RECEIVE - SEND
%         Status      : the status of the node
%         Cycle       : Life cycle of the node
%         Maxcycle    : Max life cycle
%         Work        : D List for collecting datas

make_node((N,NE,E,SE,S,SW,W,NW),Status,Maxcycle,Contr) :- true |
        N  = Nr - Ns, NE = NEr - NEs, E = Er - Es,
        SE = SEr - SEs, S = Sr - Ss, SW = SWr - SWs,
        W = Wr   Ws, NW = NWr - NWs,
        node(Nr-Ns,NEr-NEs,Er-Es,SEr-SEs,Sr-Ss,SWr-SWs,Wr-Ws,NWr-NWs,Contr,
                Status,wait,Maxcycle,Work-Work).

%%%%%%%%%%%%%%%%%%%%%%%%%%%
%        Merge              %
%%%%%%%%%%%%%%%%%%%%%%%%%%%
mg([],X,Y) :- true | X = Y.
mg(X,[],Y) :- true | X = Y.
mg([X|Y],Z,L) :- true | L = [X|LL],mg(Y,Z,LL).
mg(Z,[X|Y],L) :- true | L = [X|LL],mg(Z,Y,LL).


%%%%%%%%%%%%%%%%%%%%%%%%%%%
%        Temp Node          %
%%%%%%%%%%%%%%%%%%%%%%%%%%%

temp_node([end_line(Status,(NW,N,NE))|Cn],Contr) :- true |
```

```
        Contr = [schizo(Status,(NW,N,NE)) | C1],
        temp_node(Cn,C1).
temp_node([schizo(Status,XNE,(YNW,YN,YC1-YCn)) | XCn],Contr) :- Status \= [] |
        Contr = [set_ne(XNE)],
        YC1 = [schizo1(Status,(YNW,YN)) | Cz],
        mg(C1,XCn,Cz),
        temp_node(YCn,C1).
temp_node([schizo([],XNE,(YNW,YN,YC1-YCn)) | XCn],Contr) :- true |
        Contr = [set_ne(XNE)],
        YC1 = [end(YNW,YN) | XCn],
        end_node(YCn,[]).
temp_node([result(A,B) |C],Contr) :- true |
        Contr = [result(A,B)|CC],
        temp_node(C,CC).

%%%%%%%%%%%%%%%%%%%%%%%%%
%       End Node        %
%%%%%%%%%%%%%%%%%%%%%%%%%

end_node([],S) :- true | S = [].
end_node([end|C],S) :- true | end_node(C,S).
end_node([result([R1|R2],Rt)|C],T) :- R1 \= end |
        T = Rt,
        end_node(C,[R1|R2]).
end_node([result([end|R],Rt)|C],T) :- true |
        T = Rt.

%%%%%%%%%%%%%%%%%%%%%%%%%
%       Connecter       %       It connects one hands to another.
%%%%%%%%%%%%%%%%%%%%%%%%%
con(X,Y) :- true | X = A-B,Y=B-A.

%%%%%%%%%%%%%%%%%%%%%%%%%
%       Node            %
%%%%%%%%%%%%%%%%%%%%%%%%%

%%%%%%%%%%%%%%%
%  schizo/2     .... for the first node of the first array

node(XN,XNE,XE,XSE,XS,XSW,XW,XNW,[schizo(Status,(NW,N,NE))|Contr]-Next,
                Self,wait,Maxcycle,Work) :- true |
        con(NW,XNW),con(N,XN),
        node(XN,_,XE1,XSE1,XS,XSW,XW,XNW,Contr-Next1,Self,wait,Maxcycle,Work),
        node_work1(NE,XE,XSE,XE1,XSE1,Status,Maxcycle,Next-Next1).

node_work1(NE,XE,XSE,XE1,XSE1,[Stat|Status],Maxcycle,Next-Next1) :- true |
        con(XE1,YW),con(NE,XNE),
        make_node((_,XNE,XE,XSE,YS,YSW,YW,_),Stat,Maxcycle,YC1-YC2),
        Next = [schizo(Status,YSW,(XSE1,YS,YC1-YC2)) | Next1].

%%%%%%%%%%%%%%%%
%   schizo/3    ..... for the general nodes

node(XN,XNE,XE,XSE,XS,XSW,XW,XNW,
                [schizo(Status,NEx,(NWy,Ny,YC1-YC2)) | Contr]-Next,
                Self,wait,Max,Work) :- true |
        con(XNE1,NEx),
        node(XN,XNE1,XE1,XSE1,XS,XSW,XW,XNW,Contr-Next1,Self,0,Max,Work),
        node_work2(XNE,XE,XSE,XE1,XSE1,Ny,NWy,YC1-YC2,Status,Max,Next-Next1).

node_work2(XNE,XE,XSE,XE1,XSE1,Ny,NWy,YC1-YC2,
                                        [Stat|Status],Max,Next-Next1) :- true |
        con(Ny,YN),con(XE1,YW),con(NWy,YNW),
        make_node((YN,XNE,XE,XSE,YS,YSW,YW,YNW),Stat,Max,YC2-YC3),
        Next = [schizo(Status,YSW,(XSE1,YS,YC1-YC3)) | Next1].
```

```
%%%%%%%%%%%%%%%%%%
%   set_ne        .... set the North-West hand (for the first line)

node(XN,XNE,XE,XSE,XS,XSW,XW,XNW,[set_ne(NEx)|Contr]-Next,
                Self,wait,Maxcycle,Work) :- true |
        con(NEx,XNE),
        node(XN,XNE,XE,XSE,XS,XSW,XW,XNW,Contr-Next,Self,0,Maxcycle,Work).

%%%%%%%%%%%%%%%%%%
%   schizo1/2     ... for the arrays except the first on

node(XN,XNE,XE,XSE,XS,XSW,XW,XNW,[schizo1(Status,(NW,N))|Contr]-Next,
                Self,wait,Maxcycle,Work) :- true |
        con(N,XN),con(NW,XNW),
        node(XN,_,XE1,XSE1,XS,XSW,XW,XNW,Contr-Next1,Self,wait,Maxcycle,Work),
        node_work3(XNE,XE,XSE,XE1,XSE1,Status,Maxcycle,Next-Next1).

node_work3(XNE,XE,XSE,XE1,XSE1,[Stat|Status],Maxcycle,Next-Next1) :- true |
        con(XE1,YW),
        make_node((_,XNE,XE,XSE,YS,YSW,YW,_),Stat,Maxcycle,YC1-YC2),
        Next = [schizo(Status,YSW,(XSE1,YS,YC1-YC2)) | Next1].

%%%%%%%%%%%%%%%%%%
%   end/2          for the first node of the last array

node(XN,XNE,XE,XSE,XS,XSW,XW,XNW,[end(NW,N)|Contr]-Next,
                Self,wait,Maxcycle,Work) :- true |
        con(N,XN),con(NW,XNW),
        node(XN,XNE,XE,XSE,XS,XSW,XW,XNW,Contr-Next1,Self,0,Maxcycle,Work),
        Next = [end|Next1].

%%%%%%%%%%%%%%%%%%
%   end           ... for the last array

node(XN,XNE,XE,XSE,XS,XSW,XW,XNW,[end|Contr]-Next,
                Self,wait,Maxcycle,Work) :- true |
        node(XN,XNE,XE,XSE,XS,XSW,XW,XNW,Contr-Next1,Self,0,Maxcycle,Work),
        Next = [end|Next1].

%%%%%%%%%%%%%%%%%%%%
%   going next life cycle

node([N|XN]-Ns,[NE|XNE]-NEs,[E|XE]-Es,[SE|XSE]-SEs,
                [S|XS]-Ss,[SW|XSW]-SWs,[W|XW]-Ws,[NW|XNW]-NWs,
        Contr,Self,Life,Max,Work) :- Life < Max, NewLife := Life + 1 |
        my_life([N,NE,E,SE,S,SW,W,NW],Self,NewSelf,0),
        node_work4(XN-Ns,XNE-NEs,XE-Es,XSE-SEs,XS-Ss,XSW-SWs,XW-Ws,XNW-NWs,
                        Contr,NewSelf,NewLife,Max,Work).

node_work4(XN-Ns,XNE-NEs,XE-Es,XSE-SEs,XS-Ss,XSW-SWs,XW-Ws,XNW-NWs,
                Contr,NewSelf,NewLife,Max,Work) :- true |
        send([Ns-XNs,NEs-XNEs,Es-XEs,SEs-XSEs,
                Ss-XSs,SWs-XSWs,Ws-XWs,NWs-XNWs],NewSelf),
        node(XN-XNs,XNE-XNEs,XE-XEs,XSE-XSEs,XS-XSs,XSW-XSWs,XW-XWs,XNW-XNWs,
                        Contr,NewSelf,NewLife,Max,Work).

%%%%%%%%%%%%%%%%%%%%%%
%   dead

node(_,_,_,_,_,_,_,_,Contr-Next,Self,Life,Max,Work-Wt) :- Life >= Max |
        Wt = [Self|St],
        Next = [result(Work,St)|Contr].

%%%%%%%%%%%%%%%%%%%%%%
%   collect datas
```

```
node(A,B,C,D,E,F,G,H,[result(S,St)|Contr]-Next,Self,wait,Max,Work-Wt) :- true |
        St = Work,
        node(A,B,C,D,E,F,G,H,Contr-Next,Self,wait,Max,S-Wt).
node(A,B,C,D,E,F,G,H,[result(S,St)|Contr]-Next,Self,Life,Max,Work-Wt) :-
        Life < Max |
        St = Work,
        node(A,B,C,D,E,F,G,H,Contr-Next,Self,Life,Max,S-Wt).

%%%%%%%%%%%%%%%%%%%%%%%%%
%    for the first time  ... when the node's life cycle is 0

node(N-Ns,NE-NEs,E-Es,SE-SEs,S-Ss,SW-SWs,W-Ws,NW-NWs,
                                Cont-Next,Self,0,Max,Work) :- true |
        send([Ns-Nsn,NEs-NEsn,Es-Esn,SEs-SEsn,
              Ss-Ssn,SWs-SWsn,Ws-Wsn,NWs-NWsn],Self),
        node(N-Nsn,NE-NEsn,E-Esn,SE-SEsn,S-Ssn,SW-SWsn,W-Wsn,NW-NWsn,
              Cont-Next,Self,1,Max,Work).

%%%%%%%%%%%%%%%%%%%%%%%%%
%       Send            %       send my status to the neighbours
%%%%%%%%%%%%%%%%%%%%%%%%%
send([],_) :- true | true.
send([A-B|C],S) :- true | A = [S|B],send(C,S).


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%       Rule of Life Game       %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%       mys_life(Neighbours_list,
%                Me,
%                NewStatus,
%                Number_of_Black_neighbours)

my_life([],1,Return,2) :- true | Return = 1.
my_life([],1,Return,3) :- true | Return = 1.
my_life([],0,Return,3) :- true | Return = 1.
my_life([],1,Return,N) :- N < 2 | Return = 0.
my_life([],1,Return,N) :- N > 3 | Return = 0.
my_life([],0,Return,N) :- N < 3 | Return = 0.
my_life([],0,Return,N) :- N > 3 | Return = 0.
my_life([A|B],Self,Return,Sum) :- Sum1 := Sum + A |
        my_life(B,Self,Return,Sum1).


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%               Test Program for This game                      %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
%       The patter    *     moves by 4 cyles.  (as if it is a Glider !)
%                      *
%                     ***
%

life_test(N) :- true |
        life_game(8,   [0,0,0,0,0,0,0,0,
                        0,0,0,0,0,0,0,0,
                        0,0,0,0,0,0,0,0,
                        0,0,0,0,1,0,0,0,
                        0,0,0,0,0,1,0,0,
                        0,0,0,1,1,1,0,0,
                        0,0,0,0,0,0,0,0,
                        0,0,0,0,0,0,0,0],
                N).
```

(0) Date: 1987-Jul-18, written by E. Sugino
Modified: 28-Jul-87 by S. Takagi
Modified: 12-Aug-87 by E. Sugino

(1) Program name: life

(2) Author: E. Sugino, ICOT 4th lab.

(3) Runs on: GHC1 on DEC2065
Pseudo Multi PSI (if you change the program a little)

(4) Description of the problem:

The life game is a popular game.

(5) Algorithm: See US2:<MPSI.BENCH>LIFE_SUGINO_580.DOC

(6) Process structure: See US2:<MPSI.BENCH>LIFE_SUGINO_580.DOC

(7) Pragma: None

(8) Program:

It is almost the same as the program in
US2:<MPSI.BENCH>LIFE_SUGINO_580.GHC.
The following points differ.

(a) It displays the status of the mesh with Prolog functions
(or with KL0 functions through esp_call on PSI).
Each node has its own coordinates on the mesh so that
it can be displayed on the terminal.
(b) The parts of node which are used to make the mesh,
are renamed node1.
(c) When you use life cycle 0, the original pattern is displayed.
(You must use life cycle 1 in LIFE_SUGINO_580.)

(9) Source file: US2:<MPSI.BENCH>LIFE_SUGINO_580NEW.GHC
This document: US2:<MPSI.BENCH>LIFE_SUGINO_580NEW.DOC
Another sample pattern: US2:<MPSI.BENCH>LIFE_SUGINO_OPTION.GHC

(10) Example:

Compile the program, and try it.

```
? ghc life_test(8).
```

8 is the number of the life cycle.
You can substitute any positive integer fo 8.

(11) Evaluation data: Not recorded

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%       LIFE GAME  Program      1987.4.16. by E.Sugino                        %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

/*
module life.

 :- public life_game/3 ,
           life_test/1 ,life_test1/1 .
*/


life_game(X,List,Max) :-  true |
     headder(R,Max),
     make_mesh(R,(X,List,Max,Result)).

headder(R,Max) :- true | erase(R1),headder1(R1,R,Max).
headder1(end,R,Max) :- true | home(R2),headder2(R2,R,Max).
headder2(end,R,Max) :-
     prolog(
          (
          nl,
          write('Life Game Now Starts ! '),                    %
          write(Max),                                          % Comment
          (Max > 1,write(' cycles') ; write(' cycle'))         %   on PSI
          ,nl)) |                                              %
/*
     write('Life Game Now Starts ! ') |
*/
     R = end.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%          Tool                  %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
home(R) :- prolog(write('$[H')) | R = end.                     % com. on PSI
/*
home(R) :- esp_call(move,home,ok) | R = end.
*/

erase(R) :- prolog(write('$[2J')) | R = end.                   % com. on PSI
/*
erase(R) :- esp_call(erase,a,ok) | R = end.
*/

show(X,Y,S,S,End) :- true | End = end.
show(X,Y,S,S1,End) :- S =\= S1 | show(X,Y,S,End).

show(X,Y,1,E) :- wait(X),wait(Y),                                  %
     prolog((nl,                                                   %
          write('$['),write(Y),write(';'),write(X),write('H'),     %
          put("*"),ttyflush)) |                                    % Comment
     E = end.                                                      %   on PSI
show(X,Y,0,E) :- wait(X),wait(Y),                                  %
     prolog((nl,                                                   %
          write('$['),write(Y),write(';'),write(X),write('H'),     %
          put("-"),ttyflush)) |                                    %
     E = end.                                                      %
/*
show(X,Y,1,E) :- wait(X),wait(Y), esp_call(move,(X,Y),ok),
                 esp_call(write_term,('*'),ok) |
     E = end.
show(X,Y,0,E) :- wait(X),wait(Y), esp_call(move,(X,Y),ok),
                 esp_call(write_term,('-'),ok) |
     E = end.
*/
```

```
start_node_ad(X) :- true |
            X = (5,5).
%             X = (2,2).
node_ad(K,YY) :-
            Y := 5 + (K - 1) * 3 | YY =(5,Y).
%             Y := 2 + (K - 1) * 2 | YY =(2,Y).

%%%%%%%%%%%%%%%%%%%%%%%%%%
%     Mesh maker      %
%%%%%%%%%%%%%%%%%%%%%%%%%%

make_mesh(end,(X,List,Max,Result)) :- true |
     make_mesh(X,List,Max,Result).
make_mesh(Max,[Status|Next],Maxcycle,Result) :- true |
     start_node_ad(XY),
     make_node1(SE_S_SW,Status,Maxcycle,Contr1-C1,XY),
     make_mesh(2,Max,SE_S_SW,Next,C1-C2,Maxcycle),
     temp_node(C2,Contr),
     mg(Contr,[result([end|Result],Result)],Contr1).

make_mesh(K,Max,(NW,N,NE),[Status|Status_list],C1-C3,Maxcycle) :-
     K =< Max, K1 := K + 1 |
     node_ad(K,YY),
     make_node2((N,NE,NW),SE_S_SW,Status,Maxcycle,C1-C2,YY),
     make_mesh(K1,Max,SE_S_SW,Status_list,C2-C3,Maxcycle).
make_mesh(K,Max,(NW,N,NE),Status,C1-C2,Maxcycle) :- K > Max |
     C2=[end_line(Status,(NW,N,NE))|C1].

%%%%%%%%%%%%%%%%%%%%%%%%%%
%     Node maker      %
%%%%%%%%%%%%%%%%%%%%%%%%%%

make_node1(SE_S_SW,Status,Maxcycle,Contr,XY) :- true |
     SE_S_SW = (SEr - SEs, Sr - Ss, SWr - SWs),
     node1(Contr,_,_,_,Er-Es,SEr-SEs,Sr-Ss,SWr-SWs,Es-Er,_,_,
           Status,Maxcycle,Work-Work,XY).

make_node2((Ns-Nr,NEs-NEr,NWs-NWr),SE_S_SW,Status,Maxcycle,Contr,XY) :- true |
     SE_S_SW = (SEr - SEs, Sr - Ss, SWr - SWs),
     node1(Contr,Nr-Ns,NEr-NEs,Er-Es,SEr-SEs,Sr-Ss,SWr-SWs,Es-Er,NWr-NWs,
           Status,Maxcycle,Work-Work,XY).

make_node3((NEs-NEr,Er-Es,SEr-SEs,Ws-Wr),(S,SW),Status,Maxcycle,Contr,XY)
    :- true |
     S = Sr - Ss, SW = SWr - SWs,
     node1(Contr,_,_,NEr-NEs,Er-Es,SEr-SEs,Sr-Ss,SWr-SWs,Wr-Ws,_,_,
           Status,Maxcycle,Work-Work,XY).

make_node4((Ns-Nr,NEr-NEs,Er-Es,SEr-SEs,Ws-Wr,NWs-NWr),(S,SW),
                                        Status,Maxcycle,Contr,XY) :- true |
     S = Sr - Ss, SW = SWr - SWs,
     node1(Contr,Nr-Ns,NEr-NEs,Er-Es,SEr-SEs,Sr-Ss,SWr-SWs,Wr-Ws,NWr-NWs,
       Status,Maxcycle,Work-Work,XY).

make_node5((NEr-NEs,Er-Es,SEr-SEs,Ws-Wr),(S,SW),Status,Maxcycle,Contr,XY)
   :- true |
     S = Sr - Ss, SW = SWr - SWs,
     node1(Contr,Nr-Ns,NEr-NEs,Er-Es,SEr-SEs,Sr-Ss,SWr-SWs,Wr-Ws,NWr-NWs,
           Status,Maxcycle,Work-Work,XY).

%%%%%%%%%%%%%%%%%%%%%%%%%%
%        Merge          %
%%%%%%%%%%%%%%%%%%%%%%%%%%
mg([],X,Y) :- true | X = Y.
mg(X,[],Y) :- true | X = Y.
mg([X|Y],Z,L) :- true | L = [X|LL],mg(Y,Z,LL).
```

```
mg(Z,[X|Y],L) :- true | L = [X|LL],mg(Z,Y,LL).


%%%%%%%%%%%%%%%%%%%%%%%%%%
%      Temp Node          %
%%%%%%%%%%%%%%%%%%%%%%%%%%

temp_node([end_line(Status,(NW,N,NE))|Cn],Contr) :- true |
     Contr = [schizo(Status,(NW,N,NE)) | C1],
     temp_node(Cn,C1).
temp_node([schizo(Status,XNE,(YNW,YN,YC1-YCn)) | XCn],Contr) :- Status \= [] |
     Contr = [set_ne(XNE)],
     YC1 = [schizo1(Status,(YNW,YN)) | Cz],
     mg(C1,XCn,Cz),
     temp_node(YCn,C1).
temp_node([schizo([],XNE,(YNW,YN,YC1-YCn)) | XCn],Contr) :- true |
     Contr = [set_ne(XNE)],
     YC1 = [end(YNW,YN) | XCn],
     end_node(YCn,[]).
temp_node([result(A,B) |C],Contr) :- true |
     Contr = [result(A,B)|CC],
     temp_node(C,CC).

%%%%%%%%%%%%%%%%%%%%%%%%%%
%      End Node           %
%%%%%%%%%%%%%%%%%%%%%%%%%%

end_node([],S) :- true | S = [].
end_node([end|C],S) :- true | end_node(C,S).
end_node([result([R1|R2],Rt)|C],T) :- R1 \= end |
     T = Rt,
     end_node(C,[R1|R2]).
end_node([result([end|R],Rt)|C],T) :- true | home(End),end_node1(End,T,Rt).
end_node1(end,T,Rt) :- true | T=Rt.

%%%%%%%%%%%%%%%%%%%%%%%%%%
%      Connecter          %      It connects one hands to another.
%%%%%%%%%%%%%%%%%%%%%%%%%%
% con(X,Y) :- true | X = A-B,Y=B-A.
con(A-B,C-D) :- true | A=D,B=C.

%%%%%%%%%%%%%%%%%%%%%%%%%%
%      Node               %
%%%%%%%%%%%%%%%%%%%%%%%%%%

%%%%%%%%%%%%%%%
%  schizo/2      .... for the first node of the first array

node1([schizo(Status,(NW,N,NE))|Contr]-Next,XN,XNE,XE,XSE,XS,XSW,XW,XNW,
          Self,Maxcycle,Work,XY) :- true |
     con(NW,XNW),con(N,XN),
     node1(Contr-Next1,XN,_-_,C-D,A-B,XS,XSW,XW,XNW,Self,Maxcycle,Work,XY),
     node_work1(NE,XE,XSE,C-D,A-B,Status,Maxcycle,Next-Next1,XY).

%%%%%%%%%%%%%%%
%   schizo/3      ..... for the general nodes

node1([schizo(Status,NEx,(NWy,Ny,YC1-YC2)) | Contr]-Next,
        XN,XNE,XE,XSE,XS,XSW,XW,XNW,
          Self,Max,Work,XY) :- true |
     con(NEx,C-D),
     node(XN,C-D,E-F,A-B,XS,XSW,XW,XNW,Contr-Next1,Self,0,Max,Work,XY),
     node_work2(XNE,XE,XSE,E-F,A-B,Ny,NWy,YC1-YC2,Status,Max,Next-Next1,XY).

%%%%%%%%%%%%%%%%
%   set_ne        .... set the North-West hand (for the first line)
```

```
node1([set_ne(NEx)|Contr]-Next,XN,XNE,XE,XSE,XS,XSW,XW,XNW,
          Self,Maxcycle,Work,XY) :- true |
     con(NEx,XNE),
     node(XN,XNE,XE,XSE,XS,XSW,XW,XNW,Contr-Next,Self,0,Maxcycle,Work,XY).

%%%%%%%%%%%%%%%%%
%   schizo1/2     ... for the arrays except the first one

node1([schizo1(Status,(NW,N))|Contr]-Next,XN,XNE,XE,XSE,XS,XSW,XW,XNW,
          Self,Maxcycle,Work,XY) :- true |
     con(N,XN),con(NW,XNW),
     node1(Contr-Next1,XN,_-_,C-D,A-B,XS,XSW,XW,XNW,Self,Maxcycle,Work,XY),
     node_work3(XNE,XE,XSE,C-D,A-B,Status,Maxcycle,Next-Next1,XY).

%%%%%%%%%%%%%%%%%
%   end/2       for the first node of the last array

node1([end(NW,N)|Contr]-Next,XN,XNE,XE,XSE,XS,XSW,XW,XNW,
          Self,Maxcycle,Work,XY) :- true |
     con(N,XN),con(NW,XNW),
     node(XN,XNE,XE,XSE,XS,XSW,XW,XNW,Contr-Next1,Self,0,Maxcycle,Work,XY),
     Next = [end|Next1].

%%%%%%%%%%%%%%%%%
%   end           ... for the last array

node1([end|Contr]-Next,XN,XNE,XE,XSE,XS,XSW,XW,XNW,
          Self,Maxcycle,Work,XY) :- true |
     node(XN,XNE,XE,XSE,XS,XSW,XW,XNW,Contr-Next1,Self,0,Maxcycle,Work,XY),
     Next = [end|Next1].

%%%%%%%%%%%%%%%%%%%%%%%%
%   collect datas

node1([result(S,St)|Contr]-Next,A,B,C,D,E,F,G,H,Self,Max,Work-Wt,XY) :-
     true |
     St = Work,
     node1(Contr-Next,A,B,C,D,E,F,G,H,Self,Max,S-Wt,XY).

node_work1(NE,XE,XSE,XE1,XSE1,[Stat|Status],Maxcycle,Next-Next1,XY) :-
     true |
     neighb_node_ad(XY,NXY),
     make_node3((NE,XE,XSE,XE1),(YS,YSW),Stat,Maxcycle,YC1-YC2,NXY),
     Next = [schizo(Status,YSW,(XSE1,YS,YC1-YC2)) | Next1].

neighb_node_ad((X,Y),XY) :- X1 := X + 6 | XY = (X1,Y).


node_work2(NE,E,SE,XE,XSE,N,NW,YC1-YC2,
                          [Stat|Status],Max,Next-Next1,(X,Y)) :-
     X1 := X + 6 |
     make_node4((N,NE,E,SE,XE,NW),(YS,YSW),Stat,Max,YC2-YC3,(X1,Y)),
     Next = [schizo(Status,YSW,(XSE,YS,YC1-YC3)) | Next1].


node_work3(NE,E,SE,W,XSE1,[Stat|Status],Maxcycle,Next-Next1,(X,Y)) :-
     X1 := X + 6 |
     make_node5((NE,E,SE,W),(YS,YSW),Stat,Maxcycle,YC1-YC2,(X1,Y)),
     Next = [schizo(Status,YSW,(XSE1,YS,YC1-YC2)) | Next1].

%%%%%%%%%%%%%%%%%%%%%%%%
%   for the first time  ... when the node's life cycle is 0

node(N,NE,E,SE,S,SW,W,NW,Cont,Self,0,Max,Work,XY) :-
     XY = (X,Y) |
```

```
      show(X,Y,Self,End),
      send_first(End,[N,NE,E,SE,S,SW,W,NW],
                     [Nr,NEr,Er,SEr,Sr,SWr,Wr,NWr],Send,Self),
      node(Nr,NEr,Er,SEr,Sr,SWr,Wr,NWr,Send,Cont,Self,1,Max,Work,XY).

%%%%%%%%%%%%%%%%%%
%    going next life cycle

node([N|XN],[NE|XNE],[E|XE],[SE|XSE],[S|XS],[SW|XSW],[W|XW],[NW|XNW],Send,
                                          Contr,Self,Life,Max,Work,XY) :-
      0 < Life,Life =< Max,
      NewLife := Life + 1 |
      my_life([N,NE,E,SE,S,SW,W,NW],Self,NewSelf,0),
      node_work4(XN,XNE,XE,XSE,XS,XSW,XW,XNW,Send,
                                Contr,NewSelf,NewLife,Max,Work,XY,Self).

%%%%%%%%%%%%%%%%%%%%%%%%
%    dead

node(_,_,_,_,_,_,_,_,Contr-Next,Self,Life,Max,Work-Wt,(X,Y)) :-
      Life > Max |
      Wt = [Self|St],
      Next = [result(Work,St)|Contr].

node(A,B,C,D,E,F,G,H,Sd,[result(S,St)|Contr]-Next,Self,Life,Max,Work-Wt,XY) :-
      0 < Life,Life =< Max |
      St = Work,
      node(A,B,C,D,E,F,G,H,Sd,Contr-Next,Self,Life,Max,S-Wt,XY).

node_work4(XN,XNE,XE,XSE,XS,XSW,XW,XNW,Send,
           Contr,NewSelf,NewLife,Max,Work,(X,Y),Self) :-
      true |
      show(X,Y,NewSelf,Self,End),
      send(End,Send,NewSend,NewSelf),
      node(XN,XNE,XE,XSE,XS,XSW,XW,XNW,NewSend,
                                Contr,NewSelf,NewLife,Max,Work,(X,Y)).

%%%%%%%%%%%%%%%%%%%%%%%%
%     Send          %     send my status to the neighbours
%%%%%%%%%%%%%%%%%%%%%%%%

send(end,Send,NewSend,Self) :- true | Send = [Self|NewSend].

send_first(end,In,Out,Send,Self) :- true | send_all(In,Out,Send,Self).
send_all([],[],_,_) :- true | true.
send_all([A-B|C],[AA|CC],Send,Self) :- true |
        B = [Self|Send],AA=A,
        send_all(C,CC,Send,Self).


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%     Rule of Life Game     %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%     my_life(Neighbours_list,
%           Me,
%           NewStatus,
%           Number_of_Black_neighbours)

my_life([],1,Return,2) :- true | Return = 1.
my_life([],1,Return,3) :- true | Return = 1.
my_life([],0,Return,3) :- true | Return = 1.
my_life([],1,Return,N) :- N < 2 | Return = 0.
my_life([],1,Return,N) :- N > 3 | Return = 0.
my_life([],0,Return,N) :- N < 3 | Return = 0.
my_life([],0,Return,N) :- N > 3 | Return = 0.
my_life([A|B],Self,Return,Sum) :- Sum1 := Sum + A |
```

```
     my_life(B,Self,Return,Sum1).


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%          Test Program for This game          %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
%     The patter    *          moves by 4 cyles.  (as if it is a Glider !)
%                    *
%                  ***
%

 life_test(N) :- true |
      life_game(8,    [0,0,0,0,0,0,0,0,
                0,0,0,0,0,0,0,0,
                0,0,0,0,0,0,0,0,
                0,0,0,0,1,0,0,0,
                0,0,0,0,0,1,0,0,
                0,0,0,1,1,1,0,0,
                0,0,0,0,0,0,0,0,
                0,0,0,0,0,0,0,0],
           N).

 life_test1(N) :- true |
        life_game(6,    [0,0,0,0,0,0,
                     0,0,1,0,0,0,
                     0,1,1,0,0,0,
                     0,0,0,1,0,0,
                     0,0,0,0,0,0,
                     0,0,0,0,0,0],
              N).


/*
end.
*/
```

(0) Date: 1987-July-30, written by Kazuaki Rokusawa
Modified: 3-Aug-87 by S. Takagi

(1) Program name: maxflow1-ex1

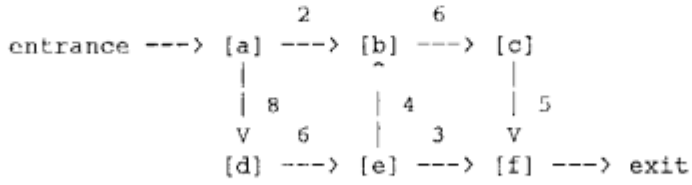
(2) Author: Kazuaki Rokusawa

(3) Runs on: DEC-10 Prolog GHC system (GHC and GHC3)

(4) Description of the problem:

Finding the maximum amount of flow through the network.
The network is composed of nodes and one-directional arcs.
Each node has up to two arcs each for output and input.
The flow through each arc is limited.

Example:



node = [a], [b], [c], [d], [e], [f]

Node [b] has 1 arc for output and 2 arcs for input.
Node [e] has 2 arcs for output and 1 arc for input.

Node [b] can send flow 6 to node [c].
Node [e] can send flow 4 to node [b] and 3 to node [f].

The maximum flow of this network is 8.

(5) Algorithm:

Each node works as follows.

On receiving input flow, the node tries to send the same amount of flow.

When the input flow is beyond the limit of the output flow, the node sends as much as possible and returns the remainder to the sender node.

When the input flow is less than the limit of the output flow, the node sends all the input. If that node has two output arcs, there is some preference to decide the amount for each output.

When the sum of the flow to the exit and the flow back to the entrance comes to the same amount as the original flow to the entrance, each node stops processing.

Sending flow 10 to node [a] on the example of (4), the following occur.

Node [a] sends flow 2 to [b] and 8 to [d].
On receiving input flow 2 from [a], node [b] sends 2 to [c].
This flow 2 reaches exit through nodes [c] and [f].

On receiving input flow 8, node [d] sends 6 to [e] and returns flow 2 to [a].
Returned flow 2 is returned to the entrance.

Node [e] receives 6 and sends 4 to [b] and 2 to [f]
(when the arc to [b] has a higher priority than the one to [f]).
Flow 2 reaches the exit through node [f].

Flow 4 reaches node [c] through node [b].
Since node [c] has already been sent 2,
it sends 3 to [f] and returns 1 to [b].
Flow 3 reaches the exit through node [f].

On receiving this backward input flow 1, node [b] returns
again to node [e].
Since node [e] has sent only 2 to node [f],
it receives backward input flow 1 and sends it to [f].
Total flow 8 reaches exit in all.

The sum of the flow which arrived at exit and returned to entrance
becomes equal to the original input.
Each node stops processing.

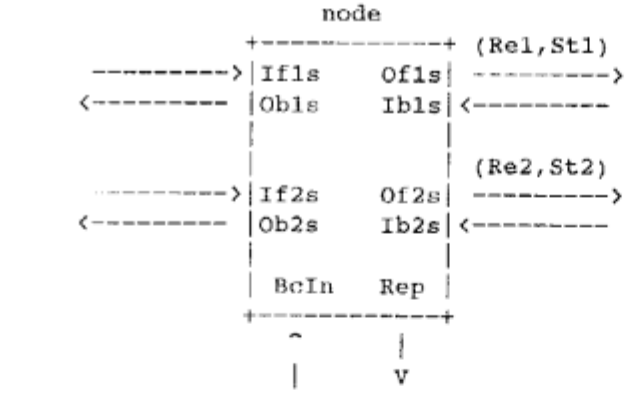
(6) Process structure:

The main processes are:

node: The node process which sends or receives a flow.
This process creates the same number of nodes.
The figure below shows this process.

exit: Detecting the end

outFlow:Display process



If1s and If2s, Of1s and Of2s, Ib1s and Ib2s, Ob1s and Ob2s are all
forward or backward input or output streams. "f" and "b" mean forward
and backward. "I" and "O" mean input and output.

Re1 and Re2 indicate the amount of flow that the node can send through
forward output streams Of1s and Of2s.

St1 and St2 have the value open or closed. open indicates that the
output stream still has space and the node can send more flow through
this stream. closed indicates that the output stream is already full
and the node cannot send through this stream any more.

The flow forms as follows.

```
 [amount(Amount-of-flow),link(1 or 2,Node-name),
                         link( , ),...,link(0,entrance)]
```

1 means that the node received the flow through If1s, while 2 means through If2s.

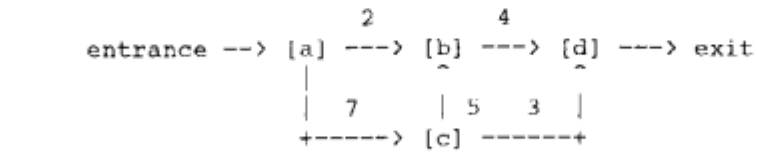
In example (4), flow 2, which reaches the exit through nodes [a], [b], [c], and [f], forms as follows.

```
 [amount(2),link(2,f),link(1,c),link(2,b),link(1,a)].
```

(7) Pragma: None

(8) Program:

The network is as follows.



The main predicates used in this program are:

ex1/1:	Top level predicate to start this program
entrance/2:	Sends the original flow to the entrance
exit/6:	Detects the end
node/14:	The main process which sends or receives a flow. This process works as: Waits for the input-flow Changes the status Sends the flow
outFlow/4:	Displays the result

(9) Source file: US2:<MPSI.BENCH>MAXFLOW1-EX1.GHC
Log file: US2:<MPSI.BENCH>MAXFLOW1-EX1.LOG (on GHC system)
US2:<MPSI.BENCH>MAXFLOW1-EX1-GHC3.LOG (on GHC3 system)
This document: US2:<MPSI.BENCH>MAXFLOW1-EX1.DOC

(10) Examples:

Copy the source file (<MPSI.BENCH>MAXFLOW1-EX1.GHC) to a temporary file whose file name is less than 7 characters and whose file extension is less than 4 (i.e. up to 6 characters for the file name and 3 characters for the extension).

```
        @copy US2:<MPSI.BENCH>MAXFLOW1-EX1.GHC -t.ghc<CR>
```

Invoke the GHC system on the DEC2065.

```
        @ghc<CR>.
```

Compile the program.

```
        ?- ghccompile('-t.ghc').<CR>
```

Execute the program.

```
        ?- ghc ex1(10).<CR>     % The first argument of ex1/1 is the amount
                                % of the original flow to the entrance.
```

(11) Evaluation data:

The results of the execution on the GHC3 system are:

```
| ?- ghc ex1(10).
pass([amount(2),link(2,d),link(2,b),link(1,a),link(0,entrance)])
        % This output shows the flow to exit.
return([amount(1),link(0,entrance)])
        % This output shows the flow back to entrance.
pass([amount(2),link(1,d),link(1,c),link(1,a),link(0,entrance)])
pass([amount(2),link(2,d),link(1,b),link(1,c),link(1,a),link(0,entrance)])
pass([amount(1),link(1,d),link(1,c),link(1,a),link(0,entrance)])
return([amount(2),link(0,entrance)])
remain(node(a,(0,closed),(2,closed)))
        % This output is a report of the node in which Re1>0 or Re2>0
        % after execution.
remain(node(c,(3,closed),(0,closed)))

maxFlow(7)
        % This output shows the maximum flow.

61 reductions and 15 suspensions in 18 cycles and 456 msec (133 rps)
The maximum number of reducible goals is 7/9 at the 9th cycle.
The maximum length of the queue is 9.

yes
| ?- ghc ex1(7).
pass([amount(2),link(2,d),link(2,b),link(1,a),link(0,entrance)])
pass([amount(2),link(2,d),link(1,b),link(1,c),link(1,a),link(0,entrance)])
pass([amount(3),link(1,d),link(1,c),link(1,a),link(0,entrance)])
remain(node(a,(0,closed),(2,open)))
remain(node(c,(3,closed),(0,closed)))

maxFlow(7)

48 reductions and 17 suspensions in 17 cycles and 346 msec (138 rps)
The maximum number of reducible goals is 6/6 at the 11th cycle.
The maximum length of the queue is 8.
```

(12) References

L. Hellerstein and E. Shapiro,
Implementing Parallel Algorithms in Concurrent Prolog:
    The MAXFLOW Experience,
Journal of Logic Programming, volume 3, number 2, July 1986, pp.157-184

```
%
%        MAXFLOW
%       ---------
%
% example. 1      ( 4 nodes )
%
%                          2           4           7
%     entrance --> [ a ] ---> [ b ] ---> [ d ] ---> exit
%                    |           A           A
%                    |   7       | 5     3   |
%                    +-------> [ c ] -------+
%
% ex1( LimFlow ) :- true |
%       entrance( LimFlow, Fsa ),
%       node( a, (2,open), (7,open),   Fsa, [],  Fab, Fac, Bab, Bac, Bsa, _,
%                                                BcIn, RepHead, RepNext1 ),
%       node( b, (4,open), (0,closed), Fcb, Fab, Fbd, _,   Bbd, [],  Bcb, Bab,
%                                                BcIn, RepNext1, RepNext2 ),
%       node( c, (5,open), (3,open),   Fac, [],  Fcb, Fcd, Bcb, Bcd, Bac, _,
%                                                BcIn, RepNext2, RepNext3 ),
%       node( d, (7,open), (0,closed), Fcd, Fbd, Fdg, _,   [], [],  Bcd, Bbd,
%                                                BcIn, RepNext3, [] ),
%       exit( LimFlow, 0, Fdg, Bsa, BcIn, Prs ),
%       outFlow( Prs, RepHead, 0, Os ),
%       outstream( Os ).
%
ex1( LimFlow ) :- true |
       entrance( LimFlow, Fsa ),
       nodes( Fsa, Fdg, Bsa, BcIn, RepHead ),
       exit( LimFlow, 0, Fdg, Bsa, BcIn, Prs ),
       outFlow( Prs, RepHead, 0, Os ),
       outstream( Os ).
nodes( Fsa, Fdg, Bsa, BcIn, RepHead ) :- true |
       node( a, (2,open), (7,open),   Fsa, [],  Fab, Fac, Bab, Bac, Bsa, _,
                                                BcIn, RepHead, RepNext1 ),
       node( b, (4,open), (0,closed), Fcb, Fab, Fbd, _,   Bbd, [],  Bcb, Bab,
                                                BcIn, RepNext1, RepNext2 ),
       node( c, (5,open), (3,open),   Fac, [],  Fcb, Fcd, Bcb, Bcd, Bac, _,
                                                BcIn, RepNext2, RepNext3 ),
       node( d, (7,open), (0,closed), Fcd, Fbd, Fdg, _,   [], [],  Bcd, Bbd,
                                                BcIn, RepNext3, [] ).
%
%        MAXFLOW         - main part -
%       ---------
%
%                  +---  ---  ------------------+
%                  |                           |
%                  |                           V
%   entrance ---> node <--> ... <--> node <--> exit --> outFlow --> outstream
%                  A                  A          |          A
%                  |                  |          V          |
%                  V                  V      all nodes  all nodes
%                 node <--> ... <--> node
%                  A                  A
%                  |                  |
%                  :                  :
%
%                     stream diagram
%                    ----------------
%
%                        node
%                      +-------------+ (Rc1,St1)
%             --------->|If1s    Of1s| --------->
%            <--------- |Ob1s    Ib1s|<---------
%                       |            |
%                       |            | (Re2,St2)
```

```
%              --------->|If2s    Of2s| --------->
%             <---------  |Ob2s    Ib2s|<---------
%                         |            |
%                         | BcIn   Rep |
%                         +------------+
%                            A       |
%                            |       V
%
%               Ifs   : input  forward  stream
%               Ofs   : output forward  stream
%               Ibs   : input  backward stream
%               Obs   : output backward stream
%               Re    : remainder
%               St    : status ( open / closed )
%               BcIn  : broadcast in
%               Reph  : reply head
%               Rept  : reply tail
%
%                    node specification
%                   --------------------
%
%       entrance
%      ----------
entrance( LimFlow, Ofs ) :- true |
       Ofs = [[amount(LimFlow),link(0,entrance)]].
%
%       exit
%      ------
% continue       ( LimFlow > NextSum )
exit( LimFlow, Sum, Ifs, Irs, BcOut, Prs ) :-
              Ifs = [If|RestIfs], If = [amount(Flow)|_],
              NextSum := Sum+Flow, LimFlow > NextSum |
       Prs = [pass(If)|NextPrs],
       exit( LimFlow, NextSum, RestIfs, Irs, BcOut, NextPrs ).
exit( LimFlow, Sum, Ifs, Irs, BcOut, Prs ) :-
              Irs = [Ir|RestIrs], Ir = [amount(Flow)|_],
              NextSum := Sum+Flow, LimFlow > NextSum |
       Prs = [return(Ir)|NextPrs],
       exit( LimFlow, NextSum, Ifs, RestIrs, BcOut, NextPrs ).
%
% terminate       ( LimFlow = NextSum )
exit( LimFlow, Sum, Ifs, Irs, BcOut, Prs ) :-
              Ifs = [If|RestIfs], If = [amount(Flow)|_],
              NextSum := Sum+Flow, LimFlow = NextSum |
       BcOut = end, Prs = [pass(If)].
exit( LimFlow, Sum, Ifs, Irs, BcOut, Prs ) :-
              Irs = [Ir|RestIrs], Ir = [amount(Flow)|_],
              NextSum := Sum+Flow, LimFlow = NextSum |
       BcOut = end, Prs = [return(Ir)].
%
%       node
%      ------
% 1 --> 1
%       ( Flow < Rel )
node( Node, (Rel,St1), (Re2,St2), If1s,If2s,Of1s,Of2s,
                            Ib1s,Ib2s,Ob1s,Ob2s, BcIn, Reph, Rept ) :-
              If1s = [If|RestIf1s], If = [amount(Flow)|RestIf],
              St1 = open, Flow < Rel |
       NextRel := Rel-Flow,
       Of1s = [ [amount(Flow),link(1,Node)|RestIf] | NextOf1s ],
       node( Node, (NextRel,St1), (Re2,St2),
                     RestIf1s, If2s, NextOf1s, Of2s,
                     Ib1s,     Ib2s, Ob1s,     Ob2s, BcIn, Reph, Rept ).
%       ( Flow = Rel )
node( Node, (Rel,St1), (Re2,St2), If1s,If2s,Of1s,Of2s,
                            Ib1s,Ib2s,Ob1s,Ob2s, BcIn, Reph, Rept ) :-
```

```
                If1s = [If|Rest1f1s], If = [amount(Flow)|RestIf],
                St1 = open, Flow = Rel |
        Of1s = [ [amount(Flow),link(1,Node)|RestIf] ],
        node( Node, (0,closed), (Re2,St2),
                        RestIf1s, If2s, _,    Of2s,
                        Ib1s,     Ib2s, Ob1s, Ob2s, BcIn, Reph, Rept ).
% 1 ---> 1 & call again
%       ( Flow > Rel )
node( Node, (Rel,St1), (Re2,St2), If1s,If2s,Of1s,Of2s,
                                Ib1s,Ib2s,Ob1s,Ob2s, BcIn, Reph, Rept ) :-
                If1s = [If|Rest1f1s], If = [amount(Flow)|RestIf],
                St1 = open, Flow > Rel |
        RestFlow :=  Flow-Rel,
        Of1s = [ [amount(Rel),link(1,Node)|RestIf] | NextOf1s ],
        AgainIf1s = [ [amount(RestFlow)|RestIf] | RestIf1s ],
        node( Node, (0,closed), (Re2,St2),
                        AgainIf1s, If2s, NextOf1s, Of2s,
                        Ib1s,      Ib2s, Ob1s,     Ob2s, BcIn, Reph, Rept ).
% 2 --> 1
%       ( Flow < Rel )
node( Node, (Rel,St1), (Re2,St2), If1s,If2s,Of1s,Of2s,
                                Ib1s,Ib2s,Ob1s,Ob2s, BcIn, Reph, Rept ) :-
                If2s = [If|RestIf2s], If = [amount(Flow)|RestIf],
                St1 = open, Flow < Rel |
        NextRel := Rel-Flow,
        Of1s = [ [amount(Flow),link(2,Node)|RestIf] | NextOf1s ],
        node( Node, (NextRel,St1), (Re2,St2),
                        If1s, RestIf2s, NextOf1s, Of2s,
                        Ib1s,     Ib2s, Ob1s,     Ob2s, BcIn, Reph, Rept ).
%       ( Flow = Rel )
node( Node, (Rel,St1), (Re2,St2), If1s,If2s,Of1s,Of2s,
                                Ib1s,Ib2s,Ob1s,Ob2s, BcIn, Reph, Rept ) :-
                If2s = [If|RestIf2s], If = [amount(Flow)|RestIf],
                St1 = open, Flow = Rel |
        Of1s = [ [amount(Flow),link(2,Node)|RestIf] ],
        node( Node, (0,closed), (Re2,St2),
                        If1s, RestIf2s, _,    Of2s,
                        Ib1s,     Ib2s, Ob1s, Ob2s, BcIn, Reph, Rept ).
% 2 ---> 1 & call again
%       ( Flow > Rel )
node( Node, (Rel,St1), (Re2,St2), If1s,If2s,Of1s,Of2s,
                                Ib1s,Ib2s,Ob1s,Ob2s, BcIn, Reph, Rept ) :-
                If2s = [If|RestIf2s], If = [amount(Flow)|RestIf],
                St1 = open, Flow > Rel |
        RestFlow :=  Flow-Rel,
        Of1s = [ [amount(Rel),link(2,Node)|RestIf] | NextOf1s ],
        AgainIf2s = [ [amount(RestFlow)|RestIf] | RestIf2s ],
        node( Node, (0,closed), (Re2,St2),
                        If1s, AgainIf2s, NextOf1s, Of2s,
                        Ib1s,      Ib2s, Ob1s,     Ob2s, BcIn, Reph, Rept ).
% 1 --> 2
%       ( Flow < Re2 )
node( Node, (Rel,St1), (Re2,St2), If1s,If2s,Of1s,Of2s,
                                Ib1s,Ib2s,Ob1s,Ob2s, BcIn, Reph, Rept ) :-
                If1s = [If|Rest1f1s], If = [amount(Flow)|RestIf],
                St1 = closed, St2 = open, Flow < Re2 |
        NextRe2 := Re2-Flow,
        Of2s = [ [amount(Flow),link(1,Node)|RestIf] | NextOf2s ],
        node( Node, (Rel,St1), (NextRe2,St2),
                                RestIf1s, If2s, Of1s, NextOf2s,
                        Ib1s,     Ib2s, Ob1s, Ob2s, BcIn, Reph, Rept ).
%       ( Flow = Re2 )
node( Node, (Rel,St1), (Re2,St2), If1s,If2s,Of1s,Of2s,
                                Ib1s,Ib2s,Ob1s,Ob2s, BcIn, Reph, Rept ) :-
                If1s = [If|Rest1f1s], If = [amount(Flow)|RestIf],
                St1 = closed, St2 = open, Flow = Re2 |
```

```
        Of2s = [ [amount(Flow),link(1,Node)|RestIf] ],
        node( Node, (Re1,St1), (0,closed),
                              RestIf1s, If2s, Of1s, _,
                        Ib1s, Ib2s, Ob1s, Ob2s, BcIn, Reph, Rept ).
% 1 --+--> 2
%     |
% 1<--+
%       ( Flow > Re2 )
node( Node, (Re1,St1), (Re2,St2), If1s,If2s,Of1s,Of2s,
                              Ib1s,Ib2s,Ob1s,Ob2s, BcIn, Reph, Rept ) :-
              If1s = [If|RestIf1s], If = [amount(Flow)|RestIf],
              St1 = closed, St2 = open, Flow > Re2 |
        RestFlow := Flow-Re2 ,
        Of2s = [ [amount(Re2),link(1,Node)|RestIf] ],
        Ob1s = [ [amount(RestFlow)|RestIf]|NextOb1s ],
        node( Node, (Re1,St1), (0,closed),
                      RestIf1s, If2s, Of1s,     _,
                      Ib1s,     Ib2s, NextOb1s, Ob2s, BcIn, Reph, Rept ).
% 2 --> 2
%       ( Flow < Re2 )
node( Node, (Re1,St1), (Re2,St2), If1s,If2s,Of1s,Of2s,
                              Ib1s,Ib2s,Ob1s,Ob2s, BcIn, Reph, Rept ) :-
              If2s = [If|RestIf2s], If = [amount(Flow)|RestIf],
              St1 = closed, St2 = open, Flow < Re2 |
        NextRe2 := Re2-Flow,
        Of2s = [ [amount(Flow),link(2,Node)|RestIf] | NextOf2s ],
        node( Node, (Re1,St1), (NextRe2,St2),
                           If1s, RestIf2s, Of1s, NextOf2s,
                      Ib1s, Ib2s,     Ob1s, Ob2s, BcIn, Reph, Rept ).
%       ( Flow = Re2 )
node( Node, (Re1,St1), (Re2,St2), If1s,If2s,Of1s,Of2s,
                              Ib1s,Ib2s,Ob1s,Ob2s, BcIn, Reph, Rept ) :-
              If2s = [If|RestIf2s], If = [amount(Flow)|RestIf],
              St1 = closed, St2 = open, Flow = Re2 |
        Of2s = [ [amount(Flow),link(2,Node)|RestIf] ],
        node( Node, (Re1,St1), (0,closed),
                       If1s, RestIf2s, Of1s, _,
                      Ib1s, Ib2s,     Ob1s, Ob2s, BcIn, Reph, Rept ).
% 2 --+--> 2
%     |
% 2<--+
%       ( Flow > Re2 )
node( Node, (Re1,St1), (Re2,St2), If1s,If2s,Of1s,Of2s,
                              Ib1s,Ib2s,Ob1s,Ob2s, BcIn, Reph, Rept ) :-
              If2s = [If|RestIf2s], If = [amount(Flow)|RestIf],
              St1 = closed, St2 = open, Flow > Re2 |
        RestFlow := Flow-Re2,
        Of2s = [ [amount(Re2),link(2,Node)|RestIf] ],
        Ob2s = [ [amount(RestFlow)|RestIf]|NextOb2s ],
        node( Node, (Re1,St1), (0,closed),
                      If1s, RestIf2s, Of1s, _,
                      Ib1s,     Ib2s, Ob1s, NextOb2s, BcIn, Reph, Rept ).
%  +-- 1
%  |
%  +-->2
%       ( Flow < Re2 )
node( Node, (Re1,St1), (Re2,St2), If1s,If2s,Of1s,Of2s,
                              Ib1s,Ib2s,Ob1s,Ob2s, BcIn, Reph, Rept ) :-
              Ib1s = [Ib|RestIb1s], Ib = [amount(Flow)|RestIf],
              St2 = open, Flow < Re2 |
        NextRe1 := Re1+Flow, NextRe2 := Re2-Flow, Of2s = [Ib|NextOf2s],
        node( Node, (NextRe1,closed), (NextRe2,St2),
                           If1s,     If2s, _,    NextOf2s,
                      RestIb1s, Ib2s, Ob1s, Ob2s, BcIn, Reph, Rept ).
%       ( Flow = Re2 )
node( Node, (Re1,St1), (Re2,St2), If1s,If2s,Of1s,Of2s,
```

```
                              Ib1s,Ib2s,Ob1s,Ob2s, BcIn, Reph, Rept ) :-
              Ib1s = [Ib|RestIb1s], Ib = [amount(Flow)|RestIf],
              St2 = open, Flow = Re2 |
       NextRe1 := Re1+Flow, Of2s = [Ib],
       node( Node, (NextRe1,closed), (0,closed),
                              If1s,     If2s, _,     _,
                     RestIb1s, Ib2s, Ob1s, Ob2s, BcIn, Reph, Rept ).
% 1<--+-- 1
%     |
%     +-->2
%       ( Flow > Re2 )
node( Node, (Re1,St1), (Re2,St2), If1s,If2s,Of1s,Of2s,
                              Ib1s,Ib2s,Ob1s,Ob2s, BcIn, Reph, Rept ) :-
              Ib1s = [Ib|RestIb1s], Ib = [amount(Flow),link(1,Node)|RestIb],
              St2 = open, Flow > Re2 |
       NextRe1 := Re1+Flow, RestFlow := Flow-Re2,
       Of2s = [[amount(Re2),link(1,Node)|RestIb]],
       Ob1s = [[amount(RestFlow)|RestIb]|NextOb1s],
       node( Node, (NextRe1,closed), (0,closed),
                              If1s,     If2s, _,         _,
                     RestIb1s, Ib2s, NextOb1s, Ob2s, BcIn, Reph, Rept ).
%     +-- 1
%     |
% 2<--+-->2
%       ( Flow > Re2 )
node( Node, (Re1,St1), (Re2,St2), If1s,If2s,Of1s,Of2s,
                              Ib1s,Ib2s,Ob1s,Ob2s, BcIn, Reph, Rept ) :-
              Ib1s = [Ib|RestIb1s], Ib = [amount(Flow),link(2,Node)|RestIb],
              St2 = open, Flow > Re2 |
       NextRe1 := Re1+Flow, RestFlow := Flow-Re2,
       Of2s = [[amount(Re2),link(2,Node)|RestIb]],
       Ob2s = [[amount(RestFlow)|RestIb]|NextOb2s],
       node( Node, (NextRe1,closed), (0,closed),
                              If1s,     If2s, _,     _,
                     RestIb1s, Ib2s, Ob1s, NextOb2s, BcIn, Reph, Rept ).
% 1 <-- 1
node( Node, (Re1,St1), (Re2,St2), If1s,If2s,Of1s,Of2s,
                              Ib1s,Ib2s,Ob1s,Ob2s, BcIn, Reph, Rept ) :-
              Ib1s = [[amount(Flow),link(1,_)|RestIb]|RestIb1s],
              St2 = closed |
       NextRe1 := Re1+Flow, Ob1s = [[amount(Flow)|RestIb]|NextOb1s],
       node( Node, (NextRe1,closed), (Re2,St2),
                              If1s, If2s, _,  _,
                     RestIb1s, Ib2s, NextOb1s, Ob2s, BcIn, Reph, Rept ).
% 2 <-- 1
node( Node, (Re1,St1), (Re2,St2), If1s,If2s,Of1s,Of2s,
                              Ib1s,Ib2s,Ob1s,Ob2s, BcIn, Reph, Rept ) :-
              Ib1s = [[amount(Flow),link(2,_)|RestIb]|RestIb1s],
              St2 = closed |
       NextRe1 := Re1+Flow, Ob2s = [[amount(Flow)|RestIb]|NextOb2s],
       node( Node, (NextRe1,closed), (Re2,St2),
                              If1s, If2s, _,  _,
                     RestIb1s, Ib2s, Ob1s, NextOb2s, BcIn, Reph, Rept ).
% 1 <-- 2
node( Node, (Re1,St1), (Re2,St2), If1s,If2s,Of1s,Of2s,
                              Ib1s,Ib2s,Ob1s,Ob2s, BcIn, Reph, Rept ) :-
              Ib2s = [[amount(Flow),link(1,_)|RestIb]|RestIb2s] |
       NextRe2 := Re2+Flow, Ob1s = [[amount(Flow)|RestIb]|NextOb1s],
       node( Node, (Re1,St1), (NextRe2,closed),
                              If1s, If2s, _,  _,
                     Ib1s, RestIb2s, NextOb1s, Ob2s, BcIn, Reph, Rept ).
% 2 <-- 2
node( Node, (Re1,St1), (Re2,St2), If1s,If2s,Of1s,Of2s,
                              Ib1s,Ib2s,Ob1s,Ob2s, BcIn, Reph, Rept ) :-
              Ib2s = [[amount(Flow),link(2,_)|RestIb]|RestIb2s] |
       NextRe2 := Re2+Flow, Ob2s = [[amount(Flow)|RestIb]|NextOb2s],
```

```
        node( Node, (Re1,St1), (NextRe2,closed),
                              If1s, If2s, _,  _,
                        Ib1s, RestIb2s, Ob1s, NextOb2s, BcIn, Reph, Rept ).
% 1 --+
%      |
% 1<--+
node( Node, (Re1,St1), (Re2,St2), If1s,If2s,Of1s,Of2s,
                                  Ib1s,Ib2s,Ob1s,Ob2s, BcIn, Reph, Rept ) :-
              If1s = [If|RestIf1s], St1 = closed, St2 = closed |
       Ob1s = [If|NextOb1s],
       node( Node, (Re1,St1), (Re2,St2),
                              RestIf1s, If2s, _,        _,
                        Ib1s,     Ib2s, NextOb1s, Ob2s, BcIn, Reph, Rept ).
% 2 --+
%     |
% 2<--+
node( Node, (Re1,St1), (Re2,St2), If1s,If2s,Of1s,Of2s,
                                  Ib1s,Ib2s,Ob1s,Ob2s, BcIn, Reph, Rept ) :-
              If2s = [If|RestIf2s], St1 = closed, St2 = closed |
       Ob2s = [If|NextOb2s],
       node( Node, (Re1,St1), (Re2,St2),
                              If1s, RestIf2s, _,     _,
                        Ib1s, Ib2s,     Ob1s, NextOb2s, BcIn, Reph, Rept ).
% 1,2 <-- end, terminate
node( Node, (Re1,St1), (Re2,St2), If1s,If2s,Of1s,Of2s,
                                  Ib1s,Ib2s,Ob1s,Ob2s, BcIn, Reph, Rept ) :-
              BcIn = end |
       Reph = [node(Node,(Re1,St1),(Re2,St2))|Rept].
%
%       outFlow
%       -------
outFlow( [Pr|RestPrs], Reps, SumFlow, Os ) :- Pr = pass([amount(Flow)|_]) |
       NextSumFlow := SumFlow + Flow, Os = [write(Pr),nl|NextOs],
       outFlow( RestPrs, Reps, NextSumFlow, NextOs ).
outFlow( [Pr|RestPrs], Reps, SumFlow, Os ) :- Pr = return(_) |
       Os = [write(Pr),nl|NextOs],
       outFlow( RestPrs, Reps, SumFlow, NextOs ).
outFlow( Prs, [Rep|RestReps], SumFlow, Os ) :- Rep = node(_,(0,_),(0,_)) |
       outFlow( Prs, RestReps, SumFlow, Os ).
outFlow( Prs, [Rep|RestReps], SumFlow, Os ) :- Rep = node(_,(Re1,_),(Re2,_)),
              Re1+Re2 > 0 |
       Os = [write(remain(Rep)),nl|NextOs],
       outFlow( Prs, RestReps, SumFlow, NextOs ).
outFlow( [], [], SumFlow, Os ) :- true |
       Os = [nl,write(maxFlow(SumFlow)),nl,nl,nl].
```

(0) Date: 1987-July-30, written by Kazuaki Rokusawa
Modified: 3-aug-87 by S. Takagi

(1) Program name: maxflow1-ex2-3

(2) Author: Kazuaki Rokusawa

(3) Runs on: DEC-10 Prolog GHC system (GHC and GHC3)

(4) Description of the problem: See US2:<MPSI.BENCH>MAXFLOW1-EX1.DOC

(5) Algorithm: See US2:<MPSI.BENCH>MAXFLOW1-EX1.DOC

(6) Process structure: See US2:<MPSI.BENCH>MAXFLOW1-EX1.DOC

(7) Pragma:

The pragma (PE-number) is attached to each node process. The user can attach it to some or all node processes. When attaching it to some node processes, the program attaches the pragma to other node processes.

(8) Program:

There are two networks. One (example 2) includes 17 nodes, and the other (example 3) includes 80 nodes (see the source file).

The main predicates used in this program are:

ex2/3, ex3/3: Top level predicates to start this program
ex2 -- example 2, ex3 -- example 3

maxflow/5: Main part of this program

genNodeList/3, findInitAllocPE/3, genNode/4, find2Links/6, findOtherLink/4, initNode/15, findLinkToExit/2:
Predicates that generate node process

entrance/3: Sends the initial flow to the entrance

node/17: The main process which sends and receives a flow.
This process works as:
Waits for the input flow
Changes the status
Sends the flow

exit/6: Detects the end

outFlow/3, outByPE/1, totalByPE/8, outMsgByPE/4:
Displays the result

(9) Source file:US2:<MPSI.BENCH>MAXFLOW2-EX2-3.GHC
Log file: US2:<MPSI.BENCH>MAXFLOW2-EX2-3.LOG (on GHC system)
US2:<MPSI.BENCH>MAXFLOW2-EX2-3-GHC3.LOG (on GHC3 system)
This file: US2:<MPSI.BENCH>MAXFLOW2-EX2-3.DOC

(10) Examples:

Copy the source file (<MPSI.BENCH>MAXFLOW2-EX2-3.GHC) to a temporary file whose file name is less than 7 characters and whose file extension is less than 4 characters: (i.e. up to 6 characters for the file name and 3 characters for the extension).

```
        @copy <MPSI.BENCH>MAXFLOW2-EX2-3.GHC -t.ghc<CR>
```

Invoke the GHC system on the DEC2065.

```
        @ghc<CR>.
```

Compile the program.

```
        ?- ghccompile('-t.ghc').<CR>
```

Execute the program.

```
        ?- ex2(LimFlow, InitAllocList, EntrancePE).<CR>

        % where LimFlow is the amount of initial flow to the entrance,
        % InitAllocList is a list of Pragma (PE-number),
        % EntrancePE is the Pragma(PE-number) attached to the entrance node.

        % Example of InitAllocList:
        %       InitAllocList = [node(k,3),node(h,1),node(q,2)]
        % This means attaching PE#3 to node k, PE#1 to node h,
        %  and PE#2 to node q.
```

(11) Evaluation data:

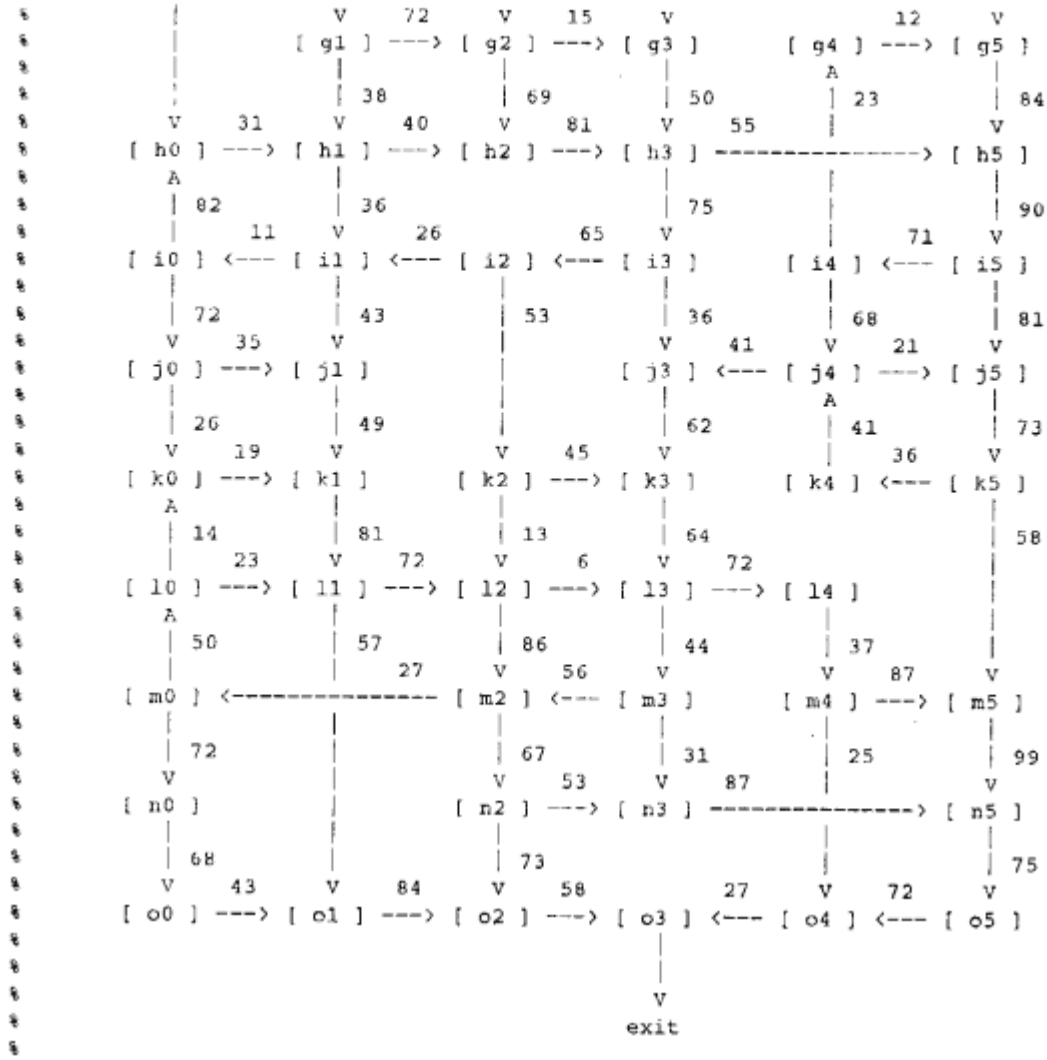
The results of the execution on the GHC3 system are as follows.

```
| ?- ghc ex2(50, [node(f,2),node(g,3),node(p,1)], 1).
        :
maxFlow(40)
        :
639 reductions and 104 suspensions in 81 cycles and 9530 msec (67 rps)
The maximum number of reducible goals is 28/44 at the 20th cycle.
The maximum length of the queue is 44.

yes
| ?- ghc ex3(200, [node(a4,2),node(c0,3),node(d3,4),
|                  node(h1,5),node(i5,6),node(l2,1),node(l3,2)], 1).
        :
maxFlow(83)
        :
7984 reductions and 720 suspensions in 409 cycles and
 xwd(0,-78733) msec (119680 rps)
The maximum number of reducible goals is 81/82 at the 321th cycle.
The maximum length of the queue is 153.
```

xwd(0,-78733) means that the value is more than 18 bits and is represented in the 2 half-words format.

(12) Reference: See US2:<MPSI.BENCH>MAXFLOW1-EX1.DOC

```
%
%        MAXFLOW
%        -------
%
% example. 2      ( 17 nodes )
%
%                        100                      15          20
%    entrance --> [ a ] ---> [ b ]      [ c ] ---> [ d ] ---> [ e ] ---> exit
%                   |          |          A                    A
%                   | 50       | 120      | 20                 | 30
%                   V    20    |          |    40         5    |
%                 [ f ] ------------------> [ g ] ---> [ h ] ---> [ i ]
%                   |          |          A          |         A
%                   | 10       |          | 25       | 20      | 20
%                   V    80    V    40    |          |         |
%                 [ j ] ---> [ k ] ---> [ l ]        |       [ m ]
%                              |          |          |         A
%                              | 10       | 10       |         | 25
%                              V          V    10    V    25   |
%                            [ n ]      [ o ] ---> [ p ] ---> [ q ]
%
ex2( LimFlow, InitAllocList, EntrancePE ) :- true |
        NodeSpecList = [node(a,out(100-b, 50-f)), node(b,out(120-k,   [])),
                        node(c,out( 15-d,   [])), node(d,out( 20-e,   [])),
                        node(e,out( 50-exit,[])), node(f,out( 20-g, 10-j)),
                        node(g,out( 20-c, 40-h)), node(h,out(  5-i, 20-p)),
                        node(i,out( 30-e,   [])), node(j,out( 80-k,   [])),
                        node(k,out( 40-l, 10-n)), node(l,out( 25-g, 10-o)),
                        node(m,out( 20-i,   [])), node(n,out(   [],   [])),
                        node(o,out( 10-p,   [])), node(p,out( 25-g,   [])),
                        node(q,out( 25-m,   []))],
        EntranceNode = a,
        maxflow( LimFlow, NodeSpecList, EntranceNode,
                                              InitAllocList, EntrancePE ).
%
%-------------------------------------------------------------------------------
%
% example. 3      ( 80 nodes )
%
%       entrance
%          |
%          |
%          V    31          40          62          75          89
%       [ a0 ] ---> [ a1 ] ---> [ a2 ] ---> [ a3 ] ---> [ a4 ] ---> [ a5 ]
%          |                      A                      A           |
%          | 76                   | 41                   | 30        | 61
%          V     80               |    52          15    |    15     V
%       [ b0 ] ---------------> [ b2 ] ---> [ b3 ] ---> [ b4 ] ---> [ b5 ]
%          |                      A          |          A           |
%          | 63                   | 14       | 61       | 20        | 72
%          V    70          40    |          |          |    17     V
%       [ c0 ] ---> [ c1 ] ---> [ c2 ]        |       [ c4 ] ---> [ c5 ]
%                     |          |           |          A           |
%                     | 37       | 28        |          | 25        | 68
%                49   V      3   V    46     V     25   |    32     V
%       [ d0 ] <--- [ d1 ] <--- [ d2 ] ---> [ d3 ] ---> [ d4 ] ---> [ d5 ]
%          |          |          A           |                      |
%          | 74       | 22       | 7        | 77                   | 53
%          V    45    V    40    |     61   V      9          23   V
%       [ e0 ] ---> [ e1 ] ---> [ e2 ] <--- [ e3 ] <--- [ e4 ] <--- [ e5 ]
%          |          |          |           |          |           |
%          | 36       | 13       | 99        | 48       | 35        | 61
%          V    26    V          |           V     18   V     8     V
%       [ f0 ] ---> [ f1 ]       |        [ f3 ] <--- [ f4 ] ---> [ f5 ]
%          |          |          |           |                      |
%          | 25       | 54       |           | 39                   | 76
```



```
ex3( LimFlow, InitAllocList, EntrancePE ) :- true ]
        NodeSpecList = [node(a0,out(31-a1,76-b0)), node(a1,out(40-a2,   [])),
                        node(a2,out(62-a3,   [])), node(a3,out(75-a4,   [])),
                        node(a4,out(89-a5,   [])), node(a5,out(61-b5,   [])),
                        node(b0,out(80-b2,63-c0)),
                        node(b2,out(41-a2,52-b3)), node(b3,out(15-b4,61-d3)),
                        node(b4,out(30-a4,15-b5)), node(b5,out(72-c5,   [])),
                        node(c0,out(70-c1,   [])), node(c1,out(40-c2,37-d1)),
                        node(c2,out(14-b2,28-d2)),
                        node(c4,out(20-b4,17-c5)), node(c5,out(68-d5,   [])),
                        node(d0,out(74-e0,   [])), node(d1,out(49-d0,22-e1)),
                        node(d2,out( 3-d1,46-d3)), node(d3,out(25-d4,77-e3)),
                        node(d4,out(25-c4,32-d5)), node(d5,out(53-e5,   [])),
                        node(e0,out(45-e1,36-f0)), node(e1,out(40-e2,13-f1)),
                        node(e2,out( 7-d2,99-g2)), node(e3,out(61-e2,48-f3)),
                        node(e4,out( 9-e3,35-f4)), node(e5,out(23-e4,61-f5)),
                        node(f0,out(26-f1,25-h0)), node(f1,out(54-g1,   [])),
                                                   node(f3,out(39-g3,   [])),
                        node(f4,out(18-f3, 8-f5)), node(f5,out(76-g5,   [])),
                                                   node(g1,out(72-g2,38-h1)),
                        node(g2,out(15-g3,69-h2)), node(g3,out(50-h3,   [])),
                        node(g4,out(12-g5,   [])), node(g5,out(84-h5,   [])),
                        node(h0,out(31-h1,   [])), node(h1,out(40-h2,36-i1)),
                        node(h2,out(81-h3,   [])), node(h3,out(55-h5,75-i3)),
                                                   node(h5,out(90-i5,   [])),
                        node(i0,out(72-j0,82-h0)), node(i1,out(11-i0,43-j1)),
                        node(i2,out(26-i1,53-k2)), node(i3,out(65-i2,36-j3)),
```

```
                        node(i4,out(23-g4,68-j4)), node(i5,out(71-i4,81-j5)),
                        node(j0,out(35-j1,26-k0)), node(j1,out(49-k1,   [])),
                                                   node(j3,out(62-k3,   [])),
                        node(j4,out(41-j3,21-j5)), node(j5,out(73-k5,   [])),
                        node(k0,out(19-k1,   [])), node(k1,out(81-l1,   [])),
                        node(k2,out(45-k3,13-l2)), node(k3,out(64-l3,   [])),
                        node(k4,out(41-j4,   [])), node(k5,out(36-k4,58-m5)),
                        node(l0,out(14-k0,23-l1)), node(l1,out(72-l2,57-o1)),
                        node(l2,out( 6-l3,86-m2)), node(l3,out(72-l4,44-m3)),
                        node(l4,out(37-m4,   [])),
                        node(m0,out(50-l0,72-n0)),
                        node(m2,out(27-m0,67-n2)), node(m3,out(56-m2,31-n3)),
                        node(m4,out(87-m5,25-o4)), node(m5,out(99-n5,   [])),
                        node(n0,out(68-o0,   [])),
                        node(n2,out(53-n3,73-o2)), node(n3,out(87-n5,   [])),
                                                   node(n5,out(75-o5,   [])),
                        node(o0,out(43-o1,   [])), node(o1,out(84-o2,   [])),
                        node(o2,out(58-o3,   [])), node(o3,out(85-exit, [])),
                        node(o4,out(27-o3,   [])), node(o5,out(72-o5,   [])),
        EntranceNode = a0,
        maxflow( LimFlow, NodeSpecList, EntranceNode,
                                              InitAllocList, EntrancePE ).
%
%-------------------------------------------------------------------------
%
%        MAXFLOW         - main part -
%        ---------
%
%       NodeSpecList = [node(name,out(amount1-out1,amount2-out2)),...]
%
%       < example>
%
%       NodeSpecList = [node(a,out(2-b,7-c)),node(b,out(4-d,[])),
%                       node(c,out(5-b,3-d)),node(d,out(7-exit,[]))]
%
%                         2          4          7
%                  [ a ] ---> [ b ] ---> [ d ] ---> exit
%                    |          A          A
%                    |   7      | 5    3   |
%                    +------> [ c ] -------+
%
maxflow( LimFlow, NodeSpecList, EntranceNode, InitAllocList, EntrancePE ) :-
                true |
        Ent = (node(_,out(_-EntranceNode,[])),
                    channel(ToEntrance,_,BackToExit,_),_),
        genNodeList( NodeSpecList, InitAllocList, NodeList ),
        genNode( NodeList, [Ent|NodeList], BcIn, RepHead ),
        entrance( LimFlow, EntrancePE, ToEntrance ),
        findLinkToExit( NodeList, ToExit ),
        exit( LimFlow, 0, ToExit, BackToExit, BcIn, Prs ),
        outFlow( Prs, RepHead, 0 ),
        outByPE( RepHead ).
%
%-------------------------------------------------------------------------
%       process generation
%-------------------------------------------------------------------------
%
%       genNodeList
%      -------------
%
%       NodeSpecList ----> NodeList
%
%       NodeList = [(node(...),channel(Of1s,Of2s,Ib1s,Ib2s),pe(PE#)),...]
%               ( where node(...) is the same as node(...) in NodeSpecList )
%
genNodeList( [OneNodeSpec|RestNodeSpecList], InitAllocList, NodeList ) :-
```

```
                OneNodeSpec = node(Node,_) |
        findInitAllocPE( Node, InitAllocList, InitAllocPE ),
        OneNode = (OneNodeSpec,channel(Of1s,Of2s,Ib1s,Ib2s),pe(InitAllocPE)),
        NodeList = [OneNode|NextNodeList],
        genNodeList( RestNodeSpecList, InitAllocList, NextNodeList ).
genNodeList( [], _, NodeList ) :- true |
        NodeList = [].
%
%       findInitAllocPE
%      -----------------
%
%       find the initially allocated PE according to InitAllocList
%
findInitAllocPE( Node, InitAllocList, InitAllocPE ) :-
                InitAllocList = [node(Node,PE)|_] |
        InitAllocPE = PE.
findInitAllocPE( Node, InitAllocList, InitAllocPE ) :-
                InitAllocList = [node(OtherNode,_)|RestInitAllocList],
                Node \= OtherNode |
        findInitAllocPE( Node, RestInitAllocList, InitAllocPE ).
findInitAllocPE( _, InitAllocList, InitAllocPE ) :- InitAllocList = [] |
        InitAllocPE = notAllocated.
%
%       genNode
%      ---------
%
%       generation of "node" process
%
%       ANodeList = [entrance-spec|NodeList]     ( see maxflow predicate )
%
%               Of1s, Ib1s
%       [node] ------------>
%          |
%          |    Of2s, Ib2s
%          +---------------->
%
genNode( [OneNode|RestNodeList], ANodeList, BcIn, Reph ) :-
                OneNode = (node(Node,out(Amount1-_,Amount2-_)),
                           channel(Of1s,Of2s,Ib1s,Ib2s),pe(AllocPE)) |
        find2Links( Node, ANodeList, If1s, If2s, Ob1s, Ob2s ),
        initNode( AllocPE, Node, (Amount1,open), (Amount2,open),
              If1s,If2s, Of1s,Of2s, Ib1s,Ib2s, Ob1s,Ob2s, BcIn, Reph,Rept ),
        genNode( RestNodeList, ANodeList, BcIn, Rept ).
%
%               Of1s, Ib1s
%       [node] ------------>
%
genNode( [OneNode|RestNodeList], ANodeList, BcIn, Reph ) :-
                OneNode = (node(Node,out(Amount1-_,[])),
                           channel(Of1s,_,Ib1s,_),pe(AllocPE)) |
        find2Links( Node, ANodeList, If1s, If2s, Ob1s, Ob2s ),
        initNode( AllocPE, Node, (Amount1,open), (0,closed),
              If1s,If2s, Of1s,_, Ib1s,[], Ob1s,Ob2s, BcIn, Reph,Rept ),
        genNode( RestNodeList, ANodeList, BcIn, Rept ).
%
%       [node]
%          |
%          |    Of2s, Ib2s
%          +---------------->
%
genNode( [OneNode|RestNodeList], ANodeList, BcIn, Reph ) :-
                OneNode = (node(Node,out([],Amount2-_)),
                           channel(_,Of2s,_,Ib2s),pe(AllocPE)) |
        find2Links( Node, ANodeList, If1s, If2s, Ob1s, Ob2s ),
        initNode( AllocPE, Node, (0,closed), (Amount2,open),
              If1s,If2s, _,Of2s, [],Ib2s, Ob1s,Ob2s, BcIn, Reph,Rept ),
```

```
        genNode( RestNodeList, ANodeList, BcIn, Rept ).
%
%       [node]          ( having no output-link )
%
genNode( [OneNode|RestNodeList], ANodeList, BcIn, Reph ) :-
                OneNode = (node(Node,out([],[])),_,pe(AllocPE)) |
        find2Links( Node, ANodeList, If1s, If2s, Ob1s, Ob2s ),
        initNode( AllocPE, Node, (0,closed), (0,closed),
              If1s,If2s, _,_, [],[], Ob1s,Ob2s, BcIn, Reph,Rept ),
        genNode( RestNodeList, ANodeList, BcIn, Rept ).
%
genNode( [], _, _, Reph ) :- true |
        Reph = [].
%
%       findLink
%      ----------
%
%       find the link to node(Node)
%
%                       Of1s = If1s, Ib1s = Ob1s
%       [other-node] ------------------------> [Node]
%                                                A
%                                                |
%                       Of2s = If2s, Ib2s = Ob2s |
%       [other-node] -----------------------------+
%
find2Links( Node, [OneNode|RestANodeList], If1s, If2s, Ob1s, Ob2s ) :-
                OneNode = (node(_,out(_,Node,_)),channel(Of1s,_,Ib1s,_),_) |
        If1s = Of1s, Ob1s = Ib1s,
        findOtherLink( Node, RestANodeList, If2s, Ob2s ).
find2Links( Node, [OneNode|RestANodeList], If1s, If2s, Ob1s, Ob2s ) :-
                OneNode = (node(_,out(_,_,Node)),channel(_,Of2s,_,Ib2s),_) |
        If1s = Of2s, Ob1s = Ib2s,
        findOtherLink( Node, RestANodeList, If2s, Ob2s ).
find2Links( Node, [OneNode|RestANodeList], If1s, If2s, Ob1s, Ob2s ) :-
                OneNode = (node(_,out(_,_,OutNode1,_,OutNode2)),_,_),
                OutNode1 \= Node, OutNode2 \= Node |
        find2Links( Node, RestANodeList, If1s, If2s, Ob1s, Ob2s ).
find2Links( Node, [OneNode|RestANodeList], If1s, If2s, Ob1s, Ob2s ) :-
                OneNode = (node(_,out(_,OutNode1,[])),_,_), OutNode1 \= Node |
        find2Links( Node, RestANodeList, If1s, If2s, Ob1s, Ob2s ).
find2Links( Node, [OneNode|RestANodeList], If1s, If2s, Ob1s, Ob2s ) :-
                OneNode = (node(_,out([],_,OutNode2)),_,_), OutNode2 \= Node |
        find2Links( Node, RestANodeList, If1s, If2s, Ob1s, Ob2s ).
find2Links( Node, [OneNode|RestANodeList], If1s, If2s, Ob1s, Ob2s ) :-
                OneNode = (node(_,out([],[])),_,_) |
        find2Links( Node, RestANodeList, If1s, If2s, Ob1s, Ob2s ).
find2Links( Node, [], If1s, If2s, Ob1s, Ob2s ) :- true |
        If1s = [], If2s = [], Ob1s = _, Ob2s = _.
%
findOtherLink( Node, [OneNode|RestANodeList], If1s, Ob1s ) :-
                OneNode = (node(_,out(_,Node,_)),channel(Of1s,_,Ib1s,_),_) |
        If1s = Of1s, Ob1s = Ib1s.
findOtherLink( Node, [OneNode|RestANodeList], If1s, Ob1s ) :-
                OneNode = (node(_,out(_,_,Node)),channel(_,Of2s,_,Ib2s),_) |
        If1s = Of2s, Ob1s = Ib2s.
findOtherLink( Node, [OneNode|RestANodeList], If1s, Ob1s ) :-
                OneNode = (node(_,out(_,_,OutNode1,_,OutNode2)),_,_),
                OutNode1 \= Node, OutNode2 \= Node |
        findOtherLink( Node, RestANodeList, If1s, Ob1s ).
findOtherLink( Node, [OneNode|RestANodeList], If1s, Ob1s ) :-
                OneNode = (node(_,out(_,OutNode1,[])),_,_), OutNode1 \= Node |
        findOtherLink( Node, RestANodeList, If1s, Ob1s ).
findOtherLink( Node, [OneNode|RestANodeList], If1s, Ob1s ) :-
                OneNode = (node(_,out([],_,OutNode2)),_,_), OutNode2 \= Node |
        findOtherLink( Node, RestANodeList, If1s, Ob1s ).
```

```
findOtherLink( Node, [OneNode|RestANodeList], Ifs, Obs ) :-
                OneNode = (node(_,out([],[])),_,_) |
        findOtherLink( Node, RestANodeList, Ifs, Obs ).
findOtherLink( Node, [], Ifs, Obs ) :- true |
        Ifs = [], Obs = _.
%
%       initNode
%      -----------
%
%       initial generation of "node" process
%
%       ---  initially allocated  ---
%
initNode( AllocPE, Node, (Re1,St1), (Re2,St2),
            Ifls,If2s,Ofls,Of2s, Ibls,Ib2s,Obls,Ob2s, BcIn, Reph, Rept ) :-
                AllocPE \= notAllocated |
        InitIfls = [alloc(AllocPE)|Ifls],
        node( notAllocated, Node, 0, 0, (Rel,St1), (Re2,St2),
              InitIfls,If2s,Ofls,Of2s,Ibls,Ib2s,Obls,Ob2s, BcIn, Reph, Rept ).
%
%       ---  initially not allocated  ---
%
initNode( AllocPE, Node, (Rel,St1), (Re2,St2),
            Ifls,If2s,Ofls,Of2s, Ibls,Ib2s,Obls,Ob2s, BcIn, Reph, Rept ) :-
                AllocPE = notAllocated |
        node( notAllocated, Node, 0, 0, (Rel,St1), (Re2,St2),
              Ifls,If2s,Ofls,Of2s,Ibls,Ib2s,Obls,Ob2s, BcIn, Reph, Rept ).
%
%       findLinkToExit
%      ----------------
%
%       find the link to "exit"
%
%                       Ofls or Of2s = ToExit
%       [other-node] ------------------------> exit
%
findLinkToExit( [OneNode|RestNodeList], ToExit ) :-
                OneNode = (node(_,out(_-OutNode1,_-OutNode2)),_,_),
                OutNode1 \= exit, OutNode2 \= exit |
        findLinkToExit( RestNodeList, ToExit ).
findLinkToExit( [OneNode|RestNodeList], ToExit ) :-
                OneNode = (node(_,out(_-OutNode1,[])),_,_), OutNode1 \= exit |
        findLinkToExit( RestNodeList, ToExit ).
findLinkToExit( [OneNode|RestNodeList], ToExit ) :-
                OneNode = (node(_,out([],_-OutNode2)),_,_), OutNode2 \= exit |
        findLinkToExit( RestNodeList, ToExit ).
findLinkToExit( [OneNode|RestNodeList], ToExit ) :-
                OneNode = (node(_,out([],[])),_,_) |
        findLinkToExit( RestNodeList, ToExit ).
findLinkToExit( [OneNode|RestNodeList], ToExit ) :-
                OneNode = (node(_,out(_-exit,_)),channel(Ofls,_,_,_),_) |
        ToExit = Ofls.
findLinkToExit( [OneNode|RestNodeList], ToExit ) :-
                OneNode = (node(_,out(_,_-exit)),channel(_,Of2s,_,_),_) |
        ToExit = Of2s.
findLinkToExit( [], _ ) :- Msg = 'illegal NodeSpecList',
                prolog( ( write( Msg ), nl ) ) |
        true.
%
%-------------------------------------------------------------------------------
%       process specification
%-------------------------------------------------------------------------------
%
%                   +-----------------------------+
%                   |                             |
%                   |                             v
```

```
%    entrance ---> node <--> ... <--> node <--> exit --> outFlow --> outstream
%                   A                   A          |           A
%                   |                   |          V           |
%                   V                   V      all nodes  all nodes
%                  node <--> ... <--> node
%                   A                   A
%                   |                   |
%                   :                   :
%
%                        stream diagram
%                       ----------------
%
%                            node
%                       +------------+ (Re1,St1)
%             --------->|If1s    Of1s| --------->
%            <---------- |Ob1s    Ib1s|<----------
%                        |            |
%                        |            | (Re2,St2)
%             --------->|If2s    Of2s| --------->
%            <---------- |Ob2s    Ib2s|<----------
%                        |            |
%                         BcIn   Rep  |
%                       +------------+
%                          A      |
%                          |      V
%
%               Ifs  : input  forward  stream
%               Ofs  : output forward  stream
%               Ibs  : input  backward stream
%               Obs  : output backward stream
%               Re   : remainder
%               St   : status ( open / closed )
%               BcIn : broadcast in
%               Reph : reply head
%               Rept : reply tail
%
%                    node specification
%                   --------------------
%
%       entrance
%      ----------
entrance( LimFlow, EntrancePE, Ofs ) :- Msg = push(LimFlow),
                prolog( ( write( Msg ), nl ) ) |
        Ofs = [alloc(EntrancePE),[amount(LimFlow),link(0,entrance)]].
%
%       node
%      ------
%
% allocation to PE specified in the message
%
%     receiving "alloc-message" from If1s
%
%     ---  first receiving  ---
%
node( PE, Node, Ifc, Ibc, (Re1,St1), (Re2,St2), If1s,If2s,Of1s,Of2s,
                                  Ib1s,Ib2s,Ob1s,Ob2s, BcIn, Reph, Rept ) :-
                If1s = [If|RestIf1s], If = alloc(AllocPE), PE = notAllocated,
                Msg = allocate(node(Node),pe(AllocPE)),
                prolog( ( write(Msg), nl ) ) |
        sendAlloc( AllocPE, St1, Of1s, NextOf1s ),
        sendAlloc( AllocPE, St2, Of2s, NextOf2s ),
        node( AllocPE, Node, Ifc, Ibc, (Re1,St1), (Re2,St2),
                        RestIf1s, If2s, NextOf1s, NextOf2s,
                        Ib1s,     Ib2s, Ob1s,     Ob2s, BcIn, Reph, Rept ).
%
%     ---  already received  ---
```

```
%
node( PE, Node, Ifc, Ibc, (Rel,St1), (Re2,St2), If1s,If2s,Of1s,Of2s,
                                    Ib1s,Ib2s,Ob1s,Ob2s, BcIn, Reph, Rept ) :-
                If1s = [If|RestIf1s], If = alloc(AllocPE), PE \= notAllocated |
        node( PE, Node, Ifc, Ibc, (Rel,St1), (Re2,St2),
                        RestIf1s, If2s, Of1s, Of2s,
                        Ib1s,     Ib2s, Ob1s, Ob2s, BcIn, Reph, Rept ).
%
%     receiving "alloc-message" from If2s
%
%     ---  first receiving  ---
%
node( PE, Node, Ifc, Ibc, (Rel,St1), (Re2,St2), If1s,If2s,Of1s,Of2s,
                                    Ib1s,Ib2s,Ob1s,Ob2s, BcIn, Reph, Rept ) :-
                If2s = [If|RestIf2s], If = alloc(AllocPE), PE = notAllocated,
                Msg = allocate(node(Node),pe(AllocPE)),
                prolog( ( write(Msg), nl ) ) |
        sendAlloc( AllocPE, St1, Of1s, NextOf1s ),
        sendAlloc( AllocPE, St2, Of2s, NextOf2s ),
        node( AllocPE, Node, Ifc, Ibc, (Rel,St1), (Re2,St2),
                        If1s, RestIf2s, NextOf1s, NextOf2s,
                        Ib1s,     Ib2s, Ob1s,     Ob2s, BcIn, Reph, Rept ).
%
%     ---  already received  ---
%
node( PE, Node, Ifc, Ibc, (Rel,St1), (Re2,St2), If1s,If2s,Of1s,Of2s,
                                    Ib1s,Ib2s,Ob1s,Ob2s, BcIn, Reph, Rept ) :-
                If2s = [If|RestIf2s], If = alloc(AllocPE), PE \= notAllocated |
        node( PE, Node, Ifc, Ibc, (Rel,St1), (Re2,St2),
                        If1s, RestIf2s, Of1s, Of2s,
                        Ib1s,     Ib2s, Ob1s, Ob2s, BcIn, Reph, Rept ).
%
sendAlloc( AllocPE, St, Ofs, NextOfs ) :- St = open |
        Ofs = [alloc(AllocPE)|NextOfs].
%
sendAlloc( AllocPE, St, Ofs, NextOfs ) :- St = closed |
        Ofs = NextOfs.
%
% general processing
%
%     receiving [amount(Flow),link(_,_),...] message
%
% 1 --> 1
%       ( Flow < Rel )
node( PE, Node, Ifc, Ibc, (Rel,St1), (Re2,St2), If1s,If2s,Of1s,Of2s,
                                    Ib1s,Ib2s,Ob1s,Ob2s, BcIn, Reph, Rept ) :-
                If1s = [If|RestIf1s], If = [amount(Flow)|RestIf],
                St1 = open, Flow < Rel, NextRel := Rel-Flow, NewIfc := Ifc+1,
                Msg = ope(pe(PE),node(Node),send_forward(Flow)),
                prolog( ( write(Msg), nl ) ) |
        Of1s = [ [amount(Flow),link(1,Node)|RestIf] | NextOf1s ],
        node( PE, Node, NewIfc, Ibc, (NextRel,St1), (Re2,St2),
                        RestIf1s, If2s, NextOf1s, Of2s,
                        Ib1s,     Ib2s, Ob1s,     Ob2s, BcIn, Reph, Rept ).
%       ( Flow = Rel )
node( PE, Node, Ifc, Ibc, (Rel,St1), (Re2,St2), If1s,If2s,Of1s,Of2s,
                                    Ib1s,Ib2s,Ob1s,Ob2s, BcIn, Reph, Rept ) :-
                If1s = [If|RestIf1s], If = [amount(Flow)|RestIf],
                St1 = open, Flow = Rel, NewIfc := Ifc+1,
                Msg = ope(pe(PE),node(Node),send_forward(Flow)),
                prolog( ( write(Msg), nl ) ) |
        Of1s = [ [amount(Flow),link(1,Node)|RestIf] ],
        node( PE, Node, NewIfc, Ibc, (0,closed), (Re2,St2),
                        RestIf1s, If2s, _,     Of2s,
                        Ib1s,     Ib2s, Ob1s, Ob2s, BcIn, Reph, Rept ).
% 1 ---> 1 & call again
```

```
%       ( Flow > Rel )
node( PE, Node, Ifc, Ibc, (Rel,St1), (Re2,St2), If1s,If2s,Of1s,Of2s,
                                  Ib1s,Ib2s,Ob1s,Ob2s, BcIn, Reph, Rept ) :-
                If1s = [If|RestIf1s], If = [amount(Flow)|RestIf],
                St1 = open, Flow > Rel, RestFlow :=  Flow-Rel, NewIfc := Ifc+1,
                Msg = ope(pe(PE),node(Node),send_forward(Rel)),
                prolog( ( write(Msg), nl ) ) |
        Of1s = [ [amount(Rel),link(1,Node)|RestIf] | NextOf1s ],
        AgainIf1s = [ [amount(RestFlow)|RestIf] | RestIf1s ],
        node( PE, Node, NewIfc, Ibc, (0,closed), (Re2,St2),
                        AgainIf1s, If2s, NextOf1s, Of2s,
                        Ib1s,      Ib2s, Ob1s,     Ob2s, BcIn, Reph, Rept ).
% 2 --> 1
%       ( Flow < Rel )
node( PE, Node, Ifc, Ibc, (Rel,St1), (Re2,St2), If1s,If2s,Of1s,Of2s,
                                  Ib1s,Ib2s,Ob1s,Ob2s, BcIn, Reph, Rept ) :-
                If2s = [If|RestIf2s], If = [amount(Flow)|RestIf],
                St1 = open, Flow < Rel, NextRel := Rel-Flow, NewIfc := Ifc+1,
                Msg = ope(pe(PE),node(Node),send_forward(Flow)),
                prolog( ( write(Msg), nl ) ) |
        Of1s = [ [amount(Flow),link(2,Node)|RestIf] | NextOf1s ],
        node( PE, Node, NewIfc, Ibc, (NextRel,St1), (Re2,St2),
                        If1s, RestIf2s, NextOf1s, Of2s,
                        Ib1s,      Ib2s, Ob1s,     Ob2s, BcIn, Reph, Rept ).
%       ( Flow = Rel )
node( PE, Node, Ifc, Ibc, (Rel,St1), (Re2,St2), If1s,If2s,Of1s,Of2s,
                                  Ib1s,Ib2s,Ob1s,Ob2s, BcIn, Reph, Rept ) :-
                If2s = [If|RestIf2s], If = [amount(Flow)|RestIf],
                St1 = open, Flow = Rel, NewIfc := Ifc+1,
                Msg = ope(pe(PE),node(Node),send_forward(Flow)),
                prolog( ( write(Msg), nl ) ) |
        Of1s = [ [amount(Flow),link(2,Node)|RestIf] ],
        node( PE, Node, NewIfc, Ibc, (0,closed), (Re2,St2),
                        If1s, RestIf2s, _,     Of2s,
                        Ib1s,      Ib2s, Ob1s, Ob2s, BcIn, Reph, Rept ).
% 2 ---> 1 & call again
%       ( Flow > Rel )
node( PE, Node, Ifc, Ibc, (Rel,St1), (Re2,St2), If1s,If2s,Of1s,Of2s,
                                  Ib1s,Ib2s,Ob1s,Ob2s, BcIn, Reph, Rept ) :-
                If2s = [If|RestIf2s], If = [amount(Flow)|RestIf],
                St1 = open, Flow > Rel, RestFlow :=  Flow-Rel, NewIfc := Ifc+1,
                Msg = ope(pe(PE),node(Node),send_forward(Rel)),
                prolog( ( write(Msg), nl ) ) |
        Of1s = [ [amount(Rel),link(2,Node)|RestIf] | NextOf1s ],
        AgainIf2s = [ [amount(RestFlow)|RestIf] | RestIf2s ],
        node( PE, Node, NewIfc, Ibc, (0,closed), (Re2,St2),
                        If1s, AgainIf2s, NextOf1s, Of2s,
                        Ib1s,      Ib2s, Ob1s,     Ob2s, BcIn, Reph, Rept ).
% 1 --> 2
%       ( Flow < Re2 )
node( PE, Node, Ifc, Ibc, (Rel,St1), (Re2,St2), If1s,If2s,Of1s,Of2s,
                                  Ib1s,Ib2s,Ob1s,Ob2s, BcIn, Reph, Rept ) :-
                If1s = [If|RestIf1s], If = [amount(Flow)|RestIf],
                St1 = closed, St2 = open, Flow < Re2, NextRe2 := Re2-Flow,
                NewIfc := Ifc+1,
                Msg = ope(pe(PE),node(Node),send_forward(Flow)),
                prolog( ( write(Msg), nl ) ) |
        Of2s = [ [amount(Flow),link(1,Node)|RestIf] | NextOf2s ],
        node( PE, Node, NewIfc, Ibc, (Rel,St1), (NextRe2,St2),
                        RestIf1s, If2s, Of1s, NextOf2s,
                        Ib1s,      Ib2s, Ob1s, Ob2s, BcIn, Reph, Rept ).
%       ( Flow = Re2 )
node( PE, Node, Ifc, Ibc, (Rel,St1), (Re2,St2), If1s,If2s,Of1s,Of2s,
                                  Ib1s,Ib2s,Ob1s,Ob2s, BcIn, Reph, Rept ) :-
                If1s = [If|RestIf1s], If = [amount(Flow)|RestIf],
                St1 = closed, St2 = open, Flow = Re2, NewIfc := Ifc+1,
```

```
                Msg = ope(pe(PE),node(Node),send_forward(Flow)),
                prolog( ( write(Msg), nl ) ) |
        Of2s = [ [amount(Flow),link(1,Node)|RestIf] ],
        node( PE, Node, NewIfc, Ibc, (Rel,St1), (0,closed),
                        RestIf1s, If2s, Of1s, _,
                        Ib1s, Ib2s, Ob1s, Ob2s, BcIn, Reph, Rept ).
% 1 --+--> 2
%     |
% 1<--+
%       ( Flow > Re2 )
node( PE, Node, Ifc, Ibc, (Rel,St1), (Re2,St2), If1s,If2s,Of1s,Of2s,
                                Ib1s,Ib2s,Ob1s,Ob2s, BcIn, Reph, Rept ) :-
                If1s = [If|RestIf1s], If = [amount(Flow)|RestIf],
                St1 = closed, St2 = open, Flow > Re2, RestFlow :=  Flow-Re2,
                NewIfc := Ifc+1,
                Msg = ope(pe(PE),node(Node),
                                        send_forward(Re2),send_back(RestFlow)),
                prolog( ( write(Msg), nl ) ) |
        Of2s = [ [amount(Re2),link(1,Node)|RestIf] ],
        Ob1s = [ [amount(RestFlow)|RestIf]|NextOb1s ],
        node( PE, Node, NewIfc, Ibc, (Rel,St1), (0,closed),
                        RestIf1s, If2s, Of1s,     _,
                        Ib1s,     Ib2s, NextOb1s, Ob2s, BcIn, Reph, Rept ).
% 2 --> 2
%       ( Flow < Re2 )
node( PE, Node, Ifc, Ibc, (Rel,St1), (Re2,St2), If1s,If2s,Of1s,Of2s,
                                Ib1s,Ib2s,Ob1s,Ob2s, BcIn, Reph, Rept ) :-
                If2s = [If|RestIf2s], If = [amount(Flow)|RestIf],
                St1 = closed, St2 = open, Flow < Re2, NextRe2 := Re2-Flow,
                NewIfc := Ifc+1,
                Msg = ope(pe(PE),node(Node),send_forward(Flow)),
                prolog( ( write(Msg), nl ) ) |
        Of2s = [ [amount(Flow),link(2,Node)|RestIf] | NextOf2s ],
        node( PE, Node, NewIfc, Ibc, (Rel,St1), (NextRe2,St2),
                        If1s, RestIf2s, Of1s, NextOf2s,
                        Ib1s, Ib2s,     Ob1s, Ob2s, BcIn, Reph, Rept ).
%       ( Flow = Re2 )
node( PE, Node, Ifc, Ibc, (Rel,St1), (Re2,St2), If1s,If2s,Of1s,Of2s,
                                Ib1s,Ib2s,Ob1s,Ob2s, BcIn, Reph, Rept ) :-
                If2s = [If|RestIf2s], If = [amount(Flow)|RestIf],
                St1 = closed, St2 = open, Flow = Re2, NewIfc := Ifc+1,
                Msg = ope(pe(PE),node(Node),send_forward(Flow)),
                prolog( ( write(Msg), nl ) ) |
        Of2s = [ [amount(Flow),link(2,Node)|RestIf] ],
        node( PE, Node, NewIfc, Ibc, (Rel,St1), (0,closed),
                        If1s, RestIf2s, Of1s, _,
                        Ib1s, Ib2s,     Ob1s, Ob2s, BcIn, Reph, Rept ).
% 2 --+--> 2
%     |
% 2<--+
%       ( Flow > Re2 )
node( PE, Node, Ifc, Ibc, (Rel,St1), (Re2,St2), If1s,If2s,Of1s,Of2s,
                                Ib1s,Ib2s,Ob1s,Ob2s, BcIn, Reph, Rept ) :-
                If2s = [If|RestIf2s], If = [amount(Flow)|RestIf],
                St1 = closed, St2 = open, Flow > Re2, RestFlow :=  Flow-Re2,
                NewIfc := Ifc+1,
                Msg = ope(pe(PE),node(Node),
                                        send_forward(Re2),send_back(RestFlow)),
                prolog( ( write(Msg), nl ) ) |
        Of2s = [ [amount(Re2),link(2,Node)|RestIf] ],
        Ob2s = [ [amount(RestFlow)|RestIf]|NextOb2s ],
        node( PE, Node, NewIfc, Ibc, (Rel,St1), (0,closed),
                        If1s, RestIf2s, Of1s, _,
                        Ib1s,     Ib2s, Ob1s, NextOb2s, BcIn, Reph, Rept ).
%  +-- 1
%  |
```

```
%  +-->2
%       ( Flow < Re2 )
node( PE, Node, Ifc, Ibc, (Re1,St1), (Re2,St2), If1s,If2s,Of1s,Of2s,
                                  Ib1s,Ib2s,Ob1s,Ob2s, BcIn, Reph, Rept ) :-
                Ib1s = [Ib|RestIb1s], Ib = [amount(Flow)|RestIf],
                St2 = open, Flow < Re2,
                NextRe1 := Re1+Flow, NextRe2 := Re2-Flow, NewIbc := Ibc+1,
                Msg = ope(pe(PE),node(Node),send_forward(Flow)),
                prolog( ( write(Msg), nl ) ) |
        Of2s = [Ib|NextOf2s],
        node( PE, Node, Ifc, NewIbc, (NextRe1,closed), (NextRe2,St2),
                        If1s,     If2s, _,    NextOf2s,
                        RestIb1s, Ib2s, Ob1s, Ob2s, BcIn, Reph, Rept ).
%       ( Flow = Re2 )
node( PE, Node, Ifc, Ibc, (Re1,St1), (Re2,St2), If1s,If2s,Of1s,Of2s,
                                  Ib1s,Ib2s,Ob1s,Ob2s, BcIn, Reph, Rept ) :-
                Ib1s = [Ib|RestIb1s], Ib = [amount(Flow)|RestIf],
                St2 = open, Flow = Re2, NextRe1 := Re1+Flow, NewIbc := Ibc+1,
                Msg = ope(pe(PE),node(Node),send_forward(Flow)),
                prolog( ( write(Msg), nl ) ) |
        Of2s = [Ib],
        node( PE, Node, Ifc, NewIbc, (NextRe1,closed), (0,closed),
                        If1s,     If2s, _,    _,
                        RestIb1s, Ib2s, Ob1s, Ob2s, BcIn, Reph, Rept ).
% 1<--+-- 1
%     |
%     +-->2
%       ( Flow > Re2 )
node( PE, Node, Ifc, Ibc, (Re1,St1), (Re2,St2), If1s,If2s,Of1s,Of2s,
                                  Ib1s,Ib2s,Ob1s,Ob2s, BcIn, Reph, Rept ) :-
                Ib1s = [Ib|RestIb1s], Ib = [amount(Flow),link(1,Node)|RestIb],
                St2 = open, Flow > Re2,
                NextRe1 := Re1+Flow, RestFlow := Flow-Re2, NewIbc := Ibc+1,
                Msg = ope(pe(PE),node(Node),
                                        send_forward(Re2),send_back(RestFlow)),
                prolog( ( write(Msg), nl ) ) |
        Of2s = [[amount(Re2),link(1,Node)|RestIb]],
        Ob1s = [[amount(RestFlow)|RestIb]|NextOb1s],
        node( PE, Node, Ifc, NewIbc, (NextRe1,closed), (0,closed),
                        If1s,     If2s, _,        _,
                        RestIb1s, Ib2s, NextOb1s, Ob2s, BcIn, Reph, Rept ).
%     +-- 1
%     |
% 2<--+-->2
%       ( Flow > Re2 )
node( PE, Node, Ifc, Ibc, (Re1,St1), (Re2,St2), If1s,If2s,Of1s,Of2s,
                                  Ib1s,Ib2s,Ob1s,Ob2s, BcIn, Reph, Rept ) :-
                Ib1s = [Ib|RestIb1s], Ib = [amount(Flow),link(2,Node)|RestIb],
                St2 = open, Flow > Re2,
                NextRe1 := Re1+Flow, RestFlow := Flow-Re2, NewIbc := Ibc+1,
                Msg = ope(pe(PE),node(Node),
                                        send_forward(Re2),send_back(RestFlow)),
                prolog( ( write(Msg), nl ) ) |
        Of2s = [[amount(Re2),link(2,Node)|RestIb]],
        Ob2s = [[amount(RestFlow)|RestIb]|NextOb2s],
        node( PE, Node, Ifc, NewIbc, (NextRe1,closed), (0,closed),
                        If1s,     If2s, _,    _,
                        RestIb1s, Ib2s, Ob1s, NextOb2s, BcIn, Reph, Rept ).
% 1 <-- 1
node( PE, Node, Ifc, Ibc, (Re1,St1), (Re2,St2), If1s,If2s,Of1s,Of2s,
                                  Ib1s,Ib2s,Ob1s,Ob2s, BcIn, Reph, Rept ) :-
                Ib1s = [[amount(Flow),link(1,_)|RestIb]|RestIb1s],
                St2 = closed, NextRe1 := Re1+Flow, NewIbc := Ibc+1,
                Msg = ope(pe(PE),node(Node),send_back(Flow)),
                prolog( ( write(Msg), nl ) ) |
        Ob1s = [[amount(Flow)|RestIb]|NextOb1s],
```

```
        node( PE, Node, Ifc, NewIbc, (NextRe1,closed), (Re2,St2),
                        If1s, If2s, _,  _,
                        RestIb1s, Ib2s, NextOb1s, Ob2s, BcIn, Reph, Rept ).
% 2 <-- 1
node( PE, Node, Ifc, Ibc, (Re1,St1), (Re2,St2), If1s,If2s,Of1s,Of2s,
                                Ib1s,Ib2s,Ob1s,Ob2s, BcIn, Reph, Rept ) :-
                Ib1s = [[amount(Flow),link(2,_)|RestIb]|RestIb1s],
                St2 = closed, NextRe1 := Re1+Flow, NewIbc := Ibc+1,
                Msg = ope(pe(PE),node(Node),send_back(Flow)),
                prolog( ( write(Msg), nl ) ) |
        Ob2s = [[amount(Flow)|RestIb]|NextOb2s],
        node( PE, Node, Ifc, NewIbc, (NextRe1,closed), (Re2,St2),
                        If1s, If2s, _,  _,
                        RestIb1s, Ib2s, Ob1s, NextOb2s, BcIn, Reph, Rept ).
% 1 <-- 2
node( PE, Node, Ifc, Ibc, (Re1,St1), (Re2,St2), If1s,If2s,Of1s,Of2s,
                                Ib1s,Ib2s,Ob1s,Ob2s, BcIn, Reph, Rept ) :-
                Ib2s = [[amount(Flow),link(1,_)|RestIb]|RestIb2s],
                NextRe2 := Re2+Flow, NewIbc := Ibc+1,
                Msg = ope(pe(PE),node(Node),send_back(Flow)),
                prolog( ( write(Msg), nl ) ) |
        Ob1s = [[amount(Flow)|RestIb]|NextOb1s],
        node( PE, Node, Ifc, NewIbc, (Re1,St1), (NextRe2,closed),
                        If1s, If2s, _,  _,
                        Ib1s, RestIb2s, NextOb1s, Ob2s, BcIn, Reph, Rept ).
% 2 <-- 2
node( PE, Node, Ifc, Ibc, (Re1,St1), (Re2,St2), If1s,If2s,Of1s,Of2s,
                                Ib1s,Ib2s,Ob1s,Ob2s, BcIn, Reph, Rept ) :-
                Ib2s = [[amount(Flow),link(2,_)|RestIb]|RestIb2s],
                NextRe2 := Re2+Flow, NewIbc := Ibc+1,
                Msg = ope(pe(PE),node(Node),send_back(Flow)),
                prolog( ( write(Msg), nl ) ) |
        Ob2s = [[amount(Flow)|RestIb]|NextOb2s],
        node( PE, Node, Ifc, NewIbc, (Re1,St1), (NextRe2,closed),
                        If1s, If2s, _,  _,
                        Ib1s, RestIb2s, Ob1s, NextOb2s, BcIn, Reph, Rept ).
% 1 --+
%     |
% 1<--+
node( PE, Node, Ifc, Ibc, (Re1,St1), (Re2,St2), If1s,If2s,Of1s,Of2s,
                                Ib1s,Ib2s,Ob1s,Ob2s, BcIn, Reph, Rept ) :-
                If1s = [If|RestIf1s], St1 = closed, St2 = closed,
                If = [amount(Flow)|_], NewIfc := Ifc+1,
                Msg = ope(pe(PE),node(Node),send_back(Flow)),
                prolog( ( write(Msg), nl ) ) |
        Ob1s = [If|NextOb1s],
        node( PE, Node, NewIfc, Ibc, (Re1,St1), (Re2,St2),
                        RestIf1s, If2s, _,      _,
                        Ib1s,     Ib2s, NextOb1s, Ob2s, BcIn, Reph, Rept ).
% 2 --+
%     |
% 2<--+
node( PE, Node, Ifc, Ibc, (Re1,St1), (Re2,St2), If1s,If2s,Of1s,Of2s,
                                Ib1s,Ib2s,Ob1s,Ob2s, BcIn, Reph, Rept ) :-
                If2s = [If|RestIf2s], St1 = closed, St2 = closed,
                If = [amount(Flow)|_], NewIfc := Ifc+1,
                Msg = ope(pe(PE),node(Node),send_back(Flow)),
                prolog( ( write(Msg), nl ) ) |
        Ob2s = [If|NextOb2s],
        node( PE, Node, NewIfc, Ibc, (Re1,St1), (Re2,St2),
                        If1s, RestIf2s, _,    _,
                        Ib1s, Ib2s,     Ob1s, NextOb2s, BcIn, Reph, Rept ).
%
% termination of "node" process
%
%     detecting   "BcIn = end"
```

```
%
node( PE, Node, Ifc, Ibc, (Rel,St1), (Re2,St2), Ifls,If2s,Ofls,Of2s,
                                    Ibls,Ib2s,Obls,Ob2s, BcIn, Reph, Rept ) :-
                BcIn = end,
                Msg = terminate(pe(PE),node(Node)),
                prolog( ( write(Msg), nl ) ) |
        Ope = ope(byFin(Ifc),byBin(Ibc)),
        Status = status((Rel,St1),(Re2,St2)),
        Reph = [result(pe(PE),node(Node),Ope,Status)|Rept].
%
%       exit
%      ------
% continue       ( LimFlow > NextSum )
exit( LimFlow, Sum, Ifs, Irs, BcOut, Prs ) :-
                Ifs = [If|RestIfs], If = [amount(Flow)|_],
                NextSum := Sum+Flow, LimFlow > NextSum |
        Prs = [pass(If)|NextPrs],
        exit( LimFlow, NextSum, RestIfs, Irs, BcOut, NextPrs ).
exit( LimFlow, Sum, Ifs, Irs, BcOut, Prs ) :-
                Irs = [Ir|RestIrs], Ir = [amount(Flow)|_],
                NextSum := Sum+Flow, LimFlow > NextSum |
        Prs = [return(Ir)|NextPrs],
        exit( LimFlow, NextSum, Ifs, RestIrs, BcOut, NextPrs ).
exit( LimFlow, Sum, Ifs, Irs, BcOut, Prs ) :-
                Ifs = [If|RestIfs], If = alloc(_) |
        exit( LimFlow, Sum, RestIfs, Irs, BcOut, Prs ).
exit( LimFlow, Sum, Ifs, Irs, BcOut, Prs ) :-
                Irs = [Ir|RestIfs], Ir = alloc(_) |
        exit( LimFlow, Sum, Ifs, RestIrs, BcOut, Prs ).
%
% terminate      ( LimFlow = NextSum )
exit( LimFlow, Sum, Ifs, Irs, BcOut, Prs ) :-
                Ifs = [If|RestIfs], If = [amount(Flow)|_],
                NextSum := Sum+Flow, LimFlow = NextSum |
        BcOut = end, Prs = [pass(If)].
exit( LimFlow, Sum, Ifs, Irs, BcOut, Prs ) :-
                Irs = [Ir|RestIrs], Ir = [amount(Flow)|_],
                NextSum := Sum+Flow, LimFlow = NextSum |
        BcOut = end, Prs = [return(Ir)].
%
%       outFlow  --  output flow (pass,return) and result on each node
%      ---------
outFlow( [Pr|RestPrs], Reps, SumFlow ) :-
                Pr = pass([amount(Flow)|_]), NextSumFlow := SumFlow + Flow,
                prolog( ( write( Pr ), nl ) ) |
        outFlow( RestPrs, Reps, NextSumFlow ).
outFlow( [Pr|RestPrs], Reps, SumFlow ) :-
                Pr = return(_),
                prolog( ( write( Pr ), nl ) ) |
        outFlow( RestPrs, Reps, SumFlow ).
outFlow( Prs, [Rep|RestReps], SumFlow ) :-
                prolog( ( write( Rep ), nl ) ) |
        outFlow( Prs, RestReps, SumFlow ).
outFlow( [], [], SumFlow ) := Msg = maxFlow(SumFlow),
                prolog( ( nl, write(Msg), nl ) ) |
        true.
%
%       outByPE  --  output result on each PE
%      ---------
outByPE( Reps ) :- Reps = [result(pe(PE),_,_,_)|_] |
        totalByPE( PE, Reps, 0, 0, SumIfc, SumIbc, AllocNodes, NewReps ),
        outMsgByPE( PE, SumIfc, SumIbc, AllocNodes ),
        outByPE( NewReps ).
outByPE( Reps ) :- Reps = [] |
        true.
totalByPE( PE, [Rep|RestReps], SumIfc, SumIbc, FinalSumIfc, FinalSumIbc,
```

(0) Date: 1987-Jul-18, written by E. Sugino
Modified: 29-Jul-87 by S. Takagi
Modified: 12-Aug-87 by E. Sugino

(1) Program name: pascal

(2) Author: E. Sugino, ICOT 4th lab.

(3) Runs on: GHC1 on DEC2065

(4) Description of the problem:

Binomial coefficients are calculated using Pascal's triangle.
Pascal's triangle is:

```
                    1 1
                  1 2 1
                1 3 3 1
              1 4 6 4 1
            ...........
        1   a   b ... b a   1
      1 1+a a+b ... b+a a+1 1
    ........................
```

(5) Algorithm:

The (N)th layer of coefficients is calculated from the (N-1)th layer. When there is no (N-1)th layer, it is calculated from the (N-2)th layer. If the (N-2)th layer is not calculated yet, the (N-3)th layer is calculated. This algorithm is recursively applied. Once the (N)th layer is calculated, it is kept as a process.

(6) Process structure:

Layers of coefficients which were already calculated are kept as processes. These processes are waiting for a message from the same stream. The message requires a certain number of layers. When this number matches the number of process layers, the coefficient value which the process is keeping is sent in reply. When the number is larger than any number of layers which is currently kept, the process that currently has the largest number of layers will start calculating the next layers.

(7) Pragma: None

(8) Program:

Top level loop:
pascal/0, pascal/3, pascal_go, max,
result, result_write, next, nextgo

Starter: pascal/4

Data process: pascal_data

Calculator: new_pascal, make_pascal_data

(9) Source file: US2:<MPSI.BENCH>PASCAL_SUGINO_591.GHC
This document: US2:<MPSI.BENCH>PASCAL_SUGINO_591.DOC

(10) Example:

Compile the program, and try it.

```
?- ghc pascal.
Binomial coefficient using Pascal's triangle
                              87.4.3. by Eiji Sugino
Give me Some integer !
> 10. <CR>                    % You can insert any positive integer.
```

(11) Evaluation data: Not recorded

```
%
%       Binomial coefficient generator using Pascal's triangle
%                               87.4.3.  by Eiji Sugino
%       This Program can be used less than (X + Y)^19
%       so you should give a number that is from 1 to 19 .
%

%%%%%%%%%%%%%%%%%%%%%%%%%
%       PASCAL          %       Top Level procedure
%%%%%%%%%%%%%%%%%%%%%%%%%
pascal :-
        prolog((write('Binomial coefficient using Pascal''s triangle'),nl,
                write('                          87.4.3. by Eiji Sugino'),nl,
                write('Give me Some integer !'),nl)) |
        pascal_data(Stream,1,[1,1],1),
        pascal(Stream,1).

%%%%%%%%%%%%%%%%%       Loop
pascal(Stream,Max) :-
        prolog((prompt(X,'> '),read(N))) | pascal_go(N,Stream,Max).

pascal_go(N,Stream,Max) :-
        prolog((integer(N), statistics(runtime,_))) |
        pascal(N,Result,Stream,S),
        max(N,Max,M),
        result(Result,S,M).
pascal_go(N,Stream,Max) :- prolog((\+(integer(N)))) | pascal(Stream,Max).

max(N,M,X) :- N =< M | X = M.
max(N,M,X) :- N >= M | X = N.

%%%%%%%%%%%%%%%%%       Result writer
result(Result,Stream,N) :- true |
        result_write(Result,End,N),
        next(End,Stream,N).

result_write([],E,N) :- prolog((statistics(runtime,[_,Time]),
                                write('END'),nl,
                                write('Runtime : '),
                                write(Time),write(' msec'),nl,
                                write('Max coefficient : '),
                                write(N),nl)) |
        E = end.
result_write([A|B],E,N) :- wait(A),
                           prolog((write(A),write(', '))) |
        result_write(B,E,N).

%%%%%%%%%%%%%%%%        End of one cycle
next(end,Stream,N) :- prolog((prompt(_,'more ? '),read(X))) |
        nextgo(X,Stream,N).

nextgo(n,Stream,_) :- true| Stream = [].
nextgo('N',Stream,_) :- true | Stream = [].
nextgo(y,Stream,N) :- true | pascal(Stream,N).
nextgo('Y',Stream,N) :- true | pascal(Stream,N).
nextgo(X,Stream,N) :- X \= n, X \= y, X \= 'N', X \= 'Y' |
        next(end,Stream,N).

%%%%%%%%%%%%%%%%%%      Main routine
pascal(N,Result,Stream,S) :- prolog(integer(N)) |
        Stream = [data(N,Result)|S].

%%%%%%%%%%%%%%%%%%      Pascal's triangle data
pascal_data([data(N,D)|S],N,Data,Max) :- true |
        D = Data,
        pascal_data(S,N,Data,Max).
```

```
pascal_data([data(N,D)|S],M,Data,Max) :- N =\= M,N =< Max |
        pascal_data(S,M,Data,Max).
pascal_data([data(N,D)|S],Max,Data,Max) :- Max < N,M1 := Max + 1 |
        new_pascal(M1,N,Data,D,S),
        pascal_data(S,Max,Data,N).
pascal_data([data(N,D)|S],M,Data,Max) :-  Max < N, M < Max |
        pascal_data(S,M,Data,N).
pascal_data([],_,_,_) :- true | true.

%%%%%%%%%%%%%%%%%%%     Data creator
new_pascal(N,N,Data,D,Stream) :- true |
        make_pascal_data(Data,D),
        pascal_data(Stream,N,D,N).
new_pascal(N,M,Data,D,Stream) :- N < M, N1 := N+1 |
        make_pascal_data(Data,Data1),
        pascal_data(Stream,N,Data1,M),
        new_pascal(N1,M,Data1,D,Stream).

make_pascal_data([F1,F2|Data],New) :- Nf2 := F1+F2 |
        New = [F1,Nf2|New1],
        make_pascal_data([F2|Data],New1,[Nf2,F1]).
make_pascal_data([N],New,E) :- true | New = [N].
make_pascal_data([A,A|C],New,E) :- B := A+A | New = [B|E].
make_pascal_data([A,B|C],New,E) :- A > B | New = E.
make_pascal_data([A,B|C],New,E) :- A < B, D := A+B |
        New = [D|New1],
        make_pascal_data([B|C],New1,[D|E]).
```

(0) Date: 1987-Jul-18, written by E. Sugino
Modified: 29-Jul-87 by S. Takagi
Modified: 12-Aug-87 by E. Sugino

(1) Program name: queenf

(2) Author: Original GHC version by K. Kumon, Fujitsu Limited
Rewritten for speed by E. Sugino, ICOT 4th Lab.

(3) Runs on: Pseudo multi-PSI

(4) Description of the problem:

The N-queen problem is familiar to logic programmers.

(5) Algorithm: See the file US2:<MPSI.BENCH>QUEENW.DOC.

(6) Process structure: See the file US2:<MPSI.BENCH>QUEENW.DOC.

(7) Pragma: See the file US2:<MPSI.BENCH>QUEENW.DOC.

(8) Program:

In this program, append is used as little as possible. Two lists which may be consumed at the first and the second arguments of queen/6 are not conjoined.

go is for a 6-PEs demonstration.
queen/3 is for the top level.
queen_n is for putting the first queen.
queen/6 is for finding sub-resolutions.
cl is for checking if a queen can be put on the chessboard.
append is for appending lists.

(9) Source file: US2:<MPSI.BENCH>QUEENF_SUGINO.GHC
This document: US2:<MPSI.BENCH>QUEENF_SUGINO.DOC

(10) Example: See US2:<MPSI.BENCH>QUEENW.DOC.

(11) Evaluation data:

The number of reductions is about 20% fewer than the queenw program on the pseudo MPSI system.

(87.6.30. 6-queen)

```
module queenf.

:- public queen/2,go/1,go/2.

go(6) :- true | queen([1,2,3,4,5,6],_,6).
go(7) :- true | queen([1,2,3,4,5,6,7],_,6).
go(8) :- true | queen([1,2,3,4,5,6,7,8],_,6).

queen(List,Result,N) :- true |
      queen_n(List,[],Result,[],1,N).

queen_n([A|At],B,I,O,PE,N) :-
      P1 := (PE  mod N)+1 |
      queen(At,B,[],[A],I,X),
      alloc(P1)@@queen_n(At,[A|B],X,O,P1,N).
queen_n([],_,I,O,_,_) :- true | I=O.

queen([A|At],B,C,R,I,O) :- true |
      append(B,C,E),
      c1(R,A,1,At,E,R,I,X),
      queen(At,B,[A|C],R,X,O).
queen([],[A|At],E,R,I,O) :- true |
      c1(R,A,1,At,E,R,I,X),
      queen(At,[],[A|E],R,X,O).
queen([],[],[_|_],R,I,O) :- true | I=O.
queen([],[],[],R,I,O) :-
      write(R)
%      true
      | I=[R|O].

c1([R1|Rt],A,D,At,B,R,I,O) :-
      A =\= R1 + D, A =\= R1 - D, D1 := D + 1 |
      c1(Rt,A,D1,At,B,R,I,O).
c1([R1|Rt],A,D,At,B,R,I,O) :-
      A =:= R1 + D | I = O.
c1([R1|Rt],A,D,At,B,R,I,O) :-
      A =:= R1 - D | I = O.
c1([],A,D,At,B,R,I,O) :- true |
      queen(At,B,[],[A|R],I,O).

append([],A,B) :- true | A=B.
append([A|B],C,D) :- true | D = [A|E],append(B,C,E).

end.
```

(0) Date: 1987-Jul-01, written by Reiko Ezaki
Modification: 29-jul-87 by S. Takagi

(1) Program name: RUCS (RUle Check System)

(2) Author: Reiko Ezaki, MELCO-IEL

(3) Runs on: Sequential FGHC System (on PSI)
Pseudo multi-PSI

(4) Description of problem:

This is a verification system for logic circuit design rules.
In this system, two design rules are considered.
1) There should be no loop that consists of only primitive elements.
* Primitive elements: AND, OR, EXOR (exclusive or),
NAND, NOR, EXNOR gates
2) There should be no data transfer between two registers
to which the same phase clock is supplied.
These two rules have been applied to the two logic circuits. One has
15 gates and 4 registers. The other has 31 gates and 11 registers.

(5) Algorithm:

1) The system has a list of all gates.
Each gate is considered as an active process.
Each gate iteratively traces the net list
from its own input pin to output pin,
and make the history of the trace.
If it finds a primary terminal or register during tracing,
there is no loop.
If it finds a gate that is included in the history during tracing,
there is a loop.
2) The system has a list of all registers.
Each register is considered as an active process.
Each register iteratively traces the net list
from its own input pin to output pin,
and makes the history of the trace.
If it finds a primary terminal during tracing, there is no data transfer
between two registers to which the same phase clock is supplied.
If it finds a register that is supplied with the same phase
clock as the start point register, a violation is detected.
The complexity is not considered in this algorithm. All computation
trees are generated and no pruning is done.

(6) Process structure:

The tracing routine for each gate (or register) is regarded as a process.
A d-list is used to gather the solutions.

(7) Pragma:

1) Each gate's trace execution assigns different PEs in turn.
2) Each register's trace execution assigns different PEs in turn.
When there are more gates or registers than PE numbers, the PE
assignment is cyclic.

(8) Program:

The first part of the program consists of:
obtaining the list of the gates (or registers) from the database,
and creating trace processes.
All processes execute each trace and when a violation is found,
it is reported to the window that is exclusively used.

1) gate_loop/0 is the top level.
   gate_loop/6 is each process.
   gate_or_other is the routine to check for violation.
2) same_phase/0 is the top level.
   same_phase/6 is each process.
   register is a routine to check
      whether the element is a register.
   terminal_check is a routine to check
      whether the element is a terminal.
   check_same_phase is a routine to check
      whether the register has the same phase clock
      as the start point register.

(9) Source file: US2:<MPSI.BENCH>RUCS_EZAKI.GHC
    This document: US2:<MPSI.BENCH>RUCS_EZAKI.DOC

(10) Examples:

Compile two modules, scan1 and scan_data.
Invocation:
        scan1@gate_loop(N).   where N is the number of PEs.
        scan1@same_phase(N).   where N is the number of PEs.
A window is created by this program and the solutions are written on it.

(11) Evaluation data:

Sequential FGHC system (on PSI)
                (circuit: 15 gates, 4 registers)
        gate_loop   2.568 sec.
        same_phase   1.617 sec.

Reference data (different algorithm)
                (circuit: 15 gates, 4 registers)
    Prolog version
       gate_loop   2.2 sec.
       same_phase   2.0 sec.
    ESP version
       gate_loop   0.036 sec.
       same_phase   0.024 sec.

(12) Acknowledgment:

The Prolog version of this program was written by Yasushi Koseko, MELCO ASIC Design Engineering Center.

```
module scan1.

:- public same_phase/0, search_clock_terminal/1,
          clock_register/2, clock_connect/2, clock_terminal/1.

same_phase :- true |
    search_clock_terminal(Clock),
    connect_check(Clock, Results),
    print_result(Results) .

search_clock_terminal(C) :- true |
    clock_terminal(C) .

clock_terminal(C) :- true | C = [sc1,sc2] .

connect_check([Clock|CRest], Results) :- true |
    clock_connect(Clock, Register),
    connect_mutual(Register, Result, Register),
    Results = [Result|RRest],
    connect_check(CRest, RRest) .
connect_check([], R) :- true | R = success .

clock_connect(sc1, C) :- true | C = [ls11, ls14] .
clock_connect(sc2, C) :- true | C = [ls12, ls13] .

connect_mutual([Register|RRest], Results, Register1) :- true |
    connect_element(Register, in, [Element]),
    register(Element, R),
    check_same_phase(R, Element, Result, Register1).
connect_mutual([], _, _) :- true | true .

connect_mutual1([Element|RELe], Results, Register1) :- true |
    register(Element, R),
    check_same_phase(R, Element, Results, Register1).
connect_mutual1([], _, _) :- true | true .

check_same_phase(success, Register, Result, Register1) :- true |
    belong(Register, Register1, R),
    check_belong(R, Register, Result).
check_same_phase(fail, Element, Result, Register1) :- true |
    connect_element(Element, in, CEle),
    connect_mutual1(CEle, Result, Register1) .

register(Element, R) :- true |
    register_list(RList),
    belong(Element, RList, R) .

register_list(RList) :- true | RList = [ls11, ls12, ls13, ls14] .

belong(A, [B|C], R) :- A=B | R = success.
belong(A, [B|C], R) :- A\=B | belong(A, C, R) .
belong(A, [], R) :- otherwise | R = fail .

check_belong(success, Register, Result) :- true |
    Result = [Register] .
check_belong(fail, Register, Result) :- true |
    Result = [] .

print_result(R) :- true | true.

terminal(Element, R) :- true |
    terminal_list(TList),
    belong(Element, TList, R) .

terminal_list(List) :-
    true |
```

```
    List = [vcc, di1, di2, di3, di4, sc1, sc2, do1, do2, gnd]

connect_element(lsl1, in, E) :- true | E = [g2].  % sc1
connect_element(lsl1, out1, E) :- true | E = [g5].
connect_element(lsl1, out0, E) :- true | E = [g7, g8].
connect_element(lsl2, in, E) :- true | E = [g4].  % sc2
connect_element(lsl2, out1, E) :- true | E = [g8].
connect_element(lsl2, out0, E) :- true | E = [g5, g7].
connect_element(lsl3, in, E) :- true | E = [g6].  % sc2
connect_element(lsl3, out1, E) :- true | E = [g10].
connect_element(lsl3, out0, E) :- true | E = [].
connect_element(lsl4, in, E) :- true | E = [g9].  % sc1
connect_element(lsl4, out1, E) :- true | E = [g12].
connect_element(lsl4, out0, E) :- true | E = [].
connect_element(g1, in, E) :- true | E = [].  % vcc,di1
connect_element(g1, out1, E) :- true | E = [g2].
connect_element(g2, in, E) :- true | E = [g1].  % constant,gnd
connect_element(g2, out1, E) :- true | E = [lsl1].
connect_element(g3, in, E) :- true | E = [].  % gnd,di3
connect_element(g3, out1, E) :- true | E = [g4].
connect_element(g4, in, E) :- true | E = [g3, g15].
connect_element(g4, out1, E) :- true | E = [lsl2].
connect_element(g5, in, E) :- true | E = [lsl1, lsl2].
connect_element(g5, out1, E) :- true | E = [g6].
connect_element(g6, in, E) :- true | E = [constant, g5].
connect_element(g6, out1, E) :- true | E = [lsl3].
connect_element(g7, in, E) :- true | E = [lsl1, lsl1].
connect_element(g7, out1, E) :- true | E = [g16].
connect_element(g8, in, E) :- true | E = [lsl1, lsl2].
connect_element(g8, out1, E) :- true | E = [g9].
connect_element(g9, in, E) :- true | E = [g8, g16].
connect_element(g9, out1, E) :- true | E = [lsl4].
connect_element(g10, in, E) :- true | E = [lsl3].
connect_element(g10, out1, E) :- true | E = [].  % do1
connect_element(g12, in, E) :- true | E = [lsl4].
connect_element(g12, out1, E) :- true | E = [].  % do2
connect_element(g13, in, E) :- true | E = [].  % di4
connect_element(g13, out1, E) :- true | E = [g15].
connect_element(g14, in, E) :- true | E = [].  % di4
connect_element(g14, out1, E) :- true | E = [g16].
connect_element(g15, in, E) :- true | E = [g13].  % di2
connect_element(g15, out1, E) :- true | E = [g4].
connect_element(g16, in, E) :- true | E = [g7, g14].
connect_element(g16, out1, E) :- true | E = [g9].

end.
```

(0) Date: 1987-Jul-3, written by M. Koshimura
Modified: 30-jul-87 by S. Takagi

(1) Program name: tep

(2) Author: M. Koshimura, ICOT 4th Lab.

(3) Runs on: Ueda's GHC compiler V1.7 on DEC2060

(4) Description of the problem:

A tiny propositional calculus solver by LK (Logistischer Kalkul).

A problem is represented by a sequent. A sequent consists of 2 parts.
One is the left hand side (LHS). The other is the right hand side (RHS).
For example,

```
        a&(a->b) --> b
```

means that if a and a implies b is true, then b is true.
Here a&(a->b) is the LHS and b is the RHS.
In general, both the LHS and RHS consist of a sequence of formulas.

Let ai, bi be a formula, then a1,...,an --> b1,...,bm is a sequent.
Here, the LHS is a1,...,an , and the RHS is b1,...,bm.
This means that (a1 and ... and an) implies (b1 or ... or bm).
The sequent a1,...,an --> b1,...,bm is called an axiom
if there exists i,j such that ai = bj.

The following logical symbols are supported.

~	Strong (classical) negation
\/	Disjunction
/\	Conjunction
->	Implication
<->	Equivalence

This system can prove predicates such as:

```
        --> (~ ~a) <-> a
        --> (b -> a) <-> (~a -> ~b)
```

(5) Algorithm:

The program keeps two different lists, which present the LHS and RHS,
i.e. a problem is represented by these two lists.

First,
Check the intersection of the LHS and RHS.
If the intersection is not empty, this is an axiom, and returns true.
If the intersection is empty, continue to the next step.
Second,
Try to decompose the problem to subproblem(s), according to the
decomposition rules. The subproblems have an &-relation.
If succeeds, recursively apply the algorithm to the subproblem(s).
If fails, it is not provable in the algorithm.

(6) Process structure:

Run in one process when crunching the term. When two or more
subproblems are obtained, these subproblems are processed by newly
created processes. The original process merges the results of the
forked processes.

(7) Pragma: Not supported

(8) Program:

The program consists of the decompose part and fork part.
tep/2 is the top level.

The decompose part checks whether the problem is an axiom,
using intersect/3.

Next, it tries to decompose the problem, using a decompose predicate,
such as lnot/7 or rnot.

l means the LHS, and r means the RHS. '***' of l'***', r'***' means
a propositional symbol, which l'***' (r'***') tries to decompose.

If two or more decomposing rules can be applied to the problem, it is
necessary to choose one. This is done by decompose_decide1/3.

The fork part forks the processes which correspond to each decomposed
subproblem(s). This part is included in tep1/2.

The rest is a pretty print routine for the proof tree.
If the procedure fails to prove the problem,
the printed proof tree includes the atom, fail.

(9) Source file: US2:<MPSI.BENCH>TEP_KOSHIMURA.GHC
This document: US2:<MPSI.BENCH>TEP_KOSHIMURA.DOC

(10) Examples:

Invocation:

tep_go(N). where N is the trial problem number.
N = 1 to 8 are effective.
The more tries the more complicated sequent.

test number 1: --> (~ ~a) <-> a
test number 2: --> ~ (~a /\ a)
test number 3: --> ~ (b /\ a) <-> (~b \/ ~a)
test number 4: --> ~ (b \/ a) <-> (~b /\ ~a)
test number 5: --> (b -> a) <-> (~a -> ~b)
test number 6: --> (~ (b -> a)) <-> (b /\ ~a)
test number 7: --> (~a \/ b) <-> (a -> b)
test number 8: --> ((a<->b) <-> c) <-> (a <-> (b<->c))

(11) Evaluation data: Not recorded

```
:- op(200, fy,[~]).
:- op(500,yfx,[\/]).
:- op(500,yfx,[/\]).
:- op(1050,xfy,[->]).
:- op(1060,xfx,[<->]).
%************************************************
%
% tep(+Sequent, -Proof)
%
%************************************************
tep(Seq, Proof) :- true |
        decompose(Seq, Inf,UpSeqs),
        tep(Inf, Seq, UpSeqs, Proof).
%
% tep(+Inf, +Sequent, +UpperSequents, -Proof)
%
tep(axiom, Seq, _, Proof) :- true | Proof = [axiom, Seq].
tep(fail, Seq, _, Proof) :- true | Proof = [fail, Seq].
tep(Inf, Seq, UpSeqs, Proof) :- Inf \= axiom, Inf \= fail |
        Proof = [Inf, Seq, Proof1],
        tep1(UpSeqs, Proof1).

%
% tep1(+UpperSequents, -Proof)
%
tep1([Seq|Seqs], Proof) :- true |
        Proof = [SProof|Proof1],
        tep(Seq, SProof),
        tep1(Seqs, Proof1).
tep1([], Proof) :- true | Proof = [].

%
% lnot(+Left,+Right, +LeftHead,-LeftTail, +LeftHead,-LeftTail, -UpperSequents)
%
lnot([(~A)|Left],Right, LHead,TLeft, _,_, Ans) :- true |
        TLeft = Left,
        Ans = [[LHead,[A|Right]]].
lnot([Other|Left],Right, LHead,TLeft, _,_, Ans) :- Other \= (~A) |
        TLeft = [Other|CLHead],
        lnot(Left,Right, LHead,CLHead, _,_, Ans).

%
% rnot(+Left,+Right, +RightHead,-RightTail, +RightHead,-RightTail,
%                         -UpperSequents)
%
rnot(Left,[(~A)|Right], RHead,TRight, _,_, Ans) :- true |
        TRight = Right,
        Ans = [[[A|Left],RHead]].
rnot(Left,[Other|Right], RHead,TRight, _,_, Ans) :- Other \= (~A) |
        TRight = [Other|CRHead],
        rnot(Left,Right, RHead,CRHead, _,_, Ans).

land([(A/\B)|Left],Right, LHead,TLeft, _,_, Ans) :- true |
        TLeft = [A,B|Left],
        Ans = [[LHead,Right]].
land([Other|Left],Right, LHead,TLeft, _,_, Ans) :- Other \= (A/\B) |
        TLeft = [Other|CLHead],
        land(Left,Right, LHead,CLHead, _,_, Ans).

rand(Left,[(A/\B)|Right], RHead,TRight, RHead1,TRight1, Ans) :- true |
        TRight = [A|Right],
        TRight1 = [B|Right],
        Ans = [[Left,RHead],[Left,RHead1]].
rand(Left,[Other|Right], RHead,TRight, RHead1,TRight1, Ans) :-
        Other \= (A/\B) |
        TRight = [Other|CRHead],
```

```
        TRight1 = [Other|CRHead1],
        rand(Left,Right, RHead,CRHead, RHead1,CRHead1, Ans).

lor([(A\/B)|Left],Right, LHead,TLeft, LHead1,TLeft1, Ans) :- true |
        TLeft = [A|Left],
        TLeft1 = [B|Left],
        Ans = [[LHead,Right],[LHead1,Right]].
lor([Other|Left],Right, LHead,TLeft, LHead1,TLeft1, Ans) :-  Other \= (A\/B) |
        TLeft = [Other|CLHead],
        TLeft1 = [Other|CLHead1],
        lor(Left,Right, LHead,CLHead, LHead1,CLHead1, Ans).

ror(Left,[(A\/B)|Right], RHead,TRight, _,_, Ans) :- true |
        TRight = [A,B|Right],
        Ans = [[Left,RHead]].
ror(Left,[Other|Right], RHead,TRight, _,_, Ans) :- Other \= (A\/B) |
        TRight = [Other|CRHead],
        ror(Left,Right, RHead,CRHead, _,_, Ans).

limpl([(A->B)|Left],Right, LHead,TLeft, LHead1,TLeft1, Ans) :- true |
        TLeft = Left,
        TLeft1 = [B|Left],
        Ans = [[LHead,[A|Right]],[LHead1,Right]].
limpl([Other|Left],Right, LHead,TLeft, LHead1,TLeft1, Ans) :- Other \= (A->B) |
        TLeft = [Other|CLHead],
        TLeft1 = [Other|CLHead1],
        limpl(Left,Right, LHead,CLHead, LHead1,CLHead1, Ans).

rimpl(Left,[(A->B)|Right], RHead,TRight, _,_, Ans) :- true |
        TRight = [B|Right],
        Ans = [[[A|Left],RHead]].
rimpl(Left,[Other|Right], RHead,TRight, _,_, Ans) :- Other \= (A->B) |
        TRight = [Other|CRHead],
        rimpl(Left,Right, RHead,CRHead, _,_, Ans).

lequiv([(A<->B)|Left],Right, LHead,TLeft, LHead1,TLeft1, Ans) :- true |
        TLeft = [A,B|Left],
        TLeft1 = Left,
        Ans = [[LHead,Right],[LHead1,[A,B|Right]]].
lequiv([Other|Left],Right, LHead,TLeft, LHead1,TLeft1, Ans) :-
        Other \= (A<->B) |
        TLeft = [Other|CLHead],
        TLeft1 = [Other|CLHead1],
        lequiv(Left,Right, LHead,CLHead, LHead1,CLHead1, Ans).

requiv(Left,[(A<->B)|Right], RHead,TRight, RHead1,TRight1, Ans) :- true |
        TRight = [B|Right],
        TRight1 = [A|Right],
        Ans = [[[A|Left],RHead],[[B|Left],RHead1]].
requiv(Left,[Other|Right], RHead,TRight, RHead1,TRight1, Ans) :-
        Other \= (A<->B) |
        TRight = [Other|CRHead],
        TRight1 = [Other|CRHead1],
        requiv(Left,Right, RHead,CRHead, RHead1,CRHead1, Ans).

%
% decompose(+Sequent, -Inference,-UpperSequent)
%
decompose([Left,Right], Inf,UpSeqs) :- true |
        intersect(Left,Right, Ans),
        decompose_decide(Ans, [Left,Right], Inf,UpSeqs).

%
% decompose_decide(+Result, +Sequent, -Inference,-UpperSequent)
%
decompose_decide(true, _, Inf,UpSeqs) :- true |
```

```
        Inf = axiom,
        UpSeqs = [].
decompose_decide(fail, [Left,Right], Inf,UpSeqs) :- true |
        decompose_decide1(Left,Right, Ans),
        decompose1(Ans, Left,Right, Inf,UpSeqs).

%
% decompose_decide1(+Left,+Right, -Answer)
%
decompose_decide1(Left,Right, Ans) :- true |
        decompose_decide1_left(Left, Ans1),
        decompose_decide1_right(Right, Ans2),
        check_ans(Ans1,Ans2, Ans).

%
% decompose_decide1_left(+Left, -Answer)
%
decompose_decide1_left([], Ans) :- true | Ans = [].
decompose_decide1_left([(~A)|Left], Ans) :- true |
        Ans = [(10,lnot)|Ans1],
        decompose_decide1_left(Left, Ans1).
decompose_decide1_left([(A/\B)|Left], Ans) :- true |
        Ans = [(10,land)|Ans1],
        decompose_decide1_left(Left, Ans1).
decompose_decide1_left([(A\/B)|Left], Ans) :- true |
        Ans = [(20,lor)|Ans1],
        decompose_decide1_left(Left, Ans1).
decompose_decide1_left([(A->B)|Left], Ans) :- true |
        Ans = [(20,limpl)|Ans1],
        decompose_decide1_left(Left, Ans1).
decompose_decide1_left([(A<->B)|Left], Ans) :- true |
        Ans = [(20,lequiv)|Ans1],
        decompose_decide1_left(Left, Ans1).
decompose_decide1_left([A|Left], Ans) :-
        A \= (~B), A \= (B/\C), A \= (B\/C), A \= (B->C), A \= (B<->C) |
        decompose_decide1_left(Left, Ans).

%
% decompose_decide1_right(+Right, -Answer)
%
decompose_decide1_right([], Ans) :- true | Ans = [].
decompose_decide1_right([(~A)|Right], Ans) :- true |
        Ans = [(10,rnot)|Ans1],
        decompose_decide1_right(Right, Ans1).
decompose_decide1_right([(A/\B)|Right], Ans) :- true |
        Ans = [(20,rand)|Ans1],
        decompose_decide1_right(Right, Ans1).
decompose_decide1_right([(A\/B)|Right], Ans) :- true |
        Ans = [(10,ror)|Ans1],
        decompose_decide1_right(Right, Ans1).
decompose_decide1_right([(A->B)|Right], Ans) :- true |
        Ans = [(10,rimpl)|Ans1],
        decompose_decide1_right(Right, Ans1).
decompose_decide1_right([(A<->B)|Right], Ans) :- true |
        Ans = [(20,requiv)|Ans1],
        decompose_decide1_right(Right, Ans1).
decompose_decide1_right([A|Right], Ans) :-
        A \= (~B), A \= (B/\C), A \= (B\/C), A \= (B->C), A \= (B<->C) |
        decompose_decide1_right(Right, Ans).

%
% check_ans(+Answer,+Answer, -Answer)
%
check_ans(Ans1,Ans2, Ans) :- true |
        merge(Ans1,Ans2, Ans3),
        check_ans1(Ans3, cand(1000,fail), Ans).
```

```
%
% check_ansl(+Answer, +Candidate, -Answer)
%
check_ansl([(N,OP)|Ansl], cand(CN,COP), Ans) :- N < CN |
        check_ansl(Ansl, cand(N,OP), Ans).
check_ansl([(N,OP)|Ansl], cand(CN,COP), Ans) :- N >= CN |
        check_ansl(Ansl, cand(CN,COP), Ans).
check_ansl([], cand(CN,COP), Ans) :- true | Ans = COP.

%
% decomposel(+Inference, +Left,+Right, -Inference,-UpperSequents)
%
decomposel(lnot, Left,Right, Inf,UpSeqs) :- true |
        Inf = lnot,
        lnot(Left,Right, Left1,Left1, Left2,Left2, UpSeqs).
decomposel(rnot, Left,Right, Inf,UpSeqs) :- true |
        Inf = rnot,
        rnot(Left,Right, Right1,Right1, Right2,Right2, UpSeqs).
decomposel(land, Left,Right, Inf,UpSeqs) :- true |
        Inf = land,
        land(Left,Right, Left1,Left1, Left2,Left2, UpSeqs).
decomposel(ror, Left,Right, Inf,UpSeqs) :- true |
        Inf = ror,
        ror(Left,Right, Right1,Right1, Right2,Right2, UpSeqs).
decomposel(rimpl, Left,Right, Inf,UpSeqs) :- true |
        Inf = rimpl,
        rimpl(Left,Right, Right1,Right1, Right2,Right2, UpSeqs).
decomposel(rand, Left,Right, Inf,UpSeqs) :- true |
        Inf = rand,
        rand(Left,Right, Right1,Right1, Right2,Right2, UpSeqs).
decomposel(lor, Left,Right, Inf,UpSeqs) :- true |
        Inf = lor,
        lor(Left,Right, Left1,Left1, Left2,Left2, UpSeqs).
decomposel(limpl, Left,Right, Inf,UpSeqs) :- true |
        Inf = limpl,
        limpl(Left,Right, Left1,Left1, Left2,Left2, UpSeqs).
decomposel(lequiv, Left,Right, Inf,UpSeqs) :- true |
        Inf = lequiv,
        lequiv(Left,Right, Left1,Left1, Left2,Left2, UpSeqs).
decomposel(requiv, Left,Right, Inf,UpSeqs) :- true |
        Inf = requiv,
        requiv(Left,Right, Right1,Right1, Right2,Right2, UpSeqs).
decomposel(fail, Left,Right, Inf,UpSeqs) :- true | Inf = fail.

%%%%%%%%%%%%%%%%%%%%
%
% Pretty Print
%
%%%%%%%%%%%%%%%%%%%%
pp([axiom,Sequent],Tab,HOut-TOut) :- true |
        HOut = [tab(Tab), write(Sequent), nl, tab(Tab), write(axiom), nl|TOut].
pp([fail,Sequent],Tab,HOut-TOut) :- true |
        HOut = [tab(Tab), write(Sequent), nl, tab(Tab), write(fail), nl |TOut].
pp([Inf,Sequent,Proof],Tab,HOut-TOut) :-
        Inf \= axiom, Inf \= fail |
        HOut = [tab(Tab), write(Sequent), nl, tab(Tab), write(Inf), nl| HOut1],
        Tab1 := Tab+5,
        pp_subproof(Proof,Tab1,HOut1-TOut).

pp_subproof([Proof|RestProof],Tab,HOut-TOut) :- true |
        pp(Proof,Tab,HOut-HOut1),
        pp_subproof(RestProof,Tab,HOut1-TOut).
pp_subproof([],_,HOut-TOut) :- true | HOut = TOut.

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

```
%
% Library
%
%%%%%%%%%%%%%%%%%%%%%
member(X,[X|Xs],Ans) :- true | Ans = true.
member(A,[X|Xs],Ans) :- A \= X | member(A,Xs,Ans).
member(_,[],Ans) :- true | Ans = fail.

merge([A|X], Y, Z) :- true | Z = [A|Zr], merge(Y, X, Zr).
merge(X, [A|Y], Z) :- true | Z = [A|Zr], merge(Y, X, Zr).
merge([], X, Z) :- true | Z = X.
merge(X, [], Z) :- true | Z = X.

intersect([X|Xs],Ys,Ans) :- true |
        member(X,Ys,Ans1),
        intersect_decide(Ans1,Xs,Ys,Ans).
intersect([],_,Ans) :- true | Ans = fail.

intersect_decide(true,_,_,Ans) :- true | Ans = true.
intersect_decide(fail,Xs,Ys,Ans) :- true | intersect(Xs,Ys,Ans).


tep(Sequent) :- true |
        tep(Sequent,Proof),
        pp(Proof,0,NewProof-[]),
        outstream(NewProof).

tep_go(1) :- true |
        tep([[],
             [(( ~ ~ a) <-> a)]]).
tep_go(2) :- true |
        tep([[],
             [~ (~ a /\ a)]]).
tep_go(3) :- true |
        tep([[],
             [(~ (b/\a) <-> (~ b \/ ~ a))]]).
tep_go(4) :- true |
        tep([[],
             [(~ (b\/a) <-> (~ b /\ ~ a))]]).
tep_go(5) :- true |
        tep([[],
             [((b -> a) <-> (~ a -> ~ b))]]).
tep_go(6) :- true |
        tep([[],
             [(~ (b -> a) <-> (b /\ ~a))]]).
tep_go(7) :- true |
        tep([[],
             [((~ a \/ b) <-> (a -> b))]]).
tep_go(8) :- true |
        tep([[],
             [(((a<->b) <-> c) <-> (a <-> (b<->c)))]]).
```

(0) Date: 1987-Jul-29, written by S. Takagi

(1) Program name: trsl

(2) Author: Original Prolog version by Ko Sakai, ICOT 1st Lab.
Rewritten for GHC by S. Takagi, ICOT 4th Lab.

(3) Runs on: Pseudo multi-PSI

(4) Description of the problem:

A small predicate logic solver using the term rewriting system.
Simple classic predicate logic is handled.
The following operators are supported.

~	Strong (classical) negation
^	Heyting negation (not implemented)
/	Necessity (not implemented)
\	Possibility (not implemented)
\/	Disjunction
/\	Conjunction
->	Implication
->>	Strict implication
<->	Equivalence

This system can prove predicates such as

```
(~ ~a) <-> a
(b -> a) <-> (~a -> ~b)
```

(5) Algorithm:

Not understood well.
Program keeps two different lists, say P and N.
They are empty at the beginning.
The predicate term that should be proved is processed as follows:
Crunch the term if it is a compound term.
Try to prove each element.
Merge the result depending on the and/or combination.
If the term is atomic, look for it in the P list.
If found, it is proved.
If not found, add it to the N list.
The positions of P and N are exchanged if ~ is specified.

(6) Process structure:

Run in one process when crunching the term.
When one of disjunction, conjunction, implication, strict implication, and equivalence is encountered, its left part proceeds in the same process, and its right part is processed by a newly created process.
The results of the two processes are merged in the first process.

(7) Pragma:

Use next_pe to determine where to throw goals.
The current definition is the next PE number.

(8) Program:

The main parts use almost the same name: classic, classic1, classic2, and so on. Other routines are added for the GHC interface.

go/1 is for a six-PE demonstration.
test/2 is the main program to specify what term should be tried.
classic/3 is the top level.
classic/6 is for proving a positive result.
classic/5 is for proving a negative result.
classic1 is for the subroutine for classic/6.
classic2 is for the subroutine for classic/5.

(9) Source file: US2:<MPSI.BENCH>TRS1_TAKAGI.GHC
This document: US2:<MPSI.BENCH>TRS1_TAKAGI.DOC

(10) Examples:

Invocation:

```
trs1@go(N).      where N is the trial problem number.
                 N = 1 to 8 are effective.
                 The more tries the more complicated the terms.

test number 1: (~ ~a) <-> a
test number 2: ~ (~a /\ a)
test number 3: ~ (b /\ a) <-> (~b \/ ~a)
test number 4: ~ (b \/ a) <-> (~b /\ ~a)
test number 5: (b -> a) <-> (~a -> ~b)
test number 6: (~ (b -> a)) <-> (b /\ ~a)
test number 7: (~a \/ b) <-> (a -> b)
test number 8: ((a<->b) <-> c) <-> (a <-> (b<->c))
```

On running the program, each processor displays the term that it is currently processing. This looks like:

```
>(a,b)          means a->b is being processed
a               means a is being processed
```

Finally, the starting processor reports "proved".

If it says "fail(..., ...)", there is a bug in the processor, not in the program. This example includes only successful cases.

The goal distribution strategy for current examples is poor.
For example, even test number 8 uses up to five PEs.

(11) Evaluation data: Not recorded

```
% <MPSI.BENCH>TRS1_TAKAGI.GHC.3, 29-Jun-87 13:35:48, Edit by TAKAGI

module trs1.

create_io_stream(_,X) :- true | outstream(X).
outstream([]) :- true | true.
outstream([X|Y]) :- true | outstream(X), outstream(Y).
outstream(nl) :- true | true.
outstream(write(X)) :- write(X) | true.

:- op(200,fy,(~)).      % Strong (Classical) Negation
:- op(200,fy,(^)).      % Heyting Negation
:- op(400,fy,(/)).      % necessity
:- op(400,fy,(\)).      % possibility
:- op(500,yfx,(\/)).    % Disjunction
:- op(500,yfx,(/\)).    % Conjunction
:- op(1040,xfy,(->)).   % Implication
:- op(1040,xfy,(->>)).  % Strict Implication
:- op(1040,xfy,(<->)).  % Equivalence

% test data

:-public go/1.
% :- mode go(+).
go(N) :- true | test(N,1).

:- public test/2.
% :- mode test(+,+).
test(1,PE) :- true | doclassic((( ~ ~a) <-> a),PE).
test(2,PE) :- true | doclassic(~ (~a /\ a),PE).
test(3,PE) :- true | doclassic((~ (b /\ a) <-> (~b \/ ~a)),PE).
test(4,PE) :- true | doclassic((~ (b \/ a) <-> (~b /\ ~a)),PE).
test(5,PE) :- true | doclassic(((b -> a) <-> (~a -> ~b)),PE).
test(6,PE) :- true | doclassic(((~ (b -> a)) <-> (b /\ ~a)),PE).
test(7,PE) :- true | doclassic(((~a \/ b) <-> (a -> b)),PE).
test(8,PE) :- true | doclassic((((a<->b) <-> c) <-> (a <-> (b<->c))),PE).

% :- mode next_pe(+,-).
next_pe(6,X) :- true | X=1.
next_pe(5,X) :- true | X=6.
next_pe(4,X) :- true | X=5.
next_pe(3,X) :- true | X=4.
next_pe(2,X) :- true | X=3.
next_pe(1,X) :- true | X=2.

:- public doclassic/2.
% :- mode doclassic(+,+).
doclassic(X,PE) :- true |
    classic(X,Result,PE),
    print_result(Result,O),
    create_io_stream([],[write(' test for : '), write(X), nl|O]).

% :- mode print_result(+,-).
print_result(proved,O) :- true | O=[nl, write('proved'), nl].
print_result(fail(SP,SN),O) :- true |
    O=[write('false: '), write(SP), nl, write(' true: '), write(SN), nl].
```

```
:- public classic/3.
% :- mode classic(+,-,+).
classic(A,Result,PE) :- true | classic([A],[],[],[],Result,PE).

% :- mode classic(+,+,+,+,-,+).
classic([X|Y],N,SP,SN,Result,PE) :- true | classic1(X,Y,N,SP,SN,Result,PE).
classic([],N,SP,SN,Result,PE) :- true | classic(N,SP,SN,Result,PE).

% :- mode classic1(+,+,+,+,+,-,+).
classic1((~A),P,N,SP,SN,Result,PE) :- write((~A)) |
    classic(P,[A|N],SP,SN,Result,PE).
classic1((A\/B),P,N,SP,SN,Result,PE) :- write((A\/B)) |
    classic([A,B|P],N,SP,SN,Result,PE).
classic1((A/\B),P,N,SP,SN,Result,PE) :- write((A/\B)) |
    alloc(PE1)@@classic([A|P],N,SP,SN,Result0,PE1),
    next_pe(PE,PE1), classic([B|P],N,SP,SN,Result1,PE),
    check_result(Result0,Result1,Result).
classic1((A->B),P,N,SP,SN,Result,PE) :- write((A->B)) |
    classic([B|P],[A|N],SP,SN,Result,PE).
classic1((A<->B),P,N,SP,SN,Result,PE) :- write((A<->B)) |
    alloc(PE1)@@classic([A|P],[B|N],SP,SN,Result0,PE1),
    next_pe(PE,PE1), classic([B|P],[A|N],SP,SN,Result1,PE),
    check_result(Result0,Result1,Result).
classic1(A,P,N,SP,SN,Result,PE) :- member(A,SN), write(A) |
    Result=proved.      % A is atomic and in Left.
classic1(A,P,N,SP,SN,Result,PE) :- otherwise, write(A) |
    classic(P,N,[A|SP],SN,Result,PE).

% :- mode classic(+,+,+,-,+).
classic([X|Y],SP,SN,Result,PE) :- true | classic2(X,Y,SP,SN,Result,PE).
classic([],SP,SN,Result,PE) :- true | Result=fail(SP,SN).

% :- mode classic2(+,+,+,+,-,+).
classic2((~A),N,SP,SN,Result,PE) :- write((~A)) |
    classic([A],N,SP,SN,Result,PE).
classic2((A\/B),N,SP,SN,Result,PE) :- write((A\/B)) |
    alloc(PE1)@@classic([A|N],SP,SN,Result0,PE1),
    next_pe(PE,PE1), classic([B|N],SP,SN,Result1,PE),
    check_result(Result0,Result1,Result).
classic2((A/\B),N,SP,SN,Result,PE) :- write((A/\B)) |
    classic([A,B|N],SP,SN,Result,PE).
classic2((A->B),N,SP,SN,Result,PE) :- write((A->B)) |
    alloc(PE1)@@classic([B|N],SP,SN,Result0,PE1),
    next_pe(PE,PE1), classic([A],N,SP,SN,Result1,PE),
    check_result(Result0,Result1,Result).
classic2((A<->B),N,SP,SN,Result,PE) :- write((A->B)) |
    alloc(PE1)@@classic([A,B|N],SP,SN,Result0,PE1),
    next_pe(PE,PE1), classic([A,B],N,SP,SN,Result1,PE),
    check_result(Result0,Result1,Result).
classic2(A,N,SP,SN,Result,PE) :- member(A,SP), write(A) |
    Result=proved.
classic2(A,N,SP,SN,Result,PE) :- otherwise, write(A) |
    classic(N,SP,[A|SN],Result,PE).

% :- mode check_result(+,+,-).
check_result(proved,X,Result) :- true | Result=X.
check_result(X,proved,Result) :- true | Result=X.
check_result(fail(SPX,SNX),fail(SPY,SNY),Result) :- true |
    Result=fail((SPX,SPY),(SNX,SNY)).

end.
```