Simple Facts
In Prolog we can make some statements by using facts. Facts either consist of a particular item or a relation between items.

For example, we can represent the fact that it is sunny by writing the program:

```
sunny.
```

We can now ask a query of Prolog by asking

```
?- sunny.
```

?- is the Prolog prompt.

To this query, Prolog will answer yes. sunny is true because (from above) Prolog matches it in its database of facts.

Facts have some simple rules of syntax. Facts should always begin with a lowercase letter and end with a full stop. The facts themselves can consist of any letter or number combination, as well as the underscore _ character. However, names containing the characters -,+,*,/, or other mathematical operators should be avoided.

Examples of Simple Facts
Here are some simple facts about an imaginary world. /* and */ are comment delimiters

```
john_is_cold.                    /* john is cold */
raining.                         /* it is raining */
john_Forgot_His_Raincoat.        /*  john forgot his raincoat */
fred_lost_his_car_keys.          /* fred lost is car keys */
peter_footballer.                /* peter plays football */
```

These describe a particular set of circumstances for some character john. We can interrogate this database of facts, by again posing a query. For example: {note the responses of the Prolog interpreter are shown in italics}

```
?- john_Forgot_His_Raincoat.

yes

?- raining.

yes

?- foggy.

no
```

The first two queries succeed since they can be matched against facts in the database above. However, foggy fails (since it cannot be matches) and Prolog answers no since we have not told it this fact.
Simple Fact Exercises

Which of the following are syntactically correct ?

```
Hazlenuts.    No  because it starts with a capital letter
tomsRedCar.   Yes capital letters within the word are ok
2Ideas.       No  it should not start with a number
Prolog.       No  starts with a capital letter
```

Given the database below, study the queries below it. Again indicate whether you think the goal will succeed or not by answering yes or no as prompted.

blue_box.
red_box.
green_circle.
blue_circle.
orange_triangle.

?- green_circle.

?- circle_green.

?- red_triangle.

?- red_box.

?- orange_Triangle.

?- green_circle. **Yes**
    It appears as the third entry in the database.
?- circle_green. **No**
    This is not the same statement as above.
?- red_triangle. **No**
    There is no identical entry in the database.
?- red_box. **Yes**
    It appears as the second entry in the database.
?- orange_Triangle. **No**
    The capitalisation of the t in triangle means that this will not match with the last entry in the database.

Facts with Arguments

More complicated facts consist of a relation and the items that this refers to. These items are called arguments. Facts can have arbitrary number of arguments from zero upwards. A general model is shown below:

```
relation(<argument1>,<argument2>,....,<argumentN> ).
```

Relation names must begin with a lowercase letter

```
likes(john,mary).
```

The above fact says that a relationship likes links john and mary. This fact may be read as either john likes mary or mary likes john. This reversibility can be very useful to the programmer, however it can also lead to some pitfalls. Thus you should always be clear and consistent how you intend to interpret the relation and when.

Finally, remember names of the relations are defined by you. With the exception of a few relations that the system has built into it, the system only knows about relations that you tell it about.

Facts with Arguments Examples 1

An example database. It details who eats what in some world model.

```
eats(fred,oranges).          /* "Fred eats oranges" */
eats(fred,t_bone_steaks).       /* "Fred eats T-bone steaks" */
eats(tony,apples).           /* "Tony eats apples" */
eats(john,apples).           /* "John eats apples" */
eats(john,grapefruit).        /* "John eats grapefruit" */
```

If we now ask some queries we would get the following interaction:

```
?- eats(fred,oranges).     /* does this match anything in the database? */
yes                        /* yes, matches the first clause in the database */

?- eats(john,apples).      /*  do we have a fact that says john eats apples? */
yes                        /* yes we do, clause 4 of our eats database */

?- eats(mike,apples).      /* how about this query, does mike eat apples */
no                         /* not according to the above database. */

?- eats(fred,apples).      /* does fred eat apples */
no                         /* again no, we don't know whet
```

Variables and Unification

How do we say something like "What does Fred eat"? Suppose we had the following fact in our database:

```
eats(fred,mangoes).
```

How do we ask what fred eats. We could type in something like

```
?- eats(fred,what).
```

However Prolog will say no. The reason for this is that what does not match with mangoes. In order to match arguments in this way we must use a Variable. The process of matching items with variables is known as unification. Variables are distinguished by starting with a capital letter. Here are some examples:

```
X         /* a capital letter */

VaRiAbLe  /* a word - it be made up or either case of letters */

My_name   /* we can link words together via '_' (underscore) */
```

Thus returning to our first question we can find out what fred eats by typing

```
?- eats(fred,What).

What=mangoes

yes
```

As a result of this query, the variable What has matched (or unified) with mangoes. We say that the variable What now has the binding mangoes. When we pose a query, if the query is successful, Prolog prints both the variable and the variable name, as we see above.

Variable Examples 1

Let's consider some examples using facts. First consider the following database.

```
loves(john,mary).

loves(fred,hobbies).
```

Now let's look at some simple queries using variables

```
?- loves(john,Who).      /* Who does john love? */

Who=mary                 /* yes , Who gets bound to mary */

yes                      /*  and the query succeeds*/

?- loves(arnold,Who)     /* does arnold love anybody */
```

```
no                        /* no,  arnold doesn't match john or fred */

?- loves(fred,Who).       /* Who does fred love */

Who = hobbies               /*  Note the to Prolog Who is just the name of a
variable, it */

yes                      /* semantic connotations are not picked up, hence  Who
unifies */

                          /* with hobbies */
```

Variable Examples 2

Here are some more difficult object/variable comparisons. Consider the following database showing a library of cassette tapes. Notice the final argument to tape, which is the name of a favourite song from the album.

```
tape(1,van_morrison,astral_weeks,madam_george).

tape(2,beatles,sgt_pepper,a_day_in_the_life).

tape(3,beatles,abbey_road,something).

tape(4,rolling_stones,sticky_fingers,brown_sugar).

tape(5,eagles,hotel_california,new_kid_in_town).
```

Let's now look at some queries.

```
?- tape(5,Artist,Album,Fave_Song).      /* what are the contents of tape 5 */

Artist=eagles

Album=hotel_california

Fave_Song=new_kid_in_town

yes

?- tape(4,rolling_stones,sticky_fingers,Song).  /* find just  song */

Song=brown_sugar                     /* which you like best from the album */

yes
```

Proof Searching

Let’s assume we have the following knowledge base:

young(alice).
young(bob).
fit(alice). fit(bob).
happy(bob).
satisfied(X) :- young(X), fit(X), happy(X).

and we want to ask the query
?- satisfied(Who).
It should be obvious that the only result should be when Who = bob, but how does Prolog work this out?
What Prolog does is that it creates a new goal from our query, and creates a brand new variable, like _G35, to represent the shared variables. Since we gave Prolog the goal of "an individual with the property satisfied", it finds the rule associated with that property, and replaces the goal with three new goals, "an individual with the properties young, fit, and happy".

```
?- satisfied(Who).
        |
?- satisfied(_G35).
        |
?- young(_G35), fit(_G35), happy(_G35).
```

At this point, Prolog will try to prove each of these subproperties, by creating a new goal, and working left to right. First, it tries to prove young(_G35) – but we have young(alice) in our knowledge base, so _G35 becomes unified with the atom alice. So now Prolog has proved the first of the three subgoals, it continues, trying to prove the next one.

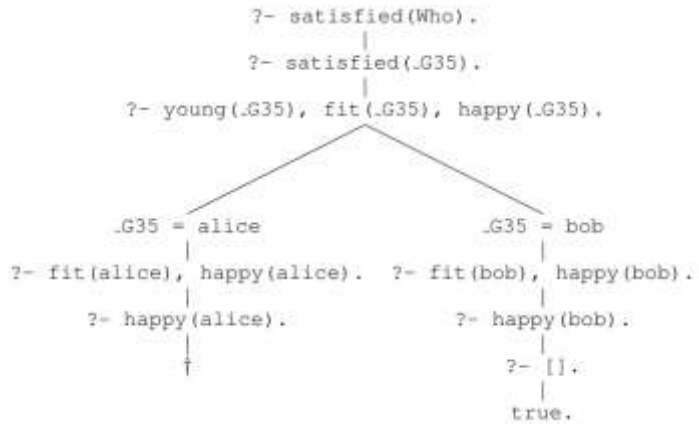
```
?- satisfied(Who).
        |
?- satisfied(_G35).
        |
?- young(_G35), fit(_G35), happy(_G35).
        |
    _G35 = alice
        |
?- fit(alice), happy(alice).
```

Since we also have the fact fit(alice) in our knowledge base, we can satisfy the next goal too.

```
?- satisfied(Who).
        |
?- satisfied(_G35).
        |
?- young(_G35), fit(_G35), happy(_G35).
        |
    _G35 = alice
        |
?- fit(alice), happy(alice).
        |
?- happy(alice).
```

However, now the goal list is just happy(alice), and there's nothing in our knowledge base which can unify with this goal, and so this line of searching fails.

So what next? Prolog decides it made a mistake, and so it backtracks to the last time it had to make a decision, and tries to see if there were any other choices it could have made.
In our example, this is when the variable _G35 was instantiated to alice – there was another choice, and so Prolog now instantiates the variable to that choice, bob.
It works through the steps as before, except that this time when it gets to the goal happy(bob), it can find this fact in the knowledge base, and so the query succeeds.



Now that we've reached the bottom of a path and found a successful route to satisfy all the goals, Prolog will pass back up the result of any variable instantiations it made. So, it will tell us that Who = _G35 = bob, though it skips out the generated variable, and just gives us back Who = bob. This backtracking till a goal is reached is the same way Prolog works when you ask it to give more results using the ; key.