Important notes about grading:

1. **Compiler errors:** All code you submit must compile. Programs that do not compile will receive an automatic zero. If you run out of time, it is better to comment out the parts that do not compile, than hand in a more complete file that does not compile.
2. **Late assignments:** You must submit your code before the deadline. Verify on Sakai that you have submitted the correct version. If you submit the incorrect version before the deadline and realize that you have done so after the deadline, we will only grade the version received before the deadline.

A Prolog interpreter

In this project, you will implement a Prolog interpreter in **OCaml**.

If you want to implement the project in **Python**, download the source code here and follow the README file. Parsing functions and test-cases are provided.

Pseudocode

Your main task is to implement the `non-deterministic` abstract interpreter covered in the lecture `Control in Prolog`. The **pseudocode** of the abstract interpreter is in the lecture note.

Bonus

There is also a bonus task for implementing a `deterministic` Prolog interpreter with support for backtracking (recover from bad choices) and choice points (produce multiple results). Please refer to the lecture notes `Programming with Lists` and `Control in Prolog` for more details about this algorithm. **Bonus will only be added to projects implmented in OCaml.**

How to Start (OCaml):

1. Download the source code here.
2. You will need to implement your code in `final/src/final.ml`. We give detailed instructions on the functions that you need to implement below. You only need to implement functions with `raise (Failure "Not implemented")`. Delete this line after implementation is done.
3. The test-cases (also seen below) are provided in `final/test/public.ml` (if using `Dune`) and `final/main.ml` (if using `Make`).
4. To compile and run your implementation, we provide two options:
   Option (1): Using `Dune`: type `dune runtest -f` in the command-line window under `final` directory.
   Option (2): Using `Make`: First, to compile the code, type `make` in the command-line window under `final` directory. Second, to run the code, type `./main.byte` in the command-line window under `final` directory.

Randomness

Because randomness is the key to the success of the non-deterministic abstract interpreter (please see the lecture note `Control in Prolog` to find why), we first initialize the randomness generator with a random seed chosen in a system-dependent way.

In [ ]:

```
open Stdlib

let _  = Random.self_init ()
```

Abstract Syntax Tree

To implement the abstract interpreter, we will start with the definition of prolog terms (i.e. abstract syntax tree).

In [ ]:

```
type term =
  | Constant of string                    (* Prolog constants, e.g., rickard, ned, robb, a, b, ... *
)
```

```
  | Variable of string                    (* Prolog variables, e.g., X, Y, Z, ... *)
  | Function of string * term list        (* Prolog functions, e.g., append(X, Y, Z), father(rickarc
, ned),
                                           ancestor(Z, Y), cons (H, T) abbreviated as [H|T] in SWI-
olog, ...
                                           The first component of Function is the name of the funct
n,
                                           whose parameters are in a term list, the second componen
of Function.*)
```

A Prolog `program` consists of `clauses` and `goals`. A `clause` can be either a `fact` or a `rule`.

A Prolog `rule` is in the shape of `head :- body`. For example,

```
ancestor(X, Y) :- father(X, Z), ancestor(Z, Y).
```

In the above rule (also called a clause), `ancestor(X, Y)` is the `head` of the rule, which is a Prolog `Function` defined in the type `term` above. The rule's `body` is a list of terms: `father(X, Z)` and `ancestor(Z, Y)`. Both are Prolog `Function`s.

A Prolog `fact` is simply a `Function` defined in the type `term`. For example,

```
father(rickard, ned).
```

In the above fact (also a clause), we say rickard is ned's father.

A Prolog goal (also called a query) is a list of Prolog `Function`s, which is typed as a list of terms. For example,

```
?- ancestor(rickard, robb), ancestor(X, robb).
```

In the above `goal`, we are interested in two queries which are Prolog `Function`s: `ancestor(rickard, robb)` and `ancestor(X, robb)`.

In [ ]:

```
type head = term            (* The head of a rule is a Prolog Function *)
type body = term list       (* The body of a rule is a list of Prolog Functions *)

type clause = Fact of head | Rule of head * body (* A rule is in the shape head :- body *)

type program = clause list (* A program consists of a list of clauses in which we have either rule
s or facts. *)

type goal = term list       (* A goal is a query that consists of a few functions *)
```

The following stringification functions should help you debug the interpreter.

In [ ]:

```
let rec string_of_f_list f tl =
  let _, s = List.fold_left (fun (first, s) t ->
    let prefix = if first then "" else s ^ ", " in
    false, prefix ^ (f t)) (true,"") tl
  in
  s

(* This function converts a Prolog term (e.g. a constant, variable or function) to a string *)
let rec string_of_term t =
  match t with
  | Constant c -> c
  | Variable v -> v
  | Function (f,tl) ->
      f ^ "(" ^ (string_of_f_list string_of_term tl) ^ ")"

(* This function converts a list of Prolog terms, separated by commas, to a string *)
let string_of_term_list fl =
  string_of_f_list string_of_term fl

(* This function converts a Prolog goal (query) to a string *)
let string_of_goal g =
  "?- " ^ (string_of_term_list g)

(* This function converts a Prolog clause (e.g. a rule or a fact) to a string *)
```

```
let string_of_clause c =
  match c with
  | Fact f -> string_of_term f ^ "."
  | Rule (h,b) -> string_of_term h ^ " :- " ^ (string_of_term_list b) ^ "."

(* This function converts a Prolog program (a list of clauses), separated by \n, to a string *)
let string_of_program p =
  let rec loop p acc =
    match p with
    | [] -> acc
    | [c] -> acc ^ (string_of_clause c)
    | c::t ->  loop t (acc ^ (string_of_clause c) ^ "\n")
  in loop p ""
```

Below are more helper functions for you to easily construct a Prolog program using the defined data types:

In [ ]:

```
let var v = Variable v           (* helper function to construct a Prolog Variable *)
let const c = Constant c         (* helper function to construct a Prolog Constant *)
let func f l = Function (f,l)    (* helper function to construct a Prolog Function *)
let fact f = Fact f              (* helper function to construct a Prolog Fact *)
let rule h b = Rule (h,b)        (* helper function to construct a Prolog Rule *)
```

Problem 1

In this problem, you will implement the function:

```
occurs_check : term -> term -> bool
```

`occurs_check v t` returns `true` if the Prolog `Variable v` occurs in `t`. Please see the lecture note `Control in Prolog` to revisit the concept of occurs-check.

Hint: You don't need to use pattern matching to deconstruct `v` (type `term`) to compare the string value of `v` with that of another variable. You can compare two `Variable`s via structural equality, e.g. `Variable "s" = Variable "s"`. Therefore, **if** `t` is a variable, it suffices to use `v = t` to make the equality checking. Note that you should use `=` rather than `==` for testing structural equality.

In [ ]:

```
let rec occurs_check v t =
  (* YOUR CODE HERE *)
   raise (Failure "Not implemented")
```

Examples and test-cases.

In [ ]:

```
assert (occurs_check (var "X") (var "X"))
assert (not (occurs_check (var "X") (var "Y")))
assert (occurs_check (var "X") (func "f" [var "X"]))
assert (occurs_check (var "E") (func "cons" [const "a"; const "b"; var "E"]))
```

The last test-case above was taken from the example we used to illustrate the occurs-check problem in the lecture note `Control in Prolog`.

```
?- append([], E, [a,b | E]).
```

Here the `occurs_check` function asks whether the `Variable` E appears on the `Function` term `func "cons" [const "a"; const "b"; var "E"]`. The return value should be `true`.

Problem 2

Implement the following functions which return the variables contained in a term or a clause (e.g. a rule or a fact):

```
variables of term      : term -> VarSet.t
```

```
variables_of_clause    : clause -> VarSet.t
```

The result must be saved in a data structure of type `VarSet` that is instantiated from OCaml Set module. The type of each element (a Prolog `Variable`) in the set is `term` as defined above (`VarSet.t = term`).

You may want to use `VarSet.singleton t` to return a singletion set containing only one element `t`, use `VarSet.empty` to represent an empty variable set, and use `VarSet.union t1 t2` to merge two variable sets `t1` and `t2`.

In [ ]:

```
module VarSet = Set.Make(struct type t = term let compare = Stdlib.compare end)
(* API Docs for Set : https://caml.inria.fr/pub/docs/manual-ocaml/libref/Set.S.html *)

let rec variables_of_term t =
  (* YOUR CODE HERE *)
   raise (Failure "Not implemented")

let variables_of_clause c =
  (* YOUR CODE HERE *)
   raise (Failure "Not implemented")
```

Examples and test-cases:

In [ ]:

```
(* The variables in a function f (X, Y, a) is [X; Y] *)
assert (VarSet.equal (variables_of_term (func "f" [var "X"; var "Y"; const "a"]))
          (VarSet.of_list [var "X"; var "Y"]))
(* The variables in a Prolog fact p (X, Y, a) is [X; Y] *)
assert (VarSet.equal (variables_of_clause (fact (func "p" [var "X"; var "Y"; const "a"])))
          (VarSet.of_list [var "X"; var "Y"]))
(* The variables in a Prolog rule p (X, Y, a) :- q (a, b, a) is [X; Y] *)
assert (VarSet.equal (variables_of_clause (rule (func "p" [var "X"; var "Y"; const "a"])
          [func "q" [const "a"; const "b"; const "a"]]))
          (VarSet.of_list [var "X"; var "Y"]))
```

Problem 3

The value of type `term Substitution.t` is a OCaml map whose key is of type `term` and value is of type `term`. It is a map from variables to terms. Implement substitution functions that takes a `term Substitution.t` and uses that to perform the substitution:

```
substitute_in_term : term Substitution.t -> term -> term
substitute_in_clause : term Substitution.t -> clause -> clause
```

See the lecture note `Control in Prolog` to revisit the concept of substitution. For example, $\sigma = \{X/a, Y/Z, Z/f(a,b)\}$ is substitution. It is a map from variables `X, Y, Z` to terms `a, Z, f(a,b)`. Given a term $E = f(X, Y, Z)$, the substitution $E\sigma$ is $f(a, Z, f(a, b))$.

You may want to use the `Substitution.find_opt t s` function. The function takes a term (variable) `t` and a substitution map `s` as input and returns `None` if `t` is not a key in the map `s` or otherwise returns `Some t'` where `t' = s[t]`.

In [ ]:

```
module Substitution = Map.Make(struct type t = term let compare = Stdlib.compare end)
(* See API docs for OCaml Map: https://caml.inria.fr/pub/docs/manual-ocaml/libref/Map.S.html *)

(* This funtion converts a substitution to a string (for debugging) *)
let string_of_substitution s =
  "{" ^ (
    Substitution.fold (
      fun v t s ->
        match v with
        | Variable v -> s ^ "; " ^ v ^ " -> " ^ (string_of_term t)
        | Constant _ -> assert false (* Note: substitution maps a variable to a term! *)
        | Function _ -> assert false (* Note: substitution maps a variable to a term! *)
    ) s ""
  ) ^ "}"
```

```
let rec substitute_in_term s t =
  (* YOUR CODE HERE *)
   raise (Failure "Not implemented")

let substitute_in_clause s c =
  (* YOUR CODE HERE *)
   raise (Failure "Not implemented")
```

Examples and test-cases:

In [ ]:

```
(* We create a substitution map s = [Y/0, X/Y] *)
let s =
  Substitution.add (var "Y") (const "0") (Substitution.add (var "X") (var "Y") Substitution.empty)

(* Function substitution - f (X, Y, a) [Y/0, X/Y] = f (Y, 0, a) *)
assert (substitute_in_term s (func "f" [var "X"; var "Y"; const "a"]) =
        func "f" [var "Y"; const "0"; const "a"])
(* Fact substitution - p (X, Y, a) [Y/0, X/Y] = p (Y, 0, a) *)
assert (substitute_in_clause s (fact (func "p" [var "X"; var "Y"; const "a"])) =
        (fact (func "p" [var "Y"; const "0"; const "a"])))
(* Given a Prolog rule, p (X, Y, a) :- q (a, b, a), after doing substitution [Y/0, X/Y],
              we have p (Y, 0, a) :- q (a, b, a) *)
assert (substitute_in_clause s (rule (func "p" [var "X"; var "Y"; const "a"]) [func "q" [const "a";
const "b"; const "a"]]) =
        (rule (func "p" [var "Y"; const "0"; const "a"]) [func "q" [const "a"; const "b"; const "a"]
]))
```

Problem 4

Implementing the function:

```
unify : term -> term -> term Substitution.t
```

The function returns a unifier of the given terms. The function should raise the exception `raise Not_unfifiable`, if the given terms are not unifiable (`Not_unfifiable` is a declared exception in `final.ml`). You may find the **pseudocode** of `unify` in the lecture note `Control in Prolog` useful.

As in problem 3, you will need to use `module Substitution` because unification is driven by subsititution. (*See API docs for OCaml Map: https://caml.inria.fr/pub/docs/manual-ocaml/libref/Map.S.html*)

You may want to use `substitute_in_term` to implement the first and second lines of the pseudocode.

You may want to use `Substitution.empty` for an empty subsitution map, `Substitution.singleton v t` for creating a new subsitution map that contains a single substitution `[v/t]`, and `Substitution.add v t s` to add a new substitution `[v/t]` to an existing substitution map `s` (this function may help implement the $\cup$ operation in the pseudocode). You may also want to use `Substitution.map` for the θ `[X/Y]` operation in the pseudocode. This operation applies the subsitition `[X/Y]` to each term in the *values* of the substitution map θ (the *keys* of the substitution map θ remain unchanged). `Subsitution.map f s` returns a new substitution map with the same keys as the input substitution map `s`, where the associated term `t` for each key of `s` has been replaced by the result of the application of `f` to `t`. Use `List.fold_left2` to implement the `fold_left` function in the pseudocode (https://caml.inria.fr/pub/docs/manual-ocaml/libref/List.html).

In [ ]:

```
exception Not_unifiable

let unify t1 t2 =
  (* YOUR CODE HERE *)
   raise (Failure "Not implemented")
```

Examples and test-cases:

In [ ]:

```
(* A substitution that can unify variable X and variable Y: {X->Y} or {Y->X} *)
```

```
assert ((unify (var "X") (var "Y")) = Substitution.singleton (var "Y") (var "X") ||
          (unify (var "X") (var "Y")) = Substitution.singleton (var "X") (var "Y"))
(* A substitution that can unify variable Y and variable X: {X->Y} pr {Y->X} *)
assert ((unify (var "Y") (var "X")) = Substitution.singleton (var "X") (var "Y") ||
          (unify (var "Y") (var "X")) = Substitution.singleton (var "Y") (var "X"))
(* A substitution that can unify variable Y and variable Y: empty set *)
assert (unify (var "Y") (var "Y") = Substitution.empty)
(* A substitution that can unify constant 0 and constant 0: empty set *)
assert (unify (const "0") (const "0") = Substitution.empty)
(* A substitution that can unify constant 0 and variable Y: {Y->0} *)
assert (unify (const "0") (var "Y") = Substitution.singleton (var "Y") (const "0"))
(* Cannot unify two distinct constants *)
assert (
  match unify (const "0") (const "1") with
  | _ -> false
  | exception Not_unifiable -> true)
(* Cannot unify two functions with distinct function symbols *)
assert (
  match unify (func "f" [const "0"]) (func "g" [const "1"]) with
  | _ -> false
  | exception Not_unifiable -> true)
(* A substitution that can unify function f(X) and function f(Y): {X->Y} or {Y->X} *)
assert (unify (func "f" [var "X"]) (func "f" [var "Y"]) = Substitution.singleton (var "X") (var "Y"
) ||
        unify (func "f" [var "X"]) (func "f" [var "Y"]) = Substitution.singleton (var "Y") (var "X"
))
```

In [ ]:

```
(* A substitution that can unify function f(X,Y,Y) and function f(Y,Z,a): {X->a;Y->a;Z->a} *)
let t1 = Function("f", [Variable "X"; Variable "Y"; Variable "Y"])
let t2 = Function("f", [Variable "Y"; Variable "Z"; Constant "a"])
let u = unify t1 t2
let _ = assert (string_of_substitution u = "{; X -> a; Y -> a; Z -> a}")
```

Problem 5

We first define a function `freshen` that given a clause, returns a clause where are the variables have been renamed with fresh variables. This function will be used for the clause **renaming** operation in the implementation of `nondet_query`.

In [ ]:

```
let counter = ref 0
let fresh () =
  let c = !counter in
  counter := !counter + 1;
  Variable ("_G" ^ string_of_int c)

let freshen c =
  let vars = variables_of_clause c in
  let s = VarSet.fold (fun v s -> Substitution.add v (fresh()) s) vars Substitution.empty in
  substitute_in_clause s c
```

For example,

In [ ]:

```
let c = (rule (func "p" [var "X"; var "Y"; const "a"]) [func "q" [var "X"; const "b"; const "a"]])
(* The string representation of a rule c is p(X, Y, a) :- q(X, b, a). *)
let _ = print_endline (string_of_clause c)
(* After renaming, the string representation is p(_G0, _G1, a) :- q(_G0, b, a).
   X is replaced by _G0 and Y is replaced by _G1. *)
let _ = print_endline (string_of_clause (freshen c))
```

The main task of problem 5 is to implement a nondeterministic Prolog interpreter.

Following the **pseudocode** `Abstract interpreter` in the lecture note `Control in Prolog` to implement the interpreter:

```
nondet_query : clause list -> term list -> term list
```

where

- the first argument is the `program` which is a list of clauses (rules and facts).
- the second argument is the `goal` which is a list of terms.

The function returns a list of terms (`results`), which is an instance of the original `goal` and is a logical consequence of the `program` . See tests cases for examples.

Please follow the **pseudocode**, which means you need to have two recursive functions in your implementation of `nondet_query` . For the goal order, choose randomly; for the rule order, choose randomly from the facts that **can be unified with the chosen goal** and the rules that **whose head can be unified with the chosen goal** . Please see the lecture note `Control in Prolog` to find why. You may find the function `Random.int n` useful. This function can randomly return an integer within `[0, n)` .

In [ ]:

```
let nondet_query program goal =
  (* YOUR CODE HERE *)
   raise (Failure "Not implemented")
```

Examples and Test-cases:

(1) Our first example is the House Stark program (lecture note `Prolog Basics`).

In [ ]:

```
(* Define the House Stark program:

father(rickard, ned).
father(ned, robb).
ancestor(X, Y) :- father(X, Y).
ancestor(X, Y) :- father(X, Z), ancestor(Z, Y).

*)
let ancestor x y = func "ancestor" [x;y]
let father x y = func "father" [x;y]
let father_consts x y =  father (Constant x) (Constant y)
let f1 = Fact (father_consts "rickard" "ned")
let f2 = Fact (father_consts "ned" "robb")
let r1 = Rule (ancestor (var "X") (var "Y"), [father (var "X") (var "Y")])
let r2 = Rule (ancestor (var "X") (var "Y"), [father (var "X") (var "Z"); ancestor (var "Z") (var "Y
")])

let pstark = [f1;f2;r1;r2]
let _ = print_endline ("Program:\n" ^ (string_of_program pstark))

(* Define a goal (query):

?- ancestor(rickard, robb)

The solution is the query itself.
*)
let g = [ancestor (const "rickard") (const "robb")]
let _ = print_endline ("Goal:\n" ^ (string_of_goal g))
let g' = nondet_query pstark g
let _ = print_endline ("Solution:\n" ^ (string_of_goal g'))
let _ = assert (g' = [ancestor (const "rickard") (const "robb")])

(* Define a goal (query):

?- ancestor(X, robb)

The solution can be either ancestor(ned, robb) or ancestor(rickard, robb)
*)
let g = [ancestor (var "X") (const "robb")]
let _ = print_endline ("Goal:\n" ^ (string_of_goal g))
let g' = nondet_query pstark g
let _ = print_endline ("Solution:\n" ^ (string_of_goal g') ^ "\n")
let _ = assert (g' = [ancestor (const "ned") (const "robb")] ||
                g' = [ancestor (const "rickard") (const "robb")])
```

(2) Our second example is the list append program (lecture note `Programming with Lists`).

In [ ]:

```
(* Define the list append program:

append(nil, Q, Q).
append(cons(H, P), Q, cons(H, R)) :- append(P, Q, R).

*)
let nil = const "nil"
let cons h t = func "cons" [h;t]
let append x y z = func "append" [x;y;z]
let c1 = fact @@ append nil (var "Q") (var "Q")
let c2 = rule (append (cons (var "H") (var "P")) (var "Q") (cons (var "H") (var "R")))
              [append (var "P") (var "Q") (var "R")]
let pappend = [c1;c2]
let _ = print_endline ("Program:\n" ^ (string_of_program pappend))


(* Define a goal (query):

?- append(X, Y, cons(1, cons(2, cons(3, nil))))

The solution can be any of the following four:
Solution 1: ?- append(nil, cons(1, cons(2, cons(3, nil))), cons(1, cons(2, cons(3, nil))))
Solution 2: ?- append(cons(1, nil), cons(2, cons(3, nil)), cons(1, cons(2, cons(3, nil))))
Solution 3: ?- append(cons(1, cons(2, nil)), cons(3, nil), cons(1, cons(2, cons(3, nil))))
Solution 4: ?- append(cons(1, cons(2, cons(3, nil))), nil, cons(1, cons(2, cons(3, nil))))
*)
let g = [append (var "X") (var "Y") (cons (const "1") (cons (const "2") (cons (const "3") nil)))]
let _ = print_endline ("Goal:\n" ^ (string_of_goal g))
let g' = nondet_query pappend g
let _ = print_endline ("Solution:\n" ^ (string_of_goal g') ^ "\n")
let _ = assert (
g' = [append nil (cons (const "1") (cons (const "2") (cons (const "3") nil))) (cons (const "1")
(cons (const "2") (cons (const "3") nil)))] ||
g' = [append (cons (const "1") nil) (cons (const "2") (cons (const "3") nil)) (cons (const "1")
(cons (const "2") (cons (const "3") nil)))] ||
g' = [append (cons (const "1") (cons (const "2") nil)) (cons (const "3") nil) (cons (const "1")
(cons (const "2") (cons (const "3") nil)))] ||
g' = [append (cons (const "1") (cons (const "2") (cons (const "3") nil))) nil (cons (const "1")
(cons (const "2") (cons (const "3") nil)))] )
```

The main problem of this non-deterministic abstract interpreter is that it cannot efficiently find all solutions, as the above two examples show. Let's fix this problem by implementing a deterministic interpreter similar to that used in SWI-Prolog.

Bonus Problem (16 lines of code)

Implement the function:

```
query : clause list -> term list -> term list list
```

where

- the first argument is the `program` which is a list of clauses.
- the second argument is the `goal` which is a list of terms.

The function returns a list of term lists (`results`). Each of these results is a instance of the original `goal` and is a logical consequence of the `program` . If the given `goal` is not a logical consequence of the `program` , then the result is an empty list. See tests cases for expected results.

For the rule and goal order, choose what Prolog does; choose the left-most subgoal for goal order and first rule in the order in which the rules appear in the program for the rule order.

Hint: Implement a purely functional recursive solution. The backtracking and choice points naturally fall out of the implementation. The reference solution is 16 lines long.

In [ ]:

```
let query ?(limit=10) program goal =
  (* YOUR CODE HERE *)
   raise (Failure "Not implemented")
```

Examples and Test-cases:

(1) Our first example is the House Stark program (lecture note `Prolog Basics`).

In [ ]:

```
(* Define the same goal (query) as above:

?- ancestor(X, robb)

The solution this time should include both ancestor(ned, robb) and ancestor(rickard, robb)
*)
let g = [ancestor (var "X") (const "robb")]
let _ = print_endline ("Goal:\n" ^ (string_of_goal g))
let g1,g2 = match det_query pstark g with [v1;v2] -> v1,v2 | _ -> failwith "error"
let _ = print_endline ("Solution:\n" ^ (string_of_goal g1))
let _ = print_endline ("Solution:\n" ^ (string_of_goal g2) ^ "\n")
let _ = assert (g1 = [ancestor (const "ned") (const "robb")])
let _ = assert (g2 = [ancestor (const "rickard") (const "robb")])
```

(2) Our second example is the list append program (lecture note `Programming with Lists`).

In [ ]:

```
(* Define the same goal (query) as above:

?- append(X, Y, cons(1, cons(2, cons(3, nil))))

The solution this time should include all of the following four:
Solution 1: ?- append(nil, cons(1, cons(2, cons(3, nil))), cons(1, cons(2, cons(3, nil))))
Solution 2: ?- append(cons(1, nil), cons(2, cons(3, nil)), cons(1, cons(2, cons(3, nil))))
Solution 3: ?- append(cons(1, cons(2, nil)), cons(3, nil), cons(1, cons(2, cons(3, nil))))
Solution 4: ?- append(cons(1, cons(2, cons(3, nil))), nil, cons(1, cons(2, cons(3, nil))))
*)
let g = [append (var "X") (var "Y") (cons (const "1") (cons (const "2") (cons (const "3") nil)))]
let _ = print_endline ("Goal:\n" ^ (string_of_goal g))
let g' = det_query pappend g
let _ = assert (List.length g' = 4)
let _ = List.iter (fun g -> print_endline ("Solution:\n" ^ (string_of_goal g))) g'
```