Prolog (part 2)

Andrea Sterbini – sterbini@di.uniroma1.it

Lists (dynamic, heterogeneous)

```
List = [ one, two, three, four ]        % list syntax

[ Head | Tail ] = List                  % how to extract the first element
    Head  = one                         % fails if the list is empty
    Tail    = [ two, three, four ]

[ First, Second | Rest ] = List         % extracting first and second element
    First       = one                   % fails if the list has less than 2 elements
    Second    = two
    Rest        = [ three, four ]

EmptyList = []                          % the empty list

is_empty([]).                           % test for empty list through unification

length([], 0).                          % recursively compute the list length
length([H|T], N1) :- length(T,N), N1 is N + 1.
```

Predicates on lists

```
% list concatenation/split (if used backward)

append([], B, B).                          % B if A is empty
% else attach the first in front of the result of appending the rest to A
append([ H | T], B, [H | C] ) :- append(T,B,C).


% member check/generation

member( A, [ A | _ ] ).                    % A is member if first element
member( A, [ _ | T] ) :- member(A, T).     % or if member of the rest
```

Predicates are relations and works in many ways/directions

```
append( [a], [b, c], L)          => L = [a, b, c]
append( A,   [b, c], [a, b, c])  => A = [a]
append( A,   B,      [a, b, c])  => A = [],        B = [a, b, c] ;
                                    A = [a],       B = [b, c] ;
                                    A = [a, b],    B = [c] ;
                                    A = [a, b, c], B = [] ; fail

member( a, [a, b, c] )  => true
member( A, [a, b, c] )  => A=a   or  A=b   or  A=c
member( a, B)           => B = [a|_] ;        % list starting with a
                           B = [_,a|_] ;      % list with a in 2° place
                           B = [_,_,a|_] ;    % list with a in 3° place
                           … (infinite solutions)
```

Functional programming

Predicates can be used as if they were functions or to test values
You just add an argument to collect the result

```
square( X, Result ) :- Result is X * X.          % function
is_odd(X) :- 1 is X mod 2.                       % test=compute+unify
```

You can map functions over lists (with the apply library)

```
List = [ 1, 2, 3, 4 ], maplist( square, List, List1 ).
    => List1 = [ 1, 4, 9, 16 ]
```

Or get all elements satisfying some property

```
List = [1, 2, 3, 4], include(is_odd, List, Odd).
    => Odd = [1, 3]
List = [1, 2, 3, 4], partition(is_odd, List, Odd, Even).
    => Odd = [1, 3]      Even = [2, 4]
```

What if predicates are used “backward”?

```
% find a list X that is partitioned this way
partition(is_odd, X, [1,3], [2,4]).

    [1,3,2,4] ; [1,2,3,4] ; [1,2,4,3] ; [2,1,3,4] ; [2,1,4,3] ; [2,4,1,3]

% What if we use maplist “backward”?
maplist(square, X, [1, 4, 9]).
            Arguments are not sufficiently instantiated
            In: [3] 1 is _1680*_1682
            ...
% We need a better definition of square(N,N2)
square(N, N2) :- nonvar(N), N2 is N*N.                    %if N is known
square(N, N2) :- var(N), between(1,N2,N), N2 is N*N.     % else
```

Or else you could collect all solutions by:

All solutions of a Predicate: bagof(Term, Predicate, ListOfTerms)

```
?- bagof( odd(X), (member(X, [3, 2, 3, 4]), 1 is X mod 2), Odd)
        => Odd = [ odd(3), odd(3) ]
```

Unique solutions: setof(Term, Predicate, Set)

```
 ?- setof( odd(X), (member(X, [3, 2, 3, 4]), 1 is X mod 2), Odd)
        => Odd = [ odd(3) ]
```

Just repeat DoSomething for each solution of a Predicate:

forall(Predicate, DoSomething)

```
?- forall( member(El, [1, 2, 3]), writeln(El) ).
        1
        2
        3
```

Meta-programming

You can build terms from lists and viceversa

term(1, two, three) =.. [term, 1, two, three]

You can call/prove predicates built from data

call(Term, AdditionalArg, ...)

You can add/remove new facts or clauses to/from memory

asserta(Head :- Body)
asserta(Fact)
retract(FactOrClause)

assertz(Head :- Body)
assertz(Fact)

Alternative syntax to write parsers/generators
Definite Clause Grammars (DCG)

Two arguments are added to each grammar rule:

- the list of input tokens
- the remaining list of tokens not yet consumed

RULE READ	TRANSFORMED TO

sentence -->
  subject,
  verb,
  complement.
	sentence(Words, Rest3) :-
	  subject(Words, Rest1),
	  verb(Rest1, Rest2),
	  complement(Rest2, Rest3).

%special: terminal tokens verb --> [run].	% simply expected as next token verb([run | Rest], Rest).

Grammar example

```
sentence          --> subject, verb, object.
subject           --> article(Gender), actor(Gender).
object            --> article(Gender), object(Gender).
article(female)   --> [ la ].
article(male)     --> [ il ].
actor(_)          --> [ chirurgo ].
actor(female)     --> [ elefantessa ].
actor(male)       --> [ elefante ].
verb              --> [ mangiava ].
verb              --> [ guardava ].
object( female )  --> [ insalata ].
object( male )    --> [ cavolfiore ].
```

Output

[la, chirurgo, mangiava, la, insalata]
[la, chirurgo, mangiava, il, cavolfiore]
[la, chirurgo, guardava, la, insalata]
[la, chirurgo, guardava, il, cavolfiore]
[la, elefantessa, mangiava, la, insalata]
[la, elefantessa, mangiava, il, cavolfiore]
[la, elefantessa, guardava, la, insalata]
[la, elefantessa, guardava, il, cavolfiore]
[il, chirurgo, mangiava, la, insalata]
[il, chirurgo, mangiava, il, cavolfiore]
[il, chirurgo, guardava, la, insalata]
…

%TASK: how can add number constraints?

Common extensions

Grammars

grammar rules map easily to Prolog predicates, both for parsing and for text generation

Constraints

the domain of the possible values of a variable can be constrained in many ways (e.g. the sudoku game)

OOP

terms could represent objects and their properties

rules could represent methods

GUI

widgets, events, callbacks and so on

Constraint example (Sudoku)

DEMO

Programming styles

Single threaded

Declarative: data AND rules

- declarative data => relational data representation (SQL-like)

Functional: rules as functions transforming data

Meta-programming: programs that BUILD programs

Predicate/Relations can be used in many directions

Recursion, recursion everywhere!

Parallelism in some particular Prolog (Sicstus, Parlog, GHC)

Simple multiprocessing with the ‘spawn’ library

Prolog Pro/Cons for teaching

PRO

- Focus on data abstraction
- Focus on relations instead than procedures
- easy Natural Language processing and generation
- easy Symbolic manipulation (Math, Algebra, Physics, …)
- AI
- Recursion everywhere!

CONS

- Not typed (but you can use terms for dynamic typing)
- There is no really nice IDE (or you can use Eclipse PDT)
- Recursion everywhere!