CSCI 4430 Programming Languages

Homework 2: Prolog

Posted: Friday, September 18.

Due: Friday, October 2 @ 1:59pm

Submission Instructions

This is an INDIVIDUAL assignment. Discussion with classmates, instructors and TAs is allowed, however, the actual work should be your own. Course staff runs plagiarism detectors regularly and will penalize for excessive similarities between submissions as outlined in the course syllabus.

Submit in Submitty. Your Prolog files must be named `foxes_and_hens.pl`, `parser.pl` and `diabolic.pl` respectively. Run `zip -r hw3.zip foxes.pl parser.pl diabolic.pl` to create the zip file, then submit `hw2.zip` for autograding.

Problems

Problem 1 (20pts): Foxes and Hens

You will write a simple program in Prolog to solve the following puzzle:

In a children's story, three foxes and three hens are traveling together. They come to a river they must cross. There is a boat, but it will hold no more than two of them at a time, so at least six trips in each direction will be required (one animal has to row the boat back each time). The problem is that the hens cannot trust the foxes; if in the process of crossing the river we ever end up with more foxes than hens on either shore, the foxes are likely to eat the hens. We need a series of moves (trips across the river in the boat) that will get the entire party across safely.

Your main predicate should be a goal **`solve(P)`**. It should instantiate **`P`** to be a list of configurations, each of which indicates the location of the animals and the boat. If you request additional solutions (by typing a semicolon) the interpreter should give you additional lists, as many as there are unique solutions. Represent a configuration as follows: give the number of foxes, hens, and boats on the near (left) shore: [LF, LH, LB], where 0 <= LF <= 3, 0 <= LH <= 3, and 0 <= LB <= 1.

You can think of the state space of this problem as a graph in which nodes are configurations and edges represent feasible boat trips. It is a large graph, however, and you do not want (nor in fact are you permitted) to represent it explicitly in your Prolog database. From any given state you will need to figure out the set of possible neighbors to explore. You will also want to keep track of where you have been, so you do not get stuck in a cycle.

You may find it helpful to use the not **`(\+)`** operator. You may also want to use the cut **`(!)`** to avoid unnecessary backtracking. You are not permitted to use **`assert`**, **`retract`**, or other database-modifying predicates. (You must compute your solutions. You are not allowed to figure them out by hand and simply write a program that prints them.)

Note on grading: This porblem is worth 20 points. 15 points will be autograded. However, we will override the autograder if you have violated the restrictions specified above. 5 points will be awarded for code quality --- you are expected to write declarative code, which tends to be neat and compact; rambling solutions as well as ones that use imperative “constructs” (e.g., if-then-else, loops) will be penalized.

Problem 2 (30pts): Table-driven Predictive Parser

You will write a generic table-driven predictive parser in Prolog. The parser will be “instantiated” with the LL(1) grammar for arithmetic expressions with operators **-** (minus) and ***** (times). Given an expression as input, the parser will parse the expression, producing as a result the sequence of prductions applied. In addition, the parser will “interpret” the expression during the predictive parse computing the expression value.

The LL(1) grammar for arithmetic expressions with operators **-** and ***** is as follows. The grammar is very similar to the one we used in class.

1. *srart* → *expr*
2. *expr* → *term term_tail*
3. *term_tail* → **-** *term term_tail*
4. *term_tail* → ε
5. *term* → **num** *factor_tail*
6. *factor_tail* → ***** **num** *factor_tail*
7. *factor_tail* → ε

First, you should build the LL(1) parsing table for the grammar. (You do not need to code it in Prolog; build the parsing table manually and encode it in terms of Prolog predicates **predict** as we explain shortly.) Given the LL(1) parsing table, the parser behaves in the standard way. At each step, it checks the top of the stack. If the top of the stack is a nonterminal *A*, the parser looks up the next input symbol *I* then consults the table to "predict" what production to apply. If the table predicts pruduction number **N**, which is, say *A* → *β*, the parser updates the stack accordingly. Otherwise, i.e., if the top of the stack is a terminal, and this terminal matches lookahead symbol *I*, the parser pops the terminal off the stack, and so on.

Next, you will augment the parser with the evaluation of arithmetic expressions. Each successful parse should print the value of the arithmetic expression along with the production sequence.

Encode the grammar and parsing table as explained below. Starter code, which makes use of the encoding, is given here (it's a .txt file).

Predicates **term(...)**, **non(...)** and **prod(...)** denote terminal, nonterminal, and production, respectively. Terminals and nonterminals have the form **term(name,V)** and **non(name,V)** respectively, where **name** is a constant that denotes the symbol, and **V** stores some "attribute" computed and associated to the parse tree node corresponding to the symbol. For example, terminal symbol **-** is represented as **term(minus,V)** and nonterminal *expr* is represented as **non(e,V)**. Productions have the form **prod(N,[H|R])** where **N** is the integer number of the production, and **[H|R]** is the list of production symbols (**H** is the left-hand-side of the production, and **R** is the sequence of symbols on the right-hand-side of the production). For example, production 1. *expr* → *term term_tail* is encoded as **prod(1,[non(e,Ve),non(t,Vt),non(tt,Vtt)])**.

The parsing table encoding uses predicate **predict(...)**. For example, **predict(non(e,Ve),term(num,Vnum),1)** stands for "on nonterminal **expr** and terminal **num**, predict production 1. *expr -> term term_tail*".

To evaluate the expression, use argument **V** in predicates **term(_,V)** and **non(_,V)**. You have to encode the computation generically, this time using predicate **attribute(...)**. There is predicate **attribute(N,[H|R])** for each production and rules for **attribute(...)** may have subgoals that perform computation.

Your parser should not be hard-coded against the productions of the given grammar. That is, one should be able to change the grammar to another LL(1) grammar (**prod(...)** **predict(...)** and **attribute(...)**) and still reuse the code for the parser. As long as productions are encoded as **(index,[H|T])** tuple, the code for the parser should work.

Note on grading: This porblem is worth 30 points. 25 points will be autograded: the sequence is worth 15 points and the value is worth 10 points. 5 points will be awarded for code quality.

EXTRA CREDIT: Problem 3 (7pts): Magic Squares

A so-called "diabolic magic square" is an **NxN** matrix whose cells contain the numbers **1...N^2**, arranged in such a way that the cells in every row, column, and diagonal sum to *(n^3 + n)/2*. Diagonals are broadly defined: they include not only the major (corner-to-corner) diagonals, but the "broken" diagonals as well. If you imagine tiling a wall with a large number of identical copies of a diabolic magic square, every set of four cells in a line, horizontally, vertically, or diagonally, will add up to the same number.

Here is a 4x4 diabolic magic square:

```
1    8    10    15

14   11   5     4

7    2    16    9

12   13   3     6
```

It turns out that there are exactly 384 distinct 4x4 diabolic magic squares. Given any one of them, the others can be created by applying combinations of five simple transformations: reflection, rotation about the center point, rotation of columns (e.g. remove the right-most column and tack it on the left), rotation of rows (remove the top row and tack it on the bottom), and one more complex convolution best described with a picture:

```
A    B    C    D                    A    D    H    E
```

```
E   F   G   H       ==>       B   C   G   F

I   J   K   L                 N   O   K   J

M   N   O   P                 M   P   L   I
```

As shown by Rosser and Walker (Bulletin of the American Mathematical Society, June 1938), these five transformations form a group (a closed mathematical system) isomorphic to the self-mappings of the three-dimensional hypercube. Further information can be found in Chapter 12 (pages 130-140) of Martin Gardner's The Second Scientific American Book of Mathematical Puzzles and Diversions, a Fireside Book published by Simon and Schuster, New York, 1961.

Your program should provide a **`diabolic`** predicate that can be driven both forward and backward. If you type **`diabolic([1, 8, 10, 15, 14, 11, 5, 4, 7, 2, 16, 9, 12, 13, 3, 6])`** the interpreter should print **`true`**. If you type **`diabolic(L)`** it should print a diabolic magic square, and should print additional squares in response to semicolons.

Finally, your program must provide a zero-argument predicate **`write_all`** that computes and lists all 384 4x4 diabolic magic squares, with no repetitions into a file **`solution_diabolic_test3.txt`**. Note that it is not acceptable, for the purposes of this assignment, to pre-compute the squares and store them in the Prolog database. You may want to use built-in predicates **`write`** and **`nl`**.

You must print the magic squares (in any order) starting with **`L =`** :
`L = [1, 8, 10, 15, 14, 11, 5, 4, 7, 2, 16, 9, 12, 13, 3, 6]`
`L = [...]`
etc.

Note on grading: This porblem is worth 7 points. All points are autograded.

Errata

None yet. Check Announcements in Submitty regularly.