1 Lab 4 Prolog

Date: Mar 25, 2021

This document first describes the aims of this lab followed by exercises which need to be performed.

You should perform all exercises using the `swipl` dialect of Prolog which is installed on remote.cs.

1.1 Aims

The aim of this lab is to introduce you to Prolog. After completing this lab, you should have some familiarity with the following topics:

- Using Prolog's bidirectional pattern matching to access and construct data structures.
- Writing simple Prolog programs which do arithmetic.
- Writing simple list processing functions in Prolog.

1.2 Exercises

1.2.1 Starting up

Follow the *provided directions* for starting up this lab in a new git `lab4` branch and a new `submit/lab4` directory.

For this lab, all code should be written in a single file `lab4-sol.pro` residing in your `submit/lab4` directory. Submit that file along with a log of your terminal interaction.

Note that you can repeatedly load that file into your Prolog interpreter using:

```
?- ['lab4-sol.pro'].
true.
```

1.2.2 Exercise 1: Prolog Pattern Matching

Recall from class that Prolog terms are either atomic (symbols or numbers), structures which are applications of constructor functions, or variables. Syntactically, numbers have the usual syntax while Prolog symbols are identifiers starting with lower-case letters or quoted within single-quotes or a sequence of certain non-word characters. Prolog variables are identifiers starting with upper-case characters or `_`.

Prolog does (almost) all computation using a general form of pattern matching called **unification**. The matching works by finding the most general substitution for Prolog variables which makes two terms identical.

Try the following attempts at pattern matching in the swipl interpreter and observe the results:

```
f(X, a) = f(a, Y).
f(X, a) = f(a, X).
f(X, a) = f(b, X).
f(X, Y) = f(a, a), g(X, Y) = g(a, b).
```

The first two matches will succeed, but the third one will fail because there is no consistent substitution for `X` which makes both terms identical.

The fourth example will also fail. The first match succeeds with `X = a` and `Y = a`, but the subsequent match for the `g/2` terms will attempt to set `Y` to `b` which fails because it already has the value `a`.

Now type matches into the Prolog interpreter to extract the first `z` from the following terms into a Prolog variable `X`: (note that you can use the special anonymous `_` variable to match terms you are not interested in):

1. `f(1, 2, z).`
2. `head(a, tail(z, B), Y).`
3. `cons(a, cons(b, cons(c, z))).`

1.2.3 Exercise 2: Matching Lists

Recall that a Prolog term is a **proper list** if:

- It is the empty list represented using the atom `[]`.
- It is a pair `[X|Y]` such that `Y` is a proper list. (Note that `[X|Y]` is syntactic sugar for `'[|]'(X, Y)`).

Which of the following Prolog terms are proper lists?

```
[a, b].
[a|b].
[a|[b]].
[a|[b|c]].
[A|[B|C]].
```

[You can test your predictions by feeding them into the builtin procedure `length`¬ `/2` which matches its second argument with the length of the proper list given by its first argument, signalling an error if there is no substitution which makes the first argument into a proper list].

Write match expressions to extract the following (using Scheme terms):

1. The `cddr` of the list `[1, 2, 3, 4]`.
2. The `caddr` of the list `[1, 2, 3, 4]`.
3. The `cdar` of the list `[[1, 2], 3, 4]`.

Use racket to ensure that you have the right interpretation of the Scheme terms.

1.2.4 Arithmetic in Prolog

The built-in Prolog operator `is/2` provides access to hardware arithmetic. Specifically, `is/2` will succeed if the result of **evaluating** its RHS can be matched with its LHS:

Type the following into your Prolog interpreter and observe the results:

```
N = 1 + 2.
N is 1 + 2.
N is 1 + 2*3.
5 is 7 mod 2.
X is 7 mod 2.
X is -7 mod 3.
N is sqrt(4).
N = pi.
N is pi.
```

1. Write a Prolog procedure `quadratic_roots(A, B, C, Z)` which matches `Z` with a root of the equation $Ax^2 + Bx + C = 0$. **Hint**: use different rules for each root.

   ```
   ?- quadratic_roots(2, 5, 2, Z).
   Z = -0.5 ;
   Z = -2.0.
   ```

2. Restructure your solution to the previous problem to write a procedure `quadratic_roots2(A, B, C, Z)` which obtains the same result as the previous exercise, but evaluates the discriminant only once. **Hint**: Evaluate the discriminant only once when `quadratic_roots2()` is first entered. Then solve the roots by passing the calculated discriminant as an argument to an auxiliary procedure having two rules.

1.2.5 List Processing

The recipe for processing lists remains the same as in Scheme. Usually:

- The empty list corresponds to the base case.
- A non-empty list is processed by recursively processing the tail and then combining the head of the list with the result of the recursive processing.

1. Write a procedure `sum_lengths(Ls, Z)` which matches Z with the sum of the lengths of the top-level sublists constituting elements `Ls`.

```
?- sum_lengths([[1, 2], [3, [4, 5], 6], [7, 8], [9]], Z).
Z = 8.
```

   This is the results of adding the sublist lengths `2 + 3 + 2 + 1`.

2. Write a Prolog procedure `sum_areas(Shapes, Sum)` which matches `Sum` with the sum of the shapes in list `Shapes`. The following shapes should be supported:

```
circle(Radius)
rectangle(Width, Height)
square(Width)
```

```
?- sum_areas([rectangle(3, 4), circle(1), square(3)], Sum).
Sum = 24.141592653589793.

?- sum_areas([triangle(3, 4, 5), circle(1), square(3)],
             Sum).
false.
```