Declarative Programming Assignment 5 (Assessed)

A System for Timetabling in English

1 Introduction

This practical is the second formally assessed exercise for MSc students on the Prolog module. The intent is to implement an AI system for planning teaching timetables, derived from natural language input.

This practical counts for 50% of the marks for this module. In itself, it is marked out of 100, and the percentage points available are shown at each section. 5 marks will be deducted for every inappropriate use of `;`, for every red cut, and for any uncommented green cut.

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 school timetable (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 a timetable, and then generating timetables (40%);

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

4. Displaying the timetable for a normal week. (10%)

2 Input Syntax

The following are examples of the input syntax of the system. This constitutes the *minimum* coverage of the parser. If you wish to extend it further, do make sure that it covers at least the sentences and the kinds of reference below. Use all lower case for your input, so that you don't need to use quotes all the time.

```
prof smith teaches class c1.
he also teaches class c4.
prof jones teaches classes c1 c2 and c3.
she also teaches class c2.
class c1 is in room 1.
classes c1 c2 and c3 are in the same room.
classes c1 and c2 have the same teacher.
class a1 has 45 students.
room 102 seats 100 students.
class c1 is before class c2.
class c4 is after class c3.
classes c1 c3 and c4 are on the same day.
```

And there are some general rules:

- You can only refer to the person mentioned in the previous sentence using “he” or “she”. If there was no person mentioned in that sentence, an error should be thrown. (See the manual for how to do this.)
- A conjunction of more than 2 items, which would normally be written ”a, b and c” in English is written here without the comma: “a b and c”, to make it easier for you.
- The school has 10 teachers, 5 classes (30, 35, 100, 40, 50 students, respectively), and there are three classrooms, that seat 35, 60, and 100, respectively. Each teacher must teach each class in the week, and each teacher must have at least two hours of preparation time each day.
- And of course, all the obvious practical rules apply: a teacher cannot be in two places at once, two classes cannot be taught in the same room, and a class of 100 cannot use a room that can seat 40. You need to consider how to implement these common sense rules as constraints, and apply them where appropriate.

Explain any assumptions that you make in designing your grammar clearly in your comments. Here are some hints:

1. Let your parser generate representations of the meaning of sentences which look like Prolog goals to generate the necessary constraints. For example, the sentence

   prof jones teaches classes a1 b2 and c3.

   might generate the representation

   ```
   teacher( jones, [a1,b2,c3] ).
   ```

2. Once you have such a goal, you can either add new arguments to it, for input and output of the constraint system you’re building, or you can use it as the input to a meta-program that interprets the command.
3. You will probably want your DCG to produce a list of these goals as its output.
4. You will probably want to associate numbers with each of the people and things mentioned in the timetable specification, so that you can use CLP(FD) to solve the constraint problem.

3 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 the timetabling problem will break down the sentences of the input into significant lumps, for example classing “class …” and “classes … … and …”.

The best way to achieve these groupings is to list them all out and decide which representation of meaning you want to match them with.

4 Using the Parser

Once you have written out the rules that the parser will use, you need to be able to pass back the various meanings. You will need to find a way to take an expression like “classes a1 a2 and a3” and convert it into something like `[a1,a2,a3]` or `[1,2,3]` if your DCG is converting names to

numbers for you. So somewhere in your program, you will need to keep track of the names which have been used, and create a new Prolog variable for each one. This will mean keeping a list of all the variables used, and passing it around as your parser works, or doing the same in the code that works on the representation after it has been parsed. *Hint: remember that you can add extra arguments to DCG clauses just the same as in ordinary Prolog clauses, and pass values around in the way that you're used to in ordinary Prolog.*

There will be some situations where you need to use meta-programming predicates to test and.or create the sub-expressions which you work with. However, a good implementation will only do so when strictly necessary – normally, unification will suffice.

5 Designing the Program

Your program will consist partly of DCG clauses (the ones with `-->` instead of `:-`). It will have some bits of Prolog to be executed independently of the parser, enclosed in `{}`. It is left for you to work out what the structure of the grammar should be. There is no single correct answer: there are many correct ways of implementing this grammar.

You will also need to design predicates or clauses that implement the meaning of the various sentences in terms of constraints, so that you can build up a constraint system to express their combined meaning. You will probably find it helpful to think separately about the DCG part and the constraint part: so long as you define the semantic goals (see above) properly, then you can think of the two completely separately.

When you are working on this code, it may be useful to check up on the `clpfd:full_answer` switch, described in the SWI Prolog manual, section "Answer Constraints".

6 Testing Your Code

DCGs take lists of symbols as input. Remembering that everything has to start with a lower-case letter, you can set up your test data in a file as a predicate called, for example, `test/1`, whose argument is a list containing the words in your data, using the symbol `fullstop` to indicate the end of a sentence. So your test predicate might look something like this:

```
test(   [prof,smith,teaches,class,a1,fullstop,
         room,a3,seats,50,students,fullstop,
         …,
        ] ).
```

Then, when you have written your solver, which should be called `timetable/2`, you can call it as follows:

```
test( Data ), timetable( Data, Timetable ), print_timetable( Timetable ).
```

As part of this assignment, we will test your program on a set of timetable constraints that use

Include a typescript (see the UNIX command `script`) of your program evaluating these constraints.

Also include in your submission a short description of how you tested your program, including a typescript showing the tests being applied and the test results. Discuss how you know that the test results are correct.

7 Printing the Results

Write a predicate, `print_timetable/1` whose argument is a data structure that you have designed that represents a weekly schedule, where there are 5 classes of 1 hour in each day, for 5 days. Make up some data to demonstrate how effectively your scheduler solves difficult clashes and follows the instructions given.

8 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.

Please make sure that you place any text in your code inside comment markers, so that the file will load without further editing. If your file will not load directly into prolog, we will not debug it for you, and we will only mark whatever does load. The submission must be made by the advertised deadline.