FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

ASSIGNMENT OF BACHELOR'S THESIS

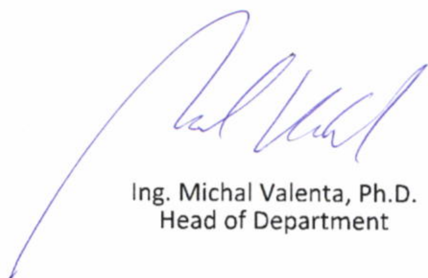
Title:	Command and script testing system for bash language
Student:	Karel Jílek
Supervisor:	Ing. Zdeněk Muzikář, CSc.
Study Programme:	Informatics
Study Branch:	Web and Software Engineering
Department:	Department of Software Engineering
Validity:	Until the end of summer semester 2018/19

Instructions

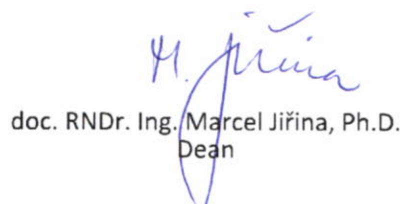
Design a system for learning and practicing the basics of Bash scripting. The system must be able to accept scripts - solution attempts from students, run them safely and to show the results. It also must be able to give hints if the student has no idea what is wrong.
More specifically:
- Provide a short review of similar systems, if there are any.
- Formalize and complete requirements and design the system architecture.
- Implement system prototype and provide suitable tests.
- Discuss your solution.

References

Will be provided by the supervisor.



Ing. Michal Valenta, Ph.D.
Head of Department



doc. RNDr. Ing. Marcel Jiřina, Ph.D.
Dean

Prague November 30, 2017

FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Bachelor’s thesis

Command and Script Testing System for Bash Language

Karel Jílek

Department of Software Engineering
Supervisor: Ing. Zdeněk Muzikář, CSc.

May 9, 2018

Acknowledgements

Although I am the author of this thesis, there are many people who directly or indirectly contributed to it and helped me a lot by doing so. This is the place where they deserve to be mentioned. My sincerest thanks go to:

- Ing. Zdeněk Muzikář, CSc., for summarizing the set of requirements for the system, for his time spent with supervising this thesis, for letting me teach the Programming in Shell course (so I could get more into the automated testing of students) and for marvellous psychical support,
- Ing. Michal Šoch, Ph. D., for his time spent with reviewing this thesis and for his incredible patience with me,
- Ing. Ladislav Vagner, Ph. D., for providing me a lot of useful information about ProgTest, a similar system to this one used at FIT CTU in Prague,
- Ing. Martin Kačer, Ph. D., for providing me a lot of useful information about online judges used in worldly recognized programming contests,
- Ing. Jiří Špaček, for helping me with installation of the server, mainly with the user authentication module,
- Ing. Radomír Polách, for helping me with the principles of building an abstract syntax tree,
- Jan Bittner, my schoolmate, for writing HTML templates so that the project looks like from this century,
- Mikuláš Poul, my schoolmate, for helping me with finding the correct configuration of Celery,
- Lukáš Hozda, my friend, for maintaining `minilinuxfs`[1], a generator of the chroot jail used in the project as a sandbox for testing the scripts,

- all teachers of the Programming in Shell course, for their patience with using the new system, for many suggested areas for improvement, and for their psychical support,
- all students of the Programming in Shell course who were forced to use this system, for their understanding and patience,
- all teachers at FIT CTU in Prague, for providing me majority of the knowledge necessary to write this thesis and for letting me through all the exams,

and lastly, but not leastly, to my friends and family, for their support not only during the time I was writing this thesis, but also during my whole life.

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 accordance with Article 46(6) of the Act, I hereby grant a nonexclusive authorization (license) to utilize this thesis, including any and all computer programs incorporated therein or attached thereto and all corresponding documentation (hereinafter collectively referred to as the "Work"), to any and all persons that wish to utilize the Work. Such persons are entitled to use the Work in any way (including for-profit purposes) that does not detract from its value. This authorization is not limited in terms of time, location and quantity. However, all persons that makes use of the above license shall be obliged to grant a license at least in the same scope as defined above with respect to each and every work that is created (wholly or in part) based on the Work, by modifying the Work, by combining the Work with another work, by including the Work in a collection of works or by adapting the Work (including translation), and at the same time make available the source code of such work at least in a way and scope that are comparable to the way and scope in which the source code of the Work is made available.

In Prague on May 9, 2018 …………………

Czech Technical University in Prague
Faculty of Information Technology
© 2018 Karel Jílek. 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

Jílek, Karel. *Command and Script Testing System for Bash Language.* Bachelor's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2018.

Abstrakt

Práce se zabývá kompletní tvorbou systému na kontrolu skriptů v jazyce Bash (online judge) – od analýzy a sběru požadavků, přes návrh, implementaci prototypu, jeho testování, až po případné nasazení do produkce. Asi největším problémem, který je u takového systému potřeba vyřešit, je možnost spustit testovaný skript tak, aby nezničil žádnou součást systému. Analýza se zabývá několika možnými přístupy. Další důležitou vlastností je jednoduché a rychlé zadávání úloh, což může být občas taky poněkud komplikované. V práci je navržen a implementován jazyk, který slouží k co možná nejjednoduššímu popisu vstupních dat. Celkově by práce měla posloužit jako návod, jak podobný systém (ne)vyvíjet.

Klíčová slova Online judge, Výuka Bashe, Automatická kontrola skriptů, Linux, Python, Django

Abstract

This thesis covers complete development of a system for automated Bash scripts checking (online judge) – starting from the analysis and collection of requirements, continuing with design, implementation, testing and deploying the system into production environment. The greatest concern of such a system is to run the tested script safely so that it never damages any part of the system. The analysis mentions several approaches to do this. A next important feature is simple and fast creation of new assignments, which might sometimes be quite complicated. This thesis contains design and implementation of custom language for generating input data. As a whole, this thesis shall serve as a guide how (not) to develop such a system.

Keywords Online judge, Tuition of Bash, Automated script checking, Linux, Python, Django

Contents

Introduction **1**

1 Analysis **3**
1.1 Existing Online Judges . . . 3
1.2 Requirements . . . 6
1.3 Fulfilling the Requirements . . . 7
1.4 Workflows . . . 13

2 Design **15**
2.1 Input Generator . . . 15
2.2 Correction Service . . . 22
2.3 Web Application . . . 22
2.4 Technology choices . . . 26

3 Implementation **29**
3.1 The Compiler . . . 29
3.2 *LI-DL* Framework . . . 38
3.3 Web Application . . . 39
3.4 Checking Script . . . 41

4 Testing and Deployment **43**
4.1 Testing . . . 43
4.2 Deployment . . . 44

Conclusion **47**

Bibliography **49**

A List of Abbreviations **53**

B LI-DL Compiler Diagrams **55**
B.1 Lexical Analyzer of *LI-DL* 55
B.2 LL(1) Grammar of *LI-DL* 59

C Screenshots of the Application **61**

D Contents of the CD **65**

List of Figures

1.1 *CodeForces* online judge 3
1.2 *Kattis* online judge 4
1.3 *DOMJudge* online judge 5
1.4 *ProgTest* online judge 6
1.5 Overwriting data, basic scenario 10
1.6 Overwriting data, incorrect locking 10
1.7 Workflow of testing a script 14

2.1 Schema of the input generator 15
2.2 Class diagram of *LI-DL* framework 21
2.3 Database schema of the web application 25
2.4 Deployment diagram of *learnshell* 27

3.1 Lexical analyzer for arithmetics 31
3.2 An abstract syntax tree for arithmetics 35
3.3 Schema of the Django framework 39

B.1 Part of lexer handling numbers in *LI-DL* 56
B.2 Part of lexer handling strings in *LI-DL* 57
B.3 Part of lexer handling identifiers in *LI-DL* 58
B.4 Part of lexer handling linebreaks in *LI-DL* 58

C.1 Teacher's view, entering a new problem 61
C.2 Teacher's view, managing test cases of a problem 61
C.3 Teacher's view, editing test case 62
C.4 Teacher's view, editing *LI-DL* generator (continuation of the form from the previous screenshot) 62
C.5 Student's view, solving a problem 63
C.6 Student's view, results of testing 63
C.7 Student's view, revealed all hints 64
C.8 Student's view, correct script 64

List of Tables

2.1 Some regular expressions and their *LI-DL* equivalents 17

3.1 LL(1) parsing table of a simplified language of arithmetics 33

4.1 Results of security testing 44

Introduction

Two programmers went together to a pub. After a few beers, they started to argue who is actually better: “I can create a mobile app, while you can’t!” says one of them. “But it will be too slow, because you don’t have the underlying knowledge!” replies the other. They continue for hours and hours, days and days, and they will most probably never agree on who is actually better.

If only they knew there is actually a way to objectively find out. Let’s give them both the same task and see who performs better. Better. What does it actually mean to perform better? For sure, the program must do what it is expected to do (e. g. have the correct output, do not crash, do not loop infinitely, …). But besides that, there are lots of other metrics. Is it more important to be fast at wriring the code? Or to consume less CPU time when the program is run? Or is it the memory consumption which bothers us most? Well, all of these criteria are (or might be) very important. Therefore, we need to balance them somehow and compute the results. Then, we can finally *judge* those two programmers. We use an *online judge*.

From the user’s point of view, an online judge is a website which displays problems to solve. The programmer solves the problem, submits the code (ideally written in a language he or she prefers, online judges usually offer a wide range of languages) and waits while his or her code is compiled and executed. Then the judge shows the result (usually accepted/failed, but may also provide some hints in case of failure).

This concept offers a possibility to organize programming competitions, usually called contests. All what needs to be done is adding some scoring (somehow balancing the metrics described above) and we are ready to start one.

Contests are not the only place where online judges are useful. They might also be used by companies when hiring new programmers or by universities to test students’ knowledge and skills.

Students at FIT CTU in Prague are required to take Programming in Shell course in their first semester of studies. They often complain that there is no

proper way to practice writing shell scripts at home. Sure, one can write a script, create some simple test data and try executing it. It seems to work. But is it actually correct? They won't probably know. Students often ask me via Facebook (I help with teaching the course and have some knowledge of Linux) for this. It takes a lot of time to check all those scripts manually, and besides eating the time, there is a possibility that a human misses something.

That is the reason why this project originated. Not only the students get their answer instantly, but the answer is also more accurate. Since the project helps students with learning the shell, it was given a really boring name: *learnshell*.

This thesis covers analysis and design of the system, describes the implementation of several components, shows how such a project can be tested, and in conclusion discusses the experience gained throughout the creation of the prototype.

Chapter 1

Analysis

1.1 Existing Online Judges

Before engineering a new system, it is generally a good idea to browse existing solutions first. It might happen that some of them is sufficient (and therefore there is not reason to develop a new one). And if not, we can at least learn some practices and patterns from them.

1.1.1 CodeForces

Figure 1.1: *CodeForces* online judge[2]

CodeForces is a typical online judge as described in the introduction. Each programmer has an account and may work as an individual or band in a team with other programmers. There are problems to practice as well as contests. A typical CodeForces contest consists of about 5 to 15 selected problems and lasts for a few hours. The goal is to solve as many problems as possible. If two programmers (or teams) have the same number of problems solved, sum of times elapsed since the beginning of the contest of each successful submission is used as the tiebreaker. The system also tracks the history of submissions and contests of each user and calculates the rank of each programmer so they can see how good they are compared to the others.

1.1.2 Kattis

Figure 1.2: *Kattis* online judge[3]

Kattis is very similar to CodeForces mentioned in 1.1.1. The main difference is that while CodeForces mainly offers programming practice and contests, Kattis aims to provide a solution for companies and universities to help them hire the best programmers and test student skills respectively. This online judge is currently used as the main system in the World Finals of *ACM ICPC*, one of the most recognized programming contests worldwide.

1.1.3 DOMJudge

Figure 1.3: *DOMJudge* online judge[4]

home | problems | login

Scoreboard NWERC Contest

final standings

RANK	TEAM	SCORE		A	B	C	D	E	F	G	H	I	J
1	**Marta, Irena & Sirup** Oxford University	**8**	1044	1/31	2/260	1/48	0	2/111	0	2/156	1/85	1/18	2/255
2	**Java the ^** Norwegian University of Science and	**7**	797	3/52	1/179	2/70	3	1/114	0	2/155	1/98	1/49	0
3	**Barton Fan Club** Helsinki University of Technology	**7**	950	2/81	1/268	2/34	7	1/119	0	1/179	2/163	1/46	0
4	**Random Oracles** University of Aarhus	**7**	1155	4/131	1/196	4/106	2	2/97	0	1/261	1/193	1/31	0
5	**Jacobs University** Jacobs University	**6**	729	1/37	10	1/113	0	3/108	0	4/257	1/78	1/36	0
6	**CU Zero** University of Cambridge	**6**	864	1/19	1/123	1/45	0	4/218	0	1	7/247	1/32	9
7	**A-Cognito** Universiteit Utrecht	**6**	871	1/116	3/98	1/188	2	4	0	1	2/109	1/125	5/95
8	**Murphys Ånglar** Lund University	**6**	1316	1/109	1/221	4/132	7/268	0	0	3	3/249	3/77	0
9	**Knuth shot first** University of Aarhus	**6**	1411	1/51	1/270	4/227	0	2/190	0	0	7/298	4/115	0
10	**Kill Dash Nine** TU Darmstadt	**5**	775	2/45	3/241	2/102	0	7	0	1	4/191	1/56	0
11	**Prime Suspects** Leiden University	**5**	863	1/57	7	1/145	2	3	0	0	2/291	1/81	2/249
12	**Aachen Mice** Rheinisch-Westfällische Technische H	**5**	911	1/124	1	1/94	0	2/209	0	0	2/252	7/72	0

DOMJudge is completely open-source, having its source code accessible to everyone on GitHub. Its website does not contain the actual judge, there is only the documentation and installation guide. Therefore, DOMJudge does not run its own contests. So whoever needs to quickly set up an online judge for free has a possibility, as DOMJudge offers a Linux installation package and a Docker image. This system is currently used as a backup for the World Finals of ACM ICPC contest mentioned before.

1.1.4 ProgTest

ProgTest, an online judge used to test students' knowledge in several courses at FIT CTU in Prague, definitely deserves a honorable mention here. The core is written and maintained by one person, *Ing. Ladislav Vagner, Ph. D.* Although it is very demanding to maintain such a system in one person, ProgTest performs surprisingly well.

Albeit ProgTest can also be considered an online judge, there are some significant differences. Firstly, it is a private project of FIT CTU in Prague and therefore it is not accessible for everyone. Secondly, it is not meant to run contests, it runs exams instead. And thirdly, there is a sophisticated system of scoring, hints, and generating of input data.

Figure 1.4: *ProgTest* online judge[5]

ProgTest ► BIE-PA1 (15/16 ZS) Logout

Name	Evaluation	Submission deadline	
Warmup homework	1.10	2015-11-01 23:59:59	View
Homework 01	5.50	2015-11-15 23:59:59	View
Homework 02	5.50	2015-11-22 23:59:59	View
Homework 03	7.50	2015-11-29 23:59:59	View
Homework 04	6.88	2015-12-06 23:59:59	View
Extra	12.16	2015-09-01 23:59:59	View
Homework 05	7.15	2015-12-13 23:59:59	View
Homework 06	6.88	2015-12-20 23:59:59	View
Homework 07	7.50	2015-12-27 23:59:59	View
Contest homework	0.00	2015-12-31 23:59:59	View
Homework 08	5.50	2016-01-03 23:59:59	View
Exam	9.71	2016-04-15 23:59:59	
Knowledge tests - 03 demo	2.50	2015-12-06 23:59:59	View
Knowledge tests - 04 demo	0.00	2015-12-31 23:59:59	View
Knowledge tests - 01	2.50	2016-01-11 23:59:59	View
Knowledge tests - 01 demo	2.50	2015-11-01 23:59:59	View
Knowledge tests - Exam - Test	16.00	2016-04-15 19:59:59	
Knowledge tests - 02	2.00	2016-01-11 23:59:59	View
Knowledge tests - 02 demo	2.50	2015-11-15 23:59:59	View
Knowledge tests - 03	2.50	2016-01-11 23:59:59	View
Knowledge tests - 04	2.50	2016-01-11 23:59:59	View
Results			View

1.2 Requirements

learnshell unfortunately requires a lot of special features which make it different from a standard online judge, therefore none of those mentioned in 1.1 is suitable as is. The closest to what *learnshell* should do is ProgTest and therefore might well serve for inspiring purposes.

A major disadvantage of ProgTest is that it takes hours to enter a problem into it. This fact makes it useless for doing weekly exams at the start of every tutorial, as the creation of about 15 new problems each week would be extremely demanding. Therefore, a huge aim of *learnshell* shall be to reduce the amount of time required to enter a new problem as much as possible. But this is not the only requirement, so let's summarize all of them.

The first and foremost requirement: like any other online judge, *learnshell* shall be accessible through a web browser. Therefore, it should be made as a web application.

In software engineering, we distinguish between functional and non-functional requirements. For better imagination of what those categories mean, say we want to design a bottle instead of a piece of software.

1.2.1 Functional Requirements

"*A functional requirement [...] essentially specifies something the system should do.*"[6]

When designing a bottle, the ability to contain fluid without leaking or the ability to be opened and closed can be considered a functional requirement.

The list of functional requirements for *learnshell* looks as follows:

1. An admin can create user accounts for students and teachers. It might happen that a student takes the Programming in Shell course again (e. g.

when they fail it) or they show up later as a teacher.

2. A teacher will be allowed to create a new problem for (and only for) a course he or she teaches. That is, he or she writes the description, solution and describes test data.

3. A student will be allowed to submit solutions for (and only for) a problem that belongs to a course he or she takes. The same is true for teachers.

4. Besides creating user accounts, an admin has a possibility to act as a student or a teacher in any course in the system.

5. The student will get to know if their solution is partially correct. That is, do not output just "yes" or "no". For an incorrect or partially correct submission, the student shall be offered a possibility to reveal a hint which could help them find the mistake.

1.2.2 Non-functional Requirements

"*A non-functional requirement [...] essentially specifies how the system should behave and [...] is a constraint upon the systems behaviour. One could also think of non-functional requirements as quality attributes for of a system.*"[6]

Let's have our bottle again. It was already specified that it has to be capable of containing fluid, but that says nothing about its shape, material, volume, durability, etc. These all could be consideres non-functional requirements.

The list of non-functional requirements for *learnshell* looks as follows:

1. The environment where the students' scripts are run must be completely isolated from the rest of the system. Not only all the files must be safe, but also excessive consumption of CPU/RAM/disk space must be prevented.

2. More correction threads should be spawned in order to speed up the process.

3. The system should do its best to prevent teachers from overwriting the work of each other.

4. The creation of a new problem should be as fast and simple as possible.

1.3 Fulfilling the Requirements

Once the requirements are collected, it is time for their further analysis, namely decisions how to fulfill all of those.

A domain model driven by functional requirements for the system will cause those to be fulfilled. The model is shown and described in 2.3.1. Let's now focus on non-functional requirements, one by one.

1.3.1 Safe Environment for Running the Scripts

This is actually a thing to be concerned about, since the server side runs potentially dangerous scripts from the users.

All in all, there are two main ways of creating an isolated environment: running a virtual machine or creating a chroot jail on the host machine. Let's discuss them both.

A virtual machine is like a computer inside a computer. It has its own (virtual) hardware, its own operating system (and therefore its own kernel), disk volumes, etc. The advantage over a chroot jail is almost complete isolation, while the disadvantage is that such a virtualization consumes some computation power and therefore it will be slower than a chroot jail.

A chroot jail is basically a directory containing executable files with their dependencies. The system can be locked in the jail using `chroot` command. At this moment, the kernel and system resources are shared with the original system and the root directory `/` is replaced with the path of the jail, making it unable to damage any files present in the outer system[1]. For sure, a malicious script could still spoil the contents of the jail, but all what it takes is to repair this is to restore a backup, since the directory has a size of a few megabytes.

Unlike the virtual machine, since the kernel and system resources are shared with the host, some more setup has to be done to make the chroot jail safe. Otherwise, the script run in the jail might consume too much memory and/or computation power. On Linux, `/etc/security/limits.conf` is our friend. In this file, maximum number of processes per user (to prevent fork bomb attacks) and maximum CPU/RAM usage (to prevent overloading of the system) can be handled. As for the limitation of disk space usage, it is easiest to create a small partition and use it for the jail. When all of these are done, the chroot jail should be safe enough to use.

The advantages and disadvantages of a chroot jail are exactly vice versa: it is faster, but it is sharing the kernel and system resources, and therefore it needs additional setup and might offer less possibilities, as for example working with the process tree (which is also shared) is potentially dangerous and therefore it must be banned in the jail.

One last concern of both options described above is infinite looping of the script. This is quite easy to solve, just add an external timer and send a kill signal to the script after n seconds in case it is still running.

learnshell uses the chroot jail approach. The key factor of this decision is its lightweightness (so that it can run well even on slower machines).

[1] Just make sure there are no hardlinks to files from the outer system in the jail, as this is a way to damage those!

1.3.2 Parallelism in Correction of Submissions

You have probably many times heard phrases like "Do it parallel, it will be faster." But how to actually achieve it?

Imagine a post office with exactly one counter opened.[2] There will be most probably a very long line of angry people. Say you are the head of such a post office and you want to make the people not wait so long. What would you do? You open more counters.[3]

As more counters are opened, a new problem arises: how to distribute the original queue between the newly opened counters? Either you let the people decide where they will be served, so a shorter line will be made in front of each counter, or you distribute numbers to the people and whenever a counter is free, the person with the currently lowest number goes there.

At the post office, the second approach is more expensive to introduce, but it is more comfortable for the customers. It might happen that a person needs a vast amount of time at the counter and therefore the whole queue of that counter is delayed. In such a case, a person that comes later and chooses a different counter might have their request finished sooner than a person already standing in the stuck queue, which is unfair.

Going back to the world of computers and *learnshell*, the customers are submitted scripts waiting to be corrected and the counters are services which check the correctness of the scripts. In the post-office-like description, we already found out that having one queue is more fair, therefore this approach is used in the project.

All what needs to be done now is to have a subsystem for distributing the customers between the workers and collecting back the results. Such a thing is called a *distributed task queue*.

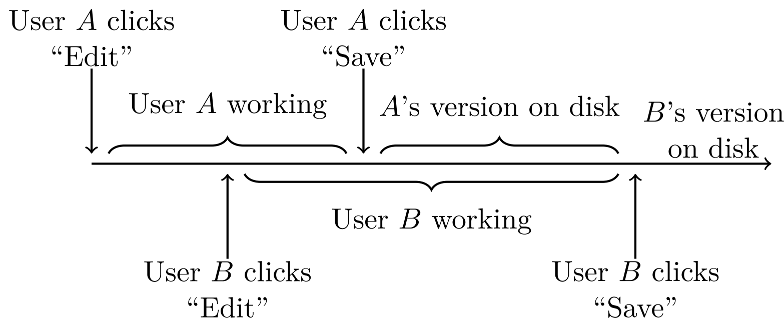
[2] It should actualy not be hard to do, this is something what you encounter the next time you will need to send a registered letter.

[3] I hope that this will be read by as many postmen as possible.

1.3.3 Protecting Data from Being Overwritten

Two teachers are editing the same problem at a time. The following diagram describes what might happen:

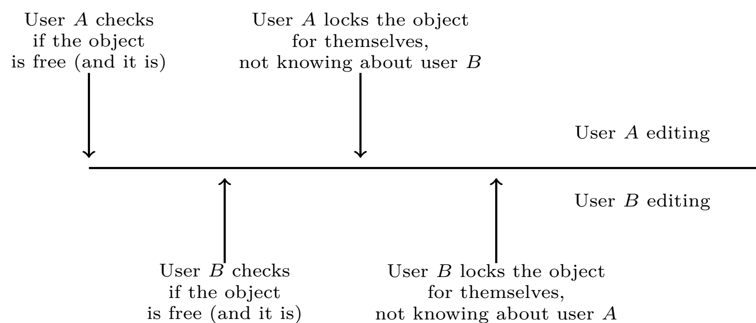
Figure 1.5: Overwriting data, basic scenario



In the end, user A's work was overwritten and thus his or her time spent with that was wasted. The system should to its best do prevent this from happening. The first idea is to introduce a database field containing the information who is currently editing the object (if anyone). Whenever a user then wants to edit, the system checks whether the object is currently "free" and if not, the request to edit is rejected and the user has to wait.

It seems like a good solution of the problem, but it is actually incorrect – it just significantly decreases the probability of a collision. What might happen is shown on the diagram below:

Figure 1.6: Overwriting data, incorrect locking



The problem is now clear: at most one request operating with the flag may be allowed at a time. This can be achieved with a concept called *atomic locks*. The operating system ensures that only one thread can hold the lock at the same time, so the solution is to lock the lock at the start of the request

and release it at the end. This way, the locking process will be safe, and thus correct.

1.3.4 Fast and Simple Creation of Problems

A problem always needs a solution, a description, and a bunch of test cases along with some hints. Therefore, its creation cannot be made easier by reducing the amount of information. The key is to simplify *how* the information is entered.

The only thing which has to stay as is is the solution script. Just go ahead and write a Bash script which solves the problem. It should not take more than a few minutes (and in some cases, even a few seconds).

When it comes to writing the description, it would be nice to have it formatted. It is not necessary to write it in pure HTML. There are tools to make it a bit easier to read and write for a human. Below is an example of one, called Markdown[7]. The code should be self-explanatory:

```
An h1 header
============

Paragraphs are separated by a blank line.

2nd paragraph. *Italic*, **bold**, and `monospace`. Itemized lists
look like:

  * this one
  * that one
  * the other one
```

And now the greatest concern, the input data. Competitive online judges test each submission with the same data to stay fair to all the competitors. This is really easy to prepare, all what needs to be done is to create a text file with the desired content. At testing time, this file is automatically redirected to `stdin` by the judge.

Unfortunately, this easy approach cannot be used here. Since we want to provide the input data to the students to help them debug their scripts, they need to be generated randomly. ProgTest solves this by uploading a script which generates all necessary input structures. This works nicely, but it takes a huge amount of time to write such a script, the probability of making a mistake there converges to 100 % and since it is a script written by a user, it can be potentially malicious, and therefore it must be run in a sandbox.

There are probably no better ways than to write a script-like piece of text. But it does not necessarily need to be the actual script. Imagine a framework (a set of classes and functions) which takes care of yielding random values and generating the actual script from it. The usage of such a framework will still consist of writing a piece of code. Maybe it would be a bit faster to write and the chance of writing a correct script would be a bit higher, but the boost in the time spent would not probably be that significant.

Whenever you need to make something better, it is a good idea to identify the worst part of the process and try to improve it. In this case, it is writing the code for the framework. Let's get inspired by another terrible language, namely JavaScript, and do the same what people did there – a *meta language*.

A meta language of a programming language is something what shall be easier to write than the original language. When the meta language code is written, a compiler converts it to a corresponding representation in the target language. A meta language could be a XML-ish or JSON-ish text file, or a completely custom language. An advantage of using XML or JSON is that there are already parsers written, and therefore there is no need to write one. A disadvantage is that the syntax of the meta language would then be bound with the JSON syntax, which may not always be nice. And a disadvantage of meta languages in general is that they might not offer the same range of options as the original language. In our case, this might be quite significant, but it is still worth if it saves the problem creators enough time.

Below is an example how such a JSON-ish meta language could look like. It puts a random valid `YYYY-MM-DD` date into a file:

```
"code": {
    "objects": [
        {
            "type": "regular_file",
            "content": [
                {
                    "type": "random_integer",
                    "range_from": 1900,
                    "range_to": 2050
                },
                {
                    "type": "string",
                    "value": "-"
                },
                {
                    "type": "random_integer",
                    "range_from": 1,
                    "range_to": 12
                },
                {
                    "type": "string",
                    "value": "-"
                },
                {
                    "type": "random_integer",
                    "range_from": 1,
                    "range_to": 28
                },
            ]
        }
    ]
}
```

Looks quite understandable and simple. But do you see all those brackets? Maybe it is simple to read, but terrible to write. So a better solution will be to have a custom meta language for this purpose, not containing that many redundant brackets. It is the most demanding option in terms of time consumed by creation of the language, but ProgTest weaknesses show that it

will be probably worth it.

1.4 Workflows

Some of the workflows in the system are a bit more complicated and thus require further analysis.

1.4.1 Testing a Script

A functional requirement for *learnshell* says that the system shall be able to recognize a partially correct script. The analysis has already revealed that a problem needs a solution, a description, and a bunch of test cases, each consisting of an input data generator. ProgTest allows different test cases to check different things – for instance, one test case might concern about the contents of `stdout`, while another one not. *learnshell* will use the same approach – besides the generator, each test case will contain a set of qualities to be verified. And also, a test case could be repeated several times to test with more random data.

The key idea is that the more test cases a script passes, the better it is. So the test cases should have increasing difficulties. To be more specific, let's introduce a real example. Say the task is to write a shell script accepting two positional parameters – file paths and it shall copy the file specified by the first into the location specified by the second. There could be three levels of testing:

- filenames following unix conventions (e. g. `file_old` and `file_new`),
- filenames containing whitespace characters (e. g. `file  old` and `file  new`), testing the quotation
- filenames resembling options (e. g. `-r` and `-p`), testing the usage of `--`.

If the test cases are executed in this order, a script featuring `cp $1 $2` should pass only the first one, while `cp -- "$1" "$2"` should pass all three.

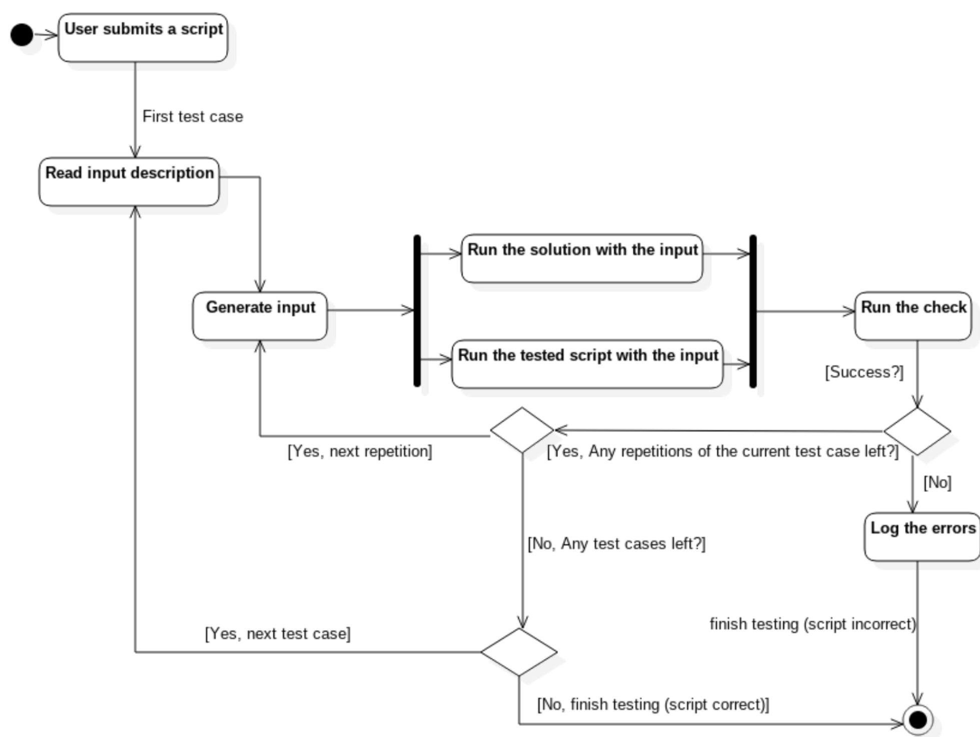
So far so good. The question now is, how to get the correct outputs and how to decide whether the tested script is correct. This is where the solution comes into play. It needs to be run in parallel with the same input data as the tested script in another chroot jail. After both scripts are finished, it is time to compare the results. For each test case, there is a given set of qualities to be checked, so the best idea seems to be to have a script which accepts those as parameters, performs the check, and tells the result.

In case the tested script is incorrect, instead of outputting plain "no", a log of errors might be returned and serve as a hint for the student. So may do the input data, so why not to provide them as well. And for each test case, a description of what is tested written by a human might potentially also help

someone. Compared to ProgTest (in shell script mode), which really struggles at providing useful hints, this shall be enough.

The workflow is summarized on the following UML activity diagram:

Figure 1.7: Workflow of testing a script



Chapter 2

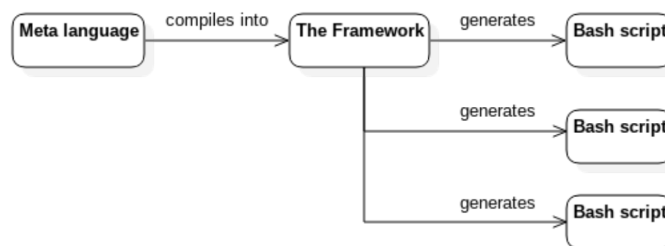
Design

2.1 Input Generator

The analysis revealed in 1.3.4 what the components of the input generator are: a framework to generate Bash scripts which are responsible for creating necessary inputs and running the tested script, and a meta language which compiles into the framework code. Since it is a brand new language, it needs a name. Because it serves for describing the inputs in the *learnshell* project, the name is very boring again: *Learnshell Input-Describing Language*, abbreviated as *LI-DL*.

The diagram below shows what exactly the role of each part is:

Figure 2.1: Schema of the input generator



2.1.1 The Language

Let's think of what the language should actually do. It shall generate files or put contents to `stdin`. All this content shall be random following certain rules. What else serves for describing a potentially random content? *Regular expressions*.

"*Regular expressions are a notation that lets you search for text that fits a particular criterion, such as 'starts with letter* `a`*'. The notation lets you write a single expression that can select, or match, multiple data strings*"[8, p. 33]

For example, a regular expression `([cC]at)|([dD]og)` matches "cat", "Cat", "dog" and "Dog".[9]

LI-DL actually needs to do the same, but vice versa: for a given "regular expression", it needs to generate a matching string. This way, things are unfortunately not that easy. For instance, think of a regular expression `.*` – it describes an arbitrary string. But the other way round, what would it mean to generate an arbitrary string? A computer completely lacks creativity[4] so further specification is needed: length of the string and a set of characters to compose the string from. The length might be a random integer from a given range. Such a string is random enough to be a good representation of `.*`.

The representation of `[abc]` and `{3,5}` should be pretty straightforward, as those already specify a set of characters and a length. The concept from the previous paragraph will be reusable here. On the other hand, complementary sets like `[^abc]` are untransformable, since we need to specify which characters *to use* (and not which characters *not to use*).

When it comes to the OR symbol – `cat|dog` – things seem to be clear at the first sight: just randomly take one of those two. But what does it mean "randomly"? That's the point, the probability of each element needs to be specified. One might want to have roughly equal amount of cats and dogs, while someone else wishes only 5 % of dogs. For simplicity, it might be useful to have a default value – the equal probability – for this. Both sets of cats and dogs are described by the same regular expression, but that expression it is insufficient when taken vice versa.

As for the capturing groups, this concept can pretty much stay the same: just generate a random string, save it, and use it later again. This can be done even better than in the original regular expressions, as the excessive usage of parentheses to denote a capturing group and backreferencing it by `\1` is not very nice. Introducing a concept of fully fledged variables as known from programming languages (assigning by `=` and reading by the variable name) would look better.

Now it is time to design a syntax for this. This is a very important point in the genesis of any programming language, since it is the syntax which can make a language very useful or very useless. An example of a great syntax is Python: "*Python is a programming language that knows how to stay out of your way when you write your programs. It enables you to implement the functionality without any hassle, and lets you write program that are clear and readable.*"[10, p. xxix]. Therefore, Python can well serve for inspiring purposes.

[4] So do I.

A random string is generated by a generator, which will be denoted by square brackets `[` and `]`. A set of characters is basically a set, which uses in Python curly brackets `{` and `}` and so it does in *LI-DL*. Ranges (Python equivalent would be slices) are specified by integers separated by a colon `:`. The separator can be a comma `,` and string, integer, float, and variable syntax can be just copied from Python as is. Let the operator for concatenating strings be `+`.

Having the syntax designed, it is time for some examples. The table below shows some regular expressions and their corresponding *LI-DL* representations discussed before:

Table 2.1: Some regular expressions and their *LI-DL* equivalents

RegEx	*LI-DL*
`[a-c]`	`[1, {"a", "b", "c"}]`
`[Hh]ello`	`[1, {"H", "h"}] + "ello"`
`a{3,5}`	`[3:5, "a"]`
`cat|dog`	`[0.5, "cat" | 0.5, "dog"]`
`([a-c])\1`	`(x=[1, {"a", "b", "c"}]) + x`

Once *LI-DL* can generate a random string, it is time to design the form of describing the actual input structures like files or positional parameters. Let's consider a regular file. It always has a name and a set of permissions, sometimes it has some content. A positional parameter always has a value. Shell variable has a name and a value, and so on. This pretty well resembles objects and their attributes. So let's introduce something similar.

An object needs to have specified what kind of object it is, a set of attributes, and possibly a set of another objects contained (for example, a directory can contain files). Let's now break a mantra of Python that no brackets will ever be used to mark off a block[5], since the indentation can be sometimes a bit painful when not using an IDE, and use curly brackets for this. Therefore, a *LI-DL* object could have the following syntax,

```
File {
    name = "hello"
    content = "hello world"
}
```

where `File` is the object type, `name` is an attribute (in *LI-DL* called *property*, as it might have an underlying procedure when having a value assigned) and `"hello"` is its value.

For more information about how *LI-DL* works, see its documentation[11].

[5]Try typing `from __future__ import braces` into your Python console.

2.1.2 The Framework

The compiler yields a piece of code. Let's design a framework which can handle it. That means not only to implement the actual stuff the *LI-DL* programmer uses, but also add some auxiliary tools needed to run and control the flow of the program.

The idea is to have a `Program` object where all the statements are stored in, and which contains a `run()` method which serves for executing the program. Upon execution, those statements are evaluated one by one in the order they were stored, the Bash code they generate is sticked together and, at the end of the day, returned as a complete Bash script.

A very important thing is that the framework shall *never* generate an invalid script. If it is forced to do so (for example by creating two files with the same name in the same directory), it throws an exception and informs the user. The only exception is when a custom piece of script is included (which is allowed in order to make the system more variable), since there is no easy way to verify the validity of the snippet. Therefore, the user must handle this with care.

Another huge point is proper escaping of characters. Everyone who has ever worked in Bash knows how painful this sometimes might be. A *LI-DL* programmer definitely does not want to wonder why his american singer `Ke$ha` ended up as `Ke` because `$ha` expanded as an undefined (and thus empty) variable. Since a lot of characters might potentially have a special meaning in Bash, the solution is to represent those using their hexadecimal values. Terrible to read, but Bash finally understands what the intended meaning was.

Also, `printf` is preferred to `echo`, since `echo` does not understand the meaning of `--`, which is important for escaping strings that resemble options (e. g. `-n`).[12]

As mentioned before, a *LI-DL* program is responsible for generating a Bash script by compiling the objects in the program. But not all objects are meant to be compiled, however those which are implement a similar logic. Plain inheritance cannot be used here, since the inheritance shall more focus on the general logic of the objects, and the ability to be compiled occurs rather randomly. This is the place where the *mixin* design pattern might come in handy.

A mixin is a class which has no meaning on its own. It just contain methods to be used by other classes without being a parent class of those. The goal is to multiple-inherit the mixin class and another class, which result in a class enriched by the functionalities of the mixin.[13] In our case, the objects to be compiled inherit from a mixin `Compilable`.

The responsibility of classes in the framework could be broken down into the following categories:

- collect raw data and assemble a higher representation from them,

- represent a generator that yields an item upon request,
- represent a primitive type and contain the logic of compiling it into Bash,
- represent a higher object (e. g. a regular file) and contain the logic of compiling it into Bash.

Let's design an appropriate inheritance hierarchy for those. Despite a factory has hardly anything in common with higher objects, it is not that bad idea to let higher objects inherit from a factory. The reason is that we want the constructions in *LI-DL* to be as commutable with each other as possible to provide more freedom to the programmers. Below is the explanation of three most important methods:

- `execute()` assembles an object. That means either create one from a factory or by "merging" two operands of an operator. If called on an assembled object, does nothing.
- `get()` forces a generator to yield an item. If called on something which is not a generator, does nothing.
- `compile()` generates a Bash code from the object.

Where does the freedom come from? For example, whenever a primitive type is needed, it suffices to call `.execute().get()` on the source object. If it is a factory, say `execute()` assembles a generator and `get()` causes this generator to provide a value. Or, if the source is already a primitive type, both `execute()` and `get()` simply return the same object. Same piece of the code, no complicated branching, and the programmer has their freedom.

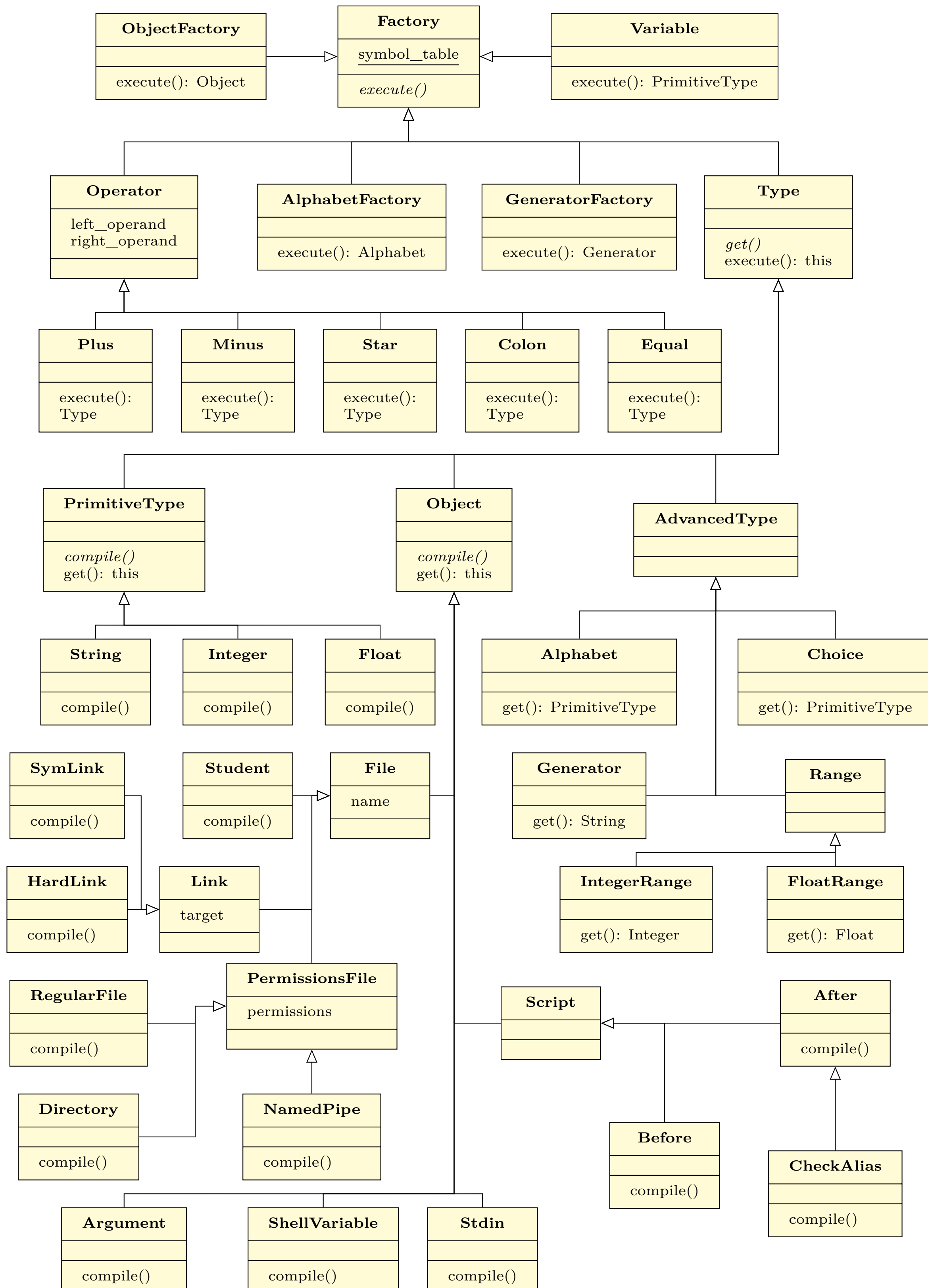
Now it is time to define the classes alongside with their functionalities.

- `Factory` itself has no meaning. It is just responsible for distributing the symbol table to all objects, as they might need it for their inner behavior.
- `ObjectFactory`, `GeneratorFactory` and `AlphabetFactory` are responsible for creating corresponding objects. Factories are written directly to the code during compilation time and must be first converted to a higher representation.
- `Variable` is an auxiliary tool to get the corresponding value from the symbol table.
- `Operator` stands always for a binary operator, therefore takes two arguments, left and right operands. Creates an object from those upon execution. Covers 5 distinct operators.

- `PrimitiveType` covers strings, integers and floats. Those might get translated to Bash as a property of an `Object` – for example, string could represent a filename.
- `AdvancedType` covers types which are capable of generating primitive types. See the *LI-DL* documentation[11] for more information how those work.
- `Object` contains some auxiliary methods to help with compiling *LI-DL* objects to Bash.
- `File` covers several unix file types. All of them have a name, some of them might have some content or permissions. Links contain a target.
- `Argument`, `ShellVariable` and `Stdin` serve for describing the contents of the corresponding input structures.
- `Script` allows to insert a piece of custom Bash code directly into the output script. It can be executed either `Before` or `After` the student's script is run. If one wants to check whether the student correctly defined an alias, the `CheckAlias` can be used.

Those are summarized in a diagram on the next page, including their inheritance relationships and most important methods and attributes. Everything what contains the `compile()` method inherits from the `Compilable` mixin which is omitted due to lack of space.

Figure 2.2: Class diagram of *LI-DL* framework



2.2 Correction Service

In 1.3.2, it was found out that a distributed task queue would be useful here. Moreover, it can also be used when compiling a *LI-DL* code, since this request might potentially take quite some time and in some rare cases, it could result in a timeout.

There are some tasks which are executed over and over again:

- compile a *LI-DL* code,
- run the input generator,
- do the actual check (including running the solution and tested script and compare their outputs).

It makes sense to divide the tasks like this, since the input generator needs to be run to both verify the correctness of the *LI-DL* code (not everything is verifiable at the compile time) and to generate the actual input for testing a script. The actual check will then consist of chain of two tasks, where the input generator passes some output to the check task.

As for the comparison of the outputs, the idea is to have two separate chroot jails. One of them is used to generate the correct output by the solution script, while the other serves for the outputs of the tested script. Then, both chroot jails are escaped and a script compares the outputs.

Besides the contents of `stdout` and `stderr`, it should also be able to verify several attributes of files (presence, permissions, content, hardlinks, …) and be capable of checking the return code.

In other words, the file tree generated by reference solution serves as a “template” and is compared with the file tree generated by the submitted script. To compare means to iterate over all the files, descending recursively into directories (if any are present), and check the required properties with the correpsonding file in the other tree.

The contents of `stdout` and `stderr` and the return code shall be redirected to auxiliary files so that the checking script is able to get to know those.

2.3 Web Application

A modern web application consists of three layers: the data layer (often called *model*) responsible for managing the data and logic of the application, the processing layer (often called *controller*) serving for taking the user input and processing it for the model, and the presentation layer (often called *view*) that renders the data for the user.[14] Putting the initial letters together, we get a shortcut MVC – a commonly used design pattern[15].

When designing a new application, the very first step is to collect the requirements and make sure they are clear. This was already done in 1.2. They are included here once again and to prevent the need to list back:

1. An admin can create user accounts for students and teachers. It might happen that a student takes the Programming in Shell course again (e. g. when they fail it) or they show up later as a teacher.

2. A teacher will be allowed to create a new problem for (and only for) a course he or she teaches. That is, he or she writes the description, solution and describes test data.

3. A student will be allowed to submit solutions for (and only for) a problem that belongs to a course he or she takes. The same is true for teachers.

4. Besides creating user accounts, an admin has a possibility to act as a student or a teacher in any course in the system.

5. The student will get to know if their solution is partially correct. That is, do not output just “yes” or “no”. For an incorrect or partially correct submission, the student shall be offered a possibility to reveal a hint which could help them find the mistake.

2.3.1 The Model

The greatest concern here is the domain model, which consists of the database and the bussiness logic.

When designing a database, look at the set of requirements first. Then define a set of things called *entities* – “*object in the system that we want to model and store information about*”[16] – and relationships between them.

Requirement 1 implies that we need entities `User` and `Course`. A user can be related to arbitrary amount of courses (a person can take the Programming in Shell course twice or even more times) and a course can be related to arbitrary amount of users (there is many students taking the same course each year). This kind of relationship is called *many-to-many*. In the database itself, such a kind of relationship needs an intermediate table, where each record contains identifiers of both entities which are related to each other, and might contain additional information. In our case, this additional information will be the kind of user-course relation, which is either a student or a teacher (let’s call this the *access level*).

If a user wishes to solve problems of a course, they need to activate the course first. This is needed to display a correct set of problems and set the access level appropriately. A user can have precisely one course activated at a time, while a course can be activated by arbitrary amount of users. This kind of relationship is called *one-to-many* and is represented by a foreign key (a

field containing the identifier of the related object) in the database. Therefore, the user will have a field denoting their currently activated course.

The next piece mentioned in requirement number 1 is that some users are admins. Being an admin is a property of a user. A user can become an admin or cease to be an admin anytime. This is a bit problematic, since, being said in the requirement number 4, admins should have arbitrary access to each course. The solution of this is to have another property for each user (even non-admins) denoting their course they have currently activated as admin, including their access level. These fields will be taken into account if and only if the user is currently an admin. Otherwise, the approach from the previous paragraph will be used.

Let's move on to the second requirement. The mentioned entities are `Problem` (along with its name, description, solution, and so on), containing several `TestCase`s. Moreover, a problem seems to contain several scripts. In general, it is not a good idea to store files to the database as they are, since they might be potentially large and thus might slow down the whole database. Therefore, a better approach is to store the file somewhere else on the disk and store only its path to the database. If such an aproach is used, it is reasonable to have a `Document` entity which can also hold another information about the file, such as its author or the creation timestamp.

When a problem is finished, it shall be first tested by the teachers to check whether it works correctly. This is what the purpose field is for: a problem can either be in testing state (visible only for teachers), or in practice state (visible for everyone taking the course). In case it reveals the problem is not correct, it must be possible to edit it. This is covered in the state field: a problem is either being edited, or is published (either for teachers only or for everyone). When one wants to edit a problem, there might happen another collision: it must be ensured that no submission of that problem is being corrected at that time. That means to have a flag whether the problem is currently accepting solutions. To edit a problem means to set this flag to `false`, wait until all enqueued submissions are corrected, and then it is safe to edit it. Also, do not forget to wrap this flag with an atomic lock to prevent possible inconsistency described in 1.3.3.

As for the test cases, each has a name and a description (both can serve as hints, as mentioned in the fifth requirement), number of repetitions (since the data are randomly generated, it has sense to let it run several times), a set of qualities that should be checked (for instance presence of files, their contents, the `stdout`, `stderr`, return code, …) and a generator in *LI-DL* language introduced in 2.1 along with its compiled form (in Python) to avoid the need to compile the code each time an input needs to be generated.

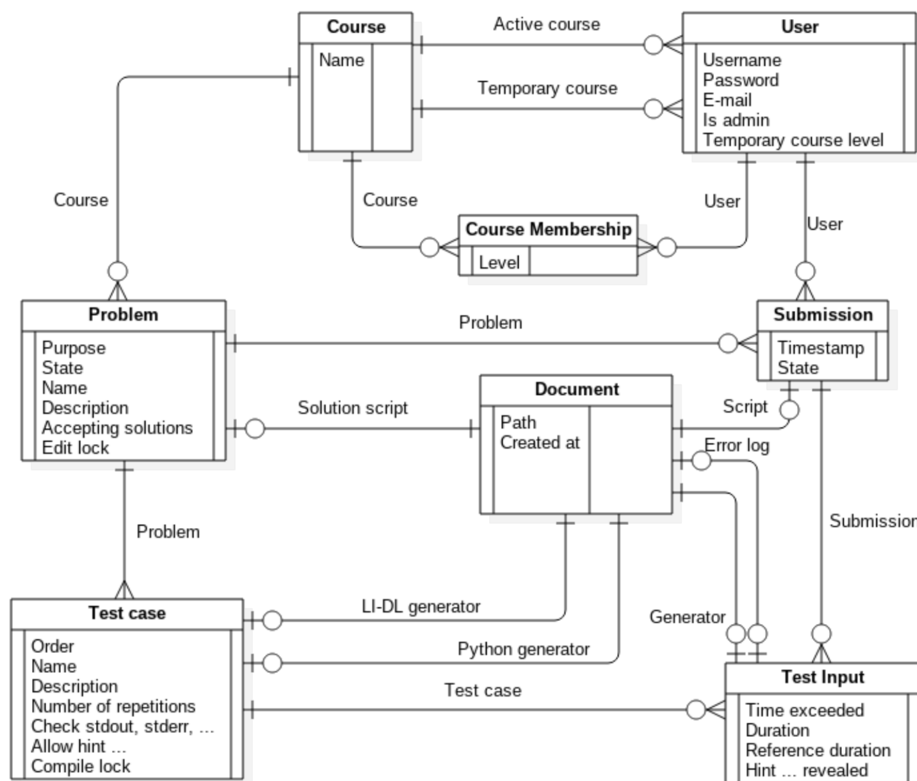
Now it is time to explain how to handle a submission from a student. For this, another two entities are needed: `Submission` and `TestInput`. The meaning of the submission entity shall be self-explanatory: it holds information like who submitted the script, for which problem, and the actual submitted script.

The `TestInput` entity serves for holding the actual data which the submitted script was tested with, and some testing results and metrics: the error log (serving as a hint), the execution time, and which hints were already revealed by the student.

To make the atomic locking mentioned in 1.3.3 a bit less demanding for the lock manager and persistent even after server reboot, the database contains some auxiliary fields: each problem has a field containing the information who is currently editing the problem, and each test case has a field preventing from running the compilation of the same *LI-DL* code at a time.

The whole database is summarized on the following diagram:

Figure 2.3: Database schema of the web application



2.3.2 The Controller

The controller shall accept the input, process it, and pass it to the model.

What does it mean to *process* the input? Besides checking if they are valid and possible slight changes in the form of it, there is also a check whether

the user actually can operate with such a piece of data in terms of atomic locking. It can happen that the data format is alright, but the key part of the application is currently locked and therefore the request cannot be completed.

Several requests in *learnshell* makes use of the same thing, the atomic locks. Therefore, the mixin pattern from 2.1.2 can be used to prevent the same piece of logic from being written more than once.

2.3.3 The View

The view is responsible for presenting the data to the user. In terms of web application, this means to generate appropriate HTML/CSS/JavaScript and send it over HTTP.

Since my skills of webdesign could be appreciated back in year 2000 but definitely not today, I am glad that *Jan Bittner*, a FIT CTU student and a friend of mine, helped me with this.

2.4 Technology choices

- The **web application** will be potentially used by hundreds of users. That is still enough for a Python backend. The Django framework combines simplicity and huge number of possibilities, while the Python itself is easy to read and fast to write. For the usage of Django in production, a proper application server is needed. An option is uWSGI.

- The ***LI-DL*** **framework** will be written from scratch as I did not find a tool which could help. The compiler itself is also written in Python from scratch. This might be a bit controversial, as the compilers should be fast and for sure, there are much faster languages than Python. On the other hand, *LI-DL* is not a Linux kernel and the compilation time will hardly exceed a few seconds. As for the other point, there are for sure many libraries to help with parsing. The reason for not using them is because I would otherwise never learn how a compiler works. Since the framework is pythonic, the compiler shall produce a Python snippet from a *LI-DL* code.

- As opposed to *LI-DL*, which was all written from scratch, Celery was chosen as the **distributed task queue** used in this project. Being written in Python and having bindings to Django, it would be a sin to not use it. Celery needs a message broker to work, an option of one is Redis.

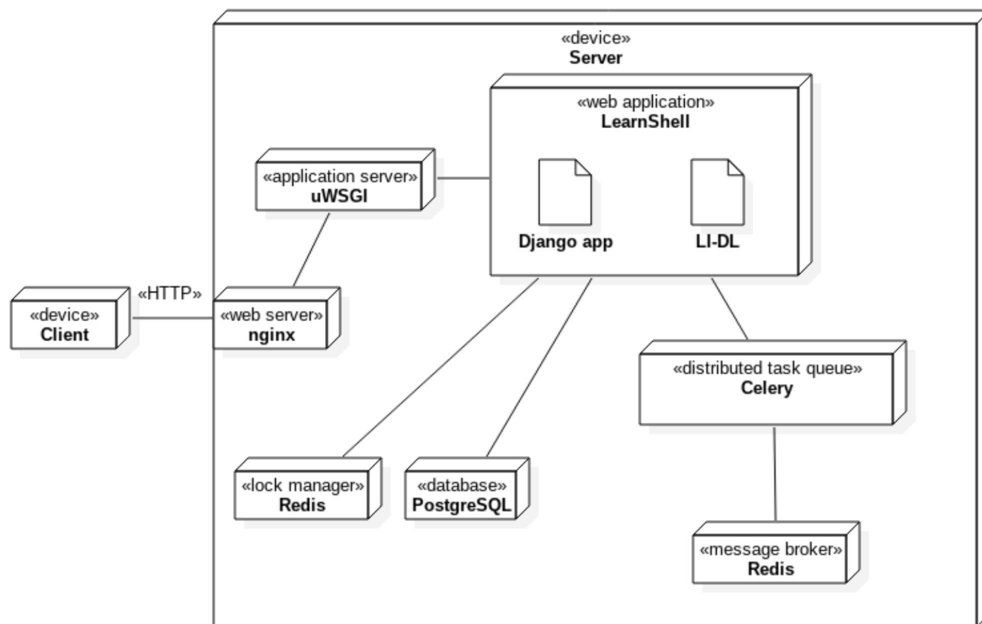
- The **distributed lock manager**. The same Redis mentioned above offers this in cooperation with `python-redis-lock` module.

- The **checking script** was again written from scratch since it is quite specific. To make things as simple as possible, it is written in Python.
- nginx is used as **the web server** since it is more modern than Apache and works nicely with uWSGI.
- **The database server** chosen is PostgreSQL, but it is rater my personal preference over another open-source options like MySQL or Firebird.

Guess my favorite programming language.

Having the technologies chosen, it is time to summarize all the components and show how they work together. A nice way of doing so is an UML Deployment diagram:

Figure 2.4: Deployment diagram of *learnshell*



Chapter 3

Implementation

3.1 The Compiler

In case the input language is simple enough, it might be easier to write a compiler intuitively rather than following these steps. *LI-DL* is definitely not the case of a simple language, it is actually very complicated. Because of that, only a part of the *LI-DL* language will be used for the explanation of the principles. Full-sized diagrams can be found in Appendix B.

Let's have the problem defined: say the input language consists of numbers, arithmetic operators $+$ and $*$ (with their usual meaning and priority) and parentheses to override the priority rules. Let the desired output be an equivalent expression in the prefix form.

I consider this as a ceiling of what could still be converted to the desired form intuitively. But only until some more operators and/or priority rules appear. An intuitively written compiler would be very hard to maintain, as the code will probably be quite messy and there will be no conventional diagrams documenting it. From now on, let's talk only about a "proper" compiler.

This compiler was written with the help of *Compiler Design Tutorial* at *TutorialsPoint*.[17] The tutorial revealed to be quite comprehensive for the needs of this compiler, therefore this chapter summarizes the techniques and principles actually used.

A compiler consists of several parts. Some of them must be always present: a lexical analyzer (often called *lexer*), a syntax analyzer (often called *parser*) and a code generator. Among the other parts, there might be several optimizers and/or intermediate code generators.

In our case, we will stick to the three mandatory parts.

3.1.1 Lexical Analyzer

Say we have a piece of the source code to compile. The first thing which needs to be done is to split it into *tokens*. A token might be for example a keyword, a constant, an identifier, a number, an operator, and so on. Besides its type, a token might need to hold a value (for example, in case of an integer) and may hold any additional auxiliary information, for example line and column numbers describing the position in the source code. This allows to raise more user-friendly exceptions containing this information.

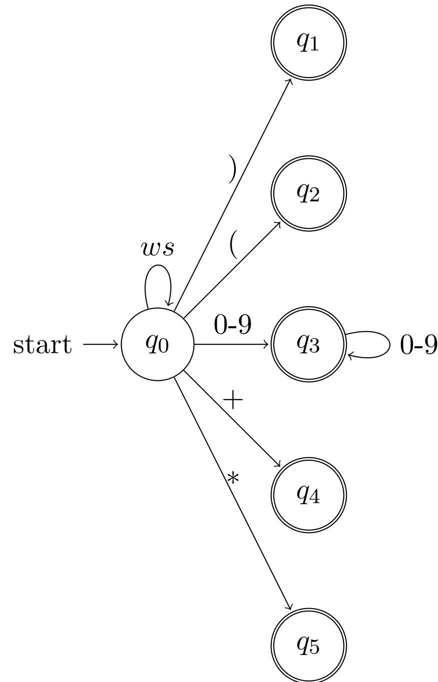
In our case of simple language of arithmetics, there are the following tokens:

- Integer
- Operator $+$
- Operator $*$
- Left parenthesis
- Right parenthesis

Once the tokens are identified, it is time to design a deterministic finite automaton which will recognize the tokens. The input for the automaton is the source code, which will be read character by character. Whenever there is no possible transition and the automaton is in a final state, it means a token is finished and it is time to read another one. In other cases, it means the current token is invalid and a compilation error happens.

Let's design the automaton first and then show how it works on some sample inputs (ws stands for whitespace):

Figure 3.1: Lexical analyzer for arithmetics



This lexical analyzer automatically skips whitespaces between tokens and for simplicity, accepts for example 0012 as a valid integer.

Let's show how it works on the input $2 * 34$:

- 2 is read, automaton goes from q_0 to q_3,
- $*$ is read, there is no transition, so 2 is taken as a token and the automaton is reset, so it goes from q_0 to q_5,
- 3 is read, no transition, $*$ is a token and automaton goes from q_0 to q_3,
- 4 is read, automaton goes from q_3 to q_3 and 4 is appended to the current token,
- end of input is read, 34 is a token.

and we are given the sequence of tokens: an integer 2, operator $*$ and an integer 34.

Let's try input))+, which is obviously syntactically incorrect:

-) is read, automaton goes from q_0 to q_1,

-) is read, there is no transition, so) is taken as a token and the automaton is reset, so it goes from q_0 to q_1,
- + is read, there is no transition, so) is taken as a token and the automaton is reset, so it goes from q_0 to q_4,
- end of input is read, + is a token.

and we are given the sequence of tokens: right parenthesis, right parenthesis and operator +. As you can see, the lexer knows nothing about the syntax, and therefore processes this as a valid piece of code. An example of something what does not pass through this lexer is an input containing a lowercase letter, let's say $+a$:

- + is read, automaton goes from q_0 to q_4,
- a is read, there is no transition, so + is taken as a token and the automaton is reset, but there is no transition and the automaton is not in a final state, therefore a compilation error occurs.

A sequence of tokens is the final product of this phase.

As the lexer is basically a finite automaton, the way of implementation is the same. There are several ways to implement one, my preferred is to have a class with methods, one for each state. Depending on the input, those methods call each other (sometimes recursively) and besides handling correct transition of the automaton, they also build up the token by sticking the current input character to its value and passing it to the next method. The only public method is *get_next_token*(). The advantage of this approach is that no ugly *goto* command is needed and the code is easily convertable from and to the automaton diagram.

3.1.2 Syntax Analyzer

This is where things start getting complicated. Syntax analyzer has two tasks: it decides whether the input is syntactically correct and if yes, it outputs an *abstract syntax tree*.

Let's do the plain syntax check first. Syntax analyzer needs two things: a sequence of tokens acquired during lexical analysis, and a set of grammar rules. There are many ways to define those rules.

Let's explain the idea of parsing. Say we have a context-free grammar of the language, which has a starting non-terminal symbol. The grammar will be made so that taking the current token from the sequence, there will be either no way to derivate to it (that will mean a syntax error) or there will be precisely one way. After applying the only possible rule, the next token will be taken again with the leftmost non-terminal symbol (which pops up after going for the first token) and the same process will be repeated. If we manage to

achieve all the tokens from the input sequence while having no non-terminals left, the input will be considered syntactically correct.

Now for the grammar itself. Given a token and a non-terminal symbol, there must be at most one possible rule to use, as mentioned above. So why not to make a 2D table for this? Such a table is called a *LL(1) parsing table*[18].

A context-free grammar is called a *LL(1) grammar* if it generates an unambiguos LL(1) parsing table table described in the previous paragraph. The first L means that the input is read from **l**eft to right, the second L means that each time, the **l**eftmost non-terminal symbol is derivated, and finally the 1 means that always **1** token is read at a time.

Let's design a LL(1) grammar for our language of arithmetics. The rules are numbered, as we will refer to them later. Terminal symbols are blue, non-terminal symbols are black, and the starting symbol is E_0:

$$E_0 \to E_1 \ E_0' \tag{3.1}$$
$$E_0' \to + \ E_1 \ E_0' \tag{3.2}$$
$$E_0' \to \varepsilon \tag{3.3}$$
$$E_1 \to E_2 \ E_1' \tag{3.4}$$
$$E_1' \to * \ E_2 \ E_1' \tag{3.5}$$
$$E_1' \to \varepsilon \tag{3.6}$$
$$E_2 \to (\ E_0 \ ) \tag{3.7}$$
$$E_2 \to int \tag{3.8}$$

This grammar will not only check the input for correctness, but will also handle the operator priority later when it comes to building the abstract syntax tree.

Now it is time to come up with the LL(1) parsing table table. There are several online tools to generate one from the grammar[19], so there is no need to do it manually. Each cell of the table means "if the leftmost non-terminal is the one mentioned in the row and the current token is the one mentioned in the column, use this rule". If there is no rule, it means a syntax error.

Table 3.1: LL(1) parsing table of a simplified language of arithmetics

	$+$	$*$	(	)	*int*	ε
E_0			3.1		3.1	
E_0'	3.2			3.3		3.3
E_1			3.4		3.4	
E_1'	3.6	3.5		3.6		3.6
E_2			3.7		3.8	

Let's do an example of how the parsing works. Say we want to parse the input $2 * (3 + 4)$. Converting this into a sequence of tokens, we get integer, operator $*$, left parenthesis, integer, operator $+$ and right parenthesis. Remember the starting symbol is E_0. So we start off with leftmost non-terminal E_0 and integer (the 2) as the current token. Looking into the table, rule 3.1 has to be used. We get:

$$E_0 \overset{3.1}{\Longrightarrow} E_1 \ E_0'$$

Now, the leftmost non-terminal is E_1 and the current token stays the same, the table says 3.4:

$$E_1 \ E_0' \overset{3.4}{\Longrightarrow} E_2 \ E_1' \ E_0'$$

Being E_2 the leftmost non-terminal and integer the current token, rule 3.8 is used:

$$E_2 \ E_1' \ E_0' \overset{3.8}{\Longrightarrow} int \ E_1' \ E_0'$$

We got a token. It corresponds with the token on the input, so this is a successful match and we can go on with the next token and the new leftmost non-terminal:

$$\begin{aligned}
&int \ E_1' \ E_0' \overset{3.5}{\Longrightarrow} int \ * \ E_2 \ E_1' \ E_0' \overset{3.7}{\Longrightarrow} int \ * \ (\ E_0 \ ) \ E_1' \ E_0' \overset{3.1}{\Longrightarrow} \\
&int \ * \ (\ E_1 \ E_0' \ ) \ E_1' \ E_0' \overset{3.4}{\Longrightarrow} int \ * \ (\ E_2 \ E_1' \ E_0' \ ) \ E_1' \ E_0' \overset{3.8}{\Longrightarrow} \\
&int \ * \ (\ int \ E_1' \ E_0' \ ) \ E_1' \ E_0' \overset{3.6}{\Longrightarrow} int \ * \ (\ int \ E_0' \ ) \ E_1' \ E_0' \overset{3.2}{\Longrightarrow} \\
&int \ * \ (\ int \ + \ E_1 \ E_0' \ ) \ E_1' \ E_0' \overset{3.4}{\Longrightarrow} \\
&int \ * \ (\ int \ + \ E_2 \ E_1' \ E_0' \ ) \ E_1' \ E_0' \overset{3.8}{\Longrightarrow} \\
&int \ * \ (\ int \ + \ int \ E_1' \ E_0' \ ) \ E_1' \ E_0' \overset{3.6}{\Longrightarrow} \\
&int \ * \ (\ int \ + \ int \ E_0' \ ) \ E_1' \ E_0' \overset{3.3}{\Longrightarrow} int \ * \ (\ int \ + \ int \ ) \ E_1' \ E_0' \overset{3.6}{\Longrightarrow} \\
&int \ * \ (\ int \ + \ int \ ) \ E_0' \overset{3.3}{\Longrightarrow} int \ * \ (\ int \ + \ int \ )
\end{aligned}$$

We managed to derivate all the way to the corresponding token sequence. Therefore, the input was syntactically correct.

Let's se what happens when the input is syntactically incorrect. Consider a single (as the input, left parenthesis in terms of tokens:

$$E_0 \overset{3.1}{\Longrightarrow} E_1 \ E_0' \overset{3.4}{\Longrightarrow} E_2 \ E_1' \ E_0' \overset{3.7}{\Longrightarrow} (\ E_0 \ ) \ E_1' \ E_0'$$

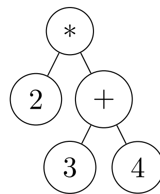
The left parenthesis is sucessfully matched. Now, the current token is ε (end of input) and the leftmost non-terminal is E_0, which is an empty cell in

the table, and therefore a syntax error. (If the tokens contain their position in the original code, the parser can even say “syntax error on line x and character y”, since it always knows the erroneous token when it comes to an error.)

The second task of a parser is to output an *abstract syntax tree*. It represents the structure of the input. The word *abstract* means that not all syntactic elements are present in the tree. In our case, it would be the parentheses, as they serve only for changing the operator priority and once the tree is built, they cease to be important.

Abstact syntax tree for the input $2 * (3 + 4)$ could look as follows:

Figure 3.2: An abstract syntax tree for arithmetics



It should be intuitive how the tree was built.

Let’s talk about the implementation now. A LL(1) parser is based on a pushdown automaton, and therefore needs a stack in order to work. One can either implement their own, but much easier is to make use of the system stack by calling methods recursively. So the parser will be a class containing a method for each non-terminal symbol, a method `match(x)` for checking whether the current token matches with a token `x`, and a variable to store the current token to. The body of each method can be easily transformed from the LL(1) grammar production rules as follows:

- For a non-terminal symbol, call the corresponding method.
- For a terminal symbol `x`, call `match(x)` to see whether the current token matches.

For example, the pseudocode of the method for production rule
$E'_0 \to + \ E_1 \ E'_0$ could look like this:

1: **procedure** E'_0
2: check for syntax error with the table
3: **if** current token = + **then**
4: *match*(+)
5: $E_1()$
6: $E'_0()$
7: ... transcript of other rules starting with $E'_0 \to$ ...

As for the building of the syntax tree, this is where things start getting a bit more complicated. First, classes for representing the nodes must be defined, in this case `IntegerNode`, `PlusNode` and `TimesNode`. Second, methods in the parser must somehow build the tree from these. Keep in mind that the methods can return a node and/or accept a node as a parameter.

Below are pseudocodes for the methods in our case:

```
1: procedure E_x
2:     check for syntax error with the table
3:     tmp ← E_{x+1}()
4:     return E'_x(tmp)
```

```
1: procedure E'_x(node)
2:     if not a syntax error and current token is different from the one this
   rule matches then
3:         return node
4:     if current token matches then
5:         match(token)
6:         tmp ← E_{x+1}()
7:         n ← new OperandNode(left_child = node, right_child = tmp)
8:         return E'_x(n)
9:     throw syntax error
```

The procedure for handling integers and parentheses is a bit different:

```
1: procedure E_2
2:     if current token = left parenthesis then
3:         match(left parenthesis)
4:         node ← E_0()
5:         match(right parenthesis)
6:         return node
7:     if current token = integer then
8:         node ← new IntegerNode(token.value)
9:         return node
10:    throw syntax error
```

Those methods produce the abstract syntax tree which is passed directly to the next piece of the compiler, code generator.

3.1.3 Code Generator

The hardest work has already been done. The source code went all the way through tokenizing and parsing to become an abstract syntax tree. It is com-

pletely up to us what we do with it at the moment. Since we have the language of arithmetics, we could for example evaluate the expression, print the prefix or postfix form of the input, or do anything else what comes up to our mind. In 3.1, I mentioned that we will try to create a prefix form of the input.

To create prefix form of a binary tree, process the node itself first, then its left subtree, and then its right subtree. Let's go straight for the pseudocode:

1: **procedure** $prefix(node)$
2: **print** $node.value$
3: $prefix(node.left_child)$
4: $prefix(node.right_child)$

And that's it. At the end of the day, we successfully converted the source code (infix form) to target code (prefix form). The compiler of *LI-DL* does a bit more difficult thing in code generation, it converts the code to Python syntax (so the output contain loads of parentheses and there are some import statements at the beginning), but the idea stays the same.

A *LI-DL* source code `1+1` would be compiled into this (note the token positions which bubble all the way to the runtime):

```
import ...
PROGRAM = Program()
PROGRAM.add(Plus((0, 1), Integer((0, 0), 1), Integer((0, 2), 1)))
...
PROGRAM.run()
```

Before code generation itself, several optimizations could usually be done, in case of arithmetics, adding a 0 or multiplying by 0 or 1 could be well optimized. Since *LI-DL* depends a lot on runtime, it does no optimizations here.

Since the compilation and running the *LI-DL* code could potentially take some time, both are implemented as Celery tasks, as described in 2.2.

3.2 *LI-DL* Framework

Although the framework is quite a complex system, it contains only about 2500 lines of source code in total. The diagram on page 21 shows a sophisticated system of inheritance, which allows to simplify the code a lot. The implementation of an object could be then as simple as shown below. The code should be self-explanatory:

```
class ShellVariable(Object):
    ALLOWED_PROPERTIES = {
        "name": (String, ),
        "value": (String, Integer, Float),
    }
    REQUIRED_PROPERTIES = ("name", "value")

    def evaluate(self, props, **kwargs):
        kwargs["output"].add_variable(props["name"], props["value"])
```

To demonstrate how the framework works together with the compiler, consider the following *LI-DL* code:

```
File {
    name = "hello"
    content = "hello world"
}
Student {}
```

The compiler converts it into the following Python representation (the imports are collapsed and the longest line is broken in the middle to fit at least a bit to this page):

```
#!/usr/bin/env python3
# compiled with LI-DL version 1

import ...
PROGRAM.add(ObjectFactory((0, 0), "File").set_property((1, 2), "name", String((1, 7),"\x68\x65\x6c\x6c\x6f"))
.set_property((2, 2), "content", String((2, 10),"\x68\x65\x6c\x6c\x6f\x20\x77\x6f\x72\x6c\x64")))
PROGRAM.add(ObjectFactory((4, 0), "Student"))
PROGRAM.run()
```

And when this script is run, it produces the following Bash script[6]:

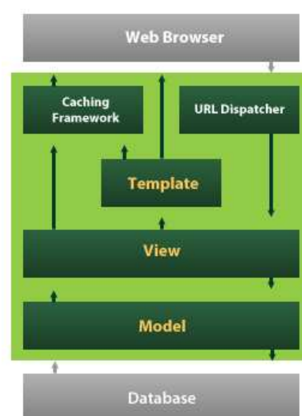
```
#!/bin/bash
shopt -s expand_aliases
[[ -z "$1" ]] && echo "Missing script path." && exit 1
if ! [[ "$1" =~ ^/ ]]; then SCRIPT_PATH="$(pwd)/$1"; else SCRIPT_PATH="$1"; fi
[[ -n "$2" ]] && cd "$2"
if [[ "$3" != "notest" ]]
then
mkdir test
cd test
fi
set --
root=$(pwd)
touch -- "$(printf -- '\x68\x65\x6c\x6c\x6f' )"
printf -- '\x68\x65\x6c\x6c\x6f\x20\x77\x6f\x72\x6c\x64'  > "$(printf -- '\x68\x65\x6c\x6c\x6f' )"
cp -- "$SCRIPT_PATH" "$(printf -- '\x53\x74\x75\x64\x65\x6e\x74' )"
chmod -- u+x "$(printf -- '%s' "${root}")"
chmod -- u+x "$(printf -- '%s\x2f\x53\x74\x75\x64\x65\x6e\x74' "${root}")"
source "$(printf -- '%s\x2f\x53\x74\x75\x64\x65\x6e\x74' "${root}")"  <<< ""
```

This script expects the path to script that shall be tested as a parameter. It handles copying it to the correct location, creating all necessary inputs, and running the script. The outputs naturally stay at their default locations, so further redirection in the process of testing is needed.

3.3 Web Application

The structure of the web application is pretty much defined by the used framework Django. Let's have a closer look on how it works:

Figure 3.3: Schema of the Django framework[20]



[6] It is not one among the nicest ones, but to be honest, have you ever seen something nice in Bash?

The journey of a user's request starts at URL dispatcher, which decides which view will handle it. The view then asks a model to retrieve some data from the database, if applicable. After the view gets the data, there is a space for additional processing, and then a response is generated, often with the use of a template[7]. This loop goes over and over with each request.

The "request wheel" of the project contains a few more steps. For instance, since there is a fairly complex system of roles, the view must verify whether the user can actually carry out the request they send. This is why the project contains a `CustomViewMixin` which adds those additional functionalities to a view. It includes:

- verifying whether the user has sufficient level to complete the request,
- passing a correct set of objects to the listing on the left (if applicable),
- passing correct texts to the heading of the page and the submit button (if applicable),
- providing an interface to pass custom data to the template.

Preventing two or more users from editing the same object at a time is also a problem which needs to be taken care of repeatedly. So let's introduce another mixin, `LockMixin`, which provides the needed behavior. Namely:

- specifying which attributes from the model shall be used to identify the lock,
- choosing whether the lock shall be automatically acquired during the request if unlocked,
- choosing whether the lock shall be automatically released at the end of processing the request,
- providing a wrapper around the calls to the lock manager, simplifying the code a bit.

Besides those mixins, the implementation makes use of class-based Django generic views[22] which implement some most common patterns, e. g. automatically retrieving an object from the database and passing it to a template. This way, some views could be as short as three lines of code.

[7]This differs a bit from the standard MVC architecture, the reasoning is explained in the Django documentation[21].

3.4 Checking Script

The checking script accepts positional parameters to determine what should be checked.

The pythonic `pathlib` module simplifies the file operations a bit, as it contains methods like `iterdir()` to iterate over a directory's content or `exists()` to check whether a file exists.

Chapter 4

Testing and Deployment

4.1 Testing

The testing of such a project goes far beyond a simple unittesting of a web application. Since the project consists of many components, it is quite important to test whether those work together correctly - such tests are called *integration tests.*[23] And most importantly, scripts written by users are run on our server. Therefore, they must not harm the system in any way.

4.1.1 Smoke Testing

A smoke test consists of performing some basic scenarios to see if the application *somehow* works. The term smoke testing comes from testing pieces of hardware – once they are plugged into electricity and start *smoking*, something probably went wrong, even before the actual behavior started to be tested.[24]

In case of this project, smoke testing consists of features which test all the components as a side effect. Therefore, it also serves well as integration testing.

Since the web application makes use of Django framework, testing is quite simple. There is a `Client` class which emulates the user. The tester only specifies which requests the user shall send. Running the tests is then as easy as writing `python manage.py test` into the console.

4.1.2 Security Testing

If the most important thing of an online judge shall be pointed out, then it would definitely be the testing whether the system is really secure. The steps made to ensure running a script is safe were already described in 1.3.1.

A disadvantage of this kind of testing is that it cannot be automated. We need a human who can think of a spot where the system can be broken, write

an exploit, and then test whether the system is protected against such an attack.

The potentially weakest place is running a submitted script from a user. Some of the system reactions to potentially malicious scripts are shown in the table below:

Table 4.1: Results of security testing

Attack description	Malicious script	System response

Attempt to damage files on the server	`rm -rf /home/$USER`
	`rm -rf .`
	`rm -rf --no-preserve-root /`
		Nothing happens, the chroot jail cannot be escaped and is restored after each run

Running a script that never finishes	`while true; do :; done`	Killed automatically after 5 seconds
Running a script that spawns infinite copies of itself (fork bomb)	`:(){ :|: & };:`	Does not allow to spawn more than N subprocesses
Running a script with high disk usage	`dd if=/dev/zero of=x \` `bs=1000000000 count=1`	Says the 50 MB partition is full
Attempt to leak some potentially sensitive data	`uname -a`	Such commands are not present in chroot jail at all

4.2 Deployment

To simplify the process as much as possible, Docker is used. To deploy an application that uses Docker, in general all what needs to be done to install Docker and run the service.

Dockerized application contains a file called Dockerfile. From a human's point of view, it contains installation instructions. But actually, those are instructions to build a thing called *Docker image*. An image is a piece of memory prepared to be run as a *container*. The term *container* comes from cargo shipping – those ships are transporting containers without any knowledge what is inside. And so does Docker: no matter which application is inside the container, no matter what technologies are used, no matter which Linux distribution is used – it just runs it. The only thing shared with the host system is the kernel, everything else is packed in the container and therefore private, including for example the process tree.

Below is a short example of a Dockerfile that creates a CentOS image with nginx web server on port 80 and supervisord to control it, assuming the configuration files for both nginx and supervisord are prepared:

```
FROM centos:7
USER 0
RUN yum -y install nginx
RUN yum -y install supervisor
COPY ./files/nginx.conf /etc/nginx/nginx.conf
COPY ./files/supervisord.conf /etc/supervisord.conf
EXPOSE 80
CMD supervisord -c /etc/supervisord.conf -n
```

Let's explain it line by line.

- `FROM centos:7` means that we start up with an image called `centos` version 7. You may think of this as if the mentioned Dockerfile was included in this one.
- `USER 0` makes the builder execute following commands as the user of UID 0 (which is root). This change of the user is valid until another `USER` directive changes it again.
- `RUN yum -y install nginx` should not be hard to guess. Just executes the given shell command.
- `COPY ./files/nginx.conf /etc/nginx/nginx.conf` copies the file specified by the first argument into the image to the location given by the second argument. The source file must be located somewhere in the building directory or its subdirectories. If it is located somewhere else, the trick is to write a wrapper shell script which copies all the necessary files into the building directory and then it is no problem to `COPY` them. Initially, there was an instruction `ADD`, which did the same except it for example also tried to unpack an archive when adding to the container. Nowadays, it is considered legacy and not recommended to use, and the only reason for presence in the current version of Docker is backward compatibility.[25]
- `EXPOSE 80` opens the port number 80 of the container so that it can communicate with the host system.
- `CMD supervisord -c /etc/supervisord.conf -n` runs the specified command every time the container is started. Since the container has its own process tree and by default, there is no `init`, it is usually a good idea to run an init-like process. `supervisord` is capable of running services, restarting them if they crash or even sending an email in case of a serious issue. Therefore, this is an option of a `CMD` command.

To have an idea of the size of a real Dockerfile, the one of *learnshell* contains nearly 100 instructions.

After a successful build, to run the container means to write one command into the console. The deployment can hardly be any simpler.

Conclusion

The goal of this thesis was to design and implement a prototype of an online judge. This goal was definitely fulfilled.

In winter semester of 2017/2018, the prototype was first tested by about 10 users at a time and then used by nearly 800 students in the Programming in Shell course at FIT CTU in Prague. All in all, both students and teachers were satisfied with it[8]. This fact certifies that the concept was right and such a system can be done this way. However, the usage in production revealed several areas for improvement.

The decision chroot jail – virtual machine could have been done better. Students often submit scripts with data races and the easiest method to check those is to insert `sleep` into some system calls, which means modifying the kernel and it is not very smart to slow down the shared kernel of the host machine. Therefore, a solution would be to run a virtual machine with a custom kernel.

Moreover, when the prototype was used in production, the whole project was running on a single virtual machine. It revealed that this was not a very smart decision, since the execution time of submitted scripts is measured (so that they can be killed if they take too long) and this can be influenced by the virtual machine manager a lot – if it decides to deny the computation power for a while, it might happen (and it happened) that even a script containing a single `echo` command might not finish in 5 seconds. The solution for this might be a physical machine with several CPUs dedicated to run the script-correcting virtual machines.

To follow the principle of loose coupling[14], the client part and the server part should be completely separated, which is not the case of *learnshell*. Django offers a DLC called Django Rest Framework which can be used to easily implement a REST API. The reason why this was not done in this prototype is that nobody got time to write a nice client part.

[8]Judging by the official FIT CTU survey done at the end of each semester.

On the other hand, *LI-DL* as the input generator was a very good idea, since it saved a significant amount of time.

It is nice to see that there is a way of automated testing that is friendly both to the students (because of revealing several hints and the ability to recognize a partially correct script) and to the teachers (because it saves their time by correcting the scripts automatically). And this is what *learnshell* provides.

Future Works

learnshell will be used as is again in winter 2018/2019. But because of some weaknesses described before and additional requirements of teachers of other courses who saw the working prototype and would like to use it as well, a new version needs to be made. Currently (May 2018), a team of about 10 people to work on this is being assembled (as opposed to the current version, which was written solely by me, only some of the frontend stuff by a friend of mine). It will take at least a year again to create a prototype of the new version. Until that time, the system from this thesis will be used. Let's believe the new version can learn from current *learnshell* a lot and be even better.

Bibliography

[1] Lukáš Hozda. *Linux almost suckless filesystem generator. Also contains bash. Statically linked.* [Online]. 2017–2018.
URL: https://github.com/luciusmagn/minilinuxfs (visited on 05/02/2018).

[2] Mike Mirzayanov. *Codeforces*. [Online]. 2010–2018.
URL: http://codeforces.com/ (visited on 05/02/2018).

[3] Kattis. *Kattis*. [Online].
URL: https://open.kattis.com/ (visited on 05/02/2018).

[4] *DOMJudge*. [Online].
URL: https://www.domjudge.org/demoweb/public/ (visited on 05/02/2018).

[5] *ProgTest*. [Online]. (Not publicly available).
URL: https://progtest.fit.cvut.cz/ (visited on 05/02/2018).

[6] Ulf Eriksson.
Functional Requirements vs Non Functional Requirements. [Online]. 2012. URL: https://reqtest.com/requirements-blog/functional-vs-non-functional-requirements/ (visited on 05/02/2018).

[7] John Gabriele. *Quick Markdown Example*. [Online]. 2014–2016.
URL: http://www.unexpected-vortices.com/sw/rippledoc/quick-markdown-example.html (visited on 05/02/2018).

[8] Arnold Robbins and Nelson H. F. Beebe. *Classic Shell Scripting*. First Edition. O'Reilly, 2005. ISBN: 0-596-00595-4.

[9] Wikibooks. *Regular Expressions/POSIX-Extended Regular Expressions — Wikibooks, The Free Textbook Project*. [Online]. 2017. URL: https://en.wikibooks.org/w/index.php?title=Regular_Expressions/POSIX-Extended_Regular_Expressions&oldid=3315722 (visited on 05/02/2018).

[10] Magnus Lie Hetland. *Beginning Python: From Novice to Professional*. Second Edition. Apress, 2008. isbn: 978-1-59059-982-2.

[11] Karel Jílek. *LI-DL documentation*. [Online]. 2018. url: http://li-dl.readthedocs.io/en/latest/ (visited on 05/02/2018).

[12] Lukáš Bařinka. *Shell FAQ*. Czech. [Online]. [Video]. 2016. url: https://youtu.be/_OnWLxRbLWM?t=5m36s (visited on 05/02/2018).

[13] Ward Cunningham. *Mix In*. [Online]. 2011. url: http://wiki.c2.com/?MixIn (visited on 05/02/2018).

[14] Microsoft. *Characteristics of modern web applications*. [Online]. url: https://docs.microsoft.com/en-us/dotnet/standard/modern-web-apps-azure-architecture/modern-web-applications-characteristics (visited on 05/02/2018).

[15] TutorialsTeacher.com. *MVC Architecture*. [Online]. 2018. url: http://www.tutorialsteacher.com/mvc/mvc-architecture (visited on 05/02/2018).

[16] SQA. *Fundamentals of Database Design*. [Online]. 2007. url: https://www.sqa.org.uk/e-learning/MDBS01CD/page_06.htm (visited on 05/02/2018).

[17] *Compiler Design Tutorial*. [Online]. url: https://www.tutorialspoint.com/compiler_design/index.htm (visited on 05/02/2018).

[18] Andrew Begel. *LL(1) Parsing*. [Online]. url: http://andrewbegel.com/cs164/ll1.html (visited on 05/02/2018).

[19] Ladislav Vagner. *Výpočet FIRST, FOLLOW a rozkladové tabulky (Calculation of FIRST, FOLLOW and parsing table)*. Czech. [Online]. url: https://users.fit.cvut.cz/~travnja3/BI-PJP/ (visited on 05/02/2018).

[20] Jeff Croft. *Standard Django architecture*. [Online]. [Image]. 2013. url: https://mytardis.readthedocs.io/en/3.0/architecture.html (visited on 05/02/2018).

[21] Django Software Foundation and individual contributors. *FAQ: General | Django documentation*. [Online]. 2005–2018. url: https://docs.djangoproject.com/en/2.0/faq/general/#django-appears-to-be-a-mvc-framework-but-you-call-the-controller-the-view-and-the-view-the-template-how-come-you-don-t-use-the-standard-names (visited on 05/02/2018).

[22] Django Software Foundation and individual contributors. *Introduction to class-based views | Django documentation*. [Online]. 2005–2018. URL: https://docs.djangoproject.com/en/2.0/topics/class-based-views/intro/ (visited on 05/02/2018).

[23] STF. *Integration Testing – Software Testing Fundamentals*. [Online]. 2018. URL: http://softwaretestingfundamentals.com/integration-testing/ (visited on 05/02/2018).

[24] STF. *Smoke Testing – Software Testing Fundamentals*. [Online]. 2018. URL: http://softwaretestingfundamentals.com/smoke-testing/ (visited on 05/02/2018).

[25] Brian DeHamer. *Dockerfile: ADD vs COPY*. [Online]. 2015. URL: https://www.ctl.io/developers/blog/post/dockerfile-add-vs-copy/ (visited on 05/02/2018).

Appendix A

List of Abbreviations

ACM Association for Computing Machinery

ACM ICPC ACM International Collegiate Programming Contest

API Application Programming Interface

CD Compact Disc

CPU Central Processing Unit

CTU Czech Technical University

DLC DownLoadable Content

FIT Faculty of Information Technology

HTML HyperText Markup Language

HTTP HyperText Transfer Protocol

IDE Integrated Development Environment

LI-DL Learnshell Input-Describing Language

MVC Model-View-Controller

REST REpresentational State Transfer

Appendix **B**

LI-DL Compiler Diagrams

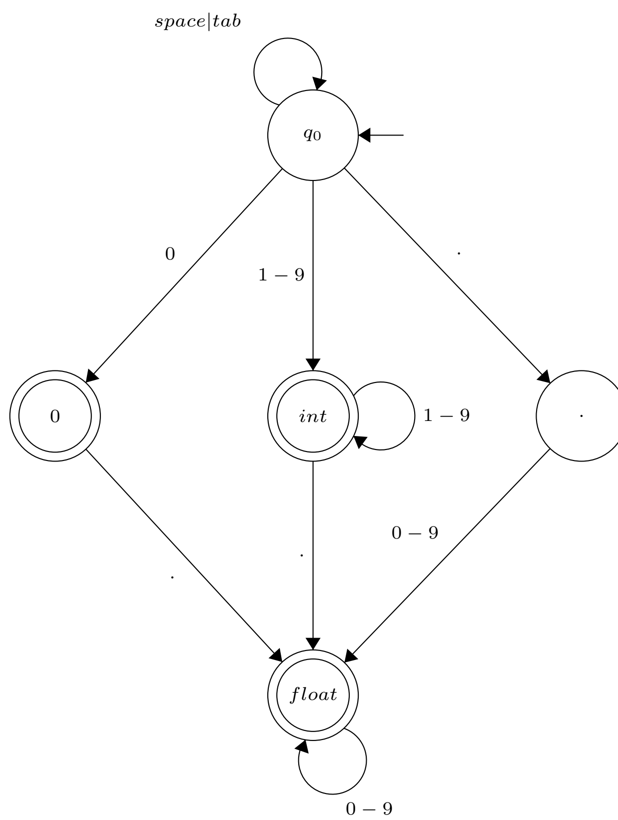
B.1 Lexical Analyzer of *LI-DL*

To make the automaton more well-arranged, it is separated into more diagrams by certain functionalities. The lexer makes a transition based on the current input symbol whenever possible. If there is no transition and the lexer is in a final state, a token is returned, the automaton is reset to the initial state, and then the transition following these rules is made. And if there is no transition with the automaton not being in a final state, an error occurs. On the diagrams, *eol* stands for "end of line" (newline, linebreak, ...), a dash (-) means a range of characters, and a | separates several alternatives for each transition.

LI-DL has several single-character tokens which are not shown on diagrams. They are ( ) { } [ ] | , = + − ∗ and :.

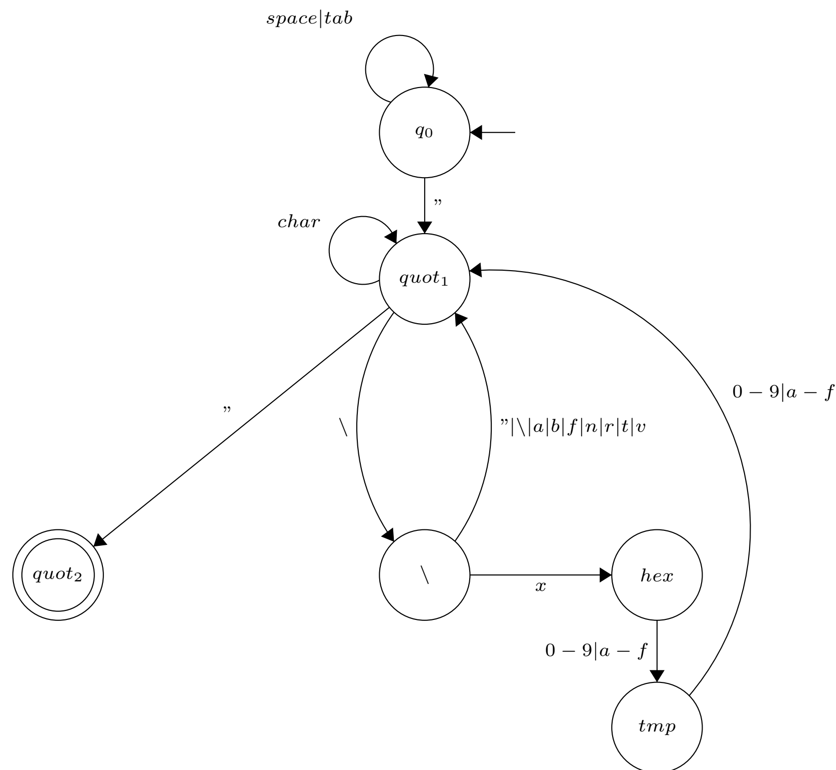
This is how *LI-DL* recognizes numbers:

Figure B.1: Part of lexer handling numbers in *LI-DL*



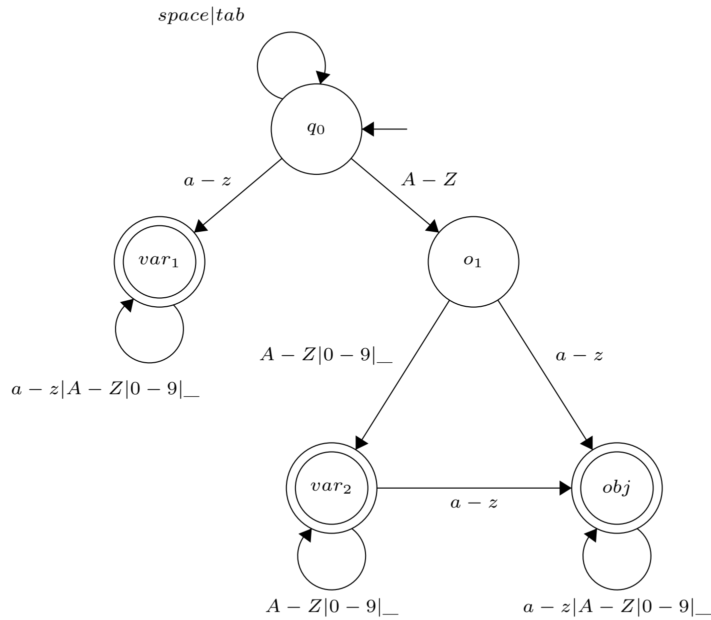
This way, *LI-DL* recognizes strings. Since there is no difference in meaning of `"` and `'` (that is, both can be used to denote a string, as long as the type of the starting quote is the same as the type of the ending one), it would be redundant to show both diagrams. You can imagine the other by replacing all instances of `"` on the following diagram by `'`. *char* stands for any character except `"` (or `'`), end of line, and `\`.

Figure B.2: Part of lexer handling strings in *LI-DL*



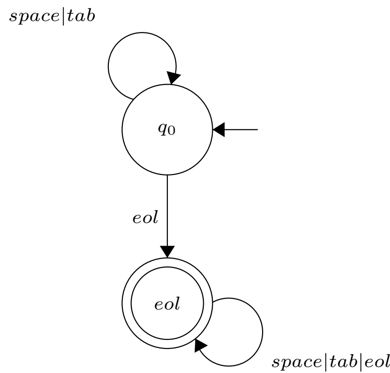
For identifiers – variable names and object names – the following automaton is used. An object name starts with a capital letter and contains at least one lowercase letter. Everything else is a variable name.

Figure B.3: Part of lexer handling identifiers in *LI-DL*



The last part of the lexical analyzer handles linebreaks, as they have a meaning: the end of an expression. However, multiple consecutive linebreaks have the same meaning as precisely one. This is something the lexer can handle.

Figure B.4: Part of lexer handling linebreaks in *LI-DL*



B.2 LL(1) Grammar of *LI-DL*

Terminal sybmols are blue, non-terminals are black, *eol* stands for “end of line”.

$$
\begin{aligned}
PROGRAM &\rightarrow EOL?\ STATGEN \\
STATGEN &\rightarrow STAT\ \ STATGEN \\
STATGEN &\rightarrow \varepsilon \\
STAT &\rightarrow STATBODY\ \ eol \\
EOL? &\rightarrow eol \\
EOL? &\rightarrow \varepsilon \\
STATBODY &\rightarrow ST1
\end{aligned}
$$

$$
\begin{aligned}
ST1 &\rightarrow ST2\ \ ST1' \\
ST1' &\rightarrow =\ \ EOL?\ \ ST2\ \ ST1' \\
ST1' &\rightarrow \varepsilon \\
ST2 &\rightarrow ST3\ \ ST2' \\
ST2' &\rightarrow +\ \ EOL?\ \ ST3\ \ ST2' \\
ST2' &\rightarrow -\ \ EOL?\ \ ST3\ \ ST2' \\
ST2' &\rightarrow \varepsilon \\
ST3 &\rightarrow ST4\ \ ST3' \\
ST3' &\rightarrow *\ \ EOL?\ \ ST4\ \ ST3' \\
ST3' &\rightarrow \varepsilon \\
ST4 &\rightarrow ST5\ \ ST4' \\
ST4' &\rightarrow :\ \ EOL?\ \ ST5\ \ ST4' \\
ST4' &\rightarrow \varepsilon \\
ST5 &\rightarrow (\ \ EOL?\ \ ST1\ \ EOL?\ \ ) \\
ST5 &\rightarrow OPERAND
\end{aligned}
$$

$$
\begin{aligned}
OPERAND &\rightarrow variable \\
OPERAND &\rightarrow string \\
OPERAND &\rightarrow integer \\
OPERAND &\rightarrow float \\
OPERAND &\rightarrow ALPHABET \\
OPERAND &\rightarrow GENERATOR \\
OPERAND &\rightarrow OBJECT
\end{aligned}
$$

$$
\begin{aligned}
ALPHABET &\rightarrow \{ \;\; EOL? \;\; ALPST1 \;\; EOL? \;\; \} \\
ALPST1 &\rightarrow ALPST2 \;\; ALPST1' \\
ALPST1 &\rightarrow \varepsilon \\
ALPST1' &\rightarrow , \;\; EOL? \;\; ALPST2 \;\; ALPST1' \\
ALPST1' &\rightarrow \varepsilon \\
ALPST2 &\rightarrow ST1
\end{aligned}
$$

$$
\begin{aligned}
GENERATOR &\rightarrow [\;\; EOL? \;\; ST1 \;\; NEXTGEN \;\; EOL? \;\;] \\
NEXTGEN &\rightarrow \varepsilon \\
NEXTGEN &\rightarrow , \;\; EOL? \;\; ST1 \;\; NEXTGEN \\
NEXTGEN &\rightarrow | \;\; EOL? \;\; ST1 \;\; NEXTGEN
\end{aligned}
$$

$$
\begin{aligned}
OBJECT &\rightarrow object \;\; EOL? \;\; \{ \;\; EOL? \;\; OBJBODY \;\; EOL? \;\; \} \\
OBJBODY &\rightarrow PROPGEN \\
PROPGEN &\rightarrow PROP \;\; PROPGEN \\
PROPGEN &\rightarrow \varepsilon \\
PROP &\rightarrow PROPBODY \;\; eol \\
PROPBODY &\rightarrow variable \;\; = \;\; ST1 \\
PROPBODY &\rightarrow OBJECT
\end{aligned}
$$

Appendix C

Screenshots of the Application

Figure C.1: Teacher's view, entering a new problem

learnshell BETA System administration Users Courses (active: Dummy Course as teacher) Problems (superuser)
Logout

PROBLEMS
Unnamed problem (Dummy Course)
+ Create problem

Edit problem Unnamed problem (Dummy Course)

Problem name: Print first parameter

Description (markdown):

```
Compose a script which print its first
positional parameters to `stdout`,
followed by a linebreak. Do not output
anything else.

**Notes:**

- The script must not leave any auxiliary
files.
- The return code might be arbitrary.
```

Solution script:

```
printf -- "$1\n"
```

Hold lock on save:
Publish on save:
Purpose: Testing

TESTS
New test

Save

© FIT ČVUT, Karel Jílek, 2017 Frontend Jan Bittner Guide to hints Report a bug

Figure C.2: Teacher's view, managing test cases of a problem

TESTS
New test

Basic test (Test #1 of problem Print first parameter (Dummy Course)) (OK) Up Down Duplicate Edit Delete
Test with whitespaces (Test #2 of problem Print first parameter (Dummy Course)) (OK) Up Down Duplicate Edit Delete
Test with option-like parameters (Test #3 of problem Print first parameter (Dummy Course)) (OK) Up Down Duplicate Edit Delete

Figure C.3: Teacher's view, editing test case

Edit test: Test with whitespaces (Test #2 of problem Print first parameter (Dummy Course))

Test name:	Test with whitespaces
Test description:	The script is run with parameters containing whitespace characters. If you are not passing this one, consider adding some quotation.
Number of repetitions:	5
Check stdout:	☑
Check stderr:	☑
Check files:	☑
Check excessive files:	☑
Check file content:	☑
Check hardlinks:	☑
Check return code:	☐

Figure C.4: Teacher's view, editing *LI-DL* generator (continuation of the form from the previous screenshot)

Check permissions:	☑
Allow hint test name:	☑
Allow hint test description:	☑
Allow hint test input:	☑
Allow hint test output:	☑
LI-DL generator:	(see code below)

```
Argument {
  count = 3:5
  value = [10, LOWER_ASCII] + [3,
WHITESPACE] + [3, LOWER_ASCII]
}
Student {}
```

Compile LI-DL

Compiled: OK
Dry run: OK

Save | Back to problem

Figure C.5: Student's view, solving a problem

Problem detail: Print first parameter (Dummy Course)

Status: Published

Compose a script which print its first positional parameters to `stdout`, followed by a linebreak. Do not output anything else.

Notes:

- The script must not leave any auxiliary files.
- The return code might be arbitrary.

Your solution:

```
echo $1
```

Submit

Figure C.6: Student's view, results of testing

Your solution:

```
echo $1
```

Submit **Submission #1 corrected (passed tests: 5 out of 15)**

#1 08. 05. 2018, 22:01:19 Passed tests: 5 out of 15

Show script Hide tests

Test #6: FAIL Execution time: 0.145s (Reference: 0.135s)
Hint "Test name"
Hint "Test description"
Hint "Test input"
Hint "Test output"

Test #5: OK Execution time: 0.166s (Reference: 0.166s)

Figure C.7: Student's view, revealed all hints

#1 08. 05. 2018, 22:01:19 Passed tests: 5 out of 15

Show script Hide tests

Test #6: FAIL Execution time: 0.145s (Reference: 0.135s)
Hint "Test name": Test with whitespaces
Hint "Test description": The script is run with parameters containing whitespace characters. If you are not passing this one, consider adding some quotation.
Hint "Test input":
Stdin: `""`
Arguments:
`ibccqywudr ggl`
`oscxhydcrn qpq`
`fflccpitrh wjn`
Variables:

Hint "Test output":

```
ERROR COUNT: 1

Stdout mismatch:
-----expected-----
ibccqywudr
  ggl

--------got-------
ibccqywudr ggl

------------------
```

Figure C.8: Student's view, correct script

```
printf -- "$1\n"
```

Your solution:

Submit **Submission #4 corrected (passed tests: 15 out of 15)**

#4 08. 05. 2018, 22:05:47 Passed tests: 15 out of 15

Show script Show tests

Appendix D

Contents of the CD

```
readme.txt ........................ description of the contents of the CD
├── lidl-compiler ................ repository containing the LI-DL compiler
├── app ........................... repository containing the web application
├── learnshell-install .... repository containing the installer using Docker
│   └── README.md ................................... installation instructions
├── assignment.pdf .................. scan of the assignment in PDF format
└── thesis.pdf ................................... this thesis in PDF format
```