1 Obtaining More Solutions

- **Backtracking** used for obtaining multiple solutions
- Results obtained one by one which meet the requirements of a single goal
- Each time we generate a new solution - we need to overwrite the previous one
  Variable *binding* to one solution - then another - and so on until *all solutions have been obtained*
- However, sometimes prefer to have all the generated objects available together - into a *List*
- This is achieved in Prolog, using *bagof* and *setof*
- Some Prolog implementations provide *findall*

```
miles(sketches_of_spain).
miles(aura).
miles(do_bop).
miles(tutu).
miles(amandala).
miles(a_kind_of_blue).

jones(back_on_the_block).
jones(right_thing).
jones(swing).

Q: miles(X)
A: X = sketches_of_spain
```

Then need to *explicitly* ask for *more solutions* to the query - in which case it fails *backtracking result* and repeats the query starting from the point where the last call was satisfied.

```
A : X = aura
```

However, if we issue the query :

```
Q: findall(X,miles(X),L)
A: X = _34 ; L = [ sketches_of_spain, aura,
 do_bop, tutu, amandala, a_kind_of_blue ]
```

1.1 findall

- findall/3
- The *List* contains all instances of the variable for which the goal has been found to be true
- Each element of the list will be a copy of the term or variable, and there will be one copy for each different solution to the query - otherwise an empty *List* is returned
- The solutions provided in *List* are NOT sorted - and appear in the same order as they do in the program
- The solutions in *List* are not necessarily *unique* - as it is possible to have *duplicate* entries in the database, it is possible to have *duplicate* entries in the solution *List*.

1.2 bagof or setof

Consider the following database :

```
age(peter,7).
age(anne,5).
age(pat,8).
age(tom,5).
```

```
Q: bagof(Child, age(Child,5), List)
A: List = [anne,tom]
```

- The *bagof* primitive is used like *findall* to obtain multiple solutions (and has the same arguments as findall)
- In the above example if the age is also left unspecified, then we obtain lists according to the ages involved

```
Q: bagof(Child, age(Child,Age), List)
A: Age = 7
   List = [peter];

   Age = 5
   List = [anne, tom];

   Age = 8
   List = [pat];
```

```
Q: bagof(Child, Age ^ age(Child,Age), List)
A: List = [peter,anne,pat,tom]

where ^ : is a predefined infix operator
```

- Essentially saying that *we do not care about the value of age, as long as a value exists*
- *bagof* fails if *goal* to satisfy fails - in this case this is *age(Child,Age)*

1.3 Removing duplicate items with setof

```
Q: setof(Child, Age ^ age(Child,Age), ChildList),
   setof(Age, Child ^ age(Child,Age), AgeList).

A: ChildList = [ anne, pat, peter, tom ]
   AgeList = [5, 7, 8]
```

- Not only is duplicate removed, but list is also sorted
- Possible to combine objects within a program in other ways

```
Q: setof(Age/Child, age(Child,Age), List)
A: List = [5/anne, 5/tom, 7/peter, 8/pat]
```

- bagof/3 is similar to setof/3 - and differs only in that the solution list is not sorted and may contain duplicates

2 Global variables and DataBase Functions

- Common practise in procedural programming to use global variables - i.e. variables accessible from the entire program
- In Prolog usually use local variables - which are restricted to the clause being used
- Therefore the need to be able to modify program (by adding variables or clauses to it)
- This is very similar to functions performed in a database
- Operations performed usually involve changing the data already present within the database - hence the need to modify the existing information
- A program in Prolog can be viewed as a database made of facts (knowledge) and operations (relations) on this knowledge
- There are predicates in Prolog which help us achieve exactly this - i.e. a program being added to or deleted from by the use of *special queries*

Achieved by the use of **retract** and **assert** and their variations.

2.1 assert

assert(Clause)

- Will add *Clause* to the current program, and will always succeed
- The *Clause* is added at the end of the clauses associated with its predicate name
- *Clause* may be a *fact* or a *rule*
- Has arity of 1

Facts should be given in the form :

```
windy
sunny
miles(new_album)
```

and rules

```
(Head :- C1, C2, _, Cn)
```

- *Note* : the rule must be enclosed in parenthesis
- Since Prolog has no global variables, *assert* must be used to save the value of a variable computed during the execution of a program

```
assert(value(x,5))

Q: value(x, Value)
A: Value = 5
```

similarly

```
Q: crisis
A: no

Q: assert(crisis)
A: yes

Q: crisis
A: yes
```

- Assert can be used as part of the program - (program modifying itself!!)
- A way *perhaps* to deal with changing situations
- One program being able to *modify* another one - even the internal structure of a program through the use of *assert* and *retract*

```
Q: assert((likes(keith,Person) :- likes(Person, jazz))).
A: yes
Q: assert(likes(james,coltrane)).
A: yes
```

The above will generate the following *definition* of likes

```
likes(keith,Person) :-
 likes(Person, jazz).

likes(james,coltrane).
```

- Assert of two types **asserta** and **assertz**
- *asserta* : Adds a clause at the beginning of the clauses associated with its predicate name

```
Q: asserta((likes(keith,Person) :- likes(Person, jazz))).
A: yes
Q: asserta(likes(james,coltrane)).
A: yes
```

The above will generate the following *definition* of likes

```
likes(james,coltrane).
likes(keith,Person) :-
 likes(Person, jazz).
```

- Similarly *assertz* : add a clause at the end of the clauses associated with its predicate name
- Something that *LPA MacProlog* offers in addition to the above two is *assertx/2* - which allows a clause to be inserted at a particular position

```
Q: assert(foo(a),0), assert(foo(b),1),
assert(foo(c),999),assert(foo(d),2)

foo(b)
foo(d)
foo(a)
foo(c)
```

- If position = 0 or greater than number of clauses acts as *assertz*

2.2 retract

- Delete the first clause that matches the given clause
- Arity of 1
- If clause does note exist - it fails

```
likes(pooh,honey).
likes(Person, Anything) :-
 nice(Anything).
likes(fred,fish).

Q: retract((likes(fred,X) :- B)).
X = _567
B = nice(_0032)
```

On backtracking, this will delete the third clause, binding the variables as follows :
$X = fish$
$Y = true$

- *retract* is non-deterministic, in that a set of clauses can be removed by a single retract clause on backtracking
- *Again* variations are possible : **retractall/1**, **retractone/1 retractx/2**
- *retractall* : delete *ALL* clauses that match the given clause head, e.g. *retractall(someone). will delete all clauses someone/0*

- *retractone* : deletes the first clause that matches the given clause head, e.g. *retractone(someone). will delete the first clause someone/0*
- *retractx* : retract a clause at a specified position, e.g. *retractx(someone/2,3). will delete the third clause for the predicate someone/2*

Combining *retract* and *assert* :

```
change( Variable, New_value) :-
 retract( value( Variable, _)),
 assert( value( Variable, New_value)).
```

and

```
fast(anne).
slow(tom).
slow(pat).

Q: assert((faster(X,Y) :- fast(X), slow(Y))).
A: yes

Q: faster(A,B).
A: A = anne   B = tom

Q: retract(slow(X)).
A: X = tom; X = pat; no

Q: faster(anne,_)
A: no
```