3.1 PROLOG PROGRAMMING

Prolog (which stands for ―Programming in Logic‖) is one of the most interesting languages in use today. Prolog is a declarative language that focuses on logic. Programming in Prolog involves specifying rules and facts and allowing Prolog to derive a solution. Prolog was created around 1972 by Alain Colmerauer and Robert Kowalski as a competitor to the LISP language.

- The earliest Prolog applications were in the domain of natural-language programming, as this was the object of research by its creators. Natural language programming (NLP) is supported by a unique feature of Prolog.
- Prolog includes a built-in mechanism for parsing context-free grammars, making it ideal for NLP or parsing grammars.
- In fact, like LISP, a Prolog interpreter can be easily implemented in Prolog.
- Today, Prolog remains a language used primarily by researchers, but the language has advanced to support multiple programming paradigms.
- Prolog is ideal for developing knowledge-based systems such as expert systems and also systems for research into computer algebra.

- Prolog‘s problem solving capabilities maintain it as an important language for problems in the knowledge domain.

3.2.1 Overview of the Prolog Language

Prolog comprised of two fundamental elements, the **database** and the **interpreter.**

1. The database contains the facts and rules that are used to describe the problem.
2. The interpreter provides the control in the form of a deduction method.

Prolog doesn‘t include data types, but instead what can be referred to as lexical elements.
Some of the important lexical elements of Prolog include atoms, numbers, variables, and lists.

An **atom** is a string made up of characters (lower and upper case), digits, and theunderscore. Numbers are simply sequences of digits with an optional preceding minus sign. Variables have the same format as atoms, except that the first character is always capitalized. A **list** is represented as a comma delimited set of elements, surrounded by brackets. For example

person Atom, f_451 Atom, _A complex atom.‘ Atom,
86 Number, -451 Number, Person Variable,
A_variable Variable, [a, simple, list] List, [a, [list of lists]] List.

Constructing a list is performed simply as follows:

```
| ?- [a, b, c, d] = X
X = [a,b,c,d]Yes | ?-
```

The X variable refers to the list [a, b, c, d], which is identified by the Prolog interpreter. We can also concatenate two lists using the **append** predicate.

```
| ?- append( [[a, b], [c, d]], [[e, f]], Y ).
Y = [[a,b],[c,d],[e,f]]
yes | ?-
```

List has **length and member** functions and **[Head | Tail]** variables.

What separates Prolog from all other languages is its **built-in ability of deduction.**

Define fruit(pear). Query the Prolog database, such as:

```
| ?- fruit(pear).
Yes | ?-
```

for which Prolog would reply with _yes‘ (a pear is a fruit). Create a **rule** oddity to define that thetomato is both a fruit and a vegetable.

```
oddity(X) :-
fruit(X), vegetable(X).
```

Similarly, family tree is constructed with the following rules. To be a husband, a person of the male gender is married to a person of the female gender and a wife is a female person that is alsomarried to a male person.

husband(Man, Woman) : male(Man), married(Man, Woman).

wife(Woman, Man) :

female(Woman), married(Man, Woman).

We create son/daughter rule, which defines that a son/daughter is a male/female child of a parent.

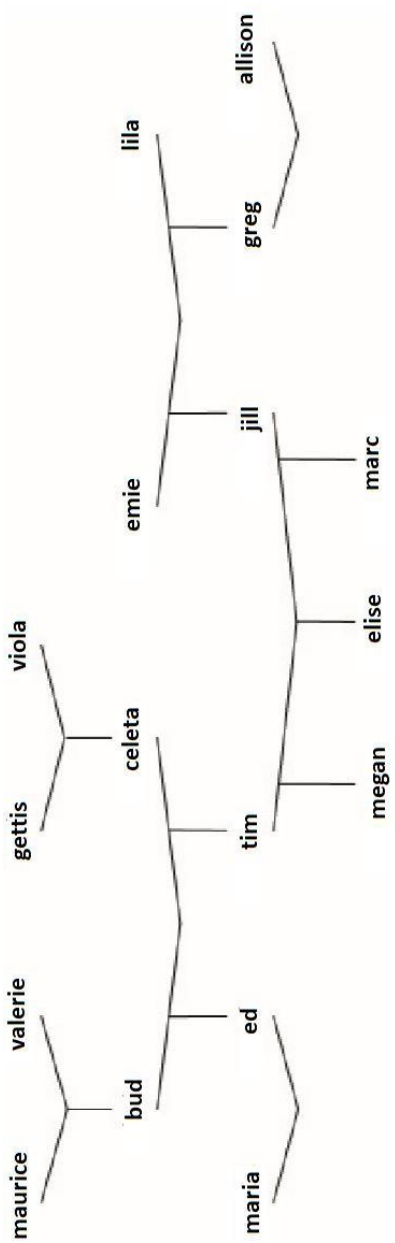
son(Child, Parent) :

male(Child), child(Child, Parent). daughter(Child, Parent) :

female(Child), child(Child, Parent).

3.2.2 Querying the family tree database with gprolog.

```
| ?- consult(_family.pl‘).
compiling /home/mtj/family.pl for byte code...
/home/mtj/family.pl
compiled, 71 lines read - 5920 bytes written, 51 ms (10 ms)yes
| ?- grandchild(marc, X).X
= bud
X = ernie X
= celetaX =
lila No
| ?-
| ?- trace.
The debugger will first show everything (trace)
yes
{trace}
| ?- grandparent(bud, marc).
        1  1  Call: grandparent(bud,marc)
        2  2  Call: parent(bud,_79)
        3  3  Call: father(bud,_103)
3               3  Exit: father(bud,tim)
        2  2  Exit: parent(bud,tim)
        3  2  Call: parent(tim,marc)
        4  3  Call: father(tim,marc)
        5  3  Exit: father(tim,marc)
4               2  Exit: parent(tim,marc)
1               1  Exit: grandparent(bud,marc)
true ?
yes
{trace} | ?-
```



Let‘s now have a look at this database in action. This is illustrated below using the GNU prolog interpreter (gprolog). **Trace** allows us to trace the flow of the application of rules for a given query. For example, the following query checks to see if a person is a grandparent of another person. Note here that the variables preceded by underscores are temporary variables created by Prolog during its rule checking.

3.2.3 Arithmetic Expressions

Finally, let‘s look at an example that combines arithmetic expressions with logic programming. In this example, we‘ll maintain facts about the prices of products, and then a rule that can compute the total cost of a product given a quantity (again, encoded as a fact). First, let‘sdefine our initial set of facts that maintain the price list of the available products.

cost(banana, 0.35). cost(apple, 0.25). cost(orange, 0.45). cost(mango, 1.50).

We can also provide a quantity that we plan to purchase, using another relation for the fruit, for example: qty(mango, 4).

We can then add a rule that calculates the total cost of an item purchase:

```
total_cost(X,Y) :
    cost(X, Cost), qty(X, Quantity),
    Y is Cost * Quantity.
```

What this rule says is that if we have a product with a cost fact, and a quantity fact, then we can create a relation to the product cost and quantity values for the item. We can invoke this from the Prolog interpreter as:

```
| ?- total_cost( mango, TC ).
TC = 6.0
```

yes | ?- This tells us that the total cost for four mangos is $6.00.

3.2 UNIFICATION

We have noticed that the propositional logic approach is rather inefficient. For example,given the query Evil(x) and the knowledge base, it seems perverse to generate sentences such as

King(Richard) A Greedy(Richard) ⇒ Evil(Richard).

∀ x King(x) A Greedy(x) ⇒ Evil(x)King(John) Greedy(John)

Indeed, the inference of Evil(John) from the sentences seems completely obvious. We now show how to make it completely obvious to a computer.

3.3.1 A first-order inference rule

The inference that John is evil, that is, that {x/John} solves the query Evil(x) works like this: to use the rule that greedy kings are evil, find some x such that x is a king and x is greedy, and then infer that this x is evil. More generally, if there is some substitution θ that makes eachof the conjuncts of the premise of the implication identical to sentences already in the knowledgebase, then we can assert the conclusion of the implication, after applying θ. In this case, the substitution θ ={x/John} achieves that aim. Suppose that instead of knowing Greedy(John), we know that *everyone* is greedy: ∀ y Greedy(y) .

Then we would still be able to conclude that Evil(John), substitute {x/John, y/John} to the implication premises King(x) and Greedy(x) and the knowledge-base sentences King(John) and Greedy(y) will make them identical. Thus, we can infer the conclusion of the implication. This inference process can be captured as **Generalized Modus Ponens.** For atomic sentences p_i, p_i‘, and q, where there is a substitution θ such that SUBST(θ , p_i‘)= SUBST(θ , pi), for all i,

$$\frac{p_1',\ p_2',\ \ldots,\ p_n',\ (p_1 \wedge p_2 \wedge \ldots \wedge p_n \Rightarrow q)}{\text{SUBST}(\theta, q)}$$

There are n+1 premises to this rule: the n atomic sentences p_i‘ and the one implication.The conclusion is the result of applying the substitution θ to the consequent q. For our example:

p_1‘ is King(John) p1 is King(x)

p_2‘ is Greedy(y) p2 is Greedy(x)

θ is {x/John, y/John} q is Evil(x)

SUBST(θ , q) is Evil(John) .

For any substitution θ , p |= SUBST(θ , p)

holds by Universal Instantiation. It holds in particular for a θ that satisfies the conditions of the Generalized Modus Ponens rule. Thus, from $p_1, \ldots, p_n$‘ we can infer

SUBST(θ , p_1‘) ∧ . . . ∧ SUBST(θ , p_n‘)

and from the implication p1 ∧ . . . ∧ pn ⇒ q we can infer

SUBST(θ , p_1) ∧ . . . ∧ SUBST(θ , p_n) ⇒ SUBST(θ , q) .

Now, θ in Generalized Modus Ponens is defined so that SUBST(θ, p_i‘)= SUBST(θ, p_i), forall i; therefore the first of these two sentences matches the premise of the second exactly.

Hence, SUBST(θ, q) follows by Modus Ponens.

Generalized Modus Ponens is a **lifted** version of Modus Ponens since it raises Modus Ponens from ground (variable-free) propositional logic to first-order logic.

3.3.2 Unification

Lifted inference rules require finding substitutions that make different logical expressions look identical. This process is called **unification** and is a key component of all first-order inference algorithms. The UNIFY algorithm takes two sentences and returns a **unifier** for them ifone exists:

$$\text{UNIFY}(p, q) = \theta \text{ where SUBST}(\theta, p) = \text{SUBST}(\theta, q) .$$

Suppose we have a query Ask(Knows(John, x)): whom does John know? Answers to this query can be found by finding all sentences in the knowledge base that unify with Knows(John, x). The results of unification with four different sentences that might be in the knowledge base:

UNIFY(Knows(John, x), Knows(John, Jane)) = {x/Jane} UNIFY(Knows(John, x), Knows(y, Bill)) = {x/Bill, y/John} UNIFY(Knows(John, x), Knows(y,Mother (y))) = {y/John, x/Mother (John)}UNIFY(Knows(John, x), Knows(x, Elizabeth)) = fail .

The last unification fails because x cannot take on the values John and Elizabeth at the same time. Now, remember that Knows(x, Elizabeth) means ―Everyone knows Elizabeth,‖ so we *should* be able to infer that John knows Elizabeth. The problem arises only because the two sentences happen to use the same variable name, x. The problem can be avoided by **standardizing apart** one of the two sentences being unified, by renaming its variables to avoid name clashes. Let us rename x in Knows(x, Elizabeth) to x17. Now the unification will work.

UNIFY(Knows(John, x), Knows(x17, Elizabeth)) = {x/Elizabeth, x17/John} .

There is one more complication: For example, UNIFY(Knows(John, x), Knows(y, z)) could return {y/John, x/z} or {y/John, x/John, z/John}. The first unifier gives Knows(John, z) as the result of unification, whereas the second gives Knows(John, John), {z/John}; we say that the firstunifier is *more general* than the second. For every unifiable pair of expressions, there is a single **most general unifier** (MGU) that is unique up to renaming and substitution of variables.

```
function UNIFY(x, y, θ) returns a substitution to make x and y identical
  inputs: x, a variable, constant, list, or compound expression
          y, a variable, constant, list, or compound expression
          θ, the substitution built up so far (optional, defaults to empty)

  if θ = failure then return failure
  else if x = y then return θ
  else if VARIABLE?(x) then return UNIFY-VAR(x, y, θ)
  else if VARIABLE?(y) then return UNIFY-VAR(y, x, θ)
  else if COMPOUND?(x) and COMPOUND?(y) then
      return UNIFY(x.ARGS, y.ARGS, UNIFY(x.OP, y.OP, θ))
  else if LIST?(x) and LIST?(y) then
      return UNIFY(x.REST, y.REST, UNIFY(x.FIRST, y.FIRST, θ))
  else return failure
```

```
function UNIFY-VAR(var, x, θ) returns a substitution

  if {var/val} ∈ θ then return UNIFY(val, x, θ)
  else if {x/val} ∈ θ then return UNIFY(var, val, θ)
  else if OCCUR-CHECK?(var, x) then return failure
  else return add {var/x} to θ
```

Figure 9.1 The Unification algorithm.

Procedure for UNIFY (L1, L2)

1. if L1 or L2 are both variables or constant then
    a. if L1 or L2 are identical then return NIL
    b. Else if L1 is a variable then if L1 occurs in L2 then return Fail else return (L2/L1)
    c. Else if L2 is a variable then if L2 occurs in L1 then return Fail else return (L1/L2)
    d. Else return Fail.
2. If initial predicate of L1 and L2 are not equal then return Fail.
3. If length (L1) is not equal to length (L2) then return Fail.
4. Set SUBST to NIL (at the end, SUBST will contain all substitutions in L1 and L2).
5. For i = 1 to number of elements in L1 then
    a. Call UNIFY with the ith element of L1 and L2, putting the result in S
    b. If S = Fail then return Fail
    c. If S is not equal to NIL then do
        i. Apply S to the remainder of both L1 and L2

ii. SUBST := APPEND (S, SUBST) return SUBST.

The process in Figure 9.1 is simple: recursively explore the two expressionssimultaneously –side by side,‖ building up a unifier along the way, but failing if two corresponding points in the structures do not match. There is one expensive step: when matching a variable against a complex term **occur check**, one must check whether the variable itself occurs inside the term; if it does, the match fails because no consistent unifier can be constructed.For example, S(x) can‘t unify with S(S(x)). Some systems, including all logic programming systems, simply omit the occur check and sometimes make unsound inferences as a result; other systems use more complex algorithms with linear-time complexity.

3.3.3 Storage and retrieval

Underlying the TELL and ASK functions used to inform and interrogate a knowledge base are the more primitive STORE and FETCH functions. **STORE(s)** stores a sentence s into the knowledge base and **FETCH(q)** returns all unifiers such that the query q unifies with some sentence in the knowledge base. The simplest way to implement STORE and FETCH is to keep all the facts in one long list and unify each query against every element of the list. Such a processis inefficient, but it works, and it‘s all you need to understand.

We can make FETCH more efficient by ensuring that unifications are attempted only with sentences that have some chance of unifying. For example, there is no point in trying to unify Knows(John, x) with Brother (Richard, John). We can avoid such unifications by **indexing**the facts in the knowledge base. A simple scheme called **predicate indexing** puts all the Knows facts in one bucket and all the Brother facts in another.

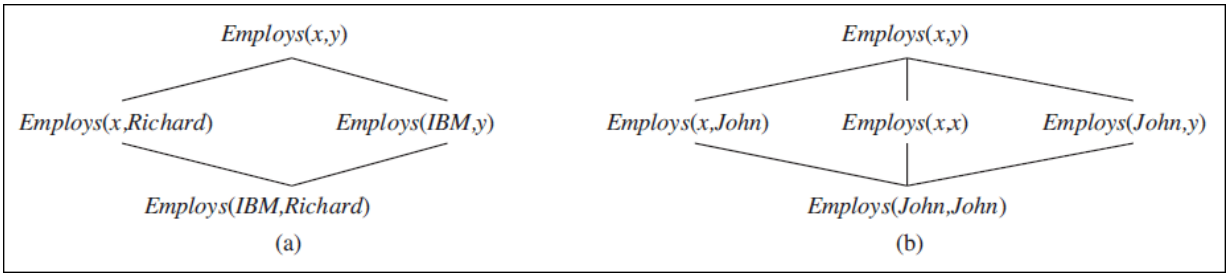


Figure 9.2 (a) The subsumption lattice whose lowest node is Employs(IBM,Richard).
(b) The subsumption lattice for the sentence Employs (John, John).

Answering a query such as Employs(x,Richard) with predicate indexing would require scanning the entire bucket. For this particular query, it would help if facts were indexed both by predicate and by second argument, perhaps using a combined hash table key.

We could simply construct the key from the query and retrieve exactly those facts that unify with the query. Therefore, facts can be stored under multiple index keys, rendering them instantly accessible to various queries that they might unify with. These queries form a **subsumption lattice**, as shown in Figure 9.2(a). The **lattice** has some interesting properties. For example, the child of any node in the lattice is obtained from its parent by a single substitution; and the ―highest‖ common descendant of any two nodes is the result of applying their most general unifier