Logical Programming in Prolog

Assignment 4

Juan Pablo Zendejas

May 3rd, 2024

In this assignment, I was tasked with creating Prolog predicates that query the database of employees given in a CSV file. I created 15 rules as specified in the assignment PDF. Overall, this assignment was very interesting to me. Prolog is very different from any other programming language I've used, even Haskell. Getting used to the eccentricities of SWI-Prolog and reading its documentation gave me a lot of insight. In addition, my background of formal logic from Discrete Mathematics taken here at SDSU gave me a lot of parallels that helped me understand how to write code.

I did not receive any external help except from consulting the SWI Prolog handbook/documentation found on their website. This documentation was very useful to figure out what predicates to use to transform some of the data.

1 CSV File Importing

First I had to change the `ReadCSV.pl` file to import all the rows from the CSV, ignoring the first title row. The source code is displayed in Listing 1.

This part wasn't too bad. I simply had to ignore the first row, which I did by using Pattern Matching on the `csv_read_file` predicate to ignore the first entry of the list and simply take the tail as the `Rows` variable.

Then, on `process_row`, I simply filled in the `assert` call with all the names of the columns.

2 Writing Rules

Now, I was tasked with using the imported facts from the CSV and writing rules that would be used as queries to the dataset.

2.1 Rule One

Simple rule, displayed in Listing 2. The `:-` symbol is reminiscent of "yields" from mathematics. I use the underscore to discard the variables from the `people` predicate that I don't need, and just check if a person exists with that name and is from Seattle.

ReadCSV.pl

```prolog
:- use_module(library(csv)).

% Predicate to read data from a CSV file and store it as rules
read_csv_and_store(Filename) :-
    csv_read_file(Filename, [_|Rows], []),
    process_rows(Rows).

% Process each row in the CSV file and store data as rules
process_rows([]).
process_rows([Row|Rows]) :-
    process_row(Row),
    process_rows(Rows).

% Store data from a row as a rule
process_row(row(EEID, Name, Job, Department, Unit, Gender, Ethnicity, Age,
 ↪  Hired, Salary, Bonus, Country, City, Exited)) :-
    assert(person(EEID, Name, Job, Department, Unit, Gender, Ethnicity, Age,
     ↪  Hired, Salary, Bonus, Country, City, Exited)).

```

Listing 1: CSV reader.

is_seattle_employee

```prolog
is_seattle_employee(Name) :- person(_, Name, _, _, _, _, _, _, _, _, _, _,
 ↪  'Seattle',_).
```

Listing 2: Rule to determine if an employee is from Seattle.

is_senior_manager_in_it
```
is_senior_manager_in_IT(Name) :- person(_, Name, 'Sr. Manger', 'IT', _, _, _,
↪  _, _, _, _, _, _, _).
```

Listing 3: Rule to determine if an employee is a senior manager in IT.

is_director_finance_miami
```
is_director_finance_miami(Name) :- person(_, Name, 'Director', 'Finance', _,
↪  _, _, _, _, _, _, _, 'Miami', _).
```

Listing 4: Rule to determine if an employee is a Director of Finance in Miami.

2.2 Rule Two

Very similar to rule one, displayed in Listing 3. Now, we just have more filters to apply, which is done by replacing some of the underscores with concrete values.

2.3 Rule Three

Displayed in Listing 4. Standard like the previous rules.

2.4 Rule Four

This rule is a little more complicated. It's displayed in Listing 5. Since this predicate has more arguments, I have them directly wired into the `person` predicate. Then, I simply have conditions that specify the values of those predicates. In addition, I had to do this for `Age` because I need to check if it's greater than 40.

2.5 Rule Five

This rule is given an employee ID, and should print a greeting for that employee. The source code is displayed in Listing 6. I looked in the SWI-Prolog documentation to figure out how to format a string, and I found the `format/2` predicate. It basically uses the symbol to define placeholders. Then, using the `person` predicate to get the data from the employee ID and outputting it with `format/2` was pretty easy.

is_asian_US_manufacturing_40M
```
is_asian_US_manufacturing_40M(Name, Unit, Gender, Ethnicity, Age) :-
↪  person(_, Name, _, _, Unit, Gender, Ethnicity, Age, _, _, _, _, _, _),
↪  Age > 40, Unit = 'Manufacturing', Gender = 'Male', Ethnicity = 'Asian'.
```

Listing 5: Rule to determine if an employee is a 40 or older Asian male working in manufacturing in the US.

greet
```
greet(EEID) :- person(EEID, Name, Job, Dept, Unit, _, _, _, _, _, _, _, _,
↪  _), format('Hello, ~w, ~w, ~w, ~w!~n', [Name, Job, Dept, Unit]).
```

Listing 6: Rule to greet an employee from their ID.

years_until_retirement
```
years_until_retirement(Name, Age, Until) :- person(_, Name, _, _, _, _, _,
↪  Age, _, _, _, _, _, _), Until is 65-Age.
```

Listing 7: Rule to determine an employee's years until retirement.

2.6 Rule Six

This rule calculates an employee's number of years until retirement. The retirement age is given as 65. This was fairly simple, I had to use the `is` operator to calculate a value and unify it with the predicate's argument. The source code is displayed in Listing 7.

2.7 Rule Seven

This rule is similar to the previous ones, but now I define two conditions on the age, using the less than and greater than operators. Prolog is interesting in this regard because it has the order of the symbols swapped from what it usually is. Something to remember for sure. The source code is displayed in Listing 8.

2.8 Rule Eight

This rule introduced me to the semicolon operator, and how to use it to make more complicated rules. Basically, the semicolon operator tells Prolog to backtrack and try other possibilities if one of the dependent propositions is false. I can wrap stuff in parentheses and use the semicolon to define "or"-like statements. The source code is displayed in Listing 9.

is_rd_black_midAge
```
is_rd_black_midAge(Name, Unit, Ethnicity, Age) :- person(_, Name, _, _, Unit,
↪  _, Ethnicity, Age, _, _, _, _, _, _), Ethnicity = 'Black', Unit =
↪  'Research & Development', Age >= 25, Age =< 50.
```

Listing 8: Rule to determine if an employee is middle-aged, black, and in research and development.

is_ITorFin_PHXorMIAorAUS
```
is_ITorFin_PHXorMIAorAUS(Name, Dept, City) :- person(_, Name, _, Dept, _, _,
↪  _, _, _, _, _, _, City, _),
        (Dept = 'IT'; Dept = 'Finance'),
        (City = 'Phoenix'; City = 'Miami'; City = 'Austin').
```

Listing 9: Rule to determine if an employee is in IT or finance, and in Phoenix or Miami or Austin.

is_female_senior_role
```
is_female_senior_role(Name, Title) :- person(_, Name,
↪  Title,_,_,'Female',_,_,_,_,_,_,_,_), atom_concat('Sr.', _, Title).
```

Listing 10: Rule to determine if an employee is both female and in a senior role.

2.9 Rule Nine

This rule introduced me to the `atom_concat` predicate. It was really interesting to learn that in Prolog, predicates like these can work in different ways based on which variable needs to be unified. Thus, even though the predicate is called `concat`, it can also split. That's what I did here to remove the 'Sr.' from the employee's title. The predicate is only true (can be unified) when the title starts with Sr. The source code is displayed in Listing 10.

2.10 Rule Ten

For this rule, I had to create a helper predicate `convert_salary/2`, which is shown in Listing 11. Now, I had to write predicates that reasoned with numbers instead of just atom equality. The source code is shown in Listing 12. This can be done with typical symbols like less than, greater than. I also have to write code to convert the string representation of the salary into a number; this was done by first removing the dollar sign, then removing the comma and parsing it into a number. The SWI Prolog documentation helped here.

2.11 Rule Eleven

This rule determines if an employee has an age that's prime. This one was difficult. I thought I could use an algorithm similar to Haskell, but that was a generator. I ended up using a naive recursive algorithm that checks if all the numbers less than the age do NOT divide the age. Probably inefficient, but it works. The source code is shown in Listing 13.

```
convert_salary
convert_salary(In, Out) :- atom_concat('$', NoDollar, In),
        % Split by commas, and additionally remove trailing whitespace
        split_string(NoDollar, ',', '\t\s', NoCommaList),
        % Concatenate the list back into an atom.
        atomic_list_concat(NoCommaList, SalaryAtom),
        % Convert the atom into a number.
        atom_number(SalaryAtom, Out).
```

Listing 11: Rule to convert a salary string to a number.

```
is_highly_paid_senior_manager
is_highly_paid_senior_manager(Name, Salary) :- person(_, Name, 'Sr. Manger',
 ↪  _, _, _, _, _, _, Salary, _, _, _, _),
        convert_salary(Salary, SalaryN),
        SalaryN > 120000.
```

Listing 12: Rule to determine if an employee is a highly paid senior manager.

```
is_prime_age
% Divisibility test
divides(N,X) :- 0 is X mod N.

% prime/2 is helper predicate.
% it is a recursive algorithm that tests all the numbers from N up to x,
% ensuring they all do not divide N.
% Base case.
prime(N, N) :- true, !.
% X should not divide N, and the next number also should not.
prime(X, N) :- not(divides(X, N)), Next is X+1, prime(Next, N).

% prime/1 calls prime/2.
prime(N) :- prime(2,N).

is_prime_age(Name, Age) :- person(_, Name, _, _, _, _, _, Age, _, _, _, _, _,
 ↪  _), prime(Age).
```

Listing 13: Rule to determine if an employee has a prime age.

```
total_salary
total_salary(Name, Salary) :- person(_, Name, _, _, _, _, _, _, _, AnnSal,
↪  BonusP, _, _, _),
        convert_salary(AnnSal, AnnSalN), atom_concat(Bonus, '%', BonusP),
         ↪  atom_number(Bonus, BonusN),
        Salary is AnnSalN + (AnnSalN * (BonusN / 100)).
```

Listing 14: Rule to determine an employee's total salary.

```
takehome_salary
takehome_salary(Name, Title, Salary) :- person(_, Name,
↪  Title,_,_,_,_,_,_,_,_,_,_,_),
        % Get the total salary.
        total_salary(Name, AS),
        % Semicolon causes backtracking.
        (
                % "One of these clauses should be true"
                AS =< 50000, Tax = 20/100;
                AS > 50000, AS =< 100000, Tax = 25/100;
                AS > 100000, AS =< 200000, Tax = 30/100;
                AS > 200000, Tax = 35/100
                % Unified value for Tax is then used here.
        ), Salary is AS - (AS * Tax).
```

Listing 15: Rule to determine an employee's take-home salary.

2.12 Rule Twelve

This rule calculates an employee's salary after bonuses, but before taxes. Now that I wrote `convert_salary/2`, this predicate has a fairly simple and easy to read implementation. It is displayed in Listing 14.

2.13 Rule Thirteen

Now, this rule calculates the take-home salary of an employee. This is done by using a tax bracketed system based on the employee's total salary. Again, I had to use the semicolon to provide backtracking and alternatives for Prolog, which worked as a conditional to set the tax level. The code is displayed in Listing 15.

2.14 Rule Fourteen

This was a pretty complicated predicate to write, but it was fun. The source code is displayed in Listing 16. I used the SWI-Prolog documentation to figure out how to use the date-time objects and predicates. One of the difficult parts is the strange date format

total_years
```
total_years(Name, Years) :- person(_,Name,_,_,_,_,_,_,Hire,_,_,_,_,Exit),
        % parse timestamp to get the year number
        split_string(Hire, '/', '', [_,_,HireYrS]),
        atom_number(HireYrS, HireYrShort),
        % since it's a short number, add 1900 if > 50
        % but otherwise add 2000
        (HireYrShort < 50, HireYr is HireYrShort + 2000;
                HireYr is HireYrShort + 1900),
        % find "exit" year
        (Exit = '', get_time(T), stamp_date_time(T, DT, local),
         ↪  date_time_value(year, DT, CurYear), Years is CurYear - HireYr, !;
                % parse exit year
                split_string(Exit, '/', '', [_,_,ExitYrS]),
                atom_number(ExitYrS, ExitYrShort),
                (ExitYrShort < 50, ExitYr is ExitYrShort + 2000;
                        ExitYr is ExitYrShort + 1900),
                        Years is ExitYr - HireYr, !).
```

Listing 16: Rule to determine an employee's years of service.

in the CSV, which I had to parse using `split_string/4` and some pattern matching. Then, I had to convert the short year number into a year in the 20th or 21st century.

2.15 Rule Fifteen

Finally, I had to write a rule to determine the average salary for an entire job title. The source code is displayed in Listing 17. Using the ways of Prolog backtracking, this actually wasn't too bad. Essentially, I wrote a predicate that simply gets the salary for a specific person with a title. There are many ways to unify the Salary variable that make the predicate true. Then, I use the Prolog `bagof/3` predicate to collect every possible value that satisfies the predicate into a list. Then, getting the average is easy.

```
average_salary
title_salary(Title, Salary) :- person(_,_,Title,_,_,_,_,_,_,SalaryS,_,_,_,_),
↪  convert_salary(SalaryS, Salary).

% bagof() gets al the possible salaries for a title. Then, i use foldl
% to sum up all of the salaries, and divide by the number of salaries,
% which is the formula for the average.
average_salary(Title, AvgSalary) :- bagof(Salary, title_salary(Title,
↪  Salary), SalList), foldl(plus, SalList, 0, Total), length(SalList, N),
↪  AvgSalary is Total / N.
```

Listing 17: Rule to determine the average salary for a job title.