12-2021

PSL-An Expert System to Evaluate Degree Plans

Robert Swanson
Taylor University, robert_swanson@taylor.edu

PSL: An Expert System to Evaluate Degree Plans

Robert Swanson

December 13, 2021

Abstract

This paper describes a general-purpose expert system to evaluate degree plans according to the individual preferences of a college student. This system implements a preference specification language (PSL) on top of this expert system to allow for the textual expression of certain requirements and preferences that the system uses for evaluation. The PSL evaluator produces a single value to describe how well it meets the student's preferences, which a plan generation system could use to create a degree plan optimized according to the specification.

1 Introduction

Constructing a valid and efficient degree plan poses one of the most stressful tasks for many college students who often feel overwhelmed either by the large number of course choices or by the constraints of their degree requirements. The college course scheduling problem explores how systems might automate this task. Such systems offer the great potential to help students succeed in their college education by taking classes that fit their needs better. They can also assist professors in the oft-undesirable job of creating a customized degree plan for every student they advise.

1.1 Existing Work

Existing solutions to this problem come in two general flavors:

1. Genetic algorithms (GAs) which generate many (often invalid) plans and evaluate them according to a certain fitness function.
2. Constraint-based searches which deterministically search a valid plan space

For example, one system implements a GA that defines a fitness function on four chosen properties including the number of prerequisite violations, and the number of course assignments that deviate from a given curriculum plan [15].

Another system used a prolog implementation to execute a backtracking search through certain constraints and could produce a list of plans that best meet these constraints [5]. This system offers the theoretical flexibility to define arbitrary constraints using Prolog's logical paradigm, which inspired the eventual design for the expressive PSL language. However, this system also makes certain assumptions about the use case, such as that non-major classes can fit easily around major classes.

Percepolis implements yet another solution that uses a graph-based approach to construct an optimized plan [12] [11]. The system defines graph relationships between degree requirements and the courses that satisfy those requirements. Such a design also allows for the expression of prerequisite constraints in the native searching algorithm, which results in a search space more constrained to the valid plan space. This relationship model offers great performance advantages but also limits the kinds of requirements that can be expressed. It also offers no ability to consider course offering times, making the assumption that any combination of courses in a given semester will work provided a certain maximum number of courses.

1.2 Objective

All these systems assert certain constraints on the optimization of the generated plans. Though they likely lead to great performance improvements, they greatly reduce the potential usefulness to students who usually have certain custom preferences for their degree plan.

This project explores how to construct highly customized degree plans, focusing more on the expression of preferences and less on the performance of the system.

Though the PSL system does not currently implement plan generation abilities, the ability to produce a single score from a plan produces a method by which to feasibly generate an optimized plan. For example, a brute force GA solution could simply use the evaluation system as a fitness function for a GA, and then construct chromosomes with the degree plan.

2 PSL Prototype

2.1 Preferences

The first PSL version served as a prototype to test out how a domain-specific language might enable students to express complex preference structures [17].

This included performing an informal survey of undergraduate students to gather a broad idea of the kinds of features of degree plans that students attempt to optimize towards. Organizing all of these preferences into a structure that could be expressed as a DSL proved a difficult task due to the large divergence in the nature of the preferences.

For example, some students stated that they optimized their plans to increase the number of classes they'd have with their friends, while others sought to minimize the walking distance between their classes. For the sake of simplicity, we constrained ourselves to the subset of preferences that depend only on information known by the registration system concerning a single student (excluding the two aforementioned preferences).

2.2 Prototype Design

The prototype limited the scope of preferences to a few values (credits, number of courses, course order) each with a quantifier hardcoded in the syntax.

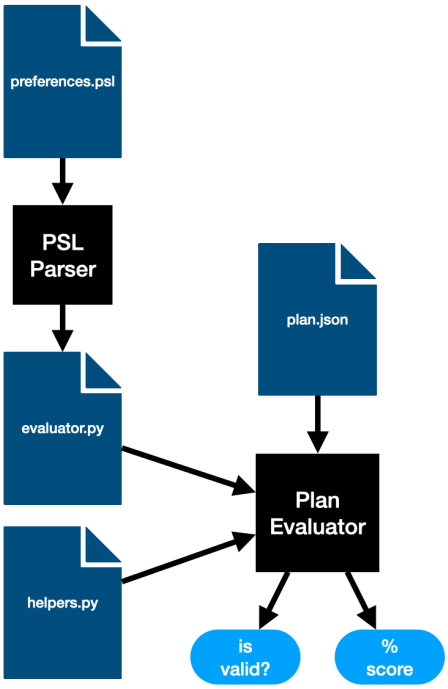


Figure 1: The Prototype's Dataflow

```
prefer more than 15 credits.
```

Listing 1: Preference Scored With Sigmoid

Figure 1 shows the high-level structure of the PSL prototype. It starts with the student preferences expressed in a `.psl` file. The PSL parser, implemented in ANTLR for Java, takes this file and produces `evaluator.py` file. This file imports the `helpers.py` file and serves as the reusable plan evaluator. This evaluator takes a `plan.json` file and produces two output values describing how well the plan fulfills the specification:

1. A percentage score describing how well the plan fulfills the **preferences** in the specification
2. A boolean value describing if all the plan fulfills all of the **requirements** in the specification

This distinction between requirements and preferences allows the student to express which components of the specification should result in the system completely rejecting the plan.

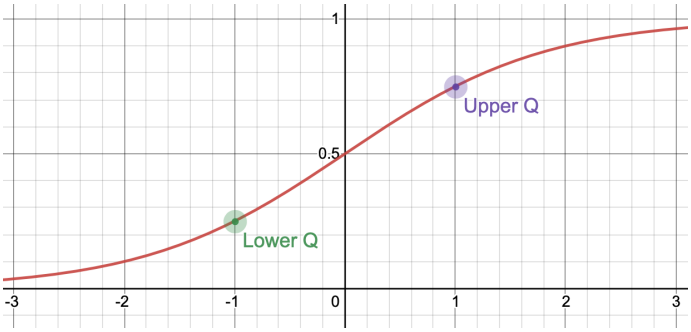
2.3 Scoring

$$s(x) = \frac{1}{1 + 9^{-x}} \tag{1}$$

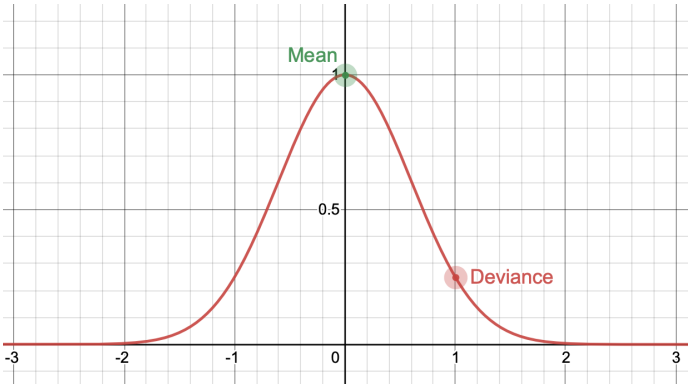
$$n(x, l, u) = \frac{x - l}{u - l} \tag{2}$$

To produce a single percentage value that described the overall result of multiple preferences, multiple scoring functions were needed to account for different desired behaviors.

Equation 1 shows a variant of the sigmoid function that can compute scores for preferences that sought to maximize or minimize a theoretically unbounded value (for example desiring more than 15 credits in a semester). The constant 9 ensures that input values of -1 and 1 result in output scores



(a) Sigmoid



(b) Optimum

Figure 2: Scoring Functions

```
prefer 15 credits.
```

Listing 2: Preference Scored With Optimum

of 25% and 75% respectively. These values serve as anchor points to equation 2 can normalize given different expected lower and upper bounds.

For example, the listing 1 describes a preference that will produce its score by passing the number of credits into $s(n(x, 15, 17))$, where 17 is computed as a hardcoded deviance from the provided value. This normalization function also allows for l to be greater than u, which allows for preferences to reward lower input values.

$$o(x) = (\frac{1}{4})^{x^2} \tag{3}$$

$$n(x, m, d) = \frac{x - m}{d} \tag{4}$$

An optimizing function scores preferences that seek closeness to a particular value. Equation 3 shows the chosen equation, which anchors -1 and 1 input values the first quartile (25%). Equation 4 implements a normalizing function that takes in the mean value representing the optimum input value and also takes in the deviance from that mean that should result in a 25% score.

For example, listing 2 describes a preference that would define the score as $o(n(15, 1))$ where 1 is the hardcoded deviance for that preference. The optimum function could also distinguish between left and right deviance to support punishing deviance to one side over the other.

These scoring functions (in conjunction with others that allow for hard boundaries) can be configured differently for different evaluators to produce percentages that stay between 0% and 100%, but show the most extreme change in

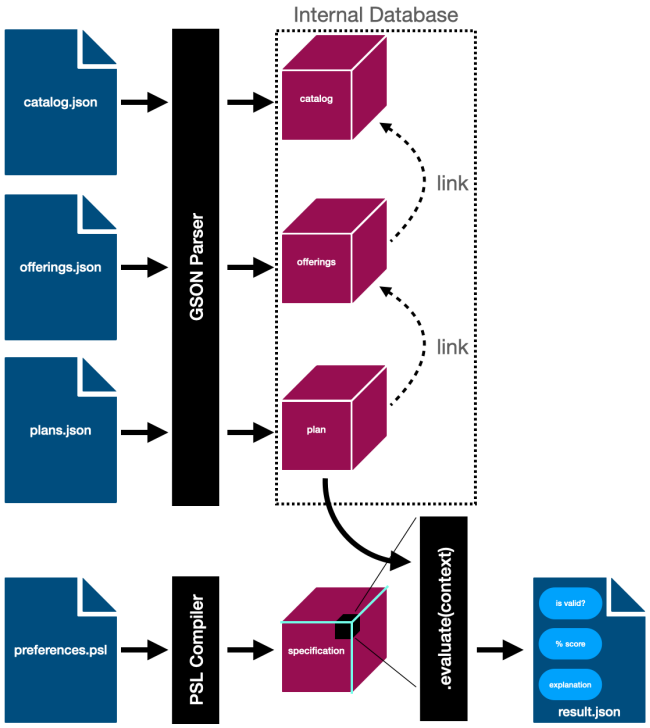


Figure 3: PSL System Overview

the portion of the domain that is most expected.

2.4 Weaknesses

The completion of the prototype demonstrated the feasibility of creating a domain-specific language to express preferences that could be used to automatically evaluate a degree plan. However, the prototype suffered several design flaws that limited its usability.

First off, the splitting between a java parser and a python evaluation system resulted in significant overhead including launching a new process for every new plan (an inhibitor to the plan generation objective) and implementing a python generation library that constructed evaluators as strings. The final design featured a unified Java implementation which resulted in a far more sensible specification construction process.

The prototype also suffered from an inflexible grammar that allowed for no construction of constraints that weren't hard-coded in the grammar. The final implementation would build on this by implementing a modular grammar that could combine evaluators with different quantifiers to construct a larger number of constraints.

3 PSL Design

3.1 Overview

As figure 3 illustrates, The PSL system inherited the prototype's basic high-level structure, but unified the parser and evaluator into a single Java project and implemented a highly robust plan evaluation SDK [16].

The system takes in four input files:

1. `Catalog.json`: contains general information about courses (eg. course name)
2. `Offerings.json`: contains term-specific information about course offerings/sections (eg. offering time)
3. `Plan.json`: lists terms and the course offerings within each to indicate the degree plan the system will evaluate
4. `Preferences.psl`: defines the student requirements and preferences for their degree plan

It produces a single output file, `Result.json` containing the evaluation result along with a structured description of the system's reasoning.

On startup, the system uses `gson` to parse the input `json` files into an object structure that serves as the system's internal database of knowledge for plan details necessary to evaluate it. After parsing the `json`, the system dynamically links the objects to each other (effectively performing inner joins). This allows for the inference of course info given a specific course offering and for the inference of a course offering given a `courseID` within a degree plan. Unlike the prototype, the system only needs to load this information once, even when the system evaluates multiple plans or uses multiple specifications.

Just as the prototype, the PSL parser takes in a `.psl` file defining the student preferences. However, unlike before, the parser constructs and produces a `Specification` java object, holding the compiled preference information. This object can evaluate degree plans and determine whether it meets all the requirements and how well it meets the preferences. It can also explain the logic behind the result of the whole specification as well as any of its sub-components.

3.2 PSL Grammar

Listing 3 shows a sample PSL file that demonstrates the expressiveness of the language. It shows how any combination of quantifiers and evaluators can construct a constraint. This design necessitates only a single definition of any evaluator rather than a separate definition as the prototype did.

Listing 6 (Appendix A.1) shows the important definitions in the PSL grammar. It consists of one or more named blocks, where each block can have a defined set up priority definitions, followed by one or more specifications.

3.2.1 Specifications

The grammar defines 5 kinds of specifications: preferences, requirements, conditionals, contextuals, and lists.

Preference specifications define a priority which determines how much that specification will influence the overall score. For example, preferences using the `strongly` priority have 5 times more impact on the plan score than preferences that use the `moderately` priority. They also contain a constraint that defines an attribute of a plan and what the student desires about that attribute.

Requirement specifications contain a single requireable constraint. Requireable constraints compose a subset of constraints that, in addition to supporting continuous scoring (for preference purposes), also support boolean validation (for requirement purposes. For example, the `more` constraint allows a user to instruct the system to maximize a particular value. Because they don't specify a minimum value, the constraint can produce a score that increases along with its input value, it cannot return the boolean value needed for requirements.

```
student_preferences (moderately=2.0, strongly=10.0) {
    require plan starting in fall 2018.
    require plan ending on or before spring 2022.

    prefer strongly starting at or after 9:00 AM.
    prefer strongly more courses.
    prefer strongly taking course "COS-121".

    if taking course "COS-120" then {
        prefer moderately taking course "COS-120" before fall 2019.
    } otherwise if taking course "SYS-120" then {
        prefer moderately taking course "SYS-120" before fall 2019.
    }

    for terms where less than 16 credits {
        prefer not meeting at 12:00 PM - 12:50 PM.

        for days where less than 120 meeting minutes {
            prefer ending before 1:00 PM.
        }
    }

    for days where (at least 2 courses or not meeting at 12:00 PM - 12:50 PM) {
        prefer meeting at 11:00 AM - 11:50 AM.
    }

    for thursdays prefer strongly not meeting at 2:00 PM - 4:00 PM.
}
```

Listing 3: Sample PSL File

Conditional specifications contain a set of condition-specification pairs. The conditional specification will evaluate any specifications paired with conditions that resolve as true. Conditions support boolean logic with an atomic unit of a requireable constraint (which supports boolean evaluation).

Contextual specifications define a context level, a condition to serve as the context filter, and a specification. Contexts allow the student to describe preferences on three levels:

1. With respect to the full plan
2. With respect to one or more terms
3. With respect to one or more weekdays

Contextual specifications use the provided condition to determine which sub-contexts will make up the new context. Evaluators may provide different values depending on their context. For example, a `credits` evaluator in the full plan context returns the total number of credits, but the same evaluator in a `terms` context will return a list of credit counts for each term.

Specification lists allow any grammatical element that expects a specification to support receiving more than one specification in a curly-braced-surrounded list. Conditional and contextual specifications serve as the primary user of the specification lists, but they can also stand alone and serve as a visual grouping element without impacting the system's behavior.

This design allows for any arbitrary nesting of conditional and contextual specifications with one notable exception: contextual specifications may not semantically broaden the context level. For example, the listener will throw an error if a `for days where...` specification nests a `for terms where...` specification.

3.2.2 Constraints

Specifications eventually boil down to a series of constraints, which describe certain restrictions on a value.

Requireable constraints generally group a value, quantifier, and evaluator. For example, an equal constraint states that the user wants a particular value to evaluate close to a given value. PSL implements 5 quantifiers corresponding to the $=$, $<$, $>$, $\leq$, and $\geq$ operators. Any boolean constraint (defined as any boolean evaluator) can also compose a requireable constraint.

Non-requireable constraints contain only the quantifier and evaluator. They allow a user to specify only that they want to maximize or minimize a particular value, but without specifying how high or low they want the value.

3.2.3 Evaluators

Digging down another level of abstraction, we find that evaluators construct constraints. Evaluators describe a particular attribute of a plan (more specifically a context from a plan). Evaluators can be characterized by the type of value they return and by how they respond to their context level.

The PSL system supports 4 return types from evaluators: numeric, term-year, time, and boolean. Notably, a term-year evaluator can also serve as a boolean evaluator, for cases where the student wants to ensure that the evaluator returns a non-null value (for example requiring a class, but for any time).

3.3 Evaluation Engine

3.3.1 Specification

The evaluation engine provides an SDK to construct a `Specification` type object capable of evaluating a degree plan. The PSL language enables a text-based interface to the evaluation engine, but the engine could also interface with a graphical user interface to construct a specification. Figure 4 shows the high-level view of the basic class construction hierarchy. Specifications are essentially constructed of `Constraint` objects, which, in turn, are constructed of `ContextEvaluator` objects.

The evaluation engine's structure closely resembles the structure of the grammar, starting with the highest level of abstraction at the level of a `Specification`. **This level takes a `Context` as input and produces a `Score` as an output.**

This `Score` object contains a boolean value describing whether all requirements have been met, an accumulator describing the "points" accumulated, and a maximum describing the maximum number of "points" to divide out of the accumulator. Because the `Score` preserves the integrity of the denominator, any specification that contributes to the score will have an impact on the final score directly proportional to its priority (regardless of the PSL structure).

The evaluation engine generally seeks to follow a functional programming paradigm, where any part of the specification can compute its value dynamically given a particular context. The `evaluate()` function of the `Specification` class shown in listing 4 demonstrates this design through its 'context' input value. This value contains all the information that the specification will evaluate according to its nature. The '`evaluateAll` parameter defines whether the evaluation engine should run through all the nested specifications, or if it should stop once a requirement violation invalidates the plan.

The specification also defines the `getSimplifiedSpecification()` function, which returns a version of itself that optimizes out any unnecessary complexity (eg. extracting out the specification from a specification list of size 1).

As figure 4 shows, 6 classes implement the `Specification` abstract class, 5 of which correspond to the PSL grammar's specification definitions. The remaining specification simply serves as the top-level container for a specification.

3.3.2 Constraints

Just as in the grammar, `Constraint` objects build up the `Specification`. **This level also takes a `Context` as input but produces a double as an output.**

Listing 4 shows that all constraints must implement the `score()` function. This function is used by preference specifications to produce a double value between 0 and 1 according to how well the provided context meets the constraint.

Listing 4 also shows the implementation for `RequireableConstraint`, which extends `Constraint`. Requireable constraints add the ability to call the `fulfilled()` function which returns a boolean value according to whether or not the context fulfills the constraint.

6 classes extend the `RequireableConstraint` class, implementing the 5 different quantifiers ($=$, $>$, $<$, $\geq$, $\leq$), and also the boolean constraint. The first five constraints take a context evaluator and compare the result to a static value. The boolean constraint simply returns the result from a boolean evaluator.

Two classes extend only the `Constraint` class: `MoreConstraint` and `LessConstraint`. These constraints take a context evaluator and score them according to the sigmoid function normalized to hardcoded values for that context evaluator (given its context level).

3.3.3 Context

The `Context` class serves two primary purposes:

1. As a container for a degree plan that can scope out (disable) certain sub-contexts (to focus on certain terms or weekdays)

2. To maintain a `ContextLevel` state that specifies the context level that context-sensitive evaluators should use

The context system employs a complex structure of sub-context as shown in figure 5. The PSL system supports 4 context layers:

1. `Context`: the top level context that contains pointers to the term contexts

2. `TermSubContext`: the second level context which contains a list of all the `CourseOfferings` scheduled for the term, and also pointers to the week contexts

3. `WeekSubContext`: the third level context (which is not exposed to the user), but facilitates courses that start or end partially through the semester. Contains pointers to weekday contexts

4. `WeekdaySubContext`: the fourth level context that directly contains the `Meeting` objects representing the planned classes meeting that day

Each context level implements a `applyContextFilter()` function which takes in a condition as defined from a contextual specification. That condition is evaluated for each sub-context to select which sub-contexts will remain active.

For example, the condition defined in line 15 of listing 3 is applied to each term in the context. Only terms that satisfy the condition are considered by the nested specifications. Notice that the condition specifies terms with less than 16 credits, even though the contextual specification lies in the full-plan context. Because the condition is applied to each term, the `credits` evaluator will be applied to each term, and thus will be scored according to the hard-coded deviance specified for that evaluator.

Contexts also support filtering explicitly by term-years or by weekdays as demonstrated in line 27 of listing 3.

Context filters are applied to the context when evaluating nested specifications by pushing the new context to a

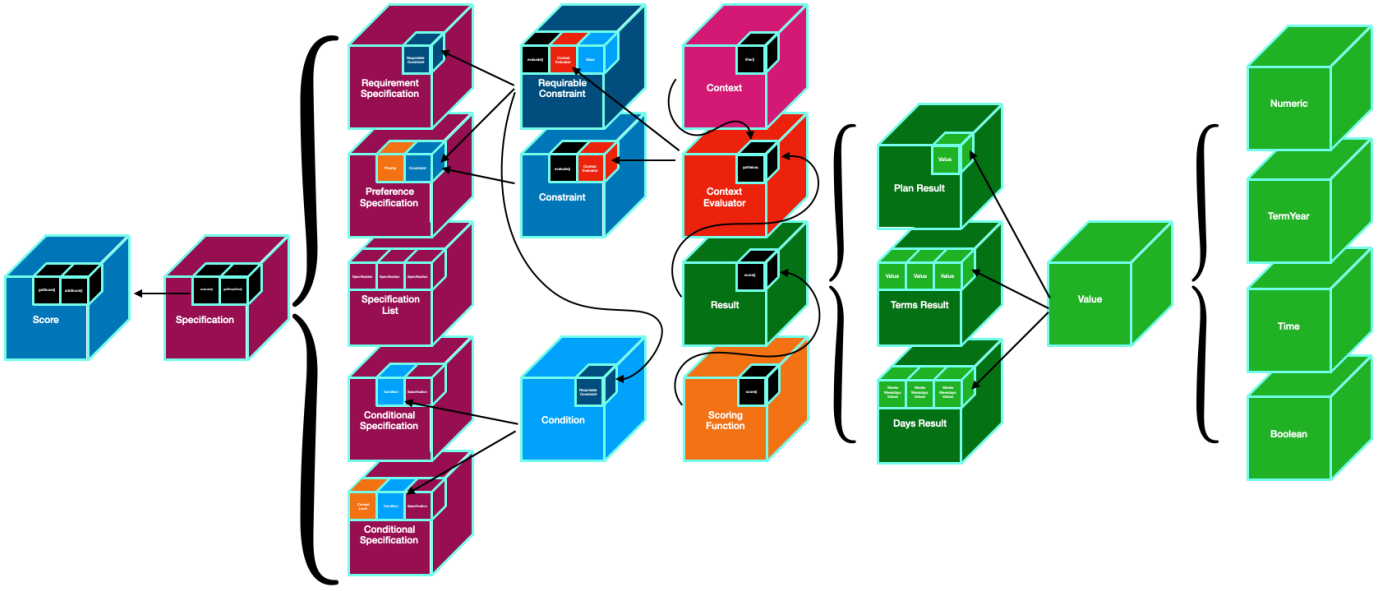


Figure 4: Evaluation Engine

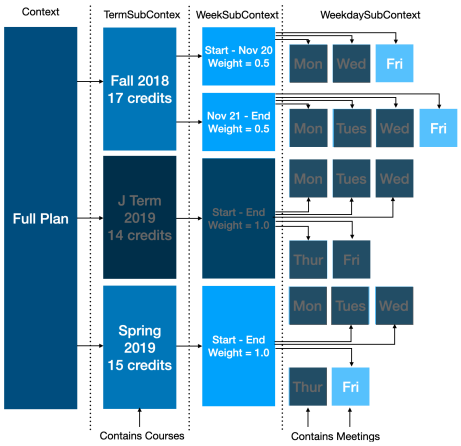


Figure 5: Context Levels

stack. Filters are then unapplied by popping that stack before evaluating specifications outside the scope of the contextual specification.

The context system exposes a series of iterable objects to context evaluators to simplify their implementation. These iterators allow for contexts to change without necessitating any overhead costs of recomputing course offerings or meetings in scope.

These iterators allow for 1st, 2nd, and 3rd degree looping through the context. For example, the `PlanMeetingIterator` simply iterates through the meetings within the plan. However, the `PlanTermMeetingIterable` iterates through `TermMeetingIterable` objects, which allow for direct iteration of the meetings for a respective term. Another level of iterators allows for iterating through the meetings in a given weekday, and all these iterators also exist for course offering iteration.

3.3.4 Context Evaluators

`ContextEvaluator` objects build up `Constraint` objects and serve as the dynamically computed value. As shown in listing 4, context evaluators must implement a `getValue()` function which returns a `Result` object corresponding to the current context. The `Result` object is extend by three classes corresponding to each of the 3 exposed context levels:

1. `PlanResult`: contains a single `Value` object
2. `TermsResult`: contains a list of `Value` objects, one for each term in context
3. `DaysResult`: a three-dimensional array of `Value` objects, for each term, for each week, for each weekday in context

The `Value` object type serves as the atomic unit for evaluator results. There are 4 types of values: `Boolean` `Numeric`, `Time`, and `TermYear`.

A given context evaluator can only return results with one value type, but they can return any of the three result types depending on the context the evaluator is run in.

The `Result` subclasses each enable scoring by exposing the `scoreResult()` function which takes in a lambda expression `resultScorer` which is appropriately applied to each value in the result and averaged to produce a single score. Context evaluators use these functions and define the lambda expression using its respective scoring function.

3.3.5 Explanation

In addition to providing a single `Score` object for the top-level specification, the PSL system can offer a comprehensive explanation for that score.

This is accomplished by implementing an object hierarchy extending the `Explanation` class. This hierarchy roughly reflects the structure of the system itself and provides runtime information regarding the scoring of specifications, filtering of contexts, and results of evaluators. The system uses the `gson` library to encode the class structure as a json file, as shown in listing 5.

The explanation system offers several advantages. First, because the explanation is comprehensive in showing the information propagated through the system, it can aid in debugging. Second, it can serve as a simplified visualization of how the evaluator engine works. A graphical user

```java
public abstract class Specification implements Explainable {
    public abstract Score evaluate(Context context, boolean evaluateAll);
    public abstract Specification getSimplifiedSpecification();
    public abstract SpecificationResultExplanation explainLastResult();
}

public abstract class Constraint implements Explainable {
    protected ContextEvaluator contextEvaluator;
    public abstract double score(Context context);
    public ConstraintResultExplanation explainLastResult() { ... }
}

public abstract class RequireableConstraint extends Constraint {
    public abstract boolean fulfilled(Context context);
}

public abstract class ContextEvaluator implements Explainable {
    abstract public Result getValue(Context context);
    public ContextEvaluatorResultExplanation explainLastResult() { ... }
}
```

Listing 4: Important Class Definitions

```
{
  "planContext": { ... },
  "specification": {
    "constraint": {
      "evaluator": {
        "result": "36",
        "description": "credits"
      },
      "description": "less credits"
    },
    "score": "valid (100%)",
    "description": "prefer [1.0] less credits"
  },
  "score": "valid (100%)",
  "description": "simple.psl"
}
```

Listing 5: Example Explanation File

interface could provide the explanation's information to the user in an understandable way, improving the usefulness of the system by helping the student understand why it thinks certain plans are better than others.

4 Future Work

4.1 Developing the PSL System

The PSL system lays out a framework by which plan evaluation can happen, but degree preferences vary widely from person to person and many people try to optimize their plans based on traits others wouldn't think about. Those applying this to a broader population could implement more context evaluators to accommodate these diverse desires. Such context evaluators would likely eventually require new value types, but should otherwise not require much change in the implementation.

Future work on PSL should also include robust debugging, as time constraints prevented such from happening as of yet. During that process, one could extract the hard-coded mean and deviance values from the context evaluators and put them into a robust, external source of truth. Alternatively, one could find a dynamic approach to constructing these values rather than hard coding them.

The explanation system could also be further developed, to include more helpful information deeper down into the system (eg show the scoring function values).

4.2 Plan Generation

The most important piece of future work would be to take the PSL system and apply it towards plan generation. One might start by implementing a genetic algorithm that uses PSL as a fitness function, and encodes a degree plan as a chromosome. One could observe the rate of improvement the GA offers and how capable such a GA is at providing a known "best plan".

However, such an approach would likely suffer extreme performance issues, so further work should include digger deeper into constraint-based searching, strategically limiting the search space, and taking advantage of the design of the PSL evaluator to better accommodate generation (eg the continuity of the scoring functions).

5 Conclusion

The PSL system demonstrates how student preferences for their degree plan can be expressed as a series of specifications. It demonstrates how a domain-specific language can facilitate highly complex preferences with a reasonably simple syntax.

The design of the evaluation engine shows how a well-constructed system, following standard software engineering practices for quality code, can allow for a customizable system that can readily accept new context evaluators.

The PSL system shows how a plan's adherence to a set of preferences can be mathematically stated and optimized. The scoring functions allow for a high degree of flexibility in what degree plans can be optimized towards.

However, the PSL system's usefulness on its own pales in comparison to its potential usefulness as a part of a degree plan generation system.

References

[1] Dennise Adrianto. Comparison using particle swarm optimization and genetic algorithm for timetable scheduling. *Journal of Computer Science*, 10(2):341, 2014.

[2] Esra Aycan and Tolga Ayav. Solving the course scheduling problem using simulated annealing. In *2009 IEEE International Advance Computing Conference*, pages 462–466. IEEE, 2009.

[3] Sorathan Chaturapruek, Thomas S Dee, Ramesh Johari, René F Kizilcec, and Mitchell L Stevens. How a data-driven course planning tool affects college students' gpa: evidence from two field experiments. In *Proceedings of the Fifth Annual ACM Conference on Learning at Scale*, pages 1–10, 2018.

[4] Tim B. Cooper and Jeffrey H. Kingston. The complexity of timetable construction problems. In Edmund Burke and Peter Ross, editors, *Practice and Theory of Automated Timetabling*, volume 1153, pages 281–295. Springer Berlin Heidelberg. Series Title: Lecture Notes in Computer Science.

[5] Joshua Eckroth and Ryan Anderson. Tarot: A course advising system for the future. *J. Comput. Sci. Coll.*, 34(3):108–116, January 2019.

[6] Wilhelm Erben and Jürgen Keppler. A genetic algorithm solving a weekly course-timetabling problem. In Edmund Burke and Peter Ross, editors, *Practice and Theory of Automated Timetabling*, volume 1153, pages 198–211. Springer Berlin Heidelberg. Series Title: Lecture Notes in Computer Science.

[7] Sadaf Naseem Jat and Shengxiang Yang. A guided search genetic algorithm for the university course timetabling problem. 2009.

[8] Rhydian Lewis. A survey of metaheuristic-based techniques for university timetabling problems. 30(1):167–190.

[9] Junrie B Matias, Arnel C Fajardo, and Ruji P Medina. A hybrid genetic algorithm for course scheduling and teaching workload management. In *2018 IEEE 10th International Conference on Humanoid, Nanotechnology, Information Technology, Communication and Control, Environment and Management (HNICEM)*, pages 1–6. IEEE, 2018.

[10] Khang Nguyen Tan Tran Minh, Nguyen Dang Thi Thanh, Khon Trieu Trang, and Nuong Tran Thi Hue. Using tabu search for solving a high school timetabling problem. In *Advances in intelligent information and database systems*, pages 305–313. Springer, 2010.

[11] Tyler Morrow, Ali R Hurson, and Sahra Sedigh Sarvestani. A multi-stage approach to personalized course selection and scheduling. In *2017 IEEE International Conference on Information Reuse and Integration (IRI)*, pages 253–262. IEEE, 2017.

[12] Tyler Morrow, Sahra Sedigh Sarvestani, and Ali R Hurson. Algorithmic decision support for personalized education. In *2016 IEEE 17th International Conference on Information Reuse and Integration (IRI)*, pages 188–197. IEEE, 2016.

[13] Clemens Nothegger, Alfred Mayer, Andreas Chwatal, and Günther R. Raidl. Solving the post enrolment course timetabling problem by ant colony optimization. 194(1):325–339.

[14] A Schaerf. A survey of automated timetabling. page 42.

[15] A. Srisamutr, T. Raruaysong, and V. Mettanant. A course planning application for undergraduate students using genetic algorithm. In *2018 Seventh ICT International Student Project Conference (ICT-ISPC)*, pages 1–5, 2018.

[16] Robert Swanson. Four Year Plan Evaluation System, 12 2021.

[17] Robert Swanson. Prototype: Preference Specification Language, 5 2021.

[18] Irving van Heuven van Staereling. School timetabling in theory and practice. Technical report, Technical report, VU University, Amsterdam, Holland, 2012.

A Appendix

A.1 PSL Grammar Specification

```
start: (block)+ EOF;
block: NAME priorityList? '{' specification+ '}';

specification:
    requirementSpecification |
    preferenceSpecification |
    specificationList |
    conditionalSpecification |
    contextualSpecification;

requirementSpecification: REQUIRE NOT? requirableConstraint DOT;
preferenceSpecification: PREFER NAME? NOT? constraint DOT;
specificationList: ('{' specification* '}');
conditionalSpecification: IF condition THEN specification (OTHERWISE_IF condition THEN
↪  specification)* (OTHERWISE specification)?;
contextualSpecification: FOR (contextLevel WHERE condition | termYearList | weekdayList)
↪  specification;

condition: requirableConstraint |
    OPEN_PAREN condition CLOSE_PAREN |
    OPEN_PAREN condition AND condition CLOSE_PAREN |
    OPEN_PAREN condition OR condition CLOSE_PAREN |
    NOT condition ;

requirableConstraint:
    equalConstraint |
    greaterThanConstraint |
    greaterThanOrEqualConstraint |
    lessThanConstraint |
    lessThanOrEqualConstraint |
    booleanConstraint;
equalConstraint: (INT numericEvaluator) | (timeEvaluators AT time) | (termYearEvaluators IN
↪  termYear);
booleanConstraint: booleanEvaluators;
constraint: requirableConstraint | moreConstraint | lessConstraint;
moreConstraint: MORE_ numericEvaluator | timeEvaluators LATER | termYearEvaluators LATER;

numericEvaluator:
    totalCredits | totalCreditsFromSet | upperDivisionCredits | totalCourses |
    totalCoursesFromSet | upperDivisionCourses | meetingMinutes |
    numCoursesWithProfessor | numTimeBlocks | termsInPlan ;
termYearEvaluators: courseTermYear | planStart | planEnd;
timeEvaluators: dayStarting | dayEnding | courseStart | courseEnd;
booleanEvaluators: meetingAtTimeRange | courseBeforeCourse | coursesInSameTerm | termExists;
```

Listing 6: A Selection of the PSL ANTLR Grammar