Prolog was invented in the early 1970s by Alan Colmerauer and his colleagues in Marseille: Their major interest was Natural Language Processing. The deductive mechanism behind Prolog is based on Robert Kowalski's work on refinements of resolution (SLD) for Horn clauses.

- NJIT has Sicstus Prolog developed by the Swedish Institute of Computer Science. http://www.sics.se
- An excellent Prolog - Amzi Logic Explore free for PLS (Both Linux and Windows). http://www.amzi.com

Horn Clauses

Horn Clauses are clauses that have at most one positive literal. If there is one positive literal, then the clause is a rule whose consequent is the single positive literal and whose antecedent is a conjunction of positive literals.

- Rules: H :- B1,...,Bn
- A fact is a single positive literal. Facts: H :-
- A goal (query) is a conjunction of negative literals. Goals :- B1,...,Bn

References

- Learn Prolog Now! by Patrick Blackburn, Johan Bos and Kristina Striegnitz
- Chapters 5 and 6 of Brachman and Levesque

Interaction

The user submits questions to the prolog system and receives answers based on information contained in the database of facts, and the rules that have been loaded into the prolog system.
A Sample Database

has_vacancy(harvard, secretary).
has_vacancy(prentice_hall, author).
has_vacancy(ibm, salesman).
has_vacancy(hertz, driver).
has_vacancy(nasa, programmer).
has_vacancy(prentice_hall, secretary).

trained_as(michael, programmer).
trained_as(fred, taxidermist).
trained_as(mary, driver).
trained_as(joe, secretary).
trained_as(michael, salesman).
trained_as(elizabeth, secretary).

Queries

• ?-clear_thinking(elizabeth).
• ?-clear_thinking(fred).
• ?-clear_thinking(X)
• ?-imaginative(X), hard_working(X).
Note that if there is more than one object satisfying the query, the user can type a semicolon (;) after the answer and Prolog will search for another binding for the variables. This can continue until Prolog cannot find another binding. It will then return no.

But Prolog can do much more than mere retrieval of facts!

Rules Continued

If the above rule is added to the database other plausible rules are:

acceptable(Candidate, Employer, Skill) :-
    has_vacancy(Employer, Skill),
    trained_as(Candidate, Skill).

acceptable(Candidate, Employer, Skill) :-
    has_vacancy(Employer, Skill),
    \+(trained_as(Candidate, Skill),
    could_be_trained_as(Candidate, Skill)

could_be_trained_as(X, secretary) :-
    accurate(X),
    literate(X),
    outgoing(X).

could_be_trained_as(X, programmer) :-
    clean_thinking(X),
    accurate(X),
    intelligent(X).

could_be_trained_as(X, driver):-
    co_ordinated(X)
    hard_working(X),

Examples

?- could_be_trained_as(michael, secretary).

?- could_be_trained_as(mary, programmer).
Negation

Note that the \+ is the negation operator in Sicstus Prolog. In Amzi prolog the negation operator is the standard not as in not (member(X, [a,b,c]))

Using Prolog

Type your program into a file and save it. Save it with the suffix pl as in kb.pl. Then enter prolog.

?- listing.

?- [kb2].

?- listing.
?- halt.

A Family

child_of(charles, philip).
child_of(charles, liz).
parent_of(philip,charles).
parent_of(liz,charles).
father_of(X,Y):- parent_of(X,Y), male(X)

Descendant

Consider the problem of trying to specify the concept of descendant.

descendant_of(X,Y) :- child_of(X,Y).
descendant_of(X,Y) :- grandchild_of(X,Y).
descendant_of(X,Y) :- great_grandchild_of(X,Y).

grandchild_of(X,Y) :- child_of(X,Y), child_of(Z,Y)
great_grandchild_of(X,Y) :- child_of(X,Z),
                           grandchild_of(Z,Y).

great_great_grandchild_of(X,Y) :- child_of(X,Z),
                            great_grandchild_of(Z,X).

Tedious !, Incomplete ! descendants of Y are Y's children, along with their descendants
Recursive Rules

But with recursive rules this is easy.

X is a descendant of Y either if X is a child of Y, or if X is a descendant of a child of Y.

\[
\text{descendant_of}(X, Y) :- \text{child_of}(X, Y).
\]
\[
\text{descendant_of}(X, Y) :- \text{child_of}(C, Y),
\text{descendant_of}(X, C).
\]

?- \text{descendant_of}(X, elizabeth).

Structured Objects

Use of term structure enables one to fully utilize relatively simple expressivity of Prolog.

Object-kind(component1, component2, ......)

1. date(Day, Month, Year)
   date(31, january, 1988)
   date(25, december,1990)

Structured Objects Continued

2. meal(starter, main_course, desert)
   meal
   starter main_course desert
   melon ginger steak peas chips peaches cream
   main_course(steak, peas, chips)
   meal(melon, ginger),
   main_course(steak, peas, chips),
   desert(peaches, cream)

3. book(Author, Title, Classification)
   book( shakespeare, macbath, qt-13....)

   date_of_birth(Person, Date)
   date_of_birth( fred, date(1, february, 1959)).
   date_of_birth( shakespeare, date(26, april, 1564))
   ?- date_of_birth( shakespeare, D).
   D= date(26, april, 1564)).

   ?- date_of_birth( P, date(24, april, 1564)).
   P=shakespeare
Example: Library Catalogue

\[
\text{in\_library( book( melville, moby\_dick, 4r\_14\_s8)).}
\]

\[
\text{in\_library( book( shakespeare, romeo\_and\_juliet, 4r\_49\_s35)).}
\]

\[
\text{on\_loan( book( melville, moby\_dick, 4r\_14\_s8), robinson, date(21, november, 1988)).}
\]

\[
\text{on\_loan( book( shakespeare, romeo\_and\_juliet, 4r\_49\_s25), wilson, date(7, september, 1988)).}
\]

Lists

But the most important structured object of all is the List – treated specially in Prolog.

\[
[ ] - \text{empty list}
\]

\[
[ \text{tennis, baseball, sailing, reading, judo} ]
\]

\[
[ \text{computing, programming, prolog, AI} ]
\]

\[
[ \text{tennis | X} ]
\]

Lists Cont

(compare as : head , tail) The head of the list above is tennis and X is the tail. Example:

\[
\text{all\_rich( List )}
\]

A list is all rich if either the list is empty or the list has the structure \([ \text{Person1 | Tail} ]\) and \text{Person1} is rich and \text{Tail} is all rich.

\[
\text{all\_rich([ ]).}
\]

\[
\text{all\_rich([Person1 | Tail]) :- rich(Person1), all\_rich(Tail ).}
\]

Example: Member

A very simple program defines the member relation

\[
\text{Definition of Member}
\]

\[
\text{member(X, [X | _]).}
\]

\[
\text{member(X, [ _ | Y]) :- member(X,Y).}
\]

\[
?\text{- member(d, [a,b,c,d,e,f,g])}.}
\]

\[
\text{YES}
\]

\[
?\text{- member(2, [3,a,4,f])}.}
\]

\[
\text{NO}
\]
Example: Append

Another very simple function defines the append rule:

```prolog
append([], L, L).
append([X|L1], L2, [X | L3]) :- append(L1, L2, L3).
?- append(X, Y, [a, b, c]).
X= []
Y= [a, b, c]?

X= [a];
Y= [b, c]?

X= [a, b],
Y= [c]?

X= [a, b, c],
Y= []?

no
```

Arithmetic

A variable is an expression.

```prolog
?- X = 2 * 8 + 5.
X=21

?- X = 12, X is 10.
no.

?- X is 12, Y is 3 * X - 1.
X=12, Y=35.
```

DB Continued

```prolog
in_range(N, Lower, Upper)
    :- N >= Lower, N <= Upper.

?- in_range(10, 1, 100)
Yes

?- in_range(0, 1, 10)
No.

Sum of a List

Sum([], 0)
Sum([Head | Tail], S) :- Sum(Tail, T), S is Head + T.
```

Cut!

```prolog
foo :- a, b, c, !, d, e, f

When a cut is encountered as a goal, the system thereupon becomes committed to all choices made since the parent goal was invoked. All other alternatives are discarded. Hence an attempt to re-satisfy any goal between the parent goal and the cut goal will fail.

facility(Pers, Fac):-
    book_overdue(Pers, Book), !,
    basic_facility(Fac).
```
facility(Pers, Fac):= general_facility(Fac).
basic_facility(references).
basic_facility(enquiries).
additional_facility(borrowing).
additional_facility(inter_library_loan).
general_facility(X) :- basic_facility(X).
general_facility(X) :- additional_facility(X).
client('A. Sones').
client('R.Scherl').
?-client(X), Facility(X,Y).