CSc 372
Comparative Programming Languages

16: Prolog — Techniques

Christian Collberg
collberg@cs.arizona.edu

Department of Computer Science
University of Arizona

Copyright © 2004 Christian Collberg
A generate-and-test procedure has two parts:

1. A *generator* which can generate a number of possible solutions.

2. A *tester* which succeeds iff the generated result is an acceptable solution.

When the tester fails, the generator will backtrack and generate a new possible solution.
We can define integer arithmetic (inefficiently) in Prolog:

% Integer generator.
is_int(0).
is_int(X) :- is_int(Y), X is Y+1.

% Result = N1 / N2.
divide(N1, N2, Result) :-
    is_int(Result),
    P1 is Result*N2,
    P2 is (Result+1)*N2,
    P1 =< N1, P2 > N1,
    !.

?- divide(6,2,R).
   R = 3
Generate & Test – Division...

\[
\text{is\_int}(0).
\]
\[
\text{is\_int}(X) :- \text{is\_int}(Y), \ X \text{ is } Y+1.
\]
\[
\text{divide}(N1, N2, Result) :-
\]
\[
\text{is\_int}(Result),
\]
\[
P1 \text{ is } Result*N2, \ P2 \text{ is } (Result+1)*N2,
\]
\[
P1 =\leq N1, \ P2 > N1, !.
\]

<table>
<thead>
<tr>
<th>divide(6,2,R) --- N1=6, N2=2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Res</td>
</tr>
<tr>
<td>-----</td>
</tr>
<tr>
<td>0</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>3</td>
</tr>
</tbody>
</table>
This is a part of a program to play Tic-Tac-Toe (Naughts and Crosses).

Two players take turns to put down $\text{x}$ and $\text{o}$ on a 3x3 board. Whoever gets a line of 3 (horizontal, vertical, or diagonal) markers has won.

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

threatened

Naught must put 0 in square 4.

<table>
<thead>
<tr>
<th></th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>X</td>
<td></td>
<td>X</td>
</tr>
</tbody>
</table>

[5]
We’ll look at the predicate `forced_move` which answers the question:

- Am I (the naught-person) forced to put a marker at a particular position?

The program tries to find a line with two crosses.

It only makes sense to find one forced move, hence the cut.
a line(L) is a generator – it generates all possible lines(L).

threatening(L, B, Sq) is a tester – it succeeds if Sq is a threatened square in line L of board B.

forced_move(Board, Sq) :-
    a line(Line),
    threatening(Line, Board, Sq), !.

?- forced_move(b(x, _, o, _, _, _, x, o, x), 4).
   yes

a line([1,2,3]).  a line([4,5,6]).  a line([7,8,9]).
a line([1,4,7]).  a line([2,5,8]).  a line([3,6,9]).
a line([1,5,9]).  a line([3,5,7]).
threatening succeeds if it finds a line with two crosses and one empty square.

threatening([X,Y,Z],B,X) :-
    empty(X,B), cross(Y,B), cross(Z,B).

threatening([X,Y,Z],B,Y) :-
    cross(X,B), empty(Y,B), cross(Z,B).

threatening([X,Y,Z],B,Z) :-
    cross(X,B), cross(Y,B), empty(Z,B).
A square is empty if it is an uninstantiated variable.

\[ \text{arg}(N, S, V) \] returns the \( N \)th element of a structure \( S \).

\[
\text{empty}(\text{Sq}, \text{Board}) :–
  \quad \text{arg}(\text{Sq}, \text{Board}, \text{Val}), \text{var}(\text{Val}).
\]

\[
\text{cross}(\text{Sq}, \text{Board}) :–
  \quad \text{arg}(\text{Sq}, \text{Board}, \text{Val}), \text{nonvar}(\text{Val}), \text{Val}=x.
\]

\[
\text{naught}(\text{Sq}, \text{Board}) :–
  \quad \text{arg}(\text{Sq}, \text{Board}, \text{Val}), \text{nonvar}(\text{Val}), \text{Val}=o.
\]
Generate & Test – Arbitrage

From the Online Webster’s:

**arbitrage** *simultaneous purchase and sale of the same or equivalent security in order to profit from price discrepancies*

?– arbitrage.

dollar dmark yen 1.03751
yen dollar dmark 1.03751
dmark yen dollar 1.03751
arbitrage :-
    profit3(From, Via, To, Profit), % Gen
    Profit > 1.03, % Test
    write(From), write(' '),
    write(Via), write(' '),
    write(To), write(' '),
    write(Profit), nl, fail.

arbitrage.

% Find three currencies, and the profit:
profit3(From, Via, To, Profit) :-
    best_rate(From, Via, P1, R1),
    best_rate(Via, To, P2, R2),
    best_rate(To, From, P3, R3),
    Profit is R1 * R2 * R3.
exchange (pound, dollar, london, 1.550).
exchange (pound, dollar, new_york, 1.555).
exchange (pound, dollar, tokyo, 1.559).
exchange (pound, yen, london, 153.97).
exchange (pound, yen, new_york, 154.05).
exchange (pound, yen, tokyo, 154.3).
exchange (pound, dmark, london, 2.4075).
exchange (pound, dmark, new_york, 2.44).
exchange (pound, dmark, tokyo, 2.408).
exchange (dollar, yen, london, 98.3).
exchange (dollar, yen, new_york, 98.35).
exchange (dollar, yen, tokyo, 98.25).
exchange (dollar, dmark, london, 1.537).
exchange (dollar, dmark, new_york, 1.58).
exchange (dollar, dmark, tokyo, 1.57).
exchange (yen, dmark, london, 0.015635).
exchange (yen, dmark, new_york, 0.0155).
exchange (yen, dmark, tokyo, 0.0158).
% We can convert back and forth % between currencies:
rate(From, To, P, R) :-
    exchange(From, To, P, R).
rate(From, To, P, R) :-
    exchange(To, From, P, S), R is 1/S.

% Find the best place to convert % between currencies From & To:
best_rate(From, To, Place, Rate) :-
    rate(From, To, Place, Rate),
    not((rate(From, To, P1, R1), R1>Rate)).
Stable Marriages

- Suppose there are $N$ men and $N$ women who want to get married to each other.

- Each man (woman) has a list of all the women (men) in his (her) preferred order. The problem is to find a set of marriages that is stable.

A set of marriages is **unstable** if two people who are not married both prefer each other to their spouses. If $A$ and $B$ are men and $X$ and $Y$ women, the pair of marriages $A - Y$ and $B - X$ is unstable if

- $A$ prefers $X$ to $Y$, and
- $X$ prefers $A$ to $B$. 

![Diagram showing marriages and preferences]
# Stable Marriages – Example

<table>
<thead>
<tr>
<th>Person</th>
<th>Sex</th>
<th>1st choice</th>
<th>2nd choice</th>
<th>3rd choice</th>
</tr>
</thead>
<tbody>
<tr>
<td>Avraham</td>
<td>M</td>
<td>Chana</td>
<td>Ruth</td>
<td>Zvia</td>
</tr>
<tr>
<td>Binyamin</td>
<td>M</td>
<td>Zvia</td>
<td>Chana</td>
<td>Ruth</td>
</tr>
<tr>
<td>Chaim</td>
<td>M</td>
<td>Chana</td>
<td>Ruth</td>
<td>Zvia</td>
</tr>
<tr>
<td>Zvia</td>
<td>F</td>
<td>Binyamin</td>
<td>Avraham</td>
<td>Chaim</td>
</tr>
<tr>
<td>Chana</td>
<td>F</td>
<td>Avraham</td>
<td>Chaim</td>
<td>Binyamin</td>
</tr>
<tr>
<td>Ruth</td>
<td>F</td>
<td>Avraham</td>
<td>Binyamin</td>
<td>Chaim</td>
</tr>
</tbody>
</table>

- Chaim-Ruth, Binyamin-Zvia, Avraham-Chana is stable.
- Chaim-Chana, Binyamin-Ruth, Avraham-Zvia is unstable, since Binyamin prefers Zvia over Ruth and Zvia prefers Binyamin over Avraham.
Write a program which takes a set of people and their preferences as input, and produces a set of stable marriages as output.

Input Format:

prefer(avraham, man,
    [chana, tamar, zvia, ruth, sarah]).

men([avraham, binyamin, chaim, david, elazar]).
women([zvia, chana, ruth, sarah, tamar]).

The first rule, says that avraham is a man and that he prefers chana to tamar, tamar to zvia, zvia to ruth, and ruth to sarah.
Stable Marriages — Database . . .

prefer(avraham, man, [chana, tamar, zvia, ruth, sarah]).
prefer(binyamin, man, [zvia, chana, ruth, sarah, tamar]).
prefer(chaim, man, [chana, ruth, tamar, sarah, zvia]).
prefer(david, man, [zvia, ruth, chana, sarah, tamar]).
prefer(elazar, man, [tamar, ruth, chana, zvia, sarah]).
prefer(zvia, woman, [elazar, avraham, david, binyamin, chaim]).
prefer(chana, woman, [david, elazar, binyamin, avraham, chaim]).
prefer(ruth, woman, [avraham, david, binyamin, chaim, elazar]).
prefer(sarah, woman, [chaim, binyamin, david, avraham, elazar]).
prefer(tamar, woman, [david, binyamin, chaim, elazar, avraham]).
gen generates all possible sets of marriages, unstable tests if they are stable.

go :-
  men(ML), women(WL),
  gen(ML, WL, [], L), \+unstable(L),
  show(L), fail.
  go.

?- men(ML), women(WL), gen(ML,WL, [],L).
  L = [m(elazar,tamar), m(david,sarah),
       m(chaim,ruth), m(binyamin,chana),
       m(avraham,zvia)] ;
  ........
Stable Marriages — Generate

\[
\text{gen}([A|\text{M1}], W, \text{In}, \text{Out}) : - \\
\text{delete}(B, W, W1), \\
\text{gen}(\text{M1}, W1, [m(A,B)|\text{In}], \text{Out}). \\
\text{gen}([],[],\text{L},\text{L}).
\]

\[
\text{delete}(A, [A|\text{L}], \text{L}). \\
\text{delete}(A, [X|\text{L}], [X|\text{L1}]) : - \\
\text{delete}(A, \text{L}, \text{L1}).
\]
\% A prefers B to C.
\texttt{pref}(A, B, C) :-
\texttt{prefer}(A, _, L),
append(_, [B|S], L), !,
member(C, S), !.

\texttt{unstable}(L) :-
append(_, [A|R], L),
member(B, R),
(is\_unstable(A,B); 
\texttt{is\_unstable}(B,A)).

\texttt{is\_unstable}(m(A,Y), m(B,X)) :-
\texttt{pref}(A, X, Y),
\texttt{pref}(X, A, B).
Stable Marriages...
“Helder, a poor scientist, was in love with the daughter of an admiral. One day, a general captured the girl. Helder rode to the general’s barrack and killed the general. The girl was grateful and fell in love with Helder. The admiral was so happy to have his daughter back he gave Helder half of all his boats.”

“Who is the father of the girl?”

“Who is rich?”

“Who loves who?”

“Who is poor?”

“Who captured who?”

“Who killed who?”
Puzzles – Bedtime Story...

:- op(500, xfy, 'is_').
:- op(500, yfx, 'loves').
:- op(500, yfx, 'kills').
:- op(500, yfx, 'to').
:- op(500, yfx, 'captures').
:- op(500, yfx, 'rides_to').
:- op(500, yfx, 'gives').
:- op(500, yfx, 'is_father_of').
:- op(800, yfx, 'and').

X and Y :- X, Y.
helder is poor.
helder is scientist.
admiral is happy.
admiral is father of girl.
helder loves girl.
girl loves helder.
general captures girl.
helder kills general.
admiral gives half boats to helder.
Puzzles – Bedtime Story...

% Who loves who?
?- Z loves Y, write(Z), write(‘ loves ’),
     write(Y), nl, fail.
     helder loves girl
     girl loves helder

% Who captures who?
?- Z captures Y.
     Z = general
     Y = girl
% Who kills who?
?- Z kills Y.
  Z = helder
  Y = general

% Who loves who’s daughter?
?- Z loves G and F is_father_of G.
  Z = helder
  G = girl
  F = admiral
The Crewes, Dews, Grandes, and Lands of Bower Street each have a front-yard tree: Catalpa, Dogwood, Gingko, Larch.

The Grandes’ tree and the Catalpa are on the same side of the street.

The Crewes live across the street from the Larch.

The Larch is across the street from the Dews’ house.

No tree starts with the same letter as its owner’s name.

Who owns which tree?
Puzzles – Trees

?- solve.

Grandes owns the Larch
Crewes owns the Dogwood
Dews owns the Ginko
Lands owns the Catalpa
Puzzles – Trees...

Situation 1

Situation 2
% Let’s assume that the Larch is on the north side of the street.
northside('Larch').

% The Crewes live across the street from the Larch. The Larch is across the street from the Dews’ house.
southside('Crewes').
southside('Dews').

% The Grandes’ tree and the 'Catalpa' are on the same side of the street.
northside('Catalpa') :-
    northside('Grandes').
% If Grandes have a 'Larch', then they
% must live on the north side.
northside('Grandes') :-
    have('Grandes', 'Larch').

% Grandes have a 'Larch', if noone
% else does.
have('Grandes','Larch') :-
    not own('Crewes','Larch'),
    not own('Dews','Larch'),
    not own('Lands','Larch')
% then the Deys’ and Crews’ will be
% on the south side. Also, if the
% Catalpa is on the north the Dogwood
% and Ginko must both be on the south
% side (since each house has one tree).
southside('Dogwood') :-
    northside('Larch'),
    northside('Catalpa').
southside('Ginko') :-
    northside('Larch'),
    northside('Catalpa').
% Are you a tree or a plant?
person(X) :- member(X, ['Grandes', 'Crewes', 'Dews', 'Lands']).
tree(X) :- member(X, ['Catalpa', 'Ginko', 'Dogwood', 'Larch']).

% No tree starts with the same letter as its owner's name.
not_own(X, Y) :-
    name(X, [A|_]), name(Y, [A|_]).

% The Grandes' tree and the 'Catalpa'
% are on the same side of the street.
not_own('Grandes', 'Catalpa').
Puzzles – Trees...

% Only a person can own a tree.
not_own(X,Y) :- person(X), person(Y).
not_own(X,Y) :- tree(X), tree(Y).

% A person can only own a tree that’s on
% the same side of the street as
% themselves.
not_own(X,Y) :- northside(X), southside(Y).
not_own(X,Y) :- southside(X), northside(Y).
% You can't own what someone else owns.
not_owns('Crewes', X) :- owns('Dews', X).
not_owns('Lands', X) :- owns('Crewes', X).
not_owns('Lands', X) :- owns('Dews', X).

owns(X, Y) :-
    person(X), tree(Y),
    not(not_owns(X, Y)).

solve :-
    owns(Person, Tree),
    write(Person), write(' owns the '),
    write(Tree), nl, fail.
solve.