Pure Prolog

Eric Martin, CSE, UNSW

COMP9021 Principles of Programming

[1]:
```python
from collections import defaultdict, deque
from copy import copy, deepcopy
from itertools import islice
from pprint import pprint
```

A *definite logic program* consists of *definite clauses*, that is, *universal closures* of implications whose right hand sides are *atomic formulas* and whose left hand sides are conjunctions of atomic formulas. Because a conjunction over an empty set of formulas is logically true, and an implication whose left hand side is a tautology is logically equivalent to its right hand side, atomic formulas, also known as *atoms* or *facts*, are particular cases of definite clauses. Here is a definite logic program consisting of 13 definite clauses, the first 8 of which are facts:

$\mathrm{father}(\mathrm{bob}, \mathrm{jack})$
$\mathrm{father}(\mathrm{bob}, \mathrm{sandra})$
$\mathrm{father}(\mathrm{john}, \mathrm{bob})$
$\mathrm{father}(\mathrm{john}, \mathrm{mary})$
$\mathrm{mother}(\mathrm{jane}, \mathrm{jack})$
$\mathrm{mother}(\mathrm{jane}, \mathrm{sandra})$
$\mathrm{mother}(\mathrm{emily}, \mathrm{bob})$
$\mathrm{mother}(\mathrm{emily}, \mathrm{mary})$
$\forall \mathrm{X} \forall \mathrm{Y}\big(\mathrm{father}(\mathrm{X}, \mathrm{Y}) \rightarrow \mathrm{parent}(\mathrm{X}, \mathrm{Y})\big)$
$\forall \mathrm{X} \forall \mathrm{Y}\big(\mathrm{mother}(\mathrm{X}, \mathrm{Y}) \rightarrow \mathrm{parent}(\mathrm{X}, \mathrm{Y})\big)$
$\forall \mathrm{X} \forall \mathrm{Y}\big(\mathrm{parent}(\mathrm{Y}, \mathrm{X}) \wedge \mathrm{male}(\mathrm{X}) \rightarrow \mathrm{son}(\mathrm{X}, \mathrm{Y})\big)$
$\forall \mathrm{X} \forall \mathrm{Y}\big(\mathrm{parent}(\mathrm{Y}, \mathrm{X}) \wedge \mathrm{female}(\mathrm{X}) \rightarrow \mathrm{daughter}(\mathrm{X}, \mathrm{Y})\big)$
$\forall \mathrm{X} \forall \mathrm{Y} \forall \mathrm{Z}\big(\mathrm{male}(\mathrm{X}) \wedge \mathrm{parent}(\mathrm{Z}, \mathrm{X}) \wedge \mathrm{parent}(\mathrm{Z}, \mathrm{Y}) \rightarrow \mathrm{brother}(\mathrm{X}, \mathrm{Y})\big)$
$\forall \mathrm{X} \forall \mathrm{Y} \forall \mathrm{Z}\big(\mathrm{parent}(\mathrm{X}, \mathrm{Z}) \wedge \mathrm{parent}(\mathrm{Z}, \mathrm{Y}) \rightarrow \mathrm{grandparent}(\mathrm{X}, \mathrm{Y})\big)$

In English, this reads as:

- Bob is a father of Jack and Sandra, John is a father of Bob and Mary, Jane is a mother of Jack and Sandra, Emily is a mother of Bob and Mary.
- For all X, for all Y, if X is a father of Y then X is a parent of Y.
- For all X, for all Y, if X is a mother of Y then X is a parent of Y.
- For all X, for all Y, if Y is a parent of X and X is male then X is a son of Y.
- For all X, for all Y, if Y is a parent of X and X is female then X is a daughter of Y.
- For all X, for all Y, for all Z, if X is male, Z is a parent of X and Z is a parent of Y then X is is a brother of Y.
- For all X, for all Y, for all Z, if X is a parent of Z and Z is a parent of Y, then X is a grandparent of Y.

Note that the last two definite clauses are logically equivalent to:

$\forall X \forall Y \Big(\mathrm{male}(X) \wedge \exists Z \big(\mathrm{parent}(Z, X) \wedge \mathrm{parent}(Z, Y)\big) \rightarrow \mathrm{brother}(X, Y)\Big)$

$\forall X \forall Y \Big(\exists Z \big(\mathrm{parent}(X, Z) \wedge \mathrm{parent}(Z, Y)\big) \rightarrow \mathrm{grandparent}(X, Y)\Big)$

which reads as:

- For all X, for all Y, if X is male and there exists Z such that Z is a parent of X and Z is a parent of Y, then X is is a brother of Y.
- For all X, for all Y, if there exists Z such that X is a parent of Z and Z is a parent of Y, then X is a grandparent of Y.

In Prolog, this definite logic program takes the form:

```
father(bob, jack).
father(bob, sandra).
father(john, bob).
father(john, mary).
mother(jane, jack).
mother(jane, sandra).
mother(emily, bob).
mother(emily, mary).
parent(X, Y) :- father(X, Y).
parent(X, Y) :- mother(X, Y).
son(X, Y) :- parent(Y, X), male(X).
daughter(X, Y) :- parent(Y, X), female(X).
brother(X, Y) :- male(X), parent(Z, X), parent(Z, Y).
grandparent(X, Y) :- parent(X, Z), parent(Z, Y).
```

An English reading of the definite clauses that are not facts that more closely follows this alternative syntax is:

- For all X, for all Y, for X to be a parent of Y, it suffices that X be a father of Y.
- For all X, for all Y, for X to be a parent of Y, it suffices that X be a mother of Y.
- For all X, for all Y, for X to be a son of Y, it suffices that Y be a parent of X and that X be male.
- For all X, for all Y, for X to be a daughter of Y, it suffices that Y be a parent of X and that X be female.
- For all X, for all Y, for X to be a brother of Y, it suffices that X be male and that some Z exists that is a parent of both X and Y.
- For all X, for all Y, for X to be a grandparent of Y, it suffices that some Z exists such that X is a parent of Z and Z is a parent of Y.

The logical syntax suggests successive applications of *instantiation* and *modus ponens* to derive some atoms such as $\mathrm{grandparent}(\mathrm{john}, \mathrm{jack})$ from the given facts, in a *bottom up* manner:

- It is known that $\mathrm{father}(\mathrm{bob}, \mathrm{jack})$. Together with $\forall X \forall Y \big(\mathrm{father}(X, Y) \rightarrow \mathrm{parent}(X, Y)\big)$, this implies that $\mathrm{parent}(\mathrm{bob}, \mathrm{jack})$.
- It is known that $\mathrm{father}(\mathrm{john}, \mathrm{bob})$. Together with $\forall X \forall Y \big(\mathrm{father}(X, Y) \rightarrow \mathrm{parent}(X, Y)\big)$, this implies that $\mathrm{parent}(\mathrm{john}, \mathrm{bob})$.

- From $\forall X \forall Y \forall Z\big(\mathrm{parent}(X, Z) \wedge \mathrm{parent}(Z, Y) \rightarrow \mathrm{grandparent}(X, Y)\big)$ together with $\mathrm{parent}(\mathrm{john}, \mathrm{bob})$ and $\mathrm{parent}(\mathrm{bob}, \mathrm{jack})$, we infer that $\mathrm{grandparent}(\mathrm{john}, \mathrm{jack})$, ending the proof.

The Prolog syntax suggests proving instances of some atomic formulas such as grandparent(john, jack) by proving instances of other atomic formulas that directly imply the latter, until nothing but given facts are eventually reached, in a *top down* manner:

To prove grandparent(john, jack), it suffices to find a value for Z and prove both parent(john, Z) and parent(Z, jack).

- To find a value for Z and prove both parent(john, Z) and parent(Z, jack), it suffices to find a value for Z and either prove both father(john, Z) and parent(Z, jack) or prove both mother(john, Z) and parent(Z, jack).
  - To find a value for Z and prove both father(john, Z) and parent(Z, jack), one can let Z be either bob or mary as indeed father(john, bob) and father(john, mary) are given, having to then prove either parent(bob, jack) or parent(mary, jack).
    - To prove parent(bob, jack), it suffices to prove either father(bob, jack) or mother(bob, jack).
      - Indeed, father(bob, jack) is given, ending the proof.

Note that the definite clauses that could be used were selected in the order in which they appeared. Suppose for instance that the clauses of the definite logic program that are not facts were listed as follows in Prolog form:

parent(X, Y) :- mother(X, Y).
parent(X, Y) :- father(X, Y).
daughter(X, Y) :- parent(Y, X), female(X).
son(X, Y) :- parent(Y, X), male(X).
brother(X, Y) :- male(X), parent(Z, X), parent(Z, Y).
grandparent(X, Y) :- parent(X, Z), parent(Z, Y).

Then the proof of grandparent(john, jack) would proceed as follows:

To prove grandparent(john, jack), it suffices to find a value for Z and prove both parent(john, Z) and parent(Z, jack).

- To find a value for Z and prove both parent(john, Z) and parent(Z, jack), it suffices to find a value for Z and either prove both mother(john, Z) and parent(Z, jack) or prove both father(john, Z) and parent(Z, jack).
  - One cannot find a value for Z and prove mother(john, Z).
  - To find a value for Z and prove both father(john, Z) and parent(Z, jack), one can let Z be either bob or mary as indeed father(john, bob) and father(john, mary) are given, having to then prove either parent(bob, jack) or parent(mary, jack).
    - To prove parent(bob, jack), it suffices to prove either mother(bob, jack) or father(bob, jack).
      - mother(bob, jack) cannot be proved.
      - Indeed, father(bob, jack) is given, ending the proof.

grandparent(john, jack) is a *closed* formula: it contains no variable. Working with formulas that do contain variables, the approach that has been described lets us do more than prove: it lets us compute. Rather than talking about proving an atomic formula, one talks about *solving a goal* or *answering a query*. For instance, solving the goal grandparent(john, X) lets us compute the instantiations of X that make the resulting formula a logical consequence of the logic program in its logical form. With the original ordering of the clauses, this proceeds as follows:

To solve grandparent(john, X), it suffices to find values for Z and Y and solve both parent(john, Z) and parent(Z, Y). One can then give X the value of Y.

- To find values for Z and Y and solve both parent(john, Z) and parent(Z, Y), it suffices to find values for Y and Y_0 and either solve both father(john, Y_0) and parent(Y_0, Y) or solve both mother(john, Y_0) and parent(Y_0, Y). One can then give X the value of Y.
  - To find values for Y and Y_0 and solve both father(john, Y_0) and parent(Y_0, Y), one can let Y_0 be either bob or mary as indeed father(john, bob) and father(john, mary) are given, having to then solve either parent(bob, Y) or parent(mary, Y). One can then give X the value of Y.
    - To find a value for Y and solve parent(bob, Y), it suffices to find a value for Y_0 and solve either father(bob, Y_0) or mother(bob, Y_0). One can then give X the value of Y_0.
      - To find a value for Y_0 and solve father(bob, Y_0), one can let Y_0 be either jack or sandra. This yields two solutions to the original goal: X can take the value jack, or it can take the value sandra.
      - One cannot find a value for Y_0 and solve mother(bob, Y_0).
    - To find a value for Y and solve parent(mary, Y), it suffices to find a value for Y_0 and solve either father(mary, Y_0) or mother(mary, Y_0). One can then give X the value of Y_0.
      - One cannot find a value for Y_0 and solve father(mary, Y_0).
      - One cannot find a value for Y_0 and solve mother(mary, Y_0).
- One cannot find a value for Y_0 and solve mother(john, Y_0).

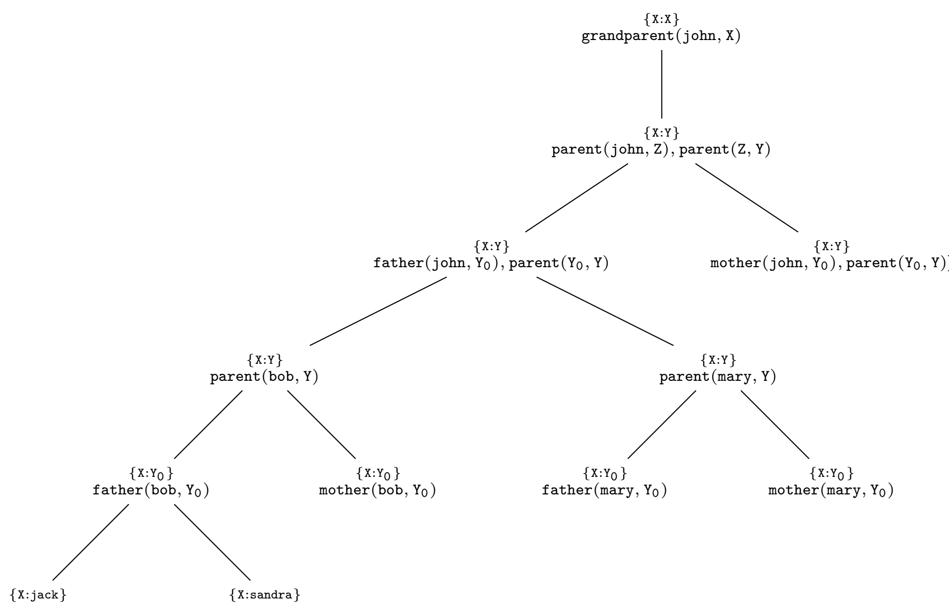
Hence there are two and only two solutions to the goal grandparent(john,X): X equal to jack, and X equal to sandra.

There is a bit of mystery in the preceding description in the way some variables are changed or new variables are introduced. This has to do with the fact that different occurrences of a given variable can be unrelated, which requires to sometimes rename variables and introduces somehow tricky technicalities:

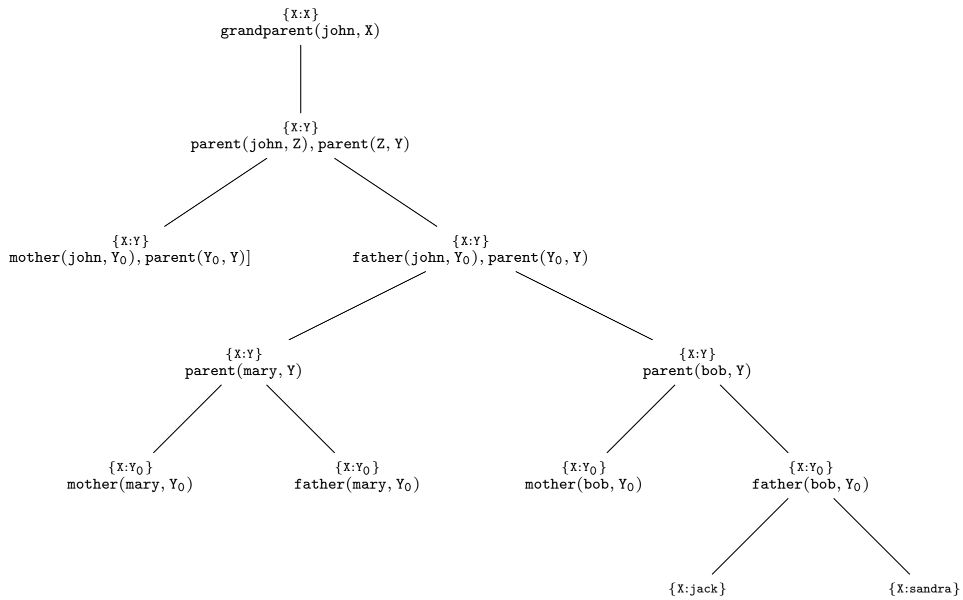
- Observe that the X in the goal grandparent(john, X) is unrelated to the X in the *rule* (a more usual name for "definite clause" with Prolog notation) grandparent(X, Y) :- parent(X, Z), parent(Z, Y). If we want to let the goal interact with the rule, it is safe to rename all occurrences of X in the rule, using a new variable, say X_0: grandparent(X_0, Y) :- parent(X_0, Z), parent(Z, Y). We can then *unify* the goal grandparent(john, X) and the *head* of grandparent(X_0, Y) :- parent(X_0, Z), parent(Z, Y), namely, grandparent(X_0, Y), by replacing X_0 in grandparent(X_0, Y) by john and replacing X in grandparent(john, X) by Y (we could as well replace Y in grandparent(X_0, Y) by X, but the code to be developed later opts for the first alternative). Then the same replacements can be performed in the *body* of grandparent(X_0, Y) :- parent(X_0, Z), parent(Z, Y), namely, parent(X_0, Z), parent(Z, Y), yielding the rule grandparent(john, Y) :- parent(john, Z), parent(Z, Y). This allows one to replace the goal grandparent(john, X) with the goals parent(john, Z) and parent(Z, Y), knowing that the values of Y that yield solutions to those goals are values of X that yield solutions to the original goal.

- Observe that the Y in the goal parent(Z, Y), one of the two goals that we now have to solve, is unrelated to the Y in the rule parent(X, Y) :- father(X, Y). Also, the X in parent(X, Y) :- father(X, Y) is unrelated to the X in the original goal (grandparent(john, X)). If we want to let the other goal we now have to solve, parent(john, Z), interact with that rule, and let the X in the original goal alone, it is safe to rename all occurrences of Y in the rule, using a new variable, say Y_0, and rename all occurrences of X in the rule using a new variable, say X_0: parent(X_0, Y_0) :- father(X_0, Y_0). We can then unify the goal parent(john, Z) and the head of parent(X_0, Y_0) :- father(X_0, Y_0), by replacing X_0 in parent(X_0, Y_0) by john and replacing Z in parent(john, Z) by Y_0. Then the same replacements can be performed in the body of parent(X_0, Y_0) :- father(X_0, Y_0), yielding the rule parent(john, Y_0) :- father(john, Y_0). This allows one to replace the goal parent(john, Z) with the goal father(john, Y_0). But the Z in the goals parent(john, Z) and parent(Z, Y) are related, hence having replaced Z by Y_0 in parent(john, Z), we should replace Z by Y_0 in parent(Z, Y). So we goals we now have to solve are father(john, Y_0) and parent(Y_0, Y).

The description of the search for solutions to the goal grandparent(john, X) has the structure of a tree. The following graphical representation of the description makes it particularly clear:



Observe that with the alternative listing considered above of the clauses of the definite logic program that are not facts, the description of the search for solutions to the goal grandparent(john, X) would be captured by the following tree:



Instead of starting with one goal, one can start with a sequence of (implicitly conjuncted) goals. For instance, if the logic program was extended with the fact female(sandra), then there would be two solutions to the goals grandparent(john, X), daughter(X, Y): X equal to sandra and Y equal to bob, and X equal to sandra and Y equal to jane.

The atoms considered up to now are built from *predicate symbols* (father, mother, parent, grandparent, male, female, son, daughter, brother), all of *arity* 2, that is, all *binary* predicate symbols, *constants* (bob, jack, sandra, john, mary, jane, emily), and variables. Constant and variables are called *terms*. If we introduce *function symbols* (of strictly positive arity, as constants are nothing but function symbols of arity 0, or *nullary* function symbols), then we can build more complex terms. For instance, to represent lists of 0s and 1s, we can use:

- 3 constants, o for 0, i for 1, and e for the empty list;
- a binary function symbol (function symbol of arity 2) l, whose first argument is meant to be o or i and represent the first element of a nonempty list, and whose second argument is meant to be a term representing the rest of the list.

For instance:

- the term l(i, e) represents the list [1].
- the term l(o, l(i, e)) represents the list [0, 1].
- the term l(o, l(o, l(i, e))) represents the list [0, 0, 1].
- the term l(i, l(o, l(o, l(i, e)))) represents the list [1, 0, 0, 1].
- the term l(i, l(i, l(o, l(o, l(i, e))))) represents the list [1, 1, 0, 0, 1].

Terms such as l(e, e), l(i, o), l(l(i, o), e) and l(l(i, e), l(i, e)) are syntactically valid, but are given no interpretation.

We can then consider the logic program that defines the join (concatenation) function:

join(e, X, X).
join(l(H, T), X, l(H, Y)) :- join(T, X, Y).

This reads as:

- Joining the empty list and a list X results in X itself.
- To join a nonempty list and a list X, it suffices to join T and X, with T the list consisting of all elements of the first list except the first element H, resulting in a list Y, and put H at the front.

This allows us to solve a goal such as join(l(i, l(i, l(o, e))), l(o, l(i, e)), X), with as unique solution X equal to l(i, l(i, l(o, l(o, l(i, e))))), reflecting the fact that joining [1, 1, 0] and [0, 1] results in the list [1, 1, 0, 0, 1]. But one can also solve a goal such as join(X, X, Y), so look for all possible ways to join a list with (a copy of) itself. There are infinitely many solutions, for instance:

- X = e and Y = e: joining the empty list with itself results in the empty list.
- X = l(o, e) and Y = l(o, l(o, e)): joining [0] with itself results in [0, 0].
- X = l(i, e) and Y = l(i, l(i, e)): joining [1] with itself results in [1, 1].
- X = l(i, l(o, e)) and Y = l(i, l(o, l(i, l(o, e)))): joining [1, 0] with itself results in [1, 0, 1, 0].

Though they are correct, these solutions are not the most general ones, with the exception of the first one: all others are instances of more general solutions, that themselves contain variables:

- X equal to l(H, e) and Y equal to l(H, l(H, e)): joining a list of the form [H] (where H is equal to 0 or 1) with itself results in the list [H, H].
- X equal to l(H, l(H_0, e)) and Y equal to l(H, l(H_0, l(H, l(H_0, e)))): joining a list of the form [H, H_0] (where H and H_0 are independently equal to 0 or 1) with itself results in the list [H, H_0, H, H_0].

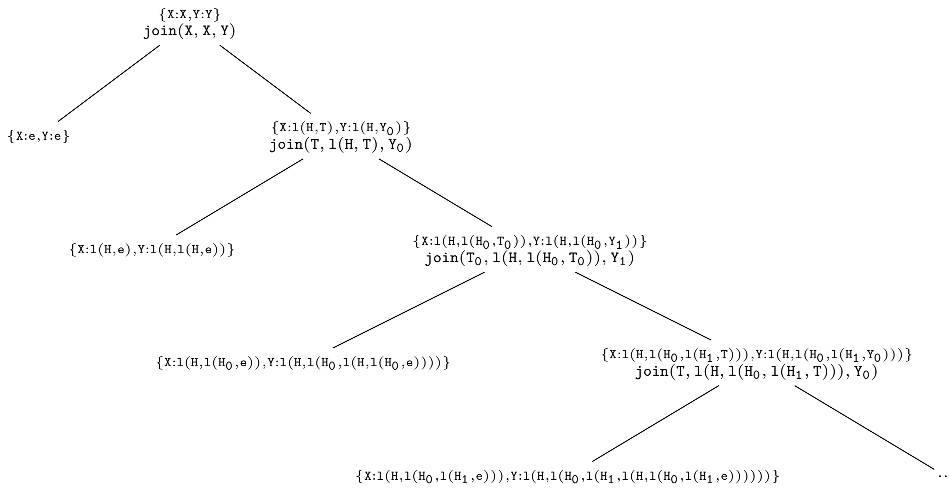
By computing *most general unifiers*, one guarantees that nothing but most general solutions be generated. To solve the goal join(X, X, Y):

- We first consider the fact join(e, X, X), change it to join(e, X_0, X_0), unify join(X, X, Y) and join(e, X_0, X_0) with the equalities X = e, X_0 = X and Y = X_0, from which we derive X = e, X_0 = e and Y = e, which yields the solution X = e and Y = e.
- We then consider the rule join(l(H, T), X, l(H, Y)) :- join(T, X, Y), change it to join(l(H, T), X_0, l(H, Y_0)) :- join(T, X_0, Y_0), unify join(X, X, Y) and join(l(H, T), X_0, l(H, Y_0)) with the equalities X = l(H, T), X_0 = X and Y = l(H, Y_0), from which we derive X = l(H, T), X_0 = l(H, T) and Y = l(H, Y_0), having to now solve the goal join(T, l(H, T), Y_0).
  - We first consider the fact join(e, X, X), change it to join(e, X_0, X_0), unify join(T, l(H, T), Y_0) and join(e, X_0, X_0) with the equalities T = e, X_0 = l(H, T) and Y_0 = X_0, from which we derive T = e, X_0 = l(H, e) and Y_0 = l(H, e), which together with X = l(H, T) and Y = l(H, Y_0), yields the solution X = l(H, e) and Y = l(H, l(H, e)).
  - We then consider the rule join(l(H, T), X, l(H, Y)) :- join(T, X, Y), change it to join(l(H_0, T_0), X_0, l(H_0, Y_1)) :- join(T_0, X_0, Y_1), unify join(T, l(H, T), Y_0) and join(l(H_0, T_0), X_0, l(H_0, Y_1)) with the equalities T = l(H_0, T_0), X_0 = l(H, T) and Y_0 = l(H_0, Y_1), from which we derive T = l(H_0, T_0), X_0 = l(H, l(H_0, T_0)), Y_0 = l(H_0, Y_1), X = l(H, l(H_0, T_0)) and Y = l(H, l(H_0, Y_1)), having to now solve the goal join(T_0, l(H, l(H_0, T_0)), Y_1).
    * We first consider the fact join(e, X, X), change it to join(e, X_0, X_0), unify join(T_0, l(H, l(H_0, T_0)), Y_1) and join(e, X_0, X_0) with the equalities T_0 = e, X_0 = l(H, l(H_0, T_0)) and Y_1 = X_0, from which we derive T_0 = e, X_0 = l(H, l(H_0, e)) and Y_1 = l(H, l(H_0, e)), which together with X = l(H, l(H_0, T_0)) and Y = l(H, l(H_0, Y_1)), yields the solution X = l(H, l(H_0, e))

and Y = l(H, l(H_0, l(H, l(H_0, e)))).

* We then consider the rule join(l(H, T), X, l(H, Y)) :- join(T, X, Y), change it to join(l(H_1, T), X_1, l(H_1, Y_0)) :- join(T, X_1, Y_0), unify join(T_0, l(H, l(H_0, T_0)), Y_1) and join(l(H_1, T), X_1, l(H_1, Y_0)) with the equalities T_0 = l(H_1, T), X_1 = l(H, l(H_0, T_0)) and Y_1 = l(H_1, Y_0), from which we derive T_0 = l(H_1, T), X_1 = l(H, l(H_0, l(H_1, T))), Y_1 = l(H_1, Y_0), X = l(H, l(H_0, l(H_1, T))) and Y = l(H, l(H_0, l(H_1, Y_0))), having to now solve the goal join(T, l(H, l(H_0, l(H_1, T))), Y_0).
  · ...

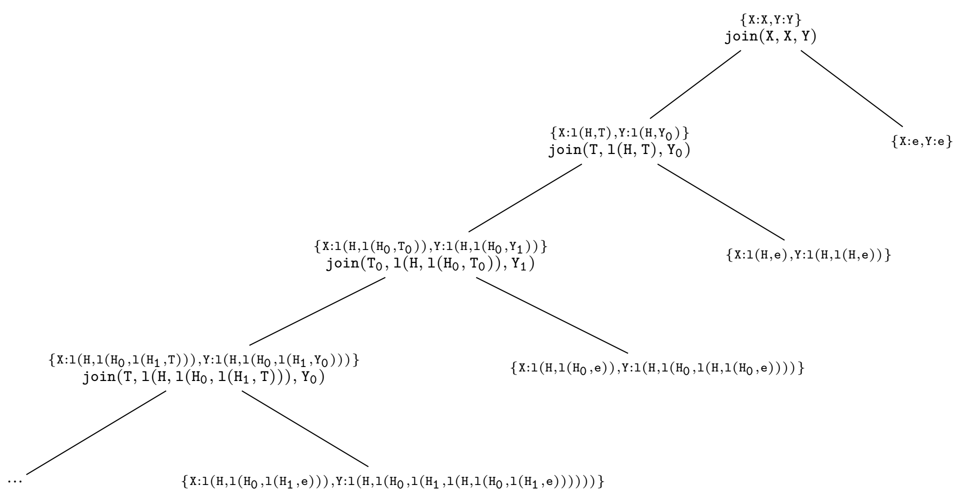
The search tree is infinite and looks as follows:



Observe that if the clauses that define the join function were listed in reverse order, that is, as

join(l(H, T), X, l(H, Y)) :- join(T, X, Y).
join(e, X, X).

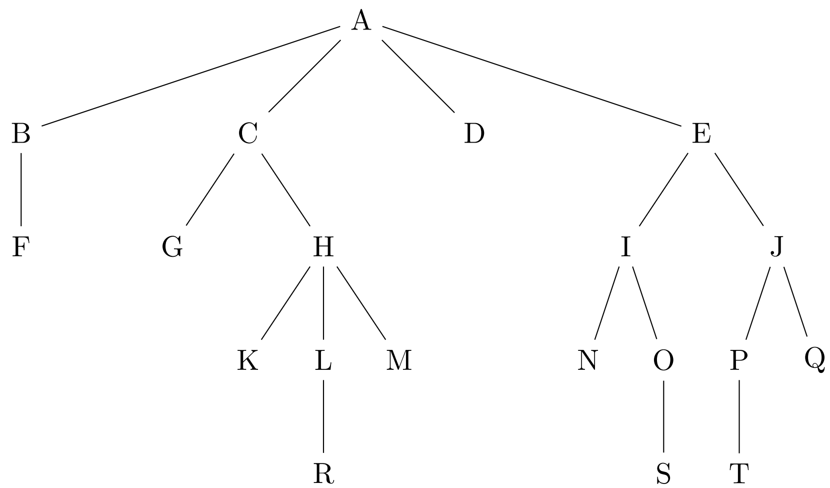
then the search tree would look as follows:



So solving goals relative to a given definite logic program amounts to exploring a tree of the kind illustrated above; in fact, it is discovering, building the tree, and reporting solutions when reaching a *leaf* associated with a solution. Two fundamental ways of exploring, or discovering, or building a tree are:

- **Depth-first**: the leftmost unexplored *branch* is explored all the way down, *backtracking* up to the first embranchment where new branches have still not been explored.
- **Breadth-first**: the *nodes* are visited *level* by level, from left to right on a given level.

To illustrate and experiment, let us consider the tree below:



The `defaultdict` class from the `collections` module offers an elegant way to represent a tree. The code in the following cell works as follows.

- `t = tree()` makes `t` denote a `defaultdict` object, say t.
- `t['A']['B']['F'] = None` executes as follows. An attempt is made to access the key `'A'` of t, which does not exist and is therefore created, with as value what `tree()`, returns, namely, a new `defaultdict` object, say t_1. So `t['A']` evaluates to t_1. An attempt is made to access the key `'B'` of t_1, which does not exist and is therefore created, with as value what `tree()`, returns, namely, a new `defaultdict` object, say t_2. So `t['A']['B']` evaluates to t_2. An attempt is made to access the key `'F'` of t_2, which does not exist and is therefore created, with as value what `tree()`, returns, namely, a new `defaultdict` object, say t_3. So `t['A']['B']['C']` evaluates to t_3 and is then changed to `None`.
- `t['A']['C']['G'] = None` executes as follows. An attempt is made to access the key `'A'` of t, which exists and with `t['A']` evaluating to t_1. An attempt is made to access the key `'C'` of t_1, which does not exist and is therefore created and becomes the second key of t_1, with as value what `tree()`, returns, namely, a new `defaultdict` object, say t_4. So `t['A']['C']` evaluates to t_4. An attempt is made to access the key `'G'` of t_4, which does not exist and is therefore created, with as value what `tree()`, returns, namely, a new `defaultdict` object, say t_5. So `t['A']['C']['G']` evaluates to t_5 and is then changed to `None`.
- …

The `pprint()` function from the `pprint` module makes it easier to see that `t` indeeds models the tree as intended:

[2]:
```
def tree():
    return defaultdict(tree)
```

```
t = tree()
t['A']['B']['F'] = None
t['A']['C']['G'] = None
t['A']['C']['H']['K'] = None
t['A']['C']['H']['L']['R'] = None
t['A']['C']['H']['M'] = None
t['A']['D'] = None
t['A']['E']['I']['N'] = None
t['A']['E']['I']['O']['S'] = None
t['A']['E']['J']['P']['T'] = None
t['A']['E']['J']['Q'] = None

pprint(t)
```

```
defaultdict(<function tree at 0x10df2c820>,
            {'A': defaultdict(<function tree at 0x10df2c820>,
                              {'B': defaultdict(<function tree at 0x10df2c820>,
                                                {'F': None}),
                               'C': defaultdict(<function tree at 0x10df2c820>,
                                                {'G': None,
                                                 'H': defaultdict(<function tree
at 0x10df2c820>,
                                                                  {'K': None,
                                                                   'L':
defaultdict(<function tree at 0x10df2c820>,
    {'R': None}),
                                                                   'M':
None})}),
                               'D': None,
                               'E': defaultdict(<function tree at 0x10df2c820>,
                                                {'I': defaultdict(<function tree
at 0x10df2c820>,
                                                                  {'N': None,
                                                                   'O':
defaultdict(<function tree at 0x10df2c820>,
    {'S': None})}),
                                                 'J': defaultdict(<function tree
at 0x10df2c820>,
                                                                  {'P':
defaultdict(<function tree at 0x10df2c820>,
    {'T': None}),
                                                                   'Q':
None})})})})
```

With this approach, a tree t is modeled as a dictionary with a unique key, namely, the label of t's **root** (with `t` above as example, `'A'`), with as associated value, a dictionary that has as many keys as t's root has **children**. That dictionary can be thought of as modeling a **forest**, namely, the collection of each **subtree** of t that has as root a child of t's root (with `t` above as example, the subtrees rooted at 'B', 'C', 'D' and 'E').

A recursive function makes it easy to explore a tree and list all nodes in a depth-first manner:

[3]:
```
def recursively_list_nodes_depth_first(t):
    if t is None:
        return
    for node in t:
        print(node, end=' ')
        recursively_list_nodes_depth_first(t[node])

recursively_list_nodes_depth_first(t)
```

```
A B F C G H K L R M D E I N O S J P T Q
```

It is easy to list the **paths** from the root of the tree in a depth-first manner, so output [A], [A, B], [A, B, F], [A, C], [A, C, G], [A, C, H]… rather than A, B, F, C, G, H…, with the help of a **stack**. It is easy to list either the nodes or the paths from the root of the tree in a breadth-first manner with the help of a **queue**. Stacks and queues are essentially lists with a limited set of methods:

- For stacks, elements can be added and removed at one end (as plates brought to and removed from the top of a stack of plates, the sequence being viewed vertically rather than horizontally, with the end where the action takes place at the top).
- For queues, elements can be added at one end and removed at the other end (as individuals queueing at a bus stop, joining the queue at its back and leaving it, boarding the bus, at its front).

  Python lists with their `append()` and `pop()` methods offer suitable implementations of stacks, as the time complexity of both operations is constant in amortised cost. On the other hand, Python lists do not offer an effective implementation of queues: removing the first element of a list and inserting an element at the beginning of a list both have time complexity that is linear in the length of the list. The `deque` class from the `collections` module combines the functionality of stacks and queues, as it has methods for adding and removing elements at boths end that all have constant time complexity (thanks to a data structure known as a **doubly linked list**). Let us first use a `dequeue` object as a stack:

[4]:
```
# Alternatively: stack = deque([])
stack = deque(); stack
stack.append(0); stack
stack.append(1); stack
stack.append(2); stack
stack.pop(); stack # Two outputs
stack.append(3); stack
stack.append(4); stack
stack.pop(); stack # Two outputs
stack.pop(); stack # Two outputs
stack.pop(); stack # Two outputs
```

[4]:
```
deque([])
```

[4]: deque([0])

[4]: deque([0, 1])

[4]: deque([0, 1, 2])

[4]: 2

[4]: deque([0, 1])

[4]: deque([0, 1, 3])

[4]: deque([0, 1, 3, 4])

[4]: 4

[4]: deque([0, 1, 3])

[4]: 3

[4]: deque([0, 1])

[4]: 1

[4]: deque([0])

We can let a `deque` object O model a queue in two ways, depending on how we match the ends of the queue with the ends of O. We can let the end of O correspond to the front of the queue:

[5]:
```
# Alternatively: queue = deque([])
queue = deque(); queue
queue.appendleft(0); queue
queue.appendleft(-1); queue
queue.appendleft(-2); queue
queue.pop(); queue # Two outputs
queue.appendleft(-3); queue
queue.appendleft(-4); queue
queue.pop(); queue # Two outputs
queue.pop(); queue # Two outputs
queue.pop(); queue # Two outputs
```

[5]: deque([])

[5]: deque([0])

[5]: deque([-1, 0])

[5]: deque([-2, -1, 0])

[5]: 0

[5]: deque([-2, -1])

[5]: deque([-3, -2, -1])

[5]: deque([-4, -3, -2, -1])

[5]: -1

[5]: deque([-4, -3, -2])

[5]: -2

[5]: deque([-4, -3])

[5]: -3

[5]: deque([-4])

Or we can let the end of the `deque` object correspond to the back of the queue:

[6]:
```
# Alternatively: queue = deque([])
queue = deque(); queue
queue.append(0); queue
queue.append(1); queue
queue.append(2); queue
queue.popleft(); queue # Two outputs
queue.append(3); queue
queue.append(4); queue
queue.popleft(); queue # Two outputs
queue.popleft(); queue # Two outputs
queue.popleft(); queue # Two outputs
```

[6]: deque([])

[6]: deque([0])

[6]: deque([0, 1])

[6]: deque([0, 1, 2])

[6]: 0

[6]: deque([1, 2])

[6]: deque([1, 2, 3])

[6]: deque([1, 2, 3, 4])

[6]: 1

[6]: deque([2, 3, 4])

[6]: 2

[6]: deque([3, 4])

[6]: 3

[6]: deque([4])

Rather than appending elements to the back of a queue one after the other, we can prefer to in one sweep move, extend the back of the queue with all elements to append. In case the end of a `deque` object corresponds to the back of the queue it models, then the `extend()` method expectedly does the job. But in case the beginning of a `deque` object corresponds to the back of the queue, then the `extendleft()` method appropriately does the job too. Note how in the following cell, intending to append -1, then -2, then -3 on the left hand side, -3 indeed becomes the leftmost element, with -2 to its right, and -1 to the right of -2:

[7]:
```
queue = deque([0])
queue.extend([1, 2, 3])
queue.extendleft([-1, -2, -3])
queue
```

[7]: deque([-3, -2, -1, 0, 1, 2, 3])

Writing a recursive function that explores a tree and lists all nodes in a depth-first manner was easy because behind the scene, a stack manages all recursive calls (more generally, a stack manages all function calls). Let us now perform that exploration without taking advantage of recursion; instead, let us explicitly use a stack. Considering the dictionary `t` that models the tree t as defined above, `iter(t)` creates an iterable that can yield all of `t`'s keys. Actually, `t` has only one key, namely, the label of t's root (`'A'`), which can be generated with `next(iter(t))`; let `root` denote it. Then `t[root]` is a dictionary that has as many keys as t's root has children (namely 4, labeled `'B'`, `'C'`, `'D'` and `'E'`). Since the keys of `t[root]` have been inserted into the dictionary starting with the label of the leftmost child of t's root (`'B'`), and proceeding from left to right all the way to the label of the rightmost child of t's root (`'E'`), `reversed(list(t[root]))` is an iterable that can yield those labels starting with the rightmost child of t's root and ending with the leftmost child of t's root (so in the order `'E'`, `'D'`, `'C'` and `'B'`). Adding to the top of a stack `(k, t[root][k])` with `k` yielded by `reversed(list(t[root]))`, we eventually get in the stack a pair of the form (l_1, f_1) where l_1 is the label (`'E'`) of the rightmost child of t's root and f_1 is the forest consisting of the trees rooted at that node (so the trees rooted at the nodes labeled `'I'` and `'J'`), and above in the stack a pair of the form (l_2, f_2) where l_2 is the label (`'D'`) of the second rightmost child of t's root and f_2 is an empty forest since that child has no child, and above in the stack a pair of the form (l_3, f_3) where l_3 is the label (`'C'`) of the third rightmost child of t's root and f_3 is the forest consisting of the trees rooted at that node (so the trees rooted at the nodes labeled `'G'` and `'H'`), and at the top of the stack a pair of the form (l_4, f_4) where l_4 is the label (`'B'`) of the leftmost child of t's root and f_4 is the forest consisting of the trees rooted at that node (so only one tree, rooted at the node labeled `'F'`). The last pair is indeed the pair that we want to pop first from the stack, since when exploring t in a depth-first manner, the nodes in f_4 are enumerated before the nodes in f_3, that are enumerated before the nodes in f_2, that are enumerated before the nodes in f_1. The next cell implements a function that explores a tree depth-first search and calls it with `t` passed as argument. The cell is followed with a cell that traces execution of that function call, replacing the forests stored in the stack with the roots of their trees:

[8]:
```python
def list_nodes_depth_first(t):
    root = next(iter(t))
    roots_and_forests = deque([(root, t[root])])
    while roots_and_forests:
        root, forest = roots_and_forests.pop()
        print(root, end=' ')
        if forest:
            roots_and_forests.extend((k, forest[k])
                                         for k in reversed(list(forest))
                                     )

list_nodes_depth_first(t)
```

```
A B F C G H K L R M D E I N O S J P T Q
```

[9]:
```python
root = next(iter(t))
roots_and_forests = deque([(root, t[root])])
print('Stack now (with trees changed to roots):\n      ',
      [(root, list(t[root]))]
     )
while roots_and_forests:
    root, forest = roots_and_forests.pop()
    print()
    if forest:
        print('Node output:', root, '\t\tRoots of forest to process:',
              list(forest)
             )
    else:
        print('Node output:', root, '\t\tEmpty forest')
    if forest:
        roots_and_forests.extend((k, forest[k])
                                     for k in reversed(list(forest))
                                 )
    print('Stack now (with trees changed to roots):\n      ',
          [(root, list(forest) if forest else None)
                 for (root, forest) in roots_and_forests
          ]
         )
```

```
Stack now (with trees changed to roots):
      [('A', ['B', 'C', 'D', 'E'])]

Node output: A          Roots of forest to process: ['B', 'C', 'D', 'E']
Stack now (with trees changed to roots):
      [('E', ['I', 'J']), ('D', None), ('C', ['G', 'H']), ('B', ['F'])]
```

```
Node output: B          Roots of forest to process: ['F']
Stack now (with trees changed to roots):
      [('E', ['I', 'J']), ('D', None), ('C', ['G', 'H']), ('F', None)]

Node output: F          Empty forest
Stack now (with trees changed to roots):
      [('E', ['I', 'J']), ('D', None), ('C', ['G', 'H'])]

Node output: C          Roots of forest to process: ['G', 'H']
Stack now (with trees changed to roots):
      [('E', ['I', 'J']), ('D', None), ('H', ['K', 'L', 'M']), ('G', None)]

Node output: G          Empty forest
Stack now (with trees changed to roots):
      [('E', ['I', 'J']), ('D', None), ('H', ['K', 'L', 'M'])]

Node output: H          Roots of forest to process: ['K', 'L', 'M']
Stack now (with trees changed to roots):
      [('E', ['I', 'J']), ('D', None), ('M', None), ('L', ['R']), ('K', None)]

Node output: K          Empty forest
Stack now (with trees changed to roots):
      [('E', ['I', 'J']), ('D', None), ('M', None), ('L', ['R'])]

Node output: L          Roots of forest to process: ['R']
Stack now (with trees changed to roots):
      [('E', ['I', 'J']), ('D', None), ('M', None), ('R', None)]

Node output: R          Empty forest
Stack now (with trees changed to roots):
      [('E', ['I', 'J']), ('D', None), ('M', None)]

Node output: M          Empty forest
Stack now (with trees changed to roots):
      [('E', ['I', 'J']), ('D', None)]

Node output: D          Empty forest
Stack now (with trees changed to roots):
      [('E', ['I', 'J'])]

Node output: E          Roots of forest to process: ['I', 'J']
Stack now (with trees changed to roots):
      [('J', ['P', 'Q']), ('I', ['N', 'O'])]

Node output: I          Roots of forest to process: ['N', 'O']
Stack now (with trees changed to roots):
      [('J', ['P', 'Q']), ('O', ['S']), ('N', None)]
```

```
Node output: N          Empty forest
Stack now (with trees changed to roots):
      [('J', ['P', 'Q']), ('O', ['S'])]

Node output: O          Roots of forest to process: ['S']
Stack now (with trees changed to roots):
      [('J', ['P', 'Q']), ('S', None)]

Node output: S          Empty forest
Stack now (with trees changed to roots):
      [('J', ['P', 'Q'])]

Node output: J          Roots of forest to process: ['P', 'Q']
Stack now (with trees changed to roots):
      [('Q', None), ('P', ['T'])]

Node output: P          Roots of forest to process: ['T']
Stack now (with trees changed to roots):
      [('Q', None), ('T', None)]

Node output: T          Empty forest
Stack now (with trees changed to roots):
      [('Q', None)]

Node output: Q          Empty forest
Stack now (with trees changed to roots):
      []
```

It is easy to modify the previous function to enumerate paths from the root of the tree rather than nodes: instead of storing nodes, we store paths, starting with the path that starts from and stops at the root of the tree (`['A']`) and creating extensions of a path p with each of the children of the last node in p, unless that node is a leaf:

[10]:
```
def list_paths_depth_first(t):
    root = next(iter(t))
    paths_and_forests = deque([([root], t[root])])
    while paths_and_forests:
        path, forest = paths_and_forests.pop()
        print(path)
        if forest:
            paths_and_forests.extend((path + [k], forest[k])
                                         for k in reversed(list(forest))
                                    )

list_paths_depth_first(t)
```

```
['A']
['A', 'B']
```

```
['A', 'B', 'F']
['A', 'C']
['A', 'C', 'G']
['A', 'C', 'H']
['A', 'C', 'H', 'K']
['A', 'C', 'H', 'L']
['A', 'C', 'H', 'L', 'R']
['A', 'C', 'H', 'M']
['A', 'D']
['A', 'E']
['A', 'E', 'I']
['A', 'E', 'I', 'N']
['A', 'E', 'I', 'O']
['A', 'E', 'I', 'O', 'S']
['A', 'E', 'J']
['A', 'E', 'J', 'P']
['A', 'E', 'J', 'P', 'T']
['A', 'E', 'J', 'Q']
```

To explore a tree in a breadth-first manner and generate either nodes or paths, it suffices to modify the previous two functions, using a queue rather than a stack. Also, the left to right ordering of children of a given node should not be reversed. More precisely, the comments for `list_nodes_depth_first()` can be modified as follows (with `t` still denoting the dictionary that models the tree t as defined above and `root` still denoting `next(iter(t))`, that is, the label of t's root (`'A'`)). Adding to the back of a queue `(k, t[root][k])` with `k` yielded by `iter(t[root])`, we eventually get in the queue a pair of the form (l_1, f_1) where l_1 is the label (`'B'`) of the leftmost child of t's root and f_1 is the forest consisting of the trees rooted at that node (so only one tree, rooted at the node labeled `'F'`), and further in the back of the queue a pair of the form (l_2, f_2) where l_2 is the label (`'C'`) of the second leftmost child of t's root and f_2 is the forest consisting of the trees rooted at that node (so the trees rooted at the nodes labeled `'G'` and `'H'`), and further in the back of the queue a pair of the form (l_3, f_3) where l_3 is the label (`'D'`) of the third leftmost child of t's root and f_3 is an empty forest since that child has no child, and at the very back of the queue a pair of the form (l_4, f_4) where l_4 is the label (`'E'`) of the rightmost child of t's root and f_4 is the forest consisting of the trees rooted at that node (so the trees rooted at the nodes labeled `'I'` and `'J'`). The first pair is indeed the pair that we want to come to the front of the queue and be removed before all others, since when exploring t in a breadth-first manner, the nodes on a given level should be enumerated from left to right. The function `list_nodes_depth_first()` is modified into the function `list_nodes_breadth_first()` in the next cell, in which the function is then called with `t` passed as argument. The cell is followed with a cell that traces execution of that function call, replacing the forests stored in the queue with the roots of their trees:

[11]:
```
def list_nodes_breadth_first(t):
    root = next(iter(t))
    roots_and_forests = deque([(root, t[root])])
    while roots_and_forests:
        root, forest = roots_and_forests.pop()
        print(root, end=' ')
        if forest:
```

```
            roots_and_forests.extendleft(forest.items())

list_nodes_breadth_first(t)
```

```
A B C D E F G H I J K L M N O P Q R S T
```

[12]:
```
root = next(iter(t))
roots_and_forests = deque([(root, t[root])])
print('Queue now (with trees changed to roots):\n      ',
      [(root, list(t[root]))]
     )
while roots_and_forests:
    root, forest = roots_and_forests.pop()
    print()
    if forest:
        print('Node output:', root, '\t\tRoots of forest to process:',
              list(forest)
             )
    else:
        print('Node output:', root, '\t\tEmpty forest')
    if forest:
        roots_and_forests.extendleft(forest.items())
    print('Queue now (with trees changed to roots):\n      ',
          [(root, list(forest) if forest else None)
                for (root, forest) in roots_and_forests
          ]
         )
```

```
Queue now (with trees changed to roots):
      [('A', ['B', 'C', 'D', 'E'])]

Node output: A          Roots of forest to process: ['B', 'C', 'D', 'E']
Queue now (with trees changed to roots):
      [('E', ['I', 'J']), ('D', None), ('C', ['G', 'H']), ('B', ['F'])]

Node output: B          Roots of forest to process: ['F']
Queue now (with trees changed to roots):
      [('F', None), ('E', ['I', 'J']), ('D', None), ('C', ['G', 'H'])]

Node output: C          Roots of forest to process: ['G', 'H']
Queue now (with trees changed to roots):
      [('H', ['K', 'L', 'M']), ('G', None), ('F', None), ('E', ['I', 'J']),
('D', None)]

Node output: D          Empty forest
Queue now (with trees changed to roots):
      [('H', ['K', 'L', 'M']), ('G', None), ('F', None), ('E', ['I', 'J'])]
```

```
Node output: E          Roots of forest to process: ['I', 'J']
Queue now (with trees changed to roots):
      [('J', ['P', 'Q']), ('I', ['N', 'O']), ('H', ['K', 'L', 'M']), ('G',
None), ('F', None)]

Node output: F          Empty forest
Queue now (with trees changed to roots):
      [('J', ['P', 'Q']), ('I', ['N', 'O']), ('H', ['K', 'L', 'M']), ('G',
None)]

Node output: G          Empty forest
Queue now (with trees changed to roots):
      [('J', ['P', 'Q']), ('I', ['N', 'O']), ('H', ['K', 'L', 'M'])]

Node output: H          Roots of forest to process: ['K', 'L', 'M']
Queue now (with trees changed to roots):
      [('M', None), ('L', ['R']), ('K', None), ('J', ['P', 'Q']), ('I', ['N',
'O'])]

Node output: I          Roots of forest to process: ['N', 'O']
Queue now (with trees changed to roots):
      [('O', ['S']), ('N', None), ('M', None), ('L', ['R']), ('K', None), ('J',
['P', 'Q'])]

Node output: J          Roots of forest to process: ['P', 'Q']
Queue now (with trees changed to roots):
      [('Q', None), ('P', ['T']), ('O', ['S']), ('N', None), ('M', None), ('L',
['R']), ('K', None)]

Node output: K          Empty forest
Queue now (with trees changed to roots):
      [('Q', None), ('P', ['T']), ('O', ['S']), ('N', None), ('M', None), ('L',
['R'])]

Node output: L          Roots of forest to process: ['R']
Queue now (with trees changed to roots):
      [('R', None), ('Q', None), ('P', ['T']), ('O', ['S']), ('N', None), ('M',
None)]

Node output: M          Empty forest
Queue now (with trees changed to roots):
      [('R', None), ('Q', None), ('P', ['T']), ('O', ['S']), ('N', None)]

Node output: N          Empty forest
Queue now (with trees changed to roots):
      [('R', None), ('Q', None), ('P', ['T']), ('O', ['S'])]

Node output: O          Roots of forest to process: ['S']
```

```
Queue now (with trees changed to roots):
      [('S', None), ('R', None), ('Q', None), ('P', ['T'])]

Node output: P           Roots of forest to process: ['T']
Queue now (with trees changed to roots):
      [('T', None), ('S', None), ('R', None), ('Q', None)]

Node output: Q           Empty forest
Queue now (with trees changed to roots):
      [('T', None), ('S', None), ('R', None)]

Node output: R           Empty forest
Queue now (with trees changed to roots):
      [('T', None), ('S', None)]

Node output: S           Empty forest
Queue now (with trees changed to roots):
      [('T', None)]

Node output: T           Empty forest
Queue now (with trees changed to roots):
      []
```

`list_paths_depth_first()` is modified into `list_paths_breadth_first()` as follows:

[13]:
```python
def list_paths_breadth_first(t):
    root = next(iter(t))
    paths_and_forests = deque([([root], t[root])])
    while paths_and_forests:
        path, forest = paths_and_forests.pop()
        print(path)
        if forest:
            paths_and_forests.extendleft((path + [k], forest[k])
                                                  for k in forest
                                         )

list_paths_breadth_first(t)
```

```
['A']
['A', 'B']
['A', 'C']
['A', 'D']
['A', 'E']
['A', 'B', 'F']
['A', 'C', 'G']
['A', 'C', 'H']
['A', 'E', 'I']
['A', 'E', 'J']
['A', 'C', 'H', 'K']
```

```
['A', 'C', 'H', 'L']
['A', 'C', 'H', 'M']
['A', 'E', 'I', 'N']
['A', 'E', 'I', 'O']
['A', 'E', 'J', 'P']
['A', 'E', 'J', 'Q']
['A', 'C', 'H', 'L', 'R']
['A', 'E', 'I', 'O', 'S']
['A', 'E', 'J', 'P', 'T']
```

Getting back to both trees for the goal grandparent(john, X):

- Exploring the first tree in a depth-first manner yields both solutions fastest, after which the rest of the exploration is "for nothing".
- Exploring the second tree in a depth-first manner yields both solutions at the very end.
- Exploring the first and second trees in a breadth-first manner yields both solutions at the very end.

Getting back to both trees for the goal join(X, X, Y):

- Exploring the first tree in a depth-first or breadth-first manner makes no difference; the solutions are generated one by one, and the exploration would have to be interrupted at some point as it could go on forever and produce infinitely many solutions.
- Exploring the second tree in a breadth-first manner is hardly different to exploring the first tree in a breadth-first manner, one node being visited before rather than after the production of a given solution.
- Exploring the second tree in a depth-first manner traps the search in an infinite descent along the leftmost branch, with no solution being ever produced.

Solving goals relative to a given definite logic program by a breadth-first exploration of the associated search tree is a **complete** proof procedure: every solution is eventually produced. This is an immediate consequence of the fact that such a tree is **finitely branching**. On the other hand, as we have just observed, a depth-first exploration does not offer a complete proof procedure. Usually, Prolog's proof engine implements a depth-first search, hence an incomplete proof procedure: it expects users to properly order the rules that make up the program, and to properly order in a rule the atoms that make up its body, so that the search trees associated with goals of interest have "a good shape" and make depth-first search a most efficient procedure. We will write a Prolog interpreter where we can chose to explore a search tree either in a depth-first or in a breadth-first manner (the difference is minor and essentially relies on using either a stack or a queue in pretty much the same way, as we have previously observed). The exploration of a search tree is the last part of the work the interpreter has to do. First, we need to be able to parse the rules that make up a definite logic program, as well as perform a number of operations such as separate head and body from a rule, identify which variables occur in an atom, perform substitution of variables by terms in an atom, etc. We first define a helper function to consistently extend a dictionary with a sequence of key-value pairs, making use of the `setdefault` method of the `dict` class:

[14]:
```
D = {}

D.setdefault('A'), D
D.setdefault('A', 1), D
```

```
D.setdefault('B', 1), D
D.setdefault('B'), D
```

[14]: (None, {'A': None})

[14]: (None, {'A': None})

[14]: (1, {'A': None, 'B': 1})

[14]: (1, {'A': None, 'B': 1})

[15]:
```
def consistently_add_to(mapping, *key_values):
    for (key, value) in key_values:
        if mapping.setdefault(key, value) != value:
            return False
    return True

D = {}

consistently_add_to(D, ('A', 1)), D
consistently_add_to(D, ('A', 1)), D
consistently_add_to(D, ('A', 2)), D
consistently_add_to(D, ('B', 2), ('C', 3)), D
consistently_add_to(D, ('A', 1), ('C', 3), ('D', 4)), D
consistently_add_to(D, ('B', 3), ('E', 5)), D
```

[15]: (True, {'A': 1})

[15]: (True, {'A': 1})

[15]: (False, {'A': 1})

[15]: (True, {'A': 1, 'B': 2, 'C': 3})

[15]: (True, {'A': 1, 'B': 2, 'C': 3, 'D': 4})

[15]: (False, {'A': 1, 'B': 2, 'C': 3, 'D': 4})

We define three classes, `Expression`, `Term` and `Atom`, with `Term` and `Atom` inheriting from `Expression`, considering that terms and atoms are two kinds of expressions.

Generalising on the examples previously examined:

- an atom is built from an n-ary predicate symbol ($n \in \mathbf{N}$) and n terms;
- a term is a variable or is built from an n-ary function symbol ($n \in \mathbf{N}$) and n terms.

So the outermost symbol in an atom is a predicate symbol while all other symbols are function symbols or variables; all symbols in a term are function symbols or variables. Prolog function symbols, predicate symbols and variables are built from alphanumeric characters and underscores, with function and predicate symbols starting with lowercase letters, and variables starting with uppercase letters or underscores.

Trees are most appropriate to represent expressions. To give `Expression` objects the structure of trees that represent expressions, we use two attributes, `root` and `children`. Given an object O of classs `Expression` meant to represent an expression E:

- the value of `root` for O will be set to the string that denotes E's outermost symbol s (a predicate symbol if E is an atom, a function symbol if E is a term);
- if n is s's arity, the value of `children` for O will be set to a list of length n consisting of n `Expression` objects, that represent the n terms that are the arguments to s in E, listed from left to right (so the list is empty in case E is a nullary predicate symbol, a constant, or a variable).

`Expression` objects will be created by parsing strings, giving atoms and terms a tree structure. We implement `__str__()` in `Expression` for what can be seen as the reverse operation: get the (properly formatted) string from the representing tree. When testing `__str__()` below, we define `Expression` objects "by hand"; parsing methods will be defined later.

[16]:
```
class Expression:
    def __init__(self, root, children=[]):
        self.root = root
        self.children = children

    def __str__(self):
        return self.root if not self.children\
                         else ''.join((self.root, '(',
                                       ', '.join(str(child)
                                                 for child in self.children
                                                ), ')'
                                       )
                                      )


class Term(Expression):
    class TermError(Exception):
        pass


class Atom(Expression):
    class AtomError(Exception):
        pass
```

[17]:
```
# bob: constant
print(Term('bob'))
# l: binary function symbol
# H, T: variables
print(Term('l', [Term('H'), Term('T')]))
# l: binary function symbol
```

```
# e: constant
# H, T: variables
print(Term('l', [Term('e'), Term('l', [Term('H'), Term('T')])]))

print()

# on: nullary predicate symbol
print(Atom('on'))
# happy: unary predicate symbol
# john: constant
print(Atom('happy', [Term('john')]))
# mother: binary predicate symbol
# jane, sandra: constants
print(Atom('mother', [Term('jane'), Term('sandra')]))
# join: binary predicate symbol
# l: binary function symbol
# H, T, X, Y: variables
print(Atom('join', [Term('l', [Term('H'), Term('T')]), Term('X'),
                    Term('l', [Term('H'), Term('Y')])
                   ]
          )
     )
```

```
bob
l(H, T)
l(e, l(H, T))

on
happy(john)
mother(jane, sandra)
join(l(H, T), X, l(H, Y))
```

When we create a `Term` or an `Atom` object, we collect all its function symbols in the form of a dictionary, with symbols as keys and symbols' arities as values. The core of the work is performed by `consistently_add_to()`, with the help of an `Expression` method, `is_variable()`. The code checks that no function symbol is used with different arities in a term, and that the predicate and function symbols in an atom are different; that part of the code will be tested when testing the functions that parse strings and create `Term` or `Atom` objects.

[18]:
```
class Expression:
    def is_variable(self):
        return self.root[0].isupper() or self.root[0] == '_'


class Term(Expression, Term):
    def __init__(self, root, children=[]):
        super().__init__(root, children)
        self.function_symbols = {} if self.is_variable()\
```

```
                                    else {root: len(self.children)}
        for child in self.children:
            if not consistently_add_to(self.function_symbols,
                                       *child.function_symbols.items()
                                      ):
                raise Term.TermError('Function symbol used with different '
                                     f'arities in {self}'
                                    )


class Atom(Expression, Atom):
    def __init__(self, root, children=[]):
        super().__init__(root, children)
        self.function_symbols = {}
        for child in self.children:
            if not consistently_add_to(self.function_symbols,
                                       *child.function_symbols.items()
                                      ):
                raise Term.TermError('Function symbol used with different '
                                     f'arities in {self}'
                                    )
        if self.root in self.function_symbols:
            raise Atom.AtomError('Predicate symbol also function symbol '
                                 f'in {self}'
                                )
```

[19]:
```
Term('bob').function_symbols
Term('l', [Term('H'), Term('T')]).function_symbols
Term('l', [Term('e'), Term('l', [Term('H'), Term('T')])]).function_symbols
Atom('on').function_symbols
Atom('happy', [Term('john')]).function_symbols
Atom('mother', [Term('jane'), Term('sandra')]).function_symbols
Atom('join', [Term('l', [Term('H'), Term('T')]), Term('X'),
              Term('l', [Term('H'), Term('Y')])
             ]
    ).function_symbols
```

[19]: {'bob': 0}

[19]: {'l': 2}

[19]: {'l': 2, 'e': 0}

[19]: {}

[19]: {'john': 0}

[19]: {'jane': 0, 'sandra': 0}

[19]: {'l': 2}

It is reasonable to embed the code that parses a string and creates an `Expression` object, and more specifically a `Term` or an `Atom` object, within the `Expression`, `Term` or `Atom` class. But such a piece of code does not naturally take the form of one of the classes' methods. Indeed, a method of any those classes is meant to operate on an object of the class, already created, with its `root` and `children` attributes, whereas the purpose of the code under discussion is precisely to create such an object. A natural design is to define a **static method** or a **class method**. Such a method can be called from an object of the class where it is defined, but it can also be called from the class itself. The easiest way to defined such methods is to use the `staticmethod` and `classmethod` decorators:

[20]:
```
class C:
    @staticmethod
    def a_static_method(*arguments):
        print(arguments)

    @classmethod
    def a_class_method(*arguments):
        print(arguments)

C.a_static_method('Class is not an argument', 'of a static method')
C().a_static_method('Class is not an argument', 'of a static method')

print()

C.a_class_method('Class is first argument', 'of a class method')
C().a_class_method('Class is first argument', 'of a class method')
```

```
('Class is not an argument', 'of a static method')
('Class is not an argument', 'of a static method')

(<class '__main__.C'>, 'Class is first argument', 'of a class method')
(<class '__main__.C'>, 'Class is first argument', 'of a class method')
```

To parse an atom or a term represented as a string, we will first get rid of all spaces in the string, if any, and convert the resulting string to a list L of characters, from last character in the string to first character in the string, so that characters can be efficiently consumed by popping them off the end of the list, as opposed to removing them from the beginning of the list. Opening and closing parentheses and commas will need special processing. The rest is predicate and function symbols and variables, which will be dealt with thanks to a function, `parse_word()`. This function receives as argument what remains of L, assumed to be at the stage where a predicate or function symbol or a variable is to be parsed; so L then ends in the characters that make up that predicate or function symbol or variable in reverse order; `parse_word()` consumes those characters from L and returns the predicate or function symbol or the variable as a string. The function can be defined as a static method in the `Expression` class:

[21]:
```python
class Expression(Expression):
    @staticmethod
    def parse_word(characters):
        word = [characters.pop()]
        while characters and (characters[-1].isalnum()
                              or characters[-1] == '_'
                             ):
            word.append(characters.pop())
        return ''.join(word)
```

[22]:
```python
characters = list(reversed('bob'.replace(' ', '')))
characters
Expression.parse_word(characters)
characters

print()

characters = list(reversed('happy(   john   )'.replace(' ', '')))
characters
Expression.parse_word(characters)
characters

print()

# Could be what remains to be parsed in "join(l(H, T), X, l(H, Y))"
# after "join(l(H, T), X, l(" has been parsed already.
characters = list(reversed('H,   Y))'.replace(' ', '')))
characters
Expression.parse_word(characters)
characters
```

[22]: ['b', 'o', 'b']

[22]: 'bob'

[22]: []

[22]: [')', 'n', 'h', 'o', 'j', '(', 'y', 'p', 'p', 'a', 'h']

[22]: 'happy'

[22]: [')', 'n', 'h', 'o', 'j', '(']

[22]: [')', ')', 'Y', ',', 'H']

[22]: 'H'

[22]: [')', ')', 'Y', ',']

Let an atom or a term E be given and let L be the list of all nonspace characters in E in reverse order. We will define in `Expression` two functions, `parse_subitem()` and `parse_subitem_sequence()`, meant to operate in a way that we now describe. To parse E (possibly as a subexpression of a larger expression), the function `parse_subitem()` will be called with L passed as argument. It will first check that E indeed starts with (that is, L indeed ends in) a character that can be the beginning of a predicate or function symbol or a variable and call `parse_word()`, passing L as argument. If E is a nullary predicate or function symbol or a variable, that predicate or function symbol or variable will be returned by `parse_word()` and L will have become empty. Otherwise, E is of the form $\sigma(t_1, \dots, t_n)$ for some nonzero $n \in \mathbf{N}$, predicate or function symbol σ, and terms t_1, ..., t_n. Then `parse_word()` will return σ, having consumed all characters that make up σ; `parse_subitem()`, finding out that L is not empty, will then check that L indeeds ends in `(`, consume that character (that it, pop `(` off the end of L), and call `parse_subitem_sequence()`, passing L as argument, whose purpose is to parse $t_1, \dots, t_n$ and return a list with as members, the n `Term` objects o_1, ..., o_n that represent t_1, ..., t_n. At this stage, all characters occurring in t_1, ..., t_n and the separating commas will have been consumed, and the only task left for `parse_subitem()` to complete will be to check that `)` is now the last symbol in L (it should also be the only symbol in L in case E is the whole expression to parse, but that will not be up to `parse_subitem()` to check: `parse_subitem()` does not know whether it parses a whole expression or a subexpression of a larger expression), consume it, and create an `Atom` or a `Term` object whose `root` attribute should be set to σ and whose `children` attribute should be set to the list $[o_1, \dots, o_n]$. Recall that `parse_subitem_sequence()` will have to parse $t_1, \dots, t_n$. It will do so with a first call to `parse_subitem()` to create a `Term` object o_1 that represents t_1. It will then find out that L now ends in a comma, consume it, and make a second call to `parse_subitem()` to create a `Term` object o_2 that represents t_2. Eventually, it will make a last call to `parse_subitem()` to create a `Term` object o_n that represents t_n, find out that L does not end in a comma, assume that the whole sequence has been successfully parsed, and return $[o_1, \dots, o_n]$ to its caller (the original `parse_subitem()` call).

Before we implement `parse_subitem()` and `parse_subitem_sequence()`, we define skeleton functions to illustrate how `parse_subitem()` and `parse_subitem_sequence()` are meant to call each other, and how characters are consumed:

[23]:
```
def parse_subitem_skeleton(characters, depth=0):
    print('  ' * depth, 'Start parsing subitem, characters left:',
          ''.join(reversed(characters))
         )
    Expression.parse_word(characters)
    if not characters or characters[-1] != '(':
        print('  ' * depth, 'End parsing subitem, characters left:',
              ''.join(reversed(characters))
             )
        return
    # Popping (
    characters.pop()
    parse_subitem_sequence_skeleton(characters, depth + 1)
```

```python
    # Popping )
    characters.pop()
    print('  ' * depth, 'End parsing subitem, characters left:',
          ''.join(reversed(characters))
         )

def parse_subitem_sequence_skeleton(characters, depth):
    print('  ' * depth, 'Start parsing subitem sequence, characters left:',
          ''.join(reversed(characters))
         )
    expressions = []
    while True:
        expressions.append(parse_subitem_skeleton(characters, depth + 1))
        if characters[-1] != ',':
            print('  ' * depth,
                  'End parsing subitem sequence, characters left:',
                  ''.join(reversed(characters))
                 )
            return
        # Popping ,
        characters.pop()

parse_subitem_skeleton(list(reversed('bob')))
print()
parse_subitem_skeleton(list(reversed('happy(john)')))
print()
parse_subitem_skeleton(list(reversed('l(e,l(H,T))')))
```

```
Start parsing subitem, characters left: bob
End parsing subitem, characters left:

Start parsing subitem, characters left: happy(john)
  Start parsing subitem sequence, characters left: john)
    Start parsing subitem, characters left: john)
    End parsing subitem, characters left: )
  End parsing subitem sequence, characters left: )
End parsing subitem, characters left:

Start parsing subitem, characters left: l(e,l(H,T))
  Start parsing subitem sequence, characters left: e,l(H,T))
    Start parsing subitem, characters left: e,l(H,T))
    End parsing subitem, characters left: ,l(H,T))
    Start parsing subitem, characters left: l(H,T))
      Start parsing subitem sequence, characters left: H,T))
        Start parsing subitem, characters left: H,T))
        End parsing subitem, characters left: ,T))
        Start parsing subitem, characters left: T))
        End parsing subitem, characters left: ))
```

```
      End parsing subitem sequence, characters left: ))
    End parsing subitem, characters left: )
  End parsing subitem sequence, characters left: )
End parsing subitem, characters left:
```

Now for the implementation of `parse_subitem()` and `parse_subitem_sequence()`.

Both `parse_subitem()` and `parse_subitem_sequence()` are defined as class methods, with a first parameter, `_class`, whose value will be provided by the class from which the method is called. So in the case of `parse_subitem()`, the first character of the string to parse can be properly valided (it should be a lowercase letter when parsing an atom; it should be a letter that can be either lowercase or uppercase, or an underscore, when parsing a term), and eventually, the right type of object can be created. At this stage of the discussion, `parse_subitem()` seems to be useful to parse either atoms or terms, whereas `parse_subitem_sequence()` seems to be useful only to parse sequences of terms, and indeed the calls to `parse_subitem_sequence()` by `parse_subitem()` will actually be calls to `Term.parse_subitem_sequence()`. But to parse the bodies of the rules of a logic program, we will have to parse sequence of atoms; `parse_subitem_sequence()` will be perfectly suitable for the task, with the `_class` parameter of `parse_subitem_sequence()` set to `Atom`.

It was tacitly assumed that `parse_subitem_skeleton()` and `parse_subitem_sequence_skeleton()` would be used only to parse a syntactically correct expression; `parse_subitem()` and `parse_subitem_sequence()` make no such assumption, so also check for syntactic correctness, and return `None` whenever they find out that there are characters that cannot be for a syntactically correct expression.

- Before it calls `parse_word()`, `parse_subitem()` checks that there is indeed at least one character to parse, and that the first character to parse is as described in the previous item; otherwise, `parse_subitem()` returns `None`. The check is achieved thanks to one of two static methods, both with the name `starts_well()`, one being part of the `Term` class, the other one being part of the `Atom` class. The value of the `_class` parameter of `parse_subitem()` allows one to call the appropriate method.
- In case `parse_subitem()` does not process a nullary predicate or function symbol or a variable, because it finds an opening parenthesis and calls `parse_subitem_sequence()`, and provided that the latter successfully parses a sequence of expressions and does not return `None` (otherwise, `parse_subitem()` should itself return `None`), it needs to finally check that there are some characters left, with as first character to parse, a closing parenthesis.
- As for `parse_subitem_sequence()`, provided that all calls to `parse_subitem()` successfully parse an expression and do not return `None` (otherwise, `parse_subitem_sequence()` should itself return `None`), it has to check whether there is at least one character left. If that character is a comma, then the character can be popped and further expressions still have to be parsed in the sequence being currently dealt with. Otherwise, `parse_subitem_sequence()` returns the sequence of expressions it has successfully parsed and leaves it to `parse_subitem()` to act appropriately, whether or not there is at least one character left.

[24]:
```python
class Expression(Expression):
    @classmethod
    def parse_subitem(_class, characters):
        if not characters or not _class.starts_well(characters[-1]):
            return
```

```
        symbol = Expression.parse_word(characters)
        if not characters or characters[-1] != '(':
            return _class(symbol)
        characters.pop()
        subitems = Term.parse_subitem_sequence(characters)
        if not subitems or not characters or characters.pop() != ')':
            return
        return _class(symbol, subitems)

    @classmethod
    def parse_subitem_sequence(_class, characters):
        items = []
        while True:
            item = _class.parse_subitem(characters)
            if not item:
                return
            items.append(item)
            if not characters or characters[-1] != ',':
                return items
            characters.pop()


class Term(Term, Expression):
    @staticmethod
    def starts_well(c):
        return c.isidentifier()


class Atom(Atom, Expression):
    @staticmethod
    def starts_well(c):
        return c.islower()
```

[25]:

```
expression = Term.parse_subitem(list(reversed('bob')))
print(type(expression), expression)

expression = Term.parse_subitem(list(reversed('l(e,l(H,T))')))
print(type(expression), expression)

expression = Atom.parse_subitem(list(reversed('father(bob,sandra)')))
print(type(expression), expression)

expression = Atom.parse_subitem(list(reversed('join(l(H,T),X,l(H,Y))')))
print(type(expression), expression)
```

```
<class '__main__.Term'> bob
<class '__main__.Term'> l(e, l(H, T))
<class '__main__.Atom'> father(bob, sandra)
```

```
<class '__main__.Atom'> join(l(H, T), X, l(H, Y))
```

We extend the `Expression`, `Term` and `Atom` classes with three functions, `parse_item()`, `parse_term()` and `parse_atom()`, respectively. Whereas `parse_term()` and `parse_atom()` are defined as static methods, `parse_item()` is defined as a class method. Calling `parse_term()` and `parse_atom()` as `Term.parse_term()` and `Atom.parse_atom()`, respectively, lets `parse_item()` be aware of the type of object it is requested to create, as its first argument. Essentially, `parse_item()` creates a list of nonspace characters from the string to parse, reversing the order of its characters, and then calls `parse_subitem()`, either as `Term.parse_subitem()` or as `Atom.parse_subitem()` depending on how `parse_item()` has been called.

We provide `parse_item()` with a third parameter, namely, `parse_subitem`, with a default value of `True`. This is because we will later have to parse rules, so rule bodies, which are sequences of atoms. As already mentioned, in order to parse a rule body, a call to `parse_item_sequence()` (passing `Atom` as value for its `_class` argument) will do the job; setting `parse_subitem` to `False` will let `parse_item()` know that it needs to call `parse_item_sequence()` rather than `parse_subitem()`.

Finally, `parse_term()` and `parse_atom()` take advantage via `parse_item()` of the syntactic checks performed by `parse_subitem()` and `parse_item_sequence()`, as these functions return `None` when one of the checks they perform fails. Even when `parse_subitem()` and `parse_item_sequence()` do not return `None`, `parse_term()` and `parse_atom()` can fail to return a `Term` or an `Atom` object, respectively, when a last round of syntactic checks by `Term`'s or `Atom`'s `__init__()` method results in a `TermError` or an `AtomError` error being raised, respectively.

[26]:
```
class Expression(Expression):
    @classmethod
    def parse_item(_class, expression, parse_subitem=True):
        characters = list(reversed(expression.replace(' ', '')))
        item = _class.parse_subitem(characters)\
                    if parse_subitem\
                    else Atom.parse_subitem_sequence(characters)
        if not item or characters:
            return
        return item


class Term(Term, Expression):
    @staticmethod
    def parse_term(expression):
        term = Term.parse_item(expression)
        if not term:
            raise Term.TermError(f'{expression} is syntactically incorrect')
        return term


class Atom(Atom, Expression):
    @staticmethod
    def parse_atom(expression):
        atom = Atom.parse_item(expression)
```

```
        if not atom:
            raise Atom.AtomError(f'{expression} is syntactically incorrect')
        return atom
```

[27]:
```
term = Term.parse_term('X')
print(type(term), term)

term = Term.parse_term('bob')
print(type(term), term)

term = Term.parse_term('l(e, l(H, T))')
print(type(term), term)

atom = Atom.parse_atom('father(bob, sandra)')
print(type(atom), atom)

atom = Atom.parse_atom('join(l(H, T), X, l(H, Y))')
print(type(atom), atom)
```

```
<class '__main__.Term'> X
<class '__main__.Term'> bob
<class '__main__.Term'> l(e, l(H, T))
<class '__main__.Atom'> father(bob, sandra)
<class '__main__.Atom'> join(l(H, T), X, l(H, Y))
```

[28]:
```
Term.parse_term('2')
```

```
---------------------------------------------------------------------------
TermError                                 Traceback (most recent call last)
<ipython-input-28-d6638d31b6d8> in <module>
----> 1 Term.parse_term('2')

<ipython-input-26-0d4a07671498> in parse_term(expression)
     16         term = Term.parse_item(expression)
     17         if not term:
---> 18             raise Term.TermError(f'{expression} is syntactically␣
 ↪incorrect')
     19         return term
     20

TermError: 2 is syntactically incorrect
```

[29]:
```
Term.parse_term('father(bob, sandra))')
```

```
---------------------------------------------------------------------------
TermError                                 Traceback (most recent call last)
<ipython-input-29-286d6c9dd374> in <module>
```

```
----> 1 Term.parse_term('father(bob, sandra))')

<ipython-input-26-0d4a07671498> in parse_term(expression)
     16         term = Term.parse_item(expression)
     17         if not term:
---> 18             raise Term.TermError(f'{expression} is syntactically␣
 ↪incorrect')
     19         return term
     20

TermError: father(bob, sandra)) is syntactically incorrect
```

[30]:
```
Term.parse_term('father(bob, , sandra)')
```

```
---------------------------------------------------------------------------
TermError                                 Traceback (most recent call last)
<ipython-input-30-23229f6b2bb6> in <module>
----> 1 Term.parse_term('father(bob, , sandra)')

<ipython-input-26-0d4a07671498> in parse_term(expression)
     16         term = Term.parse_item(expression)
     17         if not term:
---> 18             raise Term.TermError(f'{expression} is syntactically␣
 ↪incorrect')
     19         return term
     20

TermError: father(bob, , sandra) is syntactically incorrect
```

[31]:
```
Term.parse_term('father(bob, father(sandra, john, mary))')
```

```
---------------------------------------------------------------------------
TermError                                 Traceback (most recent call last)
<ipython-input-31-b43b8879a9d8> in <module>
----> 1 Term.parse_term('father(bob, father(sandra, john, mary))')

<ipython-input-26-0d4a07671498> in parse_term(expression)
     14     @staticmethod
     15     def parse_term(expression):
---> 16         term = Term.parse_item(expression)
     17         if not term:
     18             raise Term.TermError(f'{expression} is syntactically␣
 ↪incorrect')

<ipython-input-26-0d4a07671498> in parse_item(_class, expression, parse_subitem
```

```
      3     def parse_item(_class, expression, parse_subitem=True):
      4         characters = list(reversed(expression.replace(' ', '')))
----> 5         item = _class.parse_subitem(characters)\
      6                    if parse_subitem\
      7                    else Atom.parse_subitem_sequence(characters)

<ipython-input-24-20bcf86e40da> in parse_subitem(_class, characters)
     11         if not subitems or not characters or characters.pop() != ')':
     12             return
---> 13         return _class(symbol, subitems)
     14 
     15     @classmethod

<ipython-input-18-f566ace642a5> in __init__(self, root, children)
     13                                         *child.function_symbols.items()
     14                                        ):
---> 15                 raise Term.TermError('Function symbol used with␣
 ↪different '
     16                                       f'arities in {self}'
     17                                      )

TermError: Function symbol used with different arities in father(bob,␣
 ↪father(sandra, john, mary))
```

[32]:

```
Atom.parse_atom('X')
```

```
---------------------------------------------------------------------------
AtomError                                 Traceback (most recent call last)
<ipython-input-32-416faea9531c> in <module>
----> 1 Atom.parse_atom('X')

<ipython-input-26-0d4a07671498> in parse_atom(expression)
     25         atom = Atom.parse_item(expression)
     26         if not atom:
---> 27             raise Atom.AtomError(f'{expression} is syntactically␣
 ↪incorrect')
     28         return atom

AtomError: X is syntactically incorrect
```

[33]:

```
Atom.parse_atom('l(e, l(H, T))')
```

```
---------------------------------------------------------------------------
AtomError                                 Traceback (most recent call last)
<ipython-input-33-5fa32552c8f4> in <module>
```

```
----> 1 Atom.parse_atom('l(e, l(H, T))')

<ipython-input-26-0d4a07671498> in parse_atom(expression)
     23     @staticmethod
     24     def parse_atom(expression):
---> 25         atom = Atom.parse_item(expression)
     26         if not atom:
     27             raise Atom.AtomError(f'{expression} is syntactically␣
 ↪incorrect')

<ipython-input-26-0d4a07671498> in parse_item(_class, expression, parse_subitem
      3     def parse_item(_class, expression, parse_subitem=True):
      4         characters = list(reversed(expression.replace(' ', '')))
----> 5         item = _class.parse_subitem(characters)\
      6                     if parse_subitem\
      7                     else Atom.parse_subitem_sequence(characters)

<ipython-input-24-20bcf86e40da> in parse_subitem(_class, characters)
     11         if not subitems or not characters or characters.pop() != ')':
     12             return
---> 13         return _class(symbol, subitems)
     14 
     15     @classmethod

<ipython-input-18-f566ace642a5> in __init__(self, root, children)
     30                                )
     31         if self.root in self.function_symbols:
---> 32             raise Atom.AtomError('Predicate symbol also function symbol '

     33                                  f'in {self}'
     34                                 )

AtomError: Predicate symbol also function symbol in l(e, l(H, T))
```

Knowing how to parse terms and atoms and create `Term` and `Atom` objects, respectively, we can now define a number of functions to operate on `Expression` objects as needed by a Prolog interpreter. It is necessary to be able to collect all variables that occur in an expression:

[34]:
```
class Expression(Expression):
    def variables(self):
        variables = {self.root} if self.is_variable() else set()
        variables.update(*(child.variables() for child in self.children))
        return variables


class Term(Term, Expression):
    pass
```

```python
class Atom(Atom, Expression):
    pass
```

[35]:
```python
Term.parse_term('x').variables()
Term.parse_term('X').variables()
Term.parse_term('f(X, a, X)').variables()
Term.parse_term('f(X_0, Y, X_0)').variables()
Term.parse_term('f(c, f(X, f(a, Z, b), '
                'f(f(X, Z, U), a, T)), f(a, U, a))'
               ).variables()
```

[35]: set()

[35]: {'X'}

[35]: {'X'}

[35]: {'X_0', 'Y'}

[35]: {'T', 'U', 'X', 'Z'}

Prolog considers underscores within a rule as independent variables. For instance, the fact *loves*(_, _). is meant to express that everyone loves everyone (including oneself), not only that everyone loves oneself: the two occurrences of _ denote arbitrary, independant individuals, and *loves*(_, _) has the same intended meaning as *loves*(_0, _1). A Prolog interpreter needs to treat all occurrences of a given variable in a rule as denotations of the same individual (as a consequence, a Prolog interpreter needs to make sure that whenever an occurrence of a variable in a rule is replaced by a term, then all other occurrences of the variable in the rule are replaced by the term). To that aim, we define in `Expression` a recursive method, `individualise_underscores()`, meant to let in an expression each occurrence of an underscore, used as a full name for a variable, be followed by a unique natural number. We first explain how the method operates thanks to a tracing function:

[36]:
```python
def trace_individualise_underscores(expression, variables, index, depth):
    print('  ' * depth, f'Received index is {index}, processing', expression)
    if expression.root == '_':
        while True:
            index += 1
            next_underscore = '_' + str(index)
            if next_underscore not in variables:
                expression.root = next_underscore
                print('  ' * depth, f'_ changed to {expression}, returned '
                      'index is', index
                     )
                return index
    if not expression.children:
```

```python
        print('  ' * depth, 'Returned index is', index)
        return index
    for child in expression.children:
        index = trace_individualise_underscores(child, variables, index,
                                                depth + 1
                                               )
    print('  ' * depth, 'Returned index is', index)
    return index

atom = Atom.parse_atom('p(g(h(a, _), X), _2, h(a, _), _)')
variables = atom.variables()
print('Variables in expression are:', variables, end='\n\n')
trace_individualise_underscores(atom, variables, -1, 0)
print('expression now is:', atom)
```

```
Variables in expression are: {'_2', 'X', '_'}

 Received index is -1, processing p(g(h(a, _), X), _2, h(a, _), _)
   Received index is -1, processing g(h(a, _), X)
     Received index is -1, processing h(a, _)
       Received index is -1, processing a
       Returned index is -1
       Received index is -1, processing _
       _ changed to _0, returned index is 0
     Returned index is 0
     Received index is 0, processing X
     Returned index is 0
   Returned index is 0
   Received index is 0, processing _2
   Returned index is 0
   Received index is 0, processing h(a, _)
     Received index is 0, processing a
     Returned index is 0
     Received index is 0, processing _
     _ changed to _1, returned index is 1
   Returned index is 1
   Received index is 1, processing _
   _ changed to _3, returned index is 3
 Returned index is 3
```

[36]: 3

```
expression now is: p(g(h(a, _0), X), _2, h(a, _1), _3)
```

The implementation of `individualise_underscores()` in the following cell is a straightforward adaptation of `trace_individualise_underscores()`. When dealing with a fact, `individualise_underscores()` can be called on the `Atom` object that captures the fact, and the default arguments are appropriate. When dealing with a more general rule, `individualise_underscores()` can be called on the rule's head, but the argument `variables` should be given as value the set S of variables that occur in the whole rule, not just in the rule's head; that call would return an integer i_1. Then `individualise_underscores()` can be called on the first atom in the rule's body, with `variables` still set to S, but with the argument `index` given the value i_1; that call would return an integer i_2. Then `individualise_underscores()` can be called on the second atom in the rule's body, if any, with `variables` still set to S, but with the argument `index` given the value i_2… That is a way to give each underscore that occurs in the rule a unique name, not occurring anywhere in the rule. That is not something to see in action yet as for now, we are only dealing with terms and atoms, not rules and logic programs:

[37]:
```
class Expression(Expression):
    def individualise_underscores(self, variables=None, index=-1):
        if variables is None:
            variables = self.variables()
        if self.root == '_':
            while True:
                index += 1
                next_underscore = '_' + str(index)
                if next_underscore not in variables:
                    self.root = next_underscore
                    return index
        if not self.children:
            return index
        for child in self.children:
            index = child.individualise_underscores(variables, index)
        return index


class Term(Term, Expression):
    pass


class Atom(Atom, Expression):
    pass
```

[38]:
```
atom = Atom.parse_atom('p(g(h(a, _), X), _2, h(a, _), _)')
atom.individualise_underscores()
print(atom)
```

[38]: 3

```
p(g(h(a, _0), X), _2, h(a, _1), _3)
```

A Prolog interpreter needs to be able to substitute in an expression E all occurrences of some of the variables that occur in E by terms; sometimes, the terms will simply be computed fresh variables, for a particular kind of substitution referred to as a *renaming of variables in E*. We extend `Expression` with a static method, `fresh_variables()`, meant to take two arguments, `variables` and `reserved_variables`, both expected to be sets of variables, and return a dictionary D whose keys are the members of both `variables` and `reserved_variables`, and whose values are pairwise distinct variables, different to those in both sets. The intention is that given an expression E, `variables` will denote the set of variables that occur in E; any such variable v that happens to belong to `reserved_variables` can then be replaced by $D[v]$ in E, resulting in an expression where fresh variables have replaced those in both `variables` and `reserved_variables`. We let the name of a variable that belongs to both `variables` and `reserved_variables` be followed by an underscore and the smallest possible natural number for the mapped to fresh variable:

[39]:
```
class Expression(Expression):
    @staticmethod
    def fresh_variables(variables, reserved_variables):
        substitutions = {}
        # Any variable Var that occurs in both variables and
        # reserved_variables will be renamed to Var_i where i is the
        # least natural number that makes Var_i a new variable (that is,
        # occurring neither in variables nor in reserved_variables nor
        # in the set of variables that have been created so far, if
        # any).
        for variable in variables & reserved_variables:
            i = 0
            while ''.join((variable, '_', str(i))) in variables\
                                                    | reserved_variables:
                i += 1
            substitutions[variable] = ''.join((variable, '_', str(i)))
        return substitutions
```

[40]:
```
Expression.fresh_variables({'a'}, set())
Expression.fresh_variables({'Y'}, {'X'})
Expression.fresh_variables({'X'}, {'X'})
Expression.fresh_variables({'Y', 'Z'}, {'X', 'Y'})
Expression.fresh_variables({'U', 'Y', 'Z'}, {'X', 'Y', 'Z'})
```

[40]: `{}`

[40]: `{}`

[40]: `{'X': 'X_0'}`

[40]: `{'Y': 'Y_0'}`

[40]: `{'Y': 'Y_0', 'Z': 'Z_0'}`

When the Prolog interpreter needs to rename variables in an expression E, it actually always has to leave E untouched and perform the renaming on a copy of E. The `deepcopy` function from the `copy` module offers a good solution to cloning an `Expression` object. Compare:

[41]:
```
L = [0]
# Alternatively, use the copy method of the List class:
# L_copy = L.copy()
L_copy = copy(L)
L_copy[0] = 1
```

```
L_copy, L

L = [[0]]
# Alternatively, use the copy method of the List class:
# L_copy = L.copy()
L_copy = copy(L)
L_copy[0][0] = 1
L_copy, L

L = [[0]]
L_deepcopy = deepcopy(L)
L_deepcopy[0][0] = 1
L_deepcopy, L
```

[41]:
```
([1], [0])
```

[41]:
```
([[1]], [[1]])
```

[41]:
```
([[1]], [[0]])
```

We extend `Expression` with a method, `rename_variables()`, that recursively renames variables in a copy of an expression, the renaming taking the form of a dictionary mapping variables to replace to replacing variables:

[42]:
```
class Expression(Expression):
    def rename_variables(self, substitution):
        return deepcopy(self)._rename_variables(substitution)

    def _rename_variables(self, substitution):
        if self.is_variable():
            if self.root in substitution:
                self.root = substitution[self.root]
        else:
            for child in self.children:
                child._rename_variables(substitution)
        return self


class Term(Term, Expression):
    pass


class Atom(Atom, Expression):
    pass
```

[43]:
```
atom = Atom.parse_atom('join(l(H, T),X, l(H, Y))')
print(atom.rename_variables({'X': 'A', 'Y': 'B', 'Z': 'C'}))
print(atom)
```

```
join(l(H, T), A, l(H, B))
join(l(H, T), X, l(H, Y))
```

More general substitutions of variables by terms in an expression E will have to be performed sometimes in E itself, sometimes in a copy of E. We extend `Expression` with two methods, `substitute()` and `substitute_in_copy()`, for both kinds of substitutions, the latter just calling the former on a copy of the object it applies the method to. When a term t replaces a variable v in E, t might itself contain variables that belong to the domain of the substitution. In that case, we apply the substitution again, many times if needed. If the substitution requested to replace a variable by itself, or requested to replace a variable v_1 by a variable v_2 and the other way around, the procedure would loop forever. The kind of substitution that will be used in practice will not make this possible: after a finite number of applications of the substitution, the resulting expression will contain no occurrence of a variable in the domain of the substitution. Note that the assignment in the body of `substitute()` cannot be replaced by `self = substitution[self.root]`:

[44]:
```python
class Expression(Expression):
    def substitute_in_copy(self, substitution):
        return deepcopy(self).substitute(substitution)

    def substitute(self, substitution):
        if self.is_variable():
            if self.root in substitution:
                self.root, self.children = substitution[self.root].root,\
                                           substitution[self.root].children
                self.substitute(substitution)
        else:
            for child in self.children:
                child.substitute(substitution)
        return self


class Term(Term, Expression):
    pass


class Atom(Atom, Expression):
    pass
```

[45]:
```python
term = Term.parse_term('f(a, X, g(Y, b), h(h(h(Z))))')
print(term.substitute({'X': Term.parse_term('X_0'), 'Z': Term.parse_term('Z_0')
                      }
                     )
     )

term = Term.parse_term('h(h(f(U, Y, g(Z, Z))))')
print(term.substitute({'Z': Term.parse_term('V')}))

term = Term.parse_term('f(a, X, g(Y, b), h(h(h(Z))))')
```

```
print(term.substitute({'X_0' : Term.parse_term('h(Z_0)'),
                       'X': Term.parse_term('X_0'), 'Z_0': Term.parse_term('c')
                      }
                     )
     )
```

```
f(a, X_0, g(Y, b), h(h(h(Z_0))))
h(h(f(U, Y, g(V, V))))
f(a, h(c), g(Y, b), h(h(h(Z))))
```

At the heart of Prolog lies the *unification algorithm*. It computes a most general unifier (*mgu*) for two expressions E_1 and E_2, that is, a substitution θ such that:

- applying θ to E_1 and E_2 results in the same expression;
- for any substitution ψ that applied to E_1 and E_2, results in the same expression, there exists a substitution ν such that applying ψ to E_1 and E_2 is the same as applying θ to E_1 and E_2, and then ν to the resulting expressions.

For instance, if E_1 is $f(X_1, h(X_1), X_2)$ and E_2 is $f(g(X_3), X_4, X_3)$, then the substitution that maps X_1 to $g(X_3)$, X_2 to X_3 and X_4 to $h(g(X_3))$ is an mgu for E_1 and E_2; when applied to E_1 and E_2, it results in the expression $f(g(X_3), h(g(X_3)), X_3)$. If a is a constant, then the substitution that maps X_1 to $g(a)$, X_2 and X_3 to a and X_4 to $h(g(a))$ results in the same expression when applied to E_1 and E_2, namely, $f(g(a), h(g(a)), a)$; but it suffices to apply the substitution that maps X_3 to a to get $f(g(a), h(g(a)), a)$ from $f(g(X_3), h(g(X_3)), X_3)$.

So a most general unifier for two expressions E_1 and E_2 is a substitution θ that specialises E_1 and E_2 to the same expression in the most general way: any other substitution that specialises E_1 and E_2 to the same expression can be obtained by instantiating θ. A most general unifier is unique up to a renaming of variables.

The following tracing function, `trace_unify_identities()`, is meant to explain and illustrate the unification algorithm. It takes as argument a list of pairs of expressions. To compute an mgu for two expressions E_1 and E_2, `trace_unify_identities()` is called with $[(E_1, E_2)]$ provided as argument:

[46]:
```
def trace_unify_identities(identities):
    mgu = {}
    while identities:
        print('Identities left to process: ', end='')
        print(', '.join(f'{identity[0]} = {identity[1]}'
                            for identity in identities
                       )
             )
        expression_1, expression_2 = identities.pop()
        print('    Dealing with:', expression_1, '=', expression_2)
        if expression_1.root == expression_2.root:
            for (expr_1, expr_2) in zip(expression_1.children,
                                        expression_2.children
                                       ):
                print('    Adding:', expr_1, '=', expr_2)
```

```
                identities.append([expr_1, expr_2])
        elif expression_1.is_variable():
            # Occurrence check (omitted in most Prolog implementations)
            if expression_1.root in expression_2.variables():
                print(f'    {expression_1.root} occurs in {expression_2},',
                      'giving up!'
                     )
                return
            print('    Extending mgu with:', expression_1.root, '->',
                  expression_2
                 )
            mgu[expression_1.root] = expression_2
            for identity in identities:
                print('    Changing (or not)', identity[0], '=', identity[1],
                      'to ', end=''
                     )
                for i in range(2):
                    identity[i] = identity[i].substitute_in_copy(
                                        {expression_1.root: expression_2}
                                                              )
                print(identity[0], '=', identity[1])
        elif expression_2.is_variable():
            print('    Adding', expression_2, '=', expression_1)
            identities.append([expression_2, expression_1])
        else:
            print('    Equality cannot be satisfied, giving up!')
            return
        print()
    print('Mgu:')
    print('\n'.join(f'    {var} -> {mgu[var]}' for var in mgu))
    return mgu
```

[47]:
```
term_1 = Term.parse_term('X')
term_2 = Term.parse_term('X')
mgu = trace_unify_identities([(term_1, term_2)])

if mgu is not None:
    print('\nExpressions after application of mgu:')
    print('   ', term_1.substitute(mgu))
    print('   ', term_2.substitute(mgu))
```

```
Identities left to process: X = X
    Dealing with: X = X

Mgu:


Expressions after application of mgu:
```

```
    X
    X
```

[48]:
```
term_1 = Term.parse_term('X')
term_2 = Term.parse_term('a')
mgu = trace_unify_identities([(term_1, term_2)])

if mgu is not None:
    print('\nExpressions after application of mgu:')
    print('    ', term_1.substitute(mgu))
    print('    ', term_2.substitute(mgu))
```

```
Identities left to process: X = a
    Dealing with: X = a
    Extending mgu with: X -> a

Mgu:
    X -> a

Expressions after application of mgu:
    a
    a
```

[49]:
```
term_1 = Term.parse_term('X')
term_2 = Term.parse_term('Y')
mgu = trace_unify_identities([(term_1, term_2)])

if mgu is not None:
    print('\nExpressions after application of mgu:')
    print('    ', term_1.substitute(mgu))
    print('    ', term_2.substitute(mgu))
```

```
Identities left to process: X = Y
    Dealing with: X = Y
    Extending mgu with: X -> Y

Mgu:
    X -> Y

Expressions after application of mgu:
    Y
    Y
```

[50]:
```
term_1 = Term.parse_term('f(X, Y)')
term_2 = Term.parse_term('f(Y, X)')
mgu = trace_unify_identities([(term_1, term_2)])

if mgu is not None:
```

```python
    print('\nExpressions after application of mgu:')
    print('    ', term_1.substitute(mgu))
    print('    ', term_2.substitute(mgu))
```

```
Identities left to process: f(X, Y) = f(Y, X)
    Dealing with: f(X, Y) = f(Y, X)
    Adding: X = Y
    Adding: Y = X

Identities left to process: X = Y, Y = X
    Dealing with: Y = X
    Extending mgu with: Y -> X
    Changing (or not) X = Y to X = X

Identities left to process: X = X
    Dealing with: X = X

Mgu:
    Y -> X

Expressions after application of mgu:
    f(X, X)
    f(X, X)
```

[51]:
```python
term_1 = Term.parse_term('f(X1, h(X1), X2)')
term_2 = Term.parse_term('f(g(X3), X4, X3)')
mgu = trace_unify_identities([(term_1, term_2)])

if mgu is not None:
    print('\nExpressions after application of mgu:')
    print('    ', term_1.substitute(mgu))
    print('    ', term_2.substitute(mgu))
```

```
Identities left to process: f(X1, h(X1), X2) = f(g(X3), X4, X3)
    Dealing with: f(X1, h(X1), X2) = f(g(X3), X4, X3)
    Adding: X1 = g(X3)
    Adding: h(X1) = X4
    Adding: X2 = X3

Identities left to process: X1 = g(X3), h(X1) = X4, X2 = X3
    Dealing with: X2 = X3
    Extending mgu with: X2 -> X3
    Changing (or not) X1 = g(X3) to X1 = g(X3)
    Changing (or not) h(X1) = X4 to h(X1) = X4

Identities left to process: X1 = g(X3), h(X1) = X4
    Dealing with: h(X1) = X4
    Adding X4 = h(X1)
```

```
Identities left to process: X1 = g(X3), X4 = h(X1)
    Dealing with: X4 = h(X1)
    Extending mgu with: X4 -> h(X1)
    Changing (or not) X1 = g(X3) to X1 = g(X3)

Identities left to process: X1 = g(X3)
    Dealing with: X1 = g(X3)
    Extending mgu with: X1 -> g(X3)

Mgu:
    X2 -> X3
    X4 -> h(X1)
    X1 -> g(X3)

Expressions after application of mgu:
    f(g(X3), h(g(X3)), X3)
    f(g(X3), h(g(X3)), X3)
```

[52]:

```python
term_1 = Term.parse_term('f(X1 ,g(X2, X3), X2, b)')
term_2 = Term.parse_term('f(g(h(a, X5), X2), X1, h(a, X4), X4)')
mgu = trace_unify_identities([(term_1, term_2)])

if mgu is not None:
    print('\nExpressions after application of mgu:')
    print('    ', term_1.substitute(mgu))
    print('    ', term_2.substitute(mgu))
```

```
Identities left to process: f(X1, g(X2, X3), X2, b) = f(g(h(a, X5), X2), X1,
h(a, X4), X4)
    Dealing with: f(X1, g(X2, X3), X2, b) = f(g(h(a, X5), X2), X1, h(a, X4), X4)
    Adding: X1 = g(h(a, X5), X2)
    Adding: g(X2, X3) = X1
    Adding: X2 = h(a, X4)
    Adding: b = X4

Identities left to process: X1 = g(h(a, X5), X2), g(X2, X3) = X1, X2 = h(a, X4),
b = X4
    Dealing with: b = X4
    Adding X4 = b

Identities left to process: X1 = g(h(a, X5), X2), g(X2, X3) = X1, X2 = h(a, X4),
X4 = b
    Dealing with: X4 = b
    Extending mgu with: X4 -> b
    Changing (or not) X1 = g(h(a, X5), X2) to X1 = g(h(a, X5), X2)
    Changing (or not) g(X2, X3) = X1 to g(X2, X3) = X1
    Changing (or not) X2 = h(a, X4) to X2 = h(a, b)
```

```
Identities left to process: X1 = g(h(a, X5), X2), g(X2, X3) = X1, X2 = h(a, b)
    Dealing with: X2 = h(a, b)
    Extending mgu with: X2 -> h(a, b)
    Changing (or not) X1 = g(h(a, X5), X2) to X1 = g(h(a, X5), h(a, b))
    Changing (or not) g(X2, X3) = X1 to g(h(a, b), X3) = X1

Identities left to process: X1 = g(h(a, X5), h(a, b)), g(h(a, b), X3) = X1
    Dealing with: g(h(a, b), X3) = X1
    Adding X1 = g(h(a, b), X3)

Identities left to process: X1 = g(h(a, X5), h(a, b)), X1 = g(h(a, b), X3)
    Dealing with: X1 = g(h(a, b), X3)
    Extending mgu with: X1 -> g(h(a, b), X3)
    Changing (or not) X1 = g(h(a, X5), h(a, b)) to g(h(a, b), X3) = g(h(a, X5),
h(a, b))

Identities left to process: g(h(a, b), X3) = g(h(a, X5), h(a, b))
    Dealing with: g(h(a, b), X3) = g(h(a, X5), h(a, b))
    Adding: h(a, b) = h(a, X5)
    Adding: X3 = h(a, b)

Identities left to process: h(a, b) = h(a, X5), X3 = h(a, b)
    Dealing with: X3 = h(a, b)
    Extending mgu with: X3 -> h(a, b)
    Changing (or not) h(a, b) = h(a, X5) to h(a, b) = h(a, X5)

Identities left to process: h(a, b) = h(a, X5)
    Dealing with: h(a, b) = h(a, X5)
    Adding: a = a
    Adding: b = X5

Identities left to process: a = a, b = X5
    Dealing with: b = X5
    Adding X5 = b

Identities left to process: a = a, X5 = b
    Dealing with: X5 = b
    Extending mgu with: X5 -> b
    Changing (or not) a = a to a = a

Identities left to process: a = a
    Dealing with: a = a

Mgu:
    X4 -> b
    X2 -> h(a, b)
    X1 -> g(h(a, b), X3)
```

```
    X3 -> h(a, b)
    X5 -> b

Expressions after application of mgu:
    f(g(h(a, b), h(a, b)), g(h(a, b), h(a, b)), h(a, b), b)
    f(g(h(a, b), h(a, b)), g(h(a, b), h(a, b)), h(a, b), b)
```

[53]:
```
term_1 = Term.parse_term('f(X, a)')
term_2 = Term.parse_term('f(b, X)')
mgu = trace_unify_identities([(term_1, term_2)])

if mgu is not None:
    print('\nExpressions after application of mgu:')
    print('    ', term_1.substitute(mgu))
    print('    ', term_2.substitute(mgu))
```

```
Identities left to process: f(X, a) = f(b, X)
    Dealing with: f(X, a) = f(b, X)
    Adding: X = b
    Adding: a = X

Identities left to process: X = b, a = X
    Dealing with: a = X
    Adding X = a

Identities left to process: X = b, X = a
    Dealing with: X = a
    Extending mgu with: X -> a
    Changing (or not) X = b to a = b

Identities left to process: a = b
    Dealing with: a = b
    Equality cannot be satisfied, giving up!
```

[54]:
```
term_1 = Term.parse_term('f(X, Y, U)')
term_2 = Term.parse_term('f(Y, U, g(X))')
mgu = trace_unify_identities([(term_1, term_2)])

if mgu is not None:
    print('\nExpressions after application of mgu:')
    print('    ', term_1.substitute(mgu))
    print('    ', term_2.substitute(mgu))
```

```
Identities left to process: f(X, Y, U) = f(Y, U, g(X))
    Dealing with: f(X, Y, U) = f(Y, U, g(X))
    Adding: X = Y
    Adding: Y = U
    Adding: U = g(X)
```

```
Identities left to process: X = Y, Y = U, U = g(X)
    Dealing with: U = g(X)
    Extending mgu with: U -> g(X)
    Changing (or not) X = Y to X = Y
    Changing (or not) Y = U to Y = g(X)

Identities left to process: X = Y, Y = g(X)
    Dealing with: Y = g(X)
    Extending mgu with: Y -> g(X)
    Changing (or not) X = Y to X = g(X)

Identities left to process: X = g(X)
    Dealing with: X = g(X)
    X occurs in g(X), giving up!
```

The implementation of the unification algorithm in `Expression` should now be clear:

[55]:
```
class Expression(Expression):
    @staticmethod
    def unify_identities(identities):
        mgu = {}
        while identities:
            expression_1, expression_2 = identities.pop()
            if expression_1.root == expression_2.root:
                identities.extend([expr_1, expr_2] for (expr_1, expr_2) in
                                                     zip(expression_1.children,
                                                         expression_2.children
                                                        )
                                 )
            elif expression_1.is_variable():
                # Occurrence check (omitted in most Prolog
                # implementations)
                if expression_1.root in expression_2.variables():
                    return
                mgu[expression_1.root] = expression_2
                for identity in identities:
                    for i in range(2):
                        identity[i] = identity[i].substitute_in_copy(
                                          {expression_1.root: expression_2}
                                                                      )
            elif expression_2.is_variable():
                identities.append([expression_2, expression_1])
            else:
                return
        return mgu

    def unify(self, expression):
        return Expression.unify_identities([[self, expression]])
```

```python
class Term(Term, Expression):
    pass


class Atom(Atom, Expression):
    pass
```

[56]:
```python
term = Term.parse_term('X')
mgu = term.unify(Term.parse_term('X'))
{var: str(mgu[var]) for var in mgu}

term = Term.parse_term('X')
mgu = term.unify(Term.parse_term('a'))
{var: str(mgu[var]) for var in mgu}

term = Term.parse_term('X')
mgu = term.unify(Term.parse_term('Y'))
{var: str(mgu[var]) for var in mgu}

term = Term.parse_term('f(X, Y)')
mgu = term.unify(Term.parse_term('f(Y, X)'))
{var: str(mgu[var]) for var in mgu}

term = Term.parse_term('f(X1, h(X1), X2)')
mgu = term.unify(Term.parse_term('f(g(X3), X4, X3)'))
{var: str(mgu[var]) for var in mgu}

term = Term.parse_term('f(X1 ,g(X2, X3), X2, b)')
mgu = term.unify(Term.parse_term('f(g(h(a, X5), X2), X1, h(a, X4), X4)'))
{var: str(mgu[var]) for var in mgu}
```

[56]:
```
{}
```

[56]:
```
{'X': 'a'}
```

[56]:
```
{'X': 'Y'}
```

[56]:
```
{'Y': 'X'}
```

[56]:
```
{'X2': 'X3', 'X4': 'h(X1)', 'X1': 'g(X3)'}
```

[56]:
```
{'X4': 'b',
 'X2': 'h(a, b)',
 'X1': 'g(h(a, b), X3)',
 'X3': 'h(a, b)',
 'X5': 'b'}
```

We have all the code we need to build and operate on terms and atoms. We can now further compose:

- conjunctions of atoms as rule bodies;
- rule heads (atoms) and rule bodies as rules;
- sequences of rules as logic programs.

We define a new class for each: `Conjunction`, `Rule`, and `LogicProgram`.

A `Conjunction` object is intended to be a list of `Atom` objects. This is achieved by letting `Conjunction` inherit from `list`, together with a static method, `parse_conjunction()`, whose implementation is similar to `Term`'s `parse_term()` and `Atom`'s `parse_atom()`static methods, respectively: `parse_conjunction()` calls `Expression`'s `parse_item()` method, but with its keyword argument `parse_subitem` set to `False`, so that it then calls `parse_subitem_sequence()` rather than `parse_subitem()`, as previously discussed. Also, the implementation of `Conjunction`'s `__init__()` method is similar to the implementation of `Term`'s and `Atom`'s `__init__()` methods:

- It collects all function symbols and all predicate symbols that occur in at least one atom in the conjunction, in the form of two dictionaries, with symbols as keys and symbols' arities as values.
- A `ConjunctionError` error is raised by `parse_conjunction()` in case the call to `parse_item()` fails to return a list of `Atom` objects, or during a last round of syntactic checks by `Conjunction`'s `__init__()` method, to make sure that:
  - no predicate symbol and no function symbol is used with different arities in different atoms;
  - no symbol is used as a predicate symbol in an atom and as a function symbol in another atom.

[57]:
```
class Conjunction(list, Expression):
    class ConjunctionError(Exception):
        pass

    def __init__(self, conjuncts=[]):
        super().__init__(conjuncts)
        self.predicate_symbols = {}
        self.function_symbols = {}
        if not consistently_add_to(self.predicate_symbols,
                                   *((atom.root, len(atom.children))
                                         for atom in self
                                     )
                                  ):
            raise Conjunction.ConjunctionError('Predicate symbol used with '
                                               f'different arities in {self}'
                                              )
        for atom in self:
            if not consistently_add_to(self.function_symbols,
                                       *atom.function_symbols.items()
                                      ):
                raise Conjunction.ConjunctionError('Function symbol used with '
```

```
                                                'different arities in '
                                                f'{self}'
                                               )
        if self.predicate_symbols.keys() & self.function_symbols.keys():
            raise Conjunction.ConjunctionError('Predicate symbol also '
                                               f'function symbol in {self}'
                                              )

    def __str__(self):
        return ', '.join(str(atom) for atom in self)

    @staticmethod
    def parse_conjunction(expression):
        if expression.isspace():
            return Conjunction()
        conjunction = Expression.parse_item(expression, parse_subitem=False)
        if not conjunction:
            raise Conjunction.ConjunctionError(f'{expression} is '
                                               'syntactically incorrect'
                                              )
        return Conjunction(conjunction)
```

[58]:
```
conjunction = Conjunction.parse_conjunction('father')
print(conjunction)
conjunction.predicate_symbols, conjunction.function_symbols

conjunction =\
          Conjunction.parse_conjunction('male(X), parent(Z, X), parent(Z, Y)')
print(conjunction)
conjunction.predicate_symbols, conjunction.function_symbols

conjunction = Conjunction.parse_conjunction('reduce(X, add(mult(N, x), M1)), '
                                            'reduce(Y, add(mult(o, x), M2)), '
                                            'plus(M1, M2, M)'
                                           )
print(conjunction)
conjunction.predicate_symbols, conjunction.function_symbols
```

```
father
```

[58]:
```
({'father': 0}, {})
```

```
male(X), parent(Z, X), parent(Z, Y)
```

[58]:
```
({'male': 1, 'parent': 2}, {})
```

```
reduce(X, add(mult(N, x), M1)), reduce(Y, add(mult(o, x), M2)), plus(M1, M2, M)
```

[58]:
```
({'reduce': 2, 'plus': 3}, {'add': 2, 'mult': 2, 'x': 0, 'o': 0})
```

[59]:
```
Conjunction.parse_conjunction('plus(X, Y, U), plus(times(U, Y, Z)')
```

```
---------------------------------------------------------------------------
ConjunctionError                          Traceback (most recent call last)
<ipython-input-59-43ae692c35cf> in <module>
----> 1 Conjunction.parse_conjunction('plus(X, Y, U), plus(times(U, Y, Z)')

<ipython-input-57-2adb80006e67> in parse_conjunction(expression)
     37         conjunction = Expression.parse_item(expression,␣
 ↪parse_subitem=False)
     38         if not conjunction:
---> 39             raise Conjunction.ConjunctionError(f'{expression} is '
     40                                                'syntactically incorrect
     41                                               )

ConjunctionError: plus(X, Y, U), plus(times(U, Y, Z) is syntactically incorrect
```

[60]:
```
Conjunction.parse_conjunction('male(X), parent(Z, X), parent(male(Z), Y)')
```

```
---------------------------------------------------------------------------
ConjunctionError                          Traceback (most recent call last)
<ipython-input-60-dee8b9fa0fe1> in <module>
----> 1 Conjunction.parse_conjunction('male(X), parent(Z, X), parent(male(Z),␣
 ↪Y)')

<ipython-input-57-2adb80006e67> in parse_conjunction(expression)
     40                                                'syntactically incorrect
     41                                               )
---> 42         return Conjunction(conjunction)

<ipython-input-57-2adb80006e67> in __init__(self, conjuncts)
     24                                                   )
     25         if self.predicate_symbols.keys() & self.function_symbols.keys()
---> 26             raise Conjunction.ConjunctionError('Predicate symbol also '
     27                                                f'function symbol in␣
 ↪{self}'
     28                                               )

ConjunctionError: Predicate symbol also function symbol in male(X), parent(Z,␣
 ↪X), parent(male(Z), Y)
```

Then we implement the `Rule` class. Besides the `function_symbols` and `predicate_symbols` attributes, objects of type `Rule` have a `head` attribute, meant to denote an `Atom` object, a `body` attribute, meant to denote a `Conjunction` object, and a `variables` attribute, meant to denote the set of variables occurring in the atoms that make up a rule. The `Rule` class has a `parse_rule()` static method that makes sure a rule ends in a full stop and either consists of a single fact, or has a head and a body separated by `:-`; it then relies on the `Atom.parse_atoms()` and `Conjunction.parse_conjunction()` static methods to parse head and body and return `Atom` and `Conjunction` objects, respectively. Besides all syntactic checks performed by those two functions, `Rule`'s `__init__()` method makes sure that:

- the predicate symbol in the head is not used as a predicate symbol with a different arity in the body;
- no function symbol in the head is used as a function symbol with a different arity in the body;
- no symbol is used as a predicate symbol in the head and as a function symbol in the body, or the other way around.

[61]:
```
class Rule:
    class RuleError(Exception):
        pass

    def __init__(self, head, body=Conjunction()):
        self.head = head
        self.body = body
        self.variables =  self.head.variables()
        self.variables.update(*(atom.variables() for atom in self.body))
        if not self.body:
            return
        if not consistently_add_to({self.head.root: len(self.head.children)},
                                   *self.body.predicate_symbols.items()
                                  ):
            raise Rule.RuleError("Head's predicate symbol used with "
                                 f'different arities in {self}'
                                )
        if not consistently_add_to(self.head.function_symbols,
                                   *self.body.function_symbols.items()
                                  ):
            raise Rule.RuleError('Function symbol used with different '
                                 f'arities in head and body of {self}'
                                )
        if self.head.root in self.body.function_symbols:
            raise Rule.RuleError('Predicate symbol in head also '
                                 f'function symbol in body of {self}'
                                )
        if self.head.function_symbols.keys()\
           & self.body.predicate_symbols.keys():
            raise Rule.RuleError('Function symbol in head also '
                                 f'predicate symbol in body of {self}'
                                )

    def __str__(self):
        if not self.body:
            return str(self.head) + '.'
```

```
        return ' :- '.join((str(self.head),
                            ', '.join(str(atom) for atom in self.body)
                           )
                          ) + '.'

    @staticmethod
    def parse_rule(expression):
        if expression[-1] != '.':
            raise Rule.RuleError(f'{expression} does not end in a full stop')
        rule = expression[: -1].split(':-')
        if len(rule) < 1 or len(rule) > 2 or\
           len(rule) == 2 and rule[1].isspace():
            raise Rule.RuleError(f'{expression} is syntactically invalid')
        head = Atom.parse_atom(rule[0])
        if len(rule) == 1:
            return Rule(head)
        return Rule(head, Conjunction.parse_conjunction(rule[1]))
```

[62]:

```
rule = Rule.parse_rule('female(alice).')
print(rule)
rule.variables

rule = Rule.parse_rule('sisterof(X, Y)  :-  parents(X, M, F), female(X), '
                       'parents(Y, M, F).'
                      )
print(rule)
rule.variables
```

```
female(alice).
```

[62]:

```
set()
```

```
sisterof(X, Y) :- parents(X, M, F), female(X), parents(Y, M, F).
```

[62]:

```
{'F', 'M', 'X', 'Y'}
```

[63]:

```
Rule.parse_rule('female(alice)')
```

```
---------------------------------------------------------------------------
RuleError                                 Traceback (most recent call last)
<ipython-input-63-f2a9f1c71644> in <module>
----> 1 Rule.parse_rule('female(alice)')

<ipython-input-61-492361550def> in parse_rule(expression)
     43     def parse_rule(expression):
     44         if expression[-1] != '.':
---> 45             raise Rule.RuleError(f'{expression} does not end in a full␣
 ↪stop')
```

```
     46         rule = expression[: -1].split(':-')
     47         if len(rule) < 1 or len(rule) > 2 or\

RuleError: female(alice) does not end in a full stop
```

[64]:
```python
Rule.parse_rule(' :- female(alice).')
```

```
---------------------------------------------------------------------------
AtomError                                 Traceback (most recent call last)
<ipython-input-64-b9cb3c66fb32> in <module>
----> 1 Rule.parse_rule(' :- female(alice).')

<ipython-input-61-492361550def> in parse_rule(expression)
     48            len(rule) == 2 and rule[1].isspace():
     49             raise Rule.RuleError(f'{expression} is syntactically␣
 ↪invalid')
---> 50         head = Atom.parse_atom(rule[0])
     51         if len(rule) == 1:
     52             return Rule(head)

<ipython-input-26-0d4a07671498> in parse_atom(expression)
     25         atom = Atom.parse_item(expression)
     26         if not atom:
---> 27             raise Atom.AtomError(f'{expression} is syntactically␣
 ↪incorrect')
     28         return atom

AtomError:   is syntactically incorrect
```

[65]:
```python
Rule.parse_rule(' female(alice) :- .')
```

```
---------------------------------------------------------------------------
RuleError                                 Traceback (most recent call last)
<ipython-input-65-38020bbfd69c> in <module>
----> 1 Rule.parse_rule(' female(alice) :- .')

<ipython-input-61-492361550def> in parse_rule(expression)
     47         if len(rule) < 1 or len(rule) > 2 or\
     48            len(rule) == 2 and rule[1].isspace():
---> 49             raise Rule.RuleError(f'{expression} is syntactically␣
 ↪invalid')
     50         head = Atom.parse_atom(rule[0])
     51         if len(rule) == 1:

RuleError:  female(alice) :- . is syntactically invalid
```

[66]:
```
Rule.parse_rule('sisterof(X, Y)  :-  parents(X, M, F), sisterof(X),'
                ' parents(Y, M, F).'
               )
```

```
---------------------------------------------------------------------------
RuleError                                 Traceback (most recent call last)
<ipython-input-66-49a1de612b90> in <module>
----> 1 Rule.parse_rule('sisterof(X, Y)  :-  parents(X, M, F), sisterof(X),'
      2                 ' parents(Y, M, F).'
      3                )

<ipython-input-61-492361550def> in parse_rule(expression)
     51         if len(rule) == 1:
     52             return Rule(head)
---> 53         return Rule(head, Conjunction.parse_conjunction(rule[1]))

<ipython-input-61-492361550def> in __init__(self, head, body)
     13                                  *self.body.predicate_symbols.items()
     14                                 ):
---> 15             raise Rule.RuleError("Head's predicate symbol used with "
     16                                  f'different arities in {self}'
     17                                 )

RuleError: Head's predicate symbol used with different arities in sisterof(X, Y)
↪:- parents(X, M, F), sisterof(X), parents(Y, M, F).
```

[67]:
```
Rule.parse_rule('times(s(X), Y, Z) :- plus(X, Y, U), minus(times(U, Y, Z)).')
```

```
---------------------------------------------------------------------------
RuleError                                 Traceback (most recent call last)
<ipython-input-67-74926f5e6e4c> in <module>
----> 1 Rule.parse_rule('times(s(X), Y, Z) :- plus(X, Y, U), minus(times(U, Y,
↪Z)).')

<ipython-input-61-492361550def> in parse_rule(expression)
     51         if len(rule) == 1:
     52             return Rule(head)
---> 53         return Rule(head, Conjunction.parse_conjunction(rule[1]))

<ipython-input-61-492361550def> in __init__(self, head, body)
     23                                 )
     24         if self.head.root in self.body.function_symbols:
---> 25             raise Rule.RuleError('Predicate symbol in head also '
     26                                  f'function symbol in body of {self}'
     27                                 )
```

```
RuleError: Predicate symbol in head also function symbol in body of times(s(X),
↪Y, Z) :- plus(X, Y, U), minus(times(U, Y, Z)).
```

In the following cell, we implement that part of the LogicProgram class that creates a logic program object from the contents of a file, with three attributes: program, meant to denote a list of Rule objects, and predicate_symbols and function_symbols, meant to denote the sets of predicate and function symbols, respectively, occurring in the rules that make up the logic program. LogicProgram's __init__() method makes sure that:

- no predicate symbol or function symbol is used with different arities in different rules;
- no symbol is used both as a predicate symbol and as a function symbol in different rules.

Prolog comments start with %; lines in the file whose first nonspace symbol is %, as well as blank lines, are ignored.

The individualise_underscores() method of the Expression class is used to, for each rule, replace every occurrence of _ in the rule as a new, unique variable.

[68]:
```
class LogicProgram:
    class LogicProgramError(Exception):
        pass

    def __init__(self, filename):
        self.program = []
        self.predicate_symbols = {}
        self.function_symbols = {}
        rule_nb = 0
        with open(filename) as file:
            for rule in file:
                rule = rule.strip()
                if not rule or rule.startswith('%'):
                    continue
                rule = Rule.parse_rule(rule)
                rule_nb += 1
                if not consistently_add_to(self.predicate_symbols,
                                           (rule.head.root,
                                            len(rule.head.children)
                                           ),
                                           *rule.body.predicate_symbols.items()
                                          ):
                    raise LogicProgram.LogicProgramError(
                            'Predicate symbol used with different arities '
                            f'in rule nb {rule_nb} and in previous rules'
                                                              )
                if not consistently_add_to(self.function_symbols,
                                           *rule.head.function_symbols.items(),
                                           *rule.body.function_symbols.items()
```

```
                                        ):
                    raise LogicProgram.LogicProgramError(
                                'Function symbol used with different arities '
                                f'in rule nb {rule_nb} and in previous rules'
                                                        )
                if self.predicate_symbols.keys()\
                   & self.function_symbols.keys():
                    raise LogicProgram.LogicProgramError(
                          'Symbol used as both predicate and function symbols '
                          f'in rule nb {rule_nb} and in previous rules'
                                                        )
                underscore_index =\
                        rule.head.individualise_underscores(rule.variables)
                for atom in rule.body:
                    underscore_index =\
                            atom.individualise_underscores(rule.variables,
                                                           underscore_index
                                                          )
                self.program.append(rule)
```

Everything is now in place for the Prolog interpreter to solve queries, with an exploration of the search tree that defaults to depth-first, but that can be changed to breadth-first. The following tracing function, `trace_solve()`, is meant to explain and illustrate the workings of the interpreter. The second argument, `query`, is meant to be a string that represents a conjunction of atoms, possibly reduced to a single one, to be interpreted as a goal or an implicitly conjuncted sequence of goals. The only place where depth-first and breadth-first explorations differ is at the very end of the function:

[69]:
```
def trace_solve(logic_program, query, depth_first=True):
    query = Conjunction.parse_conjunction(query)
    query_variables = {var for atom in query for var in atom.variables()}
    # A list of pairs consisting of:
    # - a list of goals to be solved, and
    # - the substitution to apply to the variables that occur in the
    #   query as determined by the unifications computed so far.
    goals_solution_pairs =\
            deque([(deque(query), {var: Term(var) for var in query_variables})
                  ]
                 )
    while goals_solution_pairs:
        goals, solution = goals_solution_pairs.popleft()
        print('\nDealing with following goals and partial solution:')
        print('    ', ', '.join(str(goal) for goal in goals), end='')
        print('    ', ', '.join(f'{var} -> {solution[var]}'
                                    for var in solution)
             )
        if not goals:
```

```
            print('    No goal left, solution is complete')
            yield {var: str(solution[var]) for var in solution}
            continue
        reserved_variables =\
          query_variables | {var for atom in goals for var in atom.variables()}
        goal = goals.popleft()
        next_goals_solution_pairs = deque()
        for rule in logic_program.program:
            variable_renaming = Expression.fresh_variables(rule.variables,
                                                           reserved_variables
                                                          )
            head = rule.head.rename_variables(variable_renaming)
            mgu = goal.unify(head)
            if mgu is not None:
                print('    First goal unified with head of rule:', rule)
                print('      Renaming variables, head becomes:', head)
                print('      Mgu:',
                      ', '.join(f'{var} -> {mgu[var]}' for var in mgu)
                     )
                new_goals =\
                    deque(atom.rename_variables(variable_renaming)\
                              .substitute(mgu) for atom in rule.body
                         )
                new_goals.extend(goal.substitute_in_copy(mgu) for goal in goals
                                )
                if new_goals:
                    print('      New goals to solve: ', end='')
                    print(', '.join(str(goal) for goal in new_goals))
                    print('      New partial solution:',
                          ', '.join(f'{var} -> '
                                    f'{solution[var].substitute_in_copy(mgu)}'
                                            for var in solution
                                   )
                         )

                next_goals_solution_pairs.append(
                                   (new_goals,
                                    {var: solution[var].substitute_in_copy(mgu)
                                         for var in solution
                                    }
                                   )
                                                )
        if depth_first:
            goals_solution_pairs.extendleft(reversed(next_goals_solution_pairs)
                                           )
        else:
            goals_solution_pairs.extend(next_goals_solution_pairs)
```

[70]:
```
cat prolog_ex_1.pl
```

```
% Test queries:

% father(X, jack).
% X = bob.

% grandparent(john, X).
% X = jack ;
% X = sandra ;

father(bob, jack).
father(bob, sandra).
father(john, bob).
father(john, mary).
mother(jane, jack).
mother(jane, sandra).
mother(emily, bob).
mother(emily, mary).

parent(X, Y) :- father(X, Y).
parent(X, Y) :- mother(X, Y).
son(X, Y) :- parent(Y, X), male(X).
daughter(X, Y) :- parent(Y, X), female(X).
brother(X, Y) :- male(X), parent(Z, X), parent(Z, Y).
grandparent(X, Y) :- parent(X, Z), parent(Z, Y).
```

Tracing execution for a closed goal, so checking that the goal is a logical consequence of the logic program:

[71]:
```
logic_program = LogicProgram('prolog_ex_1.pl')
for _ in trace_solve(logic_program, 'grandparent(john, jack)'):
    pass
```

```
Dealing with following goals and partial solution:
    grandparent(john, jack)
    First goal unified with head of rule: grandparent(X, Y) :- parent(X, Z),
parent(Z, Y).
      Renaming variables, head becomes: grandparent(X, Y)
      Mgu: Y -> jack, X -> john
      New goals to solve: parent(john, Z), parent(Z, jack)
      New partial solution:

Dealing with following goals and partial solution:
    parent(john, Z), parent(Z, jack)
    First goal unified with head of rule: parent(X, Y) :- father(X, Y).
      Renaming variables, head becomes: parent(X, Y)
      Mgu: Z -> Y, X -> john
```

```
      New goals to solve: father(john, Y), parent(Y, jack)
      New partial solution:
    First goal unified with head of rule: parent(X, Y) :- mother(X, Y).
      Renaming variables, head becomes: parent(X, Y)
      Mgu: Z -> Y, X -> john
      New goals to solve: mother(john, Y), parent(Y, jack)
      New partial solution:

Dealing with following goals and partial solution:
    father(john, Y), parent(Y, jack)
    First goal unified with head of rule: father(john, bob).
      Renaming variables, head becomes: father(john, bob)
      Mgu: Y -> bob
      New goals to solve: parent(bob, jack)
      New partial solution:
    First goal unified with head of rule: father(john, mary).
      Renaming variables, head becomes: father(john, mary)
      Mgu: Y -> mary
      New goals to solve: parent(mary, jack)
      New partial solution:

Dealing with following goals and partial solution:
    parent(bob, jack)
    First goal unified with head of rule: parent(X, Y) :- father(X, Y).
      Renaming variables, head becomes: parent(X, Y)
      Mgu: Y -> jack, X -> bob
      New goals to solve: father(bob, jack)
      New partial solution:
    First goal unified with head of rule: parent(X, Y) :- mother(X, Y).
      Renaming variables, head becomes: parent(X, Y)
      Mgu: Y -> jack, X -> bob
      New goals to solve: mother(bob, jack)
      New partial solution:

Dealing with following goals and partial solution:
    father(bob, jack)
    First goal unified with head of rule: father(bob, jack).
      Renaming variables, head becomes: father(bob, jack)
      Mgu:

Dealing with following goals and partial solution:

    No goal left, solution is complete

Dealing with following goals and partial solution:
    mother(bob, jack)

Dealing with following goals and partial solution:
```

```
    parent(mary, jack)
    First goal unified with head of rule: parent(X, Y) :- father(X, Y).
      Renaming variables, head becomes: parent(X, Y)
      Mgu: Y -> jack, X -> mary
      New goals to solve: father(mary, jack)
      New partial solution:
    First goal unified with head of rule: parent(X, Y) :- mother(X, Y).
      Renaming variables, head becomes: parent(X, Y)
      Mgu: Y -> jack, X -> mary
      New goals to solve: mother(mary, jack)
      New partial solution:

Dealing with following goals and partial solution:
    father(mary, jack)

Dealing with following goals and partial solution:
    mother(mary, jack)

Dealing with following goals and partial solution:
    mother(john, Y), parent(Y, jack)
```

Tracing the depth-first search exploration for solutions to the goal `grandparent(john, X)` as illustrated with the first tree above:

[72]:
```
logic_program = LogicProgram('prolog_ex_1.pl')
for _ in trace_solve(logic_program, 'grandparent(john, X)'):
    pass
```

```
Dealing with following goals and partial solution:
    grandparent(john, X)    X -> X
    First goal unified with head of rule: grandparent(X, Y) :- parent(X, Z),
parent(Z, Y).
      Renaming variables, head becomes: grandparent(X_0, Y)
      Mgu: X -> Y, X_0 -> john
      New goals to solve: parent(john, Z), parent(Z, Y)
      New partial solution: X -> Y

Dealing with following goals and partial solution:
    parent(john, Z), parent(Z, Y)    X -> Y
    First goal unified with head of rule: parent(X, Y) :- father(X, Y).
      Renaming variables, head becomes: parent(X_0, Y_0)
      Mgu: Z -> Y_0, X_0 -> john
      New goals to solve: father(john, Y_0), parent(Y_0, Y)
      New partial solution: X -> Y
    First goal unified with head of rule: parent(X, Y) :- mother(X, Y).
      Renaming variables, head becomes: parent(X_0, Y_0)
      Mgu: Z -> Y_0, X_0 -> john
      New goals to solve: mother(john, Y_0), parent(Y_0, Y)
```

```
      New partial solution: X -> Y

Dealing with following goals and partial solution:
    father(john, Y_0), parent(Y_0, Y)    X -> Y
    First goal unified with head of rule: father(john, bob).
      Renaming variables, head becomes: father(john, bob)
      Mgu: Y_0 -> bob
      New goals to solve: parent(bob, Y)
      New partial solution: X -> Y
    First goal unified with head of rule: father(john, mary).
      Renaming variables, head becomes: father(john, mary)
      Mgu: Y_0 -> mary
      New goals to solve: parent(mary, Y)
      New partial solution: X -> Y

Dealing with following goals and partial solution:
    parent(bob, Y)    X -> Y
    First goal unified with head of rule: parent(X, Y) :- father(X, Y).
      Renaming variables, head becomes: parent(X_0, Y_0)
      Mgu: Y -> Y_0, X_0 -> bob
      New goals to solve: father(bob, Y_0)
      New partial solution: X -> Y_0
    First goal unified with head of rule: parent(X, Y) :- mother(X, Y).
      Renaming variables, head becomes: parent(X_0, Y_0)
      Mgu: Y -> Y_0, X_0 -> bob
      New goals to solve: mother(bob, Y_0)
      New partial solution: X -> Y_0

Dealing with following goals and partial solution:
    father(bob, Y_0)    X -> Y_0
    First goal unified with head of rule: father(bob, jack).
      Renaming variables, head becomes: father(bob, jack)
      Mgu: Y_0 -> jack
    First goal unified with head of rule: father(bob, sandra).
      Renaming variables, head becomes: father(bob, sandra)
      Mgu: Y_0 -> sandra

Dealing with following goals and partial solution:
        X -> jack
    No goal left, solution is complete

Dealing with following goals and partial solution:
        X -> sandra
    No goal left, solution is complete

Dealing with following goals and partial solution:
    mother(bob, Y_0)    X -> Y_0
```

```
Dealing with following goals and partial solution:
    parent(mary, Y)    X -> Y
    First goal unified with head of rule: parent(X, Y) :- father(X, Y).
      Renaming variables, head becomes: parent(X_0, Y_0)
      Mgu: Y -> Y_0, X_0 -> mary
      New goals to solve: father(mary, Y_0)
      New partial solution: X -> Y_0
    First goal unified with head of rule: parent(X, Y) :- mother(X, Y).
      Renaming variables, head becomes: parent(X_0, Y_0)
      Mgu: Y -> Y_0, X_0 -> mary
      New goals to solve: mother(mary, Y_0)
      New partial solution: X -> Y_0

Dealing with following goals and partial solution:
    father(mary, Y_0)    X -> Y_0

Dealing with following goals and partial solution:
    mother(mary, Y_0)    X -> Y_0

Dealing with following goals and partial solution:
    mother(john, Y_0), parent(Y_0, Y)    X -> Y
```

And the same but changing exploration to breadth-first:

[73]:
```python
logic_program = LogicProgram('prolog_ex_1.pl')
for _ in trace_solve(logic_program, 'grandparent(john, X)', False):
    pass
```

```
Dealing with following goals and partial solution:
    grandparent(john, X)    X -> X
    First goal unified with head of rule: grandparent(X, Y) :- parent(X, Z),
parent(Z, Y).
      Renaming variables, head becomes: grandparent(X_0, Y)
      Mgu: X -> Y, X_0 -> john
      New goals to solve: parent(john, Z), parent(Z, Y)
      New partial solution: X -> Y

Dealing with following goals and partial solution:
    parent(john, Z), parent(Z, Y)    X -> Y
    First goal unified with head of rule: parent(X, Y) :- father(X, Y).
      Renaming variables, head becomes: parent(X_0, Y_0)
      Mgu: Z -> Y_0, X_0 -> john
      New goals to solve: father(john, Y_0), parent(Y_0, Y)
      New partial solution: X -> Y
    First goal unified with head of rule: parent(X, Y) :- mother(X, Y).
      Renaming variables, head becomes: parent(X_0, Y_0)
      Mgu: Z -> Y_0, X_0 -> john
      New goals to solve: mother(john, Y_0), parent(Y_0, Y)
```

```
      New partial solution: X -> Y

Dealing with following goals and partial solution:
    father(john, Y_0), parent(Y_0, Y)    X -> Y
    First goal unified with head of rule: father(john, bob).
      Renaming variables, head becomes: father(john, bob)
      Mgu: Y_0 -> bob
      New goals to solve: parent(bob, Y)
      New partial solution: X -> Y
    First goal unified with head of rule: father(john, mary).
      Renaming variables, head becomes: father(john, mary)
      Mgu: Y_0 -> mary
      New goals to solve: parent(mary, Y)
      New partial solution: X -> Y

Dealing with following goals and partial solution:
    mother(john, Y_0), parent(Y_0, Y)    X -> Y

Dealing with following goals and partial solution:
    parent(bob, Y)    X -> Y
    First goal unified with head of rule: parent(X, Y) :- father(X, Y).
      Renaming variables, head becomes: parent(X_0, Y_0)
      Mgu: Y -> Y_0, X_0 -> bob
      New goals to solve: father(bob, Y_0)
      New partial solution: X -> Y_0
    First goal unified with head of rule: parent(X, Y) :- mother(X, Y).
      Renaming variables, head becomes: parent(X_0, Y_0)
      Mgu: Y -> Y_0, X_0 -> bob
      New goals to solve: mother(bob, Y_0)
      New partial solution: X -> Y_0

Dealing with following goals and partial solution:
    parent(mary, Y)    X -> Y
    First goal unified with head of rule: parent(X, Y) :- father(X, Y).
      Renaming variables, head becomes: parent(X_0, Y_0)
      Mgu: Y -> Y_0, X_0 -> mary
      New goals to solve: father(mary, Y_0)
      New partial solution: X -> Y_0
    First goal unified with head of rule: parent(X, Y) :- mother(X, Y).
      Renaming variables, head becomes: parent(X_0, Y_0)
      Mgu: Y -> Y_0, X_0 -> mary
      New goals to solve: mother(mary, Y_0)
      New partial solution: X -> Y_0

Dealing with following goals and partial solution:
    father(bob, Y_0)    X -> Y_0
    First goal unified with head of rule: father(bob, jack).
      Renaming variables, head becomes: father(bob, jack)
```

```
    Mgu: Y_0 -> jack
  First goal unified with head of rule: father(bob, sandra).
    Renaming variables, head becomes: father(bob, sandra)
    Mgu: Y_0 -> sandra

Dealing with following goals and partial solution:
    mother(bob, Y_0)    X -> Y_0

Dealing with following goals and partial solution:
    father(mary, Y_0)    X -> Y_0

Dealing with following goals and partial solution:
    mother(mary, Y_0)    X -> Y_0

Dealing with following goals and partial solution:
        X -> jack
    No goal left, solution is complete

Dealing with following goals and partial solution:
        X -> sandra
    No goal left, solution is complete
```

[74]:
```
cat prolog_ex_2.pl
```

```
join(e, X, X).
join(l(H, T), X, l(H, Y)) :- join(T, X, Y).
```

Tracing the depth-first search exploration for solutions to the goal `join(X, X, Y)` as illustrated with the third tree above yields the following. As there are infinitely many solutions, we use the `islice` class from the `itertools` module to trace the search for the first 3 solutions only:

[75]:
```
logic_program = LogicProgram('prolog_ex_2.pl')
for _ in islice(trace_solve(logic_program, 'join(X, X, Y)'), 3):
    pass
```

```
Dealing with following goals and partial solution:
    join(X, X, Y)    X -> X, Y -> Y
    First goal unified with head of rule: join(e, X, X).
      Renaming variables, head becomes: join(e, X_0, X_0)
      Mgu: Y -> X_0, X -> X_0, X_0 -> e
    First goal unified with head of rule: join(l(H, T), X, l(H, Y)) :- join(T,
X, Y).
      Renaming variables, head becomes: join(l(H, T), X_0, l(H, Y_0))
      Mgu: Y -> l(H, Y_0), X -> X_0, X_0 -> l(H, T)
      New goals to solve: join(T, l(H, T), Y_0)
      New partial solution: X -> l(H, T), Y -> l(H, Y_0)

Dealing with following goals and partial solution:
```

```
        X -> e, Y -> e
    No goal left, solution is complete

Dealing with following goals and partial solution:
    join(T, l(H, T), Y_0)    X -> l(H, T), Y -> l(H, Y_0)
    First goal unified with head of rule: join(e, X, X).
      Renaming variables, head becomes: join(e, X_0, X_0)
      Mgu: Y_0 -> X_0, X_0 -> l(H, T), T -> e
    First goal unified with head of rule: join(l(H, T), X, l(H, Y)) :- join(T,
X, Y).
      Renaming variables, head becomes: join(l(H_0, T_0), X_0, l(H_0, Y_1))
      Mgu: Y_0 -> l(H_0, Y_1), X_0 -> l(H, T), T -> l(H_0, T_0)
      New goals to solve: join(T_0, l(H, l(H_0, T_0)), Y_1)
      New partial solution: X -> l(H, l(H_0, T_0)), Y -> l(H, l(H_0, Y_1))

Dealing with following goals and partial solution:
        X -> l(H, e), Y -> l(H, l(H, e))
    No goal left, solution is complete

Dealing with following goals and partial solution:
    join(T_0, l(H, l(H_0, T_0)), Y_1)    X -> l(H, l(H_0, T_0)), Y -> l(H,
l(H_0, Y_1))
    First goal unified with head of rule: join(e, X, X).
      Renaming variables, head becomes: join(e, X_0, X_0)
      Mgu: Y_1 -> X_0, X_0 -> l(H, l(H_0, T_0)), T_0 -> e
    First goal unified with head of rule: join(l(H, T), X, l(H, Y)) :- join(T,
X, Y).
      Renaming variables, head becomes: join(l(H_1, T), X_0, l(H_1, Y_0))
      Mgu: Y_1 -> l(H_1, Y_0), X_0 -> l(H, l(H_0, T_0)), T_0 -> l(H_1, T)
      New goals to solve: join(T, l(H, l(H_0, l(H_1, T))), Y_0)
      New partial solution: X -> l(H, l(H_0, l(H_1, T))), Y -> l(H, l(H_0,
l(H_1, Y_0)))

Dealing with following goals and partial solution:
        X -> l(H, l(H_0, e)), Y -> l(H, l(H_0, l(H, l(H_0, e))))
    No goal left, solution is complete
```

The implementation of the Prolog interpreter in `LogicProgram` should now be clear. The only addition to the code of the tracing function is that we check that the goals to solve are built from predicate and function symbols that all occur in the logic program:

[76]:
```python
class LogicProgram(LogicProgram):
    class QueryError(Exception):
        pass

    def solve(self, query, depth_first=True):
        query = Conjunction.parse_conjunction(query)
        if any(predicate_symbol not in self.predicate_symbols
```

```
               or query.predicate_symbols[predicate_symbol]
                  != self.predicate_symbols[predicate_symbol]
                      for predicate_symbol in query.predicate_symbols
              ):
            raise LogicProgram.QueryError(f'Predicate symbol in {query} '
                                          'not in program'
                                         )
        if any(function_symbol not in self.function_symbols
               or query.function_symbols[function_symbol]
                  != self.function_symbols[function_symbol]
                      for function_symbol in query.function_symbols
              ):
            raise LogicProgram.QueryError(f'Function symbol in {query} '
                                           'not in program'
                                          )
        query_variables = {var for atom in query for var in atom.variables()}
        # A list of pairs consisting of:
        # - a list of goals to be solved, and
        # - the substitution to apply to the variables that occur in the
        #   query as determined by the unifications computed so far.
        goals_solution_pairs = deque([(deque(query),
                                       {var: Term(var)
                                            for var in query_variables
                                       }
                                      )
                                     ]
                                    )
        while goals_solution_pairs:
            goals, solution = goals_solution_pairs.popleft()
            if not goals:
                yield {var: str(solution[var]) for var in solution}
                continue
            reserved_variables = query_variables\
                                 | {var for atom in goals
                                            for var in atom.variables()
                                   }
            goal = goals.popleft()
            next_goals_solution_pairs = deque()
            for rule in self.program:
                variable_renaming =\
                        Expression.fresh_variables(rule.variables,
                                                   reserved_variables
                                                  )
                head = rule.head.rename_variables(variable_renaming)
                mgu = goal.unify(head)
                if mgu is not None:
                    new_goals = deque(atom.rename_variables(variable_renaming)
```

```
                                    .substitute(mgu) for atom in rule.body
                                )
                    new_goals.extend(goal.substitute_in_copy(mgu)
                                          for goal in goals
                                    )
                    next_goals_solution_pairs.append(
                                    (new_goals,
                                     {var: solution[var].substitute_in_copy(mgu)
                                          for var in solution
                                     }
                                    )
                                                     )
            if depth_first:
                goals_solution_pairs.extendleft(
                                        reversed(next_goals_solution_pairs)
                                            )
            else:
                goals_solution_pairs.extend(next_goals_solution_pairs)
```

[77]:
```
LP = LogicProgram('prolog_ex_1.pl')
for solution in LP.solve('grandparent(john, X)'):
        print('    ', solution)
```

```
    {'X': 'jack'}
    {'X': 'sandra'}
```