AI LAB – 7

AIM: Study Compound object and Functors in PROLOG

Compound data objects allow you to treat several pieces of information as a single item in such a way that you can easily pick them apart again. Consider, for instance, the date December 10, 1989. It consists of three pieces of information--the month, day, and year--but it's useful to treat the whole thing as a single object with a treelike structure:

```
          DATE
       /    |   \
December 10   1989
```

You can do this by declaring a domain containing the compound object date:

DOMAINS

date_cmp = date(string,integer,integer)

and then simply writing e.g. ..., D = date("October",15,1991), ...

This looks like a Prolog fact, but it isn't here--it's just a data object, which you can handle in much the same way as a symbol or number. It begins with a name, usually called a ***functor*** (in this case date), and followed by three arguments.

Note carefully that a functor in Turbo Prolog has nothing to do with a function in other programming languages. A functor does not stand for some computation to be performed. It's just a name that identifies a kind of compound data object and holds its arguments together.

The arguments of a compound data object can themselves be compound. For instance, you might think of someone's birthday as an information structure like this:

```
                 BIRTHDAY
                 /      \
                /        \
         person            date
         /    \            / | \
"Apurva"    "Mehta"  "Jun" 10 1990
```

In Prolog you would write this as: birthday(person("Apurva","Mehta"),date("Jun",10,1990)).

In this example, there are two parts to the compound object birthday: the object person("Apurva", "Mehta") and the object date("Jun", 10, 1990). The functors of these data objects are person and date.

Unification of Compound Objects

A compound object can unify either with a simple variable or with a compound object that matches it (perhaps containing variables as parts of its internal structure). This means you can use a compound object to pass a whole collection of items as a single object, and then use unification to pick them apart. For example,

    date("April",14,1960)

matches X and binds X to date("April",14,1960).

Also

    date("April",14,1960)

matches date(Mo,Da,Yr) and binds Mo to "April", Da to 14, and Yr to 1960.

```
Sample Code I

domains

        location = address(street, city, state, zip)
        street, city, state, zip = string

predicates

        person(string,symbol,location)
        go

clauses

        go:-
                person("Apurva Mehta",_,address(_,_,X,_)),
                person(Name,engineer,address(_,_,X,_)),Name<>"Apurva Mehta",
                Write(Name),nl,fail.

        person("Mrunal Sheth",engineer,address("Vivekanand","Bhavnagar","Gujarat","364002")).
        person("Nihar Panchal",businessman,address("MG Road","Bhavnagar","Gujarat","364004")).
        person("Apurva Mehta",engineer,address("Vivekanand","Bhavnagar","Gujarat","364002")).
        person("Sachita Bhatt",engineer,address("MG Road","Hyderabad","AndhraPradesh","500002")).
```

Compound objects can be regarded and treated as single objects in your Prolog clauses, which greatly simplifies programming. Consider, for example, the fact

```
   owns(apurva, book("Parva", "S L Bhyrappa")).
```

in which you state that Apurva owns the book Parva, written by S L Bhyrappa. Likewise, you could write

```
   owns(apurva, horse(blacky)).
```

which can be interpreted as apurva owns a horse named blacky.

The compound objects in these two examples are

book("Parva", "S L Bhyrappa")

and

horse(blacky)

If you had instead written two facts:

```
   owns(apurva, "Parva").
   owns(apurva, blacky ).
```

you would not have been able to decide whether blacky was the title of a book or the name of a horse. On the other hand, you can use the first component of a compound object--the functor--to distinguish between different objects. This example used the functors book and horse to indicate the difference between the objects.

Remember: Compound objects consist of a functor and the objects belonging to that functor, as follows:

```
   functor(object1, object2, ..., objectN)
```

An Example Using Compound Objects

An important feature of compound objects allows you to easily pass a group of values as one argument. Consider a case where you are keeping a telephone database. In your database, you want to include your friends' and family members' birthdays. Here is a section of code you might have come up with:

```
PREDICATES
   phone_list(symbol, symbol, symbol, symbol, integer, integer)
     /* ( First,   Last,  Phone,  Month,     Day,    Year) */
CLAUSES
   phone_list(micky, parikh, 422-0208, aug, 3, 1955).
   phone_list(vicky, patel, 433-9906, may, 12, 1962).
```

Examine the data, noticing the six arguments in the fact phone_list; five of these arguments can be broken down into two com-pound objects, like this:

```
                 person                  birthday
                /     \                 /   |   \
         First Name  Last Name       Month  Day  Year
```

It might be more useful to represent your facts so that they reflect these compound data objects. Going back a step, you can see that person is a relationship, and the first and last names are the objects. Also, birthday is a relationship with three arguments: month, day, and year. The Prolog representation of these relationships is

```
    person(First_name, Last_name)
    birthday(Month, Day, Year)
```

You can now rewrite your small database to include these compound objects as part of your database.

```
DOMAINS
    name = person(symbol, symbol)              /* (First, Last) */
    birthday = b_date(symbol, integer, integer) /* (Month, Day, Year) */
    ph_num = symbol                            /* Phone_number */
PREDICATES
    phone_list(name, ph_num, birthday)
CLAUSES
    phone_list(person(micky, parikh), "422-0208", b_date(aug, 3, 1955)).
    phone_list(person(vicky, patel), "433-9906", b_date(may, 12, 1962)).
```

The phone_list predicate now contains three arguments, as opposed to the previous six. Sometimes breaking up your data into compound objects will clarify your program and might help process the data.

Now add some rules to your small program. Suppose you want to create a list of people whose birthdays are in the current month. Here's the program code to accomplish this task; this program uses the standard predicate date to get the current date from the computer's internal clock.

Sample Code II

```
DOMAINS
    name = person(symbol,symbol)              /* (First, Last) */
    birthday = b_date(symbol,integer,integer) /* (Month, Day, Year) */
    ph_num = symbol                           /* Phone_number */

PREDICATES
    phone_list(name,symbol,birthday)
    get_months_birthdays
    convert_month(symbol,integer)
    check_birthday_month(integer,birthday)
```

```
   write_person(name)

CLAUSES
   get_months_birthdays:-
      write("************ This Month's Birthday List *************"),nl,
      write(" First name\t\t Last Name\n"),
      write("***************************************************"),nl,
      date(_, This_month, _),      /* Get month from system clock */
      phone_list(Person, _, Date),
      check_birthday_month(This_month, Date),
      write_person(Person),
      fail.

   get_months_birthdays:-
      write("\n\n Press any key to continue: "),nl,
      readchar(_).

   write_person(person(First_name,Last_name)):-
      write("  ",First_name,"\t\t   ",Last_name),nl.

   check_birthday_month(Mon,b_date(Month,_,_)):-
      convert_month(Month,Month1),
      Mon = Month1.

   phone_list(person(apurva, mehta), "767-8463", b_date(jan, 13, 1955)).
   phone_list(person(apurva, shah), "438-8400", b_date(feb, 04, 1985)).
   phone_list(person(apurva, parikh), "555-5653", b_date(mar, 22, 1935)).
   phone_list(person(apurva, doshi), "767-2223", b_date(apr, 04, 1951)).
   phone_list(person(apurva, joshi), "555-1212", b_date(may, 31, 1962)).
   phone_list(person(apurva, baxi), "438-8400", b_date(jun, 13, 1980)).
   phone_list(person(apurva, dave), "767-8463", b_date(jun, 22, 1986)).
   phone_list(person(apurva, bhatt), "555-5653", b_date(jul, 22, 1981)).
   phone_list(person(apurva, patel), "767-2223", b_date(aug, 13, 1981)).
   phone_list(person(apurva, dangar), "438-8400", b_date(sep, 22, 1981)).
```

```
    phone_list(person(apurva, pandya), "438-8400", b_date(oct, 31, 1952)).
    phone_list(person(apurva, vaishnav), "555-1212", b_date(nov, 22, 1984)).
    phone_list(person(apurva, gor), "767-2223", b_date(nov, 04, 1987)).
    phone_list(person(apurva, kanani), "438-8400", b_date(dec, 31, 1981)).


    convert_month(jan, 1).
    convert_month(feb, 2).
    convert_month(mar, 3).
    convert_month(apr, 4).
    convert_month(may, 5).
    convert_month(jun, 6).
    convert_month(jul, 7).
    convert_month(aug, 8).
    convert_month(sep, 9).
    convert_month(oct, 10).
    convert_month(nov, 11).
    convert_month(dec, 12).
```

How do compound data objects help in this program? This should be easy to see when you examine the code. Most of the processing goes on in the get_months_birthdays predicate.

First, the program makes a window to display the results.

After this, it writes a header in the window to help interpret the results.

Next, in get_months_birthdays, the program uses the built-in predicate date to obtain the current month.

After this, the program is all set to search the database and list the people who were born in the current month. The first thing to do is find the first person in the database. The call phone_list(Person, _, Date) binds the person's first and last names to the variable Person by binding the entire functor person to Person. It also binds the person's birthday to the variable Date.

Notice that you only need to use one variable to store a person's complete name, and one variable to hold the birthday. This is the power of using compound data objects.

Your program can now pass around a person's birthday simply by passing on the variable Date. This happens in the next subgoal, where the program passes the current month (represented by an integer) and the birthday (of the person it's processing) to the predicate check_birthday_month.

Look closely at what happens. Turbo Prolog calls the predicate check_birthday_month with two variables: The first variable is bound to an integer, and the second is bound to a birthday term. In the head of the rule that defines check_birthday_month, the first argument, This_month, is matched with the variable Mon. The second argument, Date, is matched against b_date(Month, _,_).

Since all you're concerned with is the month of a person's birthday, you have used the anonymous variable for both the day and the year of birth.

The predicate check_birthday_month first converts the symbol for the month into an integer value. Once this is done, Turbo Prolog can compare the value of the current month with the value of the person's birthday month. If this comparison succeeds, then the subgoal check_birthday_month succeeds, and processing can continue. If the comparison fails (the person currently being processed was not born in the current month), Turbo Prolog begins to backtrack to look for another solution to the problem.

The next subgoal to process is write_person. The person currently being processed has a birthday this month, so it's OK to print that person's name in the report. After printing the information, the clause fails, which forces backtracking.

Backtracking always goes up to the most recent non-deterministic call and tries to re-satisfy that call. In this program, the last non-deterministic call processed is the call to phone_list. It is here that the program looks up another person to be processed. If there are no more people in the database to process, the current clause fails; Turbo Prolog then attempts to satisfy this call by looking further down in the database. Since there is another clause that defines get_months_birthdays, Turbo Prolog tries to satisfy the call to get_months_birthdays by satisfying the subgoals to this other clause.

Declaring Domains of Compound Objects

Let's see, how domains for compound objects are defined. After compiling a program that contains the following relationships:

```
        owns(apurva, book("Parva", "S L Bhyrappa")).
```

and

```
         owns(apurva, horse(blacky)).
```

you could query the system with this goal:

```
        owns(apurva, X)
```

The variable X can be bound to different types of objects: a book, a horse, or perhaps other objects you define. Because of your definition of the *owns* predicate, you can no longer employ the old predicate declaration of owns:

```
   owns(symbol, symbol)
```

The second argument no longer refers to objects belonging to the domain symbol. Instead, you must formulate a new declaration to the predicate, such as

```
   owns(name, articles)
```

You can describe the articles domain in the domains section as shown here:

```
    DOMAINS
        articles = book(title,author); horse(name)
                            /* Articles are books or horses */
        title, author, name = symbol
```

The semicolon is read as or. In this case, two alternatives are possible: A book can be identified by its title and author, or a horse can be identified by its name. The domains title, author, and name are all of the standard domain symbol.

More alternatives can easily be added to the domains declaration. For example, articles could also include a boat, a house, or a bankbook. For a boat, you can make do with a functor that has no arguments attached to it. On the other hand, you might want to give a bank balance as a figure within the bankbook. The domains declaration of articles is therefore extended to:

```
    articles     = book(title, author) ; horse(name) ;
                   boat ; bankbook(balance)
    title, author, name = symbol
    balance    = real
```

Here is a full program that shows how compound objects from the domain articles can be used in facts that define the predicate owns.

```
DOMAINS
        articles = book(title, author) ;
        horse(name) ; boat ;
        bankbook(balance)
        title, author, name = symbol
        balance = real

PREDICATES
        owns(name,articles)

CLAUSES
        owns(apurva, book("Parva", "S.L.Bhyrappa")).
        owns(apurva, horse(blacky)).
        owns(apurva, boat).
        owns(apurva, bankbook(1000)).
```

Now compile and run the program with the following goal: owns(apurva, Thing) and discuss the output.

Exercises

1. Modify the sample program II so that it will also print the birth dates of the people listed. Next, add telephone numbers to the report.
2. Write a prolog program for an IT company that store employee details like Name, Address, Department, Position, Salary. Use compound objects to properly formulate the representation of each employee details. Find out
   I. employee(s) with salary higher than a threshold
   II. employee(s) available in a particular department
   III. employee(s) holding a particular position
3. Try following link and verify that whether system is intelligent or not and justify your answer.
   1. www.manifestation.com/neurotoys/eliza.php3