Declarative Programming Summative Assignment '22-'23

A System for Optimising Restaurant Bookings

1 Introduction

This project is the final formally assessed exercise for MSc students on the Prolog course, first session 2022.

Restaurants in are an important part of Brussels life. The intent of this project is to implement an AI system to help a small bistro optimise its evening service and taking bookings on line in natural language input. This practical counts for 30% of the marks for this course. In itself, it is marked out of 100, and the percentage points available are shown at each section.

This exercise uses ideas from the sections of the MSc Prolog course in definite clause grammars (DCGs), meta-programming, constraint logic programming over integer finite domains (CLP(FD)), and constraint meta-programming.

The exercise is in four parts:

1. The design and construction of a Definite Clause Grammar (DCG) in Prolog which will analyse a small subset of English used to describe a set of restaurant bookings (30%). You are not assessed here on the linguistics of your grammar: what matters is that you make logical decisions on how to implement the grammar, not whether they are conventionally correct in terms of linguistics.

2. The interfacing of this DCG to the Finite Domains equation solver built into Prolog, so as to specify constraints on an evening in a restaurant, described below, and then generating a confirmed booking list (40%);

3. Testing of your software to make sure that it covers only grammatically correct and meaningful sentences (20%);

4. Displaying the bookings list for a normal evening. (10%)

To facilitate testing of your code by the assessors, you are required to include some entry points, as specified below. If you do not do so, a 20% penalty will be applied to your grade.

- `test_dcg/2` succeeds if its first argument is an input phrase in the above format, and its second argument is whatever your DCG returns to your program.

- `test_constraints/3` succeeds if its first argument is an input phrase in the above format, its second argument is whatever your DCG returns to your program, and its third argument is your program's internal representation of the evening schedule.

- `test_all/0` always succeeds, running your entire program on each of the test sentences above and printing out the solutions.

2 Project Scenario

The scenario for this project is as follows. *Restaurant XX* is a popular street-corner bistro in central Brussels. It has a very loyal following of hipsters, all of whom are way too cool to worry about the timing of their evening dinners—they know the food will be worth waiting for. Of course, some guests will have subsequent theatre or cinema engagements, so they have to be finished by a certain time; these guests will choose the "theatre menu" which can be served in a shorter time than a standard meal, because it has only two courses from a restricted selection.

The restaurant has decided to use an unusual booking approach: it will give its diners the option to pick a time, or to let the restaurant choose, so that optimality can be attempted. Booking takes place at least a day before, by SMS (text message), in natural language. The diners must specify the date, the number of people, optionally the menu ("standard" or "theatre"—if unspecified, the booking system will presuppose "standard" unless a better fit can be arranged using "theatre"), and then either a fixed start time, a preferred start time, or no start time (in which case the restaurant selects the time). The restaurant opens at 19:00 and closes at 23:00, and the kitchen is open all that time.

The restaurant, being small, has just three tables. The tables take up to two, up to three, and up to four diners, respectively. The restaurant does not expect different groups of diners to share a table. It's not possible to move tables together to serve groups, because of access paths between the tables. To allow enough time for the diners to enjoy their meals, the restaurant allows one hour for the theatre menu, and two hours for the standard menu.

Finally, bookings for each day are processed so as to fill the restaurant optimally for that evening. Unfortunately, that means that some prospective diners may be disappointed, because there is not enough space for them.

3 Sample booking requests

The following sentences are a sequence of bookings for one evening, the 18th of March (though your parser should be capable of parsing any date in the relevant formats). Your program should be able to read these sentences, represented as Prolog lists of words and numbers (not strings), as shown below (you can cut and paste this data to use in your program, but make sure that character codes are copied correctly). *That is, you do not need to write the code to translate from SMS text to Prolog input.* First, the natural text version:

Table for 2 at 20:00 on 18 March.
Please can we have a table for 3 for the theatre menu on March 18th?
We would like a table for 5 preferably at 8pm on 18/03.
Can I book a table at 9pm for 2 people on the 19th of March for the standard menu please?
Reserve us a table on March 18 for a party of 4 for the standard menu.
9 people on 18th of March.
Book 6 of us in on 18 March at 20:00.
Reservation for 7 on March 18 preferably for standard menu at 7 o'clock.

And then the pre-processed Prolog version for you to use:

```
[table,for,2,at,20,':',00,on,18,march]
[please,can,we,have,a,table,for,3,for,the,theatre,menu,on,march,18,th]
[we,would,like,a,table,for,5,preferably,at,8,pm,on,18,'/',03]
[can,i,book,a,table,at,9,pm,for,2,people,on,the,18,th,of,march,for,the,standard,menu,please]
[reserve,us,a,table,on,march,18,for,a,party,of,4,for,the,standard,menu]
[9,people,on,18,th,of,march]
[book,6,of,us,in,on,18,march,at,20,':',00]
[reservation,for,7,on,march,18,preferably,for,standard,menu,at,7,oclock]
```

4 NLP Methodology

Remember, when you are designing your parser, that the point of parsing is to find common patterns in the language and take advantage of them to provide generality. For example, a single DCG rule which matches the whole of the sentence "The man bites the dog." is not very useful because it can only match one sentence. Much more useful is to split up the sentence into a noun phrase ("The man") and a verb phrase ("bites the dog") because the rule that does this is very general, covering many English utterances.

So a good solution to this parsing problem will break down the sentences of the input into significant lumps, for example classing "on 18th March", "on March 18th" and "on 18/03" all together, and dealing with them as a category of dates.

Explain any assumptions you make when designing your grammar in comments.

5 Constraint system

You are required to derive from the provided text a set of constraints determining the diners' requirements, and then supplement this with a list of constraints that capture the restaurant's practical limitations and policies. To do this, you will need to use not only finite domain variables, but also reified constraints, using the techniques demonstrated in the lectures.

6 Testing Your Code

You should test your code using the sample data given above, as well as provide the specified test points. At the very least, your code should succeed on the provided sample data, but should also be able to handle other text of a similar form. Also include in your submission a short description of how you tested your program, providing a few example outputs that are commented out.

You may be tempted to generalize the system so that other restaurants with more tables and longer hours may use it as well. Indicate if your system is capable of this, and how to access that functionality. That being said, remember that you are ultimately graded with respect to *this* specification, so it must at least conform to this document.

7 Displaying the Results

Write a predicate that will print out the results of your program's analysis in a human-readable form. In this part of the practical, you may reasonably use a cut, because you will want to commit to the solution your search program has found, and not allow the user to force backtracking. Make sure you use the cut, in the right place, *for this purpose only.*

8 Documentation, Style, and Testing

Your code should have concise documentation in the comments for most predicates. This means you should describe the argument modes, like input (`+`), output (`-`), both (`?`), etc., (see https://www.swi-prolog.org/pldoc/man?section=preddesc), and a one or two line description of the predicate. There is no need for a paragraph description for each predicate. For example:

```
% append(?List1, ?List2, ?List1List2)
%   List1List2 is the concatenation of List1 and List2.
append(List1, List2, List1List2) :- ...
```

If a predicate has many different clauses, or the clauses are especially complex, it is also good to describe what each clause does separately, but this is not necessary in general. It is also helpful to indicate larger sections of the code that go together with a header, e.g., Definite Clause Grammar, Constraint System, Testing, etc.

Not only should your code conform to standard best practices of software engineering, like keeping predicates small, atomic, and non-redundant, but **your code should be written in declarative, idiomatic Prolog** as much as possible. This means taking advantage of unification and backtracking to leverage the built-in search in Prolog. If you find yourself frequently using `findall/3`, then rethink how you are approaching the problem to try to use backtracking and/or negation instead.

Avoid using cuts (`!`) and if-thens (`->`). Using these actually means your program is logically incorrect. So, unless you provide a very good reason, you will be penalized for their usage. If you think you need to use a cut to stop unwanted backtracking, you can usually solve this by making your clauses mutually exclusive. If you think you need to use an if-then, you can usually just split it into separate clauses instead.

Your program should have also good test coverage in order to be confident that it behaves as desired. This means there should be decent amount of unit tests for the most important predicates (you don't need to unit test every single predicate) and integration tests that test larger and larger portions together. The best way to do this is to write test predicates that call your other predicates to ensure they are working. Please put these predicates at the bottom of the source code; do not interlace them throughout.

9 Submission

You should combine together whatever files you develop for the different parts of the exercise into a **single file** and submit it **via Canvas by Friday, January 6th at 17:00.**

Please make sure that you place any text in your code file inside comment markers, so that the file will load without further editing. You will be penalised if you fail to do this. The submission must be made by the advertised deadline, via the Canvas submission system.

10 Referencing and Cheating

When programmers are stumped on a bug or issue, we often go straight to Google and adapt the resulting hits to our needs, often directly from StackOverflow. Being able to specify and solve your issues in this way is often useful, especially when working in the industry. However, in the academic setting, this makes it difficult not only to evaluate your competency as a Prolog programmer, but also to identify which code is actually yours. Therefore, if you need to adapt code from the web, simply provide a link to that resource (i.e. URL) along with a one-line description of where it is from and what it is. For example,

```
% StackOverflow: All combinations of a list
% https://stackoverflow.com/questions/41662963/all-combinations-of-a-list-without-doubles-
combs([], []).
... and so on.
```

Doing this whenever you take code from online removes any suspicion of cheating, and **any failure to do so will be regarded as cheating!** In addition, **you cannot share code** since this is an individual project; therefore, you cannot reference another student who has taken or is currently taking this course. There are a few places where referencing in this manner is not required:

- Lecture slides
- SWIPL built-ins, libraries, and documentation (This is encouraged!)
- Non-code or educational resources (e.g., an algorithm description without Prolog code)

Remember, being caught cheating can (and in this course does) lead to the exclusion of the guilty student from the university.