Sawyer Redstone

PL 2.0 User Guide

Introduction

PL 2.0 is the second generation of a module that adds Prolog style logic programming capabilities to Python. PL was designed for users who are already very familiar with both Python and Prolog, and who seek a closer integration of the two languages. This reference manual explains how users can use each of PL's features to implement Prolog functionality in Python. In each case, it provides brief examples of translating Prolog code to the PL equivalent. Readers seeking additional information are referred to the Appendix, which provides several detailed examples of complete Prolog programs and their equivalent PL versions.

Predicates

All predicates that the user plans to work with must initially be declared by creating instances of the PL.Predicate class, using the following format:

`PredName = Predicate(`*`stringOfName`*`)`

Below is an example of defining the predicate `child`:

```
child = Predicate("child")
```

Once defined, a predicate can be used in the form of `PredName(args).`

Variables

PL treats variables identically to Prolog. Ordinary variables are represented by non-zero length strings beginning with an uppercase character. For example, valid variables would include `"X"` and `"Madonna"`. PL also fully implements Prolog's *anonymous variable* feature, in which variables starting with an `"_"` serve as temporary placeholder variables.

Constants

Constants in PL comprise any numbers or strings without spaces that do not begin with an uppercase character or "_". Examples of constants include `"apple"`, `"++PLUS"`, `"'Bubble'"`, and `19.2`. To include blank spaces within a constant, there must be single quotes within the string, e.g., `"'Hello, World!'"`.

Lists

PL lists can be written in one of two formats:

1. As a collection of terms in square brackets, e.g.: `["X", "banana", "orange", 7]`
2. In the form of `[Head, "|", Tail]`, which works equivalently to the Prolog `[Head|Tail]` format.
   Examples include:
   `["H1", "H2", "|", "T"]`
   `["a", "b", "|", ["c"]]`

Facts

To declare a *fact* in PL, use a predicate followed by the symbols ">> [ ]". The [ ] represents an empty list, and is used to differentiate facts from rules (which, see below). The full format for creating facts is shown below:

```
PredName(args) >> []
```

The following table provides examples of Prolog facts along with their corresponding PL equivalents.

Prolog	PL
`male(X).`	`male("X") >> []`
`child(finn, kyle).`	`child("finn", "kyle") >> []`

Rules

Rules are added similarly to facts. However, rather than an empty list following the >>, rules must include a list of the goals that must be completed to satisfy the rule.

```
PredName(args) >> [goalPred1(args), goalPred2(args), …]
```

The following table provides examples of Prolog rules along with their corresponding PL equivalents.

Prolog	PL
`parent(A, B) :- child(B, A).`	`parent("A", "B") >> [child("B", "A")]`
`father(A, B) :-` `male(A), parent(A, B).`	`father("A", "B") >> [` `male("A"), parent("A", "B")]`

Queries

PL *queries* consist of the keyword "query" followed by the << symbol and a list containing the goals being queried, as shown here:

```
query << [goalPred1(args), goalPred2(args), …]
```

Once the query is made, `query` is set to a list of all the results, where each element of the list is either a dictionary of the unified values or the boolean value True.

The user may optionally choose to return only a specified number of results by adding a parameter numResults to the query as shown here:

```
query(numResults) << [goalPred1(args), goalPred2(args), …]
```

This limited form of query can be especially useful in cases where a query has infinite results.

```
# Add a predicate.
color = Predicate("color")

# Add facts.
color("blue") >> []
color("red") >> []
color("green") >> []

# Run a query.
query << [color("X")]
print(query)    # Prints [{'X': 'blue'}, {'X': 'red'}, {'X': 'green'}]

# Run the same query, limited to two results.
query(2) << [color("X")]
print(query)    # Prints [{'X': 'blue'}, {'X': 'red'}]
```

While `query` is the default keyword to begin queries, it is possible to create queries with other names by creating a new `Query()` object. This can be useful for users who wish to query multiple things at once.

```
colorQuery = Query()
colorQuery << [color("X")]
```

Math

PL supports the standard Prolog math operators : `+, -, *, /, //, ^, **, mod`, and `%`. Note that some of these operators are equivalent, such as `^` with `**` and `%` with `mod`. This allows users to type math in their choice of Python or Prolog style. PL also supports the standard mathematical use of parentheses for grouping. Mathematical expressions must be input in the form of a string.

`equals/2` Predicate

The `equals/2` predicate evaluates its arguments and tries to unify them. This works similarly to the Prolog `is/2` predicate, but unlike `is/2`, `equals` works with either argument unbound.

```
# Two example cases with their results.
query << [equals("X", "(4 + 5) * 2")]    # [{'X': '18'}]
query << [equals(4, "X")]    # [{'X': '4'}]
```

`format_/2` Predicate

The `format_/2` predicate in PL combines the Python `format()` function with the Prolog `format/2` predicate. The PL predicate takes two arguments: i) a string of what the user wishes to print with `{}` (curly braces) representing places for variables to be inserted, and ii) a list of corresponding variables to fill the braces. There must be the same number of variables as curly braces. When the string is printed, the variables will be filled in for each of the placeholder braces.

Prolog	PL
`format("~w's brother is ~w.",` `    [Child1, Child2]).`	`format_("{}'s brother is {}.",` `    ["Child1", "Child2"])`

Built-In Predicates

PL implements many of the standard Prolog built-in predicates.

A full list is below:

```
fail            # fail/0
write           # write/1
nl              # nl/0
member          # member/2
append          # append/3
cut             # !/0
setEqual        # =/2
notEqual        # \=/2
call            # call/1
not_            # \+/1
lt              # </2
le              # =</2
gt              # >/2
ge              # >=/2
between         # between/3
length          # len/2
permutation     # permutation/2
reverse         # reverse/2
equals          # Explanation above.
format_         # Explanation above.
```

All predicates above must be written in the form predName(args). In the case where the predicate has an arity of 0, *args* may be empty, but the parenthesis are still needed. For example, the *failure* predicate `fail/2` is used in the form `fail()`.

Appendix

This appendix contains selected examples of full Prolog programs, followed by their PL equivalents. For brevity, some portions of code have been shortened, but the full code for every model is available at the PL GitHub: LordAsterisk/PL (github.com)

Programs include:

1. Collatz conjecture
2. Family relationship model
3. Flight-Planning System
4. Maze traversal

Sawyer Redstone

1a. Collatz conjecture in Prolog

```
% This code can be found in the following GitHub file:
% PL/collatz.pl at master · LordAsterisk/PL (github.com)
collatz(N, N).
collatz(N0, N) :-
    0 is mod(N0, 2),
    N1 is N0 / 2,
    collatz(N1, N).
collatz(N0, N) :-
    1 is mod(N0, 2),
    N1 is 3 * N0 + 1,
    collatz(N1, N ).

% Example query:
% ?- collatz(10, L).
```

Sawyer Redstone

1b. Collatz conjecture in PL

```python
# This code can be found in the following GitHub file:
# PL/collatz.py at master · LordAsterisk/PL (github.com)

from PL import *

collatz = Predicate("collatz")

collatz("N", "N") >> []
collatz("N0", "N") >> [
    is_(0, "N0 % 2"),
    is_("N1", "N0 / 2"),
    collatz("N1", "N")]
collatz("N0", "N") >> [
    is_(1, "N0 % 2"),
    is_("N1", "3 * N0 + 1"),
    collatz("N1", "N")]

#################
# Example below #
#################

# This predicate has infinite results, so the query must be limited.
query(3) << [collatz(10, "L")]     # Query for 3 results.
print(query)         # prints [{'L': '10'}, {'L': '5.0'}, {'L': '16.0'}]
```

Sawyer Redstone

2a. Family Relationship Model code written in Prolog

```
% For brevity, similar lines will be removed and replaced with "…".
% The full, unshortened code can be found at this GitHub page:
% PL/family.pl at master · LordAsterisk/PL (github.com)

male(bob).
male(john).
% … many lines omitted here.
male(alphonse).
male(jiri).

female(kathryn).
female(beatrice).
% …
female(bertha).
female(fergie).

% A is the child of B
child(bob, john).
child(bob, kathryn).
% …
child(jiri, alphonse).
child(jiri, emma).

parent(A, B) :- child(B, A).

father(A, B) :- male(A), parent(A, B).
mother(A, B) :- female(A), parent(A, B).

sibling(A, B) :- parent(X, A), parent(X, B), A \= B.

uncle(A, B) :- parent(X, B), sibling(A, X), male(A).
aunt(A, B) :- parent(X, B), sibling(A, X), female(A).

ancestor(A, B) :- parent(A, B).
ancestor(A, B) :- parent(A, X), ancestor(X, B).

first_cousin(A, B) :- parent(X, A), sibling(Y, X), parent(Y, B).
```

Sawyer Redstone

2b. Family Relationship Model code written in PL

```
# For brevity, similar lines will be removed and replaced with "…".
# The full, unshortened code can be found at this GitHub page:
# PL/family.py at master · LordAsterisk/PL (github.com)

from PL import *

male = Predicate("male")
female = Predicate("female")
child = Predicate("child")
parent = Predicate("parent")
father = Predicate("father")
mother = Predicate("mother")
sibling = Predicate("sibling")
uncle = Predicate("uncle")
aunt = Predicate("aunt")
ancestor = Predicate("ancestor")
first_cousin = Predicate("first_cousin")

male("bob") >> []
male("john") >> []
# … many lines omitted here.
male("alphonse") >> []
male("jiri") >> []

female("kathryn") >> []
female("beatrice") >> []
# …
female("bertha") >> []
female("fergie") >> []

# A is the child of B
child("bob", "john") >> []
child("bob", "kathryn") >> []
# …
child("jiri", "alphonse") >> []
child("jiri", "emma") >> []

parent("A", "B") >> [child("B", "A")]

father("A", "B") >> [male("A"), parent("A", "B")]
```

Sawyer Redstone

```
mother("A", "B") >> [female("A"), parent("A", "B")]

sibling("A", "B") >> [parent("X", "A"), parent("X", "B"), notEqual("A", "B")]

uncle("A", "B") >> [parent("X", "B"), sibling("A", "X"), male("A")]
aunt("A", "B") >> [parent("X", "B"), sibling("A", "X"), female("A")]

ancestor("A", "B") >> [parent("A", "B")]
ancestor("A", "B") >> [parent("A", "X"), ancestor("X", "B")]

first_cousin("A", "B") >> [parent("X", "A"), sibling("Y", "X"), parent("Y", "B")]

# Example query:
query << [ancestor("ben", "X")]
print(query) # Prints [{'X': 'john'}, {'X': 'lillian'}, {'X': 'bob'}, {'X':
'beatrice'}, {'X': 'marjorie'}, {'X': 'david'}]
```

Sawyer Redstone

3a. Simple Flight-Planning System written in Prolog.
Code directly taken from here, and will be removed if requested by the owner:
www.csci.viu.ca/~wesselsd/courses/csci330/code/prolog/flights.pl

```
% For brevity, similar lines will be removed and replaced with "…".
% The full, unshortened code can be found at this GitHub page:
% PL/flight.pl at master · LordAsterisk/PL (github.com)

% simple flight-planning system

% airport(City,Code)
% ------------------
% matches a city name with an airport code, e.g. "nanaimo" with "ycd"
airport('Nanaimo', 'YCD').
airport('Vancouver', 'YVR').
airport('Victoria', 'YYJ').
airport('Calgary', 'YYC').
airport('Lethbridge', 'YQL').
airport('Kamloops', 'YKA').

% flight(DeptAC, ArrAC)
% ---------------------
% as a fact, states there is a direct flight between the
%    departure airport and arrival airport,
% under the given flight code (e.g. "AC123")
flight('YCD', 'YYC').
flight('YCD', 'YVR').
flight('YKA', 'YQL').
% … many lines omitted here.
flight('YYC', 'YYJ').
flight('YYC', 'YVR').

% flights(DeptAC, ArrAC)
% ----------------------
% as a rule, finds a sequence (list) of flights connecting
%    the departure airport to the arrival airport

% direct flight
flights(D,A) :- airport(Dname,D), airport(Aname,A),
   flight(D,A), format("Direct flight ~w(~w) to ~w(~w)~n",[Dname,D,Aname,A]).
```

Sawyer Redstone

```
% one-stop
flights(D,A) :-
   flight(D,I), I \= A, flight(I,A), airport(Dname,D), airport(Iname,I),
airport(Aname,A),
   format("Flight ~w(~w) to ~w(~w) via ~w(~w)~n", [Dname,D,Aname,A,Iname,I]).

% two-stop
flights(D,A) :-
   flight(D,I), I \= A, flight(I,J), J \= A, J \= D, flight(J,A), airport(Dname,D),
   airport(Aname,A), airport(Iname,I), airport(Jname,J),
   format("Flight ~w(~w) to ~w(~w) via ~w(~w) and ~w(~w)~n",
[Dname,D,Aname,A,Iname,I,Jname,J]).

% flights(DeptCity, ArrCity)
% -------------------------
% run a flights query, but starting with the city name (translate to airport codes)
flights(DC,AC) :- airport(DC,D), airport(AC,A), flights(D,A).
```

Sawyer Redstone

3b. Simple Flight-Planning System written in PL

```
# For brevity, similar lines will be removed and replaced with "…".
# The full, unshortened code can be found at this GitHub page:
# PL/flight.py at master · LordAsterisk/PL (github.com)

from PL import *

# simple flight-planning system

airport = Predicate("airport")
flight = Predicate("flight")
flights = Predicate("flights")

# airport(City,Code)
# ------------------
# matches a city name with an airport code, e.g. "nanaimo" with "ycd"
airport("'Nanaimo'", "'YCD'") >> []
airport("'Vancouver'", "'YVR'") >> []
airport("'Victoria'", "'YYJ'") >> []
airport("'Calgary'", "'YYC'") >> []
airport("'Lethbridge'", "'YQL'") >> []
airport("'Kamloops'", "'YKA'") >> []

# flight(DeptAC, ArrAC)
# ---------------------
# as a fact, states there is a direct flight between the
#    departure airport and arrival airport,
# under the given flight code (e.g. "AC123")
flight("'YCD'", "'YYC'") >> []
flight("'YCD'", "'YVR'") >> []
flight("'YKA'", "'YQL'") >> []
# … many lines omitted here.
flight("'YYC'", "'YYJ'") >> []
flight("'YYC'", "'YVR'") >> []


# flights(DeptAC, ArrAC)
# ----------------------
# as a rule, finds a sequence (list) of flights connecting
#    the departure airport to the arrival airport
```

Sawyer Redstone

```
# direct flight
flights("D", "A") >> [
    airport("Dname", "D"),
    airport("Aname", "A"),
    flight("D", "A"),
    format_("Direct flight {}({}) to {}({})\n", ["Dname", "D", "Aname", "A"])]


# one-stop
flights("D", "A") >> [
    flight("D", "I"), notEqual("I", "A"), flight("I", "A"),
    airport("Dname", "D"), airport("Iname", "I"), airport("Aname", "A"),
    format_("Flight {}({}) to {}({}) via {}({})\n", ["Dname", "D", "Aname", "A",
"Iname", "I"])]

# two-stop
flights("D", "A") >> [
    flight("D", "I"), notEqual("I", "A"), flight("I", "J"), notEqual("J", "A"),
    notEqual("J", "D"), flight("J", "A"), airport("Dname", "D"),
    airport("Aname", "A"), airport("Iname", "I"), airport("Jname", "J"),
    format_("Flight {}({}) to {}({}) via {}({}) and {}({})\n", ["Dname", "D",
"Aname", "A", "Iname", "I", "Jname", "J"])]

# flights(DeptCity, ArrCity)
# --------------------------
# run a flights query, but starting with the city name (translate to airport codes)
flights("DC", "AC") >> [airport("DC", "D"), airport("AC", "A"), flights("D", "A")]

# Example query:
# query << [flights("'Nanaimo'", "'Calgary'")]

# This query prints the following output:

# Direct flight Nanaimo(YCD) to Calgary(YYC)
# Flight Nanaimo(YCD) to Calgary(YYC) via Vancouver(YVR)
# Flight Nanaimo(YCD) to Calgary(YYC) via Vancouver(YVR) and Lethbridge(YQL)
# Flight Nanaimo(YCD) to Calgary(YYC) via Vancouver(YVR) and Victoria(YYJ)
```

Sawyer Redstone

4a. Maze traversal written in Prolog

```
% For brevity, similar lines will be removed and replaced with "…".
% The full, unshortened code can be found at this GitHub page:
% PL/maze.pl at master · LordAsterisk/PL (github.com)

%%%%%%%%%%%%%%%%%%%%
% MAZE DATA
%%%%%%%%%%%%%%%%%%%%

% The raw map data. Each mazeWall(R,C) indicates the presence of a wall
% at row R, column C.
mazeWall(0,0).
mazeWall(0,1).
mazeWall(0,2).
mazeWall(0,3).
mazeWall(0,4).
% … many lines omitted here.
mazeWall(14,7).
mazeWall(14,8).
mazeWall(14,9).
mazeWall(14,10).
mazeWall(14,11).

% The maximum dimensions of the maze grid
mazeDimension(15,12).

% The start and end position of the maze.
mazeStartPos(13,6).
mazeEndPos(0,10).

%%%%%%%%%%%%%%%%%%%%
% MAZE PRINTING
%%%%%%%%%%%%%%%%%%%%

% Helper predicates determine which character
% to print for a particular maze position.
% We use S to signify the starting position,
% and E to signify the ending position.
% Stars are used for maze walls, space for
% paths. When drawing the solution, periods
% show the winning path.
```

```
mazeElement(R,C,'S',_) :- mazeStartPos(R,C), !.
mazeElement(R,C,'E',_) :- mazeEndPos(R,C), !.
mazeElement(R,C,'*',_) :- mazeWall(R,C), !.
mazeElement(R,C,'.',V) :- member([R,C],V), !.
mazeElement(_,_,' ',_).

% Newline at the end of each row.
mazeNewLine(C) :- mazeDimension(_,C), nl.

% Print the maze.
printMaze(WinningPath) :-
    mazeDimension(Rows, Cols),
    between(0, Rows, Row),
    between(0, Cols, Col),
    mazeElement(Row, Col, Appearance,WinningPath),
    write(Appearance),
    mazeNewLine(Col),
    fail.

% Shortcut function to print the maze
% if you don't have the solution yet.
printUnsolvedMaze :-
    printMaze([]).

% Shortcut function to find the solution
% and print it.
printSolvedMaze :-
    mazeStartPos(StartR,StartC),
    solve(StartR,StartC,_,_,[[StartR,StartC]],Visited,[],_),
    printMaze(Visited).

% Print out the winning path by calling solve.
winningPath(Path) :-
    mazeStartPos(StartR,StartC),
    solve(StartR,StartC,_,_,[[StartR,StartC]],_,[],Path1),
    reverse(Path1,Path).



%%%%%%%%%%%%%%%%%%%%
% MAZE SOLVING
%%%%%%%%%%%%%%%%%%%%
```

```
% The four cardinal directions that we can move in: north, south, east, west.
direction(n).
direction(s).
direction(e).
direction(w).

% Given a current position in the maze, and a direction of travel, calculate
% a new position.
newPos(OldRow, OldCol, e, OldRow, NewCol) :-
    NewCol is OldCol + 1,
    not(mazeWall(OldRow, NewCol)),
    mazeDimension(_, Y),
    NewCol < Y,
    NewCol >= 0.
newPos(OldRow, OldCol, w, OldRow, NewCol) :-
    NewCol is OldCol - 1,
    not(mazeWall(OldRow, NewCol)),
    mazeDimension(_, Y),
    NewCol < Y,
    NewCol >= 0.
newPos(OldRow, OldCol, n, NewRow, OldCol) :-
    NewRow is OldRow - 1,
    not(mazeWall(NewRow, OldCol)),
    mazeDimension(X, _),
    NewRow < X,
    NewRow >= 0.
newPos(OldRow, OldCol, s, NewRow, OldCol) :-
    NewRow is OldRow + 1,
    not(mazeWall(NewRow, OldCol)),
    mazeDimension(X, _),
    NewRow < X,
    NewRow >= 0.

% Generate all possible moves from given position.
move(CurrentR, CurrentC, NewR, NewC, PosVisitedIn, PosVisitedOut, MoveListIn,
MoveListOut) :-
    direction(D),
    newPos(CurrentR, CurrentC, D, NewR, NewC),
    not(member([NewR, NewC], PosVisitedIn)),
    append([[NewR, NewC]], PosVisitedIn, PosVisitedOut),
    append([D], MoveListIn, MoveListOut).
```

```
% Solve the maze by repeatedly calling move. Stops when
% we reach the maze ending position.
solve(R,C,_NextR,_NextC,VisitedIn,VisitedIn,MoveListIn,MoveListIn) :-
    mazeEndPos(R, C), !.
solve(R,C,_NextR,_NextC,VisitedIn,VisitedOut,MoveListIn,MoveListOut) :-
    move(R, C, NewR, NewC, VisitedIn, V2, MoveListIn, M2),
    solve(NewR, NewC, _, _, V2, VisitedOut, M2, MoveListOut).
```

4b. Maze traversal written in PL

```
# For brevity, similar lines will be removed and replaced with "…".
# The full, unshortened code can be found at this GitHub page:
# PL/maze.py at master · LordAsterisk/PL (github.com)

from PL import *

mazeWall = Predicate("mazeWall")
solve = Predicate("solve")
mazeDimension = Predicate("mazeDimension")
mazeStartPos = Predicate("mazeStartPos")
mazeEndPos = Predicate("mazeEndPos")
mazeElement = Predicate("mazeElement")
mazeNewLine = Predicate("mazeNewLine")
printMaze = Predicate("printMaze")
printUnsolvedMaze = Predicate("printUnsolvedMaze")
printSolvedMaze = Predicate("printSolvedMaze")
winningPath = Predicate("winningPath")
direction = Predicate("direction")
newPos = Predicate("newPos")
move = Predicate("move")

####################
# MAZE DATA
####################

# The raw map data. Each mazeWall(R,C) indicates the presence of a wall
# at row R, column C.
mazeWall(0, 0) >> []
mazeWall(0, 1) >> []
mazeWall(0, 2) >> []
mazeWall(0, 3) >> []
mazeWall(0, 4) >> []
mazeWall(0, 5) >> []
# … many lines omitted here.
mazeWall(14, 7) >> []
mazeWall(14, 8) >> []
mazeWall(14, 9) >> []
mazeWall(14, 10) >> []
mazeWall(14, 11) >> []
```

Sawyer Redstone

```
# The maximum dimensions of the maze grid
mazeDimension(15, 12) >> []

# The start and end position of the maze.
mazeStartPos(13, 6) >> []
mazeEndPos(0, 10) >> []

###################
# MAZE PRINTING
###################

# Helper predicates determine which character
# to print for a particular maze position.
# We use S to signify the starting position,
# and E to signify the ending position.
# Stars are used for maze walls, space for
# paths. When drawing the solution, periods
# show the winning path.
mazeElement("R", "C", "'S'", "_") >> [mazeStartPos("R", "C"), cut()]
mazeElement("R", "C", "'E'", "_") >> [mazeEndPos("R", "C"), cut()]
mazeElement("R", "C", "'*'", "_") >> [mazeWall("R", "C"), cut()]
mazeElement("R", "C", "'.'", "V") >> [member(["R", "C"], "V"), cut()]
mazeElement("_", "_", "' '", "_") >> []


# Newline at the end of each row.
mazeNewLine("C") >> [mazeDimension("_", "C"), nl()]


# Print the maze.
printMaze("WinningPath") >> [
    mazeDimension("Rows", "Cols"),
    between(0, "Rows", "Row"),
    between(0, "Cols", "Col"),
    mazeElement("Row", "Col", "Appearance", "WinningPath"),
    write("Appearance"),
    mazeNewLine("Col"),
    fail()]

# Shortcut function to print the maze
# if you don't have the solution yet.
printUnsolvedMaze() >> [printMaze([])]
```

```
# Shortcut function to find the solution
# and print it.
printSolvedMaze() >> [
    mazeStartPos("StartR", "StartC"),
    solve("StartR", "StartC", "_", "_", [["StartR", "StartC"]], "Visited", [],
"_"),
    printMaze("Visited")]

# Print out the winning path by calling solve.
winningPath("Path") >> [
    mazeStartPos("StartR", "StartC"),
    solve("StartR", "StartC", "_", "_", [["StartR", "StartC"]], "_", [], "Path1"),
    reverse("Path1", "Path")]

####################
# MAZE SOLVING
####################

# The four cardinal directions that we can move in: north,
# south, east, west.
direction("n") >> []
direction("s") >> []
direction("e") >> []
direction("w") >> []

# Given a current position in the maze, and a direction of travel, calculate
# a new position.
newPos("OldRow", "OldCol", "e", "OldRow", "NewCol") >> [
    equals("NewCol", "OldCol + 1"),
    not_(mazeWall("OldRow", "NewCol")),
    mazeDimension("_", "Y"),
    lt("NewCol", "Y"),
    ge("NewCol", 0)]
newPos("OldRow", "OldCol", "w", "OldRow", "NewCol") >> [
    equals("NewCol", "OldCol - 1"),
    not_(mazeWall("OldRow", "NewCol")),
    mazeDimension("_", "Y"),
    lt("NewCol", "Y"),
    ge("NewCol", 0)]
newPos("OldRow", "OldCol", "n", "NewRow", "OldCol") >> [
    equals("NewRow", "OldRow - 1"),
```

```
    not_(mazeWall("NewRow", "OldCol")),
    mazeDimension("X", "_"),
    lt("NewRow", "X"),
    ge("NewRow", 0)]
newPos("OldRow", "OldCol", "s", "NewRow", "OldCol") >> [
    equals("NewRow", "OldRow + 1"),
    not_(mazeWall("NewRow", "OldCol")),
    mazeDimension("X", "_"),
    lt("NewRow", "X"),
    ge("NewRow", 0)]


# Generate all possible moves from given position.
move("CurrentR", "CurrentC", "NewR", "NewC", "PosVisitedIn", "PosVisitedOut",
"MoveListIn", "MoveListOut") >> [
    direction("D"), newPos("CurrentR", "CurrentC", "D", "NewR", "NewC"),
    not_(member(["NewR", "NewC"], "PosVisitedIn")),
    append([["NewR", "NewC"]], "PosVisitedIn", "PosVisitedOut"),
    append(["D"], "MoveListIn", "MoveListOut")]

# Solve the maze by repeatedly calling move. Stops when
# we reach the maze ending position.
solve("R", "C", "_NextR", "_NextC", "VisitedIn", "VisitedIn", "MoveListIn",
"MoveListIn") >> [
    mazeEndPos("R", "C"), cut()]
solve("R", "C", "_NextR", "_NextC", "VisitedIn", "VisitedOut", "MoveListIn",
"MoveListOut") >> [
    move("R", "C", "NewR", "NewC", "VisitedIn", "V2", "MoveListIn", "M2"),
    solve("NewR", "NewC", "_", "_", "V2", "VisitedOut", "M2", "MoveListOut")]
```