Experiment No.8
Study and Implementation of unification algorithm in Prolog.
Date of Performance:
Date of Submission:

Aim: Study and Implementation of unification algorithm in Prolog.

Objective: To study about how to use AI Programming language (Prolog) for developing inferencing 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 finding a substitution. ... It takes two literals as input and makes them identical using substitution. Let Ψ1 and Ψ2 be two atomic sentences and σ be a unifier such that, $\Psi 1\sigma = \Psi 2\sigma$, 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 names are 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 variable THEN
      BEGIN
         let x be the variable and let t be the other term
         IF x == t THEN (unifiable, sigma) := (TRUE, NULL_SUBSTITUTION);
         ELSE IF x occurs in t THEN unifiable == FALSE;
         ELSE (unifiable, sigma) := (TRUE, {x <- t});
      END
   ELSE
      BEGIN
         assume t1 == f(x1, ..., xn) and t2 == g(y1, ... ym)
         IF f != g OR m != n THEN unifiable = FALSE;
         ELSE
            BEGIN
               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-in rules **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 and N=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. For example, 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 the single 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 fail to unify.
2. A variable should unify with anything that does not contain that variable. For example, 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 *NOT* unify:

    - 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 of unification in logic: we have two terms and we want to see if they can be made to represent the same 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 query succeeds. 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 a variable 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 of its use. Annotations are between ** symbols.

| ?- a = a.        ** Two identical atoms unify **

yes

| ?- a = b.        ** Atoms don't unify if they aren't identical **

no

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

     X=a

yes

| ?- X = Y.        ** Unification binds two differently named variables **

     X=_125451     ** to a single, unique variable name **

     Y=_125451

yes

| ?- foo(a,b) = foo(a,b).       ** Two identical complex terms unify **

yes

| ?- foo(a,b) = foo(X,Y).       ** Two complex terms unify if they are **

     X=a                              ** of the same arity, have the same
principal**

     Y=b                       ** functor and their arguments unify **

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 with X+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 other terms **
     X=a
     Y=b
     Z=c
yes
| ?- [a,b,c] = [X|Y].   ** Unification using the '|' symbol  can be used **
     X=a                  ** to find the head element, X, and tail list, Y,
**
     Y=[b,c]            ** of a list **
yes
| ?- [a,b,c] = [X,Y|Z]. ** Unification on lists doesn't have to 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 but here **
     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 **
```

EXAMPLE:

Facts.

father(homer,bart).

father(homer,lisa).

father(homer,maggie).

mother(marge,bart).

mother(marge,lisa).

mother(marge,maggie).

father(ned,rod).

father(ned,todd).

male(homer).

male(bart).

male(ned).

male(rod).

male(todd).

Rules:

parents(F,M,C) :- father(F,C), mother(M,C).

siblings(A,B) :- parents(F,M,A), parents(F,M,B).

brother(A,B) :- siblings(A,B), male(B).

Simple questions - true if the fact exists in the knowledge base, else false.

?- father(homer,bart).

yes

?- father(homer,lisa).

yes

?- father(homer,maggie).

yes

?- father(homer,rod).

no

?- father(homer,ralph).

no

Question using a variable.

?- father(homer,C).

C = bart

yes;

C = lisa

yes;

C = maggie

yes

?- father(ned,C).

C = rod

yes;

C = todd

yes

Question using two variables - evaluation of the query uses backtracking to find multiple solutions.

?- father(F,C).

C = bart

F = homer

yes;

C = lisa

F = homer

yes;

C = maggie

F = homer

yes;

C = rod

F = ned

yes;

C = todd

F = ned

yes

?- parents(homer,marge,C).

C = bart

yes;

C = lisa

yes;

C = maggie

yes

?- brother(lisa,X).

X = bart

yes;

no

?- brother(bart,X).

X = bart

yes;

no

Conclusion:

In conclusion, the unification experiment yielded valuable insights into the complexities and challenges of integrating disparate elements into a cohesive whole. While the experiment did not achieve full unification, it provided crucial understanding of the underlying mechanisms and highlighted areas for further investigation. The results underscore the importance of interdisciplinary approaches and iterative refinement in the pursuit of integration. Moving forward, future research can build upon these findings to explore more effective strategies for unification, with potential applications spanning various fields from science and technology to social and cultural domains.