Natural Language Processing Airport Assignment

Edward Fleri Soler

247696(M)

University of Malta

E-mail: edwardfsoler@gmail.com

Abstract

In this paper, we shall observe the features and functionality of a natural language processing system designed through the logic programming language of PROLOG. This system is designed to hold a database of incoming and outgoing flights from the Malta International Airport, and is based off of a definite clause grammar which can handle queries. These queries are computed and tested against the database of flight data to formulate the correct response. Additional features of this system involve attempts to maintain a dynamic database with data being dynamically updated by means of a Java program and requests to the Malta International Airport website.

1. Introduction

The assignment in-hand involves a number of tasks revolving around the handling of a natural language. Natural language processing systems allow ambiguous languages, such as the English language, to be handled in a non-ambiguous manner by a machine. This involves a number of different stages of language handling, whereby grammatical and syntactical rules must be checked and expressions translated into computational instructions. These instructions must then be handled by the back end of the system to return some valid answer to the expression.

In this paper, we shall start by introducing the tasks involved in creating this system, we shall analyse the various components making up the system in depth and we shall discuss any interesting observations made during the design and programming phase of this assignment.

2. Tasks

A number of steps were involved in the creation of this system. During the design phase of this system, the components of the program were split into 4 main parts: Building the PROLOG database and attempting to include the dynamic maintenance of the data, creating and handling the DCG grammar, inclusion and handling of semantics within the above mentioned grammar and building the database query system, which is designed to handle different types of questions in different manners.

We shall now break down each task and observe how each one was tackled.

2.1 Building the PROLOG Database

Following observations of the departure and arrivals page on the Malta International Airport website (Flight Arrivals and Departures, 2016), a decision was taken to base each outbound and inbound flight on the following criteria:

- Airline
- Source/Destination
- Flight number
- Aircraft model
- Scheduled time of arrival/departure
- Estimated time of arrival/departure
- Flight Remarks

This led to the formulation of the 'arrive' and 'depart' predicates, which each stored the above mentioned data in the respective order.

2.1.1 Database Assertions

The inbuilt PROLOG command 'asserta' was used to assert certain data, extracted from the 'arrive' and 'depart' predicates, into the PROLOG database. Assertions such as 'arriving('Air Malta')' were made to store crucial data which could later be queried to provide a response to a natural language question. The type of data and format in which it was stored, was decided upon following consideration of the type of queries which would later be made.

A number of assertions were made for each arrive or depart predicate found in the PROLOG file. These assertions were grouped up into a predicate called 'assertions()', which is called at the start of every query, so as to update the database.

However, prior to the calling of the 'assertions()' predicate, a predicate defined as 'emptyAll()' is invoked. This in-turn invokes the 'retractAll()' inbuilt predicate, which takes one of a number of predicates stored in the database. This results in all similar predicates, including the one passed as an argument, to be deleted from the database, emptying the database completely before fresh assertions are made.

```
assertions():- arrive(A,B,C,D,E,F,G), asserta(G(C)), asserta(arriving(A)),
assertions():- depart(A,B,C,D,E,F,G), asserta(G(C)), asserta(leaving(A)),
```

Figure 1: Snippet of 'assertions()' predicate

2.1.2 Dynamic Maintenance of the Database

In an attempt to improve the functionality of the system in hand, and to increase user efficiency, a Java program was designed to dynamically extract data from the website to keep the PROLOG data repository up to date. This system was designed to perform a GET request to the server holding the Malta International Airport website. This request would then return the source code of the page, which would be scanned line by line to extract the required flight data which would keep the system up to date. While the main idea was still implemented, changes to the structure, format and data layout of the website half-way through the design phase resulted in the initial plan being modified.

The above mentioned changes involved the changing of the type of data output on screen; namely the airline name and the aircraft model. Also, data was now output on screen through JavaScript interaction, which meant that source code requests were no longer possible. In an ideal scenario, an application programming interface could replace the need for a request, allowing for a much easier means of extracting the data.

Due to these changes, a number of alterations in the design and implementation of this part of the system were made. First, the source code of the page site must be manually hard copied into a text file, which is used as the source code repository instead of the GET request. The text is then scanned for key words, which allows for the required data to be located and extracted. This data is then formatted into the 'arrive'/'depart' predicate format which is then written to a text file.

```
arrive('KM 376','BERLIN - TEGEL','18:20','18:20',airborne).
arrive('TK 1372','ISTANBUL','18:55','18:45',airborne).
arrive('KM 2702','ISTANBUL','18:55','18:45',airborne).
arrive('FR 2278','BUDAPEST','19:10','N/A',closed).
arrive('IG 9242','CATANIA','19:45','N/A',n/a).
arrive('KM 642','CATANIA','19:45','N/A',n/a).
```

Figure 2: Part of Java program output in 'Flight Arrivals' file

Due to the absence of certain key data such as the aircraft model number, the extracted data could not be used in conjunction with the rest of the (by then already developed) system. Therefore, the output may simply be

observed in the text file, and may be made use of it future implementations of the system.

2.2 Development of the DCG Grammar

The definite clause grammar for this program was gradually built according to the typical type of questions which may be posed by an individual. The general structure of the grammar was first laid out, and more specific branches, as well as individual word phrases were added on as the grammar progressed. All grammar predicates were defined in the DCG format by means of the inbuilt '-->' expression.

The syntax tree was split into two main sub-trees, which further branched out into more specific expressions. The starting point (root) of all sentences was the non-terminal 's', which, in general, branched out into a noun-phrase followed by a verb-phrase, which respectively further branched out into a combination of nouns and other phrases and verbs and other phrases.

However, as progress was made with the type of sentences accepted by the system, more complex branches were mapped involving the following word groups:

- Conjunctions
- Auxiliaries
- Propositions
- Determiners
- Adjectives
- Quantity related words
- 'Wh' style words
- Verbs
- Nouns

Each individual word was then mapped to its appropriate group, which formed the basic sentence structure and was later tweaked to achieve a wider range of queries accepted.

A great deal of thought and time was spent on formulating these sentence structures, so as to provide flexibility in the manner, order and word use when querying the system. If more time was made available, further improvements and expansions to the DCG grammar of this system could allow for a more natural feel when interacting with it.

2.3 Addition of Semantics to the Grammar

The next step in the creation of a functional natural language system is the inclusion of semantic handling. The grammar so far defined simply accepts or a rejects a sentence, but does not do much else with it. What is missing is the semantic interpretation of the phrases posed.

The addition of semantics to the system is a pivotal part in this assignment, and is one of the more challenging tasks. In this system, semantics is handled by means of Beta-reduction, whereby verbs act as a semantic header (or function), which take one or more arguments, to which the function acts upon or corresponds to.

Firstly, the 'reduce' predicate was defined. The predicate is defined to take three arguments, with the first being the verb to be reduced, the second being the argument to apply to the verb, and the third being the result of the reduction application.

```
reduce(A^F,A,F).
```

Figure 3: Definition of the reduce predicate

Next, the verbs, which act as the semantic interpretation of a given sentence, are defined together with the number of arguments that they take, in the Beta-reduction format as follows:

```
v2(X^arriving(X)) --> [arrive].
v3(X^Y^arriving(Y,X)) --> [arrive].

v(X^arriving(X)) --> [flying].
v(X^leaving(X)) --> [flying].
```

Figure 4: Snippet of verb definitions in the Beta-reduction format

Verbs are reduced in order of the argument positions on the left-most side of the definition. In figure 4, we see that the first

reduction for the verb ‘v3’ ‘arrive’ is applied to the second position, while the second reduction operation will place an argument in the first position.

While defining the semantic representation and handling of verbs, both the sentence structure mentioned in part 2.2 and the database assertions mentioned in part 2.1.1 were constantly being redesigned and modified into more convenient and functional set ups.

Following the inclusion and modification of all the verbs, semantics was added to the sentence structure mentioned in part 2.2. Firstly, sentences were split up into three main groups which require three different handling techniques to retrieve an answer from the database. The first was the set of sentences which related to yes/no style questions. These were labelled as being sentences of type ‘q1’. ‘q2’ type sentences are the group of sentences starting with a ‘wh’ type word phrase, which meant that they would require some sort of textual answer from the database. These sentences would involve semi-completed semantic verbs, which would be used to query the database for answers. The final group of sentences (‘q3’) is the group of sentences which involved a quantitative analysis (counting). These sentences were listed to start with a quantity type operator.

Verb phrases, noun phrases and other word groups were regrouped into groups of type ‘vp2’, ‘vp3’, ‘np2’, ‘np3’ and so on, so as to ensure that certain sentences follow a certain path which will lead to the correct semantic interpretation (example queries which involve two parameters rather than one). Improvements were also made to consider the sentence as a whole when interpreting its meaning, so as to ensure that a correct response is formulated.

After a great deal of remodelling and reconsideration, a functional layout was achieved, and the greater half of the task of semantic inclusion was completed. Next, variables were passed to noun phrases, verb phrases and other word groups in the form of parameters which stored information which would make up the semantic interpretation. Verbs and proper nouns or other arguments were retrieved from the sentence paths, and were then passed to the ‘reduce’ predicate to form a semantic clause which will later be used to query the database. Sentences which involved semantic analysis of two parameters were directed through paths which contained two reduce clauses.

The outcome of a sentence being input is that of a semantic phrase being returned, which will be interpreted to form some sort of result.

```
s(q3(S)) --> num(_), np(NP), vp2(VP), aux(_), p(_), {reduce(VP,NP,S)}.
s(q3(S)) --> num(_), np(_), vp2(S).
s(q2(S)) --> wh(_), d(_), np(_), adj(_), np(_), vp1(S).
s(q2(S)) --> wh(_), d(_), np(_), vp(S).
s(q2(S)) --> wh(_), vp(S).
s(q2(S)) --> wh(which), np(_), vp3(S).
s(q2(S)) --> wh(_), np(NP), vp(VP), {reduce(VP,NP,S)}.
s(q1(S)) --> np(NP), vp(VP), {reduce(VP,NP,S)}.
s(q1(S)) --> np(_), vp(S).
```

Figure 5: Sentence grammar and semantics split up into the three question groups

2.4 Building the Database Query System

The final part of this assignment involved handling queries posed by the user, and returning a sensible result. The grouping of sentences into three sets made this section of the program more organised and efficient to handle.

The ‘interpret’ predicate is defined to separate sentences according to their question group, so that each group may be handled differently.

The group ‘q1’ is the simplest, and merely involves performing a call to the database with the given semantic clause (passed on

through the ‘interpret’ predicate). The inbuilt PROLOG command ‘call()’ is passed the semantic clause, and tests whether the given clause exists (in its entirety) in the database are conducted. If this is so, a simple ‘yes’ is returned, signalling to the user that their sentence, posed as a question, is true. Otherwise, ‘no’ is returned.

```
| ?- process([are,flights,arriving,from,'Rome'],A).
A = no

| ?- process([are,flights,arriving,from,'Zurich'],A).
A = yes
```

Figure 6: Handling of yes/no style questions

The second set of questions which must be handled, are those which require some sort of textual answer. Following the call of the ‘interpret’ predicate, the semantic clause is passed on to the ‘breakdown’ predicate, which further reduces the clause, passing a variable as an argument. Example:

reduce(X^landed(X),Z,Y)

would apply variable Z in the place of X. Next, the newly modified semantic clause is passed onto the inbuilt ‘findAll’ predicate, which returns all occurrences of a given term. This will return a list of all values which correctly replace X in the above example. The number of results returned is then handled, and output on screen for the user to observe.

```
| ?- process([what,is,the,status,of,flight,'FR 5226'],A).
A = airborne

| ?- process([which,airlines,arrive,from,'Zurich'],A).
A = ['Swiss International Airlines','Air Malta']
```

Figure 7: Handling of questions which require textual answers

The final form of database querying to be coded is that for quantitative analysis. The group of sentences ‘q3’ invoke the ‘counter’ predicate, which is defined to perform a counting action on a given semantic clause. First, the inbuilt PROLOG command ‘=..’ is called to transform the given semantic clause into a list, with the predicate being the head of the list and the arguments forming the tail. This is done for ease of handling by the predicates to come. Next, the ‘count’ predicate is hard-coded to count the number of existing predicates within the system which match a given semantic clause. The ‘findAll’ predicate is once again used to form a list of valid results for a given query. The length of the list is then calculated and the result is returned. Although further improvements could be made to this system, for the small set of quantitative questions which form part of this language, it functions properly.

```
| ?- process([how,many,flights,have,landed],A).
A = 3

| ?- process([how,many,'Ryanair',flights,are,there],A).
A = 4
```

Figure 8: Handling of quantity related questions

All parts of the program are then brought together under one single predicate called ‘process’. This predicate first empties the database, asserts all the data into the database, handles the sentence grammar and extracts its semantics, and passes on the semantic interpretation to formulate a response.

3. Discussion

Following the design, modelling and alteration of the system, a number of key observations have been made apparent. The first, is that the order in which such a grammar is formed is crucial. It has been understood that it is easier to define the grammar as a whole before coding of any functions or tasks, so as to avoid wasting time making future alterations.

Following the learning curve involved in the creation of this system, I now better understand which parts of the system work efficiently, and which could be improved. The counting predicate is inconveniently

hard-coded, meaning that any future expansion of the language would require further coding for counting for new semantic clauses. This could be improved by means of a better structured counting predicate which handles clauses in a tree-like fashion, having each clause follow a specific path to its respective counting method.

Improvements could also be made to the general grammar of the system, with the layout of some of the sentences becoming more challenging to understand and debug due to the similarity in the general structure, but difference in minute details.

4. Results and Future Improvements

Following rigorous debugging and testing of the program, it was apparent that the system performed its desired function with a decent set of sentences being accepted. Slight changes in sentence layout, order and word use was also handled, proving that a satisfactory level of flexibility was also present. I believe that a satisfactory result has been achieved through the design and implementation of this system.

The following is a list of general sentences which are handled by the system:[1][2]

- "has flight **AB 1234 landed**?"
- "has flight **AB 1234** been delayed?"
- "has the gate for flight **AB 1234 opened**?"
- "are flights departing to **Hawaii**?"
- "are flights arriving from **Budapest**?"
- "did any flights from **Heathrow** land?"
- "is Air Malta flying to **Zurich**?"
- "do Ryanair fly from **Paris**?"
- "When does **AB 1234 arrive**?"
- "What is the estimated time of departure of flight **AB 1234**?"
- "What aircraft is **AB 1234**?"
- "What is the status of flight **AB 1234**?"
- "What is the status of the flight to **Rome**?"
- "How many flights depart to **Barcelona**?"
- "How many flights are **airborne**?"
- "How many **Alitalia** flights are there?"
- "Which airlines depart to **Munich**?"
- "Where does **Fly Emirates** fly to?"

As was previously stated, future improvements to the grammar could see the list of accepted questions grow greatly, together with the level of flexibility in asking a question.

Having the database dynamically linked up with the website (as was attempted), could see this system have a proper use in the future, allowing for users to gain information regarding flights in a more natural fashion.

Improvements to the system could also see flight details be input into the system through natural language, with assertions being made on information given by the user.

5. References

Flight Arrivals and Departures. (2016). Retrieved from Malta International Airport: https://www.maltairport.com/passenger/flights-landing/arrivals-departures

SWI Prolog Documentation. (n.d.). Retrieved from SWI Prolog: http://www.swi-prolog.org/

[1] Sentences may be modified slightly in word use and order

[2] Bold words may be altered to achieve different results