Experiment 8

Aim: Implementation of unification algorithm in Prolog.

Objective: To study about how to use AI Programming language (Prolog) for developing interfacing engine using Unification process and knowledge declared in Prolog.

Requirement: Turbo Prolog 2.0 or above / Windows Prolog.

Theory:

Unification is a process of making two different logical atomic expressions identical by findinga substitution. .....It takes two literals as input and makes them identical using substitution. Let Ψ1 and Ψ2 be two atomic sentences and be a unifier such that, Ψ1 = Ψ2 , then it can be expressed as UNIFY(Ψ1, Ψ2).

For example, if one term is f(X, Y) and the second is f(g(Y, a), h(a)) (where upper case namesare variables and lower case are constants) then the two terms can be unified by identifying X with g(h(a), a) and Y with h(a) making both terms look like f(g(h(a),a), h(a)). The unification can be represented by a pair of substitutions {X g(h(a), a)} and {Y h(a)}.

Unification Algorithm:

```
FUNCTION unify( t1, t2 ) RETURNS (unifiable : BOOLEAN, sigma :SUBSTITUTION) BEGIN
IF t1 OR t2 is a variableTHEN BEGIN
  let x be the variable and let t be the other termIF x == t THEN (unifiable, sigma) := (TRUE,
  NULL_SUBSTITUTION); ELSE IF x occurs in t THEN
  unifiable == FALSE;
  ELSE (unifiable, sigma) := (TRUE,
    {x <- t}); ENDELSE
  BEGIN
  assume t1 == f(x1,......, xn) and t2 == g(y1,
  ... ym) IF f != g OR m != n THEN
unifiable = FALSE; ELSEBEGI
  N k
  := 0;
  unifiable := TRUE;
  sigma := NULL_SUBSTITUTION;
  WHILE k < m AND
  unifiable DO BEGIN
  k := k + 1;
  (unifiable, tau) := unify( sigma( xk ), sigma( yk ) ); IF unifiable THEN sigma := compose(tau, sigma );
  END
  END
  END
  RETURN (unifiable,sigma); END
```

Implementation Notes

1. To extract the name of a functor and its arguments, you may use the special built-inrules **functor/3**, **arg/3**, and "**=..**" . (Prolog allows overloading of rule names; the notation foo/2 denotes the foo rule that takes two arguments.) They are used as follows:

   1. functor(f(x,y),F,N) ==> F=f andN=2 2. arg(1,f(x,y),A) ==> A=x 3. f(x,y) =.. L ==> L

= [f,x,y]

Incidentally, an atom is treated as a 0-argument functor.

2. As an option, you may encode functors to be unified as lists in prefix notation. Forexample, f(x) would be encoded as [f, x]. For a more complicated example, the following function:
   f(3, g(x)) would be encoded as:
   [f, 3, [g, x]]
   This notation doesn't look as nice, but it might make the implementation simpler.
3. You must choose how to distinguish variables from atoms in the expressions you are matching. For example, if **a** and **b** are constants, then unification of **a** and **b** should fail. However, if **A** and **B** are both variables, then unification should succeed, with thesingle substitution **A** -> **B**. A reasonable choice is that t, u, v, w, x, y, and z are variables, while all other letters are constants. In any case, please document your choice.

Testing Your Unifier

Here are some tests you should try before stopping work on your unifier. Harder tests are towards the bottom.

1. Two atoms should unify iff both atoms are the same. Two different atoms should failto unify.
2. A variable should unify with anything that does not contain that variable. Forexample, x should unify with f(g(y),3,(h(a,z))), but not with f(x).
3. A variable should unify with itself. For example, x should unify with x.
4. Your algorithm should handle cases where a variable appears in multiple locations
   For example, all of the following should unify:

- f(x,x) = f(a,a)
  - f(x,g(x)) = f(a, g(x))
- f(x, y) = f(y, x)

And the following should NOTunify

- f(x,x) = f(a,b)
  - f(x,g(x)) = g(a, g(b))

5. When unifying functors, all arguments should unify. For example, g(h(1,2,3,4), 5)does not unify with g(h(1,8,3,4),5).
6. There are plenty of other things to try. These are just some examples to start.

Unification in Prolog:

The way in which Prolog matches two terms is called unification. The idea is similar to that ofunification in logic: we have two terms and we want to see if they can be made to represent thesame structure. For example, we might have in our database the single Polog clause:

parent(alan, clive). and give the query:

|?- parent(X,Y).

We would expect X to be instantiated to alan and Y to be instantiated to clive when the querysucceeds. We would say that the term parent(X,Y) unifies with the term parent(alan, clive) with X bound to alan and Y bound to clive. The unification algorithm in Prolog is roughly this:

df:un Given two terms and which are to be unified:

If and are constants (i.e. atoms or numbers) then if they are the same succeed. Otherwise fail.

If is a variable then instantiate to . Otherwise, If is avariable then instantiate to .

Otherwise, if and are complex terms with the same arity (number of arguments), find the principal functor of and principal functor of . If these are the same, then take the ordered set of arguments of of and the ordered set of arguments of . For each pair of arguments and from the same position in the term, must unify with . Otherwise fail.

For example: applying this procedure to unify foo(a,X) with foo(Y,b) we get:foo(a,X) and foo(Y,b) are complex terms with the same

arity (2). The principal functor of both terms is foo.

The arguments (in order) of foo(a,X) are a and X. The arguments (in order) of foo(Y,b)are Y and b. So a and Y must unify , and X and b must unify. Y is a variable so we instantiate Y to a.

X is a variable so we instantiate X to b.

The resulting term, after unification is foo(a,b).

The built in Prolog operator '=' can be used to unify two terms. Below are some examples ofits use. Annotations are between ** symbols.

| ?- a = a. ** Two identical atomsunify ** yes

| ?- a = b. ** Atoms don't unify if they aren'tidentical ** no

| ?- X = a. ** Unification instantiates a variable to anatom ** X=a

yes

| ?- X = Y. ** Unification binds two differently namedvariables ** X=_125451 ** to a single, unique variable name **

Y=_125451

yes

| ?- foo(a,b) = foo(a,b). ** Two identical complex termsunify ** yes

| ?- foo(a,b) = foo(X,Y).** Two complex terms unify if theyare ** X=a ** of the same arity, have the same principal**

Y=b ** functor and their argumentsunify ** yes

| ?- foo(a,Y) = foo(X,b).** Instantiation of variables may occur ** Y=b ** in either of the terms to be unified **X=

a yes

| ?- foo(a,b) = foo(X,X).** In this case there is no unification ** no ** because foo(X,X) must have the same

**

** 1st and 2nd arguments **

| ?- 2*3+4 = X+Y. ** The term 2*3+4 has principal functor + ** X=2*3 ** and therefore unifies withX+Y with X instantiated** Y=4 ** to 2*3 and Y instantiated to 4 **yes

| ?- [a,b,c] = [X,Y,Z]. ** Lists unify just like otherterms ** X=a

Y=

b Z=

cyes

| ?- [a,b,c] = [X|Y]. ** Unification using the '|' symbol can be used ** X=a ** to find the head element, X, andtail list, Y, ** Y=[b,c] ** of a list **

yes

| ?- [a,b,c] = [X,Y|Z]. ** Unification on lists doesn't haveto be ** X=a ** restricted to finding the first head element **

Y=b ** In this case we find the 1st and 2nd elements ** Z=[c] ** (X and Y) and then the tail list (Z) **

yes

| ?- [a,b,c] = [X,Y,Z|T]. ** This is a similar example buthere ** X=a** the first 3 elements are unified with

**

Y=b ** variables X, Y and Z, leaving the **Z=c ** tail, T, as an empty list

[] ** T=[]

Yes

| ?- [a,b,c] = [a|[b|[c|[]]]]. ** Prolog is quite happy to unify these ** yes** because they are just notational

**

** variants of the same Prolog term **

Code:
Code 1:

```
is_changed = True

# Add more initial facts
facts = [["vertebrate", "duck"], ["flying", "duck"], ["mammal", "cat"], ["insect", "bee"], ["vertebrate", "dog"]]

def assert_fact(fact):
  global facts
  global is_changed
  if fact not in facts:
    facts += [fact]
    is_changed = True

while is_changed:
  is_changed = False

  for A1 in facts:
    if A1[0] == "mammal":
      assert_fact(["vertebrate", A1[1]])

    if A1[0] == "vertebrate":
      assert_fact(["animal", A1[1]])

    if A1[0] == "vertebrate" and ["flying", A1[1]] in facts:
      assert_fact(["bird", A1[1]])

print(facts)
```

Output:

```
[['vertebrate', 'duck'], ['flying', 'duck'], ['mammal', 'cat'], ['insect', 'bee'], ['vertebrate', 'dog'], ['animal', 'duck'], ['animal', 'cat'], ['animal', 'dog'], ['vertebrate', 'cat'], ['bird', 'duck']]
```

Code 2:

```
is_changed = True
facts = [["can_fly", "sparrow"], ["has_feathers", "sparrow"], ["animal", "sparrow"]]
def assert_fact(fact):
  global facts
  global is_changed
  if fact not in facts:
    facts.append(fact)
    is_changed = True
while is_changed:
  is_changed = False
  for A1 in facts:
    if A1[0] == "animal":
      assert_fact(["living_organism", A1[1]])
    if A1[0] == "living_organism" and ["can_fly", A1[1]] in facts:
      assert_fact(["bird", A1[1]])
print(facts)

is_changed = True
facts = [["mammal", "lion"], ["has_fur", "lion"], ["animal", "lion"]]
def assert_fact(fact):
  global facts
  global is_changed
  if fact not in facts:
    facts.append(fact)
    is_changed = True
while is_changed:
  is_changed = False
  for A1 in facts:
    if A1[0] == "animal":
      assert_fact(["living_organism", A1[1]])
    if A1[0] == "living_organism" and ["mammal", A1[1]] in facts:
      assert_fact(["warm_blooded", A1[1]])
print(facts)
```

Output:

```
[['can_fly', 'sparrow'], ['has_feathers', 'sparrow'], ['animal', 'sparrow'], ['living_organism', 'sparrow'], ['bird', 'sparrow']]
[['mammal', 'lion'], ['has_fur', 'lion'], ['animal', 'lion'], ['living_organism', 'lion'], ['warm_blooded', 'lion']]
```

Conclusion:

Thus, we have studied about how to use AI Programming language (Prolog) for developing Interfacing engine using Unification process and knowledge declared in Prolog.