Detail On Regulations.pl

The general purpose of the Prolog program regulation.pl is to implement methods for checking if various requirements are satisfied as specified by kb_degrees.pl. First, check_transcripts/0 takes transcripts as described in kb_transcripts.pl and then creates a list that contains the arguments in each fact student/5. We then call print_results/1 which merely calls various rules that act on our list of students and uses format/2 to print the requisite output. The various other rules in regulations.pl are:

check_cp/3 : For each student go through a list of results and check if each entry has a passing mark if so increment a counter by the amount of points it is worth and check the next element, if not check the next element. Do this until all elements have been checked then unify the counter with the final result.

check_cp2/3 : Same as check_cp/3 except we check if the name of the unit we're currently looking contains '2' in either the fourth or fifth position then perform the same operations as check_cp3.

check_cp3/3 : Same as check_cp2/3 except we check if the character value is 3 not 2.

check_found/4 : Check how many foundation units are part of a given student's results, then for each one add the credit point value it has to a running total, and unify that running value with a total value once we have an empty list of results

check_major/5 : Calculate the amount of credit points that a student has for a given sublist of major units. If given a counter that is positive, this method is only run until that counter hits 0, or we've checked all units.

check_major_unit/4 : Breaks up the entire list of major units in a predefined major as satisfied in kb_degrees.pl. If we get a respective atom such as two_of(SubList) in our list we pass the arguments of the atom with a respective counter value (2 for this example). Pass these broken up components to check_major/5, and once we have an empty list of major units we unify our results.

check_pace/4 : Check if the pace_unit/2 associated with a given MajorID has been completed in a student's list of results, if true return "yes" and the amount of credit points that pace unit is worth.

complete_major/5 : We use findall/3 to collect all major_unit/2 based on the head of a list that contains the arguments for a qualifying_major/4. Then we check how many credit points are satisfied towards the current major. Then we check if we have the corresponding pace_unit/2 satisfied too. If we satisfy both, then return "yes" and our completed major name, which is the current major name. If not check the next qualifying_major/4 we have in our list.

check_people/5 : Given a list of people_unit/2 check that one of them appears in the student's results (with a passing mark) and also is not part of the student's qualifying major.

check_planet/5 : Given the faculty of a student's completed people unit check that one planet_unit/2 in a list of them is contained in a student's results (with a passing mark) and the faculty of this planet unit is different from a completed people unit (if any) and that it is not part of the student's qualifying major.

greater_equal/3 : Convert a binary operator that returns a boolean result into one that can be interpreted as a string.

Extension To Regulations.pl

The extended version of regulations.pl is classified under check_transcripts_ex/0 and print_results_ex/1. The extension I decided to implement was that if a requirement was not fulfilled it would print out a line under that requirement that would state what was needed to fulfil it (so if every fact in kb_transcript.pl satisfied every requirement in kb_degrees.pl the output of check_transcripts/0 would coincide with check_transcripts_ex/0).

This allows someone to run their own transcript through this program to get a simplified version of how many credit points they need to complete their degree. What units they need to finish a major for their degree, and finally what people and planet units they can complete. Such that if they complete the extra information listed they can finish their degree.

While implementing these changes are simple for all rules in the vein of check_cp and check_found, it is a bit harder for the rest of the qualifications. To simplify this when checking what a student needs to qualify for a major, we only consider the first qualifying_major/2 fact in the knowledge base, and return what units the student needs to qualify for this. And if any of the units needed for the major are atoms such as two_of(List), then if the student has completed one unit in that list then return an atom one_of(List \ CompletedUnits). So the student can know exactly what units they have not passed that are needed to qualify for the major. During this we also collect what PACE unit needs to be fulfilled for the major, and if the student has not completed it, display that in a separate line.

Then given this respective major, what planet and people units can be completed to satisfy this; again we only print the first valid (people, planet) combination that can be found. This reduces the amount of information that is printed and the processing time required. If we did not do this restriction then the amount of valid options would be quite large as the knowledge system increased.

A more detailed version of this system would check if the student has completed any PACE units, then print the major the student can qualify for that is linked to that respective PACE unit. If none, then find the most credit points the student satisfies towards one of the majors, or work around what people and planet units are completed by the student to determine the major they would most likely attempt to finish.

However, trying to configure for all of these cases would just make our program more complex. Since the initial program was built with a small set of features in mind, adding more and more features, while giving you more detailed output does make the program more bloated. Currently there are some rules that just do the opposite of the unextended version. If the program was created with these features in mind from the start they could be incorporated into the standard rules that are already there, yes it does make those blocks of code larger in some respects, but it is more contained. The fact that I'm attempting to keep the extensions separate from the original implementation just means there is repeated code, which just makes it harder to read for anyone else.

So as it stands, implementing extensions that are more constrained, allows easier maintenance of the code, while still giving greater depth of output.