Assignment 6
Readme

Vishal Bindal (2018CS50425)

Run instructions

```
make
./toy <path to file containing program (list of clauses)>
```

Example:

```
make
./toy data.pl
```

For cleaning generated files:

```
make clean
```

A prompt '?-' is printed to the console, and the user is expected to type a formula (query). The answer to the query is printed.
If more than 1 answers are possible, user input is awaited. The user can type (';' + ENTER) to print the next answer, or ('.'+ ENTER) to finish the query.

File structure

1. '**interpreter.ml**' contains all data type implementations, and all functions for the logic programming interpreter.
2. **'stage.ml'** reads from file and console, and calls functions from interpreter.ml
3. **'tokenize.mll**' and '**parser.mly**' are used for tokenising and parsing rules and facts from the file containing the program.
4. '**goallex.mll**' and '**goalparser.mly**' are used for tokenising and parsing a formula entered by the user on the console.
5. **'nextlex.mll**' and '**nextparser.mly**' are used for parsing the user input for printing the next solution (';') or terminating the query ('.')

Data types

type variable = string;;
type atom = string;;
type term = Var of variable | Const of atom | Node of (atom* (term list));;
type formula = atom * (term list);;
type rule = formula * (formula list);; (* head*body *)
type fact = formula;; (* head *)
type clause = Fact of fact | Rule of rule ;;
type program = clause list;;

type substitution = (term* term) list;;

Brief description of implementation

1. The program (list of clauses) is parsed from the file, and a variable of type program is created.
2. All variables in the program are assigned unique 'internal' names.
    a. For clause no. clid (>=0) and variable no. vid (>=0) (in the given clause), the variable name is assigned as '$clid$vid'.
    b. This is done to ensure that the scope of each variable is limited only to its particular rule/fact.
    c. The particular format containing '$' is chosen to distinguish internal variables from user entered variables and to ensure each variable in the program can be assigned a unique name.
    d. e.g. if 2 clauses are:
    ```
    mortal(X) :- man(X).
    great(X) :- father(X,Y). man(X).
    ```
    Then their representation in the program will be:
    ```
    [((mortal, [Var($0$0)]), [(man, [Var($0$0)])]) ,
    ((great, [Var($1$0)]), [(father, [Var($1$0), Var($1$1)]),
    (man, [Var($1$0)])])]
    ```
3. User enters a formula which is parsed. solve_goal is called which returns a tuple (found, ansls)
    a. found denotes whether a solution is found. If not then 'no\n\n' is printed.
    b. ansls denotes a list of substitutions (each substitution being an answer).
4. print_sub_list_onebyone is used to print this list one at a time.

Description of functions used for solving the query

1. **matchfact**
    a. (form:formula) -> (f:formula) -> (curfound:bool) -> (curansls:substitution list) -> bool * substitution list
    b. form = query formula
    f = formula present in program as Fact(f)
    curfound = bool indicating if solution has been found
    curansls = list of answers (substitutions)
    c. Returns: a tuple consisting of a boolean representing if a solution exists, and a list of valid substitutions.
    d. Find mgu of terms corresponding to form and f (Node(form) and Node(f)).

   e. If unifiable, return true (indicating solution found) and the unifier substitution appended to curansls
   f. else return curfound, curansls
2. **match_formula_ls**
   a. (fls:formula list) -> (lastsub:substitution) -> (pro:program) -> bool * substitution list
   b. fls = formula list, equal to or subset of body of a rule
      lastsub = substitution to be applied to all formulae in fls before unifying further pro = original program
   c. Returns: a tuple consisting of a boolean representing if a solution exists, and a list of valid substitutions.
   d. This function is called when the head of a rule matches with the query formula, passing the body of the rule as fls. lastsub then denotes the mgu of query formula and head of rule.
   e. If 'form' is head of fls, apply substitution lastsub to it and call function 'solve' (defined below). It will return a tuple (foundn, anslsn), where foundn is a bool indicating if solution(s) were found, and anslsn is a list of valid substitutions. If solution(s) exist, then it will return (true, ansls). Find composition of lastsub with each substitution in this list. Say this tuple is (curfound, curansls)
   f. If tail of fls is empty, then it means the whole formula list of the rule has been traversed, and curansls is the list of valid substitutions for the rule. Hence return curfound, curansls
   g. Else, if solution(s) has been found for 'form' but list is not empty, then call match_fls_multiple, which will solve for the remaining tail for each of the solutions found for 'form'.
   h. Else, if a solution could not be found for 'form', then there's no point traversing the tail, as for a rule to be matched, each and every formula in the body has to be matched. So return (false, [])
3. **match_fls_multiple**
   a. (cursubls:substitution list) -> (fls:formula list) -> (pro:program) -> (found:bool) -> (curansls:substitution list) -> bool * substitution list
   b. 'cursubls' = equal to or subset of list of valid substitutions which was found for the last formula (say 'form') in the rule body.
      'fls' = the tail of the rule body appearing after 'form'.
      'pro' = the original program.
      'found' = if a solution has been found for one of the valid substitutions present in cursubls when it was passed first, i.e. for one of the substitutions in (cursubls(0) - cursubls(cur)).
      'curansls' = list of valid substitutions found for (cursubls(0) - cursubls(cur))
   c. Returns: a tuple consisting of a boolean representing if a solution exists, and a list of valid substitutions.
   d. Brief description: Whenever a list of solutions is found for a formula in the body of a rule (in match_formula_ls), then match_fls_multiple is called to try to find

solutions for the tail fls, trying each of the substitutions in cursubls one-by-one, by calling match_formula_ls for each substitution.

   e. If cursubls is empty, then all valid substitutions for 'form' have been tried, so (found, curansls) is returned
   f. If ‘sub’ is the head of cursubls, then try to find a solution corresponding to sub by calling match_formula_ls. Recursively call match_fls_multiple for tail of cursubls, updating found and curansls.

4. **matchrule**
   a. (form:formula) -> (r:rule) -> (pro:program) -> (curfound:bool) -> (curansls:substitution list) -> bool * substitution list
   b. Returns: a tuple consisting of a boolean representing if a solution exists, and a list of valid substitutions.
   c. form = user entered query formula
      r = rule in program
      pro = original program
      curfound, curansls = tuple corresponding to current solution
   d. Try to unify terms corresponding to head of r and form. If unifiable, call match_formula_ls with the mgu and tail of r. If solution found, return (true, ls) where ls is concatenation of curansls and new list of valid substitutions.
   e. If not matched, return back (curfound, curansls)
5. **solve**
   a. (form:formula) -> (pro:program) -> (pro0:program) -> (found:bool) -> (ansls:substitution list) -> bool * substitution list
   b. form = user entered query formula
      pro = list of clauses yet to check
      pro0 = original program
      found = if solution has been found in (pro0 - pro)
      ansls = list of valid substitutions found in (pro0 - pro)
   c. Returns: a tuple consisting of a boolean representing if a solution exists, and a list of valid substitutions.
   d. If clause c is head of pro, call matchfact or matchrule accordingly, and call solve for tail of pro in a tail iterative way, updating found and ansls
   e. If pro is empty, then all clauses have been checked. Return (found, ansls)
6. **solve_goal**
   a. Call ‘solve’ with found=false, ansls=[ ], and pro=pro0
   b. Filter out the substitutions containing ‘internal’ variables, since they should not be printed in the answer.
7. **subst, compose, mgu**
   a. reused from previous assignment

Examples

Let program be:

```
man(socrates).
man(luke).
mortal(X) :- man(X).
father(anakin, luke).
great(X) :- man(Y), father(X,Y).
```

Consider the following queries:-

1. man(X).

a. solve calls matchfact for 1st clause, which unifies. found=true, ansls =
   `[(Var(X), Const(socrates)]`
b. solve calls matchfact for 2nd clause, which unifies. found=true, ansls =
   ```
   [ [(Var(X),Const(socrates))],
     [(Var(X), Const(luke))] ]
   ```
c. solve calls matchfact or matchrule for remaining clauses, but found, ansls remain unchanged
d. Final answer
   ```
   X = luke;
   X = socrates.
   ```

2. mortal(X).

a. solve calls matchfact for first 2 clauses, but found, ansls remain (false, [ ])
b. solve calls matchrule for 3rd clause. The head of the rule mortal($2$0) unifies with mortal(X). ($2$0 is internal representation, 2=clause id, 0=variable id)
c. match_formula_ls is called with fls=`[ (man, [Var($2$0)]) ]` and lastsub = `[ (Var(X), Var($2$0)) ]`
d. Head of fls is `(man, [Var($2$0)])`, so solve is called for this formula with lastsub applied (no substitution happened in this case). solve returns a result similar to query 1 with
   ```
   (true, [  [(Var($2$0),Const(socrates))],
          [(Var($2$0), Const(luke))] ] )
   ```
e. The substitution is composed with lastsub to yield
   ```
    [  [(Var($2$0),Const(socrates)), (Var(X),Const(socrates))
   ], [(Var($2$0), Const(luke)), (Var(X), Const(luke))] ] .
   ```
f. Since tail is empty, the above substitution is returned.

g. Finally, the tuples containing Var($2$0) are removed since they are internal representation, with final answer being
   `X = luke;`
   `X = socrates.`

`3. great(X).`

a. For first 4 clauses, (found, ansls) remains (false, [ ]).
b. matchrule is called for 5th clause, and the head unifies, so match_formula_ls is called with fls = `[ (man, [Var($4$1)]), (father, [Var($4$0), Var($4$1)]) ]`, and lastsub = `[ (Var(X), Var($4$0)) ]`
c. solve is called for head of fls, yielding
   ```
   (true, [  [(Var($4$1),Const(socrates))],
         [(Var($4$1), Const(luke))] ] )
   ```
   composed with lastsub to yield
   ```
   [  [(Var($4$1),Const(socrates)), (Var(X),Var($4$0))],
   [(Var($4$1), Const(luke)), (Var(X), Var($4$0))] ]
   ```
d. match_fls_multiple is called with cursubls = the list above, and fls = `[(father, [Var($4$0), Var($4$1)])]`
e. It calls match_formula_ls with sub = `[(Var($4$1),Const(socrates)), (Var(X),Var($4$0))]`, and fls = `[(father, [Var($4$0), Var($4$1)])]`. No solution is found so (false, [ ]) is returned.
f. It then calls match_formula_ls with sub = `[(Var($4$1), Const(luke)), (Var(X), Var($4$0))]`,and fls = `[(father, [Var($4$0), Var($4$1)])]`. It returns (true, `[(Var($4$1), Const(luke)), (Var(X), Const(anakin)), (Var($4$0), Const(anakin))]`
g. The tuples containing Var($4$0) and Var($4$1) are removed, with final answer being
   `X = anakin.`

Limitation

Since a solution consists of a tuple (found, ansls), cases with infinite answers cannot be handled in this implementation.