Chapter 3

Definite Clause Grammars for NL

In this chapter we look at a couple of simple examples of solving some NL problems using DCGs. The first shows how noun-verb agreement can be achieved using the variables which can appear in non-terminals in DCGs. Only the present tense will be illustrated. We then consider the problem of translating from one language to another. Here a sentence in French is parsed/recognised and at the same time the English translation is constructed in one of the variables.

We begin with a general introduction to the syntax and translation of DCG in Prolog. Most versions of Prolog have the capability of defining languages and operating on them by means of *definite clause grammars*.

These types of grammars are similar to *cfg*s but are strictly more powerful as they do have some context sensitive features.

3.1 Simplest Form

The most basic form of *dcg*s are essentially the same as *cfg*s. We give an example to illustrate this. Once the rules are presented to the Prolog interpreter they are translated into pure Prolog clauses. If a listing is performed then the translations can be seen.

3.1.1 Example

Here we have a grammar to recognise various forms of numbers. Notice how close it is to the normal *cfg* definition.

```
digit --> [0]|[1]|[2]|[3]|[4]|
          [5]|[6]|[7]|[8]|[9]. % a digit is in 0-9

nat_num --> digit.             % a natural number is a
nat_num --> digit,nat_num.     % sequence of digits

int -->     nat_num.           % an integer is a natural number
```

```
int -->     sign,nat_num.       % possibly with a sign

real -->    int.                % a real is given in normal
real -->    int,['.'],nat_num.  % decimal notation
real -->    sign,['.'],nat_num.

sign -->    [-]|[+].
```

The objects inside []'s are *terminal* elements of the language, the other identifiers play the role of non-terminals.

We can use '`|`' to separate alternatives on the rhs of a rule or give them as separate rules.

The above is translated into the following Prolog clauses

```
nat_num(A, B) :-
                digit(A, B).

nat_num(A, B) :- digit(A, C),
                 nat_num(C, B).
sign(A, B) :-
                 ( 'C'(A, -, B) |
                   'C'(A, +, B) ).

real(A, B) :-
                 int(A, B).
real(A, B) :-
                 int(A, C),
                 'C'(C, '.', D),
                 nat_num(D, B).
real(A, B) :-
                 sign(A, C),
                 'C'(C, '.', D),
                 nat_num(D, B).
int(A, B) :-
                 nat_num(A, B).
int(A, B) :-
                 sign(A, C),
                 nat_num(C, B).
digit(A, B) :-
                 ( 'C'(A, 0, B) |
                   'C'(A, 1, B) |
                   'C'(A, 2, B) |
                   'C'(A, 3, B) |
```

```
        'C'(A, 4, B) |
        'C'(A, 5, B) |
        'C'(A, 6, B) |
        'C'(A, 7, B) |
        'C'(A, 8, B) |
        'C'(A, 9, B) ).
```

If we take the clause `nat_num(A,B)`, then the interpretation of this predicate is that the sequence of elements in `B` is an end part of `A`, and that the sub-sequence of `A` obtained by cutting off `B` satisfies the definition of natural number. In other words we are essentially using the pair `(A,B)` as a difference list.

So for example `nat_num([1,2,3],[])` is true.
More generally so too is `nat_num([1,2,3|X],X)` for any `X`. The special built in predicate `'C'` is used for recognising the terminals and `'C'(X,Terminal,Y)` means that `Terminal` is the head of `X` and it's tail is `Y` — as if it were defined as `'C'([X|Xs],X,Xs)`.

To make the above easier to use we could write a procedure which would convert a number in normal representation into a sequence of the separate symbols i.e. `'+123'` would be converted to the list `[+,1,2,3]` and `'1.23'` would be converted to `[1,'.',2,3]` etc. Note it is essential to quote the `'.'` otherwise Prolog will take it as a terminator for a Prolog clause and so give an error. Also if the number contains a sign it must be quoted as `name` requires an atom and `+123` for example is not an atom (as begins with `'+'`).

```
convert_to_list(Atom,Lst):-
                            name(Atom,AtomLst),
                            % convert to list of ascii numbers
                            list_to_vals(AtomLst,Lst).

ascii_to_char(Asc,Chr) :-
                            name(Chr,[Asc]).

list_to_vals([],[]).        % apply down a list

list_to_vals([H|T],[HV|TV]) :-
                            ascii_to_val(H,HV),
                            list_to_vals(T,TV).
```

Instead of using the translated predicates to recognise elements of a language we can also use the built in predicate `phrase`. `phrase(NT,Lst)` means `Lst` is generated by the non-terminal `NT`. e. g. `phrase(real,[1,2,'.',4,5])` is true.

Now the above grammar allows numbers with leading zeroes such as `003`. It is not too difficult to alter the grammar by introducing new non-terminals to reject such numbers. The changes are as follows:

```
nonz_digit --> [1]|[2]|[3]|[4]|[5]|
               [6]|[7]|[8]|[9].  % a nonz_digit is in 1-9

digit -->      [0]|nonz_digit.

red_nat_num -->digit|nonz_digit,nat_num.

red_int -->    red_nat_num|sign,red_nat_num.

red_real -->   red_int | red_int,['.'],nat_num.
```

This will then return false for `phrase(red_nat_num,[0,0,2,3])` but true for `phrase(nat_num,[0,0,1,2`

3.2 More Complex Examples

*Dcg*s also have the facility of embedding Prolog code in their bodies. This is achieved by placing the required code in {}. Anything inside {} is then left unaltered by the interpreter. Parameters can also appear as arguments to the non-terminal symbols so that results can be returned as a side effect to the language recognition or used in further calculations.

3.2.1 Example

We add a parameter to the previous example so that after recognising a correct number it will contain the value of that number. We also need to carry how many digits appear in the number.

e. g. `phrase(nat_num(N),[1,2,3,1])` will instantiate the variable `N` to the value `1231`
We shall not bother with the reduced form but take the original grammar.

```
digit(0) --> [0].  % a digit is in 0-9
digit(1) --> [1].
digit(2) --> [2].
digit(3) --> [3].  % etc
         ⋮
digit(9) --> [9].

nat_num(N,1) --> digit(N). % a natural number is a
                           %sequence of digits

nat_num(N,ND) --> digit(D),nat_num(N2,ND1),
                  % ND is the number of digits
                  {plus(ND1,1,ND),
                  power_ten(D,ND1,P),
                  plus(P,N2,N)}.
```

```
int(N,D) --> nat_num(N,D). % an integer is a natural number
                           % possibly with a sign
int(N,D) --> [+],nat_num(N,D).
int(N,D) --> [-],nat_num(N1,D),
             { N is - N1}.

real(R) --> int(R,_).  % a real is given in normal decimal notation
real(R) --> int(I,_),['.'],nat_num(N,ND),
            {neg_power_ten(ND,InvP),
            ( I >= 0 -> R is I + N * InvP;
                        R is I - N * InvP) }.

real(R) --> [+],['.'],nat_num(N,D),
            { neg_power_ten(D,InvP),
            R is N * InvP }.

real(R) --> [-],['.'],nat_num(N,D),
            { neg_power_ten(D,InvP),
            R is - N * InvP }.

power_ten(D,0,D) :-!.
power_ten(D,E1,P) :-
                    E1 >0,
                    plus(E,1,E1),
                    power_ten(D,E,P1),
                    P is P1 * 10,!.

neg_power_ten(0,1):-!.
neg_power_ten(N,P):-
                    N > 0,
                    succ(N1,N),
                    neg_power_ten(N1,P1),
                    P is P1/10 .
```

A call of for example `real(R,[-,2,3,'.',4,5],[])` returns true and instantiates `R` to the real number `-23.45`.

A call of `phrase(int(N,D),[-,1,2,3])` instantiates `N` to the number `-123` and `D`, the number of digits, to `3`.

3.2.2 Example

```
% parse simple english sentence
% for time being just present as a list of identifiers
% e.g.  [the,big,cat,kicks,the,black,dog]
% first we have a straight forward generator
% will not repeat adjective like big big girl!
% Also check whether we need an 'an' or an 'a'.

adjnounph(CV) --> noun(CV).
adjnounph(CV) --> adjective(CV,Adj),noun(_).
adjnounph(CV) --> adjective(CV,Adj),adjective(_,Adj2),
                  {Adj \== Adj2},
                  noun(_) .

nounphrase --> det(CV),adjnounph(CV).
sentence   --> nounphrase,verb,nounphrase.

% now some explicit examples
det(cons)  --> [the]|[a].
det(vowel) --> [an].

verb       --> [hit]|[kicks]|[kisses].

noun(cons) --> [cat]|[boy]|[girl].
noun(vowel)--> [owl]|[ox].

adjective(cons,big)       --> [big].
adjective(cons,black)     --> [black].
adjective(cons,brown)     --> [brown].
adjective(cons,tabby)     --> [tabby].
adjective(vowel,awful)    --> [awful].
adjective(vowel,awesome)  --> [awesome].
```

The above will now correctly recognise `an awful owl hit an ox` and reject `a awesome cat kisses a awful girl`!