A Third Look At Prolog
Outline

- Numeric computation in Prolog
- Problem space search
  - Knapsack
  - 8-queens
- Farewell to Prolog
Unevaluated Terms

Prolog operators allow terms to be written more concisely, but are not evaluated.

These are all the same Prolog term:

\[ +(1,*(2,3)) \]
\[ 1+ *(2,3) \]
\[ +(1,2*3) \]
\[ +(1+(2*3)) \]
\[ 1+2*3 \]

That term does *not* unify with 7.
Evaluating Expressions

?- \( X \) is \( 1+2*3 \).
\( X = 7 \).

The predefined predicate \texttt{is} can be used to evaluate a term that is a numeric expression.

\texttt{is}(X,Y) evaluates the term \( Y \) and unifies \( X \) with the resulting atom.

It is usually used as an operator.
Instantiation Is Required

?- \( Y = X + 2, \ X = 1. \)
\( Y = 1 + 2, \)
\( X = 1. \)

?- \( Y \text{ is } X + 2, \ X = 1. \)
ERROR: \( \text{is}/2: \) Arguments are not sufficiently instantiated
?- \( X = 1, \ Y \text{ is } X + 2. \)
\( X = 1, \)
\( Y = 3. \)
Evaluable Predicates

For $X \text{ is } Y$, the predicates that appear in $Y$ have to be *evaluable predicates*

This includes things like the predefined operators $\text{+}$, $\text{-}$, $\text{*}$ and $\text{/}$

There are also other predefined evaluable predicates, like $\text{abs}(Z)$ and $\text{sqrt}(Z)$
Real Values And Integers

?- X is 1/2.
X = 0.5.

?- X is 1.0/2.0.
X = 0.5.

?- X is 2/1.
X = 2.

?- X is 2.0/1.0.
X = 2.0.

There are two numeric types: integer and real.

Most of the evaluable predicates are overloaded for all combinations.

Prolog is dynamically typed; the types are used at runtime to resolve the overloading.

But note that the goal \( 2 = 2.0 \) would fail.
Comparisons

- Numeric comparison operators: `<`, `>`, `=<`, `>=`, `=:`, `:\=`

- To solve a numeric comparison goal, Prolog evaluates both sides and compares the results numerically.

- So both sides must be fully instantiated.
Comparisons

?- 1+2 < 1*2.
false.

?- 1<2.
true.

?- 1+2=1+3.
false.

?- X is 1-3, Y is 0-2, X =:= Y.
X = -2,
Y = -2.
Equalities In Prolog

We have used three different but related equality operators:

- \texttt{X is Y} evaluates \texttt{Y} and unifies the result with \texttt{X}:
  
  \begin{itemize}
    
    \item \texttt{3 is 1+2} succeeds, but \texttt{1+2 is 3} fails
  \end{itemize}

- \texttt{X = Y} unifies \texttt{X} and \texttt{Y}, with no evaluation: both
  
  \begin{itemize}
    
    \item \texttt{3 = 1+2} and \texttt{1+2 = 3} fail
  \end{itemize}

- \texttt{X =:= Y} evaluates both and compares: both
  
  \begin{itemize}
    
    \item \texttt{3 =:= 1+2} and \texttt{1+2 =:= 3} succeed
    \item \texttt{(and so does 1 =:= 1.0)}
  \end{itemize}

Any evaluated term must be fully instantiated
Example: `mylength`

```
mylength([], 0).
mylength([_|Tail], Len) :-
    mylength(Tail, TailLen),
    Len is TailLen + 1.

?- mylength([a,b,c], X).
X = 3.

?- mylength(X, 3).
```
Counterexample: my\textit{length}

\begin{verbatim}
mylength([],0).
mylength([_|Tail], Len) :-
    mylength(Tail, TailLen),
    Len = TailLen + 1.
\end{verbatim}

?- mylength([1,2,3,4,5],X).
X = 0+1+1+1+1+1.
Example: sum

sum([],0).
sum([Head|Tail],X) :-
    sum(Tail,TailSum),
    X is Head + TailSum.

?- sum([1,2,3],X).
X = 6.

?- sum([1,2.5,3],X).
X = 6.5.
Example: **gcd**

\[
gcd(X,Y,Z) :- \quad \text{Note: not just } \quad gcd(X,X,X)
\]

\[
gcd(X,Y,Z) :- \quad X \leftarrow Y, \quad Z \leftarrow X.
\]

\[
gcd(X,Y,Denom) :- \quad X < Y, \quad NewY \leftarrow Y - X, \quad gcd(X,NewY,Denom).
\]

\[
gcd(X,Y,Denom) :- \quad X > Y, \quad NewX \leftarrow X - Y, \quad gcd(NewX,Y,Denom).
\]
The $\text{gcd}$ Predicate At Work

?- $\text{gcd}(5,5,X)$.
    $X = 5$ .

?- $\text{gcd}(12,21,X)$.
    $X = 3$ .

?- $\text{gcd}(91,105,X)$.
    $X = 7$ .

?- $\text{gcd}(91,X,7)$.
   \textbf{ERROR: Arguments are not sufficiently instantiated}
Cutting Wasted Backtracking

\[
gcd(X, Y, Z) :-
X =:= Y,
Z is X,
!.
\]

\[
gcd(X, Y, \text{Denom}) :-
X < Y,
NewY is Y - X,
gcd(X, NewY, \text{Denom}),
!.
\]

\[
gcd(X, Y, \text{Denom}) :-
X > Y,
NewX is X - Y,
gcd(NewX, Y, \text{Denom}).
\]

If this rule succeeds, there’s no point in trying the others.

Same here.

With those cuts, this test is unnecessary (but we’ll leave it there).
Example: **fact**

```prolog
fact(X,1) :-
    X =:= 1,
    !.

fact(X,Fact) :-
    X > 1,
    NewX is X - 1,
    fact(NewX,NF),
    Fact is X * NF.
```

?- fact(5,X).
X = 120.

?- fact(20,X).
X = 2432902008176640000.

?- fact(-2,X).
false.
```
Outline

- Numeric computation in Prolog
- Problem space search
  - Knapsack
  - 8-queens
- Farewell to Prolog
Problem Space Search

- Prolog’s strength is (obviously) not numeric computation
- The kinds of problems it does best on are those that involve problem space search
  - You give a logical definition of the solution
  - Then let Prolog find it
The Knapsack Problem

- You are packing for a camping trip
- Your pantry contains these items:

<table>
<thead>
<tr>
<th>Item</th>
<th>Weight in kilograms</th>
<th>Calories</th>
</tr>
</thead>
<tbody>
<tr>
<td>bread</td>
<td>4</td>
<td>9200</td>
</tr>
<tr>
<td>pasta</td>
<td>2</td>
<td>4600</td>
</tr>
<tr>
<td>peanut butter</td>
<td>1</td>
<td>6700</td>
</tr>
<tr>
<td>baby food</td>
<td>3</td>
<td>6900</td>
</tr>
</tbody>
</table>

- Your knapsack holds 4 kg.
- What choice <= 4 kg. maximizes calories?
Greedy Methods Do Not Work

<table>
<thead>
<tr>
<th>Item</th>
<th>Weight in kilograms</th>
<th>Calories</th>
</tr>
</thead>
<tbody>
<tr>
<td>bread</td>
<td>4</td>
<td>9200</td>
</tr>
<tr>
<td>pasta</td>
<td>2</td>
<td>4600</td>
</tr>
<tr>
<td>peanut butter</td>
<td>1</td>
<td>6700</td>
</tr>
<tr>
<td>baby food</td>
<td>3</td>
<td>6900</td>
</tr>
</tbody>
</table>

- Most calories first: bread only, 9200
- Lightest first: peanut butter + pasta, 11300
- (Best choice: peanut butter + baby food, 13600)
Search

- No algorithm for this problem is known that
  - Always gives the best answer, and
  - Takes less than exponential time

- So brute-force search is nothing to be ashamed of here

- That’s good, since search is something Prolog does really well
Representation

- We will represent each food item as a term \( \text{food}(N,W,C) \)
- Pantry in our example is
  \[
  \text{[food(bread,4,9200), food(pasta,2,4500), food(peanutButter,1,6700), food(babyFood,3,6900)]}
  \]
- Same representation for knapsack contents
/*
    weight(L,N) takes a list L of food terms, each
    of the form food(Name,Weight,Calories). We
    unify N with the sum of all the Weights.
*/
weight([],0).
weight([food(_,W,_) | Rest], X) :-
    weight(Rest,RestW),
    X is W + RestW.

/*
    calories(L,N) takes a list L of food terms, each
    of the form food(Name,Weight,Calories). We
    unify N with the sum of all the Calories.
*/
calories([],0).
calories([food(_,_,C) | Rest], X) :-
    calories(Rest,RestC),
    X is C + RestC.
A subsequence of a list is a copy of the list with any number of elements omitted.

(Knapsacks are subsequences of the pantry)
?- subseq([1,3],[1,2,3,4]).
true.

?- subseq(X,[1,2,3]).
X = [1, 2, 3] ;
X = [1, 2] ;
X = [1, 3] ;
X = [1] ;
X = [2, 3] ;
X = [2] ;
X = [3] ;
X = [] ;
false.

Note that subseq can do more than just test whether one list is a subsequence of another; it can generate subsequences, which is how we will use it for the knapsack problem.
/*
   knapsackDecision(Pantry,Capacity,Goal,Knapsack) takes a list Pantry of food terms, a positive number Capacity, and a positive number Goal. We unify Knapsack with a subsequence of Pantry representing a knapsack with total calories \(\geq\) goal, subject to the constraint that the total weight is \(\leq\) Capacity.
*/
knapsackDecision(Pantry,Capacity,Goal,Knapsack) :-
   subseq(Knapsack,Pantry),
   weight(Knapsack,Weight),
   Weight =\(\leq\) Capacity,
   calories(Knapsack,Calories),
   Calories \(\geq\) Goal.
This decides whether there is a solution that meets the given calorie goal

Not exactly the answer we want…
Decision And Optimization

- We solved the knapsack decision problem
- What we wanted to solve was the knapsack optimization problem
- To do that, we will use another predefined predicate: `findall`
The **findall** Predicate

- **findall**(X,Goal,L)
  - Finds all the ways of proving **Goal**
  - For each, applies to **X** the same substitution that made a provable instance of **Goal**
  - Unifies **L** with the list of all those **X**’s
Counting The Solutions

?- findall(1, subseq(_,[1,2]), L).
L = [1, 1, 1, 1].

This shows there were four ways of proving \texttt{subseq(\_, [1,2])}

Collected a list of 1’s, one for each proof
Collecting The Instances

The first and second parameters to `findall` are the same

This collects all four provable instances of the goal `subseq(X, [1, 2])`

?- `findall(subseq(X,[1,2]),subseq(X,[1,2]),L).`
L = `[subseq([1, 2], [1, 2]), subseq([1], [1, 2]), subseq([2], [1, 2]), subseq([], [1, 2])].`
Collecting Particular Substitutions

\[
?- \text{findall}(X, \text{subseq}(X, [1,2]), L).
\]
\[
L = [[1, 2], [1], [2], [\text{[]}]].
\]

- A common use of \textit{findall}: the first parameter is a variable from the second
- This collects all four \textit{X}'s that make the goal \textit{subseq}(\textit{X}, [1,2]) provable
/*
    legalKnapsack(Pantry,Capacity,Knapsack) takes a list Pantry of food terms and a positive number Capacity. We unify Knapsack with a subsequence of Pantry whose total weight is <= Capacity.
*/
legalKnapsack(Pantry,Capacity,Knapsack):- 
    subseq(Knapsack,Pantry),
    weight(Knapsack,W),
    W <= Capacity.
maxCalories(List,Result) takes a List of lists of food terms. We unify Result with an element from the list that maximizes the total calories. We use a helper predicate maxC that takes four parameters: the remaining list of lists of food terms, the best list of food terms seen so far, its total calories, and the final result.

maxC([],Sofar,_,Sofar).
maxC([First | Rest],_,MC,Result) :-
calories(First,FirstC),
MC =< FirstC,
maxC(Rest,First,FirstC,Result).
maxC([First | Rest],Sofar,MC,Result) :-
calories(First,FirstC),
MC > FirstC,
maxC(Rest,Sofar,MC,Result).
maxCalories([First | Rest],Result) :-
calories(First,FirstC),
maxC(Rest,First,FirstC,Result).
/*
   knapsackOptimization(Pantry,Capacity,Knapsack) takes
   a list Pantry of food items and a positive integer
   Capacity. We unify Knapsack with a subsequence of
   Pantry representing a knapsack of maximum total
   calories, subject to the constraint that the total
   weight is \leq Capacity.
 */
knapsackOptimization(Pantry,Capacity,Knapsack) :-
    findall(K,legalKnapsack(Pantry,Capacity,K),L),
    maxCalories(L,Knapsack).
?- knapsackOptimization(
| [food(bread,4,9200),
|  food(pasta,2,4500),
|  food(peanutButter,1,6700),
|  food(babyFood,3,6900)],
|  4,
|  Knapsack).
Knapsack = [food(peanutButter, 1, 6700),
  food(babyFood, 3, 6900)] .
Outline

- Numeric computation in Prolog
- Problem space search
  - Knapsack
  - 8-queens
- Farewell to Prolog
The 8-Queens Problem

Chess background:
- Played on an 8-by-8 grid
- Queen can move any number of spaces vertically, horizontally or diagonally
- Two queens are in check if they are in the same row, column or diagonal, so that one could move to the other’s square

The problem: place 8 queens on an empty chess board so that no queen is in check
Representation

- We could represent a queen in column 2, row 5 with the term \texttt{queen(2,5)}
- But it will be more readable if we use something more compact
- Since there will be no other pieces—no \texttt{pawn(X,Y)} or \texttt{king(X,Y)}—we will just use a term of the form \texttt{X/Y}
- (We won’t evaluate it as a quotient)
Example

- A chessboard configuration is just a list of queens
- This one is [2/5, 3/7, 6/1]
/*
   nocheck(X/Y,L) takes a queen X/Y and a list
   of queens. We succeed if and only if the X/Y
   queen holds none of the others in check.
*/
nocheck(_, []).
nocheck(X/Y, [X1/Y1 | Rest]) :-
    X =\= X1,
    Y =\= Y1,
    abs(Y1-Y) =\= abs(X1-X),
    nocheck(X/Y, Rest).
/*
   legal(L) succeeds if L is a legal placement of
   queens: all coordinates in range and no queen
   in check.
*/

legal([]).
legal([X/Y | Rest]) :-
    legal(Rest),
    member(X,[1,2,3,4,5,6,7,8]),
    member(Y,[1,2,3,4,5,6,7,8]),
    nocheck(X/Y, Rest).
Adequate

This is already enough to solve the problem: the query \texttt{legal(X)} will find all legal configurations:

\begin{verbatim}
?- legal(X).
X = [] ;
X = [1/1] ;
X = [1/2] ;
X = [1/3] ;
etc.
\end{verbatim}
8-Queens Solution

- Of course that will take too long: it finds all 64 legal 1-queens solutions, then starts on the 2-queens solutions, and so on.

- To make it concentrate right away on 8-queens, we can give a different query:

```
?- X = [_,_,_,_,_,_,_,_], legal(X).
X = [8/4, 7/2, 6/7, 5/3, 4/6, 3/8, 2/5, 1/1] .
```
Example

- Our 8-queens solution
- \([8/4, 7/2, 6/7, 5/3, 4/6, 3/8, 2/5, 1/1]\)
Room For Improvement

- Slow
- Finds trivial permutations after the first:

```prolog
?- X = [_,_,_,_,_,_,_,_], legal(X).
X = [8/4, 7/2, 6/7, 5/3, 4/6, 3/8, 2/5, 1/1] ;
X = [7/2, 8/4, 6/7, 5/3, 4/6, 3/8, 2/5, 1/1] ;
X = [8/4, 6/7, 7/2, 5/3, 4/6, 3/8, 2/5, 1/1] ;
X = [6/7, 8/4, 7/2, 5/3, 4/6, 3/8, 2/5, 1/1] ;
```

etc.
An Improvement

- Clearly every solution has 1 queen in each column
- So every solution can be written in a fixed order, like this:
  \[ x = [1/_, 2/_, 3/_, 4/_, 5/_, 6/_, 7/_, 8/_] \]
- Starting with a goal term of that form will restrict the search (speeding it up) and avoid those trivial permutations
eightqueens(X) succeeds if X is a legal placement of eight queens, listed in order of their X coordinates.

eightqueens(X) :-
    X = [1/_,2/_,3/_,4/_,5/_,6/_,7/_,8/_],
    legal(X).
nocheck(_, []).  
nocheck(X/Y, [X1/Y1 | Rest]) :-  
% X =\= X1, assume the X's are distinct  
Y =\= Y1,  
abs(Y1-Y) =\= abs(X1-X),  
nocheck(X/Y, Rest).

legal([]).  
legal([X/Y | Rest]) :-  
legal(Rest),  
% member(X,[1,2,3,4,5,6,7,8]), assume X in range  
member(Y,[1,2,3,4,5,6,7,8]),  
nocheck(X/Y, Rest).

Since all X-coordinates are already known to be in range and distinct, these can be optimized a little
Improved 8-Queens Solution

- Now much faster
- Does not bother with permutations

?- eightqueens(X).
X = [1/4, 2/2, 3/7, 4/3, 5/6, 6/8, 7/5, 8/1] ;
X = [1/5, 2/2, 3/4, 4/7, 5/3, 6/8, 7/6, 8/1] ;
etc.
An Experiment

legal([]).
legal([X/Y | Rest]) :-
    legal(Rest),
    % member(X,[1,2,3,4,5,6,7,8]), assume X in range
    1=<Y, Y=<8, % was member(Y,[1,2,3,4,5,6,7,8]),
    nocheck(X/Y, Rest).

■ Fails: “arguments not sufficiently instantiated”

■ The member condition does not just test in-range coordinates; it generates them
Another Experiment

```prolog
legal([]).
legal([X/Y | Rest]) :-
    % member(X,[1,2,3,4,5,6,7,8]), assume X in range
    member(Y,[1,2,3,4,5,6,7,8]),
    nocheck(X/Y, Rest),
    legal(Rest). % formerly the first condition
```

- Fails: “arguments not sufficiently instantiated”
- The `legal(Rest)` condition must come first, because it generates the partial solution tested by `nocheck`
Outline

- Numeric computation in Prolog
- Problem space search
  - Knapsack
  - 8-queens
- Farewell to Prolog
Parts We Skipped

- Some control predicate shortcuts
  - \( \rightarrow \) for if-then and if-then-else
  - \( ; \) for a disjunction of goals

- Exception handling
  - System-generated or user-generated exceptions
  - \texttt{throw} and \texttt{catch} predicates

- The API
  - A small ISO API; most systems provide more
  - Many public Prolog libraries: network and file I/O, graphical user interfaces, etc.
A Small Language

- We did not have to skip as much of Prolog as we did of ML and Java
- Prolog is a small language
- Yet it is powerful and not easy to master
- The most important things we skipped are the *techniques* Prolog programmers use to get the most out of it