

Prolog

1. Introduction

Prolog is a computer language which allows one to *program in logic*. The language was invented in 1970 by Alain Colmerauer and his associates. By programming in logic, we mean that instead of explicitly stating what a machine should do to solve a problem (*procedural programming* [Pascal, C, FORTRAN]), or specifying a functional transformation from data to answers (*functional programming* [LISP without `set!`]), we set forth the rules which govern the problem we are trying to solve and then ask questions to find the solution (*logic programming*). Prolog, like LISP, has a fanatical following within the Artificial Intelligence community.

Prolog is an interactive system like LISP: there is no intermediate compilation stage. Prolog is available on `cec1` and `cec2` as `'cprolog'`.

2. The Language

2.1. Primitives

In Prolog we specify the relations between objects. Hence, there are primitive objects and primitive relations. There are two types of primitive object: constants and variables. Constant objects begin with a lower-case letter and are interpreted literally. Variables begin with a capital letter and can "match" any object at all. Thus `eve` represents the literal string "eve," but `Eve` stands for "anything at all."

Primitive relations are expressed as facts about objects. Thus the relation

```
mother_of(cain, eve). (1)
```

means that the specific person `eve` is the mother of the specific person `cain`. Likewise, the relation

```
mother_of(Cain, eve).
```

means that the specific person `eve` is the mother of *every* object, because `Cain` is a variable that matches anything.

Finally, we may ask questions about relationships:

```
?- mother_of(cain, eve).
```

causes Prolog to answer `yes` if the expression (1) had been entered. The `'?-'` introduces questions that we ask Prolog.

2.2. Means of combination

The most elementary means of combination is simple juxtaposition of facts. If we enter the lines

```
mother_of(cain, eve). (2)
mother_of(abel, eve).
```

then we express our belief that `eve` was the mother of `cain` and `abel`.

The second means of combination uses the `','`:

```
?- mother_of(cain, eve), mother_of(abel, eve).
```

This will cause Prolog to reply `yes` if both of the comma-separated assertions are in the database. The `','` is a logical *and* in Prolog.

2.3. Means of abstraction

The *rule* is the means of abstraction in Prolog. Rules are formed by naming combinations of facts and other rules, much as procedures in LISP are made by naming combinations of functions. An example is the rule

```
sibling_of(Sib1, Sib2) :- mother_of(Sib1, Mother), mother_of(Sib2, Mother).
```

Here we mean that if two people have the same mother, then they are siblings. If we ask

```
?- sibling_of(cain, abel).
```

Prolog will answer *yes* if the facts (2) had been defined. The significance of this is that we have defined an entirely new relation using old rules.

3. Computational Model

When we type facts and rules into Prolog, they are entered into a database in the order typed. When we ask a question, it is viewed by Prolog as a *goal* to be *satisfied*. To satisfy the goal, Prolog scans the database from top to bottom looking for a matching fact or rule. Thus if our database is:

```
mother_of(cain, eve).                (1)
mother_of(abel, eve).                (2)
mother_of(abel, alice).              (3)
/*          G1 (first subgoal)      G2 (second subgoal)      */
sibling_of(Sib1, Sib2) :- mother_of(Sib1, Mother), mother_of(Sib2, Mother).
```

and we ask the question

```
?- mother_of(abel, eve).
```

Prolog responds *yes* because it found the fact in the database.

If there are variables in our query, Prolog responds with the *instantiations* (settings) of variables that were necessary to make the question true. So if we ask

```
?- mother_of(X, eve).
```

Prolog answers

```
X = cain
```

which is the first value for *X* that made our question true. (By true, we actually mean "derivable using the facts and rules in the database.") If we type *;*, Prolog responds with the next instantiation that makes the question true:

```
X = abel
```

If we once more type *;*, Prolog responds *no*, indicating that no more facts of the form we want can be derived from the database.

Suppose that we ask

```
?- sibling_of(abel, cain).
```

which we know should be derivable under the rules in the database. Prolog scans the database looking for matching facts and rules. The only entry that matches is the rule above. (Remember that *Sib1* and *Sib2* may stand for any constant or variable.) Prolog now instantiates *Sib1* to *abel* and *Sib2* to *cain* everywhere in the rule for *sibling_of*. After doing this we have two *subgoals* to satisfy: *mother_of(abel, Mother)* and *mother_of(cain, Mother)*, where the intermediate variable *Mother* in each expression must be the same. Prolog always works from left to right in satisfying subgoals, so we first look for a fact or rule of the form *mother_of(abel, Mother)*. The fact *mother_of(abel, eve)* matches, and so we instantiate all occurrences of *Mother* in the rule to *eve*. Now we attempt to satisfy the second subgoal, which, with the instantiation of *Mother*, is *mother_of(cain, eve)*. This is in the database and so our main goal succeeds and *yes* is printed.

As a final exercise, imagine that we ask the question

```
?- sibling_of(abel, X).
```

We scan the database and see that the last line matches the query, so we instantiate *Sib1* to *abel* there and attempt to

satisfy `mother_of(abel, Mother)` ($G1$). The second fact matches, and we instantiate `Mother` to `eve` and try to satisfy `mother_of(X, eve)` ($G2$). The first fact matches, and so we print out

```
X = cain
```

Now we type `;` to try to resatisfy the query. We therefore must resatisfy $G2$, so we uninstantiate `X` and scan the database from where we left off, at the second fact. We get a match immediately:

```
X = abel
```

We try to satisfy $G2$ once more, so again we uninstantiate `X` and scan, but this time $G2$ fails: there is no fact of the form `mother_of(X, eve)`. Now we back up one step and attempt to resatisfy $G1$. We look for a fact of the form `mother_of(abel, Mother)`, starting from the third fact, which is where we left off before. The third fact matches, so we try to satisfy $G2$: `mother_of(X, alice)`. We find no matching fact, so we again try to resatisfy $G1$, starting after the third fact. However, no more `mother_of` facts remain, so $G1$ fails. Lastly, we try to resatisfy our main goal, but no more `sibling_of` facts or rules remain. We print out `no` and stop.

4. Some Examples

The first example we will look at is some simple list manipulation rules. First we will need to know some Prolog list notation. In Prolog, the notation `[a, b, c]` is identical to the LISP `'(a b c)`. The `[` and `]` surround lists in Prolog, and the comma separates list items. The construction `[X|Y]` stands for the list whose first element is `X` and whose tail is `Y`. (This is Prolog's analog of the `car` and `cdr` operations of LISP.) The empty list is simply `[]`.

We are now ready to write a rule for finding the last element of a list. The last element of a list of one member is that member. This gives us the rule

```
last(X, [X]).
```

We use this rule as the basis in a recursive definition. If the list is of more than one element, the last element of the list is the last element of the tail of the list:

```
last(X, [_|Y]) :- last(X, Y).
```

(The variable `'_'` here is Prolog notation for a nameless variable. Multiple occurrences of `_` in a rule do *not* have to stand for the same thing.) One way to read this rule is "if `X` is the last element of a list `Y` then it is also the last element of the list formed by adding one thing to the front of `Y`."

Using the same ideas we may write `next_to`, which determines if two atoms are next to one another in a list:

```
next_to(X, Y, [X, Y|_]).
next_to(X, Y, [_|Z]) :- next_to(X, Y, Z).
```

The first rule here succeeds if first two elements of the list are `X` and `Y`. The second rule is the recursive case: it succeeds if `X` is next to `Y` in the tail of the list.

Our most ambitious project is to write a rule for appending two lists. To do this we note that appending the empty list to another list should simply produce the other list. If the first list is not empty, then we employ this rule: "if `Result` is `L1` appended to `L2`, then `Result` must have the same first element as `L1`." This is the translation of these ideas:

```
append([], L, L).
append([Head|L1], L2, [Head|Result]) :- append(L1, L2, Result).
```

Because of the way that Prolog works, we may define the first two rules in terms of `append`:

```
new_last(Elem, List) :- append(_, [Elem], List).

new_next_to(Elem1, Elem2, List) :- append(_, [Elem1,Elem2|_], List).
```

This is hard to believe at first because we (incorrectly) like to think of rules as functions with certain parameters as input and others as output. Specifically, we think of the first two parameters of `append` as input, but in `new_last`, the second and third parameters of `append` are input, and the first is not used at all. Prolog, however, thinks of rules as conditions which it must satisfy by making the necessary instantiations. Variables become output and input based on the instantiations Prolog makes. The best way to explain the action of `new_last` above is to read it as a rule: "Find the quantity `Elem` such that when you append an arbitrary list to it, the list `List` is produced." The rule `new_next_to`, likewise, succeeds when quantities `Elem1` and `Elem2` exist such that `List` is produced when we append arbitrary lists on either side of `Elem1` and `Elem2`.

The final example is a smaller version of the doctor program our class wrote earlier in the year.

```

/* Word changes */
change_word(you, i).          /* you --> i */
change_word(french, german). /* french --> german */
change_word(are, 'am not').
change_word(do, no).
change_word(X, X).          /* Default case */

/* Phrase changes */
change([], []).              /* End of recursion */
change([Head|Tail], [New_Head|New_Tail]) :- /* Otherwise... */
    change_word(Head, New_Head),          /* Change head */
    change(Tail, New_Tail).              /* and tail */

```

The facts of `change_word` tell how to change single words. We could allow for more words to be changed by adding more `change_word` facts. (The last fact picks up any words that were not in the change list. It matches everything but does not change anything.) The rules of `change` tell how to change a sentence by changing its first word and (recursively) its tail. If we ask the question

```
?- change([do, you, speak, french], X).
```

Prolog responds

```
X = [no, i, speak, german]
```

It is interesting to note that `change` can be driven backwards, that is, we can find the patient's question given the doctor's response. To the question

```
?- change(Y, [no, i, speak, german]).
```

Prolog answers

```
Y = [do, you, speak, french]
```

Prolog just uses the `change_word` mapping the other way.

5. Relationship To Formal Logic

Prolog is a first attempt at allowing people to program in logic. In logic programming, we supply rules for a model to follow and see what properties of the model those rules imply. The language finds answers to our questions about the model consistent with the rules without our having to specify how to find those answers. There are at least two advantages to this approach: we do not have to know how to find the solution, and, if we decide that we want an answer to another question about the same model, we need only ask the question, rather than finding another algorithm to answer that question.

A specific application of these ideas is known as *theorem proving*. We enter axioms into an automated theorem-prover, which will combine the axioms in new and increasingly clever ways to prove theorems about the system. For

example, we could take as our axioms the four postulates of Euclidean geometry and expect as output every theorem that you proved as a geometry student in middle school, and many others besides.

Prolog can, in fact, be seen as a theorem prover for the class of problems representable in a restricted form of the predicate calculus. The predicate calculus deals with statements like

$$\forall x (\text{man}(x) \rightarrow \text{human}(x)) \quad ,$$

which reads, "if someone is a man then they are human." Statements in the predicate calculus involve the quantifiers \forall and \exists together with the logical operations \leftrightarrow (biconditional), \rightarrow (implication), $|$ (or), $\&$ (and), and \neg (not). These statements may be translated into a set of *clauses*, and-ed together. Each clause is in turn composed of a group of *terms*, or-ed together. These terms are either functions of known constants (such as "human (john)") or functions of variables ("human (X)"). Each of these functions, or terms, may be negated or unnegated. Thus one clause looks like:

$$f_1(x_1) \mid f_2(x_2) \mid \cdots \mid f_k(x_k) \mid \neg g_1(y_1) \mid \neg g_2(y_2) \mid \cdots \mid \neg g_n(y_n) \quad .$$

Using the definition of implication and DeMorgan's rule, we rewrite this as

$$\left[g_1(y_1) \& g_2(y_2) \& \cdots \& g_n(y_n) \right] \rightarrow \left[f_1(x_1) \mid f_2(x_2) \mid \cdots \mid f_k(x_k) \right] \quad .$$

The entire predicate calculus statement is expressed as a set of these clauses. Now that we have put the clauses in this nice form, we would like to test theorems. We may do this by testing the logical not of the potential theorem against the clauses of the model. If the negation of the potential theorem results in a contradiction, then the potential theorem is proven, that is, it is a consequence of the clauses in the database. (If we derive the empty clause, which is the clause with nothing on either side of the implication sign, then we have found a contradiction.) The process of showing that adding a clause to the database results in a contradiction is called *resolution*. One step in the resolution process is a *unification*; it is accomplished by combining two clauses.

It can be shown that resolution is both *complete* and *correct*. Completeness means that if something is a theorem, resolution will be able to derive the empty clause. Correctness means that if resolution can derive the empty clause from a potential theorem, then it really is a theorem. These are the properties we want; the only problem is that we do not know the correct way to combine clauses: we might unify many clauses before we find a contradiction.

However, if we restrict ourselves to the subset of clauses which are of the form

$$\left[g_1(y_1) \& g_2(y_2) \& \cdots \& g_n(y_n) \right] \rightarrow f(x)$$

the problem becomes easier. This type of clause is called a *Horn clause*, and it can be expressed in Prolog as

$$f(x) \text{ :- } g_1(y_1) \text{ , } g_2(y_2) \text{ , } \cdots \text{ , } g_n(y_n).$$

(Remember that ' :- ' may be read "is implied by.") When we enter facts and rules into Prolog, they are entered into a database as clauses. Theorems are tested by asking questions of Prolog:

$$\text{?- } q(z).$$

In fact, the ' ?- ' is just syntactic sugar for the "headless" Horn clause

$$\text{:- } q(z).$$

Asking this question is equivalent to trying to add $\neg q(z)$ to the database. Prolog uses the resolution principle to attempt to derive the empty clause from the negation of the question. In fact, when we discussed the computational model for Prolog, we were stating the resolution principle for Horn clauses. When Prolog matches relation names and instantiates variables, it is unifying two Horn clauses.

