Implementation of a solver for 123 Puzzles using Prolog constraint programming

Carlos Jorge Direito Albuquerque[1], Tito Alexandre Trindade Griné[2]

FEUP-PLOG, Class 3MIEIC5, Group 123 Puzzle_4
[1]up201706735@fe.up.pt
[2]up201706732@fe.up.pt

Abstract. This paper was written for Logic Programming classes as part of the second practical project. The goal was to apply the theoretical knowledge on Prolog's constraint programming interface to solve a constraint satisfaction problem – a simple puzzle called 123. We then developed a program using SICStus Prolog's Constraint Logic Programming Finite Domain library. The program has a database of predefined puzzles, predicates to generate new puzzles and to display them. Unfortunately, we were not able to find a suitable global constraint predicate for our problem. Nevertheless, we developed our own predicates to solve 123 puzzles and arrived at somewhat satisfying execution times. It became clear to us that constraint programming can be a very powerful tool to solve constraint driven problems then again, discovering an effective and clean solution, can prove itself quite challenging for most problems.

Keywords: 123 Puzzle, Constraint Programming, Constraint Satisfaction Problem, Finite Domain, Prolog, SICStus.

1 Introduction

This paper is part of the second practical project of Logic Programming class from the Faculty of Engineering of the University of Porto (FEUP). The goal of the project was to solve a constraint driven problem using Constraint Logic Programming (CLP) class of programming languages. This class is mainly declarative and quite efficient on solving constraints. We also aim to introduce CLP in a practical way focused on a concrete example, so the reader can better understand the theory behind this kind of problems.

The problem we chose to explore was the 123 Puzzle from Erich Friedman. The puzzle's rules are explained in the next chapter. One can quickly understand that a puzzle is nothing more than a set of rules that lead us to arrive at a particular goal/solution. Rules are in fact just constraints of the given problem and, as such, puzzles are a perfect example of a Constraint Satisfaction Problem (CSP) – problem where the satisfaction of all (or a set of) constraints produces valid solutions.

Before moving on to the next sections, it is important to be familiarized with some concepts. First, a CSP is formally described as a tuple (V, D, C), where:

- V = {x_1, x_2, …, x_n} is the set of domain variables, i.e. the set of variables that modulate the problem;
- D is a function that maps each variable in V to a set of values – the variables' domains;
- C = {C_1, C_2, …, C_n} is the set of constraints that stand to a subset of variables in V.

Knowing this, a solution to a CSP is a set of values that when assigned to each variable satisfy all constraints involved.

In the next section, the puzzle will also be described as a formal CSP and all these concepts will be covered and explained in practice. The following sections (3 and 4) explain how we programmed the solver for our problem and the details of our implementation. Afterwards, in section 5 you can find the results and analysis of our program's efficiency.

2 Problem Description

123 Puzzle was created by Erich Friedman in 2010. Erich Friedman was a Professor of Mathematics at Stetson University, located in DeLand, Florida, as described in his bio website [1]. We will describe the puzzle's rules and show some examples, but feel free to check their original page [2].

The puzzle consists of a square board where the blank squares can only be filled with 1s, 2s or 3s. The board starts off with some squares already filled. The goal is to fill every blank square such that "each number indicates how many squares are part of the horizontally and vertically adjacent group with the same number", as described in the puzzle's website [2].

Consider the following example:

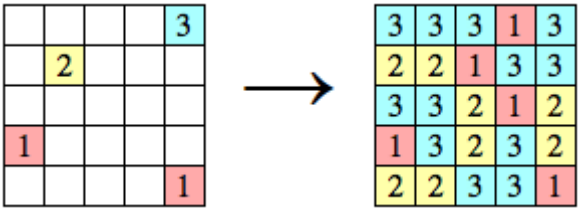

Fig. 1. Example 123 Puzzle: in the left side we have the starting board and on the right side the corresponding solution.

As you can see, every group of 1s, 2s and 3s has exactly 1, 2 and 3 squares, respectively.

This way we have a conceptual description of the puzzle. However, to understand the approach we should take to solve the problem, we need to formally describe the puzzle. The CSP associated with the puzzle can be described by the tuple (V, D, C):

- $V = \{x_{line,column} \mid line, column > 0; line, column \leq n \}$ represents the squares on a n sized square board;
- $D = \{1, 2, 3\}$ represents the possible values for each square (in this case, the domain is a subset of the integer numbers);
- $C = \{\forall x_{i,j} \in V, length(adj_{i,j}) = x_{i,j} - 1\}$ represents the constraint where $adj_{i,j}$ is a list of all adjacent squares with the same value of $x_{i,j}$.

From this moment on we could understand the problem in a mathematical way that helped us define a suitable algorithm. Note that, while the constraint was simple to describe, it proved itself difficult to implement due to the rules of constraint programming. In other words, our program had to take a board (with or without clues) and constrain all squares without knowing beforehand which would be the adjacent squares with the same value of the current one. In the next section we start by describing how SICStus Prolog's solver Constraint Logic Programming Finite Domain (CLPFD) handles constraint programming and how we approached the problem.

3 Approach

To develop our program, we used SICStus Prolog and its CLPFD solver. The solver was included as a library and provided us with valuable operators and constraint predicates. To better explain our program, we first need to establish some base knowledge on constraint programs, but we will only focus the structure of CLP in SICStus. For more information on the solver and its interface, please consult the official documentation [3].

A usual Constraint Logic Program has three main steps:

1. Declaration of variables and corresponding domains;
2. Declaration of constraints on the variables;
3. Search for a solution.

If the order is not respected or if the constraints are not carefully applied to ensure there is no grounding (assignment of a value to a variable), instead of an efficient constraint & test program, we end up with a traditional generate & test approach, which has way more computational cost than the first one. The solver works by cutting down possible values for each variable involved in a given constraint and then propagating the result to every variable of the domain. By applying all constraints, we arrive at the smallest domains possible. Sometimes, a variable may have no possible values; in this case there is no solution for the problem in the given conditions. Once the domains have been constrained, the search mechanism starts by setting possible values for each variable and propagates the consequences of that assignment to every other variable until one of two things happens: all variables have a value, in which case a solution has been

found; or a variable has no possible values, in which case the search mechanism backtracks until a point where all variables had non empty domains.

With all this information, we can now explain how we proceeded to solve the problem.

3.1 Decision variables

According to the structure of a constraint program in SICStus, we first had to define our domain/decision variables. As explained in section 2, to solve a 123 puzzle one must fill every square on the board. Considering that in a CLP problem, the decision variables are the parts of a problem which need to be "filled" (i.e. need to have a value) in order to find a solution, it becomes clear that every square on the board, that is not yet filled, is a decision variable.

To implement this, we began by doing two things:

1. Created a database of puzzles, which in fact are predicates that return a predefined, hard-coded board (see Fig. 2);
2. Used the predicate `length` of SICStus' library to generate empty boards that our solver could then populate with some clues or solve immediately (see Fig. 3).

```
puzzle4x4_1([[_, _, _, _],
             [_, _, 2, _],
             [_, _, _, 2],
             [_, _, _, _]]).
```

Fig. 2. Example of a predefined board in our database.

```
row_list(Size, Row) :-
    length(Row, Size).
generate_empty_board(Size, Board) :-
    length(Board, Size),
    maplist(row_list(Size), Board).
```

Fig. 3. Predicates that generate empty boards filled only with non-grounded variables.

The boards are simply a list of lists, where all the lists have the same size (since the board is square). This way we can easily go through every cell in the board and apply the constraints explained in the next section.

As mentioned in the description of the problem (see section 2. Problem Description), all cells must be filled with either 1s, 2s or 3s, therefore, our variables' domain was set to all integers in the set {1, 2, 3}.

3.2 Constraints

When first looking at this puzzle, it may seem as so the restrictions necessary to apply are straight forward. However, when working with SICStus CLPFD solver, it becomes apparent that applying the restrictions for this project proves to be quite a

challenge. This is because of two main reasons. Firstly, there can't be a distinction in the restrictions applied to a given cell, depending on the number it holds, that is to say, the same restriction must be applied to each cell, regardless of it containing a 1, 2 or a 3. If a distinction is made, then the solver will behave as a generate & test program, meaning it will backtrack and attempt to fill in the cells before it reaches the labelling. Secondly, unlike what happens with a cell containing a 1 or a 2, where only need to check the four surrounding cells and apply a constraint so that there are exactly N-1 cells with the value N. In the case for 1, there must be zero cells around it with the value 1, and with 2, there must be exactly one cell with the same number. The number 3 however isn't as trivial, since a cell with the number 3 can have 1 or 2 adjacent cells with the same number and still be accepted. But, as previously stated, we can't apply a different restriction depending on a cell's number.

In order solve this problem, we realized it would be necessary to use materialized constraints, also called "soft constraints". These constraints are not forced to be respected, however, depending on whether they're true or not, that information is then used by other constraints.

After much trial and error, a valid solution was found, and it consists of two predicates, one that applies a restriction to the variable of the cell it's analysing, `apply_cell_constraint`, and another that applies a restriction to the variables of the cells surrounding it, `apply_adjacent_constraint`.

The first predicate (see Fig. 4), receiving the coordinates of a given cell, starts by using a materialized constraint in order to see if the number on that cell is less than 3, and materializes it through the variable B. This means that if the number is either a 1 or a 2, then B=1, if it's a 3, then B=0. It follows this, by retrieving the variables of the surrounding cells, and applies another constraint using the predicate `exactly`. It will force the number on that cell to be either, the number of adjacent cells with that same number plus one or, the number of cells with that same number plus two minus the value on variable B. Although it may seem confusing at first, effectively what this does is make it so that, if the number on that cell is a 1, then the number of adjacent cells with a 1 will be either zero or zero. The same follows for the number 2, the number of adjacent cells with the number 2 will be either one or one. The difference lies in the number 3, since the variable B will be zero, it forces the number of adjacent cells with the number 3 to be either two or one. This way, it's possible to effectively make a distinction between the numbers 1,2 and 3, without applying a different restriction all together. It then calls the second restriction predicate.

```
apply_cell_constraint(Board, Size, R-C) :-
    get_number(Board, Size, R, C, Number),
    Number #< 3 #<=> B,
    get_adjacent_numbers(Board, Size, R, C, AdjacentNumbers),
    ((Number #= N + 1) #\/ (Number #= N + 2 - B)),
    exactly(Number, AdjacentNumbers, N),
    get_adjacent_coords(Size, R, C, Adjacent),
    apply_adjacent_contraint(Board, Size, Adjacent, Number, N, B).
```

Fig. 4. Predicate responsible for applying restrictions to a given cell.

The second predicate (see Fig. 5) receives a list of the adjacent cells coordinates, the number that is on the original cell, the number of these adjacent cells that have that same value as the first, and the variable B from the previous predicate. It goes through all the adjacent cells on the list, by calling itself until that list is empty. It begins by retrieving the number on the cell whose coordinates are in the head of that list. It then uses yet another materialized constraint, by checking if the number on that cell is equal to the number on the original cell that is being evaluated. The truth value will be materialized through the variable Diff, which will be one if the values are different, or zero if the values are the same. It then follows this by applying a constriction on the number of cells surrounding that one, that have the same number. This number must not be equal to the number of adjacent cells to the original cell with the same value, plus the variable B minus 10 times the variable Diff. The formula may seem convoluted but the reasoning behind it is simple. We must first distinguish when the number on that cell is or isn't the same as the one on the original cell, meaning, when Diff is equal to 0 or 1. In short, when the cell doesn't have the same number, then we don't want to apply any meaningful restriction to the cells surrounding it, since the problem only lies on the cells with the same value. So, when Diff is equal to one, we subtract 10 to the equation, that will result in either the number -9 or -8 and says that this number can't be equal to the number of cells adjacent with that same number. Effectively this does nothing meaningful, since the lowest possible number of adjacent cells with the same value would be zero. This way, when the number is not the same, we essentially ignore it. When the value is the same however, it forces the number of adjacent cells with the same number to not be the same as the number of adjacent cells to the original cell with the same value plus the variable B. When the number in question is a 1 or a 2, the variable B will be set to one, and the effect of this restriction will be that the adjacent cells with the same number can't be one more than the adjacent cells to the original one. In the case of the number being 1, it means that that cell can't have any adjacent cells with the number 1 that have one adjacent cell also with the number 1. Since the number 1 can't have any adjacent cells with the same number, this restriction won't make a difference. Something similar happens if the number is a 2, meaning the other cell, adjacent to the original one, also with the number 2, can't have two adjacent cells with the number 2. This is also already guaranteed by the restriction in the first predicate, since if forces the number of adjacent cells with the value 2 to be exactly one. Once again, the purpose of this restriction is meant for when the original cell has the number 3, but, since we can't distinguish the restrictions applied based on the value of that cell, we are forced to apply this restriction to cases where it wouldn't be necessary. When the cell in question has the number 3, than B will be set to zero, meaning that the restriction will force the number of adjacent cells also with a 3 to be different than the number of adjacent cells, to the original one, with the number 3. Given that when a cell has the number 3 it can only have either one or two cells adjacent to it with the same number, this essentially means that if a cell has a 3, and has two adjacent cells also with the number 3, then those cells can only have one cell adjacent to them, with the number 3. If a cell with the number 3 has only one adjacent cell with the same number, then that cell must have two adjacent cells with the number 3. This ensures that if a cell has the value 3, then it must belong to a group of exactly three cells with the same number.

```
apply_adjacent_contraint(_, _, [], _, _, _).
apply_adjacent_contraint(Board, Size, [R-C | T], Number, N, B) :-
    get_number(Board, Size, R, C, PosNumber),
    PosNumber #\= Number #<=> Diff,
    M #\= N + B - (10 * Diff),
    get_adjacent_numbers(Board, Size, R, C, AdjacentNumbers),
    exactly(Number, AdjacentNumbers, M),
    apply_adjacent_contraint(Board, Size, T, Number, N, B).
```

Fig. 5. Predicate responsible for applying restrictions to all adjacent cells to the original one.

3.3 Puzzle generation

Besides creating a solver, we thought that a puzzle generator would also be an interesting concept to explore. Therefore, we developed a simple strategy to generate a puzzle:

1. Generate an empty board and search for a random solution on that board;
2. Choose a random number of clues, being N the size of the board:

$$\frac{N}{7} \leq Clues \leq \frac{N}{6} \quad (1)$$

3. Generate a new empty board with the same dimensions as the first;
4. Constraint its domain and number of blank squares according the number of clues chosen;
5. Pick randomly from the solved board that number of clues into the empty board.

That is how our `generate_puzzle` predicate works. The trick here was to pick an exact number of clues from one board to another, which was possible by again utilizing reified constraints. The predicate `board_picking` (see Fig. 6) forces the number of equal squares from one board to the other by counting the reifications. `labeling` does the rest of the work by exploring different equal squares.

```
board_picking([], [], 0).
board_picking([Var | RB], [Number | RFB], Clues) :-
    Var #= Number #<=> N,
    Clues #= N + OtherClues,
    board_picking(RB, RFB, OtherClues).
```

Fig. 6. Predicate that picks the clues from one board to another. N is the result of the reified equality between two board squares.

We also implemented count_solutions that counts the number of solutions of a given puzzle by calling findall with solve_puzzle as the goal predicate and by measuring the length of the resulting list of possible solutions.

3.4 Search Strategy

After declaring the constraints, the next step was to search for solutions. SICStus CLPFD solver provides some search/enumeration predicates, such as `indomain`, `labeling` and `solve`. One can learn more about these predicates in SICStus' documentation [4].

For our problem we resorted only to the predicate `labeling` since it allowed us to search through all the domain variables at the same time and specify the heuristics used by the search algorithm. We were only interested in two types of heuristics for our problem: variable choice and value choice. The first defines how the mechanism will choose the next variable and the latter defines how to choose a value for each variable. Possible values for each of these heuristics will be presented in section 5 as we explore and analyse the efficiency of combining different options to solve our problem.

While developing our solver, we used the default heuristics: leftmost for variable choice and step for value choice. These proved to be very efficient heuristics for our problem, as described in section 5.

The only thing we were missing was the list of variables, which was obtained by calling the predicate `append` from the SICStus' lists library. This predicate, when called with just two arguments, will turn a list of lists (our board) into a list with all the variables. With all the steps explained, we have arrived at a complete solver for 123 Puzzle (see Fig. 7).

```
solve_puzzle(Board) :-
    length(Board, Size),
    Max is Size - 1,
    append(Board, FlatBoard),
    % Applying constraints
    domain(FlatBoard, 1, 3),
    findall(R-C, (between(0, Max, R), between(0, Max, C)), Positions),
    maplist(apply_constraint(Board, Size), Positions),
    % Searching for solutions
    labeling([], FlatBoard),
    % Displaying a possible solution
    display_board(Board, Size).
```

Fig. 7. 123 Puzzle solver using CLPDF library. Note that the domain variables are passed as the Board argument. Apart from that, the remaining steps of a CLP are all present.

4 Solution Presentation

In order to make the puzzles, both solved and unsolved, more readable when using our program, a set of predicates, were created in order to display square boards in text mode, when using SICStus Prolog.

The main predicate, `display_board` (see Fig. 8), takes in a Board, which is interpreted as a list of lists, being all of equal size (square board), and a Message, which is a string displayed before the actual board is displayed. The logic is very simple, after getting the length of the given board, it uses `maplist` from the SICStus lists library, to

display all lists (rows) in a user-friendly way. It is worth mentioning that, since the given board doesn't have to be a completed board, a `write_element` predicate was added so that, if a variable on the board is not instantiated or is a 0 (used in the `generate_puzzle` predicate to signify empty cells), it will write a space, otherwise the variable's actual value is written (see Fig. 9). Unfortunately, there is no way, that we found, to add colours in SICStus, which would have made the visualization and verification of a puzzle much easier.

```
% Displays the given board in a friendly format.
display_board(Board, Message) :-
    length(Board, L),
    nl, write(Message), write(':'), nl, nl,
    maplist(draw_board(L), Board),
    write(' '), draw_line_across(L), nl, nl, !.
```

Fig. 8. Main predicate used to display a puzzle.

```
write_element(Element) :-
    (((var(Element) ; Element is 0), write(' ')) ; write(Element)).
```

Fig. 9. Predicate used to display the value of a given cell.

When the board is displayed, either when solving a puzzle (see Fig. 10) or generating one (see Fig. 11), further information about the labelling process is given at the end, namely the time spent, and other information given by the SICStus system library predicate `statistics`.

```
| ?- puzzle6x6_1(Board), solve_puzzle(Board).

Solution:

  ___ ___ ___ ___ ___ ___
 | 3 | 3 | 3 | 2 | 2 | 1 |
  ___ ___ ___ ___ ___ ___
 | 1 | 2 | 2 | 1 | 3 | 3 |
  ___ ___ ___ ___ ___ ___
 | 3 | 3 | 1 | 2 | 3 | 2 |
  ___ ___ ___ ___ ___ ___
 | 3 | 1 | 3 | 2 | 1 | 2 |
  ___ ___ ___ ___ ___ ___
 | 2 | 3 | 3 | 1 | 3 | 3 |
  ___ ___ ___ ___ ___ ___
 | 2 | 1 | 2 | 2 | 3 | 1 |
  ___ ___ ___ ___ ___ ___

 > Duration: 0.145 s

 > Statistics:

Resumptions: 1551138
Entailments: 634225
Prunings: 603377
Backtracks: 4133
Constraints created: 1528

Board = [[3,3,3,2,2,1],[1,2,2,1,3,3],[3,3,1,2,3,2],[3,1,3,2,1,2],[2,3,3,1,3,3],[2,1,2,2,3|...]] ? ;
no
```

Fig. 10. Example of the result of executing `solve_puzzle` on a puzzle existing in the database. Since the puzzle has only one solution, when asked for more solutions, Prolog returns `no`.

```
| ?- generate_puzzle(5, Board).
Puzzle:

 ___ ___ ___ ___ ___
|   | 2 |   |   |   |
 ___ ___ ___ ___ ___
|   |   |   |   |   |
 ___ ___ ___ ___ ___
| 3 |   |   |   |   |
 ___ ___ ___ ___ ___
|   | 2 |   |   |   |
 ___ ___ ___ ___ ___
|   |   | 3 |   |   |
 ___ ___ ___ ___ ___

 > Duration: 0.000 s

 > Statistics:

Resumptions: 14582
Entailments: 4485
Prunings: 5278
Backtracks: 29
Constraints created: 1084

Board = [[0,2,0,0,0],[0,0,0,0,0],[3,0,0,0,0],[0,2,0,0,0],[0,0,3,0,0]] ?
yes
```

Fig. 11. Example of the result from executing `generate_puzzle` with the first argument being 5, meaning the puzzle should be a square board of size 5x5.

5 Results

In order to test the efficiency of our implementation, as well as to find the best variable selection and value selection heuristics to use in the labelling process, the solver was tested multiple times with the various options existing for labeling. For each option, six boards were tested, a 4x4, 5x5 and a 6x6 puzzle that had some cells already filled and had one unique solution. The other three were a 20x20, 40x40 and 80x80 empty puzzle, meaning, the first valid solution that the solver could find was given. For each board the solver was tested between 5-10 times and an average of the labelling times was taken. The measurements were performed on the same computer.

Firstly, the variable ordering was tested, and the results, measure in seconds, are as follows in table 1.

	leftmost[1]	**min**	**max**	**ff**	**ffc**	**anti_first_fail**	**occurence**	**max_regret**
4x4	0.010	0.039	0.015	0.013	0.007	0.045	0.008	0.010
5x5	0.015	0.023	0.024	0.018	0.110	0.022	0.123	0.017
6x6	0.154	31.801	1.992	0.202	0.038	30.563	0.025	0.166
20x20	0.022	*[2]	*	0.019	*	*	*	0.017
40x40	0.081	*	*	0.105	*	*	*	0.096
80x80	0.375	*	*	0.482	*	*	*	0.593

Table 1. Solver labelling times, in seconds, for each variable ordering option in `labeling`.

[1] Default option

[2] A '*' means that the time for the labelling process surpassed 5 minutes and therefore the execution of the solver was halted.

Surprisingly, the best times where achieved using the default option, therefore, when testing the value selection options, the default (leftmost) option was used. The results for the value selection options are the following, shown in table 2.

	step[3]	**enum**	**bisect**	**median**	**middle**
4x4	0.010	0.008	0.009	0.009	0.008
5x5	0.015	0.016	0.015	0.014	0.012
6x6	0.154	0.135	0.151	0.159	0.123
20x20	0.022	0.016	0.016	15.841	0.168
40x40	0.081	0.082	0.082	*[4]	*
80x80	0.375	0.384	0.384	*	*

Table 2. Solver labelling times, in seconds, for each value selection option in `labeling`.

Upon analyzing the results, we were torn on how to interpret them. The first three options provide very similar times, there is no significant disparity to conclude that one is objectively better than the others. However, the last option (middle) proved to be consistently better than the rest when it came to solving the first three puzzles, the ones that already had some cells with numbers. To our disappointment, it quickly became far worse, when used in empty boards. Thus, for the last tests we decided to continue using the default option of step, together with leftmost.

Lastly, the results for the value ordering options are shown below in table 3.

	up[3]	**down**
4x4	0.010	0.003
5x5	0.015	0.007
6x6	0.154	0.054
20x20	0.022	1.494
40x40	0.081	*[4]
80x80	0.375	*

Table 3. Solver labelling times, in seconds, for each value selection option in `labeling`.

Again, to great surprise, the down option proved itself faster in puzzles that weren't empty, but greatly underperformed when solving empty puzzles. One possible explanation is to do with the prevalence of each number in a solved board. We performed a small test to find the percentage of each number on a solved board, using the boards in

[3] Default option

[4] A '*' means that the time for the labelling process surpassed 5 minutes and therefore the execution of the solver was halted.

the database that had only one solution. We found that, on average, 21% of the cells contained the number 1, 26% contained the number 2 and 52% the number 3. Thus, one can assume that the labelling, when choosing values by descending order, would first begin with a 3, can become more successful. It can be viewed almost like a greedy approach to the problem. However, when dealing with empty board, the same approach may become costly, as trying a 3 as the first guess leads to a time-consuming backtracking, being that the 3 ends up having a more complex constraint than a 1 or a 2.

6 Conclusions and Future Work

In conclusion, working on this project showed how using Constraint Logical Programming can be a powerful tool for these types of problems, that can reach results far better than the standard generate & test approach. However, it was a good example of how even simple constraints can be difficult to implement. With more time, the next step would be to try and implement the cell constraint using an automaton, which would most likely greatly improve the efficiency of our solver.

References

1. Erich's Place homepage, https://www2.stetson.edu/~efriedma/, last accessed 2020/01/01
2. 123 Puzzle Page, https://www2.stetson.edu/~efriedma/puzzle/123/, last accessed 2020/01/02
3. SICStus Constraint Logic Programming over Finite Domains, https://sicstus.sics.se/sicstus/docs/4.1.0/html/sicstus/lib_002dclpfd.html, last accessed 2020/01/02
4. SICStus Enumeration Predicates, https://sicstus.sics.se/sicstus/docs/4.1.0/html/sicstus/Enumeration-Predicates.html#Enumeration-Predicates, last accessed 2020/01/02