EECS 3401: Report 3 Documentation

Camillo John (CJ) D'Alimonte
(212754396 - cjdal34)
&
Dinesh Kalia
(213273420 - dinesh49)

November 5, 2015

Contents

1 Abstract

This report summarizes the methodologies used to complete the series of exercises provided in the *Report 3 Specification*. It provides a detailed analysis as to the reasoning behind certain chosen heuristics, the successful implementation of various predicates as well as any difficulties encountered throughout the exercises.

2 Exercise 1

2.1 Description of Problem

This exercise serves as a continuation of the cousins problem outlined in *Exercise 6* of *Report 1*. A predicate **cousins_acc(Name1, Name2, P, Q)** is implemented to find two paths from the root of the family tree to the names of descendants using a predicate, **pathFromRoot_acc(Name, Path)**, which utilizes an accumulator to find such a path. Once found, the common ancestor is removed from the two paths using the predicate **removeCommonPrefix(Path1, Path2, Unique1, Unique2)**. The leftover values of Unique1 and Unique2 are then used to determine P and Q. A sample family tree, as shown in Figure 1, was provided in the *Report 1 Specification*.

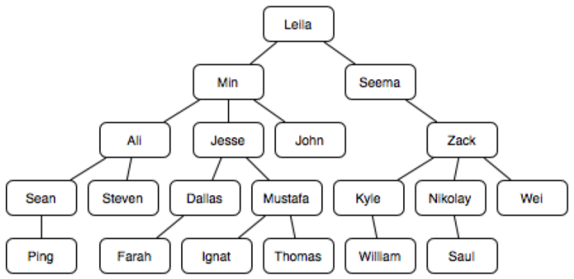

Figure 1: A simple family tree

2.2 Design and Implementation

The design of *Exercise 1* began with the implementation of the parent facts that were previously defined in *Exercise 6* of *Report 1*. This set of facts allowed for the creation of a sample family tree that would be later used for the correctness testing of the program. Once this was complete, the next goal was to develop the **pathFromRoot_acc(Name, Path)** predicate using an accumulator. In Prolog, accumulators are a device that allow for the reversing of information flow for some fundamental algorithm. This manipulation in the data flow results in an increase in the efficiency of the program. For this exercise, Leila can be classified as the "root" of the tree and the accumulator can be set to [] as a base case. The names of each current node of the family tree are then recursively appended to the accumulator, thus "accumulating" a list of cousins with respect to the root node.

The **removeCommonPrefix(Path1, Path2, Unique1, Unique2)** predicate was the next subproblem to consider. The thought process behind the implementation of the predicate is shown below in Figure 2.

Figure 2: Removing the common prefix from two lists

The last step in the implementation of *Exercise 1* was the integration of all the previously defined predicates into **cousins_acc(Name1, Name2, P, Q)**. The first step in the solution was to obtain the paths for each name defined by the predicate arguments. Referenced as Path1 and Path2 respectively, the next process was removing the common prefix from both sets of paths. Once each path was reduced of any duplicates, the length of each unique path was calculated by using the built-in **length** predicate. The final step was to determine P and Q by utilizing the formulas previously defined in *Exercise 6* of *Report 1*, as shown below.

$$p = min(|C - A|, |C - B|) - 1$$
$$q = ||C - A| - |C - B||$$

The solution to this exercise for the calculation of P and Q was very similar to the solution obtained in *Exercise 6* of *Report 1*, as demonstrated in Listings 1 and 2..

Listing 1: Exercise 6 Report 1

```
cousins(Name1, Name1, P, Q) :- P is -1, Q is -1.
cousins(Name1, Name2, P, Q) :-
ancestor(Name1, Ancestor), ancestor(Name2, Ancestor),
distance(Name1, Ancestor, D1), distance(Name2, Ancestor, D2),
Dist1 is abs(D1), Dist1 >= 2, Dist2 is abs(D2), Dist2 >= 2,
P is (min(D1,D2) - 1), Q is abs(Dist1- Dist2).
```

Listing 2: Exercise 1 Report 3

```
cousins_acc(N1,N2,P,Q) :-
    ( pathFromRoot_acc(N1,P1),
          pathFromRoot_acc(N2,P2),
          removeCommonPrefix(P1,P2,A,B),
      length(A,L1),length(B,L2),
    (
          ((0 is L1 ; 0 is L2) , Q is -1 , P is Q , !) ;
          ((L1 >= L2 , P is L2 , Q is L1 - L2 ,!);
          (L2 >= L1 , P is L1 , Q is L2 - L1,!))
    )) ; Q is -1, P is Q, !.
```

The striking similarity between the two snippets of code proves that various problem-solving methodologies can lead to a specific correct solution.

2.3 Difficulties Encountered

There were a few minor difficulties encountered during the implementation of *Exercise 1*, particularly with the development of the **removeCommonPrefix(Path1, Path2, Unique1, Unique2)**. Defining the various recursive cases used in the predicate was a bit of a struggle in the beginning but after drawing a few diagrams in an attempt to visualize the solution, the thought process became much more clearer. Minor setbacks were also experienced during the development of the **cousins_acc(Name1, Name2, P, Q)** predicate as the **length** predicates were originally not included in the development. It was not until tracing over the predicate numerous times that it became evident that the values of length were off and thus resulting in invalid solutions. It is also worth mentioning that the calculations with regards to L1 and L2 were a bit tedious as it took multiple reviews to ensure that no miscalculations were made (ie: Q is set to L1-L2 instead of L2-L1 or that ! was utilized correctly)

2.4 Testing

The following bullet points detail the reasoning behind the certain test cases used for *Exercise 1*.

- The test cases defined in Listing 3 pertain to the testing of various boundary cases. In the specification for *Exercise 6* of *Report 1*, it is noted that if the common ancestor is too close or if there in no common ancestor then return $P = Q = -1$. The test cases below all test different invalid relationships and thus should all return -1. If the name is not defined in the parent facts or if no value is stored ([] or [_]) then -1 should also be returned as there is no valid solution for the values of P and Q.

  Listing 3: Boundary Cases

  ```
  test(no_common_ancestor_or_ancestor_is_too_close, [nondet])
  :- cousins_acc(jesse, jesse, -1, -1).
  test(name_not_defined_in_facts, [nondet])
  :- cousins_acc(jimmy, jimmy, -1, -1).
  test(name_is_empty_list, [nondet])
  :- cousins_acc([],[], -1, -1).
  test(name_is_anonymous_variable, [nondet])
  :- cousins_acc([_],[_], -1, -1).
  test(one_valid_name_and_one_invalid_name, [nondet])
  :- cousins_acc(dallas, joe, -1, -1).
  ```

- The test cases defined in Listing 4 pertain to the testing of various valid cousin relationships, specifically in terms of Thomas. The P and Q values should coincide with the appropriate values defined in the structure of the family tree. These test cases are identical to the ones written for *Exercise 6* of *Report 1* since the program's output should not of changed.

  Listing 4: Valid Relationships

  ```
  test(valid_cousin_input_test_one, [nondet])
  :- cousins_acc(thomas, farah, 1, 0).
  test(valid_cousin_input_test_two, [nondet])
  ```

```
:- cousins_acc(thomas, zack, 1, 2).
test(valid_cousin_input_test_three, [nondet])
:- cousins_acc(thomas, nikolay, 2, 1).
test(valid_cousin_input_test_four, [nondet])
:- cousins_acc(thomas, saul, 3, 0).
test(valid_cousin_input_test_five, [nondet])
:- cousins_acc(wei, dallas, 2, 0).
```

- The test cases defined in Listing 5 pertain to the testing of the functionality of the **cousins_acc(Name1, Name2, P, Q)** predicate. 3 cases are used to determine if the correct path is formulated using the accumulator while a lone test case is used to ensure that the predicate will fail on an invalid input not defined in the family tree.

Listing 5: Functionality of Accumulator

```
test(path_accumulator_1) :- pathFromRoot_acc(thomas, P), write(P), nl.
test(path_accumulator_2) :- pathFromRoot_acc(saul, P), write(P), nl.
test(path_accumulator_3) :- pathFromRoot_acc(ignat, P), write(P), nl.
test(path_accumulator_4, [fail])
:- pathFromRoot_acc(dinesh, P), write(P), nl.
```

- The test case defined in Listing 6 pertains to the testing of the **removeCommonPrefix(Path1, Path2, Unique1, Unique2)** predicate. Two valid, arbitrary paths are given consisting of a common prefix. The predicate then removes the common prefix, Leila, from the path. The predicate is successful and thus, the test case passes.

Listing 6: Correct Paths & Common Prefix Removed

```
test(write_test) :-
            P1 = [leila,min,jesse,mustafa],
            P2 = [leila,seema,zack,nikolay],
            removeCommonPrefix(P1, P2, U1, U2),
            write(P1), nl,
            write(P2), nl,
            write(U1), nl,
            write(U2), nl.
```

3 Exercise 2

3.1 Description of Problem

Similar to *Exercise 1*, this question once again serves as a continuation of the cousins problem outlined in *Exercise 6* of *Report 1*. In this particular instance, the predicate that returns the two paths from the root of the tree to the names of descendants is implemented with a hole instead of an accumulator. This predicate is defined as **pathFromRoot_hole(Name, Path)**. The same auxiliary predicates used in *Exercise 1* are once again utilized in this exercise.

3.2 Design and Implementation

Since *Exercise 2* is a direct variation of *Exercise 1*, there was no need to redefine either the family structure nor the **removeCommonPrefix(Path1, Path2, Unique1, Unique2)** predicate. Instead, the only changes made were to **cousins(Name1, Name2, P, Q)** since it no longer used the **pathFromRoot_acc(Name, Path)** predicate but rather, used the **pathFromRoot_hole(Name, Path)** instead.

Fundamentally, **pathFromRoot_hole(Name, Path)** only consisted of minor changes. One of these changes was the use of the built-in **insert** predicate which was used to directly insert names into the hole with each recursive call. Although the location of the hole inside the predicate was different, the "heart" of the program remained the same throughout each variation of implementation.

The calculations of the values of P and Q as well as the use of **removeCommonPrefix(Path1, Path2, Unique1, Unique2)** did not change from *Exercise 1*.

Differences between Hole and Accumulator Implementations:

Accumulators are variables that are used in recursion to count or build a structure such as a conventional (standard) list. Holes are used in difference lists where a list is represented as a pair of terms, front and back. More formally, a difference list is a standard list except that the end of it is a logic variable, paired with that specific variable. An example of the structure of a hole is as follows: $[a, b, c\,|E] - E$. Holes are stored as queues while accumulators are stored as stacks as the items are stored in the reverse order in which they are found.

3.3 Difficulties Encountered

Since the majority of the code was reused from *Exercise 1*, there were no new difficulties faced. Although it took a while to understand how to implement a hole, which appeared to be more difficult to program than an accumulator, there was no overwhelming difficulty in the exercise.

3.4 Testing

The following bullet points detail the reasoning behind the certain test cases used for *Exercise 2*. Many of the test cases used in *Exercise 1* were also used in this exercise as the output of the program should not of changed with the use of a hole instead of an accumulator.

- The test cases defined in Listing 7 pertain to the testing of various boundary cases. In the specification for *Exercise 6* of *Report 1*, it is noted that if the common ancestor is too close or if there in no common ancestor then return $P = Q = -1$. The test cases below all test different invalid relationships and thus should all return -1. If the name is not defined in the parent facts or if no value is stored ([] or [_]) then -1 should also be returned as there is no valid solution for the values of P and Q.

Listing 7: Boundary Cases

```
test(no_common_ancestor_or_ancestor_is_too_close, [nondet])
:- cousins_hole(jesse, jesse, -1, -1).
test(name_not_defined_in_facts, [nondet])
:- cousins_hole(jimmy, jimmy, -1, -1).
test(name_is_empty_list, [nondet])
:- cousins_hole([],[], -1, -1).
test(name_is_anonymous_variable, [nondet])
:- cousins_hole([_],[_], -1, -1).
```

```
test(one_valid_name_and_one_invalid_name, [nondet])
:- cousins_hole(dallas, joe, -1, -1).
```

- The test cases defined in Listing 8 pertain to the testing of various valid cousin relationships, specifically in terms of Thomas. The P and Q values should coincide with the appropriate values defined in the structure of the family tree. These test cases are identical to the ones written for *Exercise 6* of *Report 1* since the program's output should not of changed.

Listing 8: Valid Relationships

```
test(valid_cousin_input_test_one, [nondet])
:- cousins_hole(thomas, farah, 1, 0).
test(valid_cousin_input_test_two, [nondet])
:- cousins_hole(thomas, zack, 1, 2).
test(valid_cousin_input_test_three, [nondet])
:- cousins_hole(thomas, nikolay, 2, 1).
test(valid_cousin_input_test_four, [nondet])
:- cousins_hole(thomas, saul, 3, 0).
test(valid_cousin_input_test_five, [nondet])
:- cousins_hole(wei, dallas, 2, 0).
```

- The test cases defined in Listing 9 pertain to the testing of the functionality of the **cousins_hole(Name1, Name2, P, Q)** predicate. 3 cases are used to determine if the correct path is formulated using the hole while a lone test case is used to ensure that the predicate will fail on an invalid input not defined in the family tree.

Listing 9: Functionality of Hole

```
test(path_hole_1) :- pathFromRoot_hole(thomas, P), write(P), nl.
test(path_hole_2) :- pathFromRoot_hole(saul, P), write(P), nl.
test(path_hole_3) :- pathFromRoot_hole(ignat, P), write(P), nl.
test(path_hole_4, [fail])
:- pathFromRoot_hole(dinesh, P), write(P), nl.
```

- The test case defined in Listing 10 pertains to the testing of the **removeCommonPrefix(Path1, Path2, Unique1, Unique2)** predicate. Two valid, arbitrary paths are given consisting of a common prefix. The predicate then removes the common prefix, Leila, from the path. The predicate is successful and thus, the test case passes.

Listing 10: Correct Paths & Common Prefix Removed

```
test(write_test) :-
            P1 = [leila,min,jesse,mustafa],
            P2 = [leila,seema,zack,nikolay],
            removeCommonPrefix(P1, P2, U1, U2),
            write(P1), nl,
            write(P2), nl,
            write(U1), nl,
            write(U2), nl.
```

4 Exercise 3

4.1 Description of Problem

This exercise ulilizies the predicate **chat** defined in **chatbasis.pl** to construct a Prolog program that can read and respond to English sentences. Using Prolog's built-in DCG translator, one can parse and respond to clauses to handle various types of sentences. For this exercise, the sentences will be of the following structure:

- Person has (a, an) object1 for a object2.
- How many objects does Person have?
- Who has object?

4.2 Design and Implementation

The implementation process of this exercise was broken down into two distinct parts. The first part focused on the creation of the various support predicates that were to be used in the creation of the main chat program. The second part of the design process was the implementation of the "heart" of the chat program which consisted of the specific parsing rules and response behaviour.

While programming the support predicates, much of the focus was put on ensuring that the correct grammatical notations were used. In particular, the overwhelming majority of time was spent focused on predicates designed to determine if characters were vowels, if words were singular or plural or if certain characters had to be altered from upper case to lower case or vice versa. The use of ASCII characters, as shown in Listing 11, proved to be extremely valuable for many of the support predicates.

Listing 11: Grammar Predicates using ASCII

```
vowel(N) :- name(N, [First | _]),
                  (First is 97;  %a
                  First is 101;  %e
                  First is 105;  %i
                  First is 111;  %o
                  First is 117). %u

lowercase([], []).
lowercase(U, L) :-
name(U,[H|T]), H < 91, H > 64, J is H + 32, name(L,[J|T]).
```

Once the support predicates were complete, the focus shifted to implementing the parsing and response capabilities. Although the majority of the predicates were already provided to us, there were still a handful of predicates that needed to be written. The first step was defining specific rules that would guide the process of parsing the input instances. Given multiple different sentence types, a parsing clause had to be developed for each specific sentence type. For example, for sentence type 3 (Who has object?), the parse clause divided the input into 4 distinct parts: Who, has, type(T), ?. The type(T) was the specific type of object that the person had. Once the string was parsed, the next step was to determine if the object was either singular or plural, since the latter would lead to an "s" being concatenated to the end of the object string. Once complete, the final step was to determine what the final "Goal" was; in this case what person had the specific object that was queried. Only unique solutions were to be considered hence the use of a cut was needed. The complete implementation of the code is shown in Listing 12.

Listing 12: Parse of type: Who has object?

```
 parse(Clause) --> ['Who'], ['has'], type(T), ['?'],
{ p_to_s(T, Search), Goal =.. [ Search , Who, Object],
Clause =('?-'('Who has object',Goal)),!}.
```

The last part of the implementation process was the design of the respondTo clauses which acted as a set of rules to notify the user of what the parse rules did. If the structure of the sentence is defined by the parse rules and if the query succeeds, then a sentence is displayed with the specific solution to the query. In the "Who has object?" example above, a unique list of solutions would be printed if a person, defined as Who, did in fact have the object. If the Goal was unsuccessful, then print a "No one" message indicating that the object is not owned by anyone. The complete implementation of this example is shown in Listing 13.

Listing 13: Parse of type: Who has object?

```
respondTo('?-'('Who has object', Goal)) :-
Goal =.. [T, Who, Object], Newgoal =..[findall, Who, Goal, List],
(Newgoal-> List \= [],  make_unique(List,UniqList), output(UniqList);
write('No One') ),
! , nl , nl.
```

4.3 Difficulties Encountered

Overall, this exercise was the hardest of the three thus far. Although large amounts of code were provided, it was often difficult to understand what was truly happening inside the predicates. The programming for the type 1 and 2 sentences went well enough that they executed correctly, however, programming the type 3 sentences was not entirely successful. Specific examples of the errors are included in *Section 4.4*.

4.4 Testing

The testing of *Exercise 3* was done entirely through the command line. The interactions are listed below.

- Test cases were run based on the example input given in the *Report 3 Specifications*. These test cases run specifically on type 1 and 2 sentences. All output is identical to what was expected.

  Listing 14: Command Line Testing

  ```
  1 ?- [report_3].
  true.

  2 ?- chat.
  |: Alice has an owl for a pet.
  Ok
  |: Alice has a cat for a pet.
  Ok
  ```

```
|: Bob has an ashtray for a plate.
Ok
|: Bob has a tray for a plate.
Ok
|: How many plates does Bob have?
Bob has 2 pets.

|: How many pets does Alice have?
Alice has 2 pets.

|: Who has pets?
Alice has pets
```

- Various test cases were run on multiple input types. All output is as expected for both type 1 and 2 sentences. Invalid input is also tested and appropriate error messages are displayed, thus proving that the program is robust enough. Things go astray once testing begins on type 3 sentences or more specifically, “Who has objects?” Although the pet instances work correctly, asking who has trays results in an error exception. Moreover, asking who has plates, which is irrelevant to the facts stored in the program, returns the people who have pets.

Listing 15: Command Line Testing

```
1 ?- [report_3].
true.

2 ?- chat.
|: Jim has a dog as a pet.
Can't parse that.
|: Jim has dog as pet.
Can't parse that.
|: Jim has a dog for a pet.
Ok
|: EG4G
Can't parse that.
|: 34444VV
Can't parse that.
|: ____
Can't parse that.
|: .
Can't parse that.
|: Alice does not have a cat for a pet.
Can't parse that.
|: Jim has a cat for a pet.
Ok
|: How many pets does Jim have?
Jim has 2 pets.

|: Jim has a cow for a pet.
Ok
```

```
|: How many pets does Jim have?
Jim has 3 pets.

|: Lorry has a tray for a plate.
Ok
|: Michael  has a tray for a plate.
Ok
|: Sean has a tray for a plate.
Ok
|: Who has plates?
Sean has pets
Michael has pets
Lorry has pets


|: Who has pets?
Jim has pets


|: Who has trays?
ERROR: findall_loop/4: Undefined procedure: tray/2
   Exception: (14) tray(_G2310, _G2313) ?
```

5 Exercise 4

5.1 Description of Problem

This exercise builds on the chat program implementation from *Exercise 3*. In this particular problem, each DCG parse clause used in *Exercise 3* is to be replaced by a custom translation of the individual clauses. In other words, there should be no "–>" operator utilized in the implementation of *Exercise 4*. In order to accomplish this, the predicate **chat** in **chatbasis.pl** is used to invoke the **parse_t** predicate instead of the **parse** predicate. Although the inner workings of the implementation have changed, the output of the program should be identical to that of *Exercise 3*.

5.2 Design and Implementation

The implementation of *Exercise 4* fails to work and as such, there are no design practices to detail in this section. Refer to *Section 5.3* for more details into the difficulties experienced in this exercise.

5.3 Difficulties Encountered

Many roadblocks were encountered in *Exercise 4*. While the class lecture slides were referred to numerous times, specifically Slides CH-14 & CH-15, it would ultimately not assist in solving the various problems that were encountered. In the beginning, the plan was to understand how the code discussed in the lectures worked. One such example of the parse clauses discussed is included in the following listing.

Listing 16: Lecture Example

```
parse ( Clause , S , Srem) :-
thing ( Name , S , S0 ) , det5 ( S0 , S1 ) ,
type ( T , S1 , S2 ) , det6 ( S2 , Srem).!
thing( Name , S , Srem ) :- det7 ( S , Srem ).!
type ( T , S , Srem) :- det8 ( S , Strem).!
det5 ( [ is , a ] | St ] , St).!
det6 ( [ '.' ] | St ] , St ).
det7 ( [ Name | St ] , St ).!
det8 ( [ T | St ] , St ).
```

Using this example as a reference, an attempt was made to "duplicate" it/build off of it. The attempt is as follows:

Listing 17: Code Attempt for Exercise 4

```
parse_t(Clause, S, Srem) :-
                    thing(Name, S, S0),
                    det1(S0, S1),
                    type(T, S1, S2),
                    det2(S2, Srem).
thing(Name, S, Srem) :- det3(S, Srem).
type(T, S, Srem) :- det4(S, Srem).
det1([is, a] | St], St).
det2(['.'] | St], St).
det3([Name | St], St).
det4([T | St], St).
```

In the end, it was the failure to understand the inner workings of the translation clauses that resulted in the inability to complete this exercise.

5.4 Testing

The test cases for *Exercise 4* should be equivalent to those of *Exercise 3* since the output should be the same. With that said, since the implementation of *Exercise 4* does not work, there are no test cases to run.

6 Appendix

This report was jointly authored. Both members worked on the outline of each question together as a pair. The programming of the predicates was jointly done. Dinesh wrote the documentation and comments for each predicate while CJ wrote the test cases for each exercise.