Project #5 – Writing a Basic Prolog Interpreter
CS 152 Section 6 – Fall 2021
Michael McThrow
San José State University

Introduction

Your task for Project #5 is to write a basic Prolog interpreter that reads in a Prolog source code file that contains facts and accepts queries from the command line.

For example, suppose you have a source code file named pokemon.prolog that contains the following facts:

```
evolution(bulbasaur,ivysaur).
evolution(ivysaur,venusaur).
evolution(charmander,charmeleon).
evolution(charmeleon,charizard).
evolution(squirtle,wartortle).
evolution(wartortle,blastoise).

grass(bulbasaur).
grass(ivysaur).
grass(venusaur).

poison(bulbasaur).
poison(ivysaur).
poison(venusaur).

fire(charmander).
fire(charmeleon).
fire(charizard).

flying(charizard).

water(squirtle).
water(wartortle).
water(blastoise).
```

Your goal is to implement a program that accepts queries with the following prompt:

```
prolog> evolution(bulbasaur,ivysaur)?
true.
prolog> evolution(bulbasaur,X)?
X = ivysaur.
```

```
prolog>
```

If your search tree results in multiple answers, then please print each answer on its own line. For example:

```
prolog> grass(X)?
X = bulbasaur.
X = ivysaur.
X = venusaur.
```

When running your program, please provide your program as a command line argument. For example, if you implement this in Java, you would run your program as follows:

```
$ java PrologInterpreter pokemon.prolog
prolog>
```

Once you run your interpreter with the program loaded, output the string `"prolog> "` (note the extra space after the >. This is where the user would type a Prolog query. This will continue until the user exits the program; on Unix systems this would happen when the user presses Ctrl+D, which sends the program the EOF (end-of-file) symbol.

You need to implement support for the following features in addition to whatever is implied in the above examples:

- Rules (including the :- syntax and conjunctive queries on the right-hand side).
- Conjunctive queries (e.g., `fire(X),flying(X)?` should result in `X = charizard`.)
- Unification
- Lists (e.g., `member(b,[a,b,c])` should result in true, and `flatten([[1],2,[3,[4]]],X)` should result in `X=[1,2,3,4]`).

Please use the resolution and unification algorithms described in *The Art of Prolog* and in my lectures.

In addition, please implement the following type predicates:

- `atom/1`: Resolves to true if the argument is an atom.
- `var/1`: Resolves to true if the argument is a variable.
- `compound/1`: Resolves to true if the argument is a compound.
- `is_list/1`: Resolves to true if the argument is a list. A list is defined as a chain of pairs, similar to how Lisp lists are chains of cons cells. As in Lisp, keep in mind that not all pairs are lists, the second pair element must be a list, and the final pair in the chain must be the empty list.

Do not worry about adding support for numbers, cuts, negation, and other advanced features. The only features that we will need to concern ourselves with are the features from the first six chapters of *The Art of Prolog* plus the type predicates listed above. In general, you should be able to run your Program #4 solutions in this interpreter (except for sort) as long as those solutions did not take advantage of features that appeared after Chapter 6.

Rules

- **Please stick to the project specification regarding input formats.** I reserve the right to deduct points for implementations that deviate from the specification.
- **You may choose to work with up to two partners enrolled in this class for this assignment.** This is entirely optional; you may choose to work alone or as a pair instead of as a trio. Only one partner is responsible for turning in the assignment, but all partners will earn the same grade. **Place all partners' names on your submission files.**
- You may choose Java, C, C++, Python 3, R5RS Scheme, or Racket as your implementation languages. If you choose to use Scheme or Racket, make sure your code runs in DrRacket using `#lang r5rs` or `#lang racket`, respectively. If you choose to use another programming language, please ask me before you begin your implementations. However, any request to use a logic programming language to implement this assignment will be denied.
- If you choose to use Scheme or Racket, there are no restrictions on the use of mutation, side-effects, and macros. Please write your code as you feel fit.
- Please refrain from using third-party libraries in your code, with the exception of a parsing library such as ANTLR for the purpose of parsing Prolog expressions. A third-party library is a library that is not part of the standard of the language you choose. If you are unsure whether a library is a third-party library or not, ask me. You could choose to implement your own parser, but since I did not cover parsing techniques in this course, I don't recommend this.
- If you choose to use C or C++, keep in mind that your programs will be run in a Linux environment when I grade them. Please code in such a way where your code can work on either the GCC or Clang compilers.
- **Your code must compile or be interpreted without any syntactical errors in order for it to receive credit.**
- No matter which implementation language you choose, your code *must* be runnable from the command line, and your code must support the prompts *exactly* as described above from standard input. This is to facilitate the use of automated testing tools for grading, which makes grading easier and quicker.
- Your submission will be in a *.zip file that contains your source code, a README file describing which parsing library you used (if any), instructions for how to compile your code (if you used Java), a Makefile if you're using C or C++, and how to execute your code on the command line.

Grading Rubric

- I will be running your Prolog implementation against a battery of tests. If all of these tests pass perfectly, you receive 100% before any applicable bonuses and penalties. If any of the tests fail, you will receive deductions based on the severity of the error: minor mistakes will get minor deductions, while major mistakes or omissions will receive larger deductions.
- Grade Breakdown:
  - Basic Functionality: 85% (everything except support for pairs and lists)
    - Simple lookups that don't involve variables will earn 40%.

        - The remaining 45% will involve the successful implementation of unification and backtracking.
    - Pairs and Lists: 15%
    - To maximize your chances of earning a high grade, I recommend working on simple lookups first, then implementing unification and backtracking in order to successfully complete all basic functionality. Then once this is working completely, I recommend adding support for pairs and lists.
- Recall that your code must compile or run without any syntactical issues. If I am unable to compile your code, your assignment gets a grade of zero.

Grammar

Note that this grammar is in Extended Backus-Naur Form (EBNF):

```
/*
 * EBNF Grammar for Dialect of Prolog Used in Program #5
 *
 * Michael McThrow
 * CS 152 Section 6 -- Programming Paradigms
 * San José State University
 * Fall 2021
 */

/* Atoms are identifiers that start with a lowercase letter and can contain
 * a combination of lowercase letters, numerals, and underscores.  We are
 * also going to make zero an atom in order to accommodate natural numbers
 * per Program #4.
 */
<atom>   ::= [a-z]([a-z]|[0-9]|'_')*
           | '0'

/* Variables are identifiers that start with an uppercase letter and can
 * contain a combination of letters, numerals, and underscores.
 */
<var>    ::= [A-Z]([a-z]|[A-Z]|[0-9]|'_')*

/* A term is either an atom or a variable */
<term>   ::= <atom>
           | <var>

/* An element is either a term, a list, or a compound */
<element>  ::= <term>
             | <list>
             | <compound>

/* The <elements> rule consists of one or more elements delimited by a comma */
<elements> ::= <element>
             | <element> ',' <elements>

/* A list could have the following forms:
 * 1.  The empty list []
 * 2.  A list of elements (cound contain lists)
 * 3.  A pair, where the first and rest of the pair are delimited by the
 *     pipe (|) symbol (e.g. [X|Xs]), and where the first and rest are
```

```
 *     elements (for example, [a,b,c|[d,e,f]] is valid both grammatically
 *     and semantically.  To make grammar construction simpler, the example
 *     [a,b,c|d,e,f] is grammatically valid but semantically incorrect since
 *     the latter part of the pair must either be a list or a variable that
 *     unifies with a list.
 */
<list>   ::= '[]'
           | '[' <elements> ']'
           | '[' <elements> '|' <elements> ']'

/* A compound starts with either an atom or a period, is followed by the open
 * parenthesis, then is followed by one or more elements delimited by commas,
 * and is followed by the closed parenthesis.
 *
 * e.g., evolution(bulbasaur,ivysaur)
 * e.g., natural_number(s(s(s(0))))
 *
 * Note that the special compound .(X,Y) (the dot functor) represents the pair
 * [X|Y]; please refer to page 56 in The Art of Prolog.  This is very important
 * knowledge for getting resolution and unification working correctly on lists,
 * since lists are represented internally as pairs, and pairs are represented
 * internally using the dot functor.
 */
<compound> ::= <atom> '(' <elements> ')'
             | ".(" <elements> ')'

/* A conjunction is one or more compounds delimited by commas.
<conjunction> ::= <compound>
                | <compound> ',' <conjunction>

/* A rule consists of either a compound or a compound with a conjunction,
 * joined with the :- (implies that) operator.
<rule>   ::= <compound>
           | <compound> ":-" <conjunction>

/* A program is a sequence of rules that end with a period.  Ideally the
 * sequence should be delimited by new lines; if you are using ANTLR, then
 * you should take advantage of the WS rule.
 */
<program> ::= (<rule>'.')+

/* A query is a conjunction that ends with a question mark. */
<query>  ::= <conjunction>'?'
```