Commands end with a period. Prolog has two sides rules/facts & queries.

1- Read value from user.

```
read(VARIABLE). //saves the user input (string) into the variable
```

2- display a variable's value or a string.

```
X= VALUE, write (X). // variable names must start with a capital letter.
write("string") OR write('string') OR write(string).
```

3- write a new line.

```
nl.
```

4- Use more than one command.

```
COMMAND1 , COMMAND2 , … , COMMANDn. // separate commands by a comma.
```

5- Compare a variable to the result of an arithmetic operation.

```
VARIABLE is OPERATION. // 'is' used to calculate the result & compare it to the variable's value.
                       // example: Var = 5, Var is 6-1.
```

6- Adding a fact.

```
FACT_FUNCTION(VALUE). // the function is what is used later in queries.
                     // returns true || false based on if the fact exists or not.
```

7- Assign a value to another value(Fact).

```
FUNCTION(VALUE1,VALUE2).              // function name can be anything.
                                      // example: parent(ahmed, mohamed).
                                      // example: plays(ahmed, football).
```

8- Ask if a value is linked to another value (query).

```
FUNCTION(VALUE1,VALUE2).     // returns true || false.
                             // example: plays(mohamed, football). Output: false.
```

9- Create a query function (rule).

```
QUERY_FUNCTION (PARAM1,PARAM2, …):-
         // code/functions goes here
.        // terminated by a period

example: footballer_parent(Parent,Footballer):-
         parent(Parent,Footballer), plays(Footballer,football).
```

10- Use a query function (query).

```
QUERY_FUNCTION(PARAM1,PARAM2, …).
// can be used to:
         Search for values that make the function true (can be used in rules).
                  Ex: Footballer_parent(Var1,Var2).
         Test if some values make the function true.
                  Ex: Footballer_parent(ahmed,mohamed).
         Mix between the two.
                  Ex: Footballer_parent(ahmed,Var).
```

11- Check if two values aren't equal.

```
VARIABLE1 \= VARIABLE2. // values are treated as strings.
```

12- Negate something.

```
\+ SOMETHING. // true becomes false, and false becomes true.
```

13- Create a function that returns a value.

```
FUNCTION(PARAM1, PARAM2, …, RESULT):-
                    // code goes here.
                    RESULT is // value here.
.
```

14- Comparison operators (facts, rules, queries).

Operator	Meaning
X > Y	X is greater than Y
X < Y	X is less than Y
X >= Y	X is greater than or equal to Y
X =< Y	X is less than or equal to Y
X =:= Y	the X and Y values are equal
X =\= Y	the X and Y values are not equal

15- Arithmetic operators. (facts, rules, queries).

Operator	Meaning
+	Addition
-	Subtraction
*	Multiplication
/	Division
** OR ^	Power
//	Integer Division
mod	Modulus

16- If elseif else statements (using functions & comparison operators).

```
FUNCTION(PARAM(s)):-
                    CONDITION,  // first case
.
FUNCTION(PARAM(s):-
                    CONDITION2. //second case
. // make as many functions as you want but conditions must be different
// Example:
older(Person1,Person2):-
    Person1 > Person2, write('the first person is older').

older(Person1,Person2):-
    Person1 < Person2, write('the second person is older').

older(Person1,Person2):-
    Person1 =:= Person2, write('they are the same age').
```

17- Loops (using recursion).

```
FUNCTION (PARAM(s)):-
         BASE_CASE // can be left empty to stop the function.
.
FUNCTION(PARAM(s)):-
         // code goes here.
         // increment/decrement but use a different variable (NEW).
         FUNCTION(NEW) // calling the function again using the new param.
.
// example:
factorial(0, 1).
factorial(N, Result):-
  N > 0,
  N1 is N - 1,
  factorial(N1, Result1),
  Result is N * Result1.
```

18- Add space between things.

```
tab(NUMBER). // add the number of spaces you want.
```

19- Clean queries.

```
write('\e[H\e[2J').
```

20- Disjunction (OR operator).

```
(GOAL1;GOAL2). // if goal1 is true it ignores goal2. If it isn't true, it checks if the second is.
               // can be used in functions & queries.
Example:
even_or_odd(X):-
  (X mod 2 =:= 0, write(X), write(' is even'); write(X), write(' is odd') )
.
```

21- Get ascii code of a letter.

```
char_code(CHAR_VARIABLE,CODE_VARIABLE).
// example
char_code('a',Ascii).
```

22- If else statements (using arrow operator & disjunction).

```
(CONDITION -> CODE ; ELSE_CODE).
// example
is_capital(Char_Var):-
  char_code(Char_Var, Code_Var),
  (Code_Var >= 65, Code_Var =< 90) -> write('capital letter'); write('not capital)
.
```

23- Create a list.

```
LIST_NAME = [ITEM1, ITEM2, …]. // lists are represented by a head (the first element) and a tail
                               // the rest of the elements.
                               // example: List = [1,2,3] => Head = 1, Tail = [2,3].
                               // this is represented like this List = [Head | Tail] in prolog.
```

List Operations.

1- Search.

```
search (ELEMENT,[ELEMENT|_ ]).         // cant access lists inside of lists.
search (ELEMENT,[_|TAIL]):- search(ELEMENT,TAIL).
```

2- Add to the start of the list (head).

```
search (ELEMENT,[ELEMENT|_]).
search (ELEMENT,[_|TAIL]):- search(ELEMENT,TAIL).

add_head (NEW, LIST, LIST):- // if element is already in list return false.
   search (NEW,LIST), !
.
add_head (NEW, LIST, [NEW|LIST]). // if it isn't in the list make it the head of the new list.
```

3- Add to the end of the list (tail).

```
add_tail (ELEMENT, [], [ELEMENT]).
add_tail (ELEMENT, [HEAD | TAIL], [HEAD | NEW_TAIL]) :-
    add_tail (ELEMENT, TAIL, NEW_TAIL)
. // there is a built in function called append(List, [Element], Result).
```

4- Delete an element.

```
delete_element (ELEMENT, [ELEMENT], []). // if there are no other elements, return an empty one.
delete_element (ELEMENT, [ELEMENT|TAIL],TAIL). // if the element is the head, return the tail.
delete_element (ELEMENT,[HEAD|OLD_TAIL],[HEAD|NEW_TAIL]):-
  delete_element (ELEMENT, OLD_TAIL,NEW_TAIL)
.
```

5- Calculate the length.

```
list_length ([],0). // if list is empty return 0.
list_length ([_|TAIL],LENGTH):- // works by backtracking after emptying the list and adds one each
time.
  list_length (TAIL,TAIL_LENGTH), LENGTH is TAIL_LENGTH +1
.
```

6- Add two lists together (concatenate).

```
concat_lists ([],LIST,LIST).
concat_lists ([HEAD1|TAIL1], LIST2, [HEAD1|NEW_TAIL]):-
  concat_lists (TAIL1, LIST2, NEW_TAIL)
. // the append function can replace this function.
```

7- Reverse a list.

```
add_tail (ELEMENT, [], [ELEMENT]).
add_tail (ELEMENT, [HEAD | TAIL], [HEAD | NEW_TAIL]) :-
    add_tail (ELEMENT, TAIL, NEW_TAIL)
.
reverse_list([], []).
reverse_list([HEAD | TAIL], REVERSED_LIST) :-
    reverse_list(TAIL, REVERSED_TAIL),
    add_tail(HEAD, REVERSED_TAIL, REVERSED_LIST)
. // add_tail can be replaced with: append(REVERSED_TAIL, [HEAD], REVERSED_LIST).
```

For a video explaining some list operations (length, concat, delete, append) click here.