A DCG (definite clause grammar) is a phrase structure grammar annotated by Prolog variables.

DCGs are translated by the Prolog interpreter into normal Prolog clauses.

Prolog DCG:s can be used for generation as well as parsing. I.e. we can run the program backwards to generate sentences from the grammar.

```
s --> np, vp.
v --> v, np.
v --> np --> n.
n --> [john]. n --> [lisa].
n --> [house].
v --> [died]. v --> [kissed].

?- s([john, kissed, lisa], []).  yes
?- s([lisa, died], []).  yes
?- s([kissed, john, lisa], []).  no

?- s(A, []).
A = [john,died, john] ;
A = [john,died, lisa] ;
A = [john,died, house] ;
A = [john,kissed, john] ;
A = [john,kissed, lisa] ;
A = [john,kissed, house] ;
A = [john,died] ;
A = [john,kissed] ;
A = [lisa,died, john] ;
A = [lisa,died, lisa] ;
A = [lisa,died, house] ;
A = [lisa,kissed, house] ;
A = [lisa,died] ;
```
Prolog uses **difference lists** instead.

The rule \( S \rightarrow NP \ VP \) becomes:

\[
\text{s}(Z) :- \text{np}(X), \ \text{vp}(Y), \ \text{append}(X,Y,Z).
\]

This states that \( Z \) is a sentence if \( X \) is a noun phrase, \( Y \) is a verb phrase, and \( Z \) is \( X \) followed by \( Y \).

The `append`'s are expensive.

Prolog uses difference lists instead.

\[
\text{s}(A,B) :- \text{np}(A,C), \ \text{vp}(C,B).
\]

says that there is a sentence at the beginning of \( A \) (with \( B \) left over) if there is a noun phrase at the beginning of \( A \) (with \( C \) left over), and there is a verb phrase at the beginning of \( C \) (with \( B \) left over).

\[
\text{np}(Z) :- \text{n}(Z).
\]

\[
\text{vp}(Z) :- \text{v}(X), \ \text{np}(Y), \ \text{append}(X,Y,Z).
\]

\[
\text{vp}(Z) :- \text{v}(Z).
\]

\[
\text{n}([\text{john}|R],R). \ \text{n}([\text{lisa}|R],R).
\]

\[
\text{v}([\text{died}|R],R). \ \text{v}([\text{kissed}|R],R).
\]

?- \text{s}([\text{john},\text{kissed},\text{lisa}], \text{[]}).

    yes

?- \text{s}([\text{john},\text{kissed}|R], \text{[]}).

    R = [\text{john}];
    R = [\text{lisa}] ;...
Generating Parse Trees...

- \( s(s(NP, VP)) \rightarrow np(NP), vp(VP) \) says that the top-level node of the parse tree is an \( s \) with the sub-trees generated by the \( np \) and \( vp \) rules.

\[- s(S, [john, kissed, lisa], []).\]
\( S=s(np(n(john)), vp(n(kissed), np(n(lisa)))) \)

\[- s(S, [lisa, died], []).\]
\( S=s(np(n(lisa)), vp(n(died))) \)

\[- s(S, [john, died, lisa], []).\]
\( S=s(np(n(john)), vp(n(died), np(n(lisa)))) \)

Generating Parse Trees...

- We can of course run the rules backwards, turning parse trees into sentences:

\[- s(s(np(n(john)), vp(n(kissed), np(n(lisa)))), S, []).\]
\( S=[john, kissed, lisa] \)

Generating Parse Trees

- DCGs can build parse trees which can be used to construct a semantic interpretation of the sentence.
- The tree is built bottom-up, when Prolog returns from recursive calls. We give each phrase structure rule an extra argument which represents the node to be constructed.

Generating Parse Trees...

- DCGs can build parse trees which can be used to construct a semantic interpretation of the sentence.

\[ s(s(NP, VP)) \rightarrow np(NP), vp(VP). \]
\[ vp(vp(V, NP)) \rightarrow v(V), np(NP). \]
\[ vp(vp(V)) \rightarrow v(V). \]
\[ np(np(N)) \rightarrow n(N). \]
\[ n(n(john)) \rightarrow [john]. \]
\[ n(n(lisa)) \rightarrow [lisa]. \]
\[ n(n(house)) \rightarrow [house]. \]
\[ v(n(died)) \rightarrow [died]. \]
\[ v(n(kissed)) \rightarrow [kissed]. \]
Ambiguity

- An ambiguous sentence is one which can have more than one meaning.

Lexical ambiguity: homographic
- spelled the same
  - *bat* (wooden stick/animal)
  - *import* (noun/verb)

polysemous
- different but related meanings
  - *neck* (part of body/part of bottle/narrow strip of land)

homophonic
- sound the same
  - to/too/two

Syntactic ambiguity:

- More than one parse (tree).
- Many missiles have many war-heads.

- “Duck” can be either a verb or a noun.
- “her” can either be a determiner (as in “her book”), or a noun: “I liked her dancing”.

```
s(NP,VP) --> np(NP), vp(VP).
vp(vp(V, NP)) --> v(V), np(NP).
vp(vp(V, S)) --> v(V), s(S).
vp(vp(V)) --> v(V).
np(np(Det,N)) --> det(Det), n(N).
np(np(N)) --> n(N).
n(n(i)) --> [i].
n(n(duck)) --> [duck].
v(v(duck)) --> [duck].
v(v(saw)) --> [saw]. n(n(saw)) --> [saw].
n(n(her)) --> [her].
det(det(her)) --> [her].
?- s(S, [i, saw, her, duck], []).```
Pascal Declarations...

% Type declarations
type_decl --> [ ].
type_decl --> [type], type_def, [;], type_defs.
type_defs --> [ ].
type_defs --> type_def, [;], type_defs.
type_def --> identifier, [=], type.
type --> ['INTEGER']. type --> ['REAL'].
type --> ['BOOLEAN']. type --> ['CHAR'].

Pascal Declarations...

% Constant declarations
const_decl --> [ ].
const_decl --> [const], const_def, [;], const_defs.
const_defs --> [ ].
const_defs --> const_def, [;], const_defs.
const_def --> identifier, [=], constant.
identifier --> [X], {atom(X)}.
constant --> [X], {(integer(X); float(X))}.

Pascal Declarations...

?- decl([const, a, =, 5, ;, var, x, :, 'INTEGER', ;], []). yes
?- decl([const, a, =, a, ;, var, x, :, 'INTEGER', ;], []). no
decl --> const_decl, type_decl, var_decl, proc_decl.
Pascal Declarations – Building Trees...

\[
\begin{align*}
\text{constdefs} & (\text{null}) \rightarrow [ ] . \\
\text{constdefs} & (\text{const}(D, Ds)) \rightarrow \\
& \quad \text{constdef}(D), [;], \text{constdefs}(Ds) . \\
\text{constdef} & (\text{id}(I, C)) \rightarrow \text{id}(I), [=], \text{const}(C) . \\
\text{id}(\text{id}(X)) & \rightarrow [X], \{\text{atom}(X)\} . \\
\text{constnum} & (\text{id}(X)) \rightarrow [X], \{(\text{integer}(X); \text{float}(X))\} .
\end{align*}
\]

Pascal Declarations – Example Parse

\[
\begin{align*}
\text{decl} & \rightarrow \text{decl}(C, T, V, P) . \\
\text{const#} & \text{decl}(C), \text{type#} \text{decl}(T), \\
& \text{var#} \text{decl}(V), \text{proc#} \text{declaration}(P) . \\
\text{const} & \text{decl}(\text{null}) \rightarrow [ ] . \\
\text{const} & \text{decl}(\text{const}(null)) \rightarrow [ ] . \\
\text{const} & \text{decl}(\text{const}(D, Ds)) \rightarrow \\
& [\text{const}], \text{const#} \text{def}(D), [;], \text{constdefs}(Ds) .
\end{align*}
\]
Pascal Declarations – Example Parse...

?- decl(S, [const, a, =, 5, ;, x, =, 3.14, ;], []).  
S = decl(  
    const(def(id(a),num(5)),  
    const(def(id(x),num(3.14)),  
    null)),  
null,null,null)

Number Conversion...

digit(1) --> [one].  
digit(2) --> [two].  
digit(3) --> [three].  
digit(4) --> [four].  
digit(5) --> [five].  
digit(6) --> [six].  
digit(7) --> [seven].  
digit(8) --> [eight].  
digit(9) --> [nine].  
tens(20) --> [twenty].  
tens(30) --> [thirty].  
tens(40) --> [forty].  
tens(50) --> [fifty].  
tens(60) --> [sixty].  
tens(70) --> [seventy].  
tens(80) --> [eighty].  
tens(90) --> [ninety].

Number Conversion

?- number(V, [sixty, three], []).  
V = 63  
?- number(V,[one,hundred, and, fourteen],[]).  
V = 114  
?- number(V,[nine, hundred, and, ninety,nine],[]).  
V = 999  
?- number(V, [fifty, ten], []).  
no
**Expression Evaluation**

- Evaluate infix arithmetic expressions, given as character strings.

```
?- expr(X, "234+345*456", []).
X = 157554
```

```
expr(Z) --> term(X), "+", expr(Y), {Z is X + Y}.
expr(Z) --> term(X), "-", expr(Y), {Z is X - Y}.
expr(Z) --> term(Z).
term(Z) --> num(X), "*", term(Y), {Z is X * Y}.
term(Z) --> num(X), "/", term(Y), {Z is X / Y}.
term(Z) --> num(Z).
```

**Summary**

- Read Bratko:
  - Grammar Rules 432–439
  - Constructing Parse Trees 440–444

- Grammar rule syntax:
  - A grammar rule is written `LHS --> RHS`. The left-hand side (LSH) must be a non-terminal symbol, the right-hand side (RHS) can be a combination of terminals, non-terminals, and Prolog goals.
  - Terminal symbols (words) are in square brackets: `n --> [house]`.
  - More than one terminal can be matched by one rule: `np --> [the,house]`.

**Summary...**

- Grammar rule syntax (cont):
  - Non-terminals (syntactic categories) can be given extra arguments: `s(s(N,V)) --> np(N),vp(V)`.
  - Normal Prolog goals can be embedded within grammar rules: `int(C) --> [C],{integer(C)}`.
  - Terminals, non-terminals, and Prolog goals can be mixed in the right-hand side: `x --> [y], z, {w}, [r], p`.

- Beware of left recursion! `expr --> expr ["+"|] expr` will recurse infinitely. Rules like this will have to be rewritten to use right recursion.

**Expression Evaluation...**

- Prolog grammar rules are equivalent to recursive descent parsing. Beware of left recursion!
- Anything within curly brackets is “normal” Prolog code.

```
num(C) --> "+", num(C).
num(C) --> "-", num(X), {C is -X}.
num(X) --> int(0, X).
```

```
int(L, V) --> digit(C), {V is L * 10 +C}.
int(L, X) --> digit(C), {V is L* 10 +C}, int(V, X).
digit(X) --> [C], {"0" =< C, C =< "9", X is C="0"}.
```
Homework I (A)

- Write a program which uses Prolog Grammar Rules to convert between English time expressions and a 24-hour clock (“Military Time”).

- You may assume that the following definitions are available:

  digit(1) --> [one]. ....
  digit(9) --> [nine].
  teen(10) --> [ten]. ....
  teen(19) --> [nineteen].
  tens(20) --> [twenty]. ....
  tens(90) --> [ninety].

?- time(T, [eight, am], []).  
  T = 8:0 % Or, better, 8:00
?- time(T, [eight, thirty, am], []).  
  T = 8:30
?- time(T, [eight, fifteen, am], []).  
  T = 8:15
?- time(T, [eight, five, am], []).  
  T = 8:15
?- time(T, [eight, oh, five, am], []).  
  no
?- time(T, [eight, oh, eleven, am], []).  
  no
?- time(T, [eleven, thirty, pm], []).  
  T = 23:30
?- time(T, [twelve, thirty, pm], []).  
  T = 12:30 % !!!
?- time(T, [ten, minutes, to, four, am], []).  
  T = 3:50
?- time(T, [ten, minutes, past, four, am], []).  
  T = 4:10
?- time(T, [quarter, to, four, pm], []).  
  T = 15:45
?- time(T, [quarter, past, four, pm], []).  
  T = 16:15
?- time(T, [half, past, four, pm], []).  
  T = 16:30

?- time(T, [eleven, thirty, pm], []).  
  T = 23:30
?- time(T, [twelve, thirty, pm], []).  
  T = 12:30 % !!!
?- time(T, [ten, minutes, to, four, am], []).  
  T = 3:50
?- time(T, [ten, minutes, past, four, am], []).  
  T = 4:10
?- time(T, [eleven, thirty, pm], []).  
  T = 23:30
?- time(T, [twelve, thirty, pm], []).  
  T = 12:30 % !!!