UCS1524 – Logic Programming

Data structures – Representation, Retrieval and Applications

Session Meta Data

Author	**Dr. D. Thenmozhi**
Reviewer	
Version Number	1.2
Release Date	4 September 2022

Session Objectives

- Understanding data structure with its representation and retrieval.
- Learn about some application using data structure.

Session Outcomes

- At the end of this session, participants will be able to
  - explain the representation and retrieval of structured data.
  - Apply data structure for complex applications

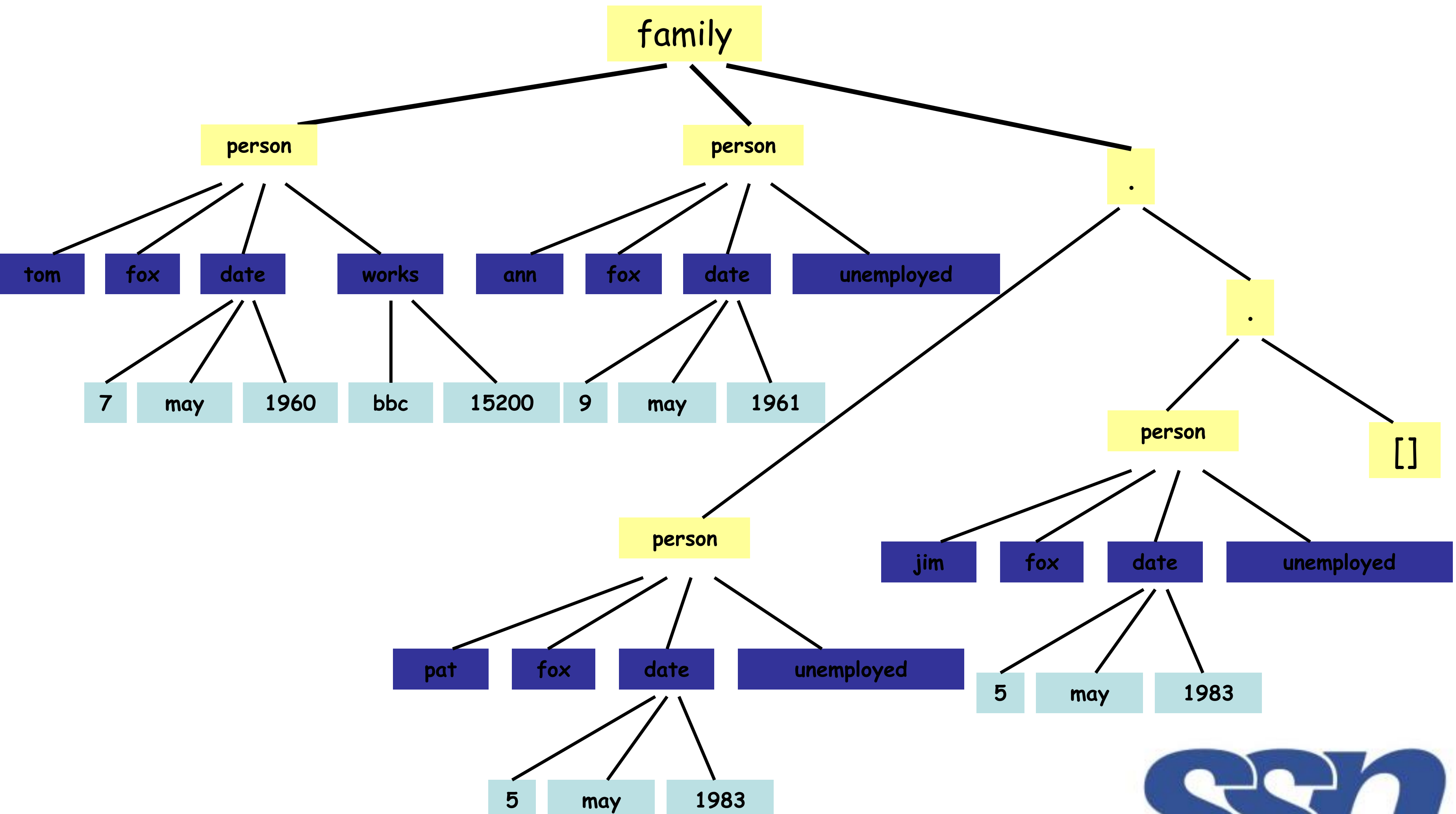
Agenda

- Retrieving structured information from a database
- Data abstraction
- Applications
  - Simulating automata
  - The eight queen problem

Retrieving structured information from a database

- The family structure:
  - Each family has three components:
    - husband,
    - wife, and
    - children.
      - The children are represented by a list.
  - Each person represented by a structure of four components:
    - name,
    - surname,
    - date of birth, and
    - job.
      - The job information is ‘unemployed’, or it specifies the working organization and salary.

Retrieving structured information from a database



Retrieving structured information from a database

- The family can be stored in the database by the clause:

```
family(
  person( tom,  fox,  date(7,may,1960),  works( bbc,  15200)),
  person( ann,  fox,  date(9,may,1961),  unemployed),
  [ person( pat,  fox,  date(5,may,1983),  unemployed),
    person( jim,  fox,  date(5,may,1983),  unemployed) ] ).
```

- How to retrieval the information from the database?

|?- family(person(Y, armstrong, _, _), _, _).

– Find all fathers of Armstrong families.

|?- family(person(_, X, _, _), _, [_, _, _]).

– Find all families with three children.

|?- family(_, person(Name, Surname, _, _), [_, _, _ |_]).

– Find all married women that have at least three children.

Retrieving structured information from a database

- These procedures can serve as a utility to make the interaction with the database more comfortable.

```
husband( X) :- family( X,  _,  _).
wife( X) :- family( _,  X,  _).
child( X) :- family( _,  _,  Children),  member( X,  Children).
exists( Persons) :- husband( Persons); wife( Persons);
                              child( Persons).
dateofbirth( person(_,  _,  Date,  _),  Date).
salary( person(_,  _,  _,  works(_,  S)),  S).
salary( person(_,  _,  _,  unemployed),  0).
```

- We can use these utilities, for example, in the following queries to the database:
  - Find the names of all the people in the database:

```
|?- exists( person( Name,  Surname,  _,  _)).
```

Retrieving structured information from a database

- We can use these utilities, for example, in the following queries to the database (con.):
  - Find all children born in 2000:

```
|?- child( X), dateofbirth( X, date( _, _, 2000)).
```

  - Find all employed wives:

```
|?- wife( person( Name, Surname, _, works(_, _))).
```

  - Find the names of unemployed people who were born before 1973:

```
|?- exists( person( Name, Surname, date(_, _, Year),
    unemployed)), Year < 1973.
```

  - Find people born before 1960 whose salary is less than 8000:

```
|?- exists( Person),
    dateofbirth( Person, date(_, _, Year)), Year < 1960,
    salary( Person, Salary), Salary < 8000.
```

  - Find the names of families with at least three children:

```
|?- family( person( _, Name, _, _), _, [_, _, _| _]).
```

Retrieving structured information from a database

- To calculate the total income of family it is useful to define the sum of salaries of a list of people as a two-argument relation:

  total(List_of_people, Sum_of_their_salaries).

- This relation can be programmed as:

```
total( [], 0).
total( [Person |List], Sum) :-
    salary( Person, S), total( List, Rest), Sum is S + Rest.
```

- The total income of families can then be found:

```
|?- family( Husband, Wife, Children),
   total( [Husband, Wife | Children], Income).
```

Retrieving structured information from a database

- Let the **length** relation count the number of elements of a list, as defined in Section 3.4. Then we can specify all families that have an income per family member of less then 2000:

```
|?- family( Husband, Wife, Children),
    total( [Husband, Wife | Children], Income),
    length( [Husband, Wife | Children], N),
    Income/N < 2000.
```

Doing data abstraction

- Data abstraction can make the use of information possible without the programmer having to think about the details of how the information is actually represented.

- In the previous section, each family was represented by a Prolog clause:

```
family(
   person( tom,  fox,  date(7,may,1960),  works( bbc,  15200)),
   person( ann,  fox,  date(9,may,1961),  unemployed),
   [ person( pat,  fox,  date(5,may,1983),  unemployed),
     person( jim,  fox,  date(5,may,1983),  unemployed) ] ).
```

- Here, a family will be represented as a structured object, for example:

```
FoxFamily = family( person( tom,  fox,  _,  _),  _,  _)
```

Doing data abstraction

- Let us define some relations through which the user can access particular components of a family without knowing the details of Figure 4.1:

selector_relation(Object, Component_selected)

- Here are some selectors for the family structure:

husband(family(Husband, _, _), Husband).
wife(family(_, Wife, _), Wife).
children(family(_, _, ChildList), ChildList).

Thus, we can also define selectors for particular children:
firstchild(Family, First):- children(Family, [First | _]).

Doing data abstraction

- More selectors for particular children:

```
firstchild( Family,  First):- children( Family,  [First | _]).

secondchild( Family,  Second):-
                    children( Family,  [_,  Second | _]).
…
nthchild( N,  Family,  Child) :-
                    children( Family,  ChildList),
                    nth_member( N,  ChildList,  Child).
```

- Some related selectors of persons:

```
firstname( person( Name,  _,  _,  _),  Name).
surname( person( _,  Surname,  _,  _),  Surname).
born( person( _,  _,  Date,  _),  Date).
```

Doing data abstraction

- How can we benefit from selector relations?
  - We can forget about the particular way that structured information is represented.
    - For example, the user does not have to know that the children are represented as a list.
      - Assume that we want to say that

        Tom Fox and Jim Fox belong to the same family and that Jim is the second child of Tom.

      - Using the selector relations above, we can define two persons, call them **Person1** and **Person2**, and the **family**.

firstname(Person1, tom), surname(Person1, fox),
firstname(Person2, jim), surname(Person2, fox),
husband(Family, Person1),
secondchild(Family, Person2)

Doing data abstraction

- As a result, the variables **Person1**, **Person2**, and **Family** are instantiated as:

  Person1 = person(tom, fox, _, _)

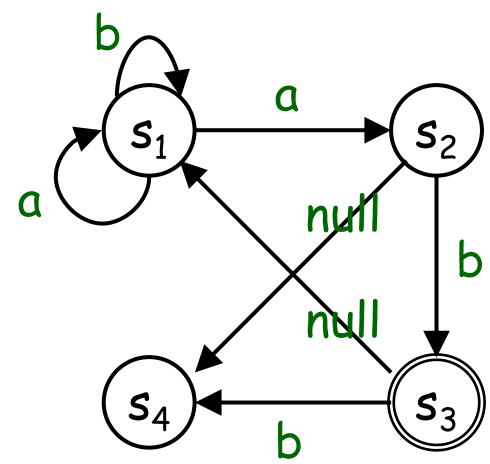
  Person2 = person(jim, fox, _, _)

  Family = family(person(fom, fox, _, _), _, [_, person(jim, fox)|_])

- The use of selector relations also make programs easier to modify.

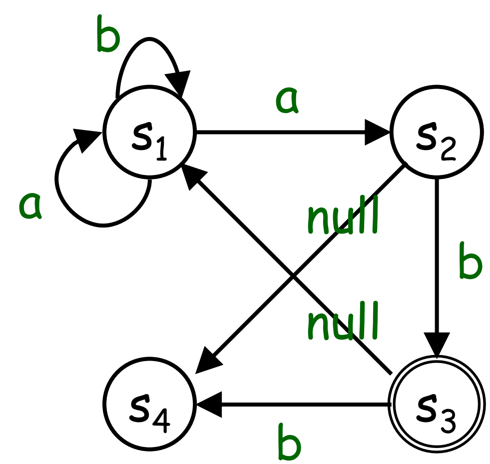
Simulating a non-deterministic automaton

- A non-deterministic finite automaton is an abstract machine that reads as input a string of symbols and decides whether to accept or to reject the input string.
  - An automaton has a number of states and it is always in one of the states.
  - It can change its state by moving from the current state to another state.
  - For example:
    - States: {s_1, s_2, s_3, s_4}.
    - Initial state: s_1.
    - Final state: s_3.
    - Symbols: {a, b}.
    - null (null symbol)
    - There arcs labeled null correspond to silent moves of the automaton. The move occurs without any reading of input.



Simulating a non-deterministic automaton

- The automaton is said to accept the input string if there is a transition path in the graph such that
  - (1) It starts with the initial state,
  - (2) It ends with a final state, and
  - (3) The arc labels along the path correspond to the complete input string.



- For example:
  - The automaton will accept the strings ab and aabaab.
  - It will reject the strings abb and abba.
  - In fact, this automaton accepts any string that terminates with ab, and rejects all others.

Simulating a non-deterministic automaton

- In Prolog, an automaton can be specified by three relations:

  (1) A unary relation **final** defines the final states of the automaton.

  final(F)

  means F is a final state.

  (2) A three-argument relation **trans** which defines the state transitions so that

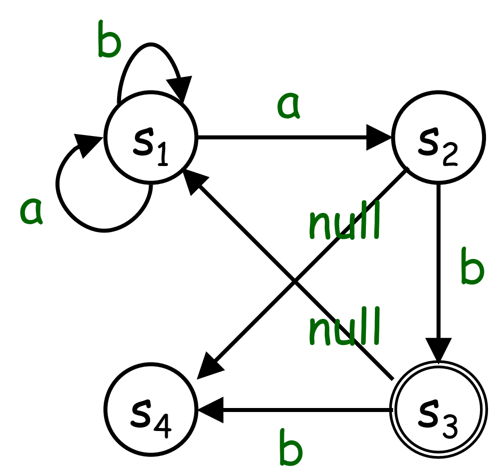
  trans(S1, X, S2)

  means that a transition from a state S1 to S2 is possible when the current input symbol X is read.

  (3) A binary relation

  silent(S1, S2)

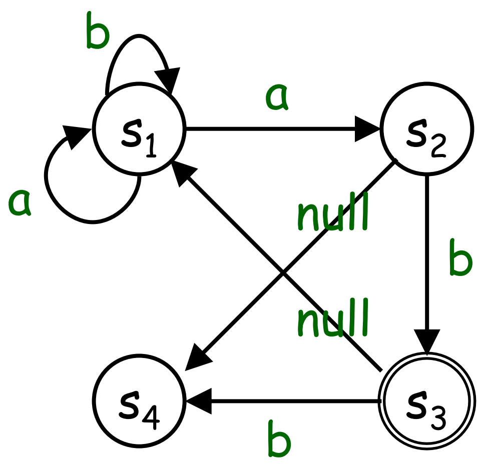
  meanings that a silent move is possible from S1 to S2.



Simulating a non-deterministic automaton

- For example:

  final(s3).
  trans(s1, a, s1).
  trans(s1, a, s2).
  trans(s1, b, s1).
  trans(s2, b, s3).
  trans(s3, b, s4).
  silent(s2, s4).
  silent(s3, s1).



v 1.2

Simulating a non-deterministic automaton

- Represent input strings as Prolog list.

  For example, [a, a, a, b].

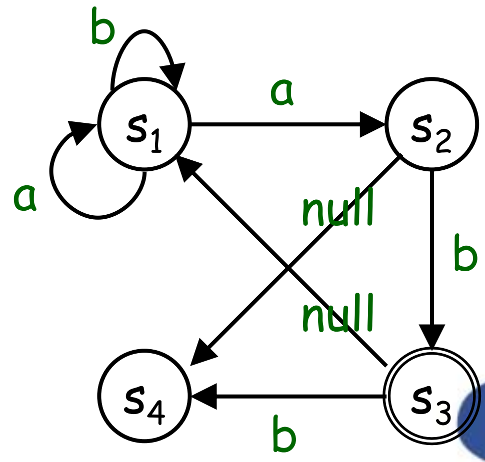
- Define the acceptance of a string from a given state:

  accepts(State, String)

  The binary relation **accepts** is true if the automaton, starting from the state **State** as initial state, accepts the string **String**.

For example:

| ?- accepts(s1, [a, a, a, b]).
true ?
yes



v 1.2

Simulating a non-deterministic automaton

- The accepts relation can be define by three clauses:

(1) The empty string, [], is accepted from a state **State** if **State** is a final state.

accepts(State, []):- final(State).

(2) A non-empty string is accepted from **State** if reading the first symbol in the string can bring the automaton into some state **State1**, and the rest of the string is accepted from **State1**.

accepts(State, [X| Rest]):-
trans(State, X, State1),
accepts(State1, Rest).

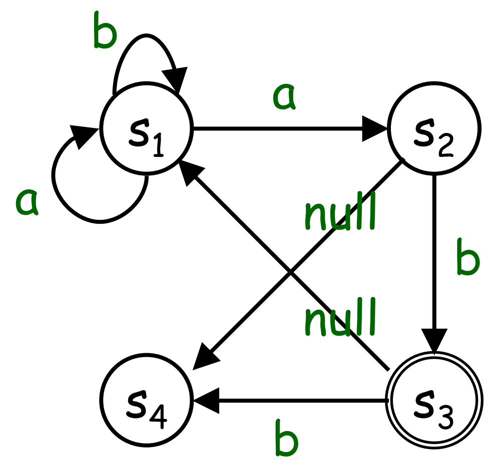
(3) A string is accepted from **State** if the automaton can make a silent move from **State** to **State1** and then accept the (whole) input string from **State1**.

accepts(State, String):-
silent(State, State1), accepts(State1, String).

Simulating a non-deterministic automaton

- For example:

```
| ?- accepts( s1,  [a,  a,  a,  b]).
true ?
yes

| ?- accepts( S,  [a,  b]).
S = s1 ? ;
S = s3 ? ;
no

| ?- accepts( s1,  [X1,  X2,  X3]).

X1 = a
X2 = a
X3 = b ? ;
X1 = b
X2 = a
X3 = b ? ;
no
```




```
| ?- String=[_,_,_,_],  accepts( s1,
     String).

String = [a,a,a,b] ? ;
String = [a,b,a,b] ? ;
String = [a,b,a,b] ? ;
String = [b,a,a,b] ? ;
String = [b,b,a,b] ? ;
no
```

The eight queens problem

- The eight queens problem:
  - The problem here is to place eight queens on the empty chessboard in such a way that no queen attacks any other queen.
  - The solution will be programmed as a unary predicate

    Solution(Pos)

    which is true if and only if **Pos** represents a position with eight queens that do not attack each other.
  - In this section, we will present three programs based on somewhat different representations on the problem.

The eight queens problem— Program 1

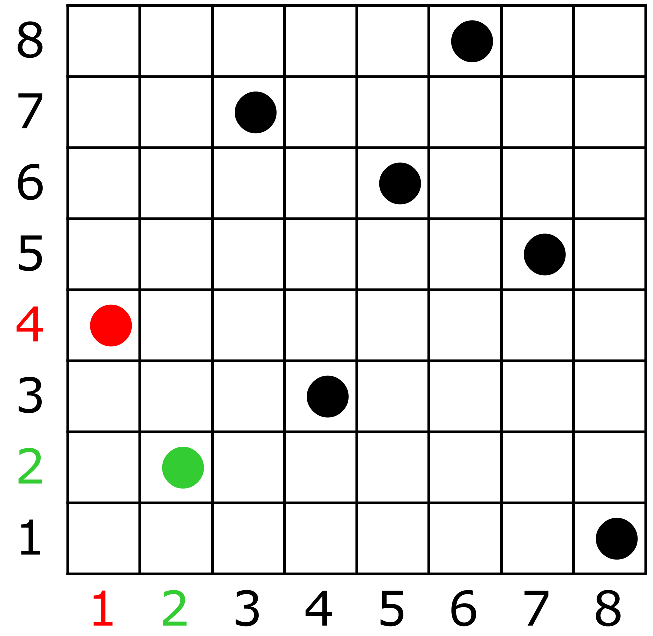
- Figure 4.6 shows one solution of the eight queens problem. And the list representation of solution is:

  [1/4, 2/2, 3/7, 4/3, 5/6, 6/8, 7/5, 8/1]
- In the program, we choose the representation of the board position:

[X1/Y1, X2/Y2, …, X8/Y8]

- We also can fix the X-coordinates so that the solution list will fit the following template:

[1/Y1, 2/Y2, …, 8/Y8]



The eight queens problem— Program 1

- The **solution** relation can be formulated by considering two cases:
  - **Case 1:** The list of queens is empty.
    - The empty list is certainly a solution because there is no attack.
  - **Case 2:** The list of queens is non-empty.
    - Then it looks like this:

      [X/Y| Others]

      (1) There must be no attack between the queens in the list **Others**; that is, **Others** itself must also be a solution.

      (2) **X** and **Y** must be integers between 1 and 8.

      (3) A queen at square **X/Y** must not attack any of the queens in the list **Others**.

```
solution([X/Y|Others]) :-
   solution( Others),  member( Y,  [1,2,3,4,5,6,7,8]),     noattack( X/Y,
   Others).
```

The eight queens problem— Program 1

- Now define the **noattack** relation:

  noattack(Q, Qlist)

  – **Case 1:**
    - If the list **Qlist** is empty then the relation is certainly true because there is no queen to be attacked.

      noattack(_, []).

  – **Case 2:**
    - If **Qlist** is not empty then it has the form **[Q1|Qlist1]** and two conditions must be satisfied:

      (1) the queen at **Q** must not attack the queen at **Q1**, and

      (2) the queen at **Q** must not attack any of the queens in **Qlist1**.
    - To specify that a queen at some square does not attack another square is easy: the two squares must not be in the same row, the same column or the same diagonal.

The eight queens problem— Program 1

- Since the two squares must not be in the same row, the same column or the same diagonal, so
  - The Y-coordinates of the queens are different, and
  - They are not in the same diagonal, either upward or downward; that is, the distance between the squares in the X-direction must not be equal to that in the Y-direction.

```
noattack( _,  [] ).                  % Nothing to attack

noattack( X/Y,  [X1/Y1 | Others] )  :-
  Y =\= Y1,                          % Different Y-coordinates
  Y1-Y =\= X1-X,                     % Different diagonals
  Y1-Y =\= X-X1,
  noattack( X/Y,  Others).
```

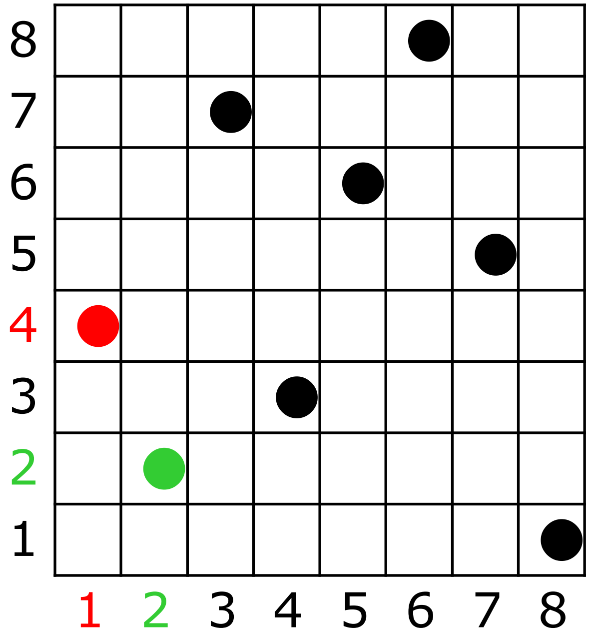
The eight queens problem— Program 1

```
% Figure 4.7  Program 1 for the eight queens problem.

solution( [] ).
solution( [X/Y | Others] )  :-
   solution( Others),  member1( Y,  [1,2,3,4,5,6,7,8] ),
   noattack( X/Y,  Others).

noattack( _,  [] ).
noattack( X/Y,  [X1/Y1 | Others] )  :-
   Y =\= Y1,   Y1-Y =\= X1-X,  Y1-Y =\= X-X1,
   noattack( X/Y,  Others).

member1( Item,  [Item | Rest] ).
member1( Item,  [First | Rest] )  :-  member1( Item,  Rest).

% A solution template

template( [1/Y1,2/Y2,3/Y3,4/Y4,5/Y5,6/Y6,7/Y7,8/Y8] ).
```

The eight queens problem— Program 1

```
template( [1/Y1,2/Y2,3/Y3,4/Y4,5/Y5,6/Y6,7/Y7,8/Y8] ).
```

```
| ?- template( S), solution( S).

S = [1/4, 2/2, 3/7, 4/3, 5/6, 6/8, 7/5, 8/1] ? ;
S = [1/5, 2/2, 3/4, 4/7, 5/3, 6/8, 7/6, 8/1] ? ;
S = [1/3, 2/5, 3/2, 4/8, 5/6, 6/4, 7/7, 8/1] ? ;
S = [1/3, 2/6, 3/4, 4/2, 5/8, 6/5, 7/7, 8/1] ? ;
S = [1/5, 2/7, 3/1, 4/3, 5/8, 6/6, 7/4, 8/2] ? ;
S = [1/4, 2/6, 3/8, 4/3, 5/1, 6/7, 7/5, 8/2] ? ;
S = [1/3, 2/6, 3/8, 4/1, 5/4, 6/7, 7/5, 8/2] ? ;
S = [1/5, 2/3, 3/8, 4/4, 5/7, 6/1, 7/6, 8/2] ? ;
S = [1/5, 2/7, 3/4, 4/1, 5/3, 6/8, 7/6, 8/2] ? ;
S = [1/4, 2/1, 3/5, 4/8, 5/6, 6/3, 7/7, 8/2] ? ;
S = [1/3, 2/6, 3/4, 4/1, 5/8, 6/5, 7/7, 8/2] ? ;
S = [1/4, 2/7, 3/5, 4/3, 5/1, 6/6, 7/8, 8/2] ? ;
S = [1/6, 2/4, 3/2, 4/8, 5/5, 6/7, 7/1, 8/3] ? ;
S = [1/6, 2/4, 3/7, 4/1, 5/8, 6/2, 7/5, 8/3] ? ;
S = [1/1, 2/7, 3/4, 4/6, 5/8, 6/2, 7/5, 8/3] ? …
```



The eight queens problem— Program 1

```
| ?- solution([1/3,2/5,3/2,4/8,5/6,6/4,7/7,8/1]).
true ?
yes

| ?- solution([1/3,2/5,3/2,4/8,5/6,6/4,7/7,8/8]).
no

| ?- solution([7/3,7/5,7/2,7/8,7/6,7/4,7/7,7/1]).
true ?
yes
```

Why?

The eight queens problem— Program 2

- In the program, we choose the representation of the board position:
  [Y1, Y2, …, Y8]
  - No information is lost if the X-coordinates were omitted.
  - Each solution is therefore represented by a permutation of the list **[1, 2, 3, 4, 5, 6, 7, 8]**
  - Such a permutation, **S**, is a solution if all the queens are safe.

```
solution( S):-
  permutation([1,2,3,4,5,6,7,8], S), safe( S).
```

The eight queens problem— Program 2

- Define the **safe** relation:

  (1) **S** is the empty list.

  - This is certainly safe as there is nothing to be attacked.

  safe([]).

  (2) **S** is a non-empty list of the form **[Queen|Others].**

  - This is safe if the list **Others** is safe, and **Queen** does not attack any queen in the list **Others**.

```
safe( [Queen | Others] )  :-
   safe( Others), noattack( Queen,  Others).
```

The eight queens problem— Program 2

- The goal

  noattack(Queen, Others)

  is meant to ensure that **Queen** does not attack **Others** when the X-distance between **Queen** and **Others** is equal to 1.

- We add this distance as the third argument of the **noattack** relation.

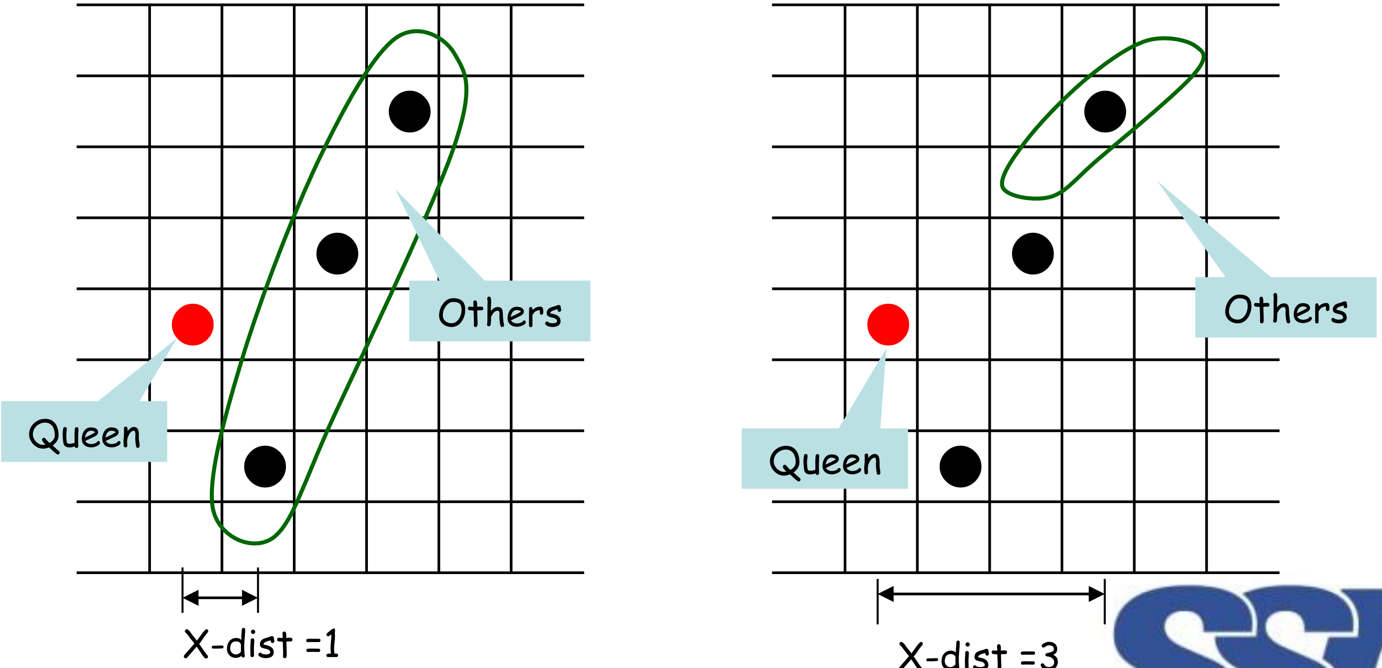
  noattack(Queen, Others, Xdist)

```
noattack( _, [], _).
noattack( Y, [Y1 | Ylist], Xdist) :-
  Y1-Y =\= Xdist, Y-Y1 =\= Xdist, Dist1 is Xdist + 1,
  noattack( Y, Ylist, Dist1).
```

The eight queens problem— Program 2

- The **noattack** goal in the **safe** relation has to be modified to

noattack(Queen, Others, 1)



The eight queens problem— Program 2

```
% Figure 4.9   Program 2 for the eight queens problem.

solution( Queens)  :-
  permutation( [1,2,3,4,5,6,7,8],  Queens),  safe( Queens).

permutation( [],  [] ).
permutation( [Head | Tail],  PermList)  :-
  permutation( Tail,  PermTail),  del( Head,  PermList,  PermTail).

del( Item,  [Item | List],  List).
del( Item,  [First | List],  [First | List1] )  :-
  del( Item,  List,  List1).

safe( [] ).
safe( [Queen | Others] ) :- safe( Others), noattack( Queen, Others, 1).

noattack( _,  [],  _).
noattack( Y,  [Y1 | Ylist],  Xdist)  :-
  Y1-Y =\= Xdist,  Y-Y1 =\= Xdist,  Dist1 is Xdist + 1,
  noattack( Y,  Ylist,  Dist1).
```

The eight queens problem— Program 2

```
| ?- solution( S).

S = [5,2,6,1,7,4,8,3] ? ;
S = [6,3,5,7,1,4,2,8] ? ;
S = [6,4,7,1,3,5,2,8] ? ;
S = [3,6,2,7,5,1,8,4] ? ;
S = [6,3,1,7,5,8,2,4] ? ;
S = [6,2,7,1,3,5,8,4] ? ;
S = [6,4,7,1,8,2,5,3] ? ;
S = [3,6,2,7,1,4,8,5] ? ;
S = [6,3,7,2,4,8,1,5] ? ;
S = [6,3,7,4,1,8,2,5] ? ;
S = [2,6,1,7,4,8,3,5] ? ;
S = [6,2,7,1,4,8,5,3] ? ;
S = [6,3,7,2,8,5,1,4] ? ;
S = [5,7,2,6,3,1,4,8] ? ;…
```

The eight queens problem— Program 2

```
| ?- solution([5,2,6,1,7,4,8,3]).
true ?
yes

| ?- solution([5,2,6,1,7,4,8,X]).
X = 3 ?
yes

| ?- solution([5,2,6,1,7,Z,Y,X]).
X = 3
Y = 8
Z = 4 ? ;
no

| ?- solution([5,2,6,1,7,7,Y,X]).
no
```

Summary

- Retrieving structured information from a database
  - Representing
  - Querying
- Data abstraction
- Applications
  - Simulating automata
  - The eight queen problem
    - 3 variations

Check your understanding

- Write queries to find the following from the family database:
  - Names of families without children;
  - All employed children;
  - Names of families with employed wives and unemployed husbands;
  - All the children whose parents differ in age by at least 15 years.

Check your understanding

- If we write a query to find people born before 1960 whose salary is less than 8000 as follows. Is it a correct query? If not, please correct this query.

  | ?- dateofbirth(Person, date(_, _, Year)), Year < 1960, salary(Person, Salary), Salary < 8000.

- Define the relation

  twins(Child1, Child2)

  to find twins in the family database.

Check your understanding

- Complete the definition of **nthchild** by defining the relation
  nth_member(N, List, X)
  which is true if X is the Nth member of List.

```
nthchild( N, Family, Child) :-
        children( Family, ChildList),
        nth_member( N, ChildList, Child).
```

Check your understanding

- What kind of strings can be accepted by this automaton?
- Please write a Prolog program and test it.