Logic Programming (PLP 11, CTM 9.2, 9.4, 12.1-12.2)

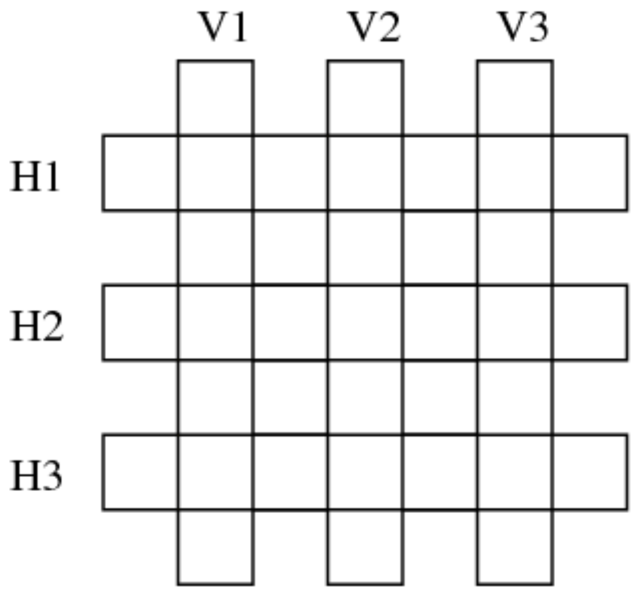
Constraint Satisfaction Problems, Natural Language Parsing (Definite Clause Grammars)

Carlos Varela
Rensselaer Polytechnic Institute

November 19, 2019

Constraint Satisfaction Example*

- Given six Italian words:
  - astante, astoria, baratto, cobalto, pistola, statale.
- They are to be arranged, crossword puzzle fashion, in the following grid:



Constraint Satisfaction Example(2)*

- The following knowledge base represents a lexicon containing these words:

```
word(astante,  a,s,t,a,n,t,e).
word(astoria,  a,s,t,o,r,i,a).
word(baratto,  b,a,r,a,t,t,o).
word(cobalto,  c,o,b,a,l,t,o).
word(pistola,  p,i,s,t,o,l,a).
word(statale,  s,t,a,t,a,l,e).
```

- Write a predicate `crossword/6` that tells us how to fill in the puzzle. The first three arguments should be the vertical words from left to right, and the last three arguments the horizontal words from top to bottom.

Constraint Satisfaction Example(3)*

- Try solving it yourself before looking at this solution!

```
crossword(V1,V2,V3,H1,H2,H3) :-
        word(V1,_,H1V1,_,H2V1,_,H3V1,_),
        word(V2,_,H1V2,_,H2V2,_,H3V2,_),
        word(V3,_,H1V3,_,H2V3,_,H3V3,_),
        word(H1,_,H1V1,_,H1V2,_,H1V3,_),
        word(H2,_,H2V1,_,H2V2,_,H2V3,_),
        word(H3,_,H3V1,_,H3V2,_,H3V3,_).
```

Constraint Satisfaction Example(Oz)

- The following relation represents the lexicon:

```
fun {Word}
   choice astante#a#s#t#a#n#t#e
   []     astoria#a#s#t#o#r#i#a
   []     baratto#b#a#r#a#t#t#o
   []     cobalto#c#o#b#a#l#t#o
   []     pistola#p#i#s#t#o#l#a
   []     statale#s#t#a#t#a#l#e
   end
end
```

- Write a predicate `Crossword/1` that tells us how to fill in the puzzle.

Constraint Satisfaction Example(Oz)

```
proc {Crossword S}
   H1V1 H2V1 H3V1 V1 H1
   H1V2 H2V2 H3V2 V2 H2
   H1V3 H2V3 H3V3 V3 H3
in
   S = [V1 V2 V3 H1 H2 H3]
   {Word V1#_#H1V1#_#H2V1#_#H3V1#_}
   {Word V2#_#H1V2#_#H2V2#_#H3V2#_}
   {Word V3#_#H1V3#_#H2V3#_#H3V3#_}
   {Word H1#_#H1V1#_#H1V2#_#H1V3#_}
   {Word H2#_#H2V1#_#H2V2#_#H2V3#_}
   {Word H3#_#H3V1#_#H3V2#_#H3V3#_}
end
```

Constraint Satisfaction Example: One Solution at a Time (Oz)

Crossword is a *relation* that corresponds to a query, represented as a one-argument procedure (or equivalent function).

Oz's Search module can produce a *lazy* list of solutions:

- especially useful when there are infinite answers, or when computation of all answers would take too long.

Solutions can be accessed via a *search engine object*:

```
% search engine
E = {New Search.object script(Crossword)}

% calculate and display one at a time
{Browse {E next($)}}
```

Constraint Satisfaction Example: One Solution or All Solutions (Oz)

The Crossword query relation can **also** be used directly by the Search module:

```
% Finding one solution
{Browse {Search.base.one Crossword}}

% Finding all solutions
{Browse {Search.base.all Crossword}}
```

Generate and Test Example

- We can use the relational computation model to generate all digits:

```
fun {Digit}
   choice 0 [] 1 [] 2 [] 3 [] 4 [] 5 [] 6 [] 7 [] 8 [] 9 end
end
{Browse {Search.base.all Digit}}
% displays [0 1 2 3 4 5 6 7 8 9]
```

Finding digit pairs that add to 10

- Using generate and test to do combinatorial search:

```
fun {PairAdd10}
   D1 D2 in
   D1 = {Digit}          % generate
   D2 = {Digit}          % generate
   D1+D2 = 10            % test
   D1#D2
end
{Browse {Search.base.all PairAdd10}}
% displays [1#9 2#8 3#7 4#6 5#5 6#4 7#3 8#2 9#1]
```

Finding digit pairs that add to 10 (Prolog)

- Using generate and test to do combinatorial search:

```
digit(D) :- between(0,9,D).

pairAdd10(D1,D2) :-
     digit(D1),
     digit(D2),
     D1 + D2 =:= 10.

allPairs(L) :-
    findall(p(D1,D2),pairAdd10(D1,D2),L).
```

Finding palindromes

- Find all four-digit palindromes that are products of two-digit numbers:

```
fun {Palindrome}
   X in
   X = (10*{Digit}+{Digit})*(10*{Digit}+{Digit})     % generate
   (X>=1000) = true                                  % test
   (X div 1000) mod 10 = (X div 1) mod 10            % test
   (X div 100) mod 10 = (X div 10) mod 10            % test
   X
end
{Browse {Search.base.all Palindrome}}                % 118 solutions
```

Finding palindromes (Prolog)

- Find all four-digit palindromes that are products of two-digit numbers:

```
palindrome(S) :-
        digit(D1), digit(D2), digit(D3), digit(D4),    % generate
        S is (10*D1+D2)*(10*D3+D4),
        S >= 1000,                                      % test
        mod(div(S,1000),10) =:= mod(S,10),              % 1st = 4th
        mod(div(S,100),10) =:= mod(div(S,10),10). % 2nd = 3rd

allPalindromes(S,L) :- findall(P,palindrome(P),S),length(S,L).
```

Propagate and Search

- The *generate and test* programming pattern can be very inefficient (e.g., Palindrome program explores 10000 possibilities).

- An alternative is to use a *propagate and search* technique.

Propagate and search filters possibilities during the generation process, to prevent combinatorial explosion when possible.

Propagate and Search

Propagate and search approach is based on three key ideas:

- *Keep partial information*, e.g., "in any solution, X is greater than 100".
- *Use local deduction*, e.g., combining "X is less than Y" and "X is greater than 100", we can deduce "Y is greater than 101" (assuming Y is an integer.)
- *Do controlled search*. When no more deductions can be done, then search. Divide original CSP problem P into two new problems: $(P \wedge C)$ and $(P \wedge \neg C)$ and where C is a new constraint. The solution to P is the union of the two new sub-problems. Choice of C can significantly affect search space.

Propagate and Search Example

- Find two digits that add to 10, multiply to more than 24:

```
D1::0#9         D2::0#9                 % initial constraints
{Browse D1}     {Browse D2}             % partial results
D1+D2 =: 10  % reduces search space from 100 to 81 possibilities
             % D1 and D2 cannot be 0.
D1*D2 >=: 24   % reduces search space to 9 possibilities
               % D1 and D2 must be between 4 and 6.
D1 <: D2       % reduces search space to 4 possibilities
               % D1 must be 4 or 5 and D2 must be 5 or 6.
               % It does not find unique solution D1=4 and D2=6
```

Propagate and Search Example(2)

- Find a rectangle whose perimeter is 20, whose area is greater than or equal to 24, and width less than height:

```
fun {Rectangle}
  W H in W::0#9   H::0#9
  W+H =: 10
  W*H >=: 24
  W <: H
  {FD.distribute naive rect(W H)}
  rect(W H)
end
{Browse {Search.base.all Rectangle}}
% displays [rect(4 6)]
```

Propagate and Search Example (Prolog)

- Find two digits that add to 10, multiply to more than 24:

```
:- use_module(library(clpfd)).
q(D1,D2) :-
        D1 in 0..9, D2 in 0..9,     % initial constraints
        D1+D2 #= 10,                % D1 and D2 cannot be 0.
        D1*D2 #>= 24,  % D1 and D2 must be between 4 and 6.
        D1 #< D2.                   % D1 must be 4 or 5 and
                                    % D2 must be 5 or 6.
             % It does not find unique solution D1=4 and D2=6.
```

Propagate and Search Example(2)

- Find a rectangle whose perimeter is 20, whose area is greater than or equal to 24, and width less than height:

```
rectangle([W,H]) :-
            W in 0..9, H in 0..9,
            W+H #= 10,
            W*H #>= 24,
            W #< H.
rectangleSolve(rect(W,H)) :-
            rectangle([W,H]),
            label([W,H]).
?- rectangleSolve(S).
S = rect(4, 6).
```

Finding palindromes (revisited)

- Find all four-digit palindromes that are products of two-digit numbers:

```
fun {Palindrome}
  A B C X Y in
  A::1000#9999   B::0#99 C::0#99
  A =: B*C
  X::1#9   Y::0#9
  A =: X*1000+Y*100+Y*10+X
  {FD.distribute ff [X Y]}
  A
end
{Browse {Search.base.all Palindrome}}              % 36 solutions
```

Finding palindromes (revisited in Prolog)

- Find all four-digit palindromes that are products of two-digit numbers:

```
palindrome(A,B,C,X,Y) :-
        A in 1000..9999, B in 0..99, C in 0..99,
        A #= B * C,
        X in 1..9, Y in 0..9,
        A #= X*1000+Y*100+Y*10+X.

palindromeSolve(A) :-
        palindrome(A,_,_,X,Y),
        labeling([ff], [X,Y]).
```

Natural Language Parsing

(Example from "Learn Prolog Now!" Online Tutorial)

```
word(article,a).
word(article,every).
word(noun,criminal).
word(noun,'big kahuna burger').
word(verb,eats).
word(verb,likes).

sentence(Word1,Word2,Word3,Word4,Word5) :-
      word(article,Word1),
      word(noun,Word2),
      word(verb,Word3),
      word(article,Word4),
      word(noun,Word5).
```

Natural Language Parsing (Oz)

(Example from "Learn Prolog Now!" Online Tutorial)

```
fun {Word}
   choice
      article#a
   [] article#every
   [] noun#criminal
   [] noun#'big kahuna burger'
   [] verb#eats
   [] verb#likes
   end
end
```

```
proc {Sentence S}
   Word1 Word2 Word3 Word4 Word5
in
   S = [Word1 Word2 Word3 Word4 Word5]
   {Word article#Word1}
   {Word noun#Word2}
   {Word verb#Word3}
   {Word article#Word4}
   {Word noun#Word5}
end
```

Parsing natural language

- *Definite Clause Grammars (DCG)* are useful for natural language parsing.
- Prolog can load DCG rules and convert them automatically to Prolog parsing rules.

DCG Syntax

`-->`

DCG *operator*, e.g.,

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

Each goal is assumed to refer to the *head* of a DCG rule.

`{prolog_code}`

Include Prolog code in generated parser, e.g.,

```
subject-->modifier,noun,{write('subject')}.
```

`[terminal_symbol]`

Terminal symbols of the grammar, e.g.,

```
noun-->[cat].
```

Natural Language Parsing

(example rewritten using DCG)

```
sentence --> article, noun, verb, article, noun.

article --> [a] | [every].

noun --> [criminal] | ['big kahuna burger'].

verb --> [eats] | [likes].
```

Natural Language Parsing (2)

(example rewritten using DCG)

Let us look at Prolog's generated Horn clause for the `sentence` non-terminal:

```
?- listing(sentence).
sentence(A, F) :-
      article(A, B),
      noun(B, C),
      verb(C, D),
      article(D, E),
      noun(E, F).
```

A-F is a *difference list*. B, C, D, and E are *accumulators*. Possible usage:

```
?- sentence([a,criminal,likes,every,'big kahuna burger'],[]).
true
```

Natural Language Parsing (3)

(example rewritten using DCG)

Now, let us look at Prolog’s generated Horn clause for the `verb` non-terminal:

```
?- listing(verb).
verb(A, B) :-
        (   A=[eats|B]
        ;   A=[likes|B]
        ).
```

A-B is a *difference list*. Possible usage:

```
?- verb([likes],[]).
true.

?- verb([likes,cats],[cats]).
true.
```

Natural Language Parsing in Oz

Let us look at an Oz relation for the `sentence` non-terminal:

```
proc {Sentence S Sn}
   S1 S2 S3 S4
in
   {Article S S1}
   {Noun S1 S2}
   {Verb S2 S3}
   {Article S3 S4}
   {Noun S4 Sn}
end
```

S-Sn is a *difference list*. S1, S2, S3, and S4 are *accumulators*. Possible usage:

```
proc {Query S} {Sentence S nil} end
{Browse {Search.base.all Query}}
```

Natural Language Parsing in Oz

Now, let us look at Oz relation for the `verb` non-terminal:

```
proc {Verb S Sn}
   choice
      S = eats|Sn
   [] S = likes|Sn
   end
end
```

S-Sn is a *difference list*. Possible usage:

```
{Browse {Search.base.all
          proc {$ V}
             {Verb V nil}
          end}}
```

Natural Language Parsing and Information Extraction

```
sentence(V) --> subject, verb(V), subject.
sentence(V) --> subject, verb(V).

subject --> article, noun.

article --> [a] | [every].

noun --> [criminal]
         | ['big kahuna burger']
         | [dog].

verb(eats) --> [eats].
verb(likes) --> [likes].
```

Natural Language Parsing and Information Extraction

Prolog's generated Horn clauses for the `sentence` non-terminal:

```
?- listing(sentence).
sentence(B, A, E) :-
      subject(A, C),
      verb(B, C, D),
      subject(D, E).
sentence(B, A, D) :-
      subject(A, C),
      verb(B, C, D).
```

A-E and A-D are *difference lists*. B is the extracted information (which could be a parse tree). C and D are *accumulators*. Possible usage:

```
?- sentence(Verb, [a,dog,eats],[]).
Verb = eats.
```

Natural Language Parsing and Information Extraction

Now, let us look at Prolog's generated Horn clause for the `verb` non-terminal:

```
?- listing(verb).
verb(eats, [eats|A], A).
verb(likes, [likes|A], A).
```

Possible usage:

```
?-  verb(Verb, [eats], []).
Verb = eats.

?- verb(Verb,S,T).
Verb = eats,
S = [eats|T] ;
Verb = likes,
S = [likes|T].
```

Natural Language Parsing and Information Extraction in Oz

Let us look at an Oz relation for the `sentence` non-terminal:

```
proc {Sentence V S Sn}
   S1 S2
in
  choice
    {Subject S S1}
    {Verb V S1 S2}
    {Subject S2 Sn}
  []
    {Subject S S1}
    {Verb V S1 Sn}
  end
end
```

S-Sn is a *difference list*. S1, and S2 are *accumulators*.

Possible usage:

```
fun {Query}
   V S
in
   {Sentence V S nil}
   V#S
end
{Browse {Search.base.all Query}}
```

Natural Language Parsing and Information Extraction in Oz

Now, let us look at Oz relation for the `verb` non-terminal:

```
proc {Verb V S Sn}
   choice
      V = eats
      S = eats|Sn
   [] V = likes
      S = likes|Sn
   end
end
```

S-Sn is a *difference list*. V is the extracted verb. Possible usage:

```
{Browse {Search.base.all
           proc {$ V}
              {Verb V _ _}
           end}}
```

Exercises

83. How would you translate DCG rules into Prolog/Oz rules?
84. PLP Exercise 11.8 (pg 571).
85. PLP Exercise 11.14 (pg 572).
86. CTM Exercise 12.6.2 (pg 774).