Prolog MP

CS 440

Revision 1.0

Assigned April 21, 2014
Due May 5, 2014

1 Objectives and Background

The objective for this MP is to build an interpreter for a small Prolog-like language.

You will be given a datatype and a parser, along with these instructions. When you are done, you will have a simple Prolog interpreter!

2 Getting Started

Update your repository with `git pull` to find a directory called `prolog-mp`. There you will find a file called `prolog-mp.hs`.

It comes with a parser and a REPL, but no computation. Here is what happens if you run it:

```
*Main> :load prolog-mp.hs
[1 of 1] Compiling Main
Ok, modules loaded: Main.
*Main> repl (H.empty) 0
PL> human(socrates).
Noted.
PL> mortal(X) :- human(X).
Noted.
PL> listing.
fromList [("human",[human(socrates)]),("implies",[implies(mortal(X),human(X))])]
PL> ? mortal(X).
You entered a query.
PL> bye.
Bye
*Main>
```

The `PL>` string is the Prolog prompt.

3 Problems

Before you get started, please read the source code and try to understand how this code works.

3.1 The Unification Engine

The first thing you will need to add is the unification engine. It will work similarly to the one you did in class, but will not need as much functionality. The main function is called `unify`, and will unify lists of type `Pattern` together.

```
unify :: [Pattern] -> [Pattern] -> Bindings -> Bindings
```

If `bindings` is `Fail`, then `unify` should return `Fail` right away. If the patterns are both empty, it should return the bindings. Here are some sample runs.

```
*Main> unify [Var "X",Var "X"] [Obj "a",Obj "a"] NoBindings
Bindings fromList [("X",a)]
*Main> unify [Var "X", Obj "a", Var "Y"] [Obj "a", Var "X", Obj "b"] NoBindings
Bindings fromList [("X",a),("Y",b)]
*Main> unify [Var "X", Obj "a", Var "Y"] [Obj "a", Var "X", Obj "b"] Fail
Fail
*Main> unify [Var "X", Funct "f" [Var "Y"]] [Obj "z", Var "A"] NoBindings
Bindings fromList [("X",z),("A",f(Y))]
*Main> unify [Var "X", Obj "a", Var "Y"] [Obj "b", Var "X", Obj "b"] NoBindings
Fail
```

3.2 The Goal Solver

The goal solver consists of two functions, `prove` and `prove-all`.

Function `prove` will take three arguments, a `Funct` pattern, a database, and the set of current bindings. It will look up the functor name in the database to get back a list of clauses. The first element of each clause is called the head. If the goal unifies with the head, we take the resulting bindings and call `prove-all` on the tail. We do this for each of the clauses in the database, and return a list of all the bindings that worked.

The next function `prove-all` is very like a folded version of `prove`. It is meant to be called on the tail of a clause we have already unified, or on the user input if the user inputs a list of things to check.

The source code (in Lisp) is in the book Principles of Artificial Intelligence Programming, if you would like a reference copy. Haskell is not Lisp, but it should make this go a lot more smoothly.

3.3.2.1 Example

Let's suppose we have the standard ``hello world'' of Prolog in our database.

```
PL> human(socrates).
Noted.
PL> human(jane).
Noted.
PL> mortal(X) :- human(X).
Noted.
PL ? mortal(Who)?
```

The database will be a hashmap with two entries in it. The keys will be the strings ``human'' and ``mortal''. If you call `listing`, you get this:

```
fromList [("mortal",[[mortal(X),human(X)]]),("human",[[human(jane)],[human(socrates)]])]
```

When we run our query, `prove` will first look up `mortal` in the database. It will try to unify `mortal(X)` with `mortal(Who)`, which succeeds. It will then call `prove-all` on the rest of the clauses, i.e. `human(X)`. Then `prove-all` will call `prove` with `human(X)`, and the lookup will return `human(jane)` and `human(socrates)`. Both of these unify, so `prove` will then return a list of bindings, one with `X` being `jane`, and one with `X` being `socrates`.

4 Where to go from here

You are done with the MP. But there are many things you could still do to improve the language. Here's some ideas if you want to continue playing, and certainly you can add your own.

- We talked in class about the possiblity of variable capture. See if you can eliminate that.
- Add cut. You will need to use continuations to do this, most likely.
- Add numbers and aritmetic.
- Add lists.
- Update the parser to handle both and (`,`) and or (`;`).