St. Joseph’s College of Engineering & Technology Palai

Department of Computer Science & Engineering

S8 CS

RT804 Artificial Intelligence

Module 5

Website: http://sites.google.com/site/sjcetcssz

Artificial Intelligence - Module 5

Syllabus

Introduction to Prolog – Representing facts – Recursive search – Abstract data types – Alternative search strategies – Meta predicates, Matching and evaluation, meta interpreters – semantic nets & frames in prolog.

Contents

Part I. Introduction to Prolog

[Luger2005]

Prolog is a logic programming language. It has the following features.

They are

Declarative semantics, a means of directly expressing problem relationships in AI,

Built in unification,

High powered techniques for pattern matching and searching.

1 Syntax of Prolog

Representing facts and rules

English	Predicate Calculus	Prolog
and	$\cap$	,
or	$\cup$	;
not	$\neg$	not
$only\, if$	$\leftarrow$	$:-$

Predicate names are expressed as a sequence of alphanumeric characters.

Variables are represented as a string of characters beginning with an uppercase alphabet.

The fact ‘ Everyone likes apple’ is represented in Prolog as

likes (X, apple)

The fact ‘George and Bill like some set of people’ is represented in Prolog as

likes (george, Y), likes(bill, Y)

‘George likes apple and George likes orange’ is represented as

likes (george, apple), likes (george, orange)

‘George likes apple or George likes orange’ is represented as likes (george, apple) ; likes (george, orange)

‘If George does not like apple, then George likes orange’ is represented as

likes(george, orange) :- not (likes (george, apple))

These examples show how the predicate calculus connectives $\bigcap$, U, $\neg$ and $\leftarrow$ are expressed in Prolog.

2 Prolog Database

A Prolog program is a set of specifications in the first order predicate calculus describing the objects and relations in the problem domain. The set of specifications is referred to as the database for the problem.

If we wish to describe a world consisting of George’s, Kate’s and Susie’s likes and dislikes, the database might contain the following set of predicates.

```
likes(george,apple)
likes(george,orange)
likes(george,wine)
likes( bill ,wine)
likes(ken,gin)
likes(ken,orange)
```

We may ask questions to the Prolog interpreter as follows:

```
?-    likes (george, apple)
yes

?-    likes (ken, orange)
yes

?-    likes (george, X)
X = apple
;
X = orange
;
X = wine
;
no
```

```
?-	likes (george, beer)
no
```

In the request, likes (george, X), successive user prompts (;) cause the interpreter to return all the terms in the database specification that may be substituted for the X in the query. They are returned in the order they are found in the database.

Suppose we add the following predicate to the database,

```
friends(X,Y) :- likes(X,Z),likes(Y,Z)
```

This means

‘X and Y are friends; if there exists a Z such that X likes Z and Y likes Z’.

If we ask the following question to the interpreter

```
?-	friends (george, bill)
yes
```

To solve the question, Prolog searches the database. The query ‘friends(george, bill)’ is matched with the conclusion of the rule

‘friends (X, Y):- likes (X, Z) , likes (Y, Z)’,

with X as ‘george’ and Y as ‘bill’.

The interpreter looks for a Z such that

‘likes (george, Z)’ is true.

It finds the predicate ’likes(george,apple)’.

The interpreter then tries to determine whether

’likes (bill, apple)’ is true. When it is found to be false, the value ’apple’ for Z is rejected. The interpreter then backtracks to find a second value for Z in

‘likes (george, Z)’.

‘likes (george, Z)’ then matches the second clause in the database, with Z bound to ‘orange’. The interpreter then tries to match ‘likes (bill, orange)’. When this also fails, the interpreter goes back to the database for yet another value for Z.

This time ’wine’ is found in the third predicate, and the interpreter goes on to show that

‘likes (bill, wine)’

is true.

In this, ‘wine’ is the binding that ties ‘george’ and ‘bill’.

Prolog tries to match goals with patterns in the order in which the patterns are entered in the database.

.

What is inference? Explain. (4 marks) [MGU/May2008]

With an example explain how rules are represented in Prolog. (4 marks) [MGU/July2007]

With an example, explain how facts are represented in Prolog. (4 marks) [MGU/June2006]

With suitable examples explain how facts are represented in Prolog.

3 Creating and Changing the Prolog environment

In creating a Prolog program, the database of specifications is created first. The predicate ‘assert’ adds new predicates to Prolog database.

```
?-      assert ( likes (david, susie) )
```

adds this predicate to the Prolog database.

Now the query

```
?-      likes (david, susie)
X = susie
```

is returned.

‘asserta (P)’ adds the predicate P at the beginning of all the predicates P, and

‘assertz (P)’ adds the predicate P at the end of all the predicates named P.

There are also other predicates such as consult, read, write, see, tell, seen, told, listing, trace, spy for monitoring the Prolog environment.

4 Lists in Prolog

The list is a structure consisting of a set of elements. Examples of Prolog lists are

```
[1, 2, 3, 4]
[tim, bill, harry, fred]
[ [george, kate], [allen,amy], [don, pat] ]
[ ]
```

The first elements of a list may be separated from the list by the ‘|’ operator.

For instance, when the list is

```
[tim, bill, harry, fred ]
```

The first element of the list is

‘tim’ and

the tail of the list is

‘[bill, harry, fred]’.

Using vertical bar operator and unification, we can break a list in to its components.

If [tim, bill, harry, fred] is matched to [X | Y], then X = tim and Y = [bill, harry, fred].

If [tim, bill, harry, fred] is matched to [X, Y | Z], then X = tim, Y = bill, and Z = [harry, fred].

If [tim, bill, harry, fred] is matched to [X, Y, Z | W], then X = tim, Y = bill, and Z = harry and W = [fred].

If [tim, bill, harry, fred] is matched to [W, X, Y, Z | V], then W = tim, X = bill, Y = harry, Z = [fred] and V = [ ].

[tim, bill, harry, fred] will not match [V, W, X, Y, Z | U].

[tim, bill, harry, fred] will match [tim, X | [harry, fred]], to give X = bill.

5 Recursion in Prolog

Member check

The predicate member is used to check whether an item is present in a list. This predicate ‘member’ takes 2 arguments, an element and a list, and returns true if the element is a member of the list.

```
?-    member ( a, [a, b, c, d] )
yes

?-    member ( a, [1, 2, 3, 4] )
no

?-    member ( X, [a, b, c] )
X = a
;
X = b
;
```

```
X = c
;
no
```

See how this predicate works.

To define member recursively, we first test if X is the first item in the list.

```
member ( X, [ X | T])
```

This tests whether X and the first element of the list are identical. If they are not, then check whether X is an element of the rest of the list. This is defined by:

```
member ( X, [Y | T]) :- member (X, T)
```

Thus the 2 lines of Prolog for checking list membership are then

```
member ( X, [ X | T])
member ( X, [Y | T]) :- member (X, T)
```

Let us trace member (c, [a, b, c]) as follows

```
1. member ( X, [ X | T])
2. member ( X, [Y | T]) :- member (X, T)
```

```
?-    member (c, [a, b, c])
            call 1. fail, since c ≠ a
            call 2. X = c, Y = a, T = [b, c], member (c, [b, c]) ?
                call 1. fail, since c ≠ b
                call 2. X = c, Y = b, T = [c], member (c, [c]) ?
                    call 1. success, c = c
                yes (to second call 2)
            yes (to first call 2)
      yes
```

The use of cut to control search in Prolog

The cut is denoted by an exclamation symbol, !.

The syntax for cut is that of a goal with no arguments. For a simple example of the effect of the cut, see the following.

Let we have the following predicates in the prolog database.

```
path2 (X, Y) :- move (X, Z), move (Z, Y)
move (1, 6)
```

```
    move (1, 8)
    move (6, 7)
    move (6, 1)
    move (8, 3)
    move (8, 1)
```

Let the prolog interpreter is asked to find all the two move paths from 1; there are 4 answers as shown below.

```
?-      path2 (1, W)
W= 7
;
W = 1
;
W = 3
;
W = 1
;
no
```

When path2 is altered with cut, only 2 answers result.

```
    path2 (X, Y) :- move (X, Z), ! , move (Z, Y)
    move (1, 6)
    move (1, 8)
    move (6, 7)
    move (6, 1)
    move (8, 3)
    move (8, 1)
```

```
?-      path2 ( 1, W)
W = 7
;
W= 1
;
no
```

This happens because variable Z takes on only one value namely 6. Once the first sub goal succeeds, Z is bound to 6 and the cut is encountered. This prohibits further backtracking to the first sub goal and no further bindings for Z.

Thus cut has several side effects.

First, when originally encountered it always succeeds, and

Second, if it is failed back to in backtracking, it causes the entire goal in which it is contained to fail.

There are several uses for the cut in programming.

First, as shown in this example, it allows the programmer to control the shape of the search tree. When further search is not required, the tree can be explicitly pruned at that point.

Second, cut can be used to control recursion.

.

Write notes on i) Recursive search in Prolog. ii) Abstract data types. (12 marks) [MGU/May2008]

Describe recursive search in Prolog.

Explain how recursive searcch is carried out in Prolog with an example. (12 marks) [MGU/Jan2007]

What is meant by recursive search? (4 marks) [MGU/Nov2010]

Part II. Abstract Data Types in Prolog

We will build the following data structures in Prolog.

Stack,

Queue and

Priority queue.

6 Stack

A stack is a Last in First out data structure. All elements are pushed on to the top of the stack. Elements are popped from the top of the stack. The operators that we define for a stack are

1. Test	check whether the stack is empty.
2. Push	inserts an element on to the stack.
3. Pop	removes the top element from the stack.
4. Member_stack	which checks whether an element is in the stack.
5. Add_list	which adds a list of elements to the stack.

We now build these operators in prolog.

1. empty_stack ([ ])

This predicate can be used to test a stack to see whether it is empty or to generate a new empty stack.

2. stack (Top, Stack, [top | Stack])

This predicate performs the push and pop operations depending on the variable bindings of its arguments.

Push produces a new stack as the 3rd argument when the first 2 arguments are bound.

Pop produces the top element of the stack when the 3rd argument is bound to the stack. The 2nd argument will then be bound to the new stack, once the top element is popped.

3. member_stack(Element, Stack) :- member (Element, Stack)

This allows us to determine whether an element is a member of the stack

4. add_list_to_stack (List, Stack, Result) : - append (List, Stack, Result)

List is added to Stack to produce Result, a new stack. (The predicate append adds Stack to the end of List and produces the new list result.)

7 Queue

A queue is a first in first out data structure. Here elements are inserted at the rear end and removed from the front end. The operations that we define for a queue are

1. empty_queue ([ ])

This predicate either tests whether a queue is empty or initializes a new empty queue.

2. enqueue (E, [ ], [E])

enqueue (E, [H | T], [H | T_{new}]) :- enqueue (E, T, T_{new})

This predicate adds the element E to a queue, the second argument. The new augmented queue is the third argument.

3. dequeue (E, [E | T], T)

This predicate produces a new queue, the third argument that is the result of taking the next element, the first argument, off the original queue, the second argument.

4. dequeue (E, [E | T], _)

This predicate lets us peek at the next element, E, of the queue.

5. member_queue (Element, Queue) :- member (Element, Queue)

This tests whether Element is a member of Queue.

6. add_list_to_queue (List, Queue, Newqueue) :- append (Queue, List, Newqueue)

This adds the list List to the end of the queue Queue.

8 Priority Queue

A priority queue orders the elements of a queue so that each new item to the priority queue is placed in its sorted order. The dequeue operator removes the best element from the priority queue. The operations that we define for a priority queue are

1. empty_queue ([ ])

This predicate either tests whether a queue is empty or initializes a new empty queue.

2. dequeue (E, [E | T], T)

This predicate produces a new queue, the third argument that is the result of taking the next element, the first argument, off the original queue, the second argument.

3. dequeue (E, [E | T], _)

This predicate lets us peek at the next element, E, of the queue.

4. member_queue (Element, Queue) :- member (Element, Queue)

This tests whether Element is a member of Queue.

5. insert_pq (State, [ ], [State]) :- !

insert_pq (State, [H | Tail], [State, H | tail]) :- precedes (State, H)

insert_pq (State, [H | T], [H | Tnew]) :- insert_pq (State, T, Tnew)

precedes (X, Y) :- X < Y

insert_pq operation inserts an element to a priority queue in sorted order.

.

What are abstract data types. (4 marks) [MGU/Jan2007]

Part III. Searching Strategies

In this section, depth first search, Breadth first search and Best first search algorithms are implemented in Prolog.

9 Depth first search in Prolog

The following was the algorithm we learned for Depth first search.

```
void depth_first_search ()
{
    open = [start];
    closed = [ ];
    while (open not empty )
    {
        remove leftmost state from open, call it X;
        if X is a goal state
             then return SUCCESS;
        else
        {
             generate children of X;
             put X on closed;
             discard children of X, if already on open or closed;
             put remaining children on left end of open;
        }
    }
    return FAIL;
}
```

The following is the Prolog program corresponding to this for depth first search..

```
go ( Start, Goal):-
     empty_stack (Empty_open),
     stack ( [Start, nil] ), Empty_open, Open_stack),
     empty_set (Closed_set),
     path (Open_stack, Closed_set, Goal).


path (Open_stack, _, _ ) :-
     empty_stack (Open_stack),
     write (No solution found with these rules).
path (Open_stack, Closed_set, Goal) :-
     stack ( [State, Parent], _, Open_stack), State = Goal,
     write ( A solution is found) , nl,
     printsolution ( [State, Parent], Closed_set).
path (Open_stack, Closed_set, Goal) :-
     stack( [State, Parent], Rest_open_stack, Open_stack),
     get_children( State, Rest_open_stack, Closed_set, Children)
     add_list_to_stack ( Children, Rest_open_stack, New_open_stack),
     union ( [ [ state, Parent ] ], Closed_set, New_closed_set ),
     path ( New_open_stack, New_closed_set, Goal ), !.
get_children ( state, Rest_open_stack, closed_set, Children) :-
     bagof ( Child, moves ( State, Rest_open_stack, Closed_set, Child ), Children).
moves ( State, Rest_open_stack, closed_set, [Next, state] ) :-
     move ( State, Next),
     not ( unsafe ( Next),
     not ( member_stack ( [Next, _ ], Rest_open_stack ) ),
     not ( member_set ( [Next, _], Closed_set ) ).
```

‘Closed_set’ holds all states on the current path plus the states that were rejected when the algorithm backtracked out of them. To find the path from the start state to the current state, we create the ordered pair [State, Parent] to keep track of each state and its parent; the start state is represented by [Start, nil].

Search starts by a go predicate that initializes the path call. Initially, [start, nil] is in the ‘Open_stack’. ‘Closed_set’ is empty.

The first path call terminates search when the ‘Open_stack’ is empty.

‘Printsolution ‘will go to the ‘Closed_set’ and recursively rebuild the solution path. Note that the solution is printed from start to goal.

The 3rd ‘path’ call uses ‘bagof’, a Prolog predicate standard to most interpreters. ‘bagof’ lets us gather all the unifications of a pattern into a single list. The 2nd parameter to ‘bagof’ is the pattern predicate to be matched in the database. The 1st parameter specifies the components of the 2nd parameter that we wish to collect.

‘bagof’ collects the states reached by firing all of the enabled production rules. This is necessary to gather all descendents of a particular state so that we can add them, in proper order, to ‘open’. The 2nd argument of ‘bagof’, a new predicate named ‘moves’, calls the ‘move’ predicates to generate all the states that may be reached using the production rules. The arguments to ‘moves’ are the present state, the open list, the closed set, and a variable that is the state reached by a good move. Before returning this state, ‘moves’ checks that the new state, ‘Next’, is not a member of either ‘rest_open_stack’, ‘open’ once the present state is removed, or ‘closed_set’. ‘bagof’ calls ‘moves’ and collects all the states that meets these conditions. The 3rd argument of ‘bagof’ thus represents the new states that are to be placed on the ‘Open_stack’.

10 Breadth first search in Prolog

The following was the algorithm we learned for breadth first search.

```
void breadth_ first _ search ( )
{
    open = [ start ];
    closed = [ ];
    while ( open not empty )
    {
         Remove the leftmost state from open, call it X;
         if X is a goal,
                then return SUCCESS;
         else
         {
                Generate children of X;
                Put X on closed;
                Discard children of x, if already on open or closed;
                Put remaining children on right end of open;
         }
     }
     return FAIL;
}
```

The following is the Prolog program corresponding to this for breadth first search..

```
go ( Start, Goal):-
    empty_queue (Empty_open_queue),
    enqueue ( [Start, nil] ), Empty_open_queue, Open_queue),
    empty_set (Closed_set),
    path (Open_queue, Closed_set, Goal).
path (Open_queue, _, _ ) :-
    empty_queue (Open_queue),
    write (No solution found with these rules).
path (Open_queue, Closed_set, Goal) :-
    dequeue ( [State, Parent], Open_queue, _ ), State = Goal,
    write ( A solution is found) , nl,
    printsolution ( [State, Parent], Closed_set).
```

```
path (Open_queue, Closed_set, Goal) :-
     dequeue ( [State, Parent] , Open_queue, Rest_open_queue ),
     get_children( State, Rest_open_queue, Closed_set, Children )
     add_list_to_queue ( Children, Rest_open_queue, New_open_queue ),
     union ( [ [ state, Parent ] ], Closed_set, New_closed_set ),
     path ( New_open_queue, New_closed_set, Goal ), !.
get_children ( state, Rest_open_queue, closed_set, Children) :-
     bagof ( Child, moves ( State, Rest_open_queue, Closed_set, Child ), Children).
moves ( State, Rest_open_queue, closed_set, [Next, state] ) :-
     move ( State, Next),
     not ( unsafe ( Next),
     not ( member_queue ( [Next, _ ], Rest_open_queue ) ),
     not ( member_set ( [Next, _], Closed_set ) ).
```

11 Best first search in Prolog

The following was the algorithm we learned for best first search.

```
function best_first_search ( )
{
     open = [start];
     closed = [ ];
     while (open not empty)
     {
          remove the left most state from open, call it X;
          If X = goal then return the path from start to X;
          else
          {
               generate children of X;
               for each child of X do
               {
                    case
                         the child is not in open or closed :
                    {
                              assign the child a heuristic value;
                              add the child to open;
```

```
                }
                case
                    the child is already on open:
                {
                        if the child was reached by a shorter path
                            then give the state on open the shorter path;
                }
                case
                    the child is already on closed :
                {
                        if the child was reached by a shorter path
                        then
                        {
                            remove the state from closed;
                            add the child to open;
                        }
                }
            } /*end of for */
            put X on closed;
            reorder states on open by heuristic merit;
        } /* end of else */
    } /* end of while */
    return FAIL;
}
```

Our algorithm for best first search is a modification of the breadth first search algorithm in which the open queue is replaced by a priority queue, ordered by heuristic merit.

To keep track of all required search information, each state is represented as a list of 5 elements: the state description, the parent of the state, an integer giving the depth of the graph of its discovery, an integer giving the heuristic measure of the state, and the integer sum of the 3rd and 4th elements.

The 1st and 2nd elements are found in the usual way; the 3rd is determined by adding one to the depth of its parent; the 4th is determined by the heuristic measure of the particular problem. The 5th element, used for ordering the states on the open_pq , is f(n) = g(n) + h(n).

```
go ( Start , Goal):-
    empty_set ( Closed_set ),
    empty_pq ( Open ),
    heuristic ( start , Goal , H) ,
    insert_pq ( [Start , nil , 0, H, H] , Open, Open_pq ) ,
    path (Open_pq, Closed_set , Goal).
path (Open_pq, _, ) :-
    empty_pq (Open_pq) ,
    write (No solution found with these rules).
path (Open_pq, Closed_set , Goal) :-
    dequeue_pq ( [ State , Parent , _, _, _ ] , Open_pq, _ ) ,
    State = Goal ,
    write ( A solution is found) , nl ,
    printsolution ( [State , Parent , _, _, _ ] , Closed_set ).
path (Open_pq, Closed_set , Goal) :-
    dequeue_pq ( [ State , Parent , D, H, S] , Open_pq, Rest_open_pq ) ,
    get_children ( [ State , Parent , D, H, S] , Rest_open_pq , Closed_set , Children , Goal )
    insert_list_pq ( Children , Rest_open_pq , New_open_pq ) ,
    union ( [ [ state , Parent , D, H, S ] ] , Closed_set , New_closed_set ) ,
    path ( New_open_queue, New_closed_set , Goal ) , !.
```

get_children is a predicate that generates all the children of state. It uses bagof and moves predicates as in the previous searches.

```
get_children ( [ state , _, D, _, _ ] , Rest_open_pq, Closed_set , Children , Goal) :-
    bagof ( Child , moves ([ state , _, D, _, _ ], Rest_open_pq ,
    Closed_set , Child , Goal ) , Children ).
moves ([ state , _, Depth , _, _ ] , Rest_open_pq, closed_set ,
    [Next , state , New_D, H, S ] , Goal ) :-
    move ( State , Next ) ,
    not ( unsafe ( Next ) ) ,
    not ( member_pq ( [Next , _ , _, _ ] , Rest_open_pq ) ) ,
    not ( member_set ( [Next , _ , _, _ ] , Closed_set ) ) ,
    New_D is Depth+1,
    Heuristic (Next , Goal , H) ,
```

S is New_D + H.

.

Discuss search strategies. (4 marks) [MGU/Nov2010]

Part IV. Meta Predicates

These predicates are designed to match, query and manipulate other predicates that make up the specifications of the problem domain.

We use meta predicates

1. to determine the type of an expression.
2. to add type constraints to logic programming applications.
3. to build, take apart and evaluate Prolog structures.
4. to compare values of expressions.
5. to convert predicates passed as data to executable code.

assert

this predicate adds a clause to the current set of clauses.

assert (C) adds the clause C to the current set of clauses.

var

var (X) succeeds only when X is an unbound variable.

nonvar

nonvar (X) succeeds only when X is bound to a non variable term.

=..

=.. creates a list from a predicate term.

For example,

foo (a, b, c) = ..Y

unifies Y with [foo, a, b, c]. The head of the list Y is the function name and its tail is the function's arguments.

functor

functor (A, B, C) succeeds with a term whose principal factor has name B and arity C.

for example, functor (foo (a, b), X, Y)

will succeed with variables X = foo and Y = 2.

clause

clause (A, B) unifies B with the body of a clause whose head unifies with A.

If we have a predicate

p(X) :- q(X) in the database,

then clause (p (a), Y) will succeed with Y = q (a).

12 Unification, the engine for predicate matching and evaluation

In prolog, the interpreter behaves as a resolution based theorem prover. As a theorem prover, Prolog performs a series of resolutions on database entries, rather than evaluating statements and expressions.

In prolog, variables are bound by unification and not by evaluation.

Unification is a powerful technique for rule based and frame based expert systems. All production systems require a form of this matching. For those languages that do not provide it, it is necessary to write a unification algorithm.

Unification performs syntactic matches. It does not evaluate expressions.

Example

If we have a predicate

```
successor (X, Y) :- Y = X + 1
```

We might have formulated this clause for checking whether Y is a successor of X. But this will fail because the = operator does not evaluate its arguments, but only attempts to unify the expressions on either side. The call successor (3, 4) fails.

For evaluation, prolog provides an operator ‘is’. ‘is’ evaluates the expression on its RHS and attempts to unify the result with the object on its left. Thus

```
X is Y + Z
```

unifies X with the value of Y added to Z.

Using ‘is’, we may define successor as

```
successor ( X, Y) :- Y is X +1
?-      successor (3, X)
X = 4
yes

?-      successor (3, 4)
yes

?-      successor (4, 2)
no
```

Thus prolog does not evaluate expressions as a default as in traditional languages. The programmer must explicitly indicate evaluation using ‘is’.

Part V. Meta Interpreters in Prolog

.

.

Explain matching in Prolog with example. (12 marks) [MGU/July2007]

What is matching? (4 marks) [MGU/Jan2007]

Explain meta interpreters. (4 marks) [MGU/May2008]

Explain the concept of meta predicates, meta interpreters in Prolog. (12 marks) [MGU/May2008]

Write notes on i) Meta predicates. ii) Meta interpreters. (12 marks) [MGU/Jan2007]

What are meta predicates? Explain. (12 marks) [MGU/July2007]

What is meta interpreter? (4 marks) [MGU/June2006]

Write notes on : a) Meta predicates. b) Meta interpreters (12 marks) [MGU/Nov2010]

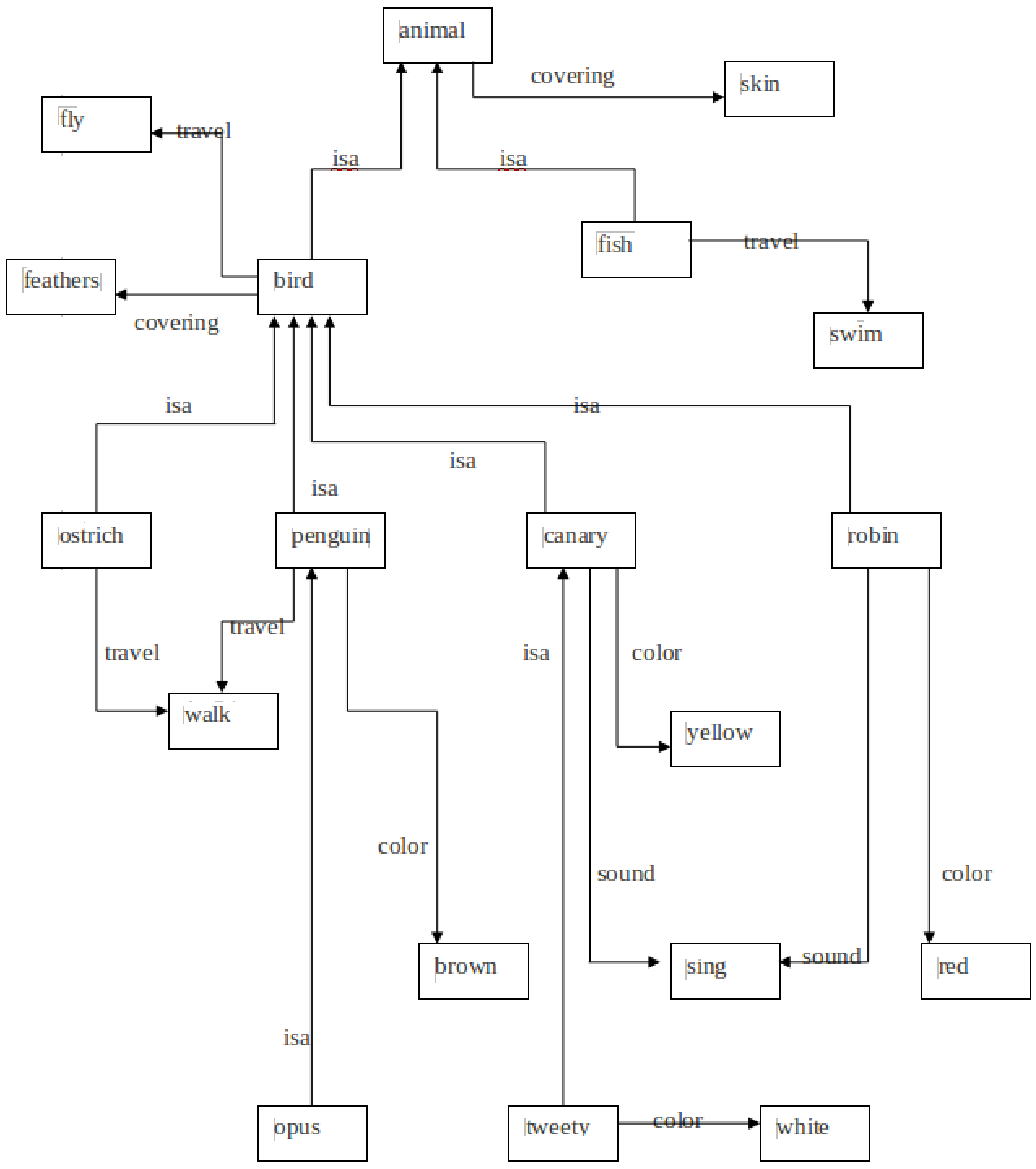
Explain the concept of evaluation in Prolog. (4 marks) [MGU/July2007]

Brief on the concept of matching and evaluation in prolog with example. (12 marks) [MGU/June2006]

Part VI. Semantic Nets in Prolog

Here we discuss the implementation of semantic nets in Prolog.

The following shows a semantic net.



Some of the prolog predicates describing the bird hierarchy in the above diagram is

```
isa (canary, bird)                    isa (canary, bird)
isa (ostrich, bird)                   isa (penguin, bird)
isa (bird, animal)                    isa (fish, animal)
isa (opus, penguin)                   isa (tweety, canary)
hasprop (tweety, color, white)        hasprop (robin, color, red)
hasprop (canary, color, yellow)       hasprop (penguin, color, brown)
hasprop (bird, travel, fly)          hasprop (fish, travel, swim)
hasprop (ostrich, travel, walk)      hasprop (penguin, travel, walk)
hasprop (robin, sound, sing)         hasprop (canary, sound, sing)
```

Following is an algorithm to find whether an object in our semantic net has a particular property.

```
hasproperty ( Object, Property, Value) :-
            hasprop ( Object, Property, Value)
hasproperty ( Object, Property, Value) :-
            isa ( Object, Parent),
hasproperty (Parent, Property, Value)
```

'hasproperty' searches the inheritance hierarchy in a depth first fashion.

.

With example explain how to represent semantic nets in prolog. (12 marks) [MGU/June2006]

Explain the implementation of semantic nets in Prolog. (12 marks) [MGU/Nov2010]

Part VII. Frames in Prolog

The following shows some of the frames from the previous semantic net.

name	:	bird
isa	:	animal
properties	:	flies
		feathers
default	:	

name	:	tweety
isa	:	canary
properties	:	
default	:	color (white)

name	:	animal
isa	:	animate
properties	:	eats
		skin
default	:	

name	:	bird
isa	:	animal
properties	:	flies
		feathers
default	:	

name	:	canary
isa	:	bird
properties	:	color (yellow)
		sound (sing)
default	:	size (small)

The 1st slot of each frame names the node, such as name (tweety) or name (bird).

The 2nd slot gives the inheritance links between the node and its parents.

The 3rd slot in the node's frame is a list of frames that describe the node.

The final slot in the frame is the list of exceptions and default values for the node.

We now represent the relationships in the frames given above using Prolog.

```
frame ( name (bird),
        isa (animal),
        [ travel (flies), feathers],
        [ ] ).

frame ( name (penguin),
        isa (bird),
               [ color (brown) ],
               [ travel (walks) ] ).

        frame ( name (canary),
               isa (bird),
               [ color (yellow), call (sing) ],
               [size (small) ] ).

        frame ( name (tweety),
               isa (canary),
               [ ],
               [ color (white) ] ).
```

The following are the procedures to infer properties from their representation.

```
        get ( Prop, Object) :-
                    frame ( name (Object), _, List_of_properties, _ ),
                    member (Prop, List_of_properties ).
        get (Prop, Object) :-
                    frame ( name (Object), _, _, List_of_defaults),
                    member ( Prop, List_of_defaults).
        get ( Prop, Object) :-
                    frame (name (Object), isa (Parent), _, _ ),
                    get ( Prop, Parent).
```

References

1. Luger, G. (2005). Artificial Intelligence. Pearson Education.