CS442
Assignment 4

University of Waterloo

Winter 2023

This assignment tests your understanding of the content of Module 6. You will implement a logic programming language in OCaml. This assignment has only one part.

In this and all assignments, any behavior which we do not explicitly define will not be tested, so you may define it however you wish, or allow your program to fail. Make sure though that it actually *is* undefined; ask on Piazza if you're unsure.

Submit all code via UWaterloo submit, e.g.:

```
submit cs442 a4 .
```

This assignment is due on Friday, March 24th, by 12PM *NOON, NOT MIDNIGHT*, Eastern time.

Conlog

Conlog is a reduced version of Prolog, with the disadvantages of both Prolog and Datalog, and the advantages of neither. In this assignment, you will implement a query engine for Conlog.

Conlog is a subset of Prolog. Like Datalog, there are no structures, no cut operation, and no meta-predicates. It also has no lists, only relations, atoms, and integers. It will be implemented with OCaml's `int` type, and so is not Turing-complete. With integers of unlimited range, it would probably be Turing-complete, but extremely awkward to use.

Conlog's search and unification algorithm follows Prolog's; that is, there is no query optimization, and certain queries which could terminate if rearranged should not. You should simply evaluate each predicate in order. Furthermore, our implementation will only give the *first* solution, and cannot resolve multiple substitutions for any query.

You are provided with two files, `conlog.ml` and `frontend.ml`. `conlog.ml` is an implementation of the parser and some utilities for Conlog; `frontend.ml` is the interactive frontend.

`conlog.ml` defines the following types:

```
type substitution = string * string
type relation = string * string list
type expr =
    | Binexp of expr * string * expr
    | Primary of string
type comparison = string * string * expr
type predicate =
    | Relation of relation
    | Comparison of comparison
type hornClause = relation * predicate list
type database = hornClause list
```

The primary element of parsed Conlog code is represented as a string, which may be in one of three forms: A variable name (starts with an upper-case letter), an atom (starts with a lower-case letter), or a number. Such elements will be called "primaries".

A database is a list of Horn clauses. A Horn clause is a pair, connecting a relation (the head) to a predicate list (the body). A predicate is either a relation or a comparison. A relation is a pair of a functor and a list of arguments. The functor and arguments are all strings. The functor must be an atom, and the arguments are primaries.

A comparison is a triple of two strings and an expression. The first string is the left-hand side of the comparison, which must be a variable, atom, or number. The second string is the comparison operator, which may be `">"`, `">="`, `"<"`, `"=<"`, `"is"`, or `"=\\="` (note that `\\` is necessary to escape strings in OCaml, but this is just written `=\=` in source code). The expression is the value to which the left-hand side is to be compared, which is built from the mathematical operators `+`, `-`, `*` and `/`. The parser accepts `=` as equivalent to `is` (which isn't technically correct Prolog), but stores the operator as `"is"`, so you don't need to worry about this case. Similarly, it accepts `//`, Prolog's integer division operator, as equivalent to `/`, since we'll only be implementing integers. You don't need to worry about expressions at all, as a function to solve them, `solve`, is provided.

A substitution is a pair of strings. The pair `(X,Y)` indicates the substitution `[Y/X]`, i.e., substituting the variable `X` for `Y`. Because of how Conlog works, `Y` can only be a primary, so only needs to be a string. You will not be asked to implement substitution (I'm sure you had enough of that in Assignments 2 and 3), but will be asked to generate correct substitution lists.

`conlog.ml` defines the following functions (and many more):

```
(* Functions to distinguish primary elements of Prolog *)
isVarName : string -> bool
isAtom : string -> bool
isNum : string -> bool

(* Since primaries can be handled differently in some cases, check if this expression is a primary *)
isPrimary : expr -> bool

(* Tokenize a program *)
tokenize : char list -> string list

(* Parse a predicate list for a query *)
parsePredicateListStr : string -> predicate list option

(* Parse a whole database from a string *)
parseDatabaseStr : string -> database option

(* Find all the free variables in this Horn clause *)
freeVars : hornClause -> string list

(* Perform a substitution over a single name *)
subName : string -> substitution list -> string

(* Perform a list of substitutions over a list of arguments *)
subArgs : string list -> substitution list -> string list

(* Perform a list of substitutions over a list of predicates *)
subPredicateList : predicate list -> substitution list -> predicate list

(* "Freshen" names in this Horn clause, to make sure they can't conflict *)
freshenHornClause : hornClause -> hornClause

(* Solve a mathematical expression *)
solve : expr -> int option

(* Print this database *)
printDatabase : database -> ()
```

To use any of these functions from another file, you must refer to them with the module name `Conlog`, such as `Conlog.solve`. If you prefer, you may instead `open Conlog`, and then use them un-prefixed.

You can use `frontend.ml` to test your code; it provides an interactive interface as described in Module 6. However, your code may not use the functions in `frontend.ml`.

The usual entry points are `parseDatabaseStr` and `parsePredicateListStr`, which will parse a database and a predicate list, respectively. The frontend uses them to parse the input database and the query, respectively. `freeVars`, the `sub*` functions, and `freshenHornClause` functions help with the mechanical parts of answering queries; you will probably want to use them to implement this assignment, and may also want to read them to get accustomed to the structure of a parsed Conlog program. The `solve` function is used in handling comparisons, and is essentially just a calculator. `printDatabase` and other `print*` functions are used to print, for debugging purposes.

Additionally, you are provided with `exa.pl`, an example Prolog database compatible with Conlog. GNU Prolog is available on the Student Linux systems, and you can test `exa.pl` with it like so:

```
$ gplc exa.pl
... [ignore the warnings] ...
$ ./exa
GNU Prolog 1.4.5 (64 bits)
By Daniel Diaz
Copyright (C) 1999-2016 Daniel Diaz
| ?-
```

1 Query solver

Create a file, `a4.ml`, which implements at least the following function:

```
query : database -> predicate list -> substitution list option
```

`query` is given a database and predicate list, and returns `Some` substitution list that satisfies the query, or if it cannot be satisfied, `None`. To do this, it should follow the algorithm in Module 6. A more precise version of the query algorithm is given here.

The algorithm to satisfy a query (predicate list), satisfy(Q), is as follows:

- If Q is empty, the query is satisfied, and the result is the empty substitution (`[]`). Otherwise, let $P :: Q' = Q$.
- If P is a comparison L op R,
  - If the operator is `"is"`,
    - Let $S = \begin{cases} U(L, R) & \text{if } R \text{ is a } \texttt{Primary} \\ \textbf{error} & \text{if } R \text{ is not a } \texttt{Primary} \text{ and doesn't solve} \\ U(L, \text{solve}(R)) & \text{otherwise} \end{cases}$
    - If S was **error**, return `None`, otherwise
    - return S satisfy(Q' S) (i.e., satisfy the remaining query as substituted by S, and concatenate its result to S)
  - otherwise,
    - If L is not a number or R doesn't solve, return `None`, otherwise
    - if L op solve(R) is false, return `None`, otherwise
    - return satisfy(Q').
- otherwise, it is a relation. For each Horn clause H :- B in the database,
  - Let H' :- B' be the "freshened" Horn clause, that is, H :- B with all free variables given new, unique names. See `freshenHornClause`.
  - Let $S = U(P, H')$. If P and H' do not unify, abandon this Horn clause and try the next one.
  - Let $Q'' = (B'\ Q')S$. That is, concatenate B' with Q', then apply the substitution S.
  - Let $S' =$ satisfy(Q''). If the Q'' cannot be satisfied, abandon this Horn clause and substitution and try the next one.
  - Return $S\ S'$, i.e., the concatenation of the substitutions from this unification to the result of the rest.
- If no Horn clause works, the query is not satisfied.

Be very careful about the order of substitutions. It is natural to *prepend* substitutions to a substitution list because of how OCaml lists work, but the substitutions will be in the wrong order if they're prepended in this way. The substitutions must be in the correct order to find the resolution of a variable, because it will usually take several steps; for instance, if the query includes the variable X and the result is $X = 3$, the substitutions might include `[...; ("X", "Y_42"); ...; ("Y_42", "3"); ...]`, which will not behave correctly if applied to X in reverse. You may want to look up the `List.rev` or `List.append` functions, with reverse lists and concatenate two lists, respectively. Technically, `List.append` is a rather inefficient way to solve this problem because of how OCaml lists are structured, but this inefficiency is irrelevant for our purposes.

U is as defined in Module 6. If $U(a, b)$ does not match any case, you must commute (try $U(b, a)$) to find a match. It is important to try the direct match first, and the commuted match second, or the result will be incorrect.

You should perform no query optimization. Queries which expand infinitely by the above definition should not halt (or only halt due to running out of memory). We will test both halting and non-halting queries.

You can compile your `a4.ml` with our frontend with the following command:

```
$ ocamlfind ocamlc -package core -linkpkg -thread conlog.ml a4.ml frontend.ml -o frontend
```

Of course, you are also free to write your own testing code and compile against that; you should only submit `a4.ml`.

Suggestions

It would be difficult to implement this iteratively, and difficult to implement this correctly using mutation. The search algorithm requires backtracking—if a query cannot be satisfied, back up and try unifying this predicate with the next Horn clause and trying again—which is difficult to implement imperatively. `query` is a naturally recursive algorithm, and this backtracking happens naturally if you implement it recursively. This is most easily implemented from the bottom up: first implement unification and make sure sensible substitutions are generated, then work on satisfying a single predicate, then build the full algorithm.

The example database, `exa.pl`, covers all interesting kinds of recursion. You may want to write smaller test cases and test cases with predicates with more arguments.

Print debugging is extremely helpful for implementing the search and unification algorithms, and `print*` functions in `conlog.ml` can print all of the data types you will be working with.

Demonstration

The file `exa.pl` demonstrates the breadth of behaviors your Conlog implementation should support. `frontend.ml` is a standard Prolog-style frontend, except that it does not support getting multiple solutions to a query (since Conlog only ever returns one solution), and supports a "debug" mode, which shows the full substitution returned for every query. This demonstration uses `frontend.ml` and `exa.pl`, and is run like so: `./frontend exa.pl`.

In this demonstration, when debug mode is on, the resulting substitution will have variable names generated by `freshenHornClause`. It is not necessary that your intermediate names are all the same, or even that they're in precisely the same order, only that the final resolution of each variable is correct.

```
?- fac(A, B).
A = 1
B = 1

?- fac(1, X).
X = 1

?- fac(10, X).
X = 3628800

?- fac(Y, 1).
Y = 1
```

```
?- fac(X, 2).
No

?- fib(2, X).
X = 1

?- .dbg
Debug mode on

?- fib(2, X).
Free variables: X

N_9 -> 2
X -> M_8
Na_7 -> 1
Nb_6 -> 0
Ma_5 -> 1
Mb_4 -> 0
M_8 -> 1

X = 1

?- .dbg
Debug mode off

?- fib(15, X). % This query is very slow!
X = 610

?- fib2(15, X). % This query is much faster.
X = 610

?- fib2(90, X). % Maximum valid query on a 64-bit machine
X = 2880067194370816120

?- isprime(7).
Yes

?- isprime(8).
No

?- iscomposite(7).
No

?- iscomposite(8).
Yes

?- prime(1, P).
P = 2

?- .dbg
Debug mode on

?- prime(2, P).
Free variables: P

X_39 -> 2
P -> P_38
Xa_37 -> 1
Pa_36 -> 2
Pb_35 -> 3
X_262 -> 3
P_38 -> 3
X_283 -> 3
Y_282 -> 1
X_306 -> 3

P = 3

?- .dbg
Debug mode off
```

```
?- prime(5, P).
P = 11

?- prime(N, 11).
No

?- cogito(foo).
No

?- cogito(descartes).
Yes

?- sum(X).
X = descartes

?- p2(Q).
Q = c

?- sumto(100, N).
N = 5050

?- sumto(N, 5050).
No

?- goodPet(porcupine).
Yes

?- goodPet(descartes).
No

?- supervisorbyname(villefort, X).
X = dantes

?- supervisorbyname(dantes, X).
No

?- supervisorbyname(B, villefort).
B = napoleon

?- supervisorbyname(B, dantes).
B = villefort
```

Rights