# 1 - Introduction To Prolog

Prolog is a declarative programming language, which requires a different mindset to that of procedural languages such as C. Prolog Knowledge bases consist of two types of statements, facts and rules. Facts look like:

```
happy(yolanda).
man(heMan).
```

They are statements, that when queried by a prolog interpreter would return true. Rules take the form:

```
listensToMusic(yolanda) :- happy(yolanda).
hasThePower(heMan) :- man(heMan).
```

The first rule above reads "Yolanda listens to music <u>if</u> yolanda is happy". Facts and Rules are both called clauses. Rules are also known as <u>predicates</u>.

*The comma expression ',' expresses conjunction (Logical AND) in prolog.*
*The semicolon expression ';' expresses disjunction (Logical OR) in prolog.*

<u>Variables in prolog begin with an uppercase letter, such as Woman.</u>

If a fact (or Atom) is called using a variable, the variable is unified with a valid answer from the knowledge base. For example man(X) would return X=heMan when using the knowledge base above.

<u>**Atoms**</u> are a series of characters beginning with a lowercase letter, an arbitrary sequence of characters enclosed in single quotes, or a sequence of special characters (':', ',', ';', '.', ':-'). They are like statements.

<u>**Variables**</u> are a sequence of characters of upper case letters, lower case letters, digits, or underscore starting with either an uppercase letter or an underscore.

<u>**Complex Terms**</u> are another term for predicates, and are built out of a functor and arguments. The functor must be an atom, and arguments are put in brackets separated by commas.

<u>**Arity**</u> is the number of arguments a complex term has.

# 2 - Unity

<u>*Two terms unify if they are the same term or if they contain variables that can be uniformly instantiated with terms in such a way that the resulting terms are equal.*</u>

Following on from this definition, it means that:

- mia and mia unify
- 42 and 42 unify
- woman(mia) and woman(mia) unify

Conversely

- vincent and mia do not unify
- woman(mia) and woman(jody) do not unify

But when trying to unify a variable with a constant, we can modify the definition as follows:

*If T1 is a variable and T2 is any type of term, then T1 and T2 unify, and T1 is instantiated to T2.*

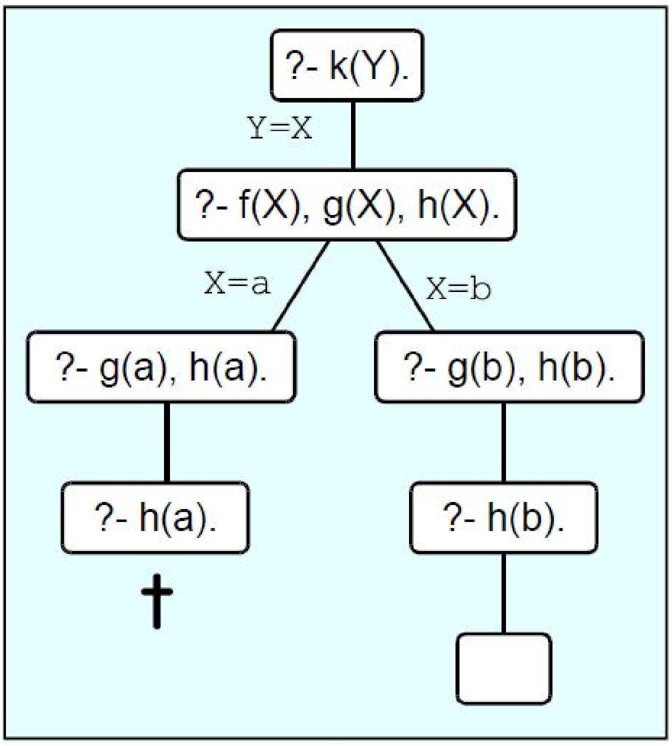
And when dealing with complex terms:

*If T1 and T2 are complex terms then they unify if they have the same functor and arity, all their corresponding arguments unify, and the variable instantiations are compatible.*

**Proof Search** in prolog refers to how prolog goes about evaluating queries to see if they are satisfied. This can be represented using a search tree, as seen below:

```
f(a).
f(b).
g(a).
g(b).
h(b).
k(X):- f(X), g(X), h(X).
```

```
?- k(Y).
Y=b
```



## 3 - Recursion

Prolog predicates can be defined recursively, meaning one or more rules in its definition refers to itself. An example of a recursive function definition is as follows:

```
child(anna,bridget).
child(bridget,caroline).
child(caroline,donna).
child(donna,emily).


descend(X,Y):- child(X,Y).
descend(X,Y):- child(X,Z), descend(Z,Y).
```

```
?- descend(anna,donna).
```

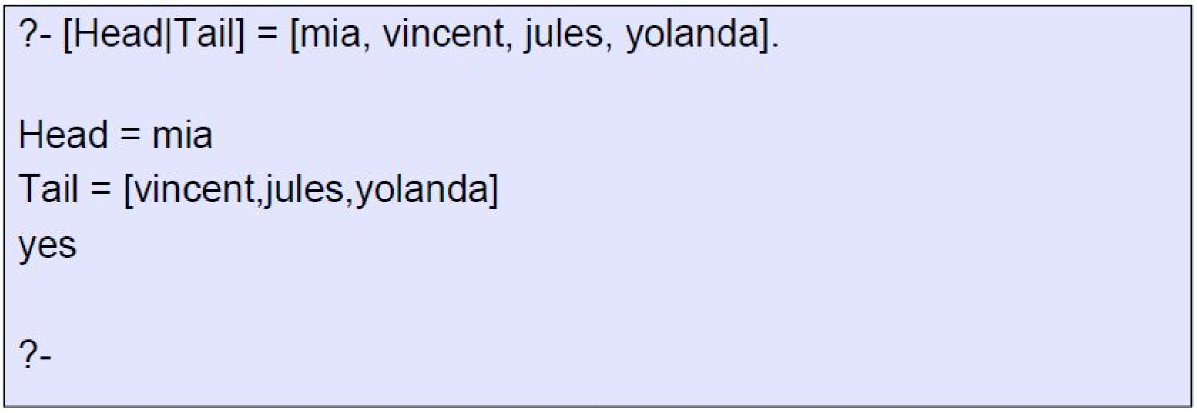
## 4 - Lists

A list is a finite sequence of elements, structured as follows in prolog:

[mia, vincent, jules, yolanda]
[mia, robber(honeybunny), X, 2, mia]
[]
[mia, [vincent, jules], [butch, friend(butch)]
[[], dead(z), [2,[b,c]], [], Z, [b,c]]]

A non empty list can be thought of as consisting of two parts, the head and tail. The head is the first item in the list, and the tail is everything else. <u>The tail of a list is always a list.</u>

The built in '|' operator in prolog is a key tool for writing prolog list manipulation predicates, and is used to split up lists as follows:



```
?- [Head|Tail] = [mia, vincent, jules, yolanda].

Head = mia
Tail = [vincent,jules,yolanda]
yes

?-
```

The [Head|Tail] syntax can also be used when defining parameters of functors of complex terms. We can access more than just the first element of the list by typing [H1, H2, H3|Tail] to bind the first three elements of the list to H1, H2, and H3 respectively. If we were only interested in the third variable, we could use the anonymous variable _ to replace H1 and H2. This doesn't unify the first and second variables with anything.

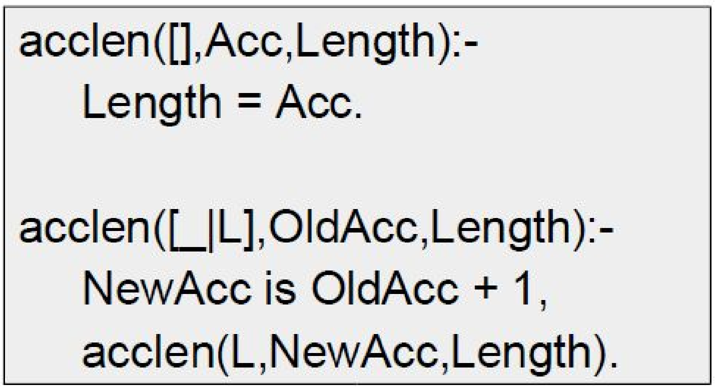
## 5 - Arithmetic

Prolog provides a number of basic arithmetic tools. They can be used as follows:

| | |
|---|---|
| 2 + 3 = X | X is 2+3. |
| 3 x 4 = X | X is 3*4. |
| 5 - 3 = X | X is 5-3. |
| 3 - 5 = X | X is 3 - 5. (-3 -> Valid) |
| 4 / 2 = X | X is 4/2. |
| 7 % 2 = X | X is mod(7,2). |

Must be careful with syntax, as X = 3+2 unifies X with 3+2. To force evaluation, you must use the is predicate. Is is an instruction for prolog to carry out calculations.

**Accumulators** are intermediate values used to store some value during recursion. Usage of it can be seen below when determining the length of a list:

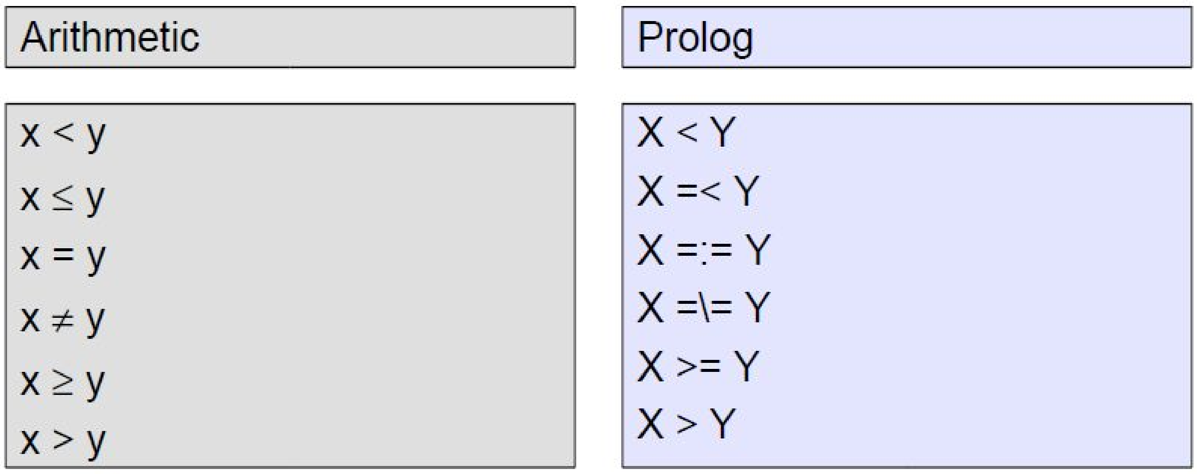


```
acclen([],Acc,Length):-
    Length = Acc.

acclen([_|L],OldAcc,Length):-
    NewAcc is OldAcc + 1,
    acclen(L,NewAcc,Length).
```

A wrapper predicate can be used to allow for shorter input on the users part. Acclen is better than the standard length function implemented through recursion as it implements tail-recursion.

In **tail recursive** predicates, the result is fully calculated once we reach the base clause. However in recursive predicates, there are still goals on the stack when we reach the base clause.

Comparing Integers in Prolog:



| Arithmetic | Prolog |
|---|---|
| $x < y$ | X < Y |
| $x \leq y$ | X =< Y |
| $x = y$ | X =:= Y |
| $x \neq y$ | X =\= Y |
| $x \geq y$ | X >= Y |
| $x > y$ | X > Y |

Comparison operators force both sides of the operator to be evaluated, since it is necessary for the comparison to take place.

# 6 - More Lists

An important predicate when working with lists is append/3, whose arguments are all lists. Declaratively, append(L1,L2,L3) is true if L3 is the result of concatenating the lists L1 and L2 together. From a procedural perspective, we can simply input L3 is a variable to get the result of appending lists. Recursive definition of append/3 is as follows:

```
append([], L, L).
append([H|L1], L2, [H|L3]):-
    append(L1, L2, L3).
```

## Uses of Append

There are many applications of the Append/3 predicate, including finding list suffixes and prefixes.

### prefix/2

A list P is a prefix of some list L when there is some list such that L is the result of concatenating P with that list. It is defined as follows:

```
prefix(P,L):-
   append(P,_,L).
```

### suffix/2

A list S is a suffix of some list L when there is some list such that L is the result of concatenating that list with S. It is defined as follows:

```
suffix(S,L):-
   append(_,S,L).
```

### sublist/2

Using the above suffix and prefix predicates, it becomes very easy to list all the sublists of a list since all sublists are prefixes of suffixes of a list. The predicate is defined as follows:

```
sublist(Sub,List):-
   suffix(Suffix,List),
   prefix(Sub,Suffix).
```

## accReverse/3

Prolog only allows easy access to the head of a list, so it is useful to be able to reverse the list and work with the head, for easier access to tail elements. There is a naive implementation, that requires appending lists but there is a more efficient way to reverse a list. Simply traverse through a list and append the head of the list to the head of the accumulator. By the time you reach the empty list, the accumulator contains the reversed list. The implementation of this is as follows:

```
accReverse([ ],L,L).
accReverse([H|T],Acc,Rev):-
   accReverse(T,[H|Acc],Rev).
```

We can now add a wrapper predicate. Yeno. To make it pretty and stuffs.

```
reverse(L1,L2):-
   accReverse(L1,[ ],L2).
```

# 7 - Definite Clause Grammars

## Context Free Grammars

Context Free Grammars are powerful mechanisms that can handle most syntactic aspects of **natural languages**. Prolog has a special notation for defining grammars, known as Definite Clause Grammars (DCG’s). Below is an example of a Context Free Grammar:

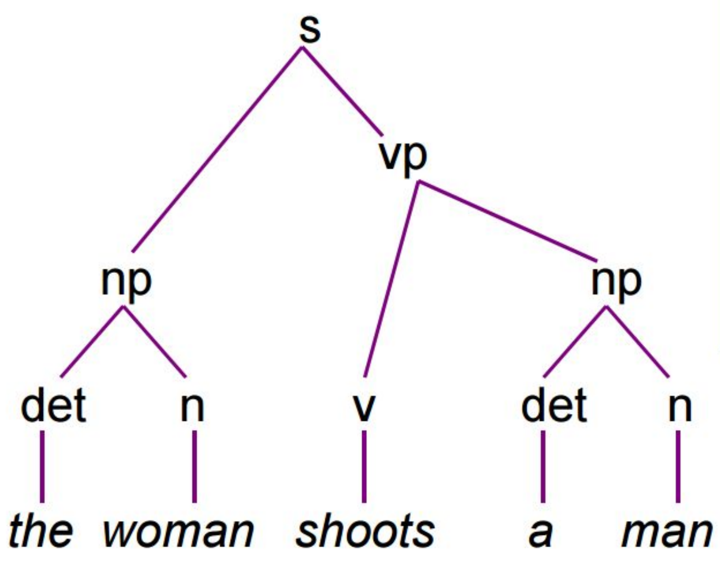
s → np vp  
np → det n  
vp → v np  
vp → v  
det → *the*  
det → *a*  
n → *man*  
n → *woman*  
v → *shoots*

The -> Symbol is used to define rules. The symbols s, np, vp, det, n, v are called the **non-terminal** symbols - meaning these symbols can be evaluated. The symbols in italics are **terminal** symbols. The non terminal symbols in this grammar have traditional meaning in linguistics:

| | |
|---|---|
| **np** | Noun Phrase |
| **vp** | Verb Phrase |
| **det** | Determiner |
| **n** | Noun |
| **v** | Verb |
| **s** | Sentence |

In a linguistic grammar, the non-terminal symbols usually correspond to **grammatical categories**, and the terminal symbols are called **lexical items**. The grammar above contains nine context free rules. A context free rule consists of A single non-terminal symbol, followed by -> and a finite sequence of terminal or non-terminal symbols.

The sentence "The woman shoots a man" would be structured as follows by our CFG:



s → np vp
np → det n
vp → v np
vp → v
det → *the*
det → *a*
n → *man*
n → *woman*
v → *shoots*

Trees representing the syntactic structure of a string are often called **parse trees**. Parse trees are important as they give us information about the string, and information about the structure.

### Grammatical Strings

If we are given a string of words and a grammar, and it turns out we can build a parse tree, then we say that the String is **grammatical.** If we cannot build a parse tree, the given string is **ungrammatical,** for example “a shoots woman” is ungrammatical.

### Generated Language

The language generated by a grammar consists of all the strings that the grammar classifies as grammatical.

## Recogniser

A context free **recogniser** is a program which correctly tells us whether or not a string belongs to the language generated by a context free grammar. In other words, a recogniser classifies strings as grammatical or ungrammatical.

## Parser

A context free **parser** correctly decides whether a string belongs to the language generated by a context free grammar, but it also gives information on the string structure.

### Tl;dr

A recogniser just says yes or no, a parser also gives us a parse tree.

## Context Free Language

A context free language is a language that can be generated by a context free grammar. Some human languages are context free, while others are not.

## Writing a Context Free Grammar Recogniser

Using lists to represent strings, we can parse rules by treating them as lists being appended to produce a result. So in the instance of s -> np vp, we can consider it concatenating a noun phrase and verb phrase to form a String. We can do this using prolog append/3.

```
s(C):- np(A), vp(B), append(A,B,C).
np(C):- det(A), n(B), append(A,B,C).
vp(C):- v(A), np(B), append(A,B,C).
vp(C):- v(C).
det([the]).       det([a]).
n([man]).         n([woman]).       v([shoots]).
```

```
?- s([the,woman,shoots,a,man]).
yes
?-
```

While this recogniser works, it's a bit inefficient since it doesn't use the input string at all to guide the search. There are also a lot of calls to predicates with uninstantiated variables. A more efficient implementation can be achieved using **difference lists**. This is a sophisticated prolog technique for representing and working with lists. Examples are:

| | |
|---|---|
| [a,b,c]-[ ] | is the list [a,b,c] |
| [a,b,c,d]-[d] | is the list [a,b,c] |
| [a,b,c\|T]-T | is the list [a,b,c] |
| X-X | is the empty list [ ] |

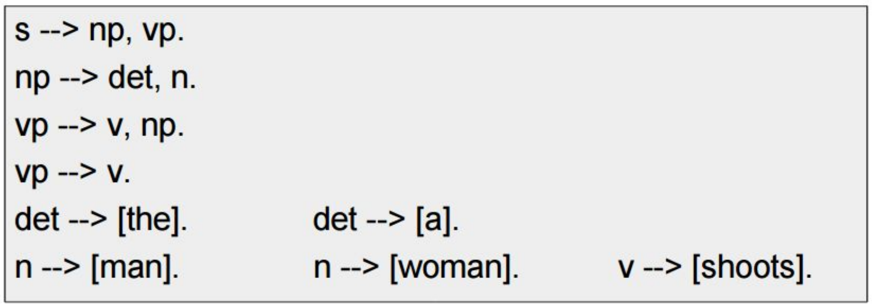
The working CFG recogniser using difference lists is implemented as follows:

```
s(A-C):- np(A-B), vp(B-C).
np(A-C):- det(A-B), n(B-C).
vp(A-C):- v(A-B), np(B-C).
vp(A-C):- v(A-C).
det([the|W]-W).         det([a|W]-W).
n([man|W]-W).   n([woman|W]-W).     v([shoots|W]-W).
```

The problem with difference lists is that although they are more efficient, it is incredibly difficult to understand what is going on list difference implementations and it can be difficult keeping track of all the variables. Luckily, there is a prolog implementation that is as efficient as using diff lists and more readable. This implementation is known as a **Definite Clause Grammar**.

## Definite Clause Grammars

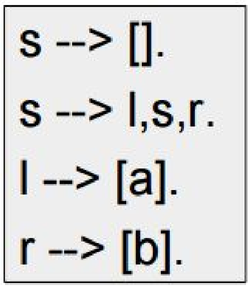
DCGs are a nice notation for writing grammars that hides the underlying difference list variables. The recogniser can be written as follows in a DCG:



```
s --> np, vp.
np --> det, n.
vp --> v, np.
vp --> v.
det --> [the].          det --> [a].
n --> [man].            n --> [woman].          v --> [shoots].
```

DCGs are a nice notation but you cannot write arbitrary context free grammars as a DCG and have it run without problems. DCG’s are ordinary prolog rules in disguise.

DCGs can also be defined for formal languages, which is simply a set of strings. We can define the language a^nb^n as follows:



```
s --> [].
s --> l,s,r.
l --> [a].
r --> [b].
```

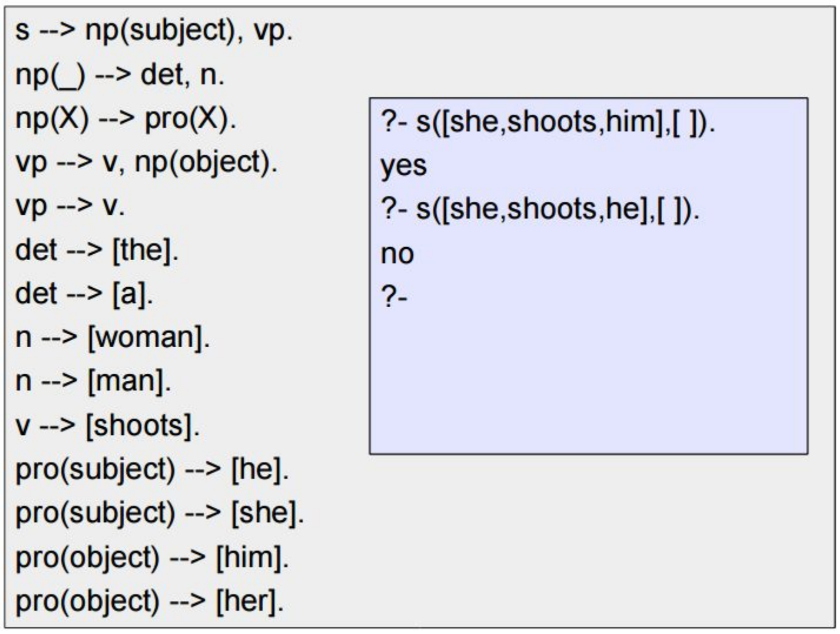
# 8 - More DCGs

DCGs offer more functionality than outlined above, in that they allow us to specify extra arguments which can be used for many purposes.

Suppose we extend the functionality of the previously defined DCG to include pronouns. Strings like “she shoots him” are now acceptable. But so is “her shoots she”. The problem is the DCG ignores basic rules of english:

- *She* and *He* are subject pronouns and cannot be used in an object position.
- *Her* and *Him* are object pronouns and cannot be used in subject position.

In order for our DCG to recognise these rules of english, we must extend the grammar to understand the difference. This can be done using parameters as follows:

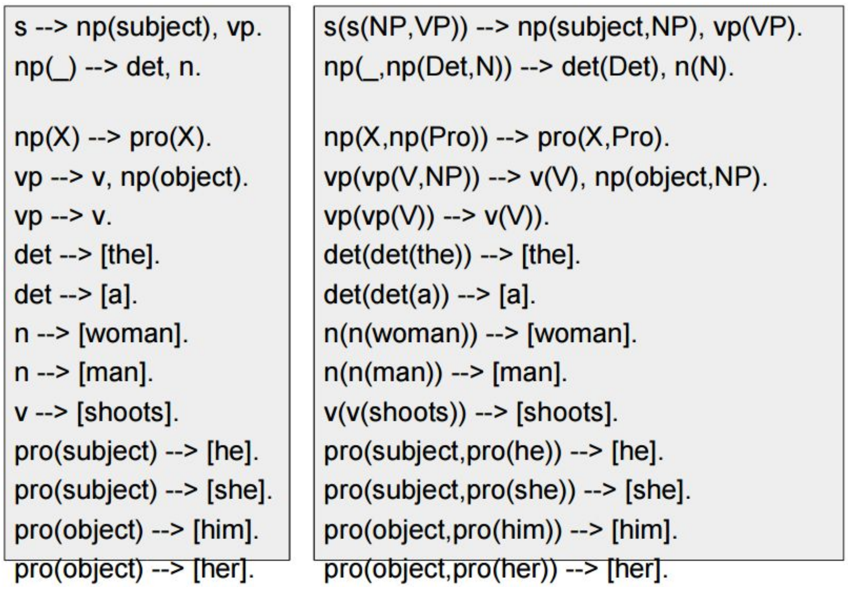


```
s --> np(subject), vp.
np(_) --> det, n.
np(X) --> pro(X).
vp --> v, np(object).
vp --> v.
det --> [the].
det --> [a].
n --> [woman].
n --> [man].
v --> [shoots].
pro(subject) --> [he].
pro(subject) --> [she].
pro(object) --> [him].
pro(object) --> [her].
```

```
?- s([she,shoots,him],[ ]).
yes
?- s([she,shoots,he],[ ]).
no
?-
```

## Parse Trees

So far we've built recognisers using DCGs, so now we need to build a parser in order to see the structure of a grammatical sentence. The above recogniser can be rewritten as a parser like this:



```
s --> np(subject), vp.
np(_) --> det, n.

np(X) --> pro(X).
vp --> v, np(object).
vp --> v.
det --> [the].
det --> [a].
n --> [woman].
n --> [man].
v --> [shoots].
pro(subject) --> [he].
pro(subject) --> [she].
pro(object) --> [him].
pro(object) --> [her].
```

```
s(s(NP,VP)) --> np(subject,NP), vp(VP).
np(_,np(Det,N)) --> det(Det), n(N).

np(X,np(Pro)) --> pro(X,Pro).
vp(vp(V,NP)) --> v(V), np(object,NP).
vp(vp(V)) --> v(V)).
det(det(the)) --> [the].
det(det(a)) --> [a].
n(n(woman)) --> [woman].
n(n(man)) --> [man].
v(v(shoots)) --> [shoots].
pro(subject,pro(he)) --> [he].
pro(subject,pro(she)) --> [she].
pro(object,pro(him)) --> [him].
pro(object,pro(her)) --> [her].
```

## Beyond Context Free Grammars

DCGs can deal with a lot more than just context free grammars, as the extra arguments give us the tools for coping with any computable language. For example, we can write a DCG to check that a string contains an equal number of as, bs and cs. This language is not context free, since for a string to be valid each element must be checked against its neighbour. This DCG can be expressed as follows:

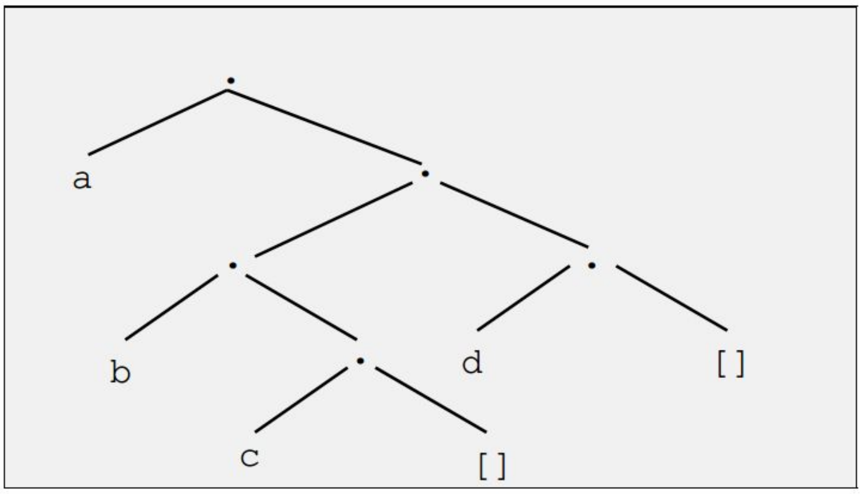
```
s(Count) --> as(Count), bc(Count), cs(Count).

as(0) --> [].
as(NewCnt) --> [a], as(Cnt), {NewCnt is Cnt + 1}.

bs(0) --> [].
bs(NewCnt) --> [b], bs(Cnt), {NewCnt is Cnt + 1}.

cs(0) --> [].
cs(NewCnt) --> [c], cs(Cnt), {NewCnt is Cnt + 1}.
```

# 9 - A Closer Look At Terms

## Built in Comparison Predicates

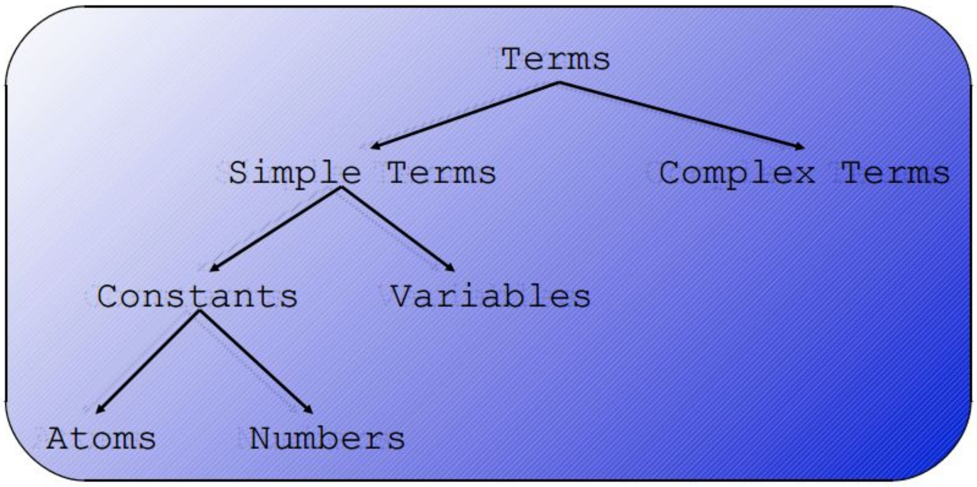
| | |
|---|---|
| = | Unification predicate |
| \= | Negation of unification predicate |
| == | Identity predicate |
| \== | Negation of identity predicate |
| =:= | Arithmetic equality predicate |
| =\= | Negation of arithmetic equality predicate |

The '.' operator is used to combine a term with a list, and is used in the form .(term, list) to produce [term|list]. Internal representations of lists use this . notation, so for example, the list [a,[b,c],d] would be represented as:



## Types of Prolog Terms

There is a type hierarchy in prolog, which is represented as follows:



The following are built in prolog predicates used for type checking.

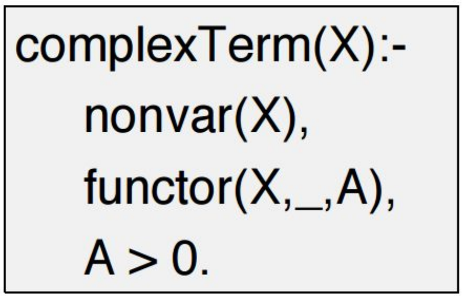
| | |
|---|---|
| atom/1 | Is the argument an atom? |
| integer/1 | Is the argument an integer? |
| float/1 | Is the argument a floating point number? |
| number/1 | Is the argument an integer or floating point number? |
| atomic/1 | Is the argument a constant |
| var/1 | Is the argument an uninstantiated variable? |

| | |
|---|---|
| nonvar/1 | Is the argument an instantiated variable or another term that is not an uninstantiated variable? |

There are also built in prolog predicates for evaluating complex terms. The functor/3 predicate is used for finding the functor name and arity of a complex prolog term. It is defined as **functor(compTerm(A1, A2, A3), F, A).** So for example:

- functor(friends(lou, andy), F, A).
  - F = friends
  - A = 2
- functor(mia, F, A).
  - F = mia
  - A = 0

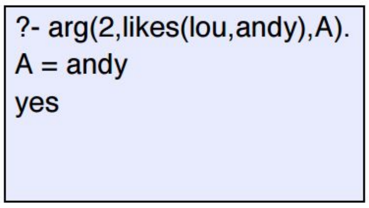
The above type checking methods and the functor method make it easy to check for complex terms if we define the predicate complexTerm(X).



```
complexTerm(X):-
    nonvar(X),
    functor(X,_,A),
    A > 0.
```

Prolog also has a predicate **arg/3** which returns the Nth argument of a complex term. It takes the following parameters:

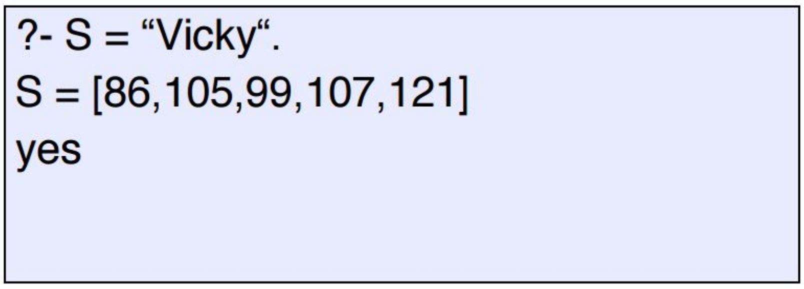
- A number N
- A complex term T
- The Nth argument of T



```
?- arg(2,likes(lou,andy),A).
A = andy
yes
```

## Strings

Strings in prolog are represented by a list of character codes. Prolog offers double quotes for an easy notation for strings.



```
?- S = “Vicky“.
S = [86,105,99,107,121]
yes
```

There are several standard predicates for working with strings, one of the more useful ones is **atom_codes/2** which takes in a string as an atom and a variable, and saves the character codes of the atom to the variable.

## Properties of Operators

**Infix Operators** are when functors are written between their arguments.
**Prefix Operators** are when functors are written before their argument.
**Postfix Operators** are when functors are written after their argument.

Every operator has it's own **precedence**, used when calculating ambiguous expressions.

Prolog uses **associativity** to disambiguate operators with the same precedence value. Operators can be specified as non-associative, in which cases you are forced to use bracketing.

### Defining Operators

Prolog lets you define your own operators, where the definitions look like:

```
:- op(Precedence, Type, Name.
```

Where:

- Precedence is a number between 0 and 1200
- Type is the type of the operator

The types of operators in prolog are defined as:

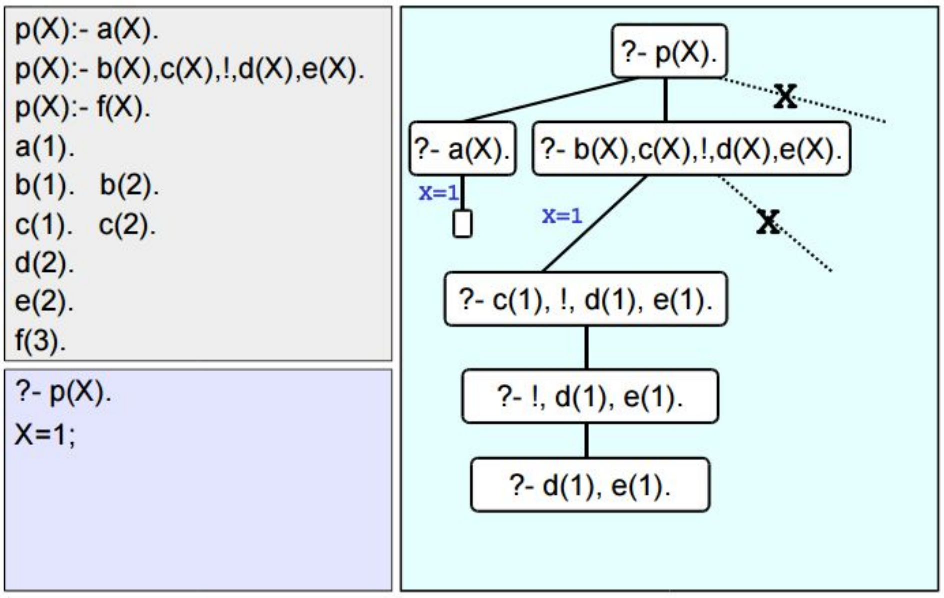
| | |
|---|---|
| yfx | Left associative, infix |
| xfy | Right associative, infix |
| xfx | Non-Associative, infix |
| fx | Non Associative, prefix |
| fy | Right associative, prefix |
| xf | Non-associative, postfix |
| yf | Left-associative, postfix |

# 10 - Cuts and Negation

## !/0

Backtracking is a characteristic feature of prolog, but can lead to inefficiency. It can explore possibilities that lead nowhere, and as such it would be helpful to have some control over backtracking. This is where cuts(!/0) come in.

Cut is a predicate that always succeeds. When used, it commits the prolog operation to the choices made since the parent goal began, i.e. the predicates that came before the cut. An example of a cut can be seen as follows:



```
p(X):- a(X).
p(X):- b(X),c(X),!,d(X),e(X).
p(X):- f(X).
a(1).
b(1).   b(2).
c(1).   c(2).
d(2).
e(2).
f(3).
```

```
?- p(X).
X=1;
```

Cuts commit us to choices the parent goal made since the parent goal was unified with the left hand side of the clause containing the cut.

### Green Cuts

Cuts that do not change the meaning of a predicate are called green cuts, meaning new code gives exactly the same answers as code without the cut, but it is now more efficient.

### Red Cuts

Cuts that change the meaning of a predicate are called red cuts. Programs containing red cuts:

- Are not fully declarative
- Can be hard to read
- Can lead to subtle programming mistakes