A Prolog interpreter ¶

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

Compile and run your code:

OCaml

After downloading and unzipping the file for the final project, in your command-line window, `cd` into the directory which contains `src`. You should write your code in `src/final.ml`. The test-cases (also seen below) are provided in `src/main.ml`.

Use `make` to compile the code. And use `./final` to excute it.

Python

After downloading and unzipping the file for the final project, in your command-line window, `cd` into the directory which contains `src`. You should write your code in `src/final.py`. The test-cases (also seen below) are provided in `src/main.py`. Data structures are provided in `src/prolog_structures.py`.

Assignment Project Exam Help

Use `python3 src/main.py` to execute your code. Please do not use `python2`.

https://powcoder.com

Submission:

Please submit to Canvas `final.ml` or `final.py` only.

Add WeChat powcoder

Important notes about grading:

Programs that cannot be compiled will receive an automatic zero. If you are having trouble getting your assignment to compile, please visit recitation or office hours.

- Problem 1, Problem 2: **4** points each
- Problem 3, Problem 4: **11** points each
- Challenge problem: **5** points

Pseudocode

The **pseudocode** of Problem 3 and 4 is given in the lecture note `Control in Prolog`.

Abstract Syntax Tree

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

OCaml Definition

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(rick
ard, ned),
                                        ancestor(Z, Y), cons (H, T) abbreviated as [H|T] in SWI-
Prolog, ...
                                        The first component of Function is the name of the funct
ion,
                                        the second component of Function is its parameters defin
ed as a term list.*)
```

Assignment Project Exam Help

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

https://powcoder.com

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.

Add WeChat powcoder

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

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

In the above fact (also as 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, e.g.,

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

In the above `goal`, we are interested in two queries: `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 r
ules 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 ""
```

Assignment Project Exam Help
https://powcoder.com
Add WeChat powcoder

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 *)
```

Python Definition

The idea is similar to the OCaml definition above. Prolog abstract syntax tree is defined in `src/prolog_structures.py` . Also see the comment below.

Note that in the python implementation, a `Rule` class is used to construct both a rule and a fact. If the underlying instance is a fact, the body of the rule `RuleBody` owns an empty term list.

In [ ]:

```
(* Every clause in a prolog program is encoded as a rule
   A fact is a rule with empty body *)
class Rule:
    (* head is a function *)
    (* body is a *list* of functions (see RuleBody) *)
    def __init__(self, head, body):
        assert isinstance(body, RuleBody)
        self.head = head
        self.body = body

(* Rule body is a list of functions (terms). *)
class RuleBody:
    def __init__(self, terms):
        assert isinstance(terms, list)
        self.terms = terms

(* The super-class for all Prolog terms *)
class Term:
    pass

(* A function is, for example, father(rickard, ned). *)
class Function(Term):
    def __init__(self, relation, terms):
        self.relation = relation (* function name, e.g. father *)
        self.terms = terms       (* funcion parameters, e.g. rickard, ned *)

(* Prolog Variables, e.g. X, Y, ... *)
class Variable(Term):
    def __init__(self, value):
        self.value = value

class Constant(Term):
    def __init__(self, value):
        self.value = value

(* Prolog Atoms, e.g. rickard, ned, ... *)
class Atom(Constant):
    def __init__(self, value):
        super().__init__(value)

(* Prolog Numbers, e.g. 1, 2, ... *)
class Number(Constant):
    def __init__(self, value):
        super().__init__(int(value))
```

Assignment Project Exam Help
https://powcoder.com
Add WeChat powcoder

Occurs_check

In this example, we 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.

OCaml Implementation

In [ ]:

```
let rec occurs_check v t =
  (* BEGIN SOLUTION *)
  match t with
  | Variable _ -> t = v
  | Constant _ -> false
  | Function (_,l) -> List.fold_left (fun acc t' -> acc || occurs_check v t') false l
  (* END SOLUTION *)
```

Note that in this implementation, when `t` is a variable, we don't need to use pattern matching to deconstruct `v` (type `term`) to compare the string value of `v` with that of `t`. We can compare two `Variable` s via structural equality, e.g. `Variable "s" = Variable "s"`. Therefore, if `t` is a variable, it suffices to use `t = v` for equality checking (one should use `=` rather than `==` for structural equality).

Assignment Project Exam Help
https://powcoder.com
Add WeChat powcoder

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`.

Python Implementation

The idea is similar to the above OCaml implementation.

In [ ]:

```
def occurs_check (v : Variable, t : Term) -> bool:
    if isinstance(t, Variable):     (* If t is a Variable instance *)
        return v == t
    elif isinstance(t, Function):  (* If t is a Function instance *)
        for t in t.terms:
            if occurs_check(v, t): (* If v occurs in a function paramter *)
                return True
        return False
    return False                    (* Otherwise t is an instance of Atom or Number *)
```

Problem 1 (4 points)

Implement the following functions which return the variables contained in a term or a clause.

OCaml Implementation

```
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`).

Assignment Project Exam Help

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

https://powcoder.com

Add WeChat powcoder

In [ ]:

```
module VarSet = Set.Make(struct type t = term let compare = Pervasives.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 *)

let variables_of_clause c =
  (* YOUR CODE HERE *)
```

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"]))
```

Python Implementation

The idea is similar to the OCaml implementation.

```
def variables_of_term (t : Term) -> set :

def variables_of_clause (c : Rule) -> set :
```

The function should return the `Variables` contained in a term or a rule using Python `set`.

The result must be saved in a Python `set`. The type of each element (a Prolog Variable) in the `set` is `Variable`.

You may want to merge two python sets `s1` and `s2` by `s1.union(s2)`, create a singleton set containing only one element `t` by `set([t])`, and construct an empty set by `set()`. Note that the `union` function returns a new set with elements from `s_1` and `s_2` and `s_1` itself is unchanged.

In [ ]:

```
def variables_of_term (t : Term) -> set :
    (* YOUR CODE HERE *)

def variables_of_clause (c : Rule) -> set :
    (* YOUR CODE HERE *)
```

Assignment Project Exam Help

https://powcoder.com

Problem 2 (4 points)

The goal of this problem is to implement substitution. Review 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 $f(a, Z, f(a, b))$.

Add WeChat powcoder

OCaml Implementation

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

where a value of type `term Substitution.t` is an OCaml map whose keys are of type `term` and values are of type `term`. It is a map from variables to terms.

You may want to use the `Substitution.find_opt t s` function. This function takes a term (that must be a 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 = Pervasives.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 *)

let substitute_in_clause s c =
  (* YOUR CODE HERE *)
```

Examples and test-cases:

Assignment Project Exam Help

In [ ]:

https://powcoder.com

Add WeChat powcoder

```
(* 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"]]))
```

Python Implementation:

The idea is similiar to the OCaml implementation. You must use a Python dictionary to represent a subsititution map.

```
def substitute_in_term (s : dict, t : Term) -> Term

def substitute_in_clause (s : dict, c : Rule) -> Rule
```

The value of type `dict` should be a Python dictionary whose keys are of type `Variable` and values are of type `Term`. It is a map from variables to terms.

The function should return `t_` obtained by applying substitution `s` to `t`.

You may want to use `s.key()` to retrieve the keys in a python dictionary `s` and use `s[t]` to get the value of a key `t` in `s`.

In [ ]:

```
def substitute_in_term (s : dict, t : Term) -> Term:
    (* YOUR CODE HERE *)

def substitute_in_clause (s : dict, c : Rule) -> Rule:
    (* YOUR CODE HERE *)
```

Assignment Project Exam Help

https://powcoder.com

Add WeChat powcoder

Updated April-29: Additional hint for the Python implementation of substitution:

Please implement the substitute_in_term and substitute_in_clause functions **both in a functional way**:

```
def substitute_in_term (self, s : dict, t : Term) -> Term:
    if isinstance(t, Function):
        new_terms = []
        for term in t.terms:      # Iterate  the function t's parameters
            new_terms.append ... # Add substituted term to new_terms

        return Function(t.relation, new_terms)
    elif ...
```

In this code, we always create a new instance and leave the original goal, fact, or rule instances unchanged. This is important because the interpreter uses the substitution functions to freshen a rule and unify the head of a rule or a fact with a goal. If substitution is applied imperatively in place in an instance like the code below:

```
def substitute_in_term (self, s : dict, t: Term) -> Term:
    if isinstance(t, Function):
        for i in range(len(t.terms)): # Iterate the function's parameters
            t.terms[i] = ...          # Update a function parameter in place in t
        return t
    elif ....
```

Assignment Project Exam Help

variables in the original goals, facts, or rules (e.g. t) are changed. This could unexpectedly alter the original program semantics! If we implement the substitution functions in a functional way so they always return a new clause or a new term (like the first piece of code above), the original program is preserved. In this case, the freshen function in Problem 4 always returns a new copy with no references pointing back to the original program. Renaming the rules would therefore leave the original program unchanged. In this way, the original program can be safely reused repeatedly in the while-loops of the pseudocode for Problem 4. If we don't do this and instead do something similar to the second piece of code above, we may have weird bugs in the code for Problem 4 because our rules could have accidentally become different than the original ones after a substitution.

https://powcoder.com
Add WeChat powcoder

Problem 3 (11 points)

Implementing the function:

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

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

OCaml Implementation

Because unification is driven by subsititution, you will need to use `module Substitution` defined in Problem 2. (See API docs for OCaml Map: https://caml.inria.fr/pub/docs/manual-ocaml/libref/Map.S.html (https://caml.inria.fr/pub/docs/manual-ocaml/libref/Map.S.html))

You may want to use the `substitute_in_term` function 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).

Assignment Project Exam Help

You may 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). `Substitution.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 `f t`.

https://powcoder.com

Add WeChat powcoder

You may want to use `List.fold_left2` to implement the `fold_left` function in the pseudocode (https://caml.inria.fr/pub/docs/manual-ocaml/libref/List.html (https://caml.inria.fr/pub/docs/manual-ocaml/libref/List.html)).

In [ ]:

```
exception Not_unifiable

let unify t1 t2 =
  (* YOUR CODE HERE *)
```

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") (v
ar "Y") ||
        unify (func "f" [var "X"]) (func "f" [var "Y"]) = Substitution.singleton (var "Y") (va
r "X"))
```

Assignment Project Exam Help
https://powcoder.com
Add WeChat powcoder

In [ ]:

```
(* A substitution that can unify function f(X,Y,a) 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}")
```

Python Implementation

The idea is similar to the above OCaml implementation. You must use a Python dictionary to represent a subsititution map.

```
def unify (t1: Term, t2: Term) -> dict:
```

The function should return a substitution map of type `dict`, which is a unifier of the given terms. You may find the pseudocode of `unify` in the lecture note `Control in Prolog` useful. The function should raise the exception `raise Not_unfifiable ()` if the given terms are not unifiable.

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

You may want to use `{}` for an empty subsitution map, `{v:t}` for creating a new subsitution map that contains a single substitution `[v/t]`, and `s[v]=t` 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 want to use a `for` loop for the θ `[X/Y]` operation in the pseudocode. This for-loop 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).

Assignment Project Exam Help

In [ ]:

```
def unify (t1: Term, t2: Term) -> dict:
    (* YOUR CODE HERE *)
```

https://powcoder.com

Problem 4 (11 points)

Add WeChat powcoder

The main task of problem 4 is to implement a nondeterministic Prolog interpreter. You may follow the **pseudocode** `Abstract interpreter` in the lecture note `Control in Prolog` to implement the interpreter.

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

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 i
n
  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 python implementation for `freshen` is given in `src/final.py`.

OCaml Implementation

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

where

- the first argument is a `program` which is a list of clauses (rules and facts).
- the second argument is a `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 the tests cases below as examples.

Assignment Project Exam Help

Please follow the **pseudocode**, which means that 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 **whose head can be unified with the chosen goal**. You may find the function `Random.int n` useful. This function returns a random integer in `[0, n)`.

https://powcoder.com

In [ ]:

Add WeChat powcoder

```
let nondet_query program goal =
  (* YOUR CODE HERE *)
```

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") (v
ar "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")])
```

Assignment Project Exam Help
https://powcoder.com
Add WeChat powcoder

(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)))] )
```

Assignment Project Exam Help
https://powcoder.com
Add WeChat powcoder

Python Implementation

```
def nondet_query (program : List[Rule], goal : List[Term]) -> List[Term]:
```

where

- the first argument is a `program` which is a list of `Rule`s.
- the second argument is a `goal` which is a list of `Term`s.

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

Please follow the **pseudocode**, which means that you need to have a nested loop 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 **whose head can be unified with the chosen goal**. You may find the function `random.randint(a, b)` useful. This function returns a random integer in `[a, b]`.

In [ ]:

```
def nondet_query (program : List[Rule], goal : List[Term]) -> List[Term]:
    (* YOUR CODE HERE *)
```

Assignment Project Exam Help

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

https://powcoder.com

Challenge Problem (5 points, 16 lines of code)

Add WeChat powcoder

Implementing a `deterministic` Prolog interpreter that supports 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.

OCaml Implementation

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

where

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

The function returns a list of term lists (`results`). Each of these results is an instance of the original `goal` and is a logical consequence of the `program`. If the given `goal` cannot be dervied as a logical consequence of the `program`, then the result is an empty list. See the tests cases below as examples.

For the goal order of the interpreter, always choose the left-most subgoal to solve. For the rule order, apply the rules in the order in which they appear in the program.

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

In [ ]:

```
let query program goal =
  (* YOUR CODE HERE *)
```

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")])
```

Assignment Project Exam Help

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

https://powcoder.com

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'
```

Add WeChat powcoder

Python Implementation

```
def det_query (program : List[Rule], goal : List[Term]) -> List[List[Term]]:
```

where

- the first argument is a `program` which is a list of `Rule` s.
- the second argument is a `goal` which is a list of `Term` s.

The function returns a list of term lists (`results`). Each of these results is an 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 the tests cases above as examples.

For the goal order of the interpreter, always choose the left-most subgoal to solve. For the rule order, apply the rules in the order in which they appear in the program.

Hint: Implement a depth-first search (DFS) procedure. Backtracking and choice points naturally fall out of the routine of DFS.

In [ ]:

```
def det_query (program : List[Rule], goal : List[Term]) -> List[List[Term]]:
    (* YOUR CODE HERE *)
```

Assignment Project Exam Help

https://powcoder.com

Add WeChat powcoder