Prolog

University of Cambridge

Ashwin Ahuja

Computer Science Tripos Part IB
Paper 7

March 2019

Contents

1 General Syntax

1.1 Basics

Declarative Programming: Describe the result of programming rather than the procedure done. Prolog programs are composed of facts and queries, with the Prolog database able to answer simple questions:

```
?- [user].
|: milestone(rousell, 1972).
|: milestone(warren, 1983). // type Control D when done
|: % user: // 1 compiled 0.01 sec, 764 bytes

?- milestone(warren, 1983).
Yes
?- milestone(warren, X).
X = 1983
Yes
?- milestone(nooo, 2000).
No
?- milestone(X, Y).
X = rousell
Y = 1972 // type ';' for more results

X = warren
Y = 1983 // press enter when you have enough
Yes
?- halt
```

Files: Can also write programs to file, using:

```
cat milestone.pl
milestone(rousell, 1972).
milestone(warren, 1983).
milestone(swiprolog, 1987).

> prolog
?- [milestone] // get prolog to load the program
?- milestone(X, Y).
...
```

1.2 Terms

Atoms / Constants: Of any kind (integer, floats, etc) - strings can either be defined with a lowercase first letter or with quotation marks.

Variables: Can have any type - can also change as required. Has a first letter which is uppercase.

Compound: Composed of a top function symbol and an arity, eg: favouriteFruit(ashwin, pineapple) has the top function symbol of 'favouriteFruit' and an arity of 2.

Unification: (1) Atoms unify if they are identical, (2) Variables unify with anything & (3) Compound terms unify if their top function symbols and arities match and all parameters unify recursively.

1.3 Rules

Rules (≡ First Order Logic) have a head which is true if the body is true:

```
rule(X, Y) :- part1(X), part2(X, Y)
[___H____] [_________B__________]
```

rule(X, Y) is true if part1(X) is true and part2(X, Y) is true. ',' is considered a logical conjunction

Variables can also be internal to a rule and recursive:

```
rule2(X) :- thing(X, Z), thing2(Z). // rule2(X) is true iff there is a Z st thing(X, Z) is
    ↪ true and thing2(Z) is true

rule3(ground).
rule3(In) :- anotherRule(In, Out), rule3(Out).
```

1.4 Arithmetic

Equals (=) in Prolog means unify with, eg: A = 1+2 (+ is the infix version of +(X, Y) - that's the compound term)
–> A= 1+2
–> Yes

The **is** operator tells prolog to evaluate the right-hand expression numerically and unify with the left. However, the right hand side must be a ground term (ie have no variables).

```
?- A is 1+2
A = 3
Yes

?- A is B+2
ERROR: Arguments are not sufficiently instantiated
```

1.5 Lists

Syntax notated with square brackets [1, 2, 3, 4] & and the empty list is denoted []. The pipe ('|') symbol is used to refer to the tail of a list [H | T] where H is an element and T is a list.

1.5.1 Useful Rules

Last Element of List

```
last([H], H).
last([_ | T], H) :- last(T, H).
```

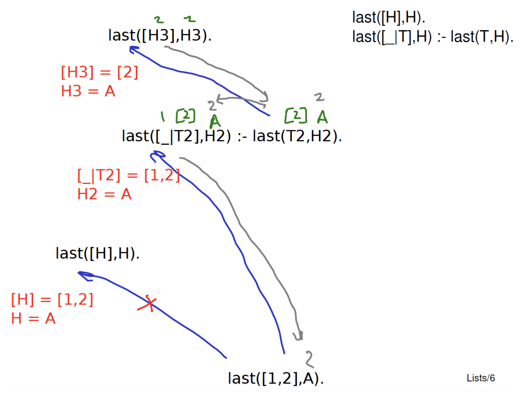


Figure 1: Trace of Last(Element, List)

Length of List (bad way)

```
len([], 0).
len([_, T], N) :- len(T, M), N is M+1.
```

This method uses O(N) stack space, as can be seen from this trace:

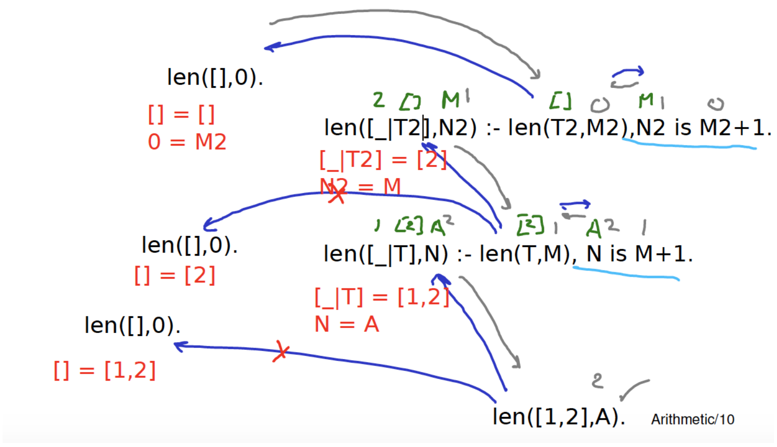


Figure 2: Trace of len(List, Answer)

Length of List using accumulator

```
LEN/3
len([], Acc, Acc).
len([_|T], Acc, Result) :- Result is Acc + 1, len(T, B, Result).

LEN/2
len(List, Result) :- len(list, 0, Result).
```

This method uses O(1) stack space, as can be seen from this trace:

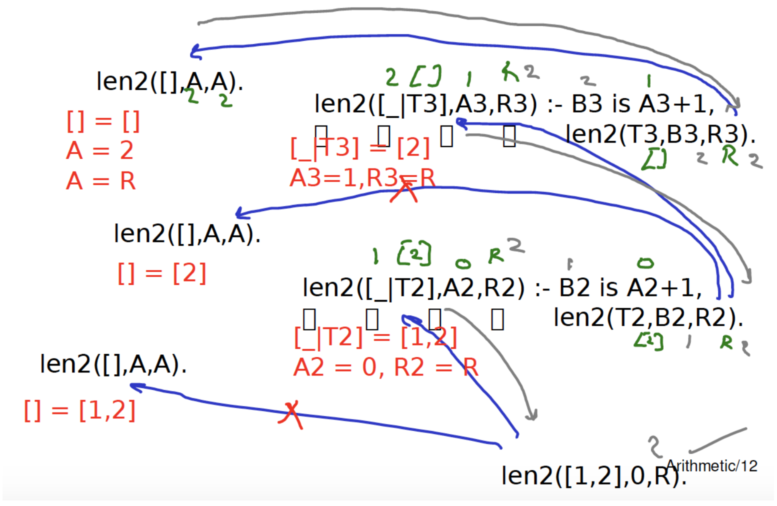


Figure 3: Trace of len(List, Answer)

Appending two lists: This is the non difference list method which is rather inefficient - O(n)

```
append([], L, L).
append([X | T], L, [X | R]) :- append(T, L, R).
```

Reversing a list:

```
reverse([], []).
reverse([X | Xs], Ys) :- reverse(Xs, Rs), append(Rs, [X], Ys).
```

1.6 Debuggers and Debugging

trace: Can be used to follow execution, eg 'trace, last([1, 2], A).' Then can use the word enter to **creep** to the next level, **s** to jump straight to the result of the call.

1.6.1 Last Call Optimisation

Technique is applied by the prolog interpreter - allows the last clause of a rule to be executed as a branch - can forget that we were ever interested in the head. However, we can only do this if the rule is **determinate** up to that point.

How to test: (1) Generate a big list, (2) Test on this list to see if we run out of space.

1.7 Backtracking

Backtracking describes what to do when there are multiple answers and therefore you need to backtrack.

Take first specific element of list: take([1, 2, 3], A, B) should give:
1: [2, 3] ... ;
2: [1, 3] ... ;
3: [1, 2] ... ;
false

```
take([H|T], H, T).
take([H|T], R, [H|S]]) :- take(T, R, S).
```

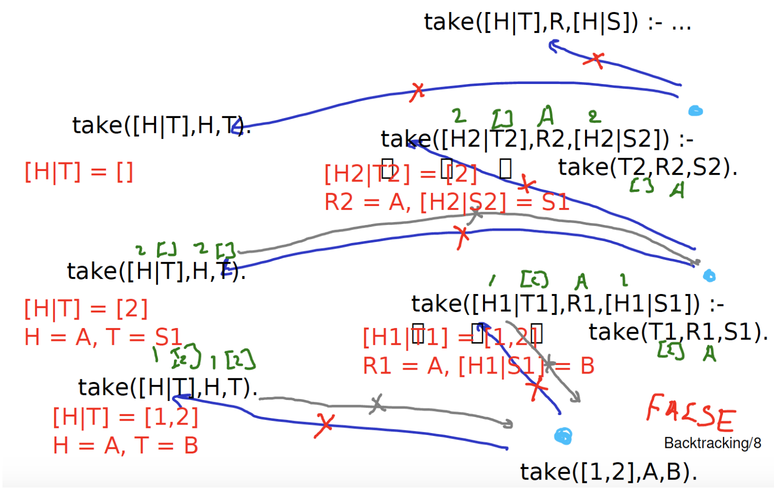


Figure 4: Trace of take

Prolog backtracks by recording 'choice points' which are locations in the search where we could take another option. If there are no choice points left then it doesn't

Doing functions backwards: It is often likely in Prolog that you can use different arrangements of inputs and outputs to get a different intentioned answer from rules.

e.g. backwards len

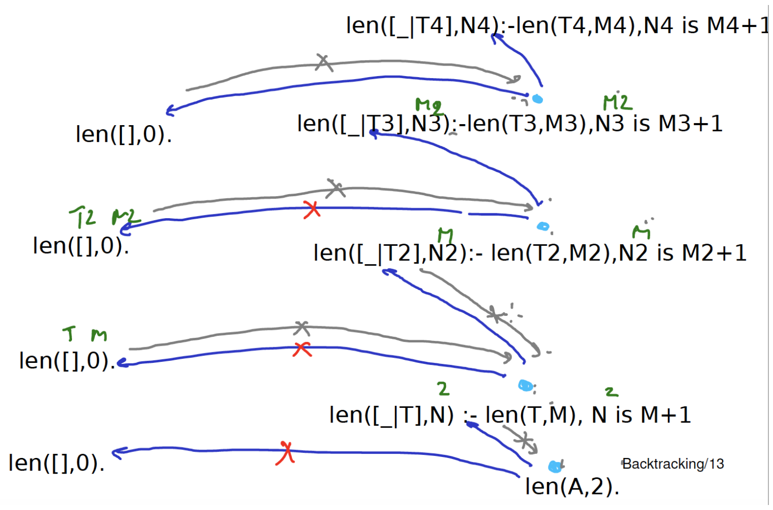

len([_|T4],N4):-len(T4,M4),N4 is M4+1
len([],0).
len([_|T3],N3):-len(T3,M3),N3 is M3+1
len([],0).
len([_|T2],N2):- len(T2,M2),N2 is M2+1
len([],0).
len([_|T],N) :- len(T,M), N is M+1
len([],0).
Backtracking/13
len(A,2).

Figure 5: Backwards len

2 Generate and Test

Generate and test is effectively a design pattern, therefore we can use the permutations function, hence:

```
perm([], []).
perm(L, [H|T]) :- take(L, H, R), perm(R, T).
```

We then combine this generation function with a test in order to check if the created permutation would successfully be tested. Therefore, the process is as follows:

1. Generate a solution
2. Test if its valid
3. If not valid, then backtrack to the next solution

3 Symbolic Evaluation

Symbolic evaluation of arithmetic has the potential of causing a backtracking problem. This can be fixed by using Prolog rules to evaluate symbolic arithmetic expressions.

```
eval(plus(A, B), C) :- eval(A, A1), eval(B, B1), C is A1 + B1.
eval(mult(A, B), C) :- eval(A, A1), eval(B, B1), C is A1 * B1.
eval(A, A).
```

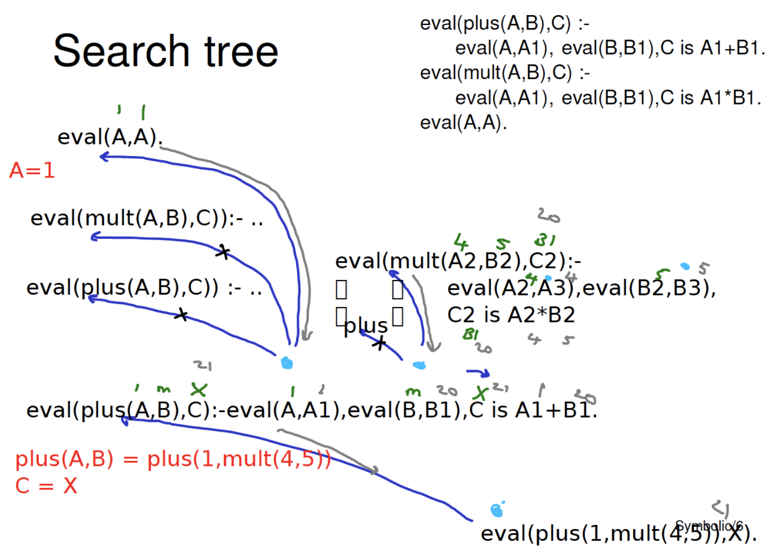


Figure 6: Trace of eval

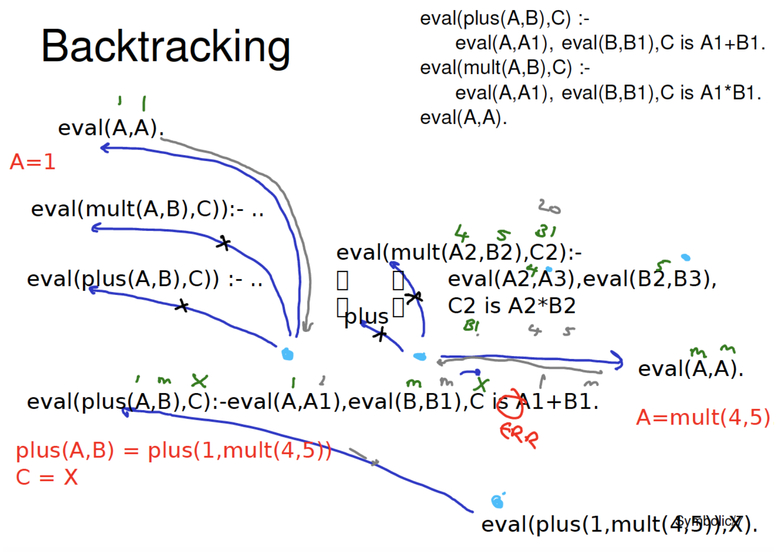


Figure 7: Trace of eval backtracking

The problem with backtracking happens when there are too many choice points, therefore make only one clause that matches at any point. This means making all clauses orthogonal

```
eval(plus(A, B), C) :- eval(A, A1), eval(B, B1), C is A1 + B1.
eval(mult(A, B), C) :- eval(A, A1), eval(B, B1), C is A1 * B1.
eval(gnd(A), A).

// gnd is the ground function - checks if there is a constant
```

4 Cuts

Cuts are a method of limiting the number of choice points that we have in order to control backtracking. It tells Prolog to commit to its choice, using the '!' character, eg:

```
eval(plus(A, B), C) :- !, eval(A, A1), eval(B, B1), C is A1 + B1.
eval(mult(A, B), C) :- !, eval(A, A1), eval(B, B1), C is A1 * B1.
eval(A, A).
```

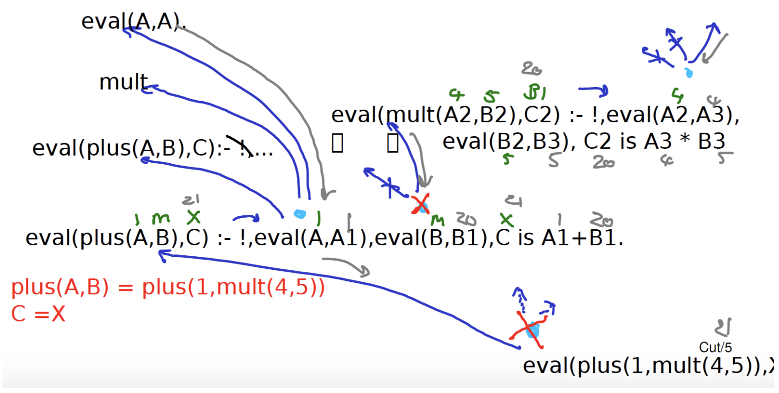

eval(A,A).
mult
eval(plus(A,B),C):-!...
eval(mult(A2,B2),C2) :- !,eval(A2,A3),
eval(B2,B3), C2 is A3 * B3
eval(plus(A,B),C) :- !,eval(A,A1),eval(B,B1),C is A1+B1.
plus(A,B) = plus(1,mult(4,5))
C =X
Cut/5
eval(plus(1,mult(4,5)),X

Figure 8: Trace of cut

Green Cut: Cut where just made to help executions go faster without changing program output.
Red Cut: Any cut that changes logical meaning of the cut.

5 Negation

Allows us to use negative reasoning. **fail** can be used as something that will always return false. Simple example: isDifferent

```
a(A, A) :- !, fail.
a(_, _).
```

However, important to be careful about whether the unification is done in the backtracking process - might undo variable bindings.

Negation-by-Failure

```
not(A) :- A, !, fail.
not(_).
```

not(A) succeeds if you can establish if A is true, then cross the cut and then return a fail. If A fails, then you get a success. not(A) can also be written in Prolog as \+A. not relies upon the closed world assumption - everything that is true in the world is stated or can be derived from the clauses in the program.

Need to be careful about using free variables in negation, for example switching clauses can result in incorrect results - important to consider quanitifiers:

$$expensive(R) = \exists R \cdot expensive(R) \tag{1}$$

$$not(expensive(R)) = \neg(\exists R \cdot expensive(R)) = \forall R \cdot \neg expensive(R) \tag{2}$$

6 Graph Search

Defined World

```
// STATE SPACE
route(a, g).
route(g, l).
route(l, s).
...
start(a).
finish(u)

// LOG SAFE
travel(A, A).
travel(A, C) :- route(A, B), travel(B, C).

solve :- start(A), finish(B), travel(A, B).
```

If we also need to remember the route, then we can change the code to this:

```
travellog(A, A, []).
travellog(A, C, [A-B | Steps]) :- route(A, B), travellog(B, C, Steps).

solve(L) :- start(A), finish(B), travellog(A, B, L).
```

Dealing with Cyclic Graphs: In order to avoid getting into cycles, we store a list of visited nodes and avoid revisiting them.

```
travelsafe(A, A, _).
travelsafe(A, C, Closed) :- route(A, B), \+member(B, Closed), travelsafe(B, C, [B | Closed])
    ↪ .
```

State Space Representation: Need to work out how to represent the problem with as little redundancy as possible. This shortens rules for which the transitions are possible.

7 Difference Lists

Method that helps us to append to a list without copying. Eg. A = 4 :: (5 :: (6 :: [])), List = 1 :: (2 :: (3 :: A))

```
L3 = L1 = [l1_{0}, l1_{1}, l1_{2} | T1]
L2 = [l2_{0}, l2_{1}, l2_{2} | T2]

append(L1, T1, L2, T2, L3, T3) :- T1 = L2, L3 = L1, T3 = T2
// three lists, L1, L2 and L3 - T1 is the pointer to the end of the list, L3 is the result -
    ↪ T3 is the pointer to the end of the result
// this is equivalent to
append(L3, T1, T1, T3, L3, T3).

// redefine this as
append(X-Y, Y-YT, X-YT).
```

[1, 2, 3 | A] - A = [1, 2, 3]
This is a difference list - therefore if you see A-B, then you should imagine that you actually have something of the form [... | B] - B where [... | B] is A

An empty difference list is A-A: this can somehow lead to problems when we try and unify things - all related to A unifying with a(A)

Can be shown by trying to unify A-A = [1, 2, 3 | V] - V, leads to the same problem.

Reverse

```
reverse([], T - T).
reverse([X | Xs], Ys-Yt :- reverse(Xs, Rs - [X | Yt]).
```

Quicksort

```
partition(Pivot, [], [], []).
partition(Pivot, [X | Xs], [X | Ys], Zs) :- X <= Pivot, partition(Pivot, Xs, Ys, Zs).
partition(Pivot, [X | Xs], Ys, [X | Zs]) :- X > Pivot, partition(Pivot, Xs, Ys, Zs).

quicksort([], S-S).

quicksort([X | Xs], Sorted - Tail) :- partition(X, Xs, Smalls, Bigs), quicksort(Smalls,
    ↪ Sorted - [X | T]), quicksort(Bigs, T - Tail).
```

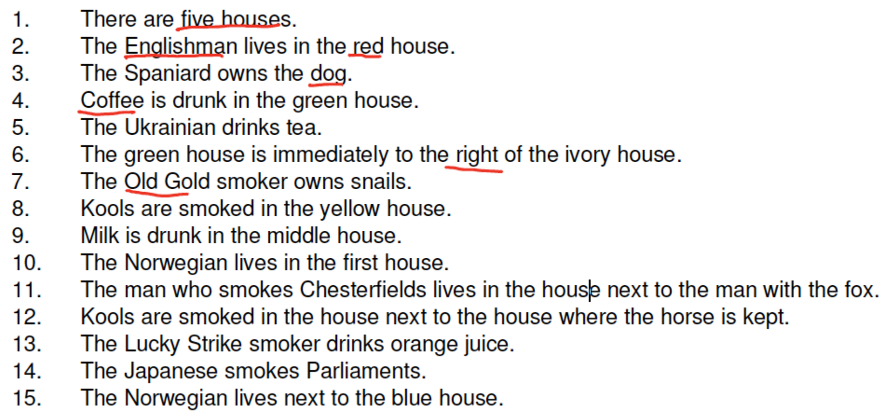
Length

```
len(A - A1, 0) :- unify_with_occurs_check(A, A1).
len([_ | T] - T1, N) :- len(T-T1, M), N is M+1
```

8 Example Problems

8.1 Zebra Puzzle

Puzzle is a form of puzzle where there a number of facts with various pieces of information and then



1. There are five houses.
2. The Englishman lives in the red house.
3. The Spaniard owns the dog.
4. Coffee is drunk in the green house.
5. The Ukrainian drinks tea.
6. The green house is immediately to the right of the ivory house.
7. The Old Gold smoker owns snails.
8. Kools are smoked in the yellow house.
9. Milk is drunk in the middle house.
10. The Norwegian lives in the first house.
11. The man who smokes Chesterfields lives in the house next to the man with the fox.
12. Kools are smoked in the house next to the house where the horse is kept.
13. The Lucky Strike smoker drinks orange juice.
14. The Japanese smokes Parliaments.
15. The Norwegian lives next to the blue house.

Figure 9: Zebra Puzzle Facts

Questions are something like: 'Who drinks water? Who owns the zebra?'

Initial method is: (1) Represent each house with the term and (2) The row of houses as a compound

```
house(Nationality, Pet, Smoke, Drinks, Colour)
(H1, H2, H3, H4, H5)
```

Define facts:

```
exists(A, (A, _, _, _, _)).
exists(A, (_, A, _, _, _)).
exists(A, (_, _, A, _, _)).
exists(A, (_, _, _, A, _)).
exists(A, (_, _, _, _, A)).\\

rightOf(A, B, (B, A, _, _, _)).
rightOf(A, B, (_, B, A, _, _)).
rightOf(A, B, (_, _, B, A, _)).
rightOf(A, B, (_, _, _, B, A)).\\

middleHouse(A, (_, _, A, _, _)).\\

firstHouse(A, (A, _, _, _, _)).\\

nextTo(A, B, C) :- rightOf(A, B, C).
nextTo(A, B, C) :- rightOf(B, A, C).
```

Then express the puzzle as a set of these facts, and then the question as other queries:

```
exists(house(british, _, _, _, red), Houses).
...
exists(house(WaterDrinker, _, _, water, _), Houses),
print(WaterDrinker).
```

8.2 Databases

Relational Databases allow - in a program independent way - allow us to store and query structured data. SQL (Structured Query Language) is the standard way to interact with a relational database.

They are set up over relations - eg. a compound in Prolog:

```
tName(aa2001, 'Ashwin Ahuja').
tGrade(aa2001, '1A', 2.1).
```

We can write queries in the form of rules, for example SELECT as this:

```
qName(N) :- tName(_, N) // SELECT name from tName

qGrade(F, Y, G) :- tName(C, F), tGrade(C, Y, G) // SELECT year, grade FROM tName JOIN tGrade
    ↪ where name = inputtedData
```

8.3 Countdown

Countdown is a search-based game which is an example of Generate and Test. You have to select 6 of 24 number tiles with there being the following:

Large numbers: 25, 50, 75, 100 Small numbers: 1, 2, 3, ..., 10 (two of each)

The contestant chooses how many large and small numbers and the 3-digit target is randomly chosen. The aim is to get as close as possible to using each of the 6 numbers at most once using addition, subtraction, multiplication and division (no fractions allowed).

Strategy:

1. Maintain list of symbolic arithmetic terms - initially consists of ground terms
2. If head of list is total, then succeed
3. Otherwise, pick two elements, combine using one arithmetic operation put result on head and repeat

8.3.1 Code

```
eval(plus(A, B), C) :- !, eval(A, A1), eval(B, B1), C is A1 + B1.
eval(mult(A, B), C) :- !, eval(A, A1), eval(B, B1), C is A1 * B1.
eval(A, A).

choose(N, L, R, S) - true if R is the result of choosing N items from L and S is the
    ↪ remaining items left in L

isGreater(A, B) :- eval(A, Av), eval(B, Bv), Av > Bv.

notOne(A) :- eval(A, Av), Av =\= 1.

isFactor(A, B) :- eval(A, Av), eval(B, Bv), 0 is Bv rem Av.

arithop(A, B, C) - true if C is a valid combination of A and B, eg arithop(A, B, plus(A, B))
    ↪ .

arithop(A, B, plus(A, B)).
arithop(A, B, minus(A, B)) :- isGreater(A, B).
arithop(A, B, minus(B, A)) :- isGreater(B, A).
arithop(A, B, mult(A, B)) :- notOne(A), notOne(B). // don't multiply by 1
arithop(A, B, div(A, B)) :- notOne(B), isFactor(B, A).
arithop(A, B, div(B, A)) :- notOne(A), isFactor(A, B).

countdown([Soln | _], Target, Soln) :- eval(Soln, Target)).
countdown(L, Target, Soln) :- choose(2, L, [A, B], R), arithop(A, B, C), countdown([C | R],
    ↪ Target, Soln).
```

Closest Solution: If there are no solutions, we want to find the closest solution using Iterative Deepening.

```
solve2([Soln | _], Target, Soln, D) :- eval(Soln, R), diff(Target, R, D).
solve2(L, Target, Soln, D) :- choose(2, L, [A | B], R), arithop(A, B, C), solve2([C | R],
    ↪ Target, Soln, D).

closest(L, Target, Soln, D) :- range(0, 100, D), solve2(L, Target, Soln, D).
```

8.4 Sudoku

Issue with Sudoku is that the using Generate and Test would lead to a vast, ridiculous search space.

Representation: Represent an empty square with a variable and known squares with the number. Represent the square as concatenations of each of the rows with the row just being composed of each of the squares in turn.

Initial Solution

```
range([])
range([H | T]) = :- range(1, 5, H), range(T).

diff([A, B, C, D]) :- A =\= B, A =\= C, A =\= D, B =\= C, B =\= D, C =\= D.

rows([A,B,C,D,E,F,G,H,I,J,K,L,M,N,O,P]) :- diff([A,B,C,D]), diff([E,F,G,H]), diff([I,J,K,L])
    ↪ , diff([M,N,O,P]).

cols([A,B,C,D,E,F,G,H,I,J,K,L,M,N,O,P]) :- diff([A,E,I,M]), diff([B,F,J,N]), diff([C,G,K,O])
    ↪ , diff([D,H,L,P]).

box([A,B,C,D,E,F,G,H,I,J,K,L,M,N,O,P]) :- diff([A,B,E,F]), diff([C,D,G,H]), diff([I,J,M,N]),
    ↪  diff([K,L,O,P]).

sudoku(L) :- range(L), rows(L), cols(L), box(L).
```

This clearly doesn't do things very efficiently - first call to range generates a board of all 1s. Can do better by reducing the search space.

Use list permutations - rows, columns and boxes are permutations of [1, 2, 3, 4]

Better Solution

```
diff(L) :- perm([1, 2, 3, 4], L).

rows([A,B,C,D,E,F,G,H,I,J,K,L,M,N,O,P]) :- diff([A,B,C,D]), diff([E,F,G,H]), diff([I,J,K,L])
    ↪ , diff([M,N,O,P]).

cols([A,B,C,D,E,F,G,H,I,J,K,L,M,N,O,P]) :- diff([A,E,I,M]), diff([B,F,J,N]), diff([C,G,K,O])
    ↪ , diff([D,H,L,P]).

box([A,B,C,D,E,F,G,H,I,J,K,L,M,N,O,P]) :- diff([A,B,E,F]), diff([C,D,G,H]), diff([I,J,M,N]),
    ↪  diff([K,L,O,P]).

sudoku(L) :- rows(L), cols(L), box(L).
```

This works as a Generate and Test solution but this brute-force approach is just rather slow to work and it just wouldn't work for a larger grid. In order to fix this, need to use constraint solving.

9 Constraint Solving

Prolog programs are effectively constraint satisfaction problem (limited to the single equality constraint that two terms must unify) - generalise this to include other types of constraint - Constraint Logic Programming

Idea is given:

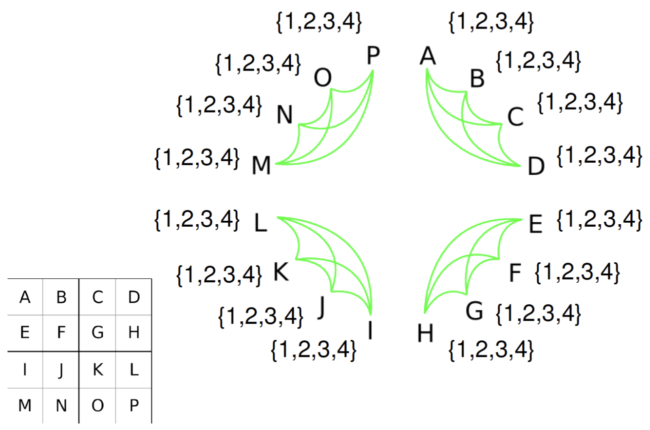
1. Set of variables *(eg the spaces)*
2. The domains of each variable *([1, ..., 9])*
3. The constraints on these variables *(row constraints, column constraints and box constraints)*

to find an assignment of values to variables satisfying the constraints.

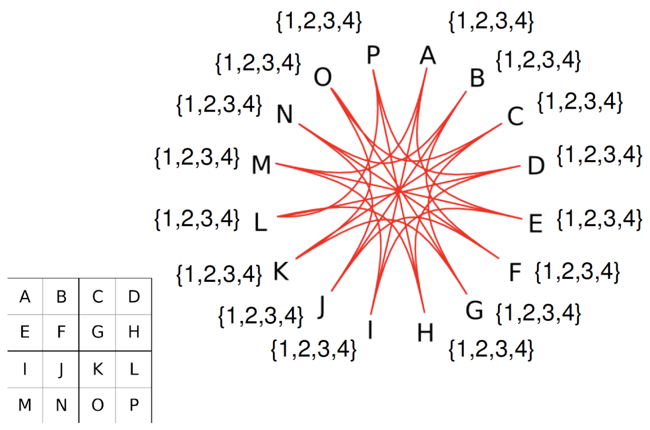
9.1 Countdown

9.1.1 Representing constraints with a graph

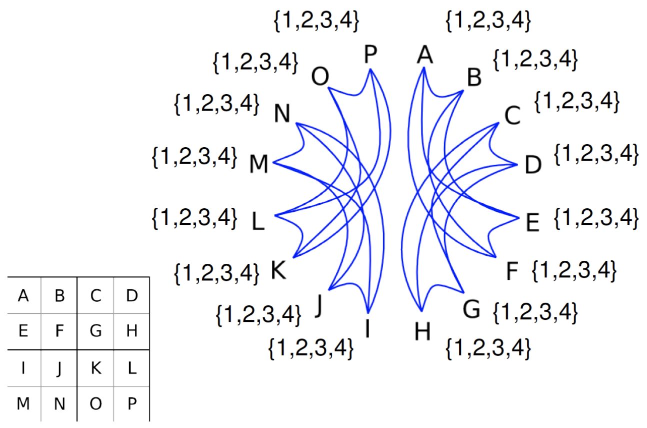
1. All variables in rows are different



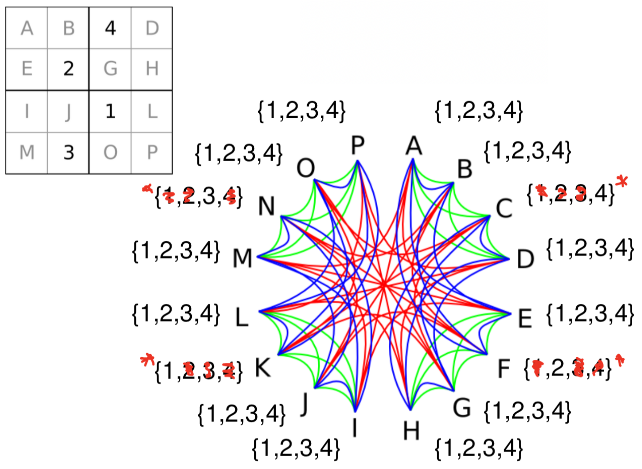
2. All variables in columns are different



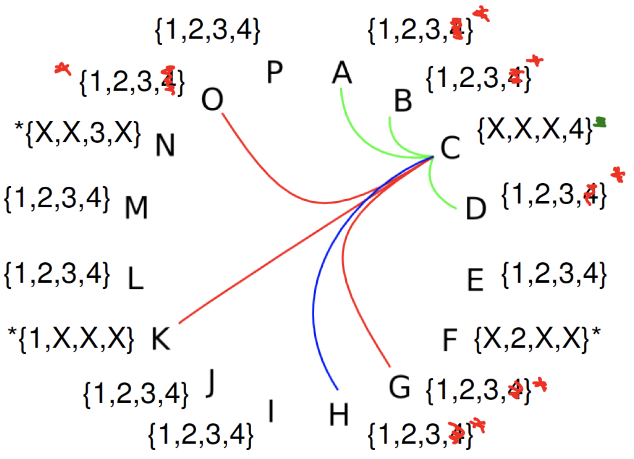
3. All variables in boxes are different



9.1.2 Reduce domains according to initial values



9.1.3 Update constraints connected to filled in variables



9.1.4 Generating solutions

Eventually, the algorithm will converge and no further changes occur

1. Single valued domains - each variable has a single possibility - there is a unique solution
2. Some empty domains - no solution to this problem
3. Multivalued domains - to find global solutions from narrowed domains, we hypothesise a solution in a domain and propagate changes and then backtrack if something goes wrong.

9.1.5 Code

```
:- use_module(library(bounds)). // easily effectively helps us to solve a CLP problem

diff(L) :- L in 1..4, all_different(L).

rows([A,B,C,D,E,F,G,H,I,J,K,L,M,N,O,P]) :- diff([A,B,C,D]), diff([E,F,G,H]), diff([I,J,K,L])
    ↪ , diff([M,N,O,P]).

cols([A,B,C,D,E,F,G,H,I,J,K,L,M,N,O,P]) :- diff([A,E,I,M]), diff([B,F,J,N]), diff([C,G,K,O])
    ↪ , diff([D,H,L,P]).

box([A,B,C,D,E,F,G,H,I,J,K,L,M,N,O,P]) :- diff([A,B,E,F]), diff([C,D,G,H]), diff([I,J,M,N]),
    ↪  diff([K,L,O,P]).
```

```
sudoku(L) :- rows(L), cols(L), box(L), label(L).
// label(L) - if we do have multivalued domains, pick one which gives us a good solution
```