Logic Programming, 16-17

Inês Dutra
DCC-FCUP
ines@dcc.fc.up.pt (room: 1.31)

October 6, 2016

Using Definite Clause Grammars

- **DCG**: Definite Clause Grammar is a formal language used to define other languages. Based on Horn clauses.
- Example:

```
program --> rule; fact.
```

- Mostly used to define grammars and for natural language processing.
- A DCG usually evaluates a sentence written in the structure defined by the user and tells if the sentence is syntactically correct.

Using Definite Clause Grammars

- DCGs are internally converted to Prolog: two arguments are added to each DCG literal. Ex:

```
DCG: sent --> noun_phrase, verbal_phrase, complement.

Prolog: sent(S0,S) :- noun_phrase(S0,S1),
                      verbal_phrase(S1,S2),
                      complement(S2,S).
```

Using Definite Clause Grammars

- Internal representation of a DCG terminal symbol: Prolog fact. Prolog classifies the term, removes it from the ist as correctly evaluated, and returns the remaining list to be evaluated. Ex:

  ```
  DCG: determiner --> [a].
  Prolog: determiner([a|R],R).
  ```

- Semantic actions can be added to DCGs in Prolog syntax. This is a way of mixing DCG with pure Prolog code. Semantic actions need to be added in Prolog syntax surrounded by curly brackest. These actions are not part of the grammar being defined. They are actions that need to be taken whenever Prolog finds a given term.

Using Definite Clause Grammars

- Example:

```
DCG: constant(Dic,I) --> [monday],
                          {lookup(monday,Dic,I)}.
Prolog: constant(Dic,I,[monday|R],R) :-
                          lookup(monday,Dic,I).
```

- In this program, after finding the term 'monday', Prolog inserts it in a symbol table and returns an index to that entry in the table. The example shows the syntax in DCG and the syntax in Prolog.

Using DCGs to define a subset of the English syntax

```
sentence(sentence(NP,VP)) -->
        noun_phrase(NP),
        verb_phrase(VP).

noun_phrase(np(D,N,C)) -->
        determiner(D),
        noun(N),
        rel_clause(C).
noun_phrase(np(PN)) -->
        proper_noun(PN).
```

Using DCGs to define a subset of the English syntax

```
verb_phrase(vp(TV,NP)) -->
        trans_verb(TV),
        noun_phrase(NP).
verb_phrase(vp(IT)) -->
        intrans_verb(IT).

rel_clause(rc(that,VP)) -->
        [that],
        verb_phrase(VP).
rel_clause(rc([])) --> [].
```

Using DCGs to define a subset of the English syntax

```
determiner(det(every)) --> [every].
determiner(det(a)) --> [a].

noun(noun(man)) --> [man].
noun(noun(woman)) --> [woman].

proper_noun(pn(john)) --> [john].

trans_verb(tv(loves)) --> [loves].

intrans_verb(iv(lives)) --> [lives].
```

Using DCGs to define a subset of the English syntax, and convert the sentence to logic

```
:-op(500,xfy,&).
:-op(600,xfy,'->').

sentence(P) -->
        noun_phrase(X,P1,P),
        verb_phrase(X,P1).

noun_phrase(X,P1,P) -->
        determiner(X,P2,P1,P),
        noun(X,P3),
        rel_clause(X,P3,P2).
noun_phrase(X,P,P) -->
        proper_noun(X).
```

Using DCGs to define a subset of the English syntax, and convert the sentence to logic

```
verb_phrase(X,P) -->
        trans_verb(X,Y,P1),
        noun_phrase(Y,P1,P).
verb_phrase(X,P) -->
        intrans_verb(X,P).

rel_clause(X,P1,(P1&P2)) -->
        [that],
        verb_phrase(X,P2).
rel_clause(_,P,P) --> [].
```

Using DCGs to define a subset of the English syntax, and convert the sentence to logic

```
determiner(X,P1,P2,all(X,(P1->P2))) --> [every].
determiner(X,P1,P2,exists(X,(P1&P2))) --> [a].

noun(X,man(X)) --> [man].
noun(X,woman(X)) --> [woman].

proper_noun(john) --> [john].

trans_verb(X,Y,loves(X,Y)) --> [loves].

intrans_verb(X,lives(X)) --> [lives].
```

Programming in Prolog: trees

- **a+b*c**, **+(a,*(b,c))**

```
  +
 / \
a   *
   / \
  b   c
```

- sentence(noun(john),verb_phrase(verb(likes),noun(mary)))

```
  sentence
    /  \
 noun  verb_phrase
  /      /    \
john   verb  noun
        |     |
       likes  mary
```

Obtaining Multiple Solutions

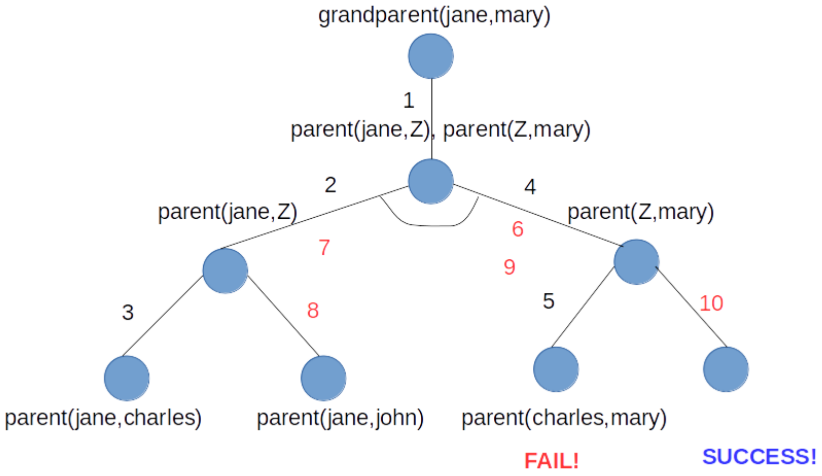
- a goal can have multiple solutions (reached through different clauses).
- *Backtracking* is used in Prolog when a fail occurs or when we want to obtain multiple solutions.
- When trying to satisfy a goal Prolog annotates that goal as a “choice-point”.
- If the goal fails, Prolog undo all work done so far till the last choicepoint created.
- It starts the search again, from this new choicepoint, looking for a new alternative to the goal.

Backtracking – Example

```
grandparent(X,Y) :- parent(X,Z), parent(Z,Y).

parent(jane,charles).
parent(john,mary).
parent(fred,jane).
parent(jane,john).

?- grandparent(jane,mary).
```



After step 5: backtracking! Step 6: undo computation (cleans var Z)
Step 7: try new alternative to parent(jane,Z)
Step 8: succeeds with Z=john Step 9: tries parent(Z,mary), with Z=john
Step 10: Succeeds!

The cut operator – **!** *(cut)*

- Controlled by the programmer.
- Reduces the search space.
- Implements exceptions.
- In combination with **fail**.
- Examples:

```
append([],X,X) :- !.
append([A|B],C,[A|D]) :- append(B,C,D).

not(P) :- call(P),!,fail.
not(P).
```

The cut operator – comments about examples

- In the first example: reduction of the search space (pruning solutions).
- Used when queries are of the kind:
  `append(L1,L2,[a,b,c])`, where there are multiple solutions (various values of `L1` and `L2` can produce the list `[a,b,c]`).
- The second example implements a cut-fail combination. It implements *negation as failure*. If `P` is true, i. e., Prolog satisfies goal P, then `not(P)` returns a failure (not(P) must be false) (falso). If Prolog can not satisfy goal `P`, then `not(P)` returns true from the second clause.

Builtin Predicates

- Input/Output.
  - Opening and Closing disk files (multiple): `open(Fd,name,rwa)`, `close(Fd)`.
  - Reading files (one at a time): `see, seeing, seen`
  - Writing files one at a time): `tell, telling, told`
  - Writing Prolog terms and characters: `nl, tab, write, put, display, format`
  - Reading terms and chars: `get, get0, read, skip`
  - Consulting (loading) programs: `[ListOfFiles], consult, reconsult` (short form for reconsulting: `[-FileName]`
  - Compiling programs: `compile(file)` (nb. dynamic and static predicates – `:-dynamic pred/n`, `:- static pred/n`)

Builtin Predicates

- Defining new operators.
  - position
  - precedence
  - associativity
- Possible operators: ('f' is the operator's position - infix, posfix or prefix, 'y' represents expressions that can contain other operators with precedence class greater than or equal than the operator we are defining, 'x' represents expressions that contain operators with precedence class greater than the precedence of the operator we are defining)
  - binary: `xfx`, `xfy`, `yfx`, `yfy`
  - unary: `fx`, `fy`, `xf`, `yf`

Builtin Predicates

- Examples:

```
:- op(255,xfx,':-').
:- op(40,xfx,'=').
:- op(31,yfx,'-').
```

- Need to check the manual of the Prolog system to know what are the characteristics of the pre-defined operators. Your own definition may override some already existing operator.

Entering new clauses (not in querying mode)

- Alternative 1: consult some file that was edited offline.
- Alternative 2: `[user]`. This is the mode for entering new clauses during your opened Prolog session. Prolog dislays another prompt ad waits the user to type in new clauses. To finish and go back to query mode, type `Ctrl-D` in unix e `Ctrl-Z` in Windows. Clauses entered this way will last only during your current session.
- Alternative 3: using the builtin predicate: `assert`. Only allowed for predicates declared as `dynamic`.
- `asserta(p(a,b))` insere no início do procedimento `p`, uma nova cláusula `p(a,b)`..
- `assertz(p(a,b))` insere no final do procedimento `p`, uma nova cláusula `p(a,b)`..

Builtin predicates

- Success and failure: `true` e `fail`.
- Types of terms: `var(X)`, `atom(X)`, `nonvar(X)`, `integer(X)`, `atomic(X)`.
- Meta-programming: `clause(X,Y)`, `listing(A)`, `retract(X)`, `abolish(X)`, `setof`, `bagof`, `findall`.
- More meta-programming: `functor(T,F,N)`, `arg(N,T,A)`, `name(A,L)`, `X =.. L`.
- Affecting backtracking: `repeat`.
- Conjunction, Disjunction and execution of predicates: `X, Y`, `X; Y`, `call(X)`, `not(X)`.
- Debugging programs: `trace`, `notrace`, `spy`, `nospy`.

Lists

- Member relationship: relation `member`, number of arguments: 2, the element to be looked for and a list (of elements, not necessarily a set).

  ```
  /* X was found, thus it belongs to the list */
  member(X,[X|_]).
  /* X was not found yet, it may be
     in the remaining list */
  member(X,[Y|L]) :- member(X,L).
  ```

Lists

- Concatenation of two lists: relation `concat`, number of arguments: 3, two input lists and the resulting list.

```
/* concatenation of an empty list with any other list
   is that list */
concat([],L,L).
/* if one concatenates L1 with L2 and obtain L3, then
   if one adds one new element to L1, the result L3
   must contain this new element. In other words: if
   one knows how to concatenate a list of n-1 elements
   with any other list, one knows how to concatenate a
   list of n elements with any other list.
*/
concat([H|L1],L2,[H|L3]) :- concat(L1,L2,L3).
```

Lists

- Find the last element of a list: relation `last`, number of arguments: 2, the element to be found and the list. Very similar to the program `member`.

```
/* if the list has exactly one element,
   this is the last. */
last(X,[X]).
/* if the list has more than one element, the last
   is not the first one. The last one can only
   be found in the remaining of the list. */
last(X,[_|L]) :- last(X,L).
```

Lists

- Reverse of a list: relation `rev`, number of arguments: 2, an input list and the resulting list reversed.

```
/* reverse of an empty list is the empty list */
rev([],[]).
/* reverse of a list L of length n is the reverse of
   the same list L1 of length n-1 (less the first
   element H) concatenated with the first element H
*/
rev([H|L1],R) :- rev(L1,L2), concat(L2,[H],R).
```

Lists

- Length of a list: relation `length`, number of arguments: 2, a list and the resulting length. Idea: the length of a list n is defined by the length of a list of length (n-1) plus 1.

```
/* length of an empty list is zero */
length([],0).
/* length of a non-empty list ([H|L1]) is obtained by
   length of L1 + 1.
*/
length([H|L1],N) :- length(L1,N1), N is N1 + 1.
```

Lists

- Removing the first occurrence of element X from a list L: relation `remove`, number of arguments: 3, the input list, the element, and the resulting list.

```
/* remove X from an empty list returns
   the empty list */
remove([],X,[]).
/* if X is found, return the remaining list
*/
remove([X|L],X,L).
/* if X was not found yet, try to remove X from
   the remaining list and return a new list,
   hopefully without X. Y, different from X, needs to
   be in the resulting list  */
remove([Y|L],X,[Y|L1]) :- remove(L,X,L1).
```

Lists

- How to modify this program to remove not just the first occurrence of X but ALL Xs?
- Can you modify the order of the clauses?

Binary Search and Insertion

- Sorted binary dictionary:
  - relation `lookup`,
  - number of arguments: 3
    - a key to be inserted or looked for,
    - a dictionary,
    - resultant information about the key.
  - data structure: `bintree(K,E,D,I)`
    - `K`: key.
    - `E`: left subtree.
    - `D`: right subtree.
    - `I`: info about K.

Sorted binary dictionary

```
/* key was found or is inserted */
lookup(K,bintree(K,_,_,I),I).
/* key was not found yet. It is either in the
   left subtree or in the right subtree.
*/
lookup(K,bintree(K1,E,_,I1),I) :-
     K < K1, lookup(K,E,I).
lookup(K,bintree(K1,_,D,I1),I) :-
     K > K1, lookup(K,D,I).
```

Meta-interpreter in Prolog for Prolog

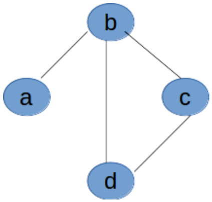
- A meta-interpreter is a program that executes other programs.
- Our example implements an interpreter in Prolog to executes Prolog programs.
- Relation: `interp`, number of arguments: 1, Prolog term to be interpreted (executed).

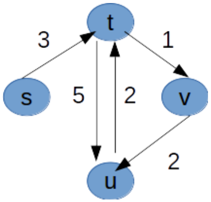
Meta-interpreter in Prolog for Prolog

```
/* true is always true. */
interp(true).
/* a conjunction is true if each one of the literals is true. */
interp((G1,G2)) :-
      interp(G1),
      interp(G2).
/* a disjunction is true if one of the literals is true. */
interp((G1;G2)) :-
      interp(G1);
      interp(G2).
/* the single goal G is true if
   it is defined in the program and
   its body is true.
*/
interp(G) :-
      clause(G,B),
      interp(B).
```

Search

- Representing graphs: set of nodes and edges/arcs. Arcs are generally represented as an ordered pair.
- Examples:





Search

- Representing graphs in Prolog:
  - Alternative 1:

    ```
    % non-directed graph          % directed graph
            connected(a,b).         arc(s,t,3).
            connected(b,c).         arc(t,v,1).
            connected(c,d).         arc(u,t,2).
               ...                     ...
    ```
  - Alternative 2: (sets of nodes and edges)

    ```
    graph([a,b,c,d],[e(a,b),e(b,d),e(b,c),e(c,d)])
    digraph([s,t,u,v],
    [a(s,t,3),a(t,v,1),a(t,u,5),a(u,t,2),a(v,u,2)])
    ```
  - Alternative 3: (adjacent list)

    ```
    [a->[b], b->[a,c,d], c->[b,d], d->[b,c]]
    [s->[t/3], t->[u/5,v/1], u->[t/2],v->[u/2]]
    ```
- Attention to the use of symbols `->` e `/`.

Search

- Typical operations in graphs:
  - find a path between two nodes in the graph,
  - find a subgraph with some given properties.
- Example: find a path between two nodes in a graph.

Search

- `G`: graph represented as in alternative 1.
- `A` and `Z`: two nodes in the graph.
- `P`: acyclic path between `A` and `Z`.
- `P` represents a list of path nodes.
- Relation **`path`**, number of arguments: 4, source node (`A`), destination node (`Z`), partial path (`L`) and complete path (`P`).
- a node can only appear once in path (cycles should be avoided).

Search

- Method to find an acyclic path `P`, between `A` and `Z`, in graph `G`.
- Find a partial path between `A` and some node `N`.
- Use the same method to find a path between `N` and `Z`. The final path goes from `A` to `Z` through `N`. On the way, skip already visited nodes.

Search

```
/* Find a path from A to Z and return path in P.
   Initial path is empty.
*/
path(A,Z,P) :- path1(A,Z,[],P).

/* If destination is reached, stop and return Path, which
   must include destnation Z */
path1(Z,Z,L,[Z|L]).

/* If destination not yet reached, find a partial path
   from A to Y. If Y has not been visited yet, add it to
   the partial path.
   Find a path from Y to Z.
*/
path1(A,Z,L,P) :-
      (conectado(A,Y);    % find partial path
       conectado(Y,A)),   % from A to Y or from Y to A
      \+ member(Y,L),     % check if Y has been visited
      path1(Y,Z,[Y|L],P). % find path from Y to Z
```