“Gheorghe Asachi” Technical University of Iași
Faculty of Automatic Control and Systems Engineering
Master programme: Systems and Control

Knowledge Representation and Reasoning

~ Project ~

Cohal Alexandru

May 2016

Contents

1. Problem statement

Develop a project using the CLIPS programming environment to manage the information on all students in one year of study. The project must carry out the following actions:

- Scheduling examination in the exam session, making use of students' preferences (provided for each class), professors' preferences and establishing of convenient intervals between exams.
- Computing students' final grades for a discipline, considering the exam grades, laboratory grades, project grades according to the user given influence of each grade in the final grade formula; students will not be able to pass the exam in they have more than 2 absences in laboratory or project activities, and in the case of 1 or 2 absences, the respective grade will be lowered by 0.5, respectively 1 point.
- Generation of statistics based on the computed grades (students graded over 9, over 8 or who failed certain exam).
- Allocation of scholarship based on the average of students' final grades, using the available funds according to the following distribution: 30% for meritorious scholarships, 50% for integral scholarships and 20% for the partial scholarships. For each type of scholarship, a given maximum number of positions is provided. For partial scholarships, students that failed in an exam can also be considered, but they will be assigned for such scholarships only after the students that succeeded in all exams.
- Students' allocation to diploma guiding professors, considering their average grades, the available positions for each professor, and students' preferences.

The program should be optimized for a fast execution time and a user interface allowing the input of needed data will be provided.

2. Ontology

Based on the problem statement given, an Ontology which defines the types, the properties and the interrelationships of the entities involved in the structure of the information regarding the students of a university from a year of study (which represents the universe of discourse) was developed. The schematics approach of this Ontology is showed in *Fig.1.*, *Fig.2.*, *Fig.3.*, *Fig.4.*, *Fig.5.*, and *Fig.6.* (with blue colour are symbolized the classes, with green the properties and with orange the types).

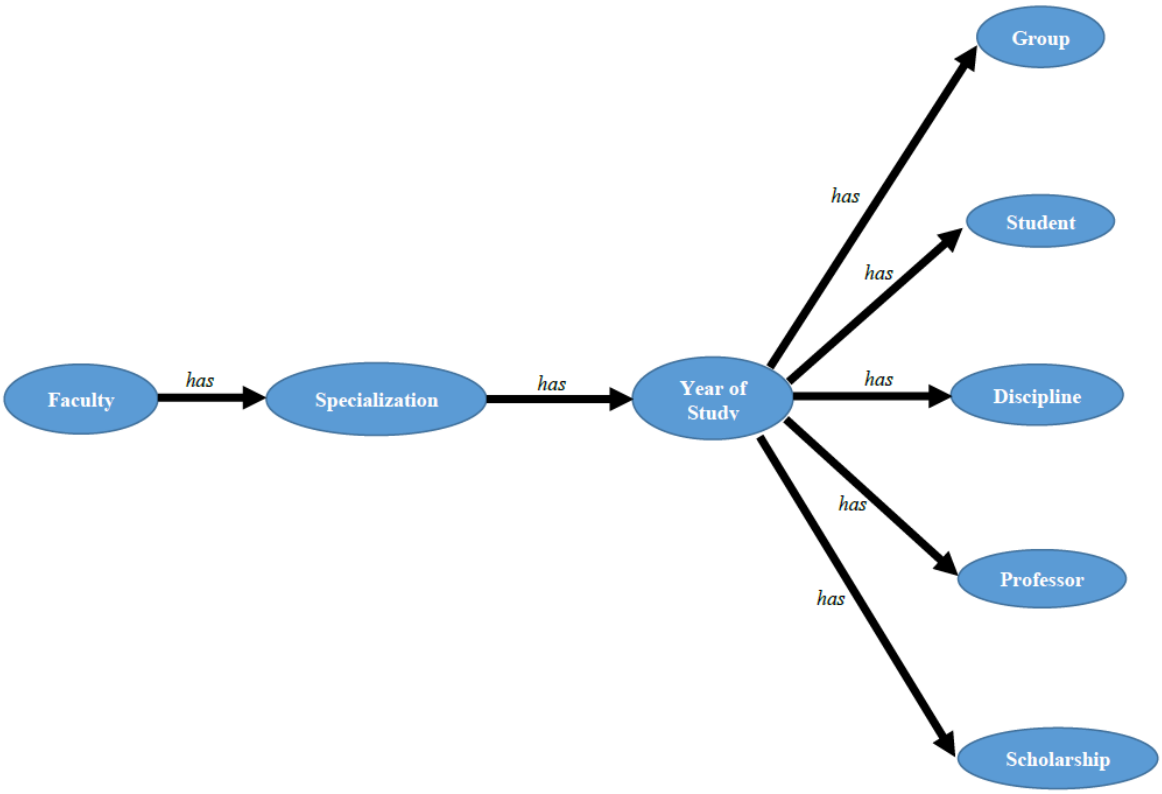

Fig.1. Overview of Ontology classes

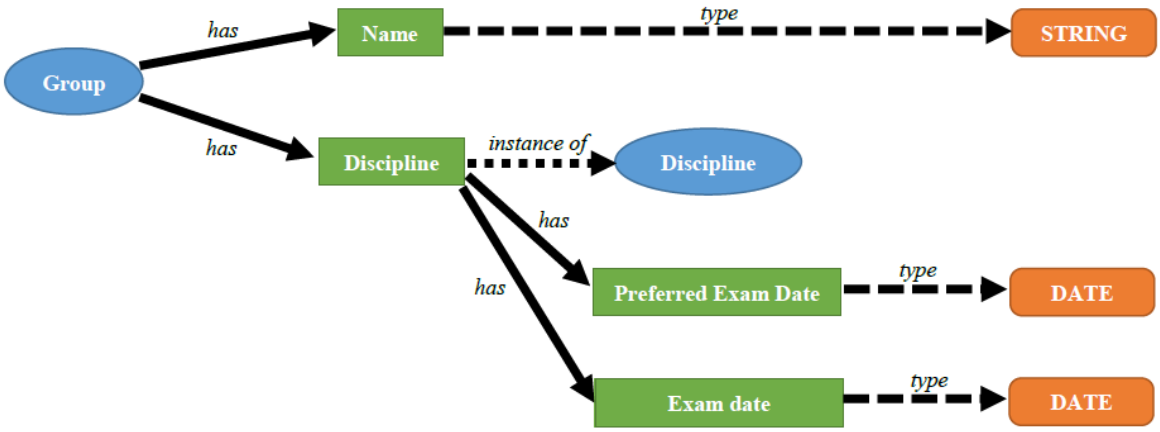

Fig.2. The structure of the 'Group' class

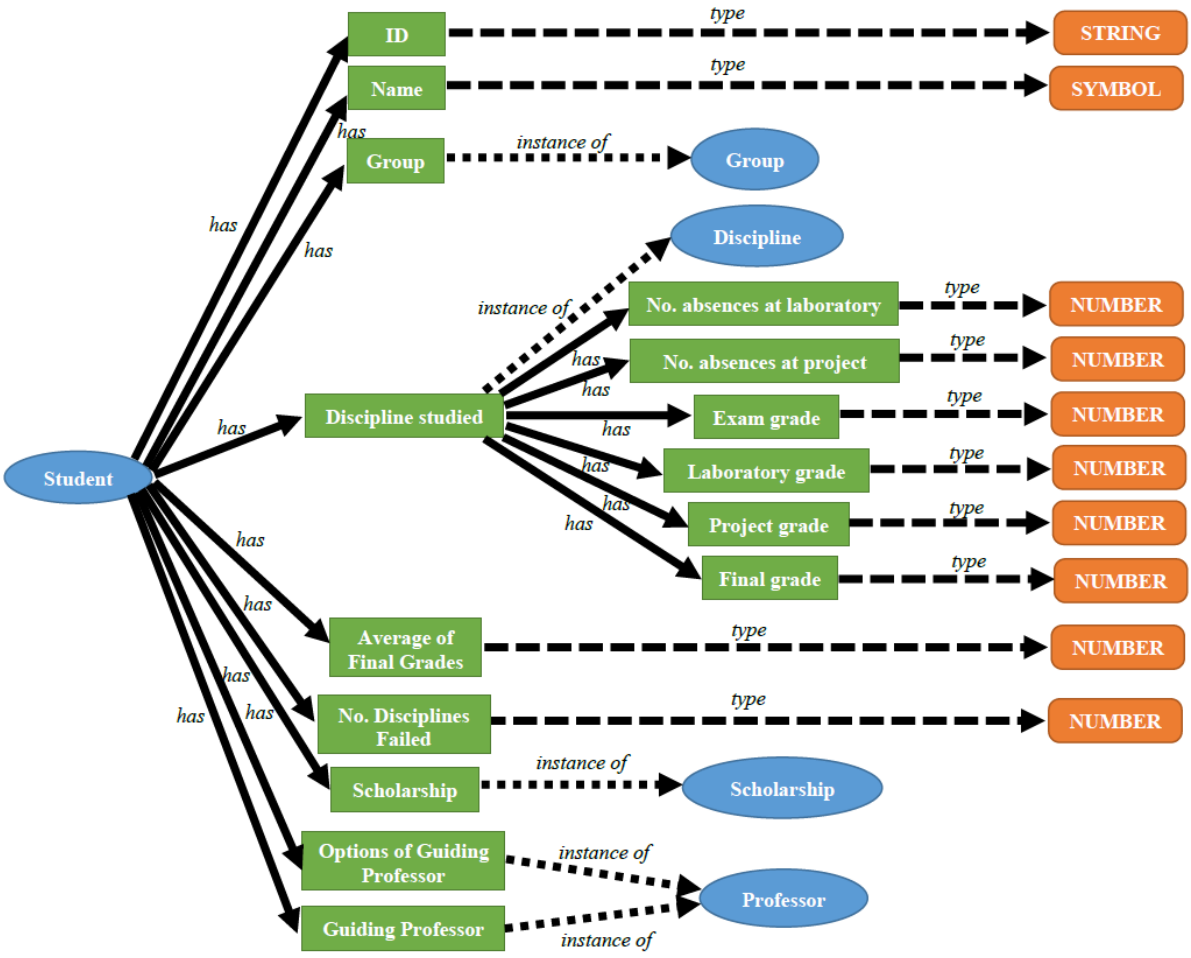

Fig.3. The structure of the 'Student class

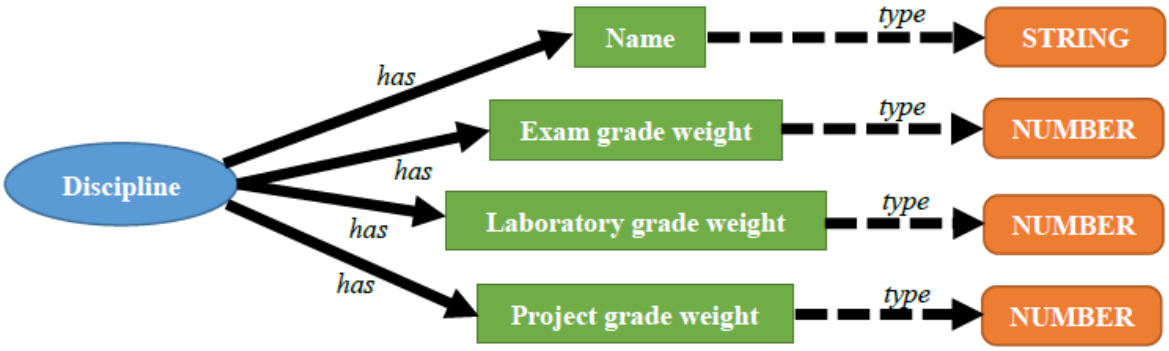

Fig.4. The structure of the 'Discipline' class

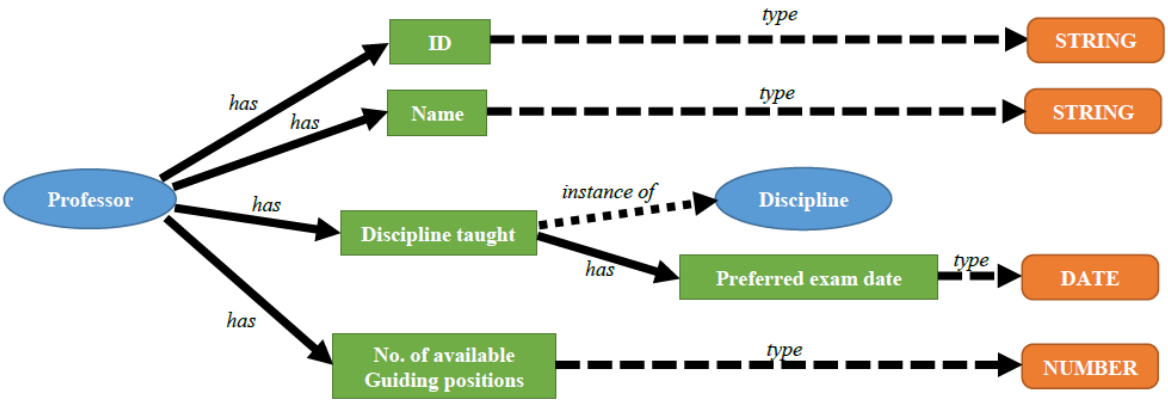

Fig.5. The structure of the 'Professor' class

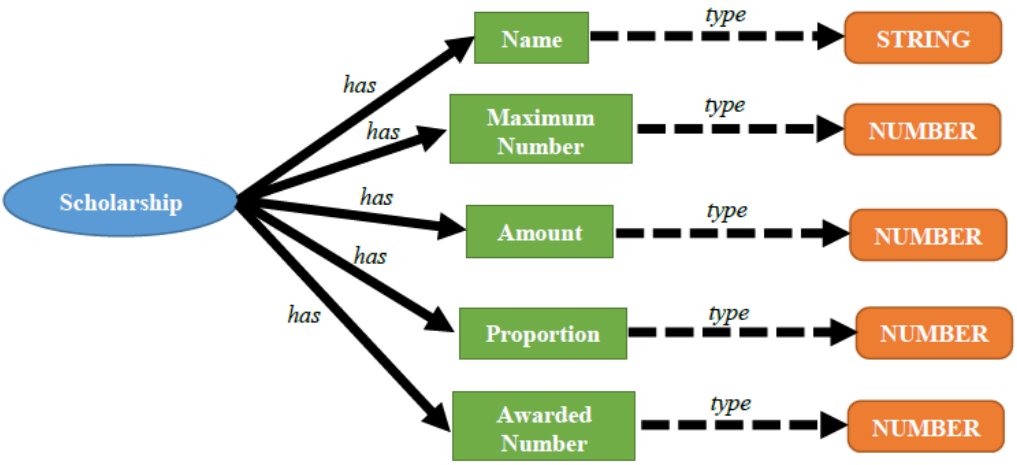

Fig.6. The structure of the 'Scholarship' class

Besides the fields specified in the previous structures, the implementation of the problem in CLIPS programming environment also required some auxiliary fields in order to obtain an easier solution:

- Class '`Student`': Index of the Guiding Professor Choice (stores the index of the chosen guiding professor according to the availability from the preferences list);
- Class '`Professor`': Number of Occupied Guiding Positions (used for checking if the selected professor has free guiding positions left), Occupied Preferred Exam Dates (used in order to not set in the same day more than one exam).

In principle, this Ontology has a fixed structure and will not need any modifications in the future. The only case when modifications should be made is when more than an Exam, a Laboratory and a Project will define a Discipline. In that case, the Ontology and the program sequence responsible for the computation of the Final Grades should be modified a little.

3. The Working Memory

Based on the Ontology presented before, structured facts were designed in order to obtain a simpler implementation and a better organization of the information used in this problem.

3.1. Patterns for Facts

The pattern of the structured fact which stores the information regarding a Group of students is:

```
(Group
    (Name <groupName>)
    (Disciplines <<<disciplinesName>>>)
    (ExamDatesOptions <<<examDatesOptions>>>)
    (ExamDates <<<examDates>>>)
)
```

- `Name`: a symbol representing the name of the Group
- `Disciplines`: a list of symbols representing Discipline's names (abbreviations) studied by the Group's Students
- `ExamDatesOptions`: a list of integers representing options for the Exam Dates. Each exam has exactly two options. The options are in the same order as the Disciplines from their list. The Dates are integers representing the relative distance in days from the beginning of the exam period.
- `ExamDates`: a list of integers representing the Exam Date for each Discipline (initialized with a number of 0 values equal with the number of Disciplines studied by the Group's Students). In some cases, a perfect match between one of the two choices expressed by a Group for a specific Discipline and the availability of that Discipline's Professor could not be made. Thus, the closest available Date of the Professor to the first option expressed by the Group was chosen.

The pattern of the structured fact which stores the information regarding a Student is:

```
(Student
    (ID <studID>)
    (Name <studFirstName> <<studMiddleName>> <studLastName>)
    (Group <groupName>)
    (DisciplineNames <<<disciplinesNames>>>)
    (NoAbsLab <<<noAbsLab>>>)
    (NoAbsProj <<<noAbsProj>>>)
    (ExamGrades <<<examGrades>>>)
    (LabGrades <<<labGrades>>>)
    (ProjGrades <<<projGrades>>>)
    (FinalGrades <<<finalGrades>>>)
    (NoDisciplinesFailed <noDisciplinesFailed>)
    (AverageFinalGrade <averageFinalGrade>)
    (Scholarship <scholarshipName>)
    (OptionsGuidingProf <<<guidingProfIDs>>>)
    (GuidingProf <guidingProfID>)
    (IndexGuidingProfChoice <guidingProfIndexFromOptionsList>)
)
```

- `ID`: a symbol representing the Student's ID number (generated by the person which introduces the Students' information in the Working Memory. The format is: Matriculation Number followed by the capital letter `S` without any separators (e.g. `15S`))
- `Name`: a sequence of symbols representing the name of the student (a blank space separator has to be used between the elements of the name)
- `Group`: a symbol representing the name of the group in which the Student enrolled
- `DisciplineNames`: a list of symbols representing Discipline's names (abbreviations) studied by the Student (in general, it is the same list as the one from the Group class but no restrictions are made in this direction (a Student can choose more optional Disciplines))
- `NoAbsLab`: a list containing integer values which represents the number of absences from Laboratory activity for each Discipline (the order is the same as the order from the Discipline's names list)
- `NoAbsProj`: a list containing integer values which represents the number of absences from Project activity for each Discipline (the order is the same as the order from the Discipline's names list) (for those Disciplines which does not have a Project activity, value `-1` is written in the corresponding field)
- `ExamGrades`: a list containing float values which represents the Exam grades for each Discipline (the order is the same as the order from the Discipline's names list)
- `LabGrades`: a list containing float values which represents the Laboratory grades for each Discipline (the order is the same as the order from the Discipline's names list)
- `ProjGrades`: a list containing float values which represents the Project activity grades for each Discipline (the order is the same as the order from the Discipline's names list) (for those Disciplines which does not have a Project activity, value `-1` is written in the corresponding field)
- `FinalGrades`: a list containing float values which represents the Final Grade for each Discipline (the order is the same as the order from the Discipline's names list). For those Disciplines which the Final Grade was not computed yet, value `-1` is written in the corresponding field (the default value). If a Disicipline is Failed, then value `0` is written in the corresponding field.
- `NoDisciplinesFailed`: the number of Disciplines failed (according to the number of absences from the Laboratory and the Project activity) (integer value)
- `AverageFinalGrades`: a float value representing the Average Final Grade
- `Scholarship`: a symbol representing the name of the Scholarship granted (`Meritorious`, `Integral` and `Partial`). The initial value for all Students is the symbol `none` (which is also kept in the case when the considered Student has no Scholarship)
- `OptionsGuidingProf`: a list of symbols representing the ID's of the Professors considered as choices for guiding the diploma project
- `GuidingProf`: a symbol representing the ID of the Professor chosen as guiding professor according to its availability and to the Student's academic results. The symbol `N/A` is stored in this field if no choices were available.
- `IndexGuidingProfChoice`: an integer representing the index of the chosen Guiding Professor. The default value is `1`.

The pattern of the structured fact which stores the information regarding a Professor is:

```
(Professor
    (ID <profID>)
    (Name <profFirstName> <<profMiddleName>> <profLastName>)
    (Disciplines <<<disciplinesNames>>>)
    (PrefferedExamDates <<<examDates>>>)
    (OccupiedPrefferedExamDates <<<occupiedPrefExamDates>>>)
    (NoMaxGuidingPositions <noMaxGuidingPositions>)
    (NoOccupiedGuidingPositions <noOccupiedGuidingPositions>)
)
```

- `ID`: a symbol representing the Professor's ID number (generated by the person which introduces the Professors' information in the Working Memory. The format is: Matriculation Number followed by the capital letter `P` without any separators (e.g. `22P`))
- `Name`: a sequence of symbols representing the name of the student (a blank space separator has to be used between the elements of the name)
- `Disciplines`: a list of symbols representing Discipline's names (abbreviations) taught by the Professor
- `PrefferedExamDates`: a list of integers representing the days when the Professor is available to conduct an exam at any Discipline (only one exam on each day)
- `OccupiedPrefferedExamDates`: a list of symbols representing the days when the Professor has to conduct an exam (in this case, in the corresponding position with the day from the `PrefferedExamDates` list is written the name of the Discipline) or not (symbol `no`).
- `NoMaxGuidingPositions`: an integer representing the maximum numbers of students which can be guided
- `NoOccupiedGuidingPositions`: an integer representing the numbers of guided students (default value is `0`).

The pattern of the structured fact which stores the information regarding a Discipline is:

```
(Discipline
    (Name <disciplineName>)
    (ExamGradeWeight <examGradeWeight>)
    (LabGradeWeight <labGradeWeight>)
    (ProjGradeWeight <projGradeWeight>)
)
```

- `Name`: a symbol representing the name of the Disicipline (abbreviation) (e.g. `DSP`)
- `ExamGradeWeight`: an integer representing the weight (percentage) of the Exam Grade in the Final Grade
- `LabGradeWeight`: an integer representing the weight (percentage) of the Exam Grade in the Final Grade
- `ProjGradeWeight`: an integer representing the weight (percentage) of the Exam Grade in the Final Grade. It is equal with `0` if the Discipline has no Project Activity.

The pattern of the structured fact which stores the information regarding a Scholarship is:

```
(Scholarship
    (Name <scholarshipName>)
    (MaxNumber <maxNumber>)
    (AwardedNumber <awardedNumber>)
    (Ratio <ratio>)
    (Amount <amount>)
)
```

- `Name`: a symbol representing the name of the Scholarship (`Meritorious`, `Integral` or `Partial`)
- `MaxNumber`: an integer representing the maximum number of Scholarship of that type which can be awarded
- `AwardedNumber`: an integer representing the number of awarded Scholarship of that type (the default value is `0`)
- `Ratio`: an integer representing the percentage from the total sum allocated for that type of Scholarship
- `Amount`: a float value representing the value of the scholarship according to the awarded number of scholarships of that type (the default value is `-1` symbolizing that the amount was not computed yet).

The budget (expressed in RON as a float value) allocated to all the scholarships is stored in a fact having the following pattern:

```
(ScolarshipsTotalSum <sum>)
```

3.2. The CLIPS implementation of the facts

The CLIPS structured facts which implements the previous patterns are listed below:

```
(deftemplate Group
    (slot Name (type SYMBOL))
    (multislot Disciplines)
    (multislot ExamDatesOptions)
    (multislot ExamDates)
)
```

```
(deftemplate Student
    (slot ID (type SYMBOL))
    (multislot Name (type SYMBOL))
    (slot Group (type SYMBOL))
    (multislot DisciplineNames (type SYMBOL))
    (multislot NoAbsLab (type INTEGER))
    (multislot NoAbsProj (type INTEGER))
    (multislot ExamGrades (type FLOAT))
    (multislot LabGrades (type FLOAT))
    (multislot ProjGrades (type FLOAT))
    (multislot FinalGrades (type FLOAT))
    (slot NoDisciplinesFailed (type INTEGER) (default 0))
    (slot AverageFinalGrades (type NUMBER) (default 0.0))
    (slot Scholarship (type SYMBOL) (default none))
    (multislot OptionsGuidingProf (type SYMBOL))
    (slot GuidingProf (type SYMBOL) (default none))
    (slot IndexGuidingProfChoice (type INTEGER) (default 1))
)
```

```
(deftemplate Professor
    (slot ID (type SYMBOL))
    (multislot Name (type SYMBOL))
    (multislot Disciplines (type SYMBOL))
    (multislot PrefferedExamDates (type INTEGER))
    (multislot OccupiedPrefferedExamDates (type SYMBOL))
    (slot NoMaxGuidingPositions (type INTEGER))
    (slot NoOccupiedGuidingPositions (type INTEGER) (default 0))
)
```

```
(deftemplate Discipline
    (slot Name (type SYMBOL))
    (slot ExamGradeWeight (type INTEGER))
    (slot LabGradeWeight (type INTEGER))
    (slot ProjGradeWeight (type INTEGER))
)
```

```
(deftemplate Scholarship
    (slot Name (type SYMBOL))
    (slot MaxNumber (type INTEGER))
    (slot Ratio (type INTEGER))
    (slot AwardedNumber (type INTEGER) (default 0))
    (slot Amount (type NUMBER) (default -1))
)
```

An example containing initial facts of each type stored in the WM is presented below:

```
    (Group
        (Name 1401B)
        (Disciplines DSP ROB MAN)
        (ExamDatesOptions 2 6 2 10 4 10)
        (ExamDates 0 0 0)
    )
```

```
(Student
    (ID 4S)
    (Name A B)
    (Group 1401B)
    (DisciplineNames DSP ROB MAN)
    (NoAbsLab 0 3 3)
    (NoAbsProj -1 1 1)
    (ExamGrades 6.0 9.0 6.0)
    (LabGrades 8.0 9.0 10.0)
    (ProjGrades -1.0 9.0 10.0)
    (FinalGrades -1.0 -1.0 -1.0)
    (NoDisciplinesFailed 0)
    (AverageFinalGrades 0.0)
    (OptionsGuidingProf 1P 2P 3P)
)
```

```
(Professor
    (ID 2P)
    (Name R N)
    (NoMaxGuidingPositions 1)
    (PrefferedExamDates 2 5 9)
    (OccupiedPrefferedExamDates no no no)
    (Disciplines ROB)
)
```

```
(Discipline
    (Name DSP)
    (ExamGradeWeight 60)
    (LabGradeWeight 40)
    (ProjGradeWeight 0)
)
```

```
(Scholarship
    (Name Meritorious)
    (MaxNumber 5)
    (Ratio 50)
)
```

All the facts like the ones presented above, are grouped in a `deffacts` structure, together with the fact which specifies the budget allocated for scholarships and with a fact in which will be stored the ordered list containing Students' IDs according to the value of the Average Final Grades:

```
(OrderedStudentsList)
```

4. The Rule base

For each main section of the program (Final Grade computing, Scholarship allocation, Guiding Professor allocation, Exam Dates scheduling, Generation of Statistics) multiple rules were designed.

4.1. Computing the Final Grades

Three rules were used to accomplish this task:

- `ComputeAverageGradeFail` – Computes the Final Grade in the case when the selected Student failed the selected Discipline
- `ComputeAverageGradeWithoutProject` – Computes the Final Grade in the case when the selected Student passed the Discipline which does not have Project Activity
- `ComputeAverageGradeWithProject` – Computes the Final Grade in the case when the selected Student passed the Discipline which has Project Activity.

The general structure of these rules follows the next steps:

- Select a Student having the Final Grade for the first Discipline in its list not computed yet
- Select the required elements for computing the Final Grade (Exam Grade, Laboratory Grade, Number of Laboratory Absences, etc.)
- Compute the Final Grade and place it in its position
- Update the Average Final Grade
- Shift all the elements of the used fields (Exam Grade, Laboratory Grade, Number of Laboratory Absences, etc.) with one position circular to the left such that the Discipline having the Final Grade computed now to be the last one in the list (thus, all the Disciplines will be one by one in the first position which ensures the computation of the Final Grade).

A final rule, `FinishAverageGrade` retracts from the Working Memory the fact which triggers the computation of the Final Grades (`(Determine FinalGrades)`).

```
(defrule ComputeAverageGradeFail
; Compute the Final Grade if the discipline is Failed - too many absences

    (Determine FinalGrades)
    ?adrStudent <- (Student
                        (DisciplineNames ?DisciplineName
$?otherDisciplines)
                        (NoAbsLab ?NoAbsLab $?otherNoAbsLab)
                        (NoAbsProj ?NoAbsProj $?otherNoAbsProj)
                        (ExamGrades ?ExamGrade $?otherExamGrades)
                        (LabGrades ?LabGrade $?otherLabGrades)
                        (ProjGrades ?ProjGrade $?otherProjGrades)
                        (FinalGrades ?FinalGrade $?otherFinalGrades)
                        (NoDisciplinesFailed ?NoDisciplinesFailed)
                        (AverageFinalGrades ?AverageFinalGrades)
                    )
    (test (and (eq ?FinalGrade -1.0)
               (or (> ?NoAbsLab 2)
                   (> ?NoAbsProj 2)
               )
          )
    )
```

```
    =>
    (modify ?adrStudent (DisciplineNames $?otherDisciplines
?DisciplineName)
                        (NoAbsLab $?otherNoAbsLab ?NoAbsLab)
                        (NoAbsProj $?otherNoAbsProj ?NoAbsProj)
                        (ExamGrades $?otherExamGrades ?ExamGrade)
                        (LabGrades $?otherLabGrades ?LabGrade)
                        (ProjGrades $?otherProjGrades ?ProjGrade)
                        (FinalGrades $?otherFinalGrades 0.0)
                        (NoDisciplinesFailed =(+ ?NoDisciplinesFailed 1))
                        (AverageFinalGrades =(+ ?AverageFinalGrades 0))
    )
)
```

```
(defrule ComputeAverageGradeWithoutProject
; Compute the Final Grade if the discipline is passed and it doesn't have
project

    (Determine FinalGrades)
    ?adrStudent <- (Student
                        (DisciplineNames ?DisciplineName
$?otherDisciplines)
                        (NoAbsLab ?NoAbsLab $?otherNoAbsLab)
                        (NoAbsProj ?NoAbsProj $?otherNoAbsProj)
                        (ExamGrades ?ExamGrade $?otherExamGrades)
                        (LabGrades ?LabGrade $?otherLabGrades)
                        (ProjGrades ?ProjGrade $?otherProjGrades)
                        (FinalGrades ?FinalGrade $?otherFinalGrades)
                        (AverageFinalGrades ?AverageFinalGrades)
                    )
    (test (and (eq ?FinalGrade -1.0)
               (eq ?ProjGrade -1)
               (<= ?NoAbsLab 2)
               (<= ?NoAbsProj 2)
          )
    )
    (Discipline
        (Name ?DisciplineName)
        (ExamGradeWeight ?ExamGradeWeight)
        (LabGradeWeight ?LabGradeWeight)
        (ProjGradeWeight ?ProjGradeWeight)
    )
    =>
    (bind ?ComputedFinalGrade ( / (+ (* ?ExamGrade ?ExamGradeWeight) (*
(- ?LabGrade (* 0.5 ?NoAbsLab)) ?LabGradeWeight)) 100 ))
    (modify ?adrStudent (DisciplineNames $?otherDisciplines
?DisciplineName)
                        (NoAbsLab $?otherNoAbsLab ?NoAbsLab)
                        (NoAbsProj $?otherNoAbsProj ?NoAbsProj)
                        (ExamGrades $?otherExamGrades ?ExamGrade)
                        (LabGrades $?otherLabGrades ?LabGrade)
                        (ProjGrades $?otherProjGrades ?ProjGrade)
                        (FinalGrades $?otherFinalGrades
?ComputedFinalGrade)
                        (AverageFinalGrades =(+ ?AverageFinalGrades ( /
?ComputedFinalGrade (+ (length$ $?otherDisciplines) 1)) ))

    )
)
```

```
(defrule ComputeAverageGradeWithProject
; Compute the Final Grade if the discipline is passed and it has project

    (Determine FinalGrades)
    ?adrStudent <- (Student
                        (DisciplineNames ?DisciplineName
$?otherDisciplines)
                        (NoAbsLab ?NoAbsLab $?otherNoAbsLab)
                        (NoAbsProj ?NoAbsProj $?otherNoAbsProj)
                        (ExamGrades ?ExamGrade $?otherExamGrades)
                        (LabGrades ?LabGrade $?otherLabGrades)
                        (ProjGrades ?ProjGrade $?otherProjGrades)
                        (FinalGrades ?FinalGrade $?otherFinalGrades)
                        (AverageFinalGrades ?AverageFinalGrades)
                    )
    (test (and (eq ?FinalGrade -1.0)
               (neq ?ProjGrade -1)
               (<= ?NoAbsLab 2)
               (<= ?NoAbsProj 2)
          )
    )
    (Discipline
        (Name ?DisciplineName)
        (ExamGradeWeight ?ExamGradeWeight)
        (LabGradeWeight ?LabGradeWeight)
        (ProjGradeWeight ?ProjGradeWeight)
    )
    =>
    (bind ?ComputedFinalGrade ( / (+ (* ?ExamGrade ?ExamGradeWeight) (*
(- ?LabGrade (* 0.5 ?NoAbsLab)) ?LabGradeWeight) (* (- ?ProjGrade (* 0.5
?NoAbsProj)) ?ProjGradeWeight)) 100 ) )
    (modify ?adrStudent (DisciplineNames $?otherDisciplines
?DisciplineName)
                        (NoAbsLab $?otherNoAbsLab ?NoAbsLab)
                        (NoAbsProj $?otherNoAbsProj ?NoAbsProj)
                        (ExamGrades $?otherExamGrades ?ExamGrade)
                        (LabGrades $?otherLabGrades ?LabGrade)
                        (ProjGrades $?otherProjGrades ?ProjGrade)
                        (FinalGrades $?otherFinalGrades
?ComputedFinalGrade)
                        (AverageFinalGrades =(+ ?AverageFinalGrades ( /
?ComputedFinalGrade (+ (length$ $?otherDisciplines) 1) )))
    )
)
```

```
(defrule FinishAverageGrade
; Retract from the WM the fact which triggers the computation of the
Avearage Grades

    ?adr <- (Determine FinalGrades)
    (not (Student
              (FinalGrades -1.0 $?)
         )
    )
    =>
    (retract ?adr)
)
```

4.2. Ordering the Students according to the Average Final Grades

Two rules are used to accomplish this task:

- `CreateStudentsList` – adds one by one all the Students' IDs to a list (if it was not previously added)
- `OrderStudentsList` – selects two students from the ID list and swaps them if they are not in the desired order. Two sorting criteria are used: the first one is the number of failed Disciplines (ascending) and the second one is the Average Final Grades (descending).

The ordering of the students is required for the Scholarship and Guiding professor allocation because Students with better academic results have higher priority.

```
(defrule CreateStudentsList
; Create a List with all the IDs of the Students

    (ToDo CreateStudentsList)
    ?adStudentsList <- (OrderedStudentsList $?Students)
    (Student
        (ID ?studentID)
    )
    (not (OrderedStudentsList $? ?studentID $?))
    =>
    (retract ?adStudentsList)
    (assert (OrderedStudentsList $?Students ?studentID))
)
```

```
(defrule OrderStudentsList
; Order the List which contains all the IDs of the Students
; Is Activated only when fact (ToDo OrderStudents) is in the WM

    (ToDo OrderStudents)
    ?adStudentsList <- (OrderedStudentsList $?IDgroup1 ?ID1 $?IDgroup2
?ID2 $?IDgroup3)
    (Student
        (ID ?ID1)
        (NoDisciplinesFailed ?NoDisciplinesFailedID1)
        (AverageFinalGrades ?AverageFinalGradesID1)
    )
    (Student
        (ID ?ID2)
        (NoDisciplinesFailed ?NoDisciplinesFailedID2)
        (AverageFinalGrades ?AverageFinalGradesID2)
    )
    (test (or (and (eq ?NoDisciplinesFailedID1 ?NoDisciplinesFailedID2)
                   (< ?AverageFinalGradesID1 ?AverageFinalGradesID2)
              )
              (> ?NoDisciplinesFailedID1 ?NoDisciplinesFailedID2)
          )
    )
    =>
    (retract ?adStudentsList)
    (assert (OrderedStudentsList $?IDgroup1 ?ID2 $?IDgroup2 ?ID1
$?IDgroup3))
)
```

4.3. Scholarships allocation

Six rules were used to accomplish this task (two rules for each type of Scholarship):

- `AwardMeritoriousScholarshipFirstStudent` – awards Meritorious scholarship for the first student in the ordered list if all the conditions are fulfilled
- `AwardMeritoriousScholarshipSecondaryStudents` – awards Meritorious scholarship for a Student from a secondary position (not from the first position) in the ordered list if all the conditions are fulfilled
- `AwardIntegralScholarshipFirstStudent` – awards Integral scholarship for the first student in the ordered list if all the conditions are fulfilled
- `AwardIntegralScholarshipSecondaryStudents` – awards Integral scholarship for a Student from a secondary position (not from the first position) in the ordered list if all the conditions are fulfilled
- `AwardPartialScholarshipFirstStudent` – awards Partial scholarship for the first student in the ordered list if all the conditions are fulfilled
- `AwardPartialScholarshipSecondaryStudents` – awards Partial scholarship for a Student from a secondary position (not from the first position) in the ordered list if all the conditions are fulfilled.

The steps for awarding any type of Scholarship to the Student placed on the first position in the list are:

- Select the Student on the first position in the list
- Verify if it is eligible for this type of Scholarship (no failed Disciplines for the case of Meritorious and Integral Scholarship; no other Scholarship previously awarded)
- Verify if there are still scholarships of this type left
- Award Scholarship
- Update the Student's Scholarship field and the Scholarship's Awarded Number field.

The steps for awarding any type of Scholarship to a Student placed on a secondary position are:

- Select 2 consecutive Students from the list
- Verify if the first student has a scholarship and the second has not (this condition ensures the awarding according to the order criteria)
- Verify if the second Student is eligible for this type of Scholarship (no failed Disciplines for the case of Meritorious and Integral Scholarship)
- Verify if there are still scholarships of this type left
- Award Scholarship
- Update the Student's Scholarship field and the Scholarship's Awarded Number field.

```
; ------------------------------ Meritorious -----------------
(defrule AwardMeritoriousScholarshipFirstStudent
; Award Meritorious Scholarship for the first student in the list

    (Determine Scholarships)
    (OrderedStudentsList ?ID1 $?)
    ?StudentID1 <- (Student
                        (ID ?ID1)
                        (NoDisciplinesFailed ?NoDisciplinesFailedID1)
                        (Scholarship ?scholarshipTypeID1)
                    )
    (test (and (eq ?scholarshipTypeID1 none)
               (eq ?NoDisciplinesFailedID1 0)
          )
    )
    ?adScholarship <- (Scholarship
                            (Name Meritorious)
                            (MaxNumber ?MaxNumber)
                            (AwardedNumber ?AwardedNumber)
                      )
    (test (< ?AwardedNumber ?MaxNumber))
    =>
    (modify ?adScholarship (AwardedNumber =(+ ?AwardedNumber 1)))
    (modify ?StudentID1 (Scholarship Meritorious))
)

(defrule AwardMeritoriousScholarshipSecondaryStudents
; Award Meritorious Scholarship for the following students after the
first one

    (Determine Scholarships)
    (OrderedStudentsList $? ?ID1 ?ID2 $?)
    ?StudentID1 <- (Student
                        (ID ?ID1)
                        (Scholarship ?scholarshipTypeID1)
                    )
    ?StudentID2 <- (Student
                        (ID ?ID2)
                        (NoDisciplinesFailed ?NoDisciplinesFailedID2)
                        (Scholarship ?scholarshipTypeID2)
                    )
    (test (and (neq ?scholarshipTypeID1 none)
               (eq ?scholarshipTypeID2 none)
               (eq ?NoDisciplinesFailedID2 0)
          )
    )
    ?adScholarship <- (Scholarship
                            (Name Meritorious)
                            (MaxNumber ?MaxNumber)
                            (AwardedNumber ?AwardedNumber)
                      )
    (test (< ?AwardedNumber ?MaxNumber))
    =>
    (modify ?adScholarship (AwardedNumber =(+ ?AwardedNumber 1)))
    (modify ?StudentID2 (Scholarship Meritorious))
)
```

```
; ------------------------------ Integral -----------------
(defrule AwardIntegralScholarshipFirstStudent
; Award Integral Scholarship for the first student in the list

    (Determine Scholarships)
    (OrderedStudentsList ?ID1 $?)
    ?StudentID1 <- (Student
                        (ID ?ID1)
                        (NoDisciplinesFailed ?NoDisciplinesFailedID1)
                        (Scholarship ?scholarshipTypeID1)
                    )
    (test (and (eq ?scholarshipTypeID1 none)
               (eq ?NoDisciplinesFailedID1 0)
          )
    )
    (Scholarship
        (Name Meritorious)
        (MaxNumber ?MaxNumberMerit)
        (AwardedNumber ?AwardedNumberMerit)
    )
    ?adScholarshipIntegral <- (Scholarship
                                    (Name Integral)
                                    (MaxNumber ?MaxNumberIntegral)
                                    (AwardedNumber
?AwardedNumberIntegral)
                                )
    (test (and (eq ?MaxNumberMerit 0)
               (< ?AwardedNumberIntegral ?MaxNumberIntegral)
          )
    )
    =>
    (modify ?adScholarshipIntegral (AwardedNumber =(+
?AwardedNumberIntegral 1)))
    (modify ?StudentID1 (Scholarship Integral))
)

(defrule AwardIntegralScholarshipSecondaryStudents
; Award Integral Scholarship for the following students after the first
one

    (Determine Scholarships)
    (OrderedStudentsList $? ?ID1 ?ID2 $?)
    ?StudentID1 <- (Student
                        (ID ?ID1)
                        (Scholarship ?scholarshipTypeID1)
                    )
    ?StudentID2 <- (Student
                        (ID ?ID2)
                        (NoDisciplinesFailed ?NoDisciplinesFailedID2)
                        (Scholarship ?scholarshipTypeID2)
                    )
    (test (and (neq ?scholarshipTypeID1 none)
               (eq ?scholarshipTypeID2 none)
               (eq ?NoDisciplinesFailedID2 0)
          )
    )
    (Scholarship
        (Name Meritorious)
        (MaxNumber ?MaxNumberMerit)
        (AwardedNumber ?AwardedNumberMerit)
    )
```

```
      ?adScholarshipIntegral <- (Scholarship
                                          (Name Integral)
                                          (MaxNumber ?MaxNumberIntegral)
                                          (AwardedNumber
      ?AwardedNumberIntegral)
                                    )
          (test (and (eq ?MaxNumberMerit ?AwardedNumberMerit)
                     (< ?AwardedNumberIntegral ?MaxNumberIntegral)
                )
          )
          =>
          (modify ?adScholarshipIntegral (AwardedNumber =(+
      ?AwardedNumberIntegral 1)))
          (modify ?StudentID2 (Scholarship Integral))
)
```

```
; ------------------------------ Partial ------------------
(defrule AwardPartialScholarshipFirstStudent
; Award Partial Scholarship for the first student in the list

    (Determine Scholarships)
    (OrderedStudentsList ?ID1 $?)
    ?StudentID1 <- (Student
                        (ID ?ID1)
                        (Scholarship ?scholarshipTypeID1)
                   )
    (test (eq ?scholarshipTypeID1 none))
    (Scholarship
        (Name Meritorious)
        (MaxNumber ?MaxNumberMerit)
        (AwardedNumber ?AwardedNumberMerit)
    )
    (Scholarship
        (Name Integral)
        (MaxNumber ?MaxNumberIntegral)
        (AwardedNumber ?AwardedNumberIntegral)
    )
    ?adScholarshipPartial <- (Scholarship
                                    (Name Partial)
                                    (MaxNumber ?MaxNumberPartial)
                                    (AwardedNumber ?AwardedNumberPartial)
                                )
    (test (and (eq ?MaxNumberMerit 0)
               (eq ?MaxNumberIntegral 0)
               (< ?AwardedNumberPartial ?MaxNumberPartial)
          )
    )
    =>
    (modify ?adScholarshipPartial (AwardedNumber =(+
?AwardedNumberPartial 1)))
    (modify ?StudentID1 (Scholarship Partial))
)
```

```
(defrule AwardPartialScholarshipSecondaryStudents
; Award Partial Scholarship for the following students after the first
one

    (Determine Scholarships)
    (OrderedStudentsList $? ?ID1 ?ID2 $?)
    ?StudentID1 <- (Student
                        (ID ?ID1)
                        (Scholarship ?scholarshipTypeID1)
                    )
    ?StudentID2 <- (Student
                        (ID ?ID2)
                        (NoDisciplinesFailed ?NoDisciplinesFailedID2)
                        (Scholarship ?scholarshipTypeID2)
                    )
    (test (and (neq ?scholarshipTypeID1 none)
               (eq ?scholarshipTypeID2 none)
          )
    )
    (Scholarship
        (Name Meritorious)
        (MaxNumber ?MaxNumberMerit)
        (AwardedNumber ?AwardedNumberMerit)
    )
    (Scholarship
        (Name Integral)
        (MaxNumber ?MaxNumberIntegral)
        (AwardedNumber ?AwardedNumberIntegral)
    )
    ?adScholarshipPartial <- (Scholarship
                                    (Name Partial)
                                    (MaxNumber ?MaxNumberPartial)
                                    (AwardedNumber ?AwardedNumberPartial)
                                )
    (test (and (or (and (eq ?MaxNumberMerit ?AwardedNumberMerit)
                        (eq ?MaxNumberIntegral ?AwardedNumberIntegral)
                   )
                   (> ?NoDisciplinesFailedID2 0)
               )
               (< ?AwardedNumberPartial ?MaxNumberPartial)
          )
    )
    =>
    (modify ?adScholarshipPartial (AwardedNumber =(+
?AwardedNumberPartial 1)))
    (modify ?StudentID2 (Scholarship Partial))
)
```

4.4. Computing the Amount for each Scholarship type

For each type of Scholarship, a particular rule which computes using a simple formula (based on the budget for that type of Scholarship and on the number of awarded Scholarship of that type) computes the sum received by each awarded Student.

```
; ------------------------------ Compute Values ------------------
(defrule ComputeValueMeritoriousScholarship
; Compute the value of the Meritorious Scholarship

    (Determine ValuesScholarships)
    (ScholarshipsTotalSum ?scholarshipsSum)
    ?adScholarship <- (Scholarship
                            (Name Meritorious)
                            (AwardedNumber ?AwardedNumber)
                            (Ratio ?ratio)
                            (Amount ?amount)
                      )
    (test (eq ?amount -1))
    =>
    (if (neq ?AwardedNumber 0)
        then (modify ?adScholarship (Amount =(/ (/ (* ?ratio
?scholarshipsSum) 100) ?AwardedNumber)))
        else (modify ?adScholarship (Amount 0.0))
    )
)

(defrule ComputeValueIntegralScholarship
; Compute the value of the Integral Scholarship

    (Determine ValuesScholarships)
    (ScholarshipsTotalSum ?scholarshipsSum)
    ?adScholarship <- (Scholarship
                            (Name Integral)
                            (AwardedNumber ?AwardedNumber)
                            (Ratio ?ratio)
                            (Amount ?amount)
                      )
    (test (eq ?amount -1))
    =>
    (if (neq ?AwardedNumber 0)
        then (modify ?adScholarship (Amount =(/ (/ (* ?ratio
?scholarshipsSum) 100) ?AwardedNumber)))
        else (modify ?adScholarship (Amount 0.0))
    )
)

(defrule ComputeValuePartialScholarship
; Compute the value of the Partial Scholarship

    (Determine ValuesScholarships)
    (ScholarshipsTotalSum ?scholarshipsSum)
    ?adScholarship <- (Scholarship
                            (Name Partial)
                            (AwardedNumber ?AwardedNumber)
                            (Ratio ?ratio)
                            (Amount ?amount)
                      )
    (test (eq ?amount -1))
    =>
    (if (neq ?AwardedNumber 0)
        then (modify ?adScholarship (Amount =(/ (/ (* ?ratio
?scholarshipsSum) 100) ?AwardedNumber)))
        else (modify ?adScholarship (Amount 0.0))
    )
)
```

4.5. Determine the Guiding Professor

Five rules are used to accomplish this task:

- `ChooseGuidingProfessorFirstStudentValidChoice` – allocate the first Student on the ordered list to the current option of Guiding Professor (which is a valid option)
- `ChooseGuidingProfessorFirstStudentWrongChoice` – move the first Student on the ordered list to the next option of Guiding Professor because the current one is not valid
- `ChooseGuidingProfessorecondaryStudentsValidChoice` – allocate the next secondary Students from the ordered list to the current option of Guiding Professor (which is a valid option)
- `ChooseGuidingProfessorSecondaryStudentsWrongChoice` – move the next secondary Students from the ordered list to the next option of Guiding Professor because the current one is not valid
- `ChooseGuidingProfessorFail` – mark the current Student as not having any Guiding Professor because not any option was a valid one.

The steps followed by these rules are similar with ones which allocate Scholarships. There is a separate selection behaviour for the first student in the list and for the secondary students. For the selected Student it is verified repeatedly (using the counter stored in the field IndexGuidingProfChoice) the options from its list until a valid one is found (a professor having free guiding positions). If a valid one is found then the WM is updated. Otherwise, the Student is marked as not having any valid option.

```
(defrule ChooseGuidingProfessorFirstStudentValidChoice
; Choose the Guiding Professor for the first student of the ordered list

    (Determine GuidingProf)
    (OrderedStudentsList ?studentID1 $?)
    ?adStudentID1 <- (Student
                        (ID ?studentID1)
                        (OptionsGuidingProf $?optionsGuidingProf)
                        (GuidingProf ?IDGuidingProf)
                        (IndexGuidingProfChoice ?indexGuidingProf)
                     )
    (test (and (eq ?IDGuidingProf none)
               (<= ?indexGuidingProf (length$ $?optionsGuidingProf))
          )
    )
    ?adGuidingProf <- (Professor
                        (ID =(nth$ ?indexGuidingProf
$?optionsGuidingProf))
                        (NoMaxGuidingPositions ?maxPositions)
                        (NoOccupiedGuidingPositions ?occupiedPositions)
                      )
    (test (< ?occupiedPositions ?maxPositions))
    =>
    (modify ?adStudentID1 (GuidingProf =(nth$ ?indexGuidingProf
$?optionsGuidingProf)))
    (modify ?adGuidingProf (NoOccupiedGuidingPositions =(+
?occupiedPositions 1)))
)
```

```
(defrule ChooseGuidingProfessorFirstStudentWrongChoice
; Move to the next option of Guiding Professor for the first student of
the ordered list

    (Determine GuidingProf)
    (OrderedStudentsList ?studentID1 $?)
    ?adStudentID1 <- (Student
                        (ID ?studentID1)
                        (OptionsGuidingProf $?optionsGuidingProf)
                        (GuidingProf ?IDGuidingProf)
                        (IndexGuidingProfChoice ?indexGuidingProf)
                     )
    (test (and (eq ?IDGuidingProf none)
               (<= ?indexGuidingProf (length$ $?optionsGuidingProf))
          )
    )
    ?adGuidingProf <- (Professor
                        (ID =(nth$ ?indexGuidingProf
$?optionsGuidingProf))
                        (NoMaxGuidingPositions ?maxPositions)
                        (NoOccupiedGuidingPositions ?occupiedPositions)
                      )
    (test (= ?occupiedPositions ?maxPositions))
    =>
    (modify ?adStudentID1 (IndexGuidingProfChoice =(+ ?indexGuidingProf
1)))
)
```

```
(defrule ChooseGuidingProfessorecondaryStudentsValidChoice
; Choose the Guiding Professors for the secondary students of the ordered
list

    (Determine GuidingProf)
    (OrderedStudentsList $? ?studentID1 ?studentID2 $?)
    ?adStudentID1 <- (Student
                        (ID ?studentID1)
                        (GuidingProf ?IDGuidingProfStudentID1)
                     )
    ?adStudentID2 <- (Student
                        (ID ?studentID2)
                        (OptionsGuidingProf
$?optionsGuidingProfStudentID2)
                        (GuidingProf ?IDGuidingProfStudentID2)
                        (IndexGuidingProfChoice
?indexGuidingProfStudentID2)
                     )
    (test (and (neq ?IDGuidingProfStudentID1 none)
               (eq ?IDGuidingProfStudentID2 none)
               (<= ?indexGuidingProfStudentID2 (length$
$?optionsGuidingProfStudentID2))
          )
    )
    ?adGuidingProf <- (Professor
                        (ID =(nth$ ?indexGuidingProfStudentID2
$?optionsGuidingProfStudentID2))
                        (NoMaxGuidingPositions ?maxPositions)
                        (NoOccupiedGuidingPositions ?occupiedPositions)
                      )
    (test (< ?occupiedPositions ?maxPositions))
    =>
    (modify ?adStudentID2 (GuidingProf =(nth$ ?indexGuidingProfStudentID2
$?optionsGuidingProfStudentID2)))
    (modify ?adGuidingProf (NoOccupiedGuidingPositions =(+
?occupiedPositions 1)))
)
```

```
(defrule ChooseGuidingProfessorSecondaryStudentsWrongChoice
; Move to the next option of Guiding Professor for the secondary students
of the ordered list

    (Determine GuidingProf)
    (OrderedStudentsList $? ?studentID1 ?studentID2 $?)
    ?adStudentID1 <- (Student
                        (ID ?studentID1)
                        (GuidingProf ?IDGuidingProfStudentID1)
                     )
    ?adStudentID2 <- (Student
                        (ID ?studentID2)
                        (OptionsGuidingProf
$?optionsGuidingProfStudentID2)
                        (GuidingProf ?IDGuidingProfStudentID2)
                        (IndexGuidingProfChoice
?indexGuidingProfStudentID2)
                     )
    (test (and (neq ?IDGuidingProfStudentID1 none)
               (eq ?IDGuidingProfStudentID2 none)
               (<= ?indexGuidingProfStudentID2 (length$
$?optionsGuidingProfStudentID2))
          )
    )
    ?adGuidingProf <- (Professor
                        (ID =(nth$ ?indexGuidingProfStudentID2
$?optionsGuidingProfStudentID2))
                        (NoMaxGuidingPositions ?maxPositions)
                        (NoOccupiedGuidingPositions ?occupiedPositions)
                      )
    (test (= ?occupiedPositions ?maxPositions))
    =>
    (modify ?adStudentID2 (IndexGuidingProfChoice =(+
?indexGuidingProfStudentID2 1)))
)
```

```
(defrule ChooseGuidingProfessorFail
; Not found a valid option for a Student for the Guiding Professor

    (Determine GuidingProf)
    (OrderedStudentsList $? ?studentID1 $?)
    ?adStudentID1 <- (Student
                        (ID ?studentID1)
                        (OptionsGuidingProf $?optionsGuidingProf)
                        (GuidingProf ?IDGuidingProf)
                        (IndexGuidingProfChoice ?indexGuidingProf)
                     )
    (test (and (eq ?IDGuidingProf none)
               (> ?indexGuidingProf (length$ $?optionsGuidingProf))
          )
    )
    =>
    (modify ?adStudentID1 (GuidingProf N/A))
)
```

4.6. Schedule the examinations

Three rules are used to accomplish this task:

- `ChooseExamDateExact_option1` – Schedule the Exam of the selected Discipline on the first option given by the Students (if it is a valid option)
- `ChooseExamDateExact_option2` – Schedule the Exam of the selected Discipline on the second option given by the Students (if it is a valid option and if the first option was not a valid one)
- `ChooseExamDateApprox_option1` – Schedule the Exam of the selected Discipline on the closest day to the first option given by the Students between the available options of the Professor (if the first and the second options were not valid).

The steps for each of these rules are similar to the ones of the rules which compute the Final Grades:

- Select a Group having the Exam for the first Discipline in its list not scheduled yet
- Verify if the first option is valid (it is also an option for the Professor and it is not occupied by another exam) (Rule ChooseExamDateExact_option1). If it is, update the Working Memory (by moving the exam which was just set to the end of the list)
- Verify if the second option is valid (it is also an option for the Professor and it is not occupied by another exam) (Rule ChooseExamDateExact_option2). If it is, update the Working memory
- Find an alternative option (the closes day between the available options of the professor to the first option given by the Students). Update the Working Memory.

```
(defrule ChooseExamDateExact_option1
; Choose the first Exam Date option if it is possible

    (declare (salience 10))

    (Determine ExamDates)
    ?adGroup <- (Group
                    (Name ?groupName)
                    (Disciplines ?disciplineName $?groupDisciplineName)
                    (ExamDatesOptions ?option1 ?option2 $?groupOptions)
                    (ExamDates 0 $?groupExamDates)
                )
    (not (Group
            (Name ?groupName)
            (ExamDates $? ?option1 $?)
         )
    )
    ?adProf <- (Professor
                    (Disciplines $? ?disciplineName $?)
                    (PrefferedExamDates $?profPrefferedDates1 ?option1 $?)
                    (OccupiedPrefferedExamDates $?profOccupiedExamDates)
                )
    (test (eq (nth$ (+ (length$ $?profPrefferedDates1) 1)
$?profOccupiedExamDates) no))
    =>
    (modify ?adGroup (Disciplines $?groupDisciplineName ?disciplineName)
                     (ExamDatesOptions $?groupOptions ?option1 ?option2)
                     (ExamDates $?groupExamDates ?option1)
    )
    (modify ?adProf (OccupiedPrefferedExamDates (replace$
$?profOccupiedExamDates (+ (length$ $?profPrefferedDates1) 1) (+ (length$
$?profPrefferedDates1) 1) ?disciplineName)))
)
```

```
(defrule ChooseExamDateExact_option2
; Choose the second Exam Date option if it is possible

    (declare (salience 10))

    (Determine ExamDates)
    ?adGroup <- (Group
                    (Name ?groupName)
                    (Disciplines ?disciplineName $?groupDisciplineName)
                    (ExamDatesOptions ?option1 ?option2 $?groupOptions)
                    (ExamDates 0 $?groupExamDates)
                )
    (not (Group
            (Name ?groupName)
            (ExamDates $? ?option2 $?)
        )
    )
    ?adProf <- (Professor
                    (Disciplines $? ?disciplineName $?)
                    (PrefferedExamDates $?profPrefferedDates1 ?option2
$?)
                    (OccupiedPrefferedExamDates $?profOccupiedExamDates)
                )
    (test (eq (nth$ (+ (length$ $?profPrefferedDates1) 1)
$?profOccupiedExamDates) no))
    =>
    (modify ?adGroup (Disciplines $?groupDisciplineName ?disciplineName)
                     (ExamDatesOptions $?groupOptions ?option1 ?option2)
                     (ExamDates $?groupExamDates ?option2)
    )
    (modify ?adProf (OccupiedPrefferedExamDates (replace$
$?profOccupiedExamDates (+ (length$ $?profPrefferedDates1) 1) (+ (length$
$?profPrefferedDates1) 1) ?disciplineName)))
)
```

```
(defrule ChooseExamDateApprox_option1
; Choose the closest Exam Date to the first option

    (Determine ExamDates)
    ?adGroup <- (Group
                    (Name ?groupName)
                    (Disciplines ?disciplineName $?groupDisciplineName)
                    (ExamDatesOptions ?option1 ?option2 $?groupOptions)
                    (ExamDates 0 $?groupExamDates)
                )
    ?adProf <- (Professor
                    (ID ?IDProf)
                    (Disciplines $? ?disciplineName $?)
                    (PrefferedExamDates $?profPrefferedDates1 ?profOption
$?)
                    (OccupiedPrefferedExamDates $?profOccupiedExamDates)
                )
    (test (eq (nth$ (+ (length$ $?profPrefferedDates1) 1)
$?profOccupiedExamDates) no))
    (not (Group
            (Name ?groupName)
            (ExamDates $? ?profOption $?)
         )
    )
    (not
        (and (Professor
                (ID ?IDProf)
                (PrefferedExamDates $?profPrefferedDates2
?profOtherOption &: (< (abs (- ?option1 ?profOtherOption)) (abs (-
?option1 ?profOption))) $?)
                (OccupiedPrefferedExamDates $?profOccupiedExamDates2 &:
(eq (nth$ (+ (length$ $?profPrefferedDates2) 1) $?profOccupiedExamDates2)
no))
             )
             (not (Group
                    (Name ?groupName)
                    (ExamDates $? ?profOtherOption $?)
                  )
             )
        )
    )
    =>
    (modify ?adGroup (Disciplines $?groupDisciplineName ?disciplineName)
                     (ExamDatesOptions $?groupOptions ?option1 ?option2)
                     (ExamDates $?groupExamDates ?profOption)
    )
    (modify ?adProf (OccupiedPrefferedExamDates (replace$
$?profOccupiedExamDates (+ (length$ $?profPrefferedDates1) 1) (+ (length$
$?profPrefferedDates1) 1) ?disciplineName)))
)
```

4.7. Statistics Generation

In order to see the results of the above presented rules, different statistics can be requested (for each one, a rule was created):

- Print Average Final Grades for all Students
- Print Students having Average Final Grades greater than a specified value
- Print Students having at least one Discipline Failed
- Print Average Final Grades for a specified Student
- Print Final Grades from a specified Discipline
- Print Final Grades greater than a specified value from a specified Discipline
- Print Students Failed at a specified Discipline
- Print all Final Grades of a specified Student
- Print Final Grade from a specified Discipline of a specified Student
- Print Scholarships for each Student
- Print Students having a specified Scholarship
- Print Scholarship of a specified Student
- Print Guiding Professors for all Students
- Print Students guided by a specified Professor
- Print Guiding Professor of a specified Student
- Print all Exam Dates
- Print Exam Dates of a specified Group
- Print Exam Dates of a specified Professor
- Print Exam Dates of a specified Student
- Print Information about a specified Student
- Print Information about a specified Professor

The implementation of the rules can be seen in *Appendix 1*.

5. Execution Control and User Interface

For each task, the rules which solve it are triggered by the presence in the Working Memory of a specific Fact. The User Interface is a basic one, which requests the choice of the next task to perform from the user. When an option is chose, its specific Fact is introduced in the Working Memory and the rules which can solve that task are activated. After the completion of a task, the User Interface requests repeatedly another task. This behaviour is assured by retracting and asserting at each iteration a fact which triggers the activation of the User Interface. A rule which loads all the files which contains the rule's implementation is activated only once, at the beginning of the program execution.

Thus, in order to start the program, the file main.clp has to be loaded in the CLIPS environment and then the command (run) has to be introduced in the command line. Then the program will work continuously, only based on the User Interface, until the key 'x' is pressed.

The two rules which ensure the Initialization and the working of the User Interface are:

- `Start`
- `ReadOptions` – the implementation can be seen in *Appendix 2*.

```
(defrule Start
     (not (Step))
     =>
     (load "FactsTemplates.clp")
     (load "InitialFacts.clp")
     (load "AverageGrades.clp")
     (load "OrderStudents.clp")
     (load "Scholarships.clp")
     (load "GuidingProfessors.clp")
     (load "ExamDatesChoice.clp")
     (load "GenerateStatistics.clp")
     (reset)
     (assert (Step))
)
```

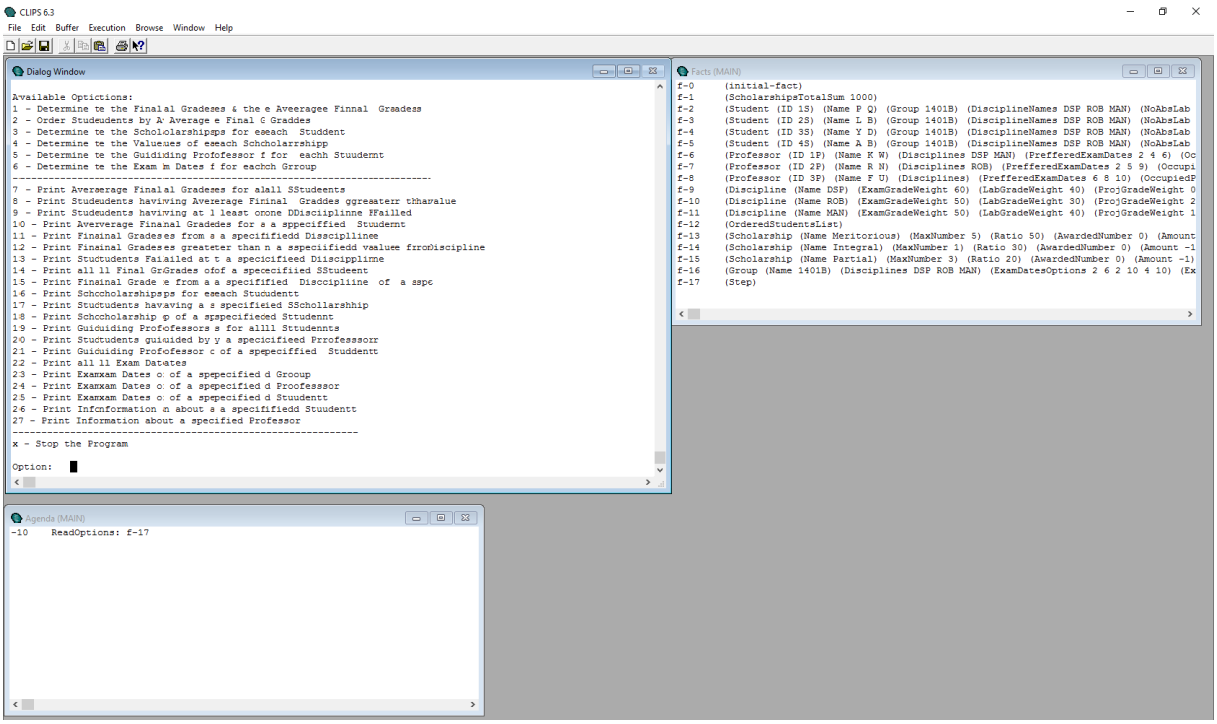

Fig.7. Screenshot showing the User Interface waiting for an option, the Facts Window and the Agenda Window

The testing was performed on a small but relevant data set (covering all the possible cases) in order to follow and to verify better the execution of the program. The results obtained are correct and are obtained in a reduced execution time (based on rough estimations).

6. Program optimization

The implemented rules are partially optimized (The rules which computes the Final grades and those which schedule the Examinations are using the information from the beginning of the patterns) but can be optimized more in order to reduce the execution time and the used memory. This can be done by:

- Moving the most specific patterns to the beginning of the rules
- Move the most specific fields to the beginning of the patterns
- Include the tests of the pattern values inside the pattern, not in the `Test` condition
- Using the = predicate instead of the `eq` predicate when the two parameters are known to be numbers (an also <> instead on `neq`).

7. Future developments

- Program optimization (as explained in the previous section)
- Delete all the trigger facts after the execution of a task (this ensures that a task can be executed multiple times during the same working session of the program)
- Implement a Graphical User Interface in another development environment and merge the two modules.

8. Conclusions

This project shows a possible implementation using rule-based programming of an information manager regarding the students in one year of study. Firstly an ontology was developed according the requirements. Then, the pattern for the facts were designed and then implemented. The rules which solve all the specified tasks (final grade computation, examination scheduling, scholarships allocation, diploma guiding professors' allocation and statistics generation) were designed and implemented in CLIPS programming environment. A simple User Interface facilitates the usage of the implemented tasks. The tests of the entire project demonstrate a good solution and implementation which can still be improved by minimizing the execution time and the used memory and by adding some other features.

References

'Knowledge Representation and Reasoning' Lectures; Pănescu, D.A.; Faculty of Automatic Control and Computer Engineering, 'Gheorghe Asachi' Technical University of Iași, Romania; 2016

Appendix 1.

The rules which are responsible for Statistics Generation

```
;-------------------------------------------------------------------------
;--------- FINAL GRADES ----------------------------------------------------
;-------------------------------------------------------------------------

(defrule PrintAverageFinalGrades
; Print the Final Average Grades for all Students

    (Print AverageFinalGrades)
    (Student
        (ID ?IDStudent)
        (AverageFinalGrades ?AverageFinalGrades)
    )
    =>
    (printout t "Student " ?IDStudent " has the Final Average Grades: "
?AverageFinalGrades crlf)
)

(defrule PrintAverageFinalGradesGreater
; Print the Students who have the Final Average Grades greater than a
specified value

    (Print AverageFinalGrades greater ?thresholdAverageFinalGrades)
    (Student
        (ID ?IDStudent)
        (AverageFinalGrades ?AverageFinalGrades &: (>=
?AverageFinalGrades ?thresholdAverageFinalGrades))
    )
    =>
    (printout t "Student " ?IDStudent " has the Final Average Grades: "
?AverageFinalGrades " (greater or equal than "
?thresholdAverageFinalGrades " )"  crlf)
)

(defrule PrintAverageFinalGradesGreater_NoMatch
; Print a message when no Students have the Final Average Grades greater
than a specified value

    ?ad <- (Print AverageFinalGrades greater
?thresholdAverageFinalGrades)
    (not (Student
            (ID ?IDStudent)
            (AverageFinalGrades ?AverageFinalGrades &: (>=
?AverageFinalGrades ?thresholdAverageFinalGrades))
         )
    )
    =>
    (retract ?ad)
    (printout t "No Student has the Final Average Grades greater or equal
than " ?thresholdAverageFinalGrades crlf)
)
```

```
(defrule PrintAverageFinalGradesFailed
; Print the Students who have at least one discipline Failed

    (Print AverageFinalGrades Fail)
    (Student
        (ID ?IDStudent)
        (FinalGrades $? 0.0 $?)
    )
    =>
    (printout t "Student " ?IDStudent " has Failed at least one
discipline " crlf)
)

(defrule PrintAverageFinalGradesStudent
; Print the Final Average Grades for a specific Student

    ?ad <- (Print AverageFinalGrades Student ?IDStudent)
    (Student
        (ID ?IDStudent)
        (AverageFinalGrades ?AverageFinalGrades)
    )
    =>
    (retract ?ad)
    (printout t "Student " ?IDStudent " has the Final Average Grades: "
?AverageFinalGrades crlf)
)

(defrule PrintFinalGradeDiscipline
; Print the final grade of all Students at a specified discipline

    (Print FinalGrade Discipline ?discipline)
    (Student
        (ID ?IDStudent)
        (DisciplineNames $?disciplineList1 ?discipline $?)
        (FinalGrades $?finalGradesList)
    )
    =>
    (printout t "Student " ?IDStudent " has the final grade " (nth$ (+
(length$ $?disciplineList1) 1) $?finalGradesList) " at Discipline "
?discipline crlf)
)

(defrule PrintFinalGradeDiscipline_NoMatch
; Print a message when no Student studies a specified Discipline

    (Print FinalGrade Discipline ?discipline)
    (not (Student
            (DisciplineNames $? ?discipline $?)
         )
    )
    =>
    (printout t "No Student studies the Discipline " ?discipline crlf)
)
```

```
(defrule PrintFinalGradeDisciplineGreater
; Print the Students having the final grade at a specified Discipline
greater than a specified value

    (Print FinalGrade Discipline ?discipline greater
?thresholdFinalGrade)
    (Student
        (ID ?IDStudent)
        (DisciplineNames $?disciplineList1 ?discipline $?)
        (FinalGrades $?finalGradesList)
    )
    (test (>= (nth$ (+ (length$ $?disciplineList1) 1) $?finalGradesList)
?thresholdFinalGrade))
    =>
    (printout t "Student " ?IDStudent " has the final grade " (nth$ (+
(length$ $?disciplineList1) 1) $?finalGradesList) " (greater or equal
than " ?thresholdFinalGrade " ) at Discipline " ?discipline crlf)
)

(defrule PrintFinalGradeDisciplineGreater_NoMatch1
; Print a message when no Student has the final grade at a specified
Discipline greater than a specified value

    ?ad <- (Print FinalGrade Discipline ?discipline greater
?thresholdFinalGrade)
    (not (Student
            (DisciplineNames $?disciplineList1 ?discipline $?)
            (FinalGrades $?finalGradesList &: (>= (nth$ (+ (length$
$?disciplineList1) 1) $?finalGradesList) ?thresholdFinalGrade))
        )
    )
    =>
    (printout t "No Student has the Final Grade greater of equal than "
?thresholdFinalGrade " at Discipline " ?discipline crlf)
    (retract ?ad)
)

(defrule PrintFinalGradeDisciplineFail
; Print the Students failed at a specified Discipline

    (Print FinalGrade Discipline ?discipline Fail)
    (Student
        (ID ?IDStudent)
        (DisciplineNames $?disciplineList1 ?discipline $?)
        (FinalGrades $?finalGradesList)
    )
    (test (eq (nth$ (+ (length$ $?disciplineList1) 1) $?finalGradesList)
0.0))
    =>
    (printout t "Student " ?IDStudent " failed at Discipline "
?discipline crlf)
)
```

```
(defrule PrintFinalGradeDisciplineFail_NoMatch1
; Print a message when no Student has failed at a specified Discipline

    (Print FinalGrade Discipline ?discipline Fail)
    (not (Student
            (DisciplineNames $?disciplineList1 ?discipline $?)
            (FinalGrades $?finalGradesList &: (eq (nth$ (+ (length$
$?disciplineList1) 1) $?finalGradesList) 0.0))
        )
    )
    =>
    (printout t "No Student has failed at Discipline " ?discipline crlf)
)

(defrule PrintFinalGradeStudent
; Print all the Average Grades of a specified Student

    (Print FinalGrade Student ?IDStudent)
    (Student
        (ID ?IDStudent)
        (DisciplineNames $?disciplineList1 ?discipline $?)
        (FinalGrades $?finalGradesList)
    )
    =>
    (printout t "Student " ?IDStudent " has at Discipline " ?discipline "
the Average Grade " (nth$ (+ (length$ $?disciplineList1) 1)
$?finalGradesList) crlf)
)

(defrule PrintFinalGradeStudentDiscipline
; Print the Average Grades at a specified Discipline of a specified
Student

    ?ad <- (Print FinalGrade Student ?IDStudent Discipline ?discipline)
    (Student
        (ID ?IDStudent)
        (DisciplineNames $?disciplineList1 ?discipline $?)
        (FinalGrades $?finalGradesList)
    )
    =>
    (printout t "Student " ?IDStudent " has at Discipline " ?discipline "
the Average Grade " (nth$ (+ (length$ $?disciplineList1) 1)
$?finalGradesList) crlf)
    (retract ?ad)
)
```

```
;-------------------------------------------------------------------------
;--------- SCHOLARSHIPS ------------------------------------------------
;-------------------------------------------------------------------------

(defrule PrintScholarshipStudent
; Print the Scholarship of a specified Student

    ?ad <- (Print Scholarship Student ?IDStudent)
    (Student
        (ID ?IDStudent)
        (Scholarship ?scholarshipType)
    )
    =>
    (if (eq ?scholarshipType "")
        then (printout t "Student " ?IDStudent " has no scholarship"
crlf)
        else (printout t "Student " ?IDStudent " has " ?scholarshipType "
scholarship" crlf)
    )
    (retract ?ad)
)

(defrule PrintScholarshipType
; Print all the Students having a specified Scholarship

    (Print Scholarship ?scholarshipType)
    (Student
        (ID ?IDStudent)
        (Scholarship ?scholarshipType)
    )
    =>
    (printout t "Student " ?IDStudent " has " ?scholarshipType "
scholarship" crlf)
)

(defrule PrintScholarshipType_NoMatch
; Print a message when no Student has a specified Scholarship

    ?ad <- (Print Scholarship ?scholarshipType)
    (not (Student
            (Scholarship ?scholarshipType)
         )
    )
    =>
    (printout t "No Student has " ?scholarshipType " scholarship" crlf)
    (retract ?ad)
)

(defrule PrintScholarshipWithoutType
; Print all the Students not having any type of Scholarship

    (Print Scholarship None)
    (Student
        (ID ?IDStudent)
        (Scholarship none)
    )
    =>
    (printout t "Student " ?IDStudent " has no scholarship" crlf)
)
```

```
(defrule PrintScholarship
; Print the Scholarship for all the Students who have Scholarships

    (Print Scholarship)
    (Student
        (ID ?IDStudent)
        (Scholarship ?scholarshipType &~ none)
    )
    (Scholarship
        (Name ?scholarshipType)
        (Amount ?amount)
    )
    =>
    (printout t "Student " ?IDStudent " has " ?scholarshipType "
scholarship ( " ?amount " RON )" crlf)
)

;-------------------------------------------------------------------------
;--------- GUIDING PROFESSOR ---------------------------------------------
;-------------------------------------------------------------------------

(defrule PrintGuidingProfStudent
; Print the Guiding Professor of a specified Student

    ?ad <- (Print GuidingProf Student ?IDStudent)
    (Student
        (ID ?IDStudent)
        (OptionsGuidingProf $?optionsGuidingProf)
        (GuidingProf ?IDGuidingProf)
        (IndexGuidingProfChoice ?indexGuidingProf)
    )
    =>
    (if (eq ?IDGuidingProf none)
        then (printout t "Student " ?IDStudent " has not a Guiding
Professor assigned yet" crlf)
        else (if (eq ?IDGuidingProf N/A)
                then (printout t "Student " ?IDStudent " has no valid
options for Guiding Professor" crlf)
                else (printout t "Student " ?IDStudent " has "
?IDGuidingProf " as Guiding Professor (option number " ?indexGuidingProf
" out of " (length$ $?optionsGuidingProf)")" crlf)
             )
    )
    (retract ?ad)
)

(defrule PrintGuidingProfProf
; Print the Students guided by a specific Professor

    (Print GuidingProf Professor ?IDProf)
    (Student
        (ID ?IDStudent)
        (GuidingProf ?IDProf)
    )
    =>
    (printout t "Professor " ?IDProf " guides student " ?IDStudent crlf)
)
```

```
(defrule PrintGuidingProfProf_NoMatch
; Print a message when a specified Professor does not guide any Students

    (Print GuidingProf Professor ?IDProf)
    (not (Student
            (ID ?IDStudent)
            (GuidingProf ?IDProf)
         )
    )
    =>
    (printout t "Professor " ?IDProf " does not guide any students" crlf)
)

(defrule PrintGuidingProf
; Print for each Student the Guiding Professor

    (Print GuidingProf)
    (Student
        (ID ?IDStudent)
    )
    =>
    (assert (Print GuidingProf Student ?IDStudent))
)

;-------------------------------------------------------------------------
;--------- EXAM DATES ----------------------------------------------------
;-------------------------------------------------------------------------

(defrule PrintExamDatesGroup_Initialization
; Prepare to Print all the Exam Dates for a specified Group

    ?ad <- (Print ExamDates Group ?groupName)
    (Group
        (Name ?groupName)
        (Disciplines $?groupDisciplines)
        (ExamDates $?groupExamDates)
    )
    =>
    (retract ?ad)
    (assert (DisciplinesList ?groupDisciplines))
    (assert (ExamDatesList ?groupExamDates))
    (assert (GroupList ?groupName))
)

(defrule PrintExamDatesGroup
; Print all the Exam Dates for a specified Group

    ?adDisc <- (DisciplinesList ?discipline $?otherDisciplines)
    ?adDates <- (ExamDatesList ?examDate $?otherExamDates)
    (GroupList ?groupName)
    =>
    (printout t "Exam Date for Discipline " ?discipline ": Day "
?examDate " ( group " ?groupName " )" crlf)
    (retract ?adDisc)
    (retract ?adDates)
    (assert (DisciplinesList $?otherDisciplines))
    (assert (ExamDatesList $?otherExamDates))
)
```

```
(defrule PrintExamDatesGroup_Clean
; Finalize to Print all the Exam Dates for a specified Group

    ?adDisc <- (DisciplinesList)
    ?adDates <- (ExamDatesList)
    ?adGroup <- (GroupList ?)
    =>
    (retract ?adDisc)
    (retract ?adDates)
    (retract ?adGroup)
)

(defrule PrintExamDatesStudent
; Print all the Exam Dates for a specified Student

    ?ad <- (Print ExamDates Student ?IDStudent)
    (Student
        (ID ?IDStudent)
        (Group ?groupName)
    )
    =>
    (printout t "Student " ?IDStudent ":" crlf)
    (retract ?ad)
    (assert (Print ExamDates Group ?groupName))
)

(defrule PrintExamDates
; Print all the Exam Dates for all Groups

    ?ad <- (Print ExamDates)
    (Group
        (Name ?groupName)
    )
    =>
    (retract ?ad)
    (assert (Print ExamDates Group ?groupName))
)

(defrule PrintExamDatesProfessor
; Print all the Exam Dates for a specified Professor

    ?ad <- (Print ExamDates Professor ?IDProf)
    (Professor
        (ID ?IDProf)
        (PrefferedExamDates $?prefferedExamDates)
        (OccupiedPrefferedExamDates $?occupiedExamDates ?discipline &~ no
$?)
    )
    =>
    (printout t "Professor " ?IDProf " has exam on Day " (nth$ (+
(length$ $?occupiedExamDates) 1) $?prefferedExamDates) " at Discipline "
?discipline crlf)
)
```

```
(defrule PrintExamDatesProfessor_NoMatch
; Print a message when a specified Professor does not have any exams

    ?ad <- (Print ExamDates Professor ?IDProf)
    (not (Professor
            (ID ?IDProf)
            (OccupiedPrefferedExamDates $? ?discipline &~ no $?)
         )
    )
    =>
    (printout t "Professor " ?IDProf " does not have any exams" crlf)
)

;-------------------------------------------------------------------------
;--------- INFORMATION ---------------------------------------------------
;-------------------------------------------------------------------------

(defrule PrintInfoStudent
; Print Information about a specified Student

    ?ad <- (Print Info Student ?IDStudent)
    (Student
        (ID ?IDStudent)
        (Name $?name)
        (Group ?group)
    )
    =>
    (printout t "Student " ?IDStudent ":" crlf)
    (printout t "Name: " $?name crlf)
    (printout t "Group: " ?group crlf)
    (assert (Print FinalGrades Student ?IDStudent))
    (assert (Print AverageFinalGrades Student ?IDStudent))
    (assert (Print Scholarship Student ?IDStudent))
    (assert (Print GuidingProf Student ?IDStudent))
    (retract ?ad)
)

(defrule PrintInfoProf
; Print Information about a specified Professor
    ?ad <- (Print Info Professor ?IDProf)
    (Professor
        (ID ?IDProf)
        (Name $?name)
        (Disciplines $?disciplines)
    )
    =>
    (printout t "Professor " ?IDProf ":" crlf)
    (printout t "Name: " $?name crlf)
    (printout t "Disciplines: " $?disciplines crlf)
    (assert (Print ExamDates Professor ?IDProf))
    (assert (Print GuidingProf Professor ?IDProf))
    (retract ?ad)
)
```

Appendix 2.

The rule which implements the User Interface

```
(defrule ReadOptions
    (declare (salience -10))

    ?ad <- (Step)
    =>
    (printout t crlf)
    (printout t "Available Options:" crlf)
    (printout t "1 - Determine the Final Grades & the Average Final
Grades" crlf)
    (printout t "2 - Order Students by Average Final Grades" crlf)
    (printout t "3 - Determine the Scholarships for each Student" crlf)
    (printout t "4 - Determine the Values of each Scholarship" crlf)
    (printout t "5 - Determine the Guiding Professor for each Student"
crlf)
    (printout t "6 - Determine the Exam Dates for each Group" crlf)
    (printout t "-------------------------------------------------------
----" crlf)
    (printout t "7 - Print Average Final Grades for all Students" crlf)
    (printout t "8 - Print Students having Average Final Grades greater
than a specified value" crlf)
    (printout t "9 - Print Students having at least one Discipline
Failed" crlf)
    (printout t "10 - Print Average Final Grades for a specified Student"
crlf)
    (printout t "11 - Print Final Grades from a specified Discipline"
crlf)
    (printout t "12 - Print Final Grades greater than a specified value
from a specified Discipline" crlf)
    (printout t "13 - Print Students Failed at a specified Discipline"
crlf)
    (printout t "14 - Print all Final Grades of a specified Student"
crlf)
    (printout t "15 - Print Final Grade from a specified Discipline of a
specified Student" crlf)
    (printout t "16 - Print Scholarships for each Student" crlf)
    (printout t "17 - Print Students having a specified Scholarship"
crlf)
    (printout t "18 - Print Scholarship of a specified Student" crlf)
    (printout t "19 - Print Guiding Professors for all Students" crlf)
    (printout t "20 - Print Students guided by a specified Professor"
crlf)
    (printout t "21 - Print Guiding Professor of a specified Student"
crlf)
    (printout t "22 - Print all Exam Dates" crlf)
    (printout t "23 - Print Exam Dates of a specified Group" crlf)
    (printout t "24 - Print Exam Dates of a specified Professor" crlf)
    (printout t "25 - Print Exam Dates of a specified Student" crlf)
    (printout t "26 - Print Information about a specified Student" crlf)
    (printout t "27 - Print Information about a specified Professor"
crlf)
    (printout t "-------------------------------------------------------
----" crlf)
    (printout t "x - Stop the Program" crlf)

    (printout t crlf)
```

```
    (printout t "Option: ")
    (bind ?optiune (readline))

    (if (eq ?optiune "1")
        then
            (assert (Determine FinalGrades))
    )
    (if (eq ?optiune "2")
        then
            (assert (ToDo CreateStudentsList))
            (assert (ToDo OrderStudents))
    )
    (if (eq ?optiune "3")
        then
            (assert (Determine Scholarships))
    )
    (if (eq ?optiune "4")
        then
            (assert (Determine ValuesScholarships))
    )
    (if (eq ?optiune "5")
        then
            (assert (Determine GuidingProf))
    )
    (if (eq ?optiune "6")
        then
            (assert (Determine ExamDates))
    )
    (if (eq ?optiune "7")
        then
            (assert (Print AverageFinalGrades))
    )
    (if (eq ?optiune "8")
        then
            (printout t "Value: ")
            (bind ?value (read))
            (assert (Print AverageFinalGrades greater ?value))
    )
    (if (eq ?optiune "9")
        then
            (assert (Print AverageFinalGrades Fail))
    )
    (if (eq ?optiune "10")
        then
            (printout t "Student ID: ")
            (bind ?value (read))
            (assert (Print AverageFinalGrades Student ?value))
    )
    (if (eq ?optiune "11")
        then
            (printout t "Discipline: ")
            (bind ?value (read))
            (assert (Print FinalGrade Discipline ?value))
    )
    (if (eq ?optiune "12")
        then
            (printout t "Value: ")
            (bind ?value (read))
            (printout t "Discipline: ")
            (bind ?value2 (read))
            (assert (Print FinalGrade Discipline ?value2 greater ?value))
    )
```

```
    (if (eq ?optiune "13")
        then
            (printout t "Discipline: ")
            (bind ?value (read))
            (assert (Print FinalGrade Discipline ?value Fail))
    )
    (if (eq ?optiune "14")
        then
            (printout t "Student ID: ")
            (bind ?value (read))
            (assert (Print FinalGrade Student ?value))
    )
    (if (eq ?optiune "15")
        then
            (printout t "Student ID: ")
            (bind ?value (read))
            (printout t "Discipline: ")
            (bind ?value2 (read))
            (assert (Print FinalGrade Student ?value Discipline ?value2))
    )
    (if (eq ?optiune "16")
        then
            (assert (Print Scholarship))
    )
    (if (eq ?optiune "17")
        then
            (printout t "Scholarship Type (Meritorious, Integral,
Partial, None): ")
            (bind ?value (read))
            (assert (Print Scholarship ?value))
    )
    (if (eq ?optiune "18")
        then
            (printout t "Student ID: ")
            (bind ?value (read))
            (assert (Print Scholarship Student ?value))
    )
    (if (eq ?optiune "19")
        then
            (assert (Print GuidingProf))
    )
    (if (eq ?optiune "20")
        then
            (printout t "Professor ID: ")
            (bind ?value (read))
            (assert (Print GuidingProf Professor ?value))
    )
    (if (eq ?optiune "21")
        then
            (printout t "Student ID: ")
            (bind ?value (read))
            (assert (Print GuidingProf Student ?value))
    )
    (if (eq ?optiune "22")
        then
            (assert (Print ExamDates))
    )
```

```
    (if (eq ?optiune "23")
        then
            (printout t "Group name: ")
            (bind ?value (read))
            (assert (Print ExamDates Group ?value))
    )
    (if (eq ?optiune "24")
        then
            (printout t "Professor ID: ")
            (bind ?value (read))
            (assert (Print ExamDates Professor ?value))
    )
    (if (eq ?optiune "25")
        then
            (printout t "Student ID: ")
            (bind ?value (read))
            (assert (Print ExamDates Student ?value))
    )
    (if (eq ?optiune "26")
        then
            (printout t "Student ID: ")
            (bind ?value (read))
            (assert (Print Info Student ?value))
    )
    (if (eq ?optiune "27")
        then
            (printout t "Professor ID: ")
            (bind ?value (read))
            (assert (Print Info Professor ?value))
    )

    (if (eq ?optiune "x")
        then
            (halt)
    )

    (printout t crlf)
    (retract ?ad)
    (assert (Step))

)
```