Exam seating chart problem

Suhail Ghafoor
Arizona State University
Tempe, AZ
Suhail.Ghafoor@asu.edu

I. Introduction

The problem that is being solved is making a seating chart for **n** students that fits certain criteria defined by the user. The problem also involved coming up with a solution that was versatile enough to fit a variety of situations, for example different number of students, different number of seats, different gap between students and different class layouts. The whole point of the program is to simplify the task of seats assignment which meant the ability to easily customize the program to fit different requirements without spending too much time on the code. This was an essential requirement for the final solution. Lastly, the final output or solution that would be presented also needed to be in a format that could be directly applied in a practical situation.

II. Problem Formation

In the initial problem there were three variables to take care of: (1) the number of seats in a class room, (2) the gaps around different seats, (3) the names of students. Out of these three variables two of them which are seats and gaps would not change often and entering them every time when assigning seats was a hassle. So the next task was coming up with individual solutions to storing all of these variables since a single solution could not be applied to all of them because of the way they are entered or used. Third task was writing modular code that would let the user change the seating criteria on the fly without too much work. For example, the user should be able to start seating from seat number 1 and have minimum of 1 gap between seats or the seating could start from last seat and have a minimum gap of 2 between each student. Ultimately, the gaps were stored as facts and the seats and students were stored as lists.

III. High Level View of Code

The idea was to write the code in a way so each rule no matter how small would be kept separate and if needed to be called or skipped the user would have the option to do so. So the first iteration of the program had two rules and first rule took two variables, a list of variables which were essentially the names of students and a list of seats and each student was assigned the first available seat. This problem just filled the seats one by one with no criteria and it is still the core of the program. This simple rule just calls itself recursively until either all the students are assigned seats or the program runs out of seats before everyone can be assigned a seat. Later on more rules were added to the same first rule to build up the program.

A. *Criteria for seating*

The next step after basic program functionality was to have a criteria for seating i.e. add a gap between students and if two seats already had a gap between them then do not add a gap. To accomplish this task a simple rule was created that would take in a list of seats and then it had two cases.

- If there was gap between the next seat and current seat then call a rule that handles that condition.
- If there was not a gap between next seat and current seat then call a different rule.

The rules that were called by this rule could have been combined but keeping them separate meant that if the user wanted to change what should happen if there is or isn't a gap available then they would be able to change that easily. For example if the user wanted to make the minimum gap between every student 2 then this would be the part that would be changed.

B. *Makelist*

Initially, the program took a list of seats as an argument to compute results but it was not very intuitive since the list of seats were always from 1 to ***n*** and everything in between. This meant that it would be much easier for both the user and the program to just get the number ***n*** from the user and make a list from 1 to ***n*** and call the first rule. This also reduced the errors that a user can make and saved time. So now instead of the program taking in two lists it took one list of students and one integer which defined the number of seats in the class room.

C. *List of students*

The next task was to simplify the process of entering students. While it was great to have a list of students and get an output with an assigned seat for each student it still was not very useful because entering 40 student names each time was not short task. To simplify this task a built-in gprolog rule was used called 'length' which has two arguments. First argument is a list and the second argument is the number of elements in the list. This meant that if you call the rule with a list then it returns the number of elements in the list but if you call the rule with number of elements in the list then it returns a list of ***n*** variables where ***n*** is the number that it is called with. This simplified the task of entering students because with this function an integer can be used as an argument instead of a list.

With the addition of this function, the original program can be used with 2 integers which made it much faster.

D. *Meaningful output*

This was by far the most difficult task to accomplish because it did not have any strict requirements which meant you could get creative with it. Ideally, the program needed to have a:

- Visually pleasing and simple output where results could be instantly understood and used.
- Does not require any additional software, download or reliance on a server so it could be used without internet connectivity.
- The visualization part should be optional, if the user wants to run the program without it then he/she should have the option to do so.

The ideal solution would have been to display a chart within command line but it would only work in small room layouts and it would not display the same results on different screen sizes. Lastly, it would also be very hard to code in a different layout for different rooms. Running prolog in a Java or C program and then visualizing results in that program could have also been a solution but it required compiling for different machines and in some cases required additional software.

In any case the next task was to output the results in a format that could be read by other applications. A new rule was added to the program that output the solution into a file.

E. *Visualizing results*

For visualizing the result an HTML based web page was chosen as the best way to do it specifically because it runs on most computers and more computers can run a web browser than GProlog. Since it was HTML and not PHP it did not require any additional software to be installed on the machine or require internet but the downside was that the things it could achieve were limited.

The final webpage was made of HTML, CSS and some Javascript that just loaded the output file of the GProlog program into the webpage and another function that cleared all the results on the page. The way HTML elements were designed also made it very simple to add new seats or gaps in the interface. However, it is more a proof of concept on how other applications can be connected with prolog by simply taking the output file and using it in your program.

F. *Extending the functionality*

By this point the program satisfied all the initial requirements that were set when designing this program. The next step was to add more features, different ways of filling in seats. Up until now since there was only one way of filling seats and only one chain of rules that were called in a certain order, some of the variables were hard coded. These included the output file name and total number of seats, which meant if a new rule was added to the program it could potentially have different number of seats or different output file. To resolve this issue any variable that would be used in more than one rule was moved out of that one rule and made into a separate rule or fact that could be easily changed. These variables were:

- Total number of seats.
- Output file name.

This allowed new rules to use the same variables as all the previous rules so they would not give mixed results in different files.

G. *Reverse order seating*

As an example a new rule was added that started assigning seats from the end of the list instead of the begining. This serves as a good starting point for someone looking to add more features to the program, for example assigning seats in a random order.

IV. Challenges

A. *Simplicity*

The biggest challenge faced was making the program as easy to use as possible. The task that this program accomplishes can also be accomplished with a pencil and a piece of paper so the question was how much time it saves you compared to a pencil. Because of this, the goal of the program was not simply come up with a solution to the problem but to come up with a solution quickly.

B. *Function overloading*

GProlog is very good at coming up with a solution with only a few lines of code if you know how to program but it is not easy to understand for someone who has never used a similar language before. Simple features that are supported in other languages that make life easier are not a part of prolog. One such feature is function overloading. This allows the user the use to use the same command with different number and types of arguments and lets the user use the program without learning an overwhelimg number of function names. To get similar functionality in this program the rules were written in a way so that the names would be intuitive and easy to remember. For example normal(X, Y) outputs result to the screen while adding a w before the rule wnormal(X) outputs the result to the file. The downside is that the user still has to remember the number of arguments a rule takes. This is also the part where tab autocomplete rule comes in handy.

C. *Drawing a chart*

The second major limitation of GProlog is that unlike other languages it does not come with libraries that you can use to extend the functionality of your program without writing code from scratch. Thus, the drawing the chart had to be done in ASCII drawings. This meant it would be very time consuming to rewrite the code everytime the layout of classroom changes. The alternative was to just output to a file and use that to extend the program. This ended up working out as a better solution since this allowed the program to be extended even further and maybe even be used in a practical situation if used as a web application.

V. Result Analysis

The final result is output in a form of a simple list in which each number is an assigned seat. The output can also be recieved in a format which assigns each seat to a student name but that requires entering a list of students which is once again unefficient. The program can produce invalid results in two cases, (a) if the program runs out of seats, in which case it assigns at many seats as possible before giving an error message, or (b) if the program runs out of memory. This happens when the random seat assignment rule is used which is very inefficient but is one of the few ways to generate a truly random list. It is also possible to pick a random permutation of the list of seats but that means generating a list of all possible permutations then picking one at random, without a criteria which is also a difficult task and can crash the program. For now this feature is disabled until a more efficient way can be found to shuffle a list in GProlog.

VI. Conclusion

Overall this program highlights both the strengths and weaknesses of GProlog. The strength is that it can solve relatively complex problems in fewer lines of code compared to other languages and the weakness is lack of GUI libraries. This program can also serve as a starting point for someone looking to solve a simple problem in GProlog and generating a meaningful result because as shown in other programs, playing chess is not difficult in GProlog but entering the state in a command and displaying results is.

Two changes which keep this program a step away from being very practical are (a) outputting the results in a popular format i.e. a format that has libraries in other languages that can parse it easily like JSON format and, (b) outputting metadata as well as the final solution, this includes outputting total number of seats in the class, total number of students assigned seats, any error messages if they exist and all the gaps between seats. This will allow any other application to fully communicate with the result without having to manually hardcode class room layouts.