Prolog (part 2)

Andrea Sterbini – sterbini@di.uniroma1.it

Recall:

FACT

```
term(argument1, argument2, arg3 …).     % rule always true
```

RULE

```
head(arg1, arg2, …) :-          % to prove this head
    body1(…),                   % we must prove this
    body2(…),                   % AND this
    …                           % …
    bodyN(…).                   % AND this
```

Details on rule execution

To prove a predicate (e.g. a prolog term) we must search for:

1) a rule with the same head (should unify with the term to prove)
2) or a fact with same term (which also should unify)

i.e.:

- the term functor must be the same
- the number of arguments must be the same
- each argument must unify with the corresponding argument

This is generally used to selectively match the predicate clauses

Example: see next slide

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).                    % base case: an empty list has length 0
length( [H|T], N1) :- length(T,N), N1 is N + 1. % recursive case: compute the list length
```

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
                % NOTICE: member obviously should fail if list empty
```

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]).      % is cannot be used “backward” in square
            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 look for
                                                % some integer N such that N*N = N2
```

Meta-programming

You can build terms from lists and viceversa with =..

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

You can call/prove predicates built from data

```
call( Term, AdditionalArg, ... )
```

(this allows using partial predicates)

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

```
% add at the beginning          % add at the end
asserta( Head :- Body )         assertz( Head :- Body )
asserta( Fact )                 assertz( Fact )
                retract( FactOrClause )     % delete FIRST matching rule
                retractall( FactOrClause ) % delete ALL matching rules
```

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

RULE READ FROM FILE

```
sentence -->
    subject,
    verb,
    complement.

%special: terminal tokens
verb --> [ run ].
```

IS TRANSFORMED TO

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

% simply expected as next token
verb( [ run | Rest ], Rest ).
```

Two arguments are added to each grammar rule:

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

Grammar example (with gender agreement)

```
sentence --> subject, verb, com_object.
subject    --> article(Gender), actor(Gender).  % same gender for article & actor
com_object      --> article(Gender), object(Gender). % same gender for article &
object
article(female)  --> [ la ].                  % female article
article(male)    --> [ il ].                  % male article
actor( _ )       --> [ chirurgo ].            % surgeon is both male/female in Italian
actor(female)    --> [ elefantessa ].         % female elephant
actor(male)      --> [ elefante ].            % male elephant
verb             --> [ mangiava ].
verb             --> [ guardava ].
object( female ) --> [ insalata ].            % salad is female in Italian
object( male )   --> [ cavolfiore ].          % cauliflower is male in Italian
```

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 we 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

Examples

Limericks

Grammar

Constraints (Sudoku)

Algebraic simplification?

Algebraic derivatives?