Assignment of bachelor's thesis

Title:	Improving LearnShell backend for exams and assignments
Student:	Ondřej Cihlář
Supervisor:	Ing. Jakub Žitný
Study program:	Informatics
Branch / specialization:	Information Systems and Management
Department:	Department of Software Engineering
Validity:	until the end of summer semester 2022/2023

Instructions

LearnShell is a modular system for managing and performing exams with programming assignments in scripting languages, especially Shell. LearnShell currently offers basic functionality for creating assignments and exams.

Propose a solution for performance and functionality improvements of assignment and exam modules of the LearnShell backend.

1. Analyze the current architecture of the LearnShell backend and APIs.
2. Propose improvements for assignment and exam modules functionality.
- analyze the time complexity of this task, rabbit-holes, and no-gos
- divide into smaller tasks of a specific type
- add tracking to all the new functions so they can be evaluated in the future
3. Optimize performance of all endpoints in assignment and exam modules.
4. Compare the pricing and pros and cons of 3 major cloud providers where the LearnShell backend modules will be migrated in the future.
5. Compile a report of improvements, future re-evaluation, and internal documentation.

Electronically approved by Ing. David Buchtela, Ph.D. on 8 March 2021 in Prague.

Bachelor's thesis

Improving LearnShell backend for exams and assignments

Ondřej Cihlář

Software Engineering
Supervisor: Ing. Jakub Žitný

May 13, 2021

Acknowledgements

I want to thank my tutor, Ing. Jakub Žitný for guiding my work and his advice. I also want to thank Bc. Karel Jílek for his help with finding the correct solution and Vojtěch Skoumal, DiS, for his constant support during tough times of my study.

Declaration

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis.

I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular that the Czech Technical University in Prague has the right to conclude a license agreement on the utilization of this thesis as a school work under the provisions of Article 60 (1) of the Act.

In Prague on May 13, 2021 . . . .

Czech Technical University in Prague
Faculty of Information Technology
© 2021 Ondřej Cihlář. All rights reserved.
This thesis is school work as defined by Copyright Act of the Czech Republic. It has been submitted at Czech Technical University in Prague, Faculty of Information Technology. The thesis is protected by the Copyright Act and its usage without author's permission is prohibited (with exceptions defined by the Copyright Act).

Citation of this thesis

Cihlář, Ondřej. *Improving LearnShell backend for exams and assignments.* Bachelor's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2021.

Abstrakt

Tato práce se zabývá analýzou backendu systému LearnShell, návrhem vylepšení a jejich následnou implementací. Hlavní navržené vylepšení je systém bodových bonusů a penalizací, který byl implementován formou doménově specifického jazyka. Logiku udělování bodových bonusů a penalizací lze naprogramovat pomocí tohoto jazyka. V této práci také analyzuji ceny vybraných poskytovatelů cloudových služeb, kam bude LearnShell v budoucnu migrovat.

Klíčová slova doménově specifické jazyky, gramatika, lexikální analýza, syntaktická analýza, interpreter, optimalizace, ceny poskytovatelů cloudových služeb

Abstract

This work deals with analysis of backend of LearnShell system, proposal of improvements and their implementation. The main proposed improvement is a score bonuses and penalties system, which was developed by implementation of a domain-specific language. The logic of applying of bonuses and penalties can be programmed for by using this language. In this work, I also analyse pricing of some major cloud providers, where the LearnShell will migrate in the future.

Keywords domain specific language, grammar, lexical analysis, syntactic analysis, interpreter, optimization, cloud providers pricing

Contents

Introduction **1**

1 Current architecture analysis **3**
1.1 Technologies . . . 3
1.1.1 Django framework . . . 3
1.1.2 GraphQL . . . 4
1.1.3 PostgreSQL . . . 4
1.1.4 Celery and Redis . . . 4
1.2 Architecture . . . 5
1.2.1 Business processes . . . 5
1.2.2 Modules . . . 6
1.2.3 Generator and evaluator services . . . 6
1.2.4 Database model . . . 7
1.2.5 API . . . 15

2 Improvements proposal **21**
2.1 Functional requirements . . . 21
2.2 Providing of hints . . . 22
2.3 Bonus and penalty system – database solution . . . 24
2.3.1 Pros and cons . . . 25
2.4 Bonus and penalty system – DSL solution . . . 25
2.4.1 Domain-specific language . . . 25
2.4.2 Scoring DSL . . . 26
2.4.3 Pros and cons . . . 27

3 Basics of formal grammars theory **29**
3.1 Formal grammar . . . 29
3.2 Syntax tree . . . 30
3.3 Lexical analyzer . . . 32
3.4 Syntactic analyzer . . . 33

3.5 Recursive descent parsing algorithm . . . 33
3.6 Symbol table . . . 33

4 Implementation **35**
4.1 Time complexity analysis . . . 35
4.2 Implementation of the scoring DSL . . . 36
4.2.1 Grammar . . . 36
4.2.2 Lexer . . . 38
4.2.3 Parser . . . 40
4.2.4 Interpreter . . . 42
4.2.5 Integration into LearnShell . . . 42
4.3 Providing of hints implementation . . . 43
4.4 Endpoints optimization . . . 44
4.5 Final report . . . 46

5 Cloud services providers comparison **47**
5.1 IBM Cloud Kubernetes Service . . . 47
5.1.1 Cheap variant . . . 47
5.1.2 Expensive variant . . . 48
5.2 Google Kubernetes Engine . . . 48
5.2.1 Cheap variant . . . 48
5.2.2 Expensive variant . . . 48
5.3 Microsoft Azure Kubernetes Service . . . 48
5.3.1 Cheap variant . . . 48
5.3.2 Expensive variant . . . 48
5.4 Summary . . . 49

Conclusion **51**

Bibliography **53**

A LearnShell git repository **57**

B Acronyms **59**

C Contents of enclosed CD **61**

List of Figures

1.1 Process of creating and writing exam 5
1.2 Package diagram of LearnShell's apps 7
1.3 Database model 8

2.1 Use case and requirements diagram 22
2.2 Hint table in database 23
2.3 EvaluationTemplate table in database 24
2.4 EvaluationTemplate table with DSL script 27

3.1 Syntax tree 31
3.2 Abstract syntax tree 32

List of Tables

1.1 User database table 9
1.2 Job database table 9
1.3 Course database table 10
1.4 Parallel database table 10
1.5 ParallelMembership database table 11
1.6 Service database table 11
1.7 AssignmentTemplate database table 12
1.8 Assignment database table 12
1.9 Generated assignment database table 13
1.10 Submission database table 13
1.11 Correction database table 13
1.12 ExamTemplate database table 14
1.13 AssignmentExamTemplate database table 14
1.14 Exam database table 14
1.15 StudentWritesExam database table 15

5.1 Cloud pricing comparison summary 50

Introduction

LearnShell is a system, which is used at the Faculty of Information Technology, Czech Technical University in Prague. It's a modular system, currently used in the Programming in shell course, for managing and performing exams with programming assignments.

In the past, students of Programming in shell course were tested only on paper. It was not quite a suitable way of testing, since it's a test of student's programming skills, nor was it comfortable for teachers to evaluate. Evaluation of a programming assignment can be complicated. Every computer program usually takes input values, and transforms them into output. In case of some more complex assignment, there can be hundreds of different input values and there are usually many possible ways of implementation. If a teacher wants to evaluate a solution of such assignment properly, it can be impossible without some form of automatic evaluation.

For this reason, the LearnShell was developed, a system which provides fully automatic generation of assignments and its evaluation. Current version provides basic functionality of assignment and exam creation and is still under development.

The aim of this work is to propose improvements of LearnShell's assignment and exam modules, implement it, and optimize all endpoints of this module. The future idea is to migrate the system's modules to a cloud service, so I will also analyze services of major cloud providers, where LearnShell could migrate in the future.

This work is divided into five chapters. In the first chapter, I analyze the current architecture of the LearnShell backend. I write about technologies and frameworks used in current implementation, databases, business processes and system's API. In the second chapter I propose possible improvements of assignment and exam modules, I also analyze time complexity of this task. The third chapter states the theory that is needed for the implementation. Implementation is described in the fourth chapter together with endpoint optimization of the assignment and exam modules. The final chapter deals

with comparison of cloud providers for future migration, where I state pricing analysis of some major providers.

Chapter 1

Current architecture analysis

In this chapter I will analyse the current architecture of the LearnShell backend. Before I start with the detailed analysis, I will briefly describe you basic usage of the system.

LearnShell is a web application, which provides the creation and evaluation of programming assignments and exams. It also keeps the evidence of students, teachers, courses and parallels. Teachers are able to create exams with specific assignments. Once the exam is created, the teacher assigns students within his parallel and starts the exam, which means that the assignments are generated for each student. Thereafter students create their solution and submit it. LearnShell evaluates it and creates a correction, where all evaluation information is stored and students can display it.

1.1 Technologies

1.1.1 Django framework

LearnShell is written in the Django framework. It is a framework for building web applications in Python programming language [1].

Applications written in this framework fulfill the model-view-controller (MVC) architecture. MVC is one of the architectural design patterns – application is divided into three components: model, view and controller. Model manages the data of the application. Controller processes the user input and interacts with data model objects, it is usually responsible for the main application logic. View is responsible for presenting data to the user. [2]

This framework also provides its own object-relational mapper and API for database access. [3]

Each project written in Django consists of apps, which is a submodule of the project. An app does not have to be connected to other apps and can basically work as a standalone Python module. Each app usually focuses on one logical piece of the project. [4]

1.1.2 GraphQL

System uses GraphQL language for the communication between server and client, which is an interesting alternative to widely used REST, originally developed by Facebook. *"GraphQL is a query language for APIs and a runtime for fulfilling those queries with existing data. GraphQL provides a complete description of the data in the API."* [5]

In contrast to REST services, where a client sends requests to endpoints, which return fixed data structure, in GraphQL, a client is able to describe what data exactly needs. It is a big advantage, since the fixed data structure returned by REST service may contain a lot of useless data, or conversely, the data may not be enough. [6]

GraphQL provides two basic types of operations – *query* and *mutation*. *Query* is an operation, which does not have any impact on the server-side data. It is usually used for data fetching – read operations. *Mutation* is an operation usually used for data modifying – create, update, delete and other actions. [7]

1.1.3 PostgreSQL

LearnShell uses PostgreSQL database for data storage. It is an open source object-relational database [8]. PostgreSQL tries to conform with the SQL standard and has a good reputation for its performance, reliability, security and extensibility [9].

1.1.4 Celery and Redis

There are tasks in the LearnShell that are done asynchronously. It means that the task runs in the background, so the server can continue doing some other work. Such tasks are handled by the Celery – distributed task queue. *"Task queues are used as a mechanism to distribute work across threads or machines. A task queue's input is a unit of work called a task. Dedicated worker processes constantly monitor task queues for new work to perform. Celery communicates via messages, usually using a broker to mediate between clients and workers. To initiate a task the client adds a message to the queue, the broker then delivers that message to a worker."* [10]

As a message broker, LearnShell uses the Redis database. Redis is a key-value data structure store, which can be used as a database, cache and message broker. [11]

1.2 Architecture

1.2.1 Business processes

The main business process the LearnShell currently implements is a process of creating and writing an exam. First, a teacher creates all assignments that are supposed to be a part of the exam. All assignments must be tested before they are assigned to an exam. Thereafter the teacher creates the exam, adds created assignments into it and enrolls students. Once the teacher completes the enrollment, the system prepares and generates assignments for all enrolled students. Then the teacher starts the exam and the system makes assignments available to students. Once the exam is started, the system starts to countdown the time limit and students are able to create and submit a solution. If a student submits a solution, the system evaluates it and displays results. If the solution is not correct, the student can recreate it and submit it again. The process ends when the countdown reaches the time limit and the system ends the exam. The following UML activity diagram describes the process of creating and writing exam:

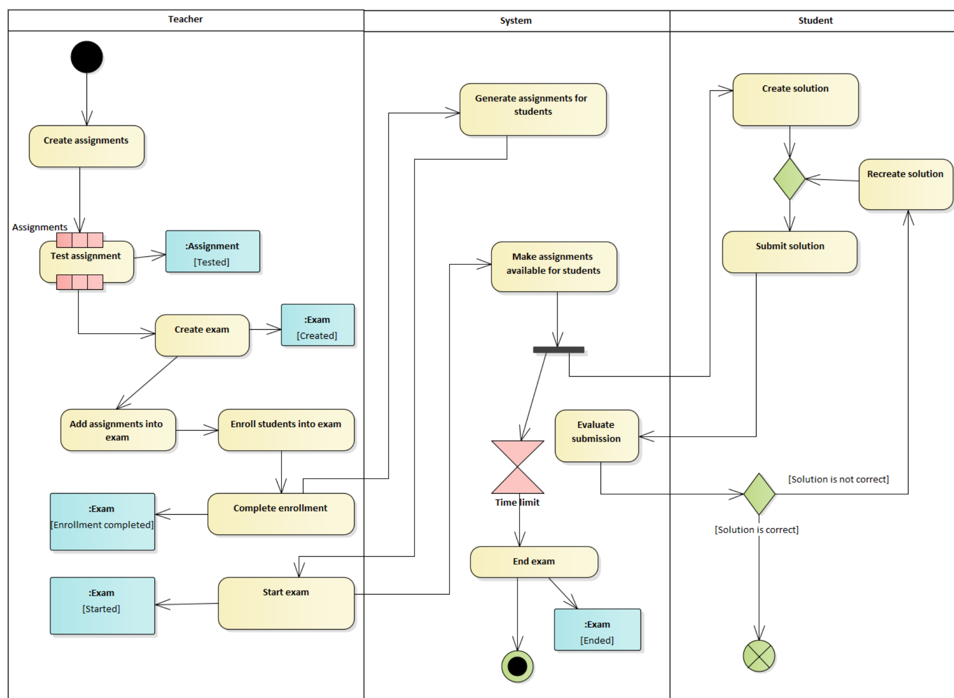

Figure 1.1: Process of creating and writing exam

1.2.2 Modules

Although the Django framework can be used for development of complete web applications, LearnShell uses it for managing the data and for business logic, thus according to the MVC architecture, it uses only the model and controller components. Presentation layer is completely separated from Django – LearnShell has its own frontend, which uses different technologies and programming languages, but this work does not focus on it.

LearnShell consists of the followin Django apps:

- Assignment
- Core
- Course
- Exam
- Generated assignment
- Service
- Submission
- User

Each app has its own data model and business logic. These apps are connected to each other and together they form the whole system.

1.2.3 Generator and evaluator services

For generating and evaluating assignments, LearnShell uses external services – Generator and Evaluator.

Generator is used for customizing an assignment for a particular student. During creation of an assignment, the teacher inputs data for the generator service. Thereafter the generator returns variables that can be used in the assignment.

Evaluator is used for evaluation of a student's submission. During the creation of an assignment, the teacher inputs data for the evaluator service, based on which the evaluation is performed. Once a student creates a submission, LearnShell sends the submission data into the evaluator service, which returns a correction.

External services communicate via an HTTP protocol. Each service must implement the following interface:

- **GET {url}/ping** – returns 200 or 204 code, indicates whether the service is running and can be used

- **GET {url}/schema** – returns a schema of data that the service expects on the input
- **POST {url}** – accepts data fulfilling the schema in JSON format, returns data in JSON format (generated data or correction)

1.2.4 Database model

In this section I am describing LearnShell's database model – all the entities, tables and their fields. The following pictures show a diagram of Learnshell's apps with entities they are composed of and a diagram of the database model:

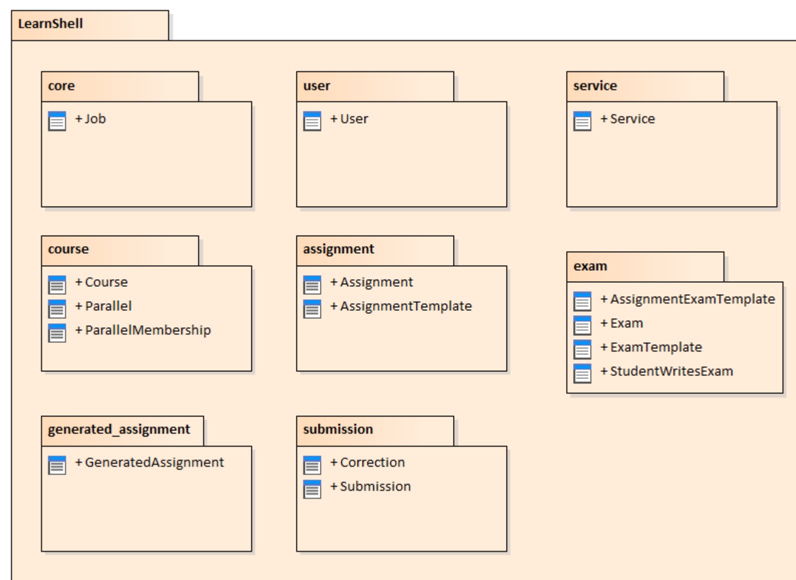

Figure 1.2: Package diagram of LearnShell's apps

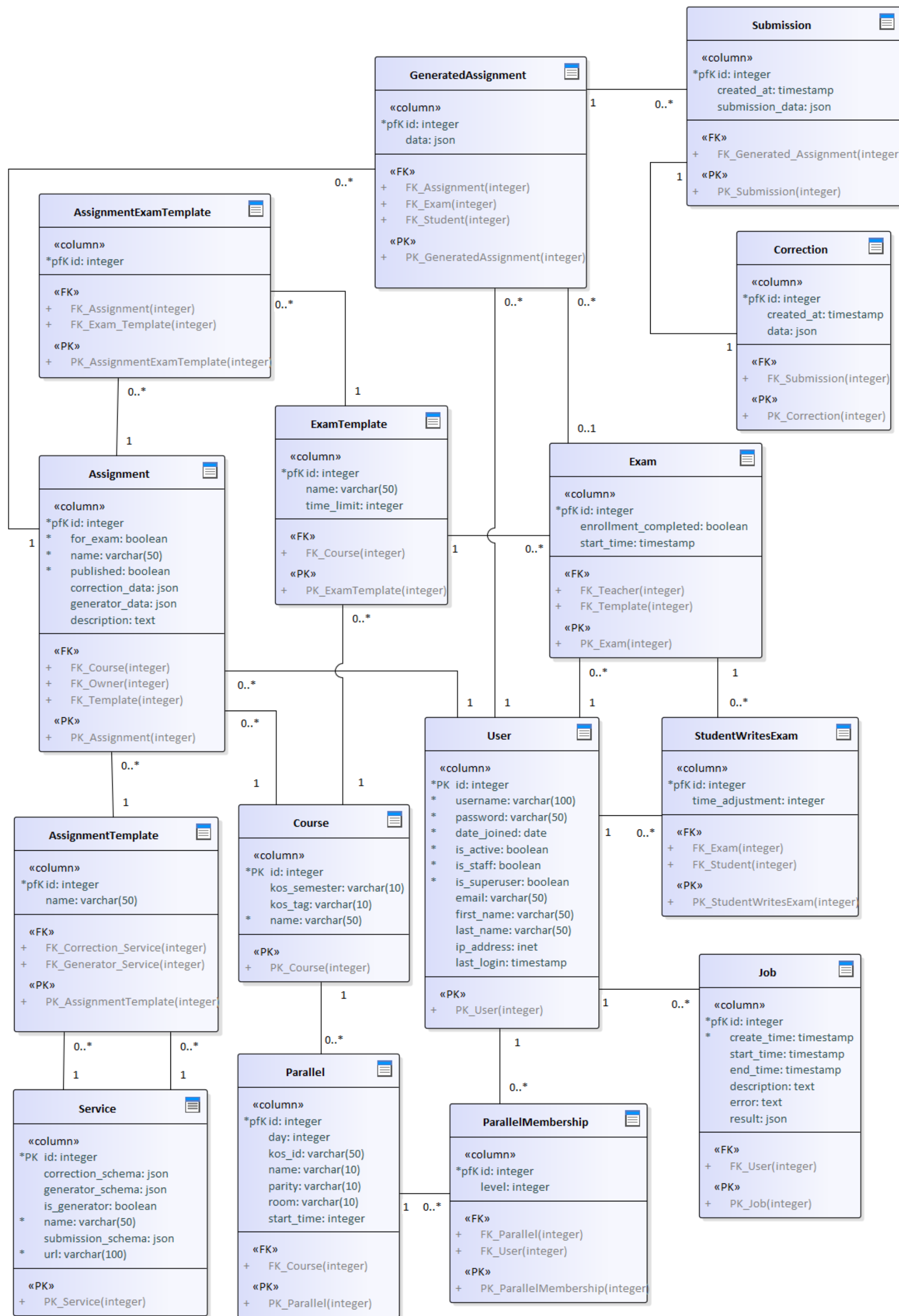

Figure 1.3: Database model

User

Table representing a user. Users are divided into regular users, administrators and superusers.

Table 1.1: User database table

Field	Datatype	Description
id	integer	primary key
username	varchar	username
password	varchar	password
email	varchar	user's email address
first_name	varchar	user's first name
last_name	varchar	user's last name
is_staff	boolean	true if the user is an administrator
is_superuser	boolean	true if the user is a superuser
is_active	boolean	true if the user is active
date_joined	date	date the user joined into system
ip_address	inet	user's ip address
last_login	timestamp	user's last login timestamp

Job

This table represents an asynchronous task, it stores its data and metadata.

Table 1.2: Job database table

Field	Datatype	Description
id	integer	primary key
user	integer	foreign key of user, who started the task
createTime	timestamp	when the task was created
startTime	timestamp	when the task was started
endTime	timestamp	when the task ended
error	text	error output of the task
description	text	textual description of the task
result	JSON	output of the task

Course
Table representing a course. Some fields correspond to course data stored in KOS (a study information system at Czech Technical University in Prague).

Table 1.3: Course database table

Field	Datatype	Description
id	integer	primary key
name	varchar	name of the course
kos_tag	varchar	tag from KOS
kos_semester	varchar	semester from KOS

Parallel
Table representing a parallel. Courses are attended by students and these students are divided into smaller groups – parallels. Parallels are taught separately at a particular time in the assigned room.

Table 1.4: Parallel database table

Field	Datatype	Description
id	integer	primary key
course	foreign key	course the parallel belongs to
name	varchar	name of the course
kos_id	varchar	id of the parallel in KOS
parity	varchar	even or odd week
room	varchar	room the parallel takes place
day	integer	week day the parallel takes place
start_time	integer	hour the parallel starts

ParallelMembership
Table representing a parallel membership. Member of a parallel can be either student – the field *level* is equal to 1, or teacher – *level* is equal to 2.

Table 1.5: ParallelMembership database table

Field	**Datatype**	**Description**
id	integer	primary key
user	foreign key	user
parallel	foreign key	parallel
level	integer	level of user's membership

Service
Table representing an external service – generator or evaluator.

Table 1.6: Service database table

Field	**Datatype**	**Description**
id	integer	primary key
name	varchar	name of the service
url	varchar	url of the service
is_generator	boolean	true if the service is a generator
correction_schema	JSON	JSON schema of the data that are passed to the evaluator by the teacher
generator_schema	JSON	JSON schema of the data that are passed to the generator by the teacher
submission_schema	JSON	JSON schema of the data that are passed to the evaluator by the student

AssignmentTemplate
Table representing an assignment template. It defines what services will be used for evaluation and assignment generation.

Table 1.7: AssignmentTemplate database table

Field	Datatype	Description
id	integer	primary key
name	varchar	name of the template
correction_service	foreign key	service for correction
generator_service	foreign key	service for assignment generation

Assignment
Table representing a specific assignment assigned by the teacher. Each assignment has its owner, it is the only user that can modify it. Assignment can be published for all students (for example as a homework) or it can be assigned to an exam for a specific group of students (usually for one parallel).

Table 1.8: Assignment database table

Field	Datatype	Description
id	integer	primary key
course	foreign key	course the assignment belongs to
owner	foreign key	owner of the assignment
template	foreign key	template of the assignment
name	varchar	name of the assignment
description	text	description of the assignment
correction_data	JSON	data for the correction service
generator_data	JSON	data for the generator service
published	boolean	true if the assignment is published
for_exam	boolean	true if the assignment is for exam

GeneratedAssignment
Table representing an assignment generated for a specific student or for a teacher to test it.

Table 1.9: Generated assignment database table

Field	Datatype	Description
id	integer	primary key
assignment	foreign key	the assignment it was generated from
student	foreign key	the student it was generated for
exam	foreign key	the exam it is part of
data	JSON	data from the assignment generator

Submission
Table representing a submission of the assignment. It stores data known before correction.

Table 1.10: Submission database table

Field	Datatype	Description
id	integer	primary key
generated_assignment	foreign key	generated assignment the submission belong to
created_at	timestamp	when the correction was created
submission_data	JSON	data corresponding to the submission schema

Correction
Table representing a correction. It belongs to one specific submission, so it is related to the submission table by one to one relation.

Table 1.11: Correction database table

Field	Datatype	Description
id	integer	primary key
submission	foreign key	submission the correction belongs to
created_at	timestamp	when the correction was created
data	JSON	data returned from the evaluator

ExamTemplate
Table representing an exam template. It defines a course the exam belongs to and time limit.

Table 1.12: ExamTemplate database table

Field	Datatype	Description
id	integer	primary key
course	foreign key	course the exam template belongs to
name	varchar	name of the template
time_limit	interval	time limit

AssignmentExamTemplate
Table representing a relation between an assignment and an exam template.

Table 1.13: AssignmentExamTemplate database table

Field	Datatype	Description
id	integer	primary key
assignment	foreign key	assignment the template belongs to
exam_template	foreign key	exam template the template belongs to

Exam
Table representing an exam. An exam can be related to multiple assignments, it is written by a specific group of students who are added to the exam by a teacher.

Table 1.14: Exam database table

Field	Datatype	Description
id	integer	primary key
teacher	foreign key	teacher who is in charge of the exam
template	foreign key	template the exam belongs to
start_time	timestamp	when the exam started
enrollment_completed	boolean	true if all students are enrolled to the exam

StudentWritesExam
Table representing a specific student writing a specific exam.

Table 1.15: StudentWritesExam database table

Field	Datatype	Description
id	integer	primary key
student	foreign key	the student writing the exam
exam	foreign key	the exam the student writes
time_adjustment	integer	adjustment of the time limit

1.2.5 API

In this section I describe the LearnShell's API. All actions of this API can be called by the GraphQL. Actions are described for each entity of the LearnShell. I will describe all entity's retrieve fields, which are fields that are not stored in the database table, but the system counts them internally, and all actions that can be called on the entity.

Entities usually have some common actions: list action – lists all entities according to the condition, detail action – returns details of one specific instance of the entity, create action – creates a new instance, delete action – deletes one specific instance, update action – updates one specific instance.

User:

- Retrieve fields:
  - score – User's total score
  - courses – courses the user participates in
  - parallels – parallels the user participates in
  - coursesAsStudent – courses the user participates in as a student
  - coursesAsTeacher – courses the user participates in as a teacher
  - parallelsAsStudent – parallels the user participates in as a student
  - parallelsAsTeacher – parallels the user participates in as a teacher
- Queries:
  - List
  - Detail
  - Myself – returns own profile details

- Mutations:
  - Create
  - Update
  - SetPassword – sets or resets password

Job:

- Retrieve fields:
  - running – is the job running
  - finished – was the job finished
  - duration – duration of running
  - status – RUNNING/FAILED/PENDING/COMPLETED
- Queries:
  - List
  - Detail

Course:

- Retrieve fields:
  - noPrallel – parallel for users without parallel
  - teacher – teachers of the course
  - students – students of the course
  - members – members of the course
- Queries:
  - List
  - Detail
- Mutations:
  - Create
  - Update
  - ImportFromKos – imports course from KOS

Parallel:

- Retrieve fields:
  - teacher – teachers of the parallel

  - students – students of the parallel
  - members – members of the parallel

- Queries:
  - Detail
- Mutations:
  - Create
  - Update

ParallelMembership:

- Queries:
  - List
  - Detail
- Mutations:
  - Create

Service:

- Retrieve fields:
  - valid – is the service valid
- Queries:
  - List
  - Detail
- Mutations:
  - Create

AssignmentTemplate:

- Queries:
  - List
  - Detail
- Mutations:
  - Create

Assignment:

- Retrieve fields:
  - generatorSchema – generator data schema
  - correctionSchema – evaluator data schema
- Queries:
  - List
  - Detail
- Mutations:
  - Create
  - Update
  - Delete
  - ChangeOwner – change owner of the assignment
  - Publish – publishes the assignment
  - Test – generates the assignment to the owner to test it

GeneratedAssignment:

- Retrieve fields:
  - name – name of the generated assignment
- Queries:
  - List
  - Detail

Submission:

- Retrieve fields:
  - submissionSchema – evaluator data schema
- Queries:
  - List
  - Detail
- Mutations:
  - Create
  - Delete

Correction:

- Retrieve fields:
  - score – achieved score
  - generatedAssignment – assignment the correction belongs to

ExamTemplate:

- Retrieve fields:
  - timelimit – time limit of the exam
- Queries:
  - List
  - Detail
- Mutations:
  - Create
  - Update
  - Delete

AssignmentExamTemplate:

- Retrieve fields:
- Queries:
  - List
  - Detail
- Mutations:
  - Create
  - Update
  - Delete

Exam:

- Retrieve fields:
  - time_ limit – time limit of the exam
  - has_started – has the exam started
- Queries:
  - List
  - Detail

- Mutations:
  - Create
  - Delete
  - Start – starts the exam
  - complete_enrollment – prepares and generates assignments for students

StudentWritesExam:

- Retrieve fields:
  - timeadjustment – time adjustment
  - time_left – time left
  - start_time – start time
  - end_time – start time
- Queries:
  - List
  - Detail
- Mutations:
  - Create
  - Delete

Chapter 2

Improvements proposal

The main improvement I am going to implement is a score bonus/penalty system and the logic of providing a hint – this is useful for usage of the score bonus/penalty system.

2.1 Functional requirements

- **Score bonuses and penalties**
  - Teachers are able to define score bonuses/penalties for the whole course and use the UI of the LearnShell to input their parameters. The final score of an assignment generated for a student is affected according to those parameters. Penalties and bonuses are applied, if a student meets the conditions of receiving them.
- **Providing of hints**
  - Teachers are able to define hint types for the whole course. If a student uploads a solution and the solution is not correct, LearnShell creates and provides a simple hint. Student is able to display it.

The following picture shows use case and requirements diagram of proposed improvements:

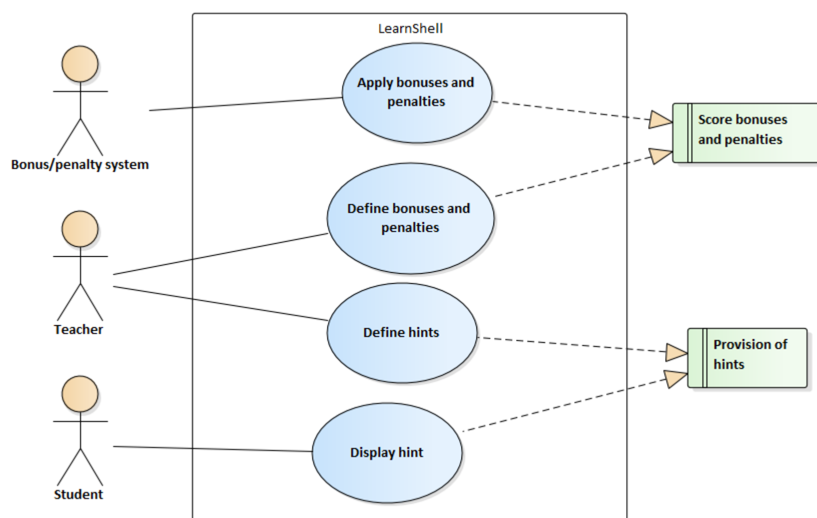

Figure 2.1: Use case and requirements diagram

2.2 Providing of hints

First, I propose the logic of providing of hints, because the bonus/penalty system depends on it and uses this functionality.

Every course has its own evaluator. In the current version of LearnShell, providing of hints is managed only at the level of frontend – after clicking the hint button, a student can see the hint returned by the evaluator. The hint data from the output of the evaluator is not standardized, LearnShell currently uses evaluator of the Programming in shell course, where hints are created from error messages returned in correction.

It is better to manage it completely at the level of the backend in order to track the information about displayed hints for student's particular submissions. All hints that can be displayed should be stored into the database, thus every evaluator should return all hints in some standardized way, so the backend can use them.

My solution to this problem is that the guarantor of every course defines all hint types. When the evaluator returns a result, it also returns an attribute *Hints* within the correction. In this attribute, all hints are available under their names. There is a newly defined table in the database, table representing a hint. This table contains the type of the hint, its text and a boolean value (true or false) – an indicator, whether it was displayed. When a student submits a solution that is not completely correct, the backend receives a correction with available hints that are stored in the *Hints* attribute and creates a table in the database for each of them. When a student sends a request for displaying a specific hint, LearnShell provides it and sets the indicator of displaying in the

corresponding database table to the value of true. Thanks to this solution, all the information about displayed hints can be used by the score bonus/penalties system.

Here is an example for better understanding: student submits a solution and evaluator service finds out that it is not correct. Standard output and return code of the student's solution are different from standard output and return code of reference solution. Evaluator returns a correction with hints. The hint attribute can look something like this:

```
"Hints": {"stdout_mismatch": "Stdout hint text",
          "retcode_mismatch": "Retcode hint text."}
```

The keys "stdout_mismatch" and "retcode_mismatch" are names of the hints. These names can be anything, it is completely up to the course guarantor. In this example, LearnShell creates two hint tables in the database. First table contains "stdout_mismatch" in the *Type* field and "Stdout hint text" in the *Text* field. Second table contains "retcode_mismatch" in the *Type* field and "Retcode hint text" in the *Text* field. At this time, the indicators of displaying are set to the value of false in both tables. Let's say that the student wants to display the *stdout_mismatch* hint. Student sends a request, LearnShell finds the corresponding *Hint* table, sets the indicator of displaying to the value of true and returns the text.

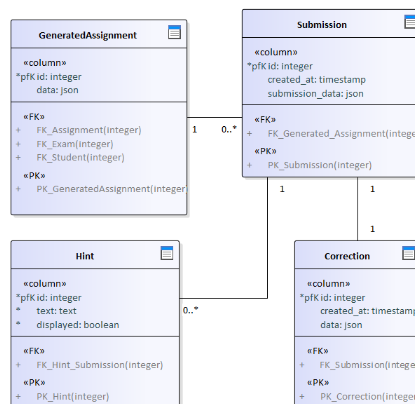

Figure 2.2: Hint table in database

There is a part of the database model diagram in the picture above. This picture shows how this solution looks in the database. The *Hint* table is related to the *Submission* table by many to one relation, so there can be multiple hints created for a submission. Hints are returned in the JSON field *data* of the *Correction* table.

2.3 Bonus and penalty system – database solution

First, I describe some examples of bonuses and penalties, so you have a better idea of how the bonus/penalty system should work. The bonus/penalty system applies a score penalty for displaying a hint. Teacher inputs parameters such as what hints can be displayed, how many hints can be displayed without any penalty and what the score penalty will be for displaying a hint above the limit. Each type of hint can have a different penalty. The number of student's submissions for an assignment is not limited in the current version of LearnShell. This could be also penalised. The parameters of this kind of penalty are the limit of how many submissions are not penalised and what the penalty will be for each submission above the limit. A typical example of a score bonus is an early submission bonus. It is a bonus for submitting a correct solution under a specific deadline of the time limit of the exam. For example, if the correct solution is submitted in the first half of the time limit, the student receives a score bonus.

The parameters of all bonuses and penalties must be stored somewhere in the database, so the system can use them – if a student meets the conditions of some penalty or bonus, the system multiplies the score of his submission by the parameter, which belongs to the particular bonus or penalty. Since all hint types are defined for each evaluator, the parameters should also be defined per evaluator. Possible solution is to add a new table into the database – *EvaluationTemplate* table. There is a possibility that different evaluators can use the same hints and bonus/penalty parameters, so this table is related to the Service table by many to one relation. This table contains the maximum numbers of penalised and free hints and also a JSON field "Schema", which stores all hint types and their penalty value. If a student displays a hint of a specific type, the system looks for the type in this field. Then it can multiply the score by the corresponding value.

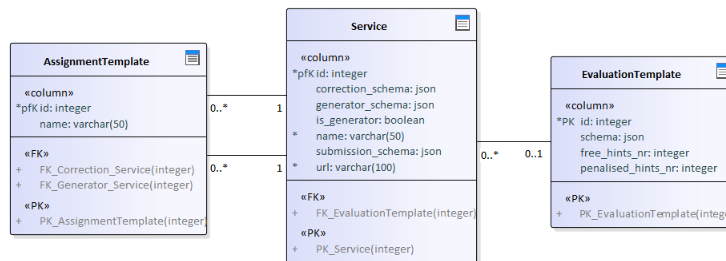

Figure 2.3: EvaluationTemplate table in database

In the example below, the "Schema" field stores two hint types: "stdout_mismatch" and "retcode_mismatch". The student would receive a ten percent penalty for displaying the "stdout_mismatch" hint, because the score would be multiplied by the value of 0.9. In case of displaying the "ret-

code_mismatch" hint, the penalisation would be twenty percent.

```
{
"stdout_mismatch": 0.9,
"stderr_mismatch": 0.8
}
```

2.3.1 Pros and cons

Pros of this solution are that all the score bonuses, penalties and hints are handled automatically. Teacher just inputs the parameters and attributes, which are saved into the database and the whole logic of score counting is already programmed in the LearnShell.

On the other hand, the main disadvantage is that this solution is not completely generic. Every course must settle for this one way of counting the score. Implementation of score bonuses and penalties must be hard-coded within functions of the system's business logic. But what if there are some other kinds of penalties/bonuses? What if the values of bonuses and penalties are not constant and depend on many other factors?

Main purpose of LearnShell is to be usable throughout other courses with different kinds of assignments and generic evaluation settings. Although this solution would work, it is not convenient, because the usability would be very limited.

Although this solution appears to be the easiest way to implement the bonus/penalty system, the maintenance of this solution would probably get complicated. I decided to introduce this inconvenient solution in this work because it gives me arguments to propose a much better solution.

2.4 Bonus and penalty system – DSL solution

Another solution of the bonus/penalty system is to implement a domain-specific language (DSL). Before I start to discuss this solution, I will explain, what exactly a domain-specific language is.

2.4.1 Domain-specific language

"Domain-specific language: a computer programming language of limited expressiveness focused on a particular domain." Humans can use a DSL for writing instructions that a computer understands in some way and executes them. DSLs are usually easier to learn and to understand than general-purpose programming languages. Unlike general-purpose programming languages that are determined to solve complex problems and build whole systems, domain-specific languages are usually focused on a small domain. DSLs are usually

integrated into software systems to solve particular problems it is designed for. [12].

According to [12], domain-specific languages can be divided into three main categories – external DSLs, internal DSLs and language workbenches. An external DSL is completely separated from the language of its parent system and has its own syntax. *"A script in an external DSL will usually be parsed by a code in the host application using text parsing techniques."* An internal DSL is defined in the hosting general-purpose programming language and uses it in a particular way. So a code in an internal DSL is a valid code in its hosting language, *"but only uses a subset of the language's features in a particular style to handle one small aspect of the overall system. The result should have the feel of a custom language, rather than its host language."* A language workbench is an IDE for designing DSLs. It usually provides tools for designing structure of a DSL, building a DSL and an environment for writing DSL scripts.

I will give a few examples of well known DSLs:

- SQL - a DSL for managing data in a relational database
- HTML - a DSL for building websites
- CSS - a DSL for styling websites
- LaTex - a DSL for writing formatted text documents

2.4.2 Scoring DSL

The main idea of this solution is that the final score of submissions is counted by a DSL, which is tailored to the LearnShell. Teachers are able to program the whole logic of bonuses and penalties from the LearnShell's UI by using the scoring DSL. The evaluation by the scoring DSL is also defined per evaluator – the source code of the DSL is a part of the EvaluationTemplate table unlike the previous solution where the table stored only the values of bonuses/penalties.

When a student submits a solution, the evaluator service returns a score as a part of the correction. All the information about the submission, such as student's username, timestamp of submitting, hints already displayed, and other relevant data together with the score, is collected into a data package and sent into the scoring DSL. All the data is available in the scoring DSL and can be used for the evaluation in the evaluation script. The scoring DSL modifies the score returned in the correction according to the script and returns a new score value.

The picture below shows how this solution looks in the database. The EvaluationTemplate table contains only the "id" field and the "evaluation_code" field. LearnShell contains a score counting function within the business logic. Once this function is called, the system finds the script of the score DSL in

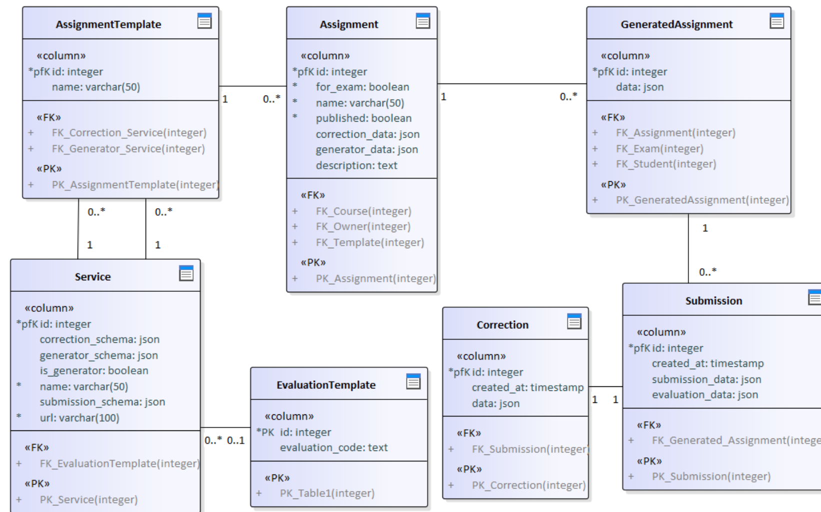

Figure 2.4: EvaluationTemplate table with DSL script

the EvaluationTemplate table, which belongs to the corresponding Service table. Thereafter the system sends the script and the submission data to the function, which executes the interpreter of the scoring DSL and returns the final score.

2.4.3 Pros and cons

The main advantage of this solution is that the evaluation is completely generic. Every course can have its own score calculation, programmed by using the scoring DSL.

Disadvantage is that users have to learn this DSL to be able to use the evaluation system properly. But it is a negligible problem since this solution offers a completely generic design.

It turns out that this solution fits right for the LearnShell, thus I will analyse it in detail and implement it.

Chapter **3**

Basics of formal grammars theory

In this chapter, I state the theory that is needed for implementation of the scoring DSL.

3.1 Formal grammar

In order to describe the scoring DSL formally, I need to define a grammar. Grammar is an important tool to define the syntax of a language. *"A grammar consists of a list of production rules, where each rule has a term and a statement of how it gets broken down."* A rule can also mention some other rules thus together they create the exact description of language's syntax. The source code of a language is a stream of text and grammar gives it an unambiguous syntax, but not a semantics, it does not tell us what the text means. Consider the following example of a grammar rule: *addition := number '+' number*. If there is a part of the language's source code in this form – for example *1 + 1*, the grammar recognizes it as the addition rule. The grammar given in the example would also contain a "number" rule, which would break down to the actual number. [13]

In this work, I only use a context-free grammar. Definition is the following [14]: Context-free grammar is a quadruple:

$$G = (V, T, P, S)$$

- V is *"a finite set of variables, also called sometimes nonterminals. Each variable represents a language, i.e., a set of strings."*
- T is a *"finite set of symbols that form the strings of the language being defined. We call this alphabet the terminals, or terminal symbols."*

- P is a *"finite set of productions or rules that represent the recursive definition of a language."*
- S is a start symbol and *"represents the language being defined. Other variables represent auxiliary classes of strings that are used to help define the language of the start symbol."*

Each production rule consists of *"a variable that is being (partially) defined by the production,"* on its left side. *"A string of zero or more terminals and variables"* on its right side, *"this string, called the body of the production, represents one way to form strings in the language of the variable of the head. In so doing, we leave terminals unchanged and substitute for each variable of the body any string that is known to be in language of that variable."* Both sides are separated by the production symbol. [14]

The following example shows a grammar, which describes a language of addition or deduction statement of two one-digit numbers:

```
G = (V, T, P, S)
V = {statement, addition, deduction, number}
T = {+, -, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9}
S = statement
P = {
 statement := addition | deduction
 addition := number '+' number
 deduction := number '-' number
 number := '0' | '1' | '2' | '3' | '4' | '5' |
           '6' | '7' | '8' | '9'
}
```

The symbol of ":=" is a production symbol terminating the left and the right side of production rules. The "|" symbol represents an alternative – for example the *statement* rule can either break down to the *addition* nonterminal or the *deduction* nonterminal. For a better clarity, the terminal symbols are written with the apostrophes.

3.2 Syntax tree

A script of a language is represented by a stream of text. Another way to represent a script is a hierarchy called a syntax tree. A grammar basically defines how a script is transformed into a syntax tree. [13]

Every node of a syntax tree refers to its child nodes according to the rules of the corresponding grammar. Consider a grammar with the following rules (the symbol of two dots at the body of the *number* rule means an interval – a number can be any from the interval of 0-9):

```
function := name '(' argument ')'
argument := number | addition
addition := number '+' number
name := 'sqrt' | 'pow'
number := '0'..'9'
```

Let's also consider the following piece of script:

```
sqrt (6 + 3)
```

The syntax tree of this script looks like this:

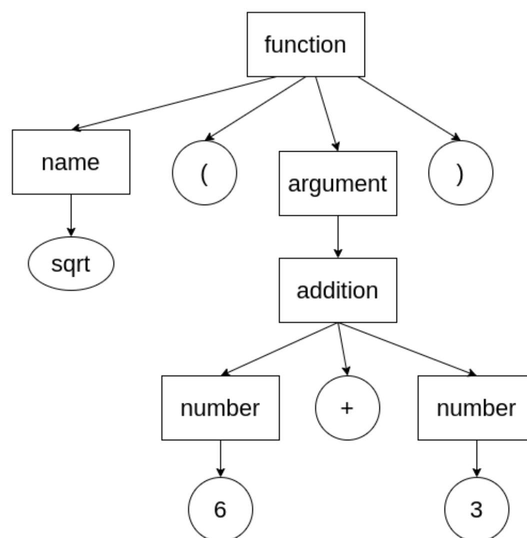

Figure 3.1: Syntax tree

During the implementation I will not need to use the actual syntax tree. Instead, I will use an abstract syntax tree (AST). *"An abstract syntax tree is a simplification of the syntax tree which provides better tree representation of the input language."* Nodes of a syntax tree represent the exact input text. An abstract syntax tree doesn't necessarily have to represent the exact input, it is reduced of unnecessary nodes that are not needed for further processing of the tree. [15]

If we consider the previous example of a syntax tree, the nodes representing parentheses and the symbol of plus are redundant. The picture below shows an AST created from the previous syntax tree:

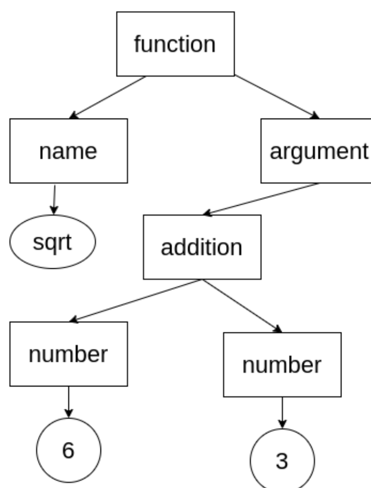

Figure 3.2: Abstract syntax tree

3.3 Lexical analyzer

A lexical analyzer, also called lexer or tokenizer *"is the first stage in processing the input text. The lexer splits the characters of the input into tokens, which represent more reasonable chunks of the input."* Tokens are objects that contain individual elements of the language which are located in the input text. Each token is basically composed of two main attributes – token type and value. Token types are usually for punctuation – keywords, operators, parentheses and separators, for domain text – identifiers, names, string literals, number literals and for whitespaces – newlines, spaces, tabs. Whitespaces are usually discarded, unless they are syntactically significant. A lexer has an internally defined rule for each token type. During the processing of the input text, the rules are applied in the order of precedence (for example a keyword token type has usually higher precedence than an identifier token type). Once the lexer finds the matching rule, it creates a token of the corresponding type. The value of the token is usually the input text that was matched. Values of punctuation token types are usually not important. Tokens don't have to be composed only of the type and the value, it can also contain some debug information, such as the position in the input text. [16]

Let's consider the example of grammar from the previous chapter and the following input script:

```
sqrt(6 + 3)
```

The lexical analyzer would transform the input into the following tokens (value is not defined where it is redundant):

```
[type: identifier, value: sqrt]
[type: left_parenthesis]
[type: number, value: 6]
[type: plus]
[type: number, value: 3]
[type: right_parenthesis]
```

3.4 Syntactic analyzer

A syntactic analyzer, also called a parser, performs the process of syntactic analysis. *"Syntactic analysis takes the stream of tokens and arranges them into a parse tree"* (syntax tree). The process of parsing ends successfully, if the given input text matches the given grammar (parser recognizes the input), if not, it ends with failure. [16]

Thereafter the syntax tree, which is built during the parsing process, can be interpreted by a tree walking code, such code evaluates the branches of the tree and produces the final output.

3.5 Recursive descent parsing algorithm

Recursive descent parsing algorithm is one of the simplest ways to implement a parser. The idea is, that there is a procedure for each nonterminal symbol of the grammar. *The goal of each such procedure is to read a sequence of input characters that can be generated by the corresponding non-terminal, and return a pointer to the root of the parse tree for the non-terminal. The structure of the procedure is dictated by the productions for the corresponding non-terminal.* The procedure processes the input and tries to match the right side of some production. In case of a terminal symbol, the procedure compares the symbol it is looking for to the input symbol, if they match, it simply moves to the next symbol and indicates success. In case of a nonterminal symbol, the procedure executes the procedure that is determined to handle that symbol, if the input matches the right side of the production being handled, it returns success. The procedures can call each other recursively. [17]

3.6 Symbol table

A symbol table is *"a location to store all identifiable objects during a parse to resolve references."* It maps the reference of stored object to its value, so the values can be referenced in the script. A symbol table is usually a map

data structure, it means that it is composed of keys and values. A key is the reference to the value, usually it is a string (but not necessary). [18]

For example, there is a name of a variable used in the script, the symbol table stores it under the key equal to the name, thus the value of the variable is available under the variable's name.

Chapter 4

Implementation

The whole LearnShell project is a part of the enclosed CD. In the appendix, I also enclosed a link for my branch of the LearnShell repository on GitHub.

4.1 Time complexity analysis

In this section, I am describing time complexity of implementation of proposed improvements. During this analysis, I am following the method of decomposition. It means that big problems are divided into smaller tasks of a specific type and time complexity is estimated for each task. Thereafter the final time complexity is counted as a sum of the complexity of each task.

Time complexity is typically measured in mandays (MD). Manday is a unit which corresponds to one person's working day. It is typically considered as 8 hours. [19]

Implementation of the language consists of intro analysis, where I will analyze the form of the language, what elements should be supported by the language and write down some possible pieces of script. Then I will create the grammar, according to which the lexical analyzer and the syntactic analyzer will be implemented. The next stage of the language implementation is the creation of an interpreter. The final stage is to integrate the language into LearnShell, so the final score of a student will be counted according to the script of the language.

The logic of providing of hints has a simpler implementation. It only requires to create a new table in the database - the *Hint* table, and implementation of two functions – one that is responsible for creating all hints from the output of the evaluator and one for providing a hint when a student sends a request for displaying it.

The estimate of time complexity is the following:

- Scoring DSL
  - Intro analysis – 2 MD
  - Grammar – 1 MD
  - Lexical analyzer – 2 MD
  - Syntactic analyzer – 4 MD
  - Interpreter – 8 MD
  - Reserve for bugs fixing – 2 MD
  - Integration into the system – 0.5 MD
- Providing of hints
  - Database table creation + implementation of the logic – 0.5 MD
- **Total estimated – 20 MD**

4.2 Implementation of the scoring DSL

There is a *scoring_dsl* python package I created in the LearnShell project for the implementation. Documentation is stored in the *scoring_dsl_docs* folder. The *scoring_dsl* package also contains the *readme.md* file, where the user manual of the scoring DSL is described.

4.2.1 Grammar

Let's discuss what the language should look like. The scoring DSL's purpose is to modify the score according to received bonuses and penalties. Simultaneously it should give users the ability to process a data package with information about the submission and use it for calculation of the final score.

Score modification is applied if a student meets the conditions that are defined for the specific bonus or penalty. It means that there is a set of conditions defined for each bonus or penalty. Users should be able to describe these conditions by the language, which uses received data to evaluate them. In case of a fulfilled condition, the language executes defined actions – mostly the score modification. Since the bonus/penalty system is always a set of conditions, the cornerstone of the language should be if statement. If statement allows a programmer to define a condition to the computer and commands it should execute in case of a fulfilled condition.

It is also necessary to ensure the ability of processing data the scoring DSL receives for evaluation. This can be done by provision of built-in functions and variables. Variables are used for data accessing, functions are able to accept

them as arguments, read them and eventually execute commands and return results. In order to work with penalties, bonuses, values of variables and results returned by functions, the language should also support the number and the string data types.

According to this analysis, the grammar I proposed for the scoring DSL looks the following way:

```
G = (V, T, P, S)
V = {
 program, if_stmt, function, argument, identifier,
 number, string
}
T = ASCII characters
S = program
P = {
 program := (if_stmt)*

 if_stmt := 'if' function 'then' function ( ';' function )* 'fi'

 function := identifier '(' (argument (',' argument)*)? ')'

 argument := function | identifier | number | string

 identifier := [ 'a'..'z' 'A'..'Z' '_' ]+

 number := '-'? [ '0'..'9' ]+ |
           '-'? [ '0'..'9' ]+ '.' [ '0'..'9' ]+

 string := '"' [ ASCII character ]* '"'
}
```

First, I explain the notation used for the grammar rules. Symbols stored in between parentheses form a subgroup. The symbol of "∗" indicates that the previous symbol or subgroup can appear zero or more times. The symbol of "+" indicates that the previous symbol or subgroup can appear one or more times. Square brackets represent a list of symbols, the symbol at this place can be any from the symbols stored in the list.

The language consists of zero or more if statements. I decided that the language can be empty, in case there is a course that will not use it. If statement is composed of two parts – condition and actions. Condition is stated after the "if" keyword and is represented by a function. Actions are stated after the "then" keyword. They are also represented by functions and there can be one or more actions at each if statement. Action functions are sep-

arated by semicolons. The whole if statement is ended by the "fi" keyword. Function consists of an identifier, which represents the name of the function and arguments. Arguments are stored in between parentheses, separated by commas and there can be zero or more of them. An argument can be a function, identifier, number or string. In case of argument, an identifier represents the name of a variable. Identifier is composed of one or more characters of lower/upper alphabet and underscore. A number can be an integer or a floating point number. String is composed of zero or more ASCII characters, which are stored in between quotation marks.

4.2.2 Lexer

The implementation of the lexer is stored in the *lexer.py* module of the *scoring_dsl* package.

The first stage of implementing the lexer is the identification of all token types, which create an output of the lexer. According to the grammar, I identified the following token types:

- Keyword
- Identifier
- Number
- String
- Comma
- Semicolon
- Left parenthesis
- Right parenthesis
- EOF

There are the following keywords in the language – *if, then* and *fi*, so the value of the *keyword* token must be from this list. Values of *identifier, number* and *string* tokens correspond to the values stated in the grammar. Values of other token types are not relevant. The *EOF* token is appended at the end of the output stream of tokens, so the parser recognizes the end of the stream.

Token is represented by the *Token* class. This class stores a token type, its value and provides methods for type and value matching.

The lexer is represented by the *Lexer* class. This class accepts a string of the script of the scoring DSL in its constructor. The *Lexer* class also stores a list of tokens, where the tokens are added during the lexical analysis, and the information about the current position in the input text. The class provides the *run* method which is responsible for the whole process of lexical analysis. The logic of this method is expressed by the following pseudocode:

```
tokens = []; position = 0;

while (position <= input length):
  if (input[position] is whitespace):
    increase position by one; continue;

  if (keyword_found()):
    append keyword to tokens; adjust position; continue;

  else if (identifier_found()):
    append identifier to tokens; adjust position; continue;

  else if (number_found()):
    append number to tokens; adjust position; continue;

  else if (string_found()):
    append string to tokens; adjust position; continue;

  else if (comma_found()):
    append comma to tokens; adjust position; continue;

  else if (semicolon_found()):
    append number to tokens; adjust position; continue;

  else if (left_parenthesis_found()):
    append parenthesis to tokens; adjust position; continue;

  else if (right_parenthesis_found()):
    append parenthesis to tokens; adjust position; continue;

  else raise "Unrecognized symbol error";
```

As stated in the lexical analysis chapter, the rules of finding a token type in the input text are applied in the order of the precedence. Each rule processes the input text and if it finds an element it is responsible for, it ends with success. In my case, the highest precedence has the rule for finding a keyword. Next in order is the rule for finding an identifier. The order of other rules is not important because the format of the found elements cannot be confused with each other.

4.2.3 Parser

The implementation of the parser is stored in the *parser.py* module. This module contains the *Parser* class, the *AST* class, which is an interface representing an abstract syntax tree node, and the classes implementing the *AST* interface. There is an AST node for each nonterminal symbol of the grammar, except for the *argument* nonterminal. The *argument* rule is stated in the grammar for better clarity, but the parser does not really implement it during the process of syntactic analysis, the right side of the *argument* rule is directly set in place of the *argument* nonterminal on the right side of the *function* rule.

The *Parser* class is responsible for syntactic analysis, it implements the recursive descent parsing algorithm. This class accepts an array of tokens created by the *Lexer* class and according to the recursive descent parsing algorithm, it provides a method for each nonterminal symbol of the grammar. Syntactic analysis is executed by running the *program* method, an output of this method is a root node of the AST. The following pseudocode expresses the logic of the *Parser* class methods:

```
program():
  if_statements = [];

  while (True):
    if (next token is EOF): break;

    append if_statement() to if_statements;

  return ProgramNode(if_statements);

identifier():
  if (next token is not 'identifier'): raise "Error";

  return IdentifierNode(token value)

number():
  if (next token is not 'number'): raise "Error";

  return NumberNode(token value)

string():
  if (next token is not 'string'): raise "Error";

  return StringNode(token value)
```

```
if_statement():
  then_functions = [];

  if (next token is not 'if' keyword): raise "Error";

  condition = function();

  if (next token is not 'then' keyword): raise "Error";

  while (True):
    append function() to then_functions;

    if (next token is 'fi' keyword): break;
    else if (next token is not 'semicolon'): raise "Error";

  return IfNode(condition, then_functions)

function():
  argument_list = [];
  if (next token is not 'identifier'): raise "Error";

  name = token value;

  if (next token is not 'left parenthesis'): raise "Error";

  while (True):
    try:
      append function() to argument_list;
    except:
      try:
        append identifier() to argument_list;
      except:
        try:
          append number() to argument_list;
        except:
          try:
            append string() to argument_list;
          except:
            pass;

    if (next token is 'right parenthesis'): break;
    else if (next token is not 'comma'): raise "Error";

  return FunctionNode(name, argument_list)
```

4.2.4 Interpreter

The interpreter is represented by the *Interpreter* class stored in the *interpreter.py* module. Its constructor accepts a string of the language's script, an array of hint types displayed by a student and a dictionary of built in variables and their values. All the hint types and variables are stored into the symbol table. The interpreter performs a tree-walking code, during which all actions are executed and returns the final score. The *AST* class provides the *evaluate* method, thus every class which represents an AST node implements it. This method executes the *evaluate* method of all child nodes and returns an object of *DataType* class. The *DataType* class is stored in the *data_types.py* module. It is an interface which provides the *execute* method. Each *AST node* class has a corresponding *Data type* class, which implements the *DataType* interface and stores data types of its child nodes.

The *execute* method performs the interpreting – it evaluates stored instances, works with the symbol table and executes actions (evaluation of if statements, functions, numbers and variables). *Number* and *String* is evaluated to the real Python data type (float and string). *Identifier* represents the name of a built-in variable, so identifier is evaluated to the value of the corresponding key in the symbol table. *If statement* evaluates the condition, if it is evaluated to the value of true, the action functions are executed. *Function* evaluates its parameters to real values and executes its action. Actions of built-in functions are handled by the *FunctionHandler* class stored in the *function_handler.py* module. This class provides the *function* method, which accepts function's name and arguments in parameters, and finds the corresponding handler.

The *Interpreter* class provides the *run* method. This method executes the lexical analysis, syntactic analysis, tree-walking code and returns the final score. The symbol table is represented by the *SymbolTable* class.

There is also the *ScoringDsl* class in the *scoring_dsl.py* module, this class accepts data sent by the LearnShell in the constructor and runs the interpreter.

4.2.5 Integration into LearnShell

Score is recalculated for each submission of a student. I created the *score* field in the *submission* table, where the score of the submission is stored. The scoring DSL is applied in the *submit_worker* function in the *utils.py* file of the LearnShell's *submission* app. This function sends the submission into the evaluator service and thereafter saves received correction. The scoring DSL is applied after receiving the correction, it sends the score of the correction into the constructor of the *ScoringDSL* class, together with the student's username and displayed hints. In case the assignment is a part of an exam, the exam's time limit and duration of the submission are also sent into the scoring DSL, otherwise these values are set to zero.

The following Python code shows part of the *submit_worker* function, where the scoring DSL is applied:

```
# source code of the scoring DSL
code = submission.generated_assignment.assignment
       .template.correction_service.evaluation_template
       .scoring_dsl_src

total_time = 0.0
time_limit = 0.0

if submission.generated_assignment.is_exam_assignment:
    start_time = submission.generated_assignment.exam.start_time
    submission_time = submission.created_at
    time_limit = submission.generated_assignment.exam.time_limit
    total_time = (submission_time - start_time).total_seconds()

scoring_dsl = ScoringDsl(code, hints_displayed,
    {"username": user.username, "score": c.score,
     "total_time": total_time, "time_limit": time_limit})
res, err = scoring_dsl.run()
if err == "":
    submission.score = res
    submission.save()
else:
    raise Exception("Error during score counting: " + err)
```

4.3 Providing of hints implementation

The implementation corresponds to the proposal stated in the improvements proposal chapter. Each hint belongs to a submission, thus the *Hint* table is implemented in the *models.py* file of the *submission* app. The following Python code shows the implementation of the table:

```
class Hint(Model):
    submission = ForeignKey(Submission, on_delete=CASCADE,
        verbose_name=_("Hint of"), related_name="hints",
        blank=True, null=True)
    type = CharField(max_length=50, verbose_name=_("Hint type"))
    text = TextField(verbose_name=_("Text"))
    displayed = BooleanField(default=False,
        verbose_name="Displayed?")
```

Service functions - *make_hints* and *display_hint* are implemented in the *utils.py* file of the *submission* app. The *make_hints* function is called in the

submit_worker function, right after receiving a *Correction* from the evaluator service. It creates all *Hint* tables in the database according to the *Hints* attribute stored in the *Correction*. The implementation is stated in the following code:

```
def make_hints(correction, submission):
    hints = correction.data["hints"]

    for key, value in hints.items():
        h = Hint(submission=submission, type=key, text=value)
        h.save()
```

The *display_hint* function can be called through the GraphQL API, it makes particular hint data available – it sets the *displayed* field to the value of true and returns a text of the hint. The implementation is stated in the following code:

```
def display_hint(request, instance, data):
    instance.displayed = True
    instance.save()

    event = EventsStat(user=request.user,
                       event_name="hint_display",
                       data={"type": instance.type,
                             "text": instance.text})
    event.save()

    return {"text": instance.text}
```

In order to track whether this new functionality is used by users, I created a new database table – the *EventsStat* table. This table can be used to track any possible functionality, not just an event of displaying a hint. It stores a foreign key field of the user who called the function, name of the tracked event, date and time of its creation and additional data about the event.

4.4 Endpoints optimization

The point of the optimization is to reduce a number of SQL queries executed by the system to retrieve tables from the database. All listing actions of the assignment/exam module were implemented in the following form:

```
    Entity.objects.all()
```

This code performs an SQL query to retrieve all *Entities* from the database. The problem of this code is that in case of a request for some foreign key object of the Entity, another query is executed to retrieve this object. I optimized all listing operations in the assignment/exam module by using the *select_related* function. This function is provided by the Django framework, it accepts names of foreign key fields in parameters. Thanks to this function, a foreign key object is retrieved during the listing action, so Django does not execute another SQL query in case of a request for that object. The form of optimized listing actions I implemented is stated in the following code:

```
Entity.objects.select_related('foreign_keys').all()
```

The *generated_assignment* class, which represents Generated assignment object contains the *score* function. It counts the final score of all submissions. This function was also ineffective, the following code shows the old implementation:

```
def score(self):
    scores = []
    for submission in self.submissions.all():
        try:
            scores.append(submission.score)
        except:
            scores.append(0)
    if len(scores) == 0:
        return 0
    return max(scores)
```

During the listing of submissions, executed by this code, Django executes queries that retrieve all fields of the *submission* table. But in this case, only the *score* field is needed, so I optimized the *score* function by using the *values_list* function. This function retrieves only the fields stated in its parameters. The following code shows the optimized *score* function:

```
def score(self):
    scores = self.submissions.values_list('score', flat=True)
    if len(scores) == 0:
        return 0
    return max(scores)
```

4.5 Final report

I implemented the logic of providing of hints. For each submission of a student, all possible hints are saved into the database. If a student wants to display a hint, he sends a request, the system indicates that the particular hint was displayed and provides text of the hint.

The main part of the implementation was the scoring DSL . Teachers are able to program the score bonus and penalty system for the courses on their own. Personal data of students are sent into the scoring DSL, they can be accessed through built-in variables and teachers are able to use them in the scoring script. Currently, there are variables for username, score, total time of the submission duration and time limit. This solution can be improved in the future by sending more data about a student into the DSL, for example all data of past submissions and corrections. This language was developed to be easily extendable by adding new built-in functions and variables.

I also optimized listing actions of assignment and exam modules, and the score counting of generated assignment.

In order to track whether students use the possibility of displaying a hint, I implemented the function tracking. It can be also used for tracking the usage of any other function in the system. This feature requires further processing and analysis of tracked data.

For future re-evaluation, there is a commercial business analytic software for measuring interaction of users with web applications. Such software usually processes the data and provides tools to analyse them, this would help for adding new features and improvements into the system. As examples of widely used analytic software that can be used for LearnShell in the future, I state the Mixpanel or Google Analytics – both can be used for tracking data about users and their interactions with the system.

Chapter 5

Cloud services providers comparison

LearnShell currently runs in docker containers on two servers. Future idea for LearnShell is to migrate to a cloud service. This section compares the pricing of major cloud providers, where LearnShell backend modules will migrate in the future. My estimates take into consideration that the system should handle approximately 500 users at one time. The LearnShell backend requires high speed performance and fast response, in order to process submissions of students fast. System runs 24 hours a day, 7 days a week.

I will compare the pricing of Google cloud, IBM cloud and Microsoft azure. I will focus on a solution for containerized apps. As for managing and deploying containerized web application, each provider offers a Kubernetes service, of which pricing I will analyze. For each provider, I will analyse two solutions – cheaper and more expensive. Estimated costs don't include tax.

5.1 IBM Cloud Kubernetes Service

Pricing estimates are created according to the IBM Cloud pricing calculator, which is available at [20].

5.1.1 Cheap variant

As for the cheaper variant, I considered 3 Kubernetes worker nodes, 4 vCPUs, 32GB RAM, 25GB on primary SSD and 100GB on secondary SSD. Price for such service is 1.01 USD per hour. Considering monthly pricing, **total estimated cost is 723.60 USD**.

5.1.2 Expensive variant

This variant provides higher performance by adding more vCPUs, it consists of 3 Kubernetes worker nodes, 8 vCPUs, 32GB RAM, 25GB on primary SSD and 100GB on secondary SSD. Price for such service is 1.57 USD per hour. Considering monthly pricing, **total estimated cost is 1127.52 USD**.

5.2 Google Kubernetes Engine

Pricing estimates are created according to the Google Cloud pricing calculator, which is available at [21]. Estimated costs don't include tax.

5.2.1 Cheap variant

For the cheaper variant, I managed to compile the following solution: 3 Kubernetes nodes, 4 vCPUs, 26GB RAM and 375 GB on local SSD. In case of this configuration, Google provides a 30% sustained use discount. Hourly rate is 0.789 US$. Considering monthly pricing, **total estimated cost is 575.30 USD**.

5.2.2 Expensive variant

This variant is composed of 3 Kubernetes nodes, 8 vCPUs, 52GB RAM and $2 \times$ 375GB on local SSDs. Google also provides a 30% sustained use discount for this configuration. Hourly rate is 1.575 USD. Considering monthly pricing, **total estimated cost is 1150.61 USD**.

5.3 Microsoft Azure Kubernetes Service

Pricing estimates are created according to the Microsoft Azure pricing calculator, which is available at [22].

5.3.1 Cheap variant

Cheaper variant, is composed 3 Kubernetes worker nodes, 4 vCPUs, 32GB RAM, 40GB temporary storage and 256GB SSD. Price for such service is 0.78 USD per hour. Considering monthly pricing, **total estimated cost is 588.60 USD**.

5.3.2 Expensive variant

More expensicve variant is composed of, 3 Kubernetes worker nodes, 8 vCPUs, 64GB RAM, 80GB temporary storage and $2 \times$ 256GB SSD. Price for such service is 1.638 USD per hour. Considering monthly pricing, **total estimated cost is 1234.14 USD**.

5.4 Summary

I compared the pricing of three cloud providers, where I focused on the Kubernetes solution these providers offer. The point of this analysis is to compare what the cloud providers are able to offer at the similar price. In my opinion, considering price performance ratio, the Google cloud came out the best, especially in case of the more expensive variant. But in case of the comparison of Microsoft Azure and Google cloud, there are no big differences, the prices are similar for similar configuration. Compared to that, the IBM cloud offers less performance for higher prices.

Table 5.1: Cloud pricing comparison summary

	IBM Cloud	**Google Cloud**	**Microsoft Azure**
Cheap	3 nds, 4 vCPUs, 32GB RAM, 125GB SSD	3 nodes, 4 vCPUs, 26GB RAM, 375 GB SSD	3 nodes, 4 vCPUs, 32GB RAM, 256GB SSD
Price USD/month	726.6	575.3	588.6
Expensive	3 nodes, 8 vCPUs, 32GB RAM, 125GB SSD	3 nodes, 8 vCPUs, 52GB RAM, 750GB SSD	3 nodes, 8 vCPUs, 64GB RAM, 512GB SSD
Price USD/month	1127.52	1150.61	1234.14

Conclusion

In this work, I analyzed the current architecture of LearnShell backend, API, and proposed improvements of assignment and exam modules. Improvements involved implementation of the score bonuses and penalties system, and the system of providing of hints. Implementation part of my work also involved optimization of endpoints of assignment and exam modules. The system of bonuses and penalties was developed to be easily extended and adjusted. The logic of bonuses and penalties can now be programmed for each course from the user interface of LearnShell by using the newly implemented scoring language. This language is able to load information about students and use it to affect the final score.

I also analyzed pricing of IBM cloud, Google cloud and Microsoft Azure where the LearnShell backend modules could migrate in the future.

During this work, I fulfilled all set goals and contributed to improve the performance of the LearnShell and extend its usability.

Bibliography

[1] Django Software Foundation. Django [online]. *Django, the web framework for perfectionists with deadlines*, 2021, [cit. 2021-04-16]. Available from: https://www.djangoproject.com/

[2] Visual Paradigm. What is Model-View and Control? [online]. *Visual Paradigm*, 2020, [cit. 2021-04-22]. Available from: https://www.visual-paradigm.com/guide/uml-unified-modeling-language/what-is-model-view-control-mvc/

[3] Django Software Foundation. Intro to django [online]. *Django, the web framework for perfectionists with deadlines*, 2021, [cit. 2021-04-16]. Available from: https://www.djangoproject.com/start/

[4] Shireen, A. Project VS App in Django [online]. *Atufa Shireen*, 04 2020, [cit. 2021-04-16]. Available from: https://atufashireen.medium.com/project-vs-app-in-django-755cf2a82312

[5] The GraphQL Foundation. A query language for your API [online]. *GraphQL*, 2021, [cit. 2021-04-23]. Available from: https://graphql.org/

[6] Prisma. GraphQL is the better REST [online]. *How to GraphQL*, 2021, [cit. 2021-04-25]. Available from: https://www.howtographql.com/basics/1-graphql-is-the-better-rest/

[7] The GraphQL Foundation. Queries and mutations [online]. *GraphQL*, 2021, [cit. 2021-05-10]. Available from: https://graphql.org/learn/queries/

[8] The PostgreSQL Global Development Group. What is PostgreSQL? [online]. *PostgreSQL: The World's Most Advanced Open Source Relational Database*, 2021, [cit. 2021-04-20]. Available from: https://www.postgresql.org/about/

[9] The PostgreSQL Global Development Group. Why use PostgreSQL? [online]. *PostgreSQL: The World's Most Advanced Open Source Relational Database*, 2021, [cit. 2021-04-20]. Available from: https://www.postgresql.org/about/

[10] Ask Solem. Introduction to Celery [online]. *Celery - Distributed Task Queue*, 2018, [cit. 2021-05-10]. Available from: https://docs.celeryproject.org/en/stable/getting-started/introduction.html

[11] Introduction to Redis [online]. *Redis*, [cit. 2021-05-10]. Available from: https://redis.io/topics/introduction

[12] Fowler, M.; Parsons, R. *Domain Specific Languages [online]*, chapter Using Domain-Specific Languages. Addison-Wesley Professional, 2010, ISBN 9780132107549, [cit. 2021-04-29]. Available from: https://learning.oreilly.com/library/view/domain-specific-languages/9780132107549/

[13] Fowler, M.; Parsons, R. *Domain Specific Languages [online]*, chapter Implementing DSLs. Addison-Wesley Professional, 2010, ISBN 9780132107549, [cit. 2021-05-03]. Available from: https://learning.oreilly.com/library/view/domain-specific-languages/9780132107549/

[14] John E. Hopcroft, R. M.; Ullman, J. D. *Introduction to Automata Theory, Languages, and Computation*, chapter Context-Free Grammars. Addison-Wesley, second edition, 2001, ISBN 0201441241, [cit. 2021-05-03].

[15] Fowler, M.; Parsons, R. *Domain Specific Languages [online]*, chapter Tree construction. Addison-Wesley Professional, 2010, ISBN 9780132107549, [cit. 2021-05-03]. Available from: https://learning.oreilly.com/library/view/domain-specific-languages/9780132107549/

[16] Fowler, M.; Parsons, R. *Domain Specific Languages [online]*, chapter Syntax-directed translation. Addison-Wesley Professional, 2010, ISBN 9780132107549, [cit. 2021-05-04]. Available from: https://learning.oreilly.com/library/view/domain-specific-languages/9780132107549/

[17] Nelson, R. C. Parsing [online]. [cit. 2021-05-07]. Available from: https://www.cs.rochester.edu/users/faculty/nelson/courses/csc_173/grammars/parsing.html

[18] Fowler, M.; Parsons, R. *Domain Specific Languages [online]*, chapter Symbol Table. Addison-Wesley Professional, 2010, ISBN 9780132107549, [cit. 2021-05-08]. Available from: https://learning.oreilly.com/library/view/domain-specific-languages/9780132107549/

[19] Managementmania.com. Man-day [online]. *ManagementMania*, 2016, [cit. 2021-05-05]. Available from: https://managementmania.com/en/man-day

[20] IBM corp. IBM Cloud Kubernetes Service pricing [online]. *IBM cloud*, 2021, [cit. 2021-05-11]. Available from: https://www.ibm.com/cloud/kubernetes-service/pricing

[21] Google LLC. Google Kubernetes Engine pricing [online]. *Google cloud*, 2021, [cit. 2021-05-11]. Available from: https://cloud.google.com/products/calculator?skip_cache=true

[22] Microsoft. Microsoft Azure Kubernetes Service pricing [online]. *Microsoft Azure*, 2021, [cit. 2021-05-11]. Available from: https://azure.microsoft.com/en-gb/pricing/calculator/

Appendix A

LearnShell git repository

In this appendix, I state the link, which refers to my branch of the GitLab repository of the LearnShell project:

- **Link to the LearnShell repository:** https://gitlab.fit.cvut.cz/learnshell-2.0/ls/tree/bpr-cihlaond

Appendix B

Acronyms

API Application programming interface

SQL Structured query language

UML Unified modeling language

MVC Model-view-controller

REST Representational state transfer

HTTP Hypertext transfer protocol

UI User interface

DSL Domain specific language

IDE Integrated development environment

CSS Cascade style sheets

HTML Hypertext markup language

AST Abstract syntax tree

MD Manday

vCPU Virtual centralized processing unit

Appendix C

Contents of enclosed CD

```
readme.txt ........................ the file with CD contents description
src ..................................... the directory of source codes
  learnshell ............................... implementation sources
  thesis .............. the directory of LaTeX source codes of the thesis
text .......................................... the thesis text directory
  thesis.pdf ......................... the thesis text in PDF format
```