Contents

1 Lab 1 **2**
1.1 Exercise 1.1 . . . 2
1.1.1 Queries . . . 3
1.1.2 Questions . . . 4
1.2 Exercise 1.2 . . . 5
1.2.1 Queries . . . 5

2 Lab 2 **6**
2.1 Exercise 2.1 . . . 6
2.1.1 Queries . . . 7
2.2 Exercise 2.2 . . . 7
2.2.1 Questions . . . 8
2.3 Exercise 2.3 . . . 12
2.3.1 Queries . . . 13
2.4 Exercise 2.4 . . . 14
2.4.1 Queries . . . 14

3 Lab 3 **16**
3.1 Exercise 1 . . . 16
3.1.1 Queries . . . 17

4 Lab 4 **18**
4.1 State Space and Print . . . 18
4.2 Exercise 1 . . . 20
4.2.1 Queries . . . 20
4.3 Exercise 2 . . . 22
4.3.1 Queries . . . 22
4.3.2 Questions . . . 23

5 Lab 5 **24**
5.1 Exercise 1 . . . 24
5.1.1 Queries . . . 26

1 Lab 1

1.1 Exercise 1.1

```
1  beautiful(ulrika).
2  beautiful(nisse).
3  beautiful(peter).
4
5  kind(bosse).
6
7  strong(bosse).
8  strong(bettan).
9
10 rich(nisse).
11 rich(bettan).
12
13 male(nisse).
14 male(peter).
15 male(bosse).
16
17 female(ulrika).
18 female(bettan).
19
20 likes(X, Y) :-
21     male(X),
22     female(Y),
23     beautiful(Y).
24 likes(ulrika, X) :-
25     male(X),
26     rich(X),
27     kind(X),
28     likes(X, ulrika).
29 likes(ulrika, X) :-
30     male(X),
31     beautiful(X),
32     strong(X),
33     likes(X, ulrika).
34 likes(nisse, X) :-
35     female(X),
36     likes(X, nisse).
37
38 happy(X) :-
39     rich(X).
40 happy(X) :-
41     male(X),
42     female(Y),
```

```
43      likes(X, Y),
44      likes(Y, X).
45  happy(X) :-
46      female(X),
47      male(Y),
48      likes(X, Y),
49      likes(Y, X).
```

1.1.1 Queries

Who is happy?

```
1  | ?- happy(X).
2  X = nisse ? ;
3  X = bettan ? ;
4  no
```

Who likes who?

```
1  | ?- likes(X, Y).
2  X = nisse,
3  Y = ulrika ? ;
4  X = peter,
5  Y = ulrika ? ;
6  X = bosse,
7  Y = ulrika ? ;
8  no
```

How many persons like Ulrika?

```
1  | ?- findall(X, likes(X, ulrika), Z), length(Z, N).
2  Z = [nisse,peter,bosse],
3  N = 3 ? ;
4  no
```

1.1.2 Questions

- In what way should you arrange the clauses (rules and facts) in the program?
  - Prolog will execute rules or facts with same name from top to down which means placing a fact before a rule is more efficient. This is because the fact gives a truth value while the truth value of a rule depends on other rules or facts.

    Readability is increased by grouping same named rules and facts together.
- In what way should the premises of rules be arranged?
  - Prolog will execute the premises of a rule from left to right. This means that facts should precede rules because verifying a rule would lead to a new node in the SLD-tree even if some fact of the group of premises is false. If a fact is refuted the branch fails and processing stops. Same reasoning could be applied to the ordering of rules, computationally simple rules should generally precede computationally complex ones.

1.2 Exercise 1.2

```
edge(a, b).
edge(a, c).
edge(b, c).
edge(c, d).
edge(c, e).
edge(d, h).
edge(d, f).
edge(e, f).
edge(e, g).
edge(f, g).

path(X, Y) :-
    edge(X, Y).
path(X, Y) :-
    edge(X, Z),
    path(Z, Y).
path(X, Y, [[X,Y]]) :-
    edge(X, Y).
path(X, Y, [[X,Z]|T]) :-
    edge(X, Z),
    path(Z, Y, T).

npath(X, Y, N) :-
    path(X, Y, L),
    length(L, N).
```

1.2.1 Queries

```
| ?- npath(a, g, L).
L = 5 ? ;
L = 4 ? ;
L = 5 ? ;
L = 4 ? ;
L = 3 ? ;
L = 4 ? ;
no
```

2 Lab 2

2.1 Exercise 2.1

```
1  sorted([_|[]]).
2  sorted([X,Y|T]) :-
3    X =< Y,
4    sorted([Y|T]).
5
6  % Selection Sort
7  smin([], Acc, Acc).
8  smin([H|T], [AccH|AccT], Res) :-
9    H < AccH,
10   smin(T, [H|[AccH|AccT]], Res).
11 smin([H|T], [AccH|AccT], Res) :-
12   H >= AccH,
13   smin(T, [AccH|[H|AccT]], Res).
14 smin([H|T], Res) :-
15   smin(T, [H], Res).
16
17 ssort([], []).
18 ssort(List, [Min|SubRes]) :-
19   smin(List, [Min|Rest]),
20   ssort(Rest, SubRes).
21
22 % QuickSort
23 qpart([], _, [], []).
24 qpart([H|T], Pivot, [H|Smaller], Larger) :-
25   H < Pivot,
26   qpart(T, Pivot, Smaller, Larger).
27 qpart([H|T], Pivot, Smaller, [H|Larger]) :-
28   H >= Pivot,
29   qpart(T, Pivot, Smaller, Larger).
30
31 qsort([], []).
32 qsort([Pivot|Rest], Res) :-
33   qpart(Rest, Pivot, Smaller, Larger),
34   qsort(Smaller, ResSmaller),
35   qsort(Larger, ResLarger),
36   append(ResSmaller, [Pivot|ResLarger], Res).
```

2.1.1 Queries

```
| ?- sorted([1,2,3]).
yes
| ?- sorted([1,3,2]).
no
| ?- qsort([2,5,1,6,8,3,4], Sorted).
Sorted = [1,2,3,4,5,6,8] ? ;
no
| ?- ssort([2,5,1,6,8,3,4], Sorted).
Sorted = [1,2,3,4,5,6,8] ? ;
no
```

2.2 Exercise 2.2

```
append1([],L,L).
append1([H|T],L2,[H|L3])  :-  append(T,L2,L3).

% Permutation 1
middle1(X, [X]).
middle1(X, [First|Xs]) :-
  append1(Middle, [Last], Xs),
  middle1(X, Middle).

% Permutation 2
middle2(X, [First|Xs]) :-
  append1(Middle, [Last], Xs),
  middle2(X, Middle).
middle2(X, [X]).

% Permutation 3
middle3(X, [X]).
middle3(X, [First|Xs]) :-
  middle3(X, Middle),
  append1(Middle, [Last], Xs).

% Permutation 4
middle4(X, [First|Xs]) :-
  middle4(X, Middle),
  append1(Middle, [Last], Xs).
middle4(X, [X]).
```

2.2.1 Questions

- Examine how the execution of the following queries is affected by the ordering of the clauses and the premises within clauses.

```
1       | ?- middle(X, [a,b,c]).
2       | ?- middle(a, X).
```

  Try all four possible permutations of middle/2 and (1) explain what happens, (2) why and (3) sketch the SLD tree (you don't have to draw the whole tree...). Don't forget to explain what happens when you ask for more than one answer! Which version is preferable for each type of query?

1. The execution path will differ between permutations. This is because Prolog choses which premise to be resolved depending on the order in the rule and which rule or fact to use depending on the ordering of the program's rules and facts.

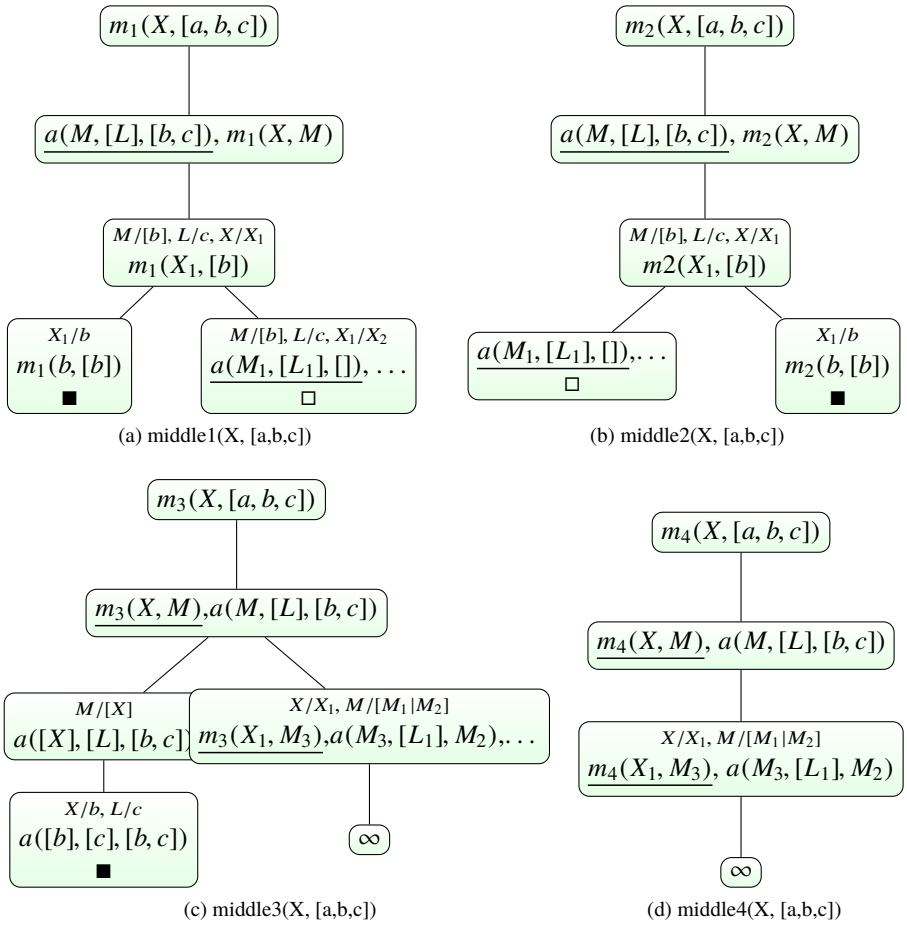
2. For the program - a rule or fact that precedes (read top to down) another rule or fact with the same name and arity will be used first. For the premises of a rule - a premise that precedes (read left-right) another premise will be resolved first.

3. The SLD-trees for each permutation and query is drawn in figure 1, 2 and 3. An underlined clause is selected for resolving. A successful leaf is marked with ■ and an unsuccessful with □.

   When asked for additional answers Prolog backtracks from the last given answer to the next closest possibility. For instance, in figure 2a the first given answer is the leftmost leaf, when asked for more answers Prolog backtracks, tries the other rule for the query which is unsuccessful. Prolog then backtracks all the way to the root since no other possible branches are found on.

   Because of Prolog's selection rule the leftmost child of an SLD-tree is reached first. This means that middle1 (figure 2a) is the most efficient permutation for the query middle(X, [a,b,c]). Although middle3 (figure 3a) reaches the first answer as quickly as middle1 it will continue to loop forever if we ask for more answers. This is due to the ordering of the clauses of middle3. Middle is always an unconstrained variable which, in this case, produces an infinite number of possibilities to try.

   The most efficient permutation for middle(a, X) is middle3 which always reaches a successful leaf (neglecting what happens during the resolving of append). All permutations for this query except middle4 produces an infinite amount of answers. Middle4 never reaches a leaf.



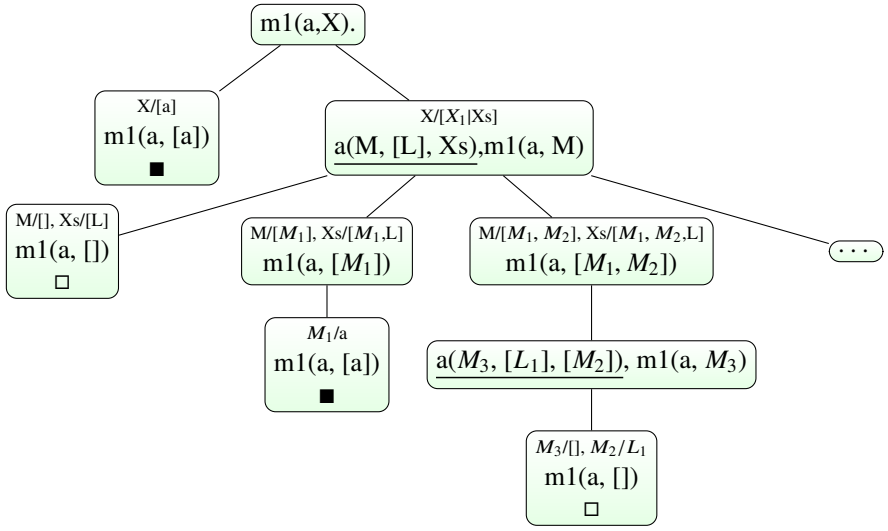
(a) middle1(X, [a,b,c])

(b) middle2(X, [a,b,c])

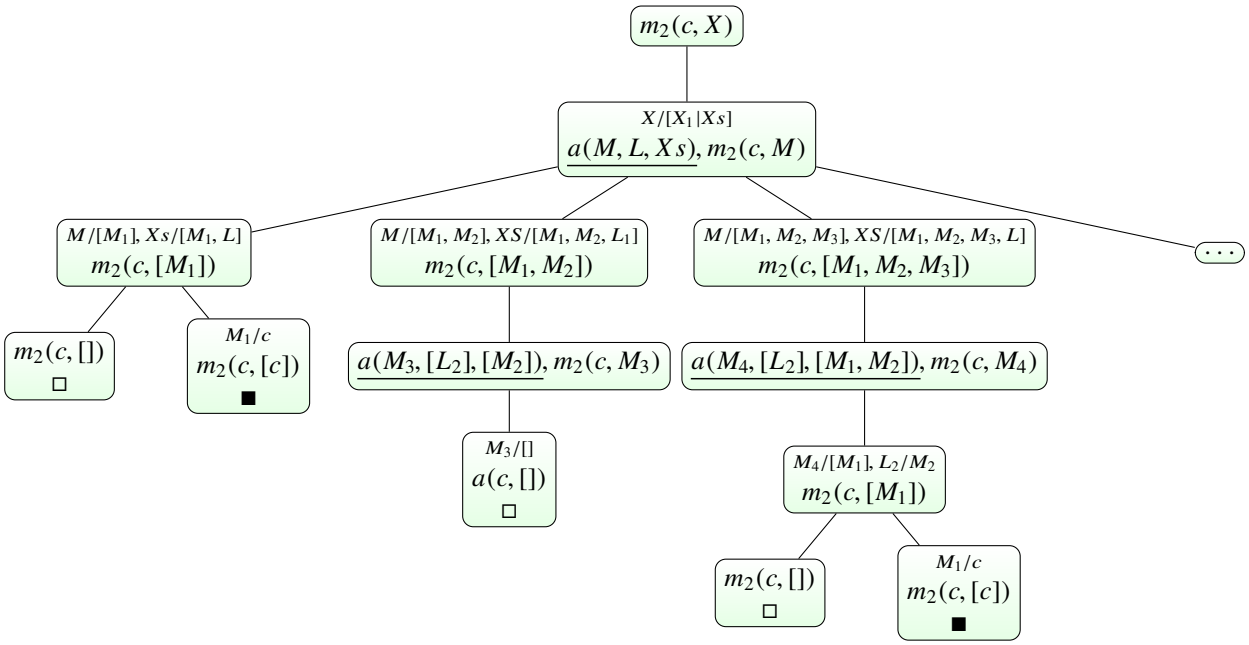
(c) middle3(X, [a,b,c])

(d) middle4(X, [a,b,c])

Figure 1: SLD-trees for queries with the arguments (X, [a,b,c])

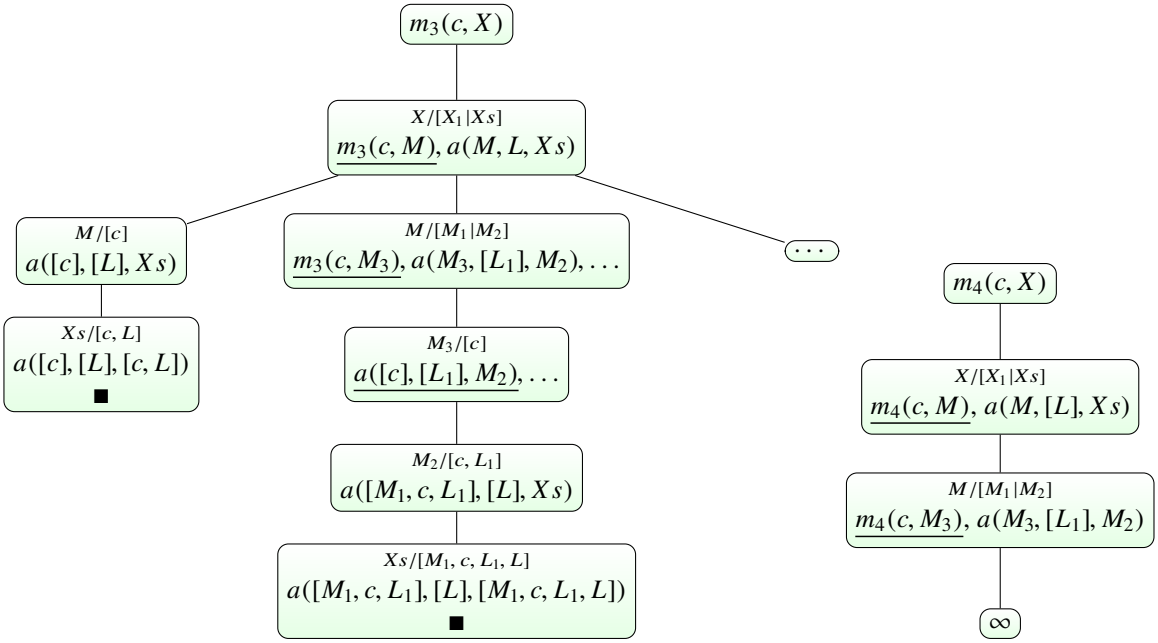


(a) middle1(c, X)



(b) middle2(c, X)

Figure 2: SLD-trees for queries with the arguments (c, X)



(a) middle3(c, X) (b) middle4(c, X)

Figure 3: SLD-trees for queries with the arguments (c, X)

2.3 Exercise 2.3

```
% Binding Environment
bind(id(I), num(N), [], [[I, N]]).
bind(id(I), num(N), [[I, _]|Rest], [[I, N]|Rest]).
bind(I, N, [[HI, HN]|Si], [[HI, HN]|Sj]) :-
  bind(I, N, Si, Sj).

% Arithmetic expressions
eval(Si, id(I), num(Val)) :-
  member([I, Val], Si).
eval(_, num(Val), num(Val)).
eval(Si, (A+B), num(Val)) :-
  eval(Si, A, num(VA)),
  eval(Si, B, num(VB)),
  Val is VA + VB.
eval(Si, (A-B), num(Val)) :-
  eval(Si, A, num(VA)),
  eval(Si, B, num(VB)),
  Val is VA - VB.
eval(Si, (A*B), num(Val)) :-
  eval(Si, A, num(VA)),
  eval(Si, B, num(VB)),
  Val is VA * VB.

% Boolean expressions
eval(Si, (A>B), tt) :-
  eval(Si, A, num(VA)),
  eval(Si, B, num(VB)),
  VA > VB.
eval(Si, (A>B), ff) :-
  eval(Si, A, num(VA)),
  eval(Si, B, num(VB)),
  VA =< VB.
eval(Si, (A<B), tt) :-
  eval(Si, A, num(VA)),
  eval(Si, B, num(VB)),
  VA < VB.
eval(Si, (A<B), ff) :-
  eval(Si, A, num(VA)),
  eval(Si, B, num(VB)),
  VA > VB.

% Commands
execute(Si, skip, Si).
execute(Si, set(I, E), Sj) :-
```

```
45    eval(Si, E, Val),
46    bind(I, Val, Si, Sj).
47  execute(Si, if(Cond, Exec, _), Sj) :-
48    eval(Si, Cond, tt),
49    execute(Si, Exec, Sj).
50  execute(Si, if(Cond, _, Exec), Sj) :-
51    eval(Si, Cond, ff),
52    execute(Si, Exec, Sj).
53  execute(Si, seq(First, Second), Sk) :-
54    execute(Si, First, Sj),
55    execute(Sj, Second, Sk).
56  execute(Si, while(Cond, Exec), Sk) :-
57    eval(Si, Cond, tt),
58    execute(Si, Exec, Sj),
59    execute(Sj, while(Cond, Exec), Sk).
60  execute(Si, while(Cond, _), Si) :-
61    eval(Si, Cond, ff).
```

2.3.1 Queries

```
1  | ?- execute([[x, 3]], seq(set(id(y),num(1)),
2    while(id(x) > num(1),
3      seq(set(id(y), id(y) * id(x)),
4        set(id(x), id(x) - num(1))
5      )
6    )
7  ), Sj).
8  Sj = [[x,1],[y,6]] ?
9  yes
```

2.4 Exercise 2.4

```
% Union
% Assumes A, B are sorted lists with disjunct elements
union([], [], []).
union([HA|TA], [], [HA|TA]).
union([], [HB|TB], [HB|TB]).
union([HA|TA], [HB|TB], [HA|Res]) :-
  HA @< HB,
  union(TA, [HB|TB], Res).
union([HA|TA], [HB|TB], [HB|Res]) :-
  HA @> HB,
  union([HA|TA], TB, Res).
union([H|TA], [H|TB], [H|Res]) :-
  union(TA, TB, Res).

% Intersection
% Assumes A, B are sorted lists with disjunct elements
intersection([], [], []).
intersection([_HA|_TA], [], []).
intersection([], [_HB|_TB], []).
intersection([H|TA], [H|TB], [H|Res]) :-
  intersection(TA, TB, Res).
intersection([HA|TA], [HB|TB], Res) :-
  HA @< HB,
  intersection(TA, [HB|TB], Res).
intersection([HA|TA], [HB|TB], Res) :-
  HA @> HB,
  intersection([HA|TA], TB, Res).

% Powerset
% Assumes A, B are sorted lists with disjunct elements
subset([], []).
subset([H|TA], [H|TB]) :-
  subset(TA, TB).
subset(A, [_|TB]) :-
  subset(A, TB).

powerset(A, Res) :-
  findall(SubA, subset(SubA, A), Sets),
  sort(Sets, Res).
```

2.4.1 Queries

```
| ?- union([a, b, c, d, f], [e, f, g], Res).
Res = [a,b,c,d,e,f,g] ? ;
```

```
3 no
4 | ?- intersection([a, d], [a, b, c, d], Res).
5 Res = [a,d] ? ;
6 no
7 | ?- powerset([a, b, c], Res).
8 Res = [[],[a],[a,b],[a,b,c],[a,c],[b],[b,c],[c]] ? ;
9 no
```

3 Lab 3

3.1 Exercise 1

```
1  term(id(X)) -->
2    [id(X)],
3    {atom(X)}.
4  term(num(X)) -->
5    [num(X)],
6    {number(X)}.
7
8  factor(X + Y) -->
9    term(X),
10   [+],
11   factor(Y).
12 factor(X) -->
13   term(X).
14
15 expr(X * Y) -->
16   factor(X),
17   [*],
18   expr(Y).
19 expr(X) -->
20   factor(X).
21
22 bool(X < Y) -->
23   expr(X),
24   [<],
25   expr(Y).
26 bool(X > Y) -->
27   expr(X),
28   [>],
29   expr(Y).
30
31 cmd(skip) -->
32   [skip].
33 cmd(set(id(I), E)) -->
34   term(id(I)),
35   [:=],
36   expr(E).
37
38 cmd(if(B, C1, C2)) -->
39   [if],
40   bool(B),
41   [then],
42   pgm(C1),
```

```
43    [else],
44    pgm(C2),
45    [fi].
46  cmd(while(B, C)) -->
47    [while],
48    bool(B),
49    [do],
50    pgm(C),
51    [od].
52
53  pgm(C) --> cmd(C).
54  pgm(seq(C1, C2)) -->
55    cmd(C1),
56    [;],
57    pgm(C2).
58
59  parse(Tokens, AbstStx) :-
60    pgm(AbstStx, Tokens, []).
61
62  run(In, String, Out) :-
63    scan(String, Tokens),
64    parse(Tokens, AbstStx),
65    execute(In, AbstStx, Out).
```

3.1.1 Queries

```
1  | ?- run([[x,3]],
2  "y:=1;␣z:=0;␣while␣x>z␣do␣z:=z+1;␣y:=y*z␣od",
3  Res).
4  Res = [[x,3],[y,6],[z,3]] ?
5  yes
```

4 Lab 4

4.1 State Space and Print

```
:- prolog_flag(toplevel_print_options, _,
 [quoted(true),numbervars(true),portrayed(true),max_depth(20)]).

% Valid states
valid(C, 0) :-
  C >= 0.
valid(C, M) :-
  M  > 0,
  M >= C,
  C >= 0.
valid([LC:LM,_,RC:RM]) :-
  valid(LC, LM),
  valid(RC, RM).

% Edges
% A cannibal from left to right
edge([LC:LM,left,RC:RM], NS) :-
  NewLC is LC - 1,
  NewRC is RC + 1,
  NS = [NewLC:LM,right,NewRC:RM],
  valid(NS).
% A missionarie from left to right
edge([LC:LM,left,RC:RM], NS) :-
  NewLM is LM - 1,
  NewRM is RM + 1,
  NS = [LC:NewLM,right,RC:NewRM],
  valid(NS).
% Two cannibals from left to right
edge([LC:LM,left,RC:RM], NS) :-
  NewLC is LC - 2,
  NewRC is RC + 2,
  NS = [NewLC:LM,right,NewRC:RM],
  valid(NS).
% Two missionaries from left to right
edge([LC:LM,left,RC:RM], NS) :-
  NewLM is LM - 2,
  NewRM is RM + 2,
  NS = [LC:NewLM,right,RC:NewRM],
  valid(NS).
% Mixed from left to right
edge([LC:LM,left,RC:RM], NS) :-
  NewRM is RM + 1,
```

```
43    NewRC is RC + 1,
44    NewLC is LC - 1,
45    NewLM is LM - 1,
46    NS = [NewLC:NewLM,right,NewRC:NewRM],
47    valid(NS).
48
49  % A cannibal from right to left
50  edge([LC:LM,right,RC:RM], NS) :-
51    NewLC is LC + 1,
52    NewRC is RC - 1,
53    NS = [NewLC:LM,left,NewRC:RM],
54    valid(NS).
55  % A missionarie from right to left
56  edge([LC:LM,right,RC:RM], NS) :-
57    NewLM is LM + 1,
58    NewRM is RM - 1,
59    NS = [LC:NewLM,left,RC:NewRM],
60    valid(NS).
61  % Two cannibals from right to left
62  edge([LC:LM,right,RC:RM], NS) :-
63    NewLC is LC + 2,
64    NewRC is RC - 2,
65    NS = [NewLC:LM,left,NewRC:RM],
66    valid(NS).
67  % Two missionaries from right to left
68  edge([LC:LM,right,RC:RM], NS) :-
69    NewLM is LM + 2,
70    NewRM is RM - 2,
71    NS = [LC:NewLM,left,RC:NewRM],
72    valid(NS).
73  % Mixed from right to left
74  edge([LC:LM,right,RC:RM], NS) :-
75    NewLM is LM + 1,
76    NewLC is LC + 1,
77    NewRC is RC - 1,
78    NewRM is RM - 1,
79    NS = [NewLC:NewLM,left,NewRC:NewRM],
80    valid(NS).
81
82  printReverse([N]) :-
83    write('START~>'), write(N), write('~>').
84  printReverse([N|Nodes]) :-
85    printReverse(Nodes),
86    write(N), write('~>').
87
88  printNodes(Nodes) :-
```

```
  nl,nl,
  printReverse(Nodes),
  write('~>END.'),nl,nl.
```

4.2 Exercise 1

```
% For every element in the second list, make a new entry with that element infro
expand(_X, [], []).
expand(X, [Y|Z], [[Y|X]|W]) :-
  expand(X, Z, W).

bfsPath([CurBranch|_Branches], Goal, Path) :-
  CurBranch = [Goal|Branch],
  nonmember(Goal, Branch),
  Path = CurBranch.
bfsPath([CurBranch|Branches], Goal, Path) :-
  CurBranch = [Leaf|Branch],
  member(Leaf, Branch),
  bfsPath(Branches, Goal, Path).
bfsPath([CurBranch|Branches], Goal, Path) :-
  CurBranch = [Leaf|Branch],
  nonmember(Leaf, Branch),
  findall(X, edge(Leaf, X), Adjacent),
  expand(CurBranch, Adjacent, Expanded),
  append(Branches, Expanded, NewBranches),
  bfsPath(NewBranches, Goal, Path).

bfs(Path) :-
  Start = [3:3,left,0:0],
  Goal  = [0:0,right,3:3],
  bfsPath([[Start]], Goal, Path).
```

4.2.1 Queries

```
| ?- bfs(P), printNodes(P).


START~>[3:3,left,0:0]~>[1:3,right,2:0]~>[2:3,left,
    1:0]~>[0:3,right,3:0]~>[1:3,left,2:0]~>[1:1,right,
    2:2]~>[2:2,left,1:1]~>[2:0,right,1:3]~>[3:0,left,
    0:3]~>[1:0,right,2:3]~>[2:0,left,1:3]~>[0:0,right,
    3:3]~>~>END.

P = [[0:0,right,3:3],[2:0,left,1:3],[1:0,right,2:3],[3:0,
    left,0:3],[2:0,right,1:3],[2:2,left,1:1],[1:1,right,
    2:2],[1:3,left,2:0],[0:3,right,3:0],[2:3,left,1:0],
    [1:3,right,2:0],[3:3,left,0:0]] ? ;
```

```
7
8
9  START~>[3:3,left,0:0]~>[1:3,right,2:0]~>[2:3,left,
       1:0]~>[0:3,right,3:0]~>[1:3,left,2:0]~>[1:1,right,
       2:2]~>[2:2,left,1:1]~>[2:0,right,1:3]~>[3:0,left,
       0:3]~>[1:0,right,2:3]~>[1:1,left,2:2]~>[0:0,right,
       3:3]~>~>END.
10
11 P = [[0:0,right,3:3],[1:1,left,2:2],[1:0,right,2:3],[3:0,
       left,0:3],[2:0,right,1:3],[2:2,left,1:1],[1:1,right,
       2:2],[1:3,left,2:0],[0:3,right,3:0],[2:3,left,1:0],
       [1:3,right,2:0],[3:3,left,0:0]] ? ;
12
13
14 START~>[3:3,left,0:0]~>[2:2,right,1:1]~>[2:3,left,
       1:0]~>[0:3,right,3:0]~>[1:3,left,2:0]~>[1:1,right,
       2:2]~>[2:2,left,1:1]~>[2:0,right,1:3]~>[3:0,left,
       0:3]~>[1:0,right,2:3]~>[2:0,left,1:3]~>[0:0,right,
       3:3]~>~>END.
15
16 P = [[0:0,right,3:3],[2:0,left,1:3],[1:0,right,2:3],[3:0,
       left,0:3],[2:0,right,1:3],[2:2,left,1:1],[1:1,right,
       2:2],[1:3,left,2:0],[0:3,right,3:0],[2:3,left,1:0],
       [2:2,right,1:1],[3:3,left,0:0]] ? ;
17
18
19 START~>[3:3,left,0:0]~>[2:2,right,1:1]~>[2:3,left,
       1:0]~>[0:3,right,3:0]~>[1:3,left,2:0]~>[1:1,right,
       2:2]~>[2:2,left,1:1]~>[2:0,right,1:3]~>[3:0,left,
       0:3]~>[1:0,right,2:3]~>[1:1,left,2:2]~>[0:0,right,
       3:3]~>~>END.
20
21 P = [[0:0,right,3:3],[1:1,left,2:2],[1:0,right,2:3],[3:0,
       left,0:3],[2:0,right,1:3],[2:2,left,1:1],[1:1,right,
       2:2],[1:3,left,2:0],[0:3,right,3:0],[2:3,left,1:0],
       [2:2,right,1:1],[3:3,left,0:0]] ? ;
22 no
```

4.3 Exercise 2

```
dfsPath(X, X, Visited, Visited).
dfsPath(X, Z, Visited, Path) :-
  edge(X, Y),
  nonmember(Y, Visited),
  dfsPath(Y, Z, [Y|Visited], Path).

dfs(Path) :-
  Start = [3:3,left,0:0],
  Goal  = [0:0,right,3:3],
  dfsPath(Start, Goal, [Start], Path).
```

4.3.1 Queries

```
| ?- dfs(P), printNodes(P).


START~>[3:3,left,0:0]~>[1:3,right,2:0]~>[2:3,left,
    1:0]~>[0:3,right,3:0]~>[1:3,left,2:0]~>[1:1,right,
    2:2]~>[2:2,left,1:1]~>[2:0,right,1:3]~>[3:0,left,
    0:3]~>[1:0,right,2:3]~>[2:0,left,1:3]~>[0:0,right,
    3:3]~>~>END.

P = [[0:0,right,3:3],[2:0,left,1:3],[1:0,right,2:3],[3:0,
    left,0:3],[2:0,right,1:3],[2:2,left,1:1],[1:1,right,
    2:2],[1:3,left,2:0],[0:3,right,3:0],[2:3,left,1:0],
    [1:3,right,2:0],[3:3,left,0:0]] ? ;


START~>[3:3,left,0:0]~>[1:3,right,2:0]~>[2:3,left,
    1:0]~>[0:3,right,3:0]~>[1:3,left,2:0]~>[1:1,right,
    2:2]~>[2:2,left,1:1]~>[2:0,right,1:3]~>[3:0,left,
    0:3]~>[1:0,right,2:3]~>[1:1,left,2:2]~>[0:0,right,
    3:3]~>~>END.

P = [[0:0,right,3:3],[1:1,left,2:2],[1:0,right,2:3],[3:0,
    left,0:3],[2:0,right,1:3],[2:2,left,1:1],[1:1,right,
    2:2],[1:3,left,2:0],[0:3,right,3:0],[2:3,left,1:0],
    [1:3,right,2:0],[3:3,left,0:0]] ? ;


START~>[3:3,left,0:0]~>[2:2,right,1:1]~>[2:3,left,
    1:0]~>[0:3,right,3:0]~>[1:3,left,2:0]~>[1:1,right,
    2:2]~>[2:2,left,1:1]~>[2:0,right,1:3]~>[3:0,left,
    0:3]~>[1:0,right,2:3]~>[2:0,left,1:3]~>[0:0,right,
```

```
    3:3]~>~>END.
15
16  P = [[0:0,right,3:3],[2:0,left,1:3],[1:0,right,2:3],[3:0,
    left,0:3],[2:0,right,1:3],[2:2,left,1:1],[1:1,right,
    2:2],[1:3,left,2:0],[0:3,right,3:0],[2:3,left,1:0],
    [2:2,right,1:1],[3:3,left,0:0]] ? ;
17
18
19  START~>[3:3,left,0:0]~>[2:2,right,1:1]~>[2:3,left,
    1:0]~>[0:3,right,3:0]~>[1:3,left,2:0]~>[1:1,right,
    2:2]~>[2:2,left,1:1]~>[2:0,right,1:3]~>[3:0,left,
    0:3]~>[1:0,right,2:3]~>[1:1,left,2:2]~>[0:0,right,
    3:3]~>~>END.
20
21  P = [[0:0,right,3:3],[1:1,left,2:2],[1:0,right,2:3],[3:0,
    left,0:3],[2:0,right,1:3],[2:2,left,1:1],[1:1,right,
    2:2],[1:3,left,2:0],[0:3,right,3:0],[2:3,left,1:0],
    [2:2,right,1:1],[3:3,left,0:0]] ? ;
22  no
```

4.3.2 Questions

- The program should be able to enumerate all loop- free solutions by means of backtracking. How many are there?

There are four loop-free solutions.

5 Lab 5

5.1 Exercise 1

```
:- use_module(library(clpfd)).

% Identifier, Persons, Duration
container(a,2,2).
container(b,4,1).
container(c,2,2).
container(d,1,1).

on(a,d).
on(b,c).
on(c,d).

% Creates a new list with time variables.
% An entry is on the form [ID, StartTime, EndTime, Persons, Duration]
getContainersList([], []).
getContainersList([[ID, P, D]|Cntrs], [[ID, _ST, _ET, P, D]|CntrList]) :-
  getContainersList(Cntrs, CntrList).

% Add constraint to start time depending on stacking.
constraintStart([ID, _ST, _ET, _P, _D], [OthID, _OthST, _OthET, _OthM, _OthD]) :
  \+on(OthID, ID).
constraintStart([ID, ST, _ET, _P, _D], [OthID, _OthST, OthET, _OthM, _OthD]) :-
  on(OthID, ID),
  ST #>= OthET.

% Apply constraintStart to Cur w.r.t all Other.
applyConstraintStart(_Cur, []).
applyConstraintStart(Cur, [Oth|Others]) :-
  constraintStart(Cur, Oth),
  applyConstraintStart(Cur, Others).

% Applies start- and end time constraints.
% Enumerate through all ordered pairs of containers and apply constraintStart to
applyConstraints([], _All).
applyConstraints([Cur|Rest], All) :-
  Cur = [_ID, ST, ET, _P, D],
  ST in 0..100,
  ET in 0..100,
  ET #= ST + D,
  applyConstraintStart(Cur, All),
  applyConstraints(Rest, All).

```

```
43 % Assemble a list of tasks for cumulative.
44 getTasks([], []).
45 getTasks([[_ID, ST, ET, P, D]|Rest], [T|Tasks]) :-
46   % Using a non-integer as ID (5th argument) in task crashes.
47   % Assigning a useless integer ID (0) instead.
48   T = task(ST, D, ET, P, 0),
49   getTasks(Rest, Tasks).
50
51 % Sets constraint on Max to be at least the last container's end time.
52 bindEndTime([], _Max).
53 bindEndTime([[_ID, _ST, ET, _P, _D]|Rest], Max) :-
54   Max #>= ET,
55   bindEndTime(Rest, Max).
56
57 printContainers([]).
58 printContainers([[ID, ST, ET, _P, _D]|Rest]) :-
59   write('| ID: '),  write(ID), write(' | Start: '), write(ST), write(' | End: ')
60   printContainers(Rest).
61
62 printSolution(Persons, EndTime, Cost, Cntrs) :-
63   write('+----------------------------------------'), nl,
64   write('| Persons: '), write(Persons), write(' | Duration: '), write(EndTime),
65   write('+----------------------------------------'), nl,
66   printContainers(Cntrs),
67   write('+----------------------------------------'), nl.
68
69 schedule(Cost) :-
70   findall([ID,P,D], container(ID,P,D), List),
71   getContainersList(List, Cntrs),
72   applyConstraints(Cntrs, Cntrs),
73   getTasks(Cntrs, Tasks),
74   Persons in 0..100,
75   EndTime in 0..100,
76   bindEndTime(Cntrs, EndTime),
77   Cost #= Persons * EndTime,
78   % http://www.swi-prolog.org/pldoc/man?predicate=cumulative/2
79   cumulative(Tasks, [limit(Persons)]),
80   % https://sicstus.sics.se/sicstus/docs/3.7.1/html/sicstus_33.html#SEC262
81   labeling([minimize(Cost)], [Cost]),
82   printSolution(Persons, EndTime, Cost, Cntrs).
```

5.1.1 Queries

```
| ?- schedule(Tasks, Cost).
+----------------------------------------
| Persons: 4 | Duration: 4 | Cost: 16
+----------------------------------------
| ID: a | Start: 1 | End: 3
| ID: b | Start: 0 | End: 1
| ID: c | Start: 1 | End: 3
| ID: d | Start: 3 | End: 4
+----------------------------------------
Tasks = [task(1,2,3,2,0),task(0,1,1,4,0),task(1,2,3,2,0),task(3,1,4,1,0)],
Cost = 16 ? ;
no
```