UCL Université catholique de Louvain

École polytechnique de Louvain (EPL)

epl

“CorrectOz – Recognizing common mistakes in the programming exercises of a computer science MOOC”

Dissertation presented by
Nathan MAGROFUOCO , Arthur PAQUOT

for obtaining the Master’s degree in
Computer Science, Computer Science and Engineering

Supervisor(s)
Peter VAN ROY

Reader(s)
Charles PÊCHEUR, Sébastien COMBÉFIS

Academic year 2015-2016

"I never teach my pupils. I only attempt to provide the conditions in which they can learn."

Albert Einstein

Acknowledgements

This Master's thesis is the accomplishment of many years of ongoing learning. It represents our contribution to this magnificent edifice which is education.

We would first like to thank sincerely our thesis advisor Prof. Peter Van Roy. His door was always open whenever we ran into an issue or had a question about our research or writing. He steered us in the right direction from our first meeting until the very last one. Even though he was travelling, a Skype conversation was always possible.

We would also like to acknowledge both readers Prof. Charles Pêcheur and Prof. Sébastien Combéfis who graciously agreed to take part to this thesis.

Finally, we must express our very profound gratitude to our parents for providing us with unfailing support and continuous encouragement throughout our own process of learning, studying, researching, coding and writing this thesis. This accomplishment would not have been possible without them. Thank you.

Nathan Magrofuoco,
Arthur Paquot

Abstract

The development of the Internet has led to radical changes in our daily lives. Among these changes, the idea of offering college-level courses freely on the Internet has quickly gained in popularity. The concept, called MOOC, is now promoted by many organizations. edX is one of them and provides on-line courses from the best universities since 2012.

The platform is developed on its own open source software that promotes the addition of new tools to improve the quality of learning. One of these tools is called INGInious. It is an automatic on-line grader that takes the submission of a programming exercise as input and provides a feedback in return. Unfortunately, these feedback lack accuracy.

Prof. Peter Van Roy offers two courses on edX with weekly exercises graded by INGInious. The courses are based on Oz, a programming language developed for learning purposes. Prof. Peter Van Roy is always looking for new tools and ideas to improve the overall quality of on-line learning.

CorrectOz was developed to achieve this goal and to provide an extension integrable into INGInious. It is an expert tool that parses the students' submissions, interprets the parsed result and provides a complete feedback on the implementation. The ability to interpret the submissions allows to observe the semantic of the submissions, and not only their syntactic correctness. This makes it a complete feedback provider for the Oz language.

This written dissertation discusses the research and the implementation of the final tool. First it describes the analysis of the submissions, the required input to understand the students' most common mistakes. Then, it discusses the implementation choices, the architecture and the overall operation of CorrectOz, as well as its integration into INGInious. Finally, the final software solution is evaluated in terms of performances and quality, and a complete conclusion on the work is provided.

Contents

Chapter 1

Introduction

Since its creation, the Internet keeps taking a bigger place in our daily lives. No one could have imagined the scale it has reached today. Everything and everyone is connected to every other thing and every other person across the globe. All you need is a click and you can have access to a wealth of information never equalled before. Long distance communications are no longer an issue and people can share both ideas and thoughts through the Internet. This global connection has lead to innovative inventions. Knowledge was not spared by these innovations and many examples prove it is more accessible than ever.

Among many others, edX has developed its own learning tool. They now provide one of the biggest platforms to take part in a MOOC, a *Massive Open on-line Course*. These MOOCs have revolutionized the way people learn and access academic courses. edX is undoubtedly an agent of change for the years to come. Needless to say that the learning of a programming language will soon become a need for everyone. The computer science field has taken a big role in our lives during the past few years with the emergence of both mobile and web applications. The computers are more and more sophisticated and major actors have understood how the computers and the Internet can change the way we live.

Prof. Peter Van Roy saw in these MOOCs the opportunity to share his passion for programming languages outside his university. In February 2014, he started his first course on the edX platform and totalled an incredible number of 21,746 registered students. In this course, the students learn a pedagogic language called Oz. This programming language covers three of the most used programming paradigms and gives some useful tools to learn more about the complexity and the correctness of a programme, making it a *must know* for future programmers.

1 Context

The subject of this Master's thesis is the improvement of the actual grading tool used on the edX platform for the courses Louv1.1x [1] and Louv1.2x [2] taught by Prof. Peter Van Roy. During their learning, students are invited to submit exercises weekly. These exercises are graded by the INGInious platform developed at UCL. Currently, the feedback provided by this tool is inaccurate and does not allow the students to quickly understand their mistakes. Therefore, this weakens the overall quality of learning. Prof. Peter Van Roy would like to improve the grading tool by further analysing the submissions of the students in order to spot common mistakes and provide a relevant feedback concerning those errors identified. The real challenge here is to improve INGInious, not only to make it an on-line grader but also to make it a feedback provider for the Oz language.

2 Objectives

There are five main objectives that can be highlighted for this Master's thesis:

1. **Creation of a parser**: this parser should cover the full syntax of the Oz programming language in order to parse any kind of submissions written in Oz.

2. **Creation of an interpreter**: this interpreter should respect the main concepts learned during both Louv1.1x and Louv1.2x courses. The idea is to simulate the full execution of a submission that was parsed by the parser. This would allow the grader to check the execution step by step and look deeper into the students' mistakes. Through the interpreter, the grading tool would not only provide feedback on syntax errors, but it would open new possibilities such as semantics analysis. Therefore, the feedback would be much more complete than previously.

3. **Analysis of the students' submissions**: in order to provide better feedback, it is a priority to discover what kinds of mistakes are usually performed by the students. This analysis would then allow us to know *what* kind of errors we have to be careful with.

4. **Improvement of the interpreter**: being able to simulate a full execution is not useful if the interpreter does not have the right tools to detect the students' mistakes. Therefore, it is necessary to improve it in order to recognize the common mistakes revealed by the submissions analysis.

5. **Integration into INGInious**: the final solution should be fully compatible with the current grader, INGInious. Indeed, the goal is not to develop a substitute for this tool but to improve its quality for learning purposes.

3 Structure of the written dissertation

The written dissertation is organized as follows:

1. **State of the Art:** this chapter provides a big picture to better understand the concept of MOOC. It also discusses the platform edX and the on-line grading tool INGInious. This is a key chapter before getting deeper into the subject.

2. **Submissions Analysis:** a good feedback provider cannot be implemented if the goals are not set properly. Therefore, this chapter discusses the analysis performed on the submissions. The objective is to identify the students' most common mistakes in order to know which kind of errors the interpreter should be looking for.

3. **The tool:** this chapter describes the tools and the implementation choices performed to develop the final software solution. First, it discusses the parser, then the interpreter and finally provides a quick guide to run the tool.

4. **Integration into INGInious:** one essential requirement was to conceive a solution integrable into INGInious. This chapter describes the features provided by the grader to meet this objective.

5. **Evaluation:** this chapter discusses the tests performed to asses the quality of the tool. Execution time and correctness of the provided feedback are extensively tested and explained.

6. **Conclusion:** finally, a conclusion ends the dissertation by discussing the encountered issues and the future improvements to implement. A complete review of the tool and its performances is also addressed.

Chapter 2

State of the Art

This chapter provides an overview of the current knowledge we have about MOOCs. The objective is to understand what assets serve this way of teaching but also to observe what platforms and tools have been developed to make it possible. This state of the art is the starting point of our thinking. This allows to visualize the problem as a whole in order to reveal areas of improvement that can be provided for the topic.

1 Massive Open On-line Courses

Massive Open On-line Course, or MOOC, is an emerging method of on-line teaching that has become very popular the past few years. The word "massive" refers to the unlimited participation aimed by the method but also to the range of activities it can involve. On these learning platforms, students can learn electronics, architecture or even management at the same time. However, the main asset of a MOOC is its ability to be "open". MOOCs promote knowledge for anyone, at any time, from anywhere. Students can register and freely access the courses, their assessment processes and supporting tools. These tools are often open source themselves.

This method challenges the traditional teaching and offers new perspectives of learning. According to Ken Masters, it is the result of an evolving process that can be described as the fourth step of on-line education development [3]:

1. **80s-90s approach**: the lecturer uploaded notes, materials and presentations on an on-line repository that was only available on a local area network.

2. **90s approach**: the 90s have seen the development of home-grown systems or externally-developed Learning Management Systems (LMS). Those platforms were centralising teaching and learning in the same environment for the first time. The materials were still available on those systems but learner-learner and teacher-learner interactions were possible through chat rooms, discussion forums or wikis.

3. **2000s approach**: the content area kept diminishing in favour of other tools encouraging interactions in the LMS. New management tools became prominent such as grade book tools or quiz.

4. **MOOC, the 2010s approach**: the learning and teaching platform is not centralised anymore and opens the door to social networking. The instructor now steps aside and becomes a "guide" that provides external ways of learning such as videos, blogs and on-line articles.

The main difference between traditional teaching and MOOC lies in the mindset of students [4]. Learners become active and are fully integrated in the learning process. The students are independent and it is up to them to take part in a course. Their motivation is the reflection of their own needs. Some of them even skip the assessment part. Their motivation is driven by the need of raising their awareness on a specific subject. Those learners aim primarily at learning and not gaining certification.

But this methodology has also several defects and the success of MOOCs requires conceptual changes for both instructors and students. Indeed it can be difficult, even impossible, to answer personally to everyone because of several thousands students that are often registered to a course. This issue highlights the need for tools to correct, grade and help the learners at any time. MOOCs still lack for technical support, efficient assessment evaluation or student feedback but innovative solutions are developed and integrated every year. The on-line grader INGInious is one of these inventions that helps the MOOC's successful implementation. We present it later.

Many platforms emerged from the MOOC concept such as Coursera, Udacity and edX. The latter will take our attention in the next section.

2 edX

In 2012, Harvard and MIT partnered to create edX, an on-line learning initiative offering college-level courses from the world's best universities and institutions. Two years later, more than 50 top-rated universities, non-profit organizations and corporations provide on-line courses on the platform. edX provides these courses at no charge and the platform runs on its own open source software called Open edX. Since June 2013, the whole project has even been available on GitHub. Nowadays, edX offers more than 650 courses and has delivered more than 580,000 certificates.

edX describes its objectives in 3 steps [5]:

1. **Increase access to high-quality education for everyone, everywhere**: the platform only gathers well-recognized universities and organizations to provide the best teaching to its members. But the key asset of the project is its availability at no charge for everyone, from everywhere.

2. **Enhance teaching and learning on campus and on-line**: edX also promotes the most innovative solutions to improve learning and teaching. Their open source solution is well-suited for the integration of new learning tools.

3. **Advance teaching and learning through research**: a complete staff of researchers works on edX to analyse the students behaviour on the platform. Their research on on-line learning should provide them the right keys to improve the overall quality of teaching in the next few years.

In February 2014, UCL stepped into the project and now offers 4 courses among 18 applications. Prof. Peter Van Roy's application was chosen among them and his programming course quickly gained in popularity with 21,746 registered students [6]. Later, the course was divided into two smaller courses and Louv1.01x was renamed Louv1.1x and Louv1.2x. By deploying those courses on the platform, UCL goes even further than the "Décret gratuité" initiated in Belgium which aims to provide course materials at no charge on a virtual support for the enrolled students [7].

The platform works as follows. Students can create an account on edX and start to register freely to the on-line courses. They learn at their own pace by visualizing relatively short videos (ranging from 3 to 20 minutes). Between two videos, they are invited to test their understanding by answering short surveys. Some are evaluated, some are not. The certification modalities are always explained at the beginning of the course. The students also have access to on-line textbooks and on-line discussion forums to answer questions and discuss about their learning. At the end of the course, the students are invited to answer a bigger quiz standing as a final exam. edX provides two kinds of certificate (1) an honour code certificate for free registrations, and (2) a verified certificate for registrations performed with a donation.

During their learning, students have access to many learning tools. For example, electronic courses are organized in an on-line lab that simulates the building of virtual circuits. The programming courses can run INGInious to automate the correction of the students' submissions. This automated on-line grader is described in the next section.

3 INGInious

Aiming at unlimited participation brings some defects to the MOOC concept. Among these issues, there are too many students involved in a MOOC for handmade corrections. Therefore, MOOC needs correction tools to assist the teachers in their correction tasks. INGInious is one of these tools. It is an automated correction and grading platform developed to be generic [8]. Indeed, INGInious can potentially run algorithms from any programming language. The tool does not trust the user inputs. Consequently, the tool is safe as it runs students' code inside specific virtual environments confined from the base system. INGInious is also scalable and can deal with large numbers of submissions

at the same time. It provides an interface to interact with and is entirely free and open-source. Like Open edX, INGInious is available on GitHub.

Teachers can set up the tool to create their own courses. Each course is made of tasks, each one comprising one or more exercises. Each of these exercises is divided in two directories, the first directory contains various informations on the exercise and the second one contains the actual script to run. This script is responsible for grading the students' submissions and provide them some feedback.

For now, the feedback is generated by providing unitary testing. If the submission delivers the right answers for the given inputs, then the submission is considered successful. Therefore, the unitary testing should include many different tests such as limit case testing and subset testing (testing functions individually). Teachers are also advised to force the students to use some pre-defined lines of code while implementing their solution. A good idea is usually to keep exercises short and to focus on one particular task at a time.

At this point, only syntax errors can be efficiently provided as feedback. Indeed, the grader is fully able to report compilation errors. Otherwise, it reports a vague runtime error or fail status. The grader actually lacks semantic feedback.

Chapter 3

Submissions Analysis

1 Motivations

The objective of submissions analysis is to identify the students' mistakes. In order to turn our interpreter into a good feedback provider, it is necessary to provide it with ways of identifying the potential programming errors. Submissions analysis is the first step towards this goal.

This section is subdivided into three parts explaining the methodology that we applied: (1) we describe the INGInious submissions received as input, (2) we explain the analysis itself to extract the common mistakes from those submissions and (3) we explain how we transformed these mistakes into markers.

2 INGInious submissions

2.1 Archive description

In order to identify the students' mistakes, we had the privilege to have access to the submissions stored by INGInious during the last semester. Those submissions have been stored anonymously for both courses Louv1.1x and Louv1.2x. They are the inputs of our analysis.

We have been provided with an archive filled with those students' submissions. Fig. 3.1 describes the organisation of a student's folder in this archive. In fact, each student has his own folder which contains one subfolder for each tried exercise. And each exercise has one subfolder for each submission sent to INGInious by the student. Note that the name of both student and submission folders are supposed to be hashed because of privacy issues. For comprehension purpose, we do not display those hash values on figure 3.1. Finally, each submission folder contains itself the following files:

- **submission.test** which contains the basic informations related to the submission

such as the date of submission, the grade obtained, the code submitted, the feedback received.

- **new_file.oz** which contains the exact code executed by the grader. Indeed, each submission is in fact integrated into a pre-defined code for testing and pre-condition purposes (sometimes, some functions are already defined and should be called by the students).
- **errC.txt** which contains the error reported by the MOZART compiler.
- **out.txt** which contains the output produced by the execution: tests or students' Browses. Note that Browse stands for *print* in Oz.

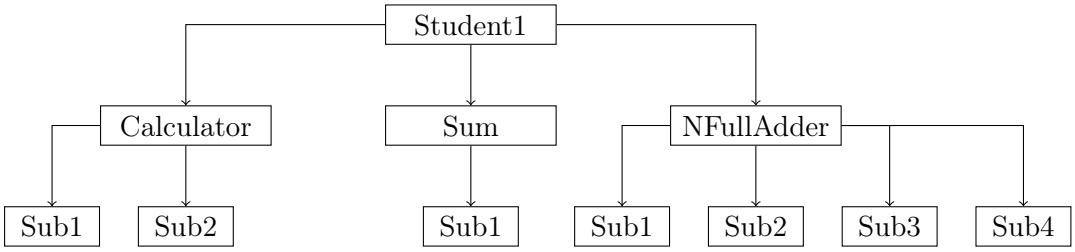


Figure 3.1: Folder organisation for each student

2.2 Statistical analysis of the archive

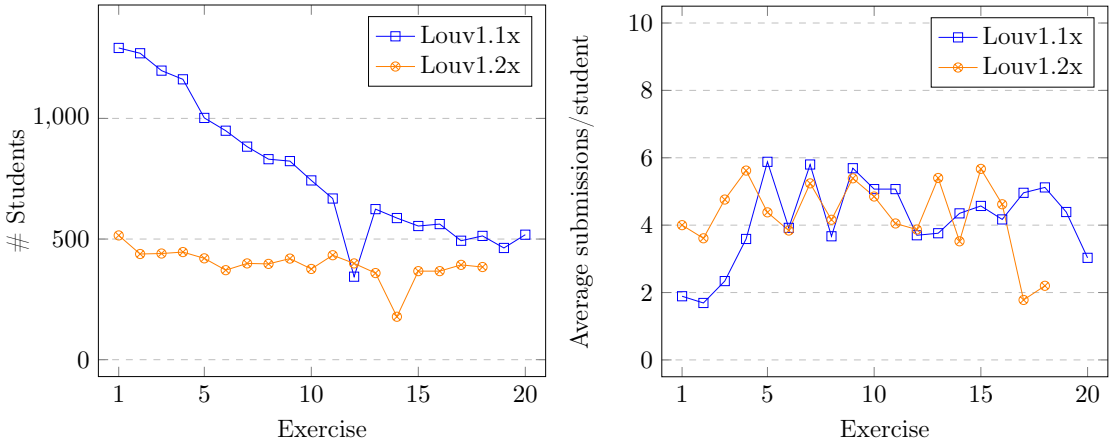
Let's take a deeper look at this archive. Table 3.1 provides some general informations about students, exercises and submissions. For this table and the next ones, the character "#" means "number of" and the character "%" means "percentage of". Unfortunately, there is no way to differentiate the students enrolled only in LOUV1.1X, only in LOUV1.2X or in both courses at the same time. Therefore, it is better to consider the two courses as one for further analysis. Note that the number of students presented in this table reflects only the number of students that actually tried at least one exercise on INGINIOUS. This is the required condition to have a trace of a student in the archive of submissions. An interesting observation is that students do not perform *all* the exercises that are proposed to them. In average, they try less than one third of those exercises even though they are graded. This feature seems similar to the conclusion of Ken MASTERS explained in the state of the art [3] : many learners do not take part to a MOOC to gain a certificate, but for their self interest.

Fig 3.2a describes the evolution of student participation for each exercise. A predictable observation is the greatest participation during the first course. But this participation falls drastically and decreases by half at the end of the course. The second course manages to keep a constant participation. Only the most motivated students took part in this course. The student participation to the last exercise of LOUV1.1X is nearly the

# Students:	2,009
# Exercises tried:	22,580
# Submissions:	90,943
Average exercises/student:	11.23
# Exercises proposed:	38
% Exercises tried in average:	29.55%
Average submissions/student:	45.26
Average submissions/exercise:	4.02

Table 3.1: Recapitulative table concerning students, exercises and submissions

same as the participation in the first exercise of LOUV1.2X. It seems that learners who have completed the first course are willing to participate to the second one afterwards. Finally, two dips can be observed on the graph in Fig 3.2a. The first dip corresponds to a non-graded exercise that was obviously skipped by many students. The second one is part of a bonus lesson taught by Prof Seif HARIDI who developed the Oz language along with Prof Peter VAN ROY. However, this exercise is still graded even if this lesson is considered as a "bonus" one. Note that you can find statistics about each exercise in appendix **??**.



(a) Student participation for each exercise (b) Avg # submissions/student for each exercise

Figure 3.2

Fig 3.2b describes the average number of submissions per student for each exercise. The students submit approximately the same amount of submissions each time they tried to perform an exercise. Therefore, the overall complexity of the exercises seem

to be equivalent for both courses. However, this amount decreases by half for the last exercises of each course. Those exercises correspond to the final exams and seem to be performed more carefully by the students. Note that the first exercises of LOUV1.1X demonstrate a predictable facility to be successful. Indeed, those exercises are intended to be easy to start slowly the learning of Oz language.

Fig 3.3 shows the count of submissions that match the same grade. INGINIOUS has 6 different grades to evaluate a submission: (1) **success** if the submission has managed to pass all the tests, (2) **failed** if the submission has not managed to pass those tests or could not be executed, (3) **crash** if INGINIOUS has encountered an internal error while grading the submission, (4) **overflow** if the execution of the submission has raised an overflow exception, (5) **timeout** if the execution has timed out and (6) **killed** if an administrator of INGINIOUS or the student himself has stopped the grading of the submission. Figure 3.3 shows that many submissions are considered as failed. Therefore those submissions are neither able to deliver the right solutions, nor syntactically correct. This is an interesting observation for our analysis. Indeed, this means that students usually perform bad computations (at least, not the expected ones) or syntax errors.

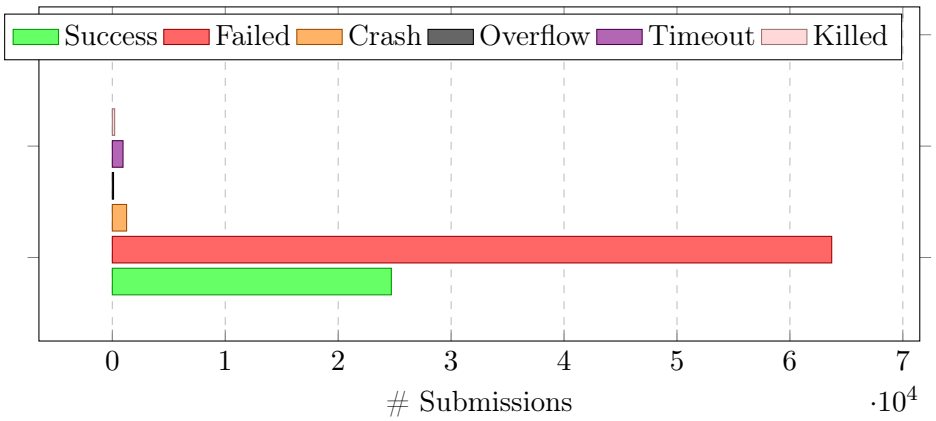


Figure 3.3: Count of submissions for each grade

The comparison between table 3.1 and fig 3.3 shows another interesting feature. As mentioned in table 3.1, 22,580 exercises have been tried by the students and exactly 24,709 submissions are considered as *successful* by INGINIOUS as shown in figure 3.3. If more submissions are successful than the number of exercises tried, this should mean that the students usually manage to provide good answers after some submissions. And some students also seem to improve their implementation by providing several successful submissions.

At this point some conclusions can already be observed:

- Most students perform less than one third of the provided exercises.

- The students' participation tends to decrease quickly at the beginning but remains constant after a given point.
- The overall complexity of the exercises seems equivalent during both courses.
- Syntax or computation mistakes seem to be the main causes of failed submissions.
- Most students manage to provide successful submissions despite their first failures.

3 Common mistakes extraction

The previous section highlighted two interesting features: (1) many students provide bad submissions but finally manage to provide a successful one, (2) among those bad submissions, most are considered as *failed*, which means they contain syntax or computation mistakes. Those observations are essential because we have set the objective to reveal the students' common mistakes. This section is about showing the extraction of those mistakes from the submissions. A key idea is also to understand how the students manage to provide successful answers from their unsuccessful ones and the feedback provided by INGInious. This is a required condition to improve the overall quality of the feedback.

The section is subdivided into five subsections: (1) explains why it was essential to perform the exercises ourselves, (2) describes the procedure applied to extract the common mistakes from the students' submissions, (3) provides a complete example of extraction for demonstration purpose, (4) provides a complete description of the markers that have been implemented and (5) stands as a conclusion.

3.1 Self exercising

It was essential to start by performing the exercises ourselves. The objective was triple: (1) to develop a better appreciation of each exercise, (2) to provide a point of comparison and (3) to clear the path for further analysis. Developing a better appreciation of each exercise is essential to understand the thoughts and reflections made by the student in his own submission. Moreover, it is obvious that no one can effectively analyse a submission without any point of comparison with the initial statement.

However, clearing the path for further analysis was the most essential step towards the extraction of common mistakes. By applying our own reflection and thoughts on each exercise, we were able to reveal a first load of mistakes. Those mistakes could be purely syntactic, e.g. a conditional statement poorly structured or a forgotten *end* keyword to close a statement. But many of them were also related to comprehension issues. Three kinds of comprehension mistakes are particularly interesting: (1) when students do not understand a concept to apply, (2) when a statement is not precise enough and needs clarifications and (3) when the student himself applies a bad reflection on the

problem. Those kinds of syntactic and comprehension issues are encountered everyday by programmers when they try out a new language or are confronted for the first time to a new problem. This is exactly representative of the situation of the students during their learning and the extraction procedure should be especially precise and careful about those kinds of mistakes.

3.2 Extraction procedure

The next step of the submissions analysis is the extraction of the common mistakes. As stated previously, a careful attention should be oriented towards syntax and comprehension issues. After this task, the interpreter should be able to detect the common issues that have been revealed. Then, the idea is to observe the solutions implemented by the students and extract, from them both assets and defects to improve the current list of mistakes and the feedback part of the interpreter.

As explained in a previous section, each submission has a *submission.test* file that stores general informations and the code submitted by the student. Two cases should be taken into account: (1) if the submission has obtained a *success* grade or (2) if the submission has *failed*, *timed out*, *overflowed*, *crashed* or has been *killed*.

1. If the submission is successful then the interest lies in the quality of the implementation. A good submission can involve the deployment of a smart strategy to solve the problem. But it can also involve a cleaner or even better code than our own implementation. Indeed, some students seem very comfortable with the Oz language and we had to improve our interpreter to add some syntactic constructions that are not explained during the EDX courses.

   Those *better* students are at the source of another feature. It consists in providing also a feedback on the number of variables and functions that have been declared. Therefore, if a student has declared 10 variables and 4 functions, the interpreter is able to warn him that he could have used less variables and functions instead (if this is actually the case).

2. If the submission is not successful, a careful analysis of the potential mistakes is required. The grade and the feedback provided by INGINIOUS can be used to start looking for those mistakes. Some syntax errors are already well handled by the grader but some are not. This is exactly when the interpreter should provide better feedback. If an experienced programmer is able to detect some mistakes, then, the interpreter should also reveal them to the students.

   There are two objectives during this careful analysis: (1) to understand and localize the error, and (2) to confirm this error by looking at different students. It is essential to confirm a mistake in the submissions of other students. Indeed, challenging the exactitude and precision of a statement should not be allowed if a comprehension issue was raised only once for this statement. If a concept was not understood by

only one student, this does not mean that the whole learning is a failure. However if many students cannot apply a concept or provide the right solution to a problem, then, the theoretical lesson or the statement of the exercise itself should be thought again.

The key idea behind extraction procedure is that many submissions should be analysed carefully. This requires that the list of mistakes grows large enough to be representative of the *common* mistakes. For each exercise, the submissions of 30 students randomly selected have been observed. This makes 120 submissions per exercise on average according to table 3.1. In fact, three kinds of students can be observed:

1. The *better* ones: they have a good understanding of algorithmic. And some already master pretty well the Oz language and its syntax. They usually do not provide new errors but they are at the source of the improvements described in the first point above.

2. The *average* ones: they are the "Good Samaritans" of the interpreter. They are the ones that perform the common mistakes. The vast majority of them are provided by those students. They gather both syntax and comprehension mistakes. They submit between four and six submissions on average before implementing a successful submission.

3. The *bad* ones: unfortunately they are not the most helpful students because they just do not get what they are trying to achieve. They can send a dozen submissions for the same exercise because they are completely lost. Sometimes, they do not even find out a right solution. Therefore, they do not provide *common* mistakes but they highlight a lack of theoretical understanding. Fortunately, those students are rather uncommon.

3.3 Example

This section presents a complete example of extraction in order to clarify the procedure that has been applied. As a brief recall, this procedure consists of : (1) trying the exercise, (2) observing some submissions and (3) reporting the common mistakes.

3.3.1 Sum

The fourth exercise proposed in the course LOUV1.1X is called *Sum*. It relies on the notion of accumulator to compute the sum of the square of the N first integers. The provided function should be tail recursive. Tail recursion is achieved when the recursive call is the last instruction of the function. The statement of the exercise itself provides a first requirement for the interpreter: it should be able to recognize if a function is tail recursive or not. This is not a *common mistake* but it is still an essential concept in the course and it should be implemented in the interpreter. The principle of communicating vases and the concept of invariant are also introduced during the theoretical lesson. For this exercise, the invariant is already provided to the students: $sum(n) = sum(i) + acc$.

They are also aware that their submissions will be added into the piece of code described in fig 3.4:

```
1       fun {MainSum N}
2         local Sum in
3           fun {Sum N Acc}
4             [YOUR CODE]
5           end
6           {Sum N 0}
7         end
8       end
```

Figure 3.4: Pre-defined piece of code for the Sum exercise

According to the principle of communicating vases, the variable N should decrease at each iteration and the accumulator Acc should keep *accumulating* the result until N is equal to zero. This is not a tough exercise for a confirmed programmer and the solution can be written in a few lines as shown in fig 3.5.

```
1       if N==0 then Acc
2       else {Sum N-1 Acc+N*N}
3       end
```

Figure 3.5: One potential solution for the Sum exercise

Let's now take a look at some submissions to find potential mistakes. Each submission is presented by providing the code and the error reported by INGInious. The first submission ended with a compilation error as shown in fig 3.6. To report this error, INGInious trusted the Mozart compiler. This compiler is able to report bad parsing caused by syntax error. In this case, there is indeed a missing bracket to start the function call. However, the interpreter does not use this compiler and should be modified to identify and report correctly those kinds of parsing issues.

Figure 3.6: First selected submission for the Sum exercise

```
1       if N==0 then Acc
2       else Sum N-1 Acc+N*N}
3       end
```

Compilation Error: There is a compilation error! The message 'Parse error' often means that you have forgotten a closing bracket, a 'end',etc. Or maybe, there are too many brackets, 'end',etc.! Take a look at the error line. The line may be incorrect

because if an end is missing, for instance, it looks too far away for the error. Parse error line 2, column 6.

Fig 3.7 shows a misunderstanding of single-assignment. Indeed, the student has tried to modify the value stored by the accumulator variable. Unfortunately, this is a semantic error that cannot be precisely reported by the MOZART compiler nor INGINIOUS. The interpreter should be able to report unsuccessful multiple-assignment.

Figure 3.7: Second selected submission for the Sum exercise

```
1       if N==0 then Acc
2       else Acc = N*N + {Sum N-1 Acc}
3       end
```

Runtime Error: There is a runtime error! The error given by the emulator is: 'Tell: 1 = 0'.

Fig 3.8 describes a series of four consecutive submissions sent by the same student. In those cases, the feedback provided by INGINIOUS are pretty relevant. Through the unitary testing, the automatic grader is able to determine the source of the problem which is the bad use of the accumulator. INGINIOUS also provides both expected and returned results. We can see that the student has been trying to find his mistake by submitting several submissions with little changes. In fact, the student has simply not understood well enough the invariant to be respected and the principle of accumulator. An even better feedback would be to advise him to read those explanations once again. Indeed, his first submission is doubly wrong: (1) he does not return the accumulator which implies a bad understanding of the concept (an accumulator is supposed to store the result) and (2) he returns a value one step earlier than expected which implies a bad understanding of the invariant.

3.3.2 Conclusion on Sum

At this point, three kinds of mistakes have been observed: (1) a syntax error, (2) a semantic issue and (3) a misunderstanding of a concept. Many more submissions should be looked over carefully to confirm that those mistakes are common among students. In practice, this extraction methodology was applied to the submissions of 27 additional students. For clarity and length, we will not present those submissions in this paper.

```
1       if N==1 then 1
2       else {Sum N-1 Acc+N*N} end
```

Test failed Error: Your code does not provide the right answers. Acc may not have the correct value at each call. Here is the state of arguments for each call of Sum 6 0. Computation of Sum 6 0: Value of Acc when N = 6: 0. Value of Acc when N = 5: 36. Value of Acc when N = 4: 61. Value of Acc when N = 3: 77. Value of Acc when N = 2: 86. Value of Acc when N = 1: 90. Your result: 1. Expected result: 91.

```
1       if N==0 then 1
2       else {Sum N-1 Acc+N*N} end
```

Test failed Error: Your code does not provide the right answers. Acc may not have the correct value at each call. Here is the state of arguments for each call of Sum 6 0. Computation of Sum 6 0: Value of Acc when N = 6: 0. Value of Acc when N = 5: 36. Value of Acc when N = 4: 61. Value of Acc when N = 3: 77. Value of Acc when N = 2: 86. Value of Acc when N = 1: 90. Your result: 1. Expected result: 91.

```
1       if N==0 then 1+Acc
2       else {Sum N-1 Acc+N*N} end
```

Test failed Error: Your code does not provide the right answers. Acc may not have the correct value at each call. Here is the state of arguments for each call of Sum 6 0. Computation of Sum 6 0: Value of Acc when N = 6: 0. Value of Acc when N = 5: 36. Value of Acc when N = 4: 61. Value of Acc when N = 3: 77. Value of Acc when N = 2: 86. Value of Acc when N = 1: 90. Your result: 92. Expected result: 91.

```
1       if N==0 then Acc
2       else {Sum N-1 Acc+N*N} end
```

Figure 3.8: Four successive submissions from the same student for the Sum exercise

3.4 Conclusion on extraction

At the end of extraction, the three kinds of mistakes observed in the demonstration exercise appear to be the most common ones across the entire diversity of exercises:

1. Most errors are caused by **comprehension issues** based on the misunderstanding of the exercise. Some of those issues involve computation mistakes because the

students do not understand the expected result or have not got a sufficient background to solve the specific problem they are facing. Sometimes, the students do not understand in which pre-defined code their submission will be integrated. In those cases, they encounter *syntax* errors against their will, e.g. they put a "*end*" keyword at the end of their submission whereas it was already included in this pre-defined code.

2. Some mistakes are also caused by **techniques and concepts misunderstanding**. In these cases, the students do not use properly those techniques and concepts and they end up facing *semantic* issues. A few students do not even manage to deliver a good solution after a dozen submissions because they do not get the error behind their inaccurate reasoning.

3. Some **syntax errors** could also be observed, but not that much. In fact, some students were so interested by the Oz language that we even learned a few kinds of conditional statements. Therefore, those students helped us to complete the interpreter.

4. Finally, we found out that LOUV1.2X is facing serious **plagiarism** issues. In some cases, a few students have *exactly* the same submission, and we "only" observed 30 submissions per exercise. Unfortunately, it is hard to quantify the plagiarism without an expert tool dedicated to this purpose. Therefore, this fact should be taken for what it is worth and the question "Does INGInious need an anti-plagiarism tool ?" is ignored in the paper.

4 Markers

Providing a complete list of common mistakes is not useful for the interpreter if it is not able to identify those mistakes. The next step of the analysis is to refine those mistakes into *markers*. A marker is intended for being directly implemented into the interpreter. Each marker can detect one issue and delivers the accurate feedback for this issue. They are classified into two categories: (1) hints and (2) errors. A hint provides a feedback about a mistake that do not compromise the final result, but that could be modified to improve the quality of the submission. It mostly relates to good practices. As for the errors, they deal with mistakes that lead to a failure and that should be handled with great care.

4.1 Syntax errors markers

4.1.1 General syntax errors

1. **"Error: You have to put the argument(s) of an object-related method between parenthesis."**

   The students often mix up the call to a procedure/function with the call to a method specific to an object. In the first case, the whole call is expected to be between

brackets. In the second case, the arguments are expected to be surrounded by parenthesis instead.

2. **"Error: The argument(s) of the method should not be between parenthesis."**

   As stated above, a method call should not be surrounded by parenthesis but by brackets instead.

3. **"Error: This operator does not exist in Oz."**

   Some students try to use unknown operators in Oz such as incrementation (++) or decrementation (−−).

4. **"Error: The end of an instruction in Oz is not marked by a ';' ."**

   An instruction in Oz has no character that indicates its end. However, a statement should be closed explicitly with the *end* keyword.

5. **"Error: A variable should always start with an uppercase character."**

   By definition a *variable* should always start with an uppercase character whereas an *atom* should start with a lowercase character.

6. **"Error: A ") " is missing."**

7. **" Error: A " (" is missing."**

8. **"Error: A " { " is missing in your procedure/function declaration."**

9. **"Error: A " } " is missing in your procedure/function declaration."**

10. **" Error: A "] " is missing."**

11. **"Error: A " [" is missing."**

12. **"Error: You cannot perform one assignment for two variables at the same time."**

    Many students are often used to other programming languages using a different syntactic sugar than Oz to declare new variables. The following syntax rule is also provided along with this marker:

```
1   <var1> = <var2> = <value>      /* not allowed in Oz */
2
3   <var1> = <value>               /* should be used instead */
4   <var2> = <value>
```

4.1.2 If statements

As a reminder, a complete conditional *if* statement respects the following syntax:

```
1  if <condition> then <statement>
2  elseif <condition> then <statement>  /* optional */
3  else <statement>
4  end
```

In practice the full syntax rule is always provided along with the feedback.

1. **"Error: A " then " keyword is missing in the if statement."**
2. **" Error: A " end " keyword is missing in the if statement."**
3. **"Error: A " then " keyword is missing in the elseif statement."**
4. **" Error: A " end " keyword is missing in the elseif statement."**
5. **"Error: A statement is missing after the " then " keyword."**

4.1.3 Case statements

As a reminder, a complete *case* statement respects the following syntax:

```
1  case <expression> of <pattern0> then <statement>
2  [<pattern1>] then <statement>     /* optional */
3  [<pattern2>] then <statement>     /* optional */
4  else <statement>
5  end
```

In practice, the full syntax rule is always provided along with the feedback.

1. **"Error: A pattern is missing in the case structure. At least one pattern should be provided."**
2. **"Error: A " then " keyword is missing in the case structure."**
3. **"Error: A " [ ] " or an " else " statement is missing in the case structure."**
4. **"Error: A " of " keyword is missing in the case structure."**
5. **"Error: A " end " keyword is missing in the case structure."**

4.1.4 Local statements

As a reminder, a complete *local* statement respects the following syntax:

```
local <variable0> <variable1> ... <variableN> in
  <statement>
end
```

In practice, the full syntax rule is always provided along with the feedback.

1. **"Error: You should use a local statement instead of a declaration statement."**

   The student does not understand the use of the *declare* keyword, confusing it with a *local* statement.

2. **"Error: A " end " keyword is missing in the local declaration."**

3. **"Error: A " in " keyword is missing in the local declaration."**

4.1.5 While and For statements

As a reminder, the complete *while* and *for* statements respect the following syntax rules:

```
while <loopDescription> do
  <statement>
end
```

```
for <variable> in <variable> do
  <statement>
end
```

In practice, the full syntax rule is always provided along with the related feedback.

1. **"Error: A " do " keyword is missing in the for/while statement."**

2. **"Error: You can only use one variable in the for loop."**

4.2 Semantic errors markers

1. **"Error: You cannot change the value of the variable because it was already assigned to one. A variable can only be assigned to one value in Oz. Use cell with ":= " for multiple assignments."**

   Oz supports only single-assignments. If a student tries to assign twice a variable, we perform the assignment but we warn him that it should have used a cell or several variables instead. We assign the variable to the new value in order to find and report other mistakes further in the submission.

2. **"Error: The variable is not declared."**

   Once again we still fake the declaration and assignment of the variable in order to retrieve other mistakes farther in the submission.

3. **"Error: The variable is not a cell/attribute. You cannot use the ":= " operator."**

   Some students encounter issues with the fact that only cells allow multiple-assignment in Oz whereas variables allow only single-assignment.

4. **"Error: You cannot apply the " ~ " operator to a non-number term."**

   In Oz, "~" is used to express negative numbers.

5. **"Error: One of the terms of your operation is a cell/attribute. Use the " @ " operator to access its content."**

   The notion of cell always involves an operator to access its content.

6. **"Error: You should use the " / " operator on floats only. The operator " div " should be used for integers instead."**

   Oz supports 2 kinds of divisions: one for integers only, one for floats only.

7. **"Error: The variable is not a number."**

   The best way to work with lists in Oz is to use recursive functions. Those functions receive a list as input and have to apply a given computation on each element of the list. Unfortunately some students consider the list as a variable on which we can apply directly those computations. In fact, they should apply it on each element of the list, one after the other.

8. **"Error: You have an error due to one of the variables."**

   Unfortunately some mistakes can raise other ones inside the submission. When the interpreter detects a variable that could raise multiple mistakes, it displays both the variable and its value.

9. **"Error: You are trying to access the field of an element which is a cell. You probably want to access the value of the cell which is a record. Try with the " @ " operator."**

   In some cases, a cell contains a record. To handle this record, the students should not forget to use the " @ " operator because it corresponds to the value stored by the cell.

10. **"Error: This variable is not storing a record. You cannot get access to a field."**

    If the variable does not store any record, then it does not contain a field either. Therefore, the student should be warned that he is trying to access a non-existent field.

11. **"Error: The record does not contain the field you are looking for."**

12. **"Error: You have a deadlock, your variable is not initialized and you only have one thread."**

13. **"Error: You are trying to perform a pattern matching on a cell. Are you sure you did not forget to use the " @ " operator to perform the pattern matching on the value of the cell ?"**

    Once again, the value stored in a cell should always be accessed with the corresponding operator.

14. **"Error: You are trying to call a method which is a cell. did not you forget to access the value of the cell thanks to the " @ " operator ?"**

15. **"Error: You want to iterate over the value of a cell. Are you sure you did not forget to use the " @ " operator ?"**

16. **"Error: The condition is not a boolean."**

    Some students use the result of a function as a conditional boolean. However, those functions do not always return a boolean value.

17. **"Error: You cannot use this predefined function."**

    In some exercises, the students are asked to implement their own version of an existing function. Clever as they are, they try to use the existing one instead of implementing their own one. This should be forbidden.

18. **"Error: The argument of Length is neither a list nor a string."**

19. **"Error: The argument of " List.last " is not a list."**

20. **"Error: You have an error when calling Nth. The argument is not a list."**

21. **"Error: You have an error when calling Nth. The element you want to access is out of range."**

22. **"Error: One of the two arguments given to the function Append is not a list."**

23. **"Error: Not enough argument for the procedure."**

24. **"Error: Too many arguments for the procedure. "**

25. **"Error: The procedure you want to call is not declared."**

26. **"Error: This procedure/function/variable already exists. Maybe the procedure you want to implement is already implemented. Otherwise, change its name."**

    As observed previously, the students have some trouble understanding the statement of the exercises. In some cases, we asked them to provide the body of a function but not its signature. Some students keep providing this signature which corresponds to a second declaration according to the interpreter.

27. **"Error: You should use the general pattern matching as the last case of your case statement. Otherwise, it will match every time and the rest of the case will be considered as a dead code, a piece of code that can never be reached."**

    Pattern matching engineering should be handled with great care or using a too general pattern would result in dead code for other patterns below it.

28. **"Error: You have to use some list structure in this exercise."**

    Some exercises only require to use a list structure to be successful.

29. **"Error: You have to use some cell structure in this exercise."**

30. **"Error: Some functions are not tail recursive."**

    Tail recursion is an essential notion in LOUV1.1X. The students are explicitly asked to use tail recursion in some exercises.

31. **"Error: Your program runs infinitely, it can be from one of those parts of the program."**

    When applying tail recursion with lists, the students often implement code that can run infinitely because they forget to update a variable or they call the function on the wrong argument. This message displays the different parts of the program that can be responsible for the overflow.

32. **"Error: One of the term is not a number nor a boolean."**

33. **"Error: Use nil to represent a empty list not " [ ] "."**

34. **"Error: The argument is not a record."**

4.3 Hint markers

1. **"Hint: An " else " statement is missing in your conditional if statement. You should provide a default case."**

2. **"Hint: The case statement does not match any pattern for the expression. You should have an else statement."**

    The students do not always consider all the potential patterns and end up with a bad case statement. A default pattern should always be implemented.

3. **"Hint: You call a function several times with the same argument. Maybe you forgot to update it."**

   This marker is arguable. In some cases, it will be very useful: e.g. when a student keeps calling a recursive function with the same argument, the interpreter is able to deliver this feedback because it detects an infinite loop. However it can report false mistakes in some other cases: e.g. when a student calls a function that performs recursive call with the same argument without the possibility to store the value in a variable.

4. **"Hint: You seem to have declared too many variables for the exercice. Probably some variables are not needed."**

5. **"Hint: You seem to have declared too many procedures for the exercice. Maybe some procedures are not needed."**

6. **"Hint: You seem to have declared too many functions for the exercice. Maybe some functions are not needed."**

7. **"Hint: You seem to have performed too many procedure calls for the exercice. Try to delete some by storing the result in a variable or by deleting some useless procedures/functions."**

8. **"Hint: You do not use enough threads for the exercise."**

9. **"Hint: You do not have the right number of classes for the exercise."**

10. **"Hint: You do not use enough 'if' statements for the exercise. We expect you to use at least X if statement(s)."**

11. **"Hint: You do not use enough case statements for the exercise."**

12. **"Hint: You do not have a 'case <var> of nil' statement. You should at least have one to succeed this exercise."**

13. **"Hint: You do not have a 'case <var> of leaf' statement. You should at least have one to succeed this exercise."**

14. **"Hint: the stack size is too big for the exercise. Try to use a tail recursive function."**

15. **"Hint: You have to use at least one list structure in this exercise."**

16. **"Hint: You have to use the bottom expression in this exercise."**

17. **"Hint: You have to use at least one cell structure in this exercise."**

18. **"Hint: Some functions are not tail recursive."**

5 Conclusion on submissions analysis

It is possible to formulate some ways of improvement after studying as many submissions. In fact, as stated above, many errors arise from a bad understanding of the exercise or from a bad understanding of the code to write. Therefore, we could improve the exercises proposed on INGInious in 2 ways:

- Provide a part or the full pre-defined code in which the student submission is integrated. Therefore, the students would be able to see easily the context in which they are implementing their answers. INGInious provide the required infrastructure to make this improvement possible.
- Finally, some exercises should be explained differently. A lot of students do not manage to complete an exercise because they do not get exactly the computation to perform.

But INGInious and the statements of the exercises are not the only issues. This is why a new tool is required to improve the overall learning on edX. This tool should implement the markers that have been made during the submissions analysis. The complete tool is discussed in the next chapter.

Chapter 4

CorrectOz

During the submissions analysis, we had to reveal the common mistakes performed by the students. The goal was to find new and better ways to provide them a complete and accurate feedback. The solution was to refine those common mistakes into markers. Then, those markers would be implemented into an interpreter. This interpreter would allow to perform a complete review of the submissions at run time. We called this final product, e.g. our feedback provider, CorrectOz.

The development of an interpreter is a long and tough task that requires some prerequisites and a careful attention. This chapter discusses about them: (1) describes the operation of the selected parser, (2) provides a complete description of the interpreter itself and (3) ends the discussion with a brief conclusion.

1 The parser

The first step for interpreting a submission is to parse its content. Parsing corresponds to the process of finding and recognizing a sentence in a stream of tokens. Each one of those tokens is made of a lexeme, a sequence of characters that is extracted from the initial content by a *lexer*. Parsing is the first process to find *syntax* errors. Indeed, a parser is built upon a predefined grammar that allows it to construct those sentences. If a sentence cannot be built from this grammar, then there is obviously a syntax error in the submission.

1.1 Motivations

When looking for a parser, two options are available for programmers:

1. **Implementing their own parser**: this requires a lot of time and effort but the reward can be very significant. In this case, the parser is entirely mastered and fit ideally to the programmers' needs which makes it even more efficient. It can also be developed to stay modular and remaining scalable over time.

2. **Opting for an existing parser**: here is a choice that should be taken with great caution. An existing tool can be very difficult, even impossible to adapt to the programmers' needs. Moreover, the tool can appear to be too limited in terms of supported grammars or execution time. However, the only waste of time is supposed to be the time spent to take control of the parser operation.

Going for an existing parser was a *safety* choice for us because we were confronted to a "major" project for the first time and we were afraid of being overloaded by the task. Then, our final choice revealed to be ANTLR4, described as "a powerful parser generator for reading, processing, executing, or translating structured text or binary files. it is widely used to build languages, tools, and frameworks. From a grammar, ANTLR generates a parser that can build and walk parse trees" [9]. Many well-known companies use ANTLR to achieve some of their tasks. For example, TWITTER search uses ANTLR for query parsing, HIVE, PIG and HADOOP also use this parser. What really worked in its favour was the easy-to-use feature pinpointed for the tool by its regular users, despite some lack of performances. We discuss the encountered performance issues later in the paper.

1.2 ANTLR4

1.2.1 Grammar

Each language has its own grammar. It is a necessary input that provides the regular expressions and syntactic rules required to form both tokens and sentences from those tokens. The Oz grammar is described by Prof. Peter VAN ROY and Prof. Seif HARIDI in [10]. We discuss some rules in this section.

Lexical grammar Each token in Oz is described by a regular expression. In the grammar syntax supported by ANTLR4, each of those rules should start with an uppercase character. For example, integers should follow the rule presented in fig 4.1. According to this rule, an integer could be described by five different ways: (1) by a single digit, (2) by a number (a sequence of digits), (3) by an octal number, (4) by an hexadecimal number, or (5) by a binary number. The '$\sim$' character can be used to express negative integers. This is an optional character as described by the surrounding interrogation mark. There are also four other special characters that are used in this description: (1) | means "or", (2) + means "1 or more", (3) $*$ means "0 or more". An integer token is defined by several other lexical rules. Those rules are useful to split the input stream of characters into different sub-tokens that facilitate the distinction between an integer, an identifier or any other token. Those rules themselves can also use other rules as shown by *Hexdigit*.

But the Oz language has many keywords as well as other programming languages. Those keywords are also considered as tokens and should respect the regular expression described in fig 4.2.

```
Integer:    ('~')?(DIGIT)
            | ('~')? Nzdigit (DIGIT)*
            | ('~')? '0' (Octdigit)+
            | ('~')? ('0x' | '0X') (Hexdigit)+
            | ('~')? ('0b' | '0B') (Bindigit)+
            ;

DIGIT       : [0-9]
            ;

Nzdigit     : [1-9]
            ;

Octdigit    : [0-7]
            ;

Hexdigit    : DIGIT
            | [a-f]
            | [A-F]
            ;
```

Figure 4.1: Regular expressions for integers

```
<TOKEN >:   '<KEYWORD >' ;

CASE:           'case';
IF:             'if';
```

Figure 4.2: Regular expression for keywords

Formal grammar Regular expressions allow to reveal the tokens from the input stream of characters. The next task is to form sentences from those extracted tokens. Each sentence should follow the syntactic rules defined by the grammar. In ANTLR4, this grammar is written in EBNF notation which stands for *Extended Backus-Naur Form*. It is used to express context-free grammar and consists of terminal symbols and non-terminal production rules. The production rules restrict how the terminal symbols can be combined to produce legal expressions. The different operators used in EBNF are provided in fig 4.3. Unfortunately, the Oz grammar described in [10] does not follow the ENBF form and adaptations were required to implement it inside an ANTLR4 environment. Indeed, the tool generates only leftmost derivative parsers which require to avoid left recursion [11]. ANTLR4 parser can deal with left recursion within the same rule, but not in between multiple rules. Therefore, it was required to add some new rules and adapt others by using techniques such as substitution or left factoring [12].

*	repetition
-	exception
,	concatenation
|	definition separator (choice)
=	definition
;	termination
.	termination (2nd form)

Figure 4.3: Operators in EBNF

Fig 4.4 provides the syntactic rule defining a *nested declaration* for the Oz language in EBNF notation. This is the result of the translation that was performed from the initial syntactic rule described in [10]. In this figure, the first line states that a nested declaration can be a procedure definition. In Oz, a procedure can be declared with the keyword *proc* followed by an opening bracket, a variable describing the name of the procedure, one or several patterns and a closing bracket. Then, those first characters are followed by an *inStatement* which respects its own syntactic rule and the keyword *end* to close the procedure definition. The next lines describe all the other kinds of nested declaration that exist in Oz.

```
nestDec  : PROC ('{')? variable ( pattern )*  ('}')? inStatement END
           | FUN ( LAZY )?  ('{')? '$'  (pattern)* ('}')? inExpression
              END
           | FUN ( LAZY )?  ('{')? variable  (pattern)* ('}')?
               inExpression END
           | FUNCTOR variable ( IMPORT ( variable (AT atom)?
           | variable  '( ' ((atom | integer)( ': ' variable)?)+ ')' ) )
               ( EXPORT ( ( (atom | integer)  ': ' )? variable )+ )?
               DEFINE ( declarationPart )+ ( IN statement )? END
           | CLASS variable (classDescriptor )* ( METH methHead
              ('=' variable )? ( inExpression | inStatement ) END)* END
           ;
```

Figure 4.4: Syntactic rule defining a nested declaration in EBNF notation

Fig 4.5 (resp. fig 4.6) provides a single-line part of the syntactic rule for a *nested condition* (resp. *expression*) as described in [10]. Those pieces of rule are left recursive with respect to each others. As stated above, left recursion is not supported by ANTLR4. By observing the whole grammar provided in [10], the *nested condition* rule appears to be used only by the syntactic rules *statement* and *expression*. Therefore, the call to *NestCon* inside those syntactic rules can be directly replaced by the piece of rule described in fig 4.5 in order to remove the left recursion. For more examples, the entire

translation of the grammar can be found on the git given in appendix **??**.

NestCon::= <expression> ('=' | ':=' | ',') <expression>

Figure 4.5: Part of the syntactic rule for a nested condition

<expression> ::= NestCon

Figure 4.6: Part of the syntactic rule for an expression

1.2.2 Lexer and parser

The parsing of a submission involves the successive work of a *lexer* and a *parser*. The lexer should reveal the sequence of characters that form the tokens and the parser should group those tokens into correct sentences. Those tokens and legal sentences have been described in the previous section through the grammar definition. This subsection focuses on the operation of the generated lexer and parser by ANTLR4 according to this grammar.

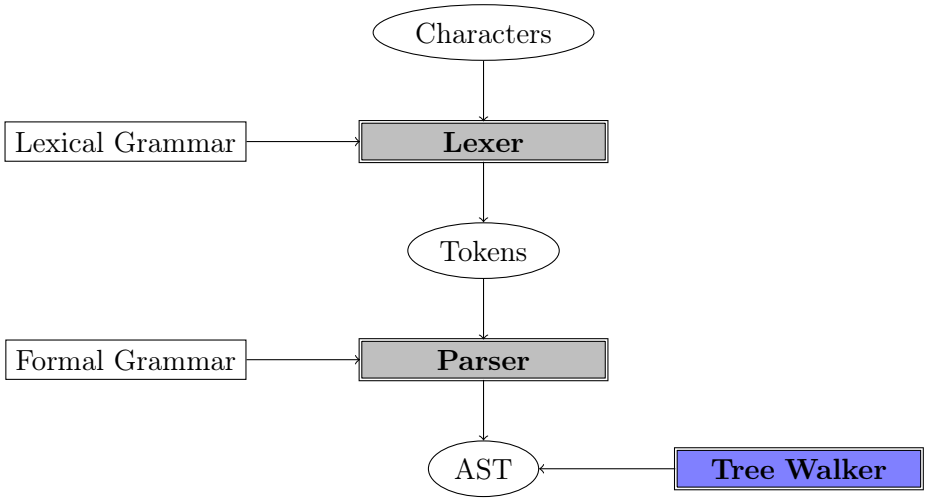


Figure 4.7: Lexer and parser workflow

Lexical analysis The lexer is in charge of the lexical analysis. As shown in fig 4.7, it corresponds to the action of splitting the input stream of characters into a sequence of corresponding tokens from the lexical grammar.

Fig 4.8 provides an input sequence of characters as an example. According to the lexical grammar described previously, the lexer should identify six different tokens: three keywords (*local*, *in* and *end*), one variable (*X*), one operator (=) and one integer (*42*).

local X in X = 42 end

Figure 4.8: A sequence of characters

Syntactic analysis Once the tokens have been identified by the lexer, the parser should group those tokens to create legal sentences, also called *syntactic entities*. There are many possible syntactic entities in Oz such as expressions, statements, procedures or classes. Those entities are defined by the formal grammar described previously. As shown in fig 4.7, the output of the parser is an AST, an *Abstract Syntax Tree* which is a tree representation of the parsed program.

Fig 4.8 describes an input sequence of characters. The lexer extracted six tokens from this sequence and the parser should now map them to a *nestConStat* entity from the formal grammar. This entity is built on top of two other entities: a *declarationPart* and a *statement*. The interesting piece of definition for each of those entities is provided in fig 4.9. "Integer" with an uppercase corresponds to the token studied in the lexical grammar subsection whereas "integer" with a lowercase corresponds to the syntactic entity. Finally, the AST produced by the parser for this sequence of tokens is drawn in fig 4.10.

nestConStat:	LOCAL (declarationPart)+ IN (statement)? (statement) END
declarationPart:	variable
statement:	expression ('=' | ':=' | ',') expression
expression:	term
term:	integer | variable
integer:	Integer

Figure 4.9: Some syntactic rules

1.2.3 Lexer and parser in ANTLR4

ANTLR4 provides very interesting features to deal with lexical and syntactic analysis. In fact, ANTLR has the ability to generate both lexer and parser based on a complete grammar. This grammar should follow the rules described previously and should be stored in a ".g4" file. The lexer and the parser can be generated in Python or Java. We decided to use Java for performance purpose (Python will not be supported for a long time). But ANTLR has another great feature that makes it a powerful tool: it provides its own *tree walker* as shown in fig 4.7. This tree walker supplies several classes and interfaces to visit the ASTs output by the parser. Fig 4.11 provides an overview of the

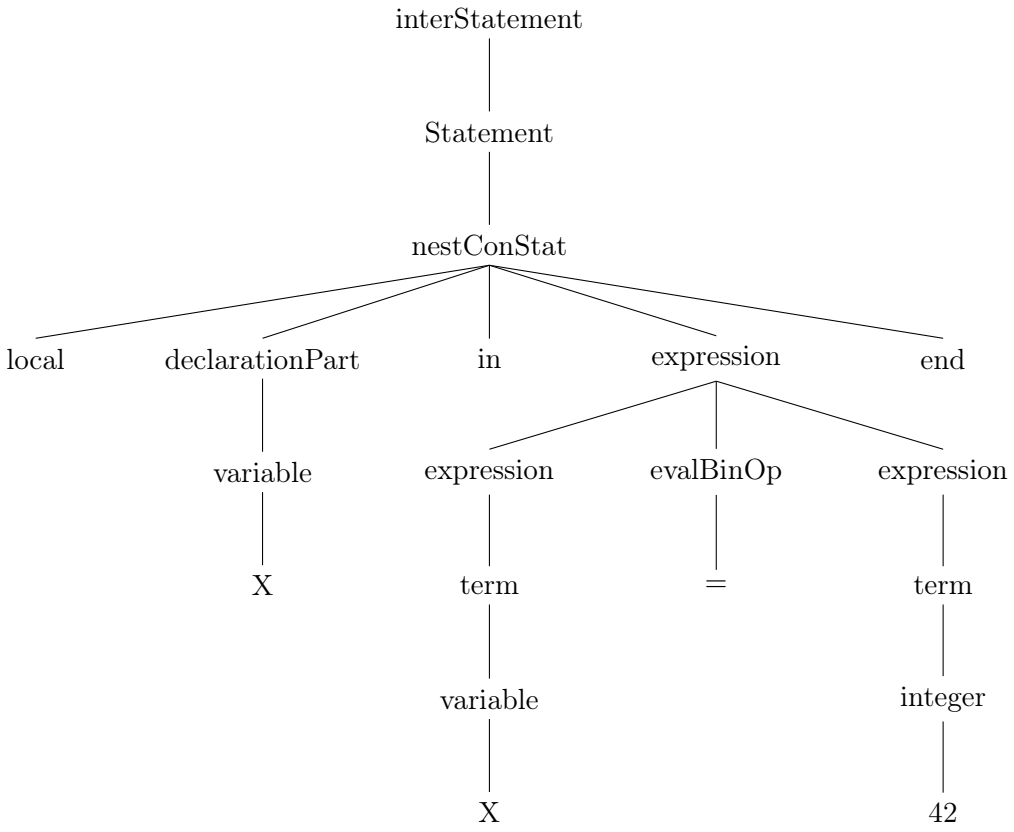


Figure 4.10: AST generated by the parser from the input at fig 4.8

files created by ANTLR when generating the lexer and the parser for the Oz language. Those files are discussed below.

OzBeta is the name we gave to the translation of the Oz grammar into the EBNF form supported by ANTLR. It contains both regular expressions from the lexical grammar and syntactic rules from the formal grammar. ANTLR4 has the ability to make the difference between both kinds of definition.

OzBetaLexer is a JAVA file that contains the lexer class definition generated by ANTLR from the lexical rules and the grammar literals of OzBeta. This lexer is fully able to extract the tokens from any piece of code written in Oz.

OzBetaParser is a JAVA file that contains the parser class definition generated by ANTLR from the syntactic rules of OzBeta. The tool creates one class for each rule in the grammar.

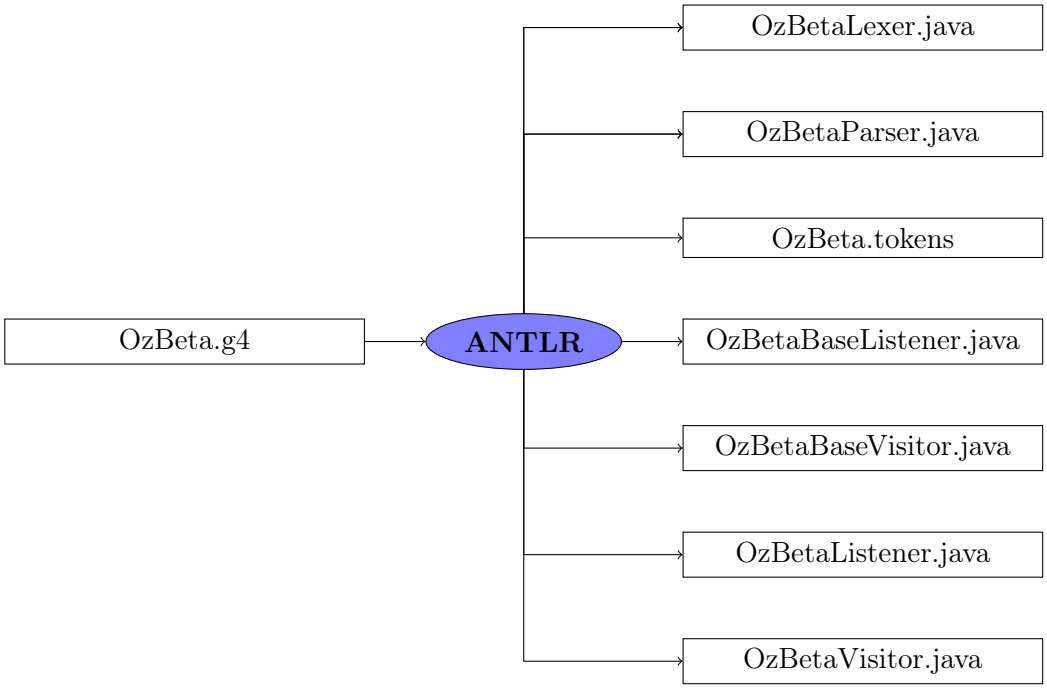


Figure 4.11: ANTLR generated files from the input grammar

OzBeta.tokens is a special file that stores the token number associated with each token in the grammar.

OzBetaBaseListener is a file that provides a first mechanism to walk the ASTs output by the OzBetaParser. Each syntactic rule has two fire events in this listener: (1) one when entering the entity and (2) one when exiting the entity. Those events can be modified. The only requirement is to respect the **OzBetaListener** interface when overriding those events. The interpreter overrides those events to verify the correctness of the syntax. This step is discussed in the next section.

OzBetaBaseVisitor is the file that provides a second mechanism to walk the ASTs output by the OzBetaParser. Each syntactic rule has its own visitor method. A visitor can perform some code and then visit its children. The strategy implemented by each visitor can be modified to walk the tree differently. The only requirement is to respect the **OzBetaVisitor** interface that has been generated. Notice that each submission in Oz starts with a *interStatement*. This entity has children in the AST that can be visited, those children have children themselves and the entire tree can be visited step by step. Each visitor can also be modified to simulate the execution of the submission in Java. This is the interpreter job which is discussed in the next section.

ParseTree is a JAVA object provided by ANTLR4 that represents each node in the ASTs. Through this object, each node has several methods that can be used in the OzBetaBaseVisitor, the OzBetaBaseListener or the interpreter itself:

- ParseTree getChild(int index): returns the child at index.
- int getChildCount(): returns the number of children.
- ParseTree getParent(): returns the parent.
- String getText(): returns a string representation of the actual node.
- Object visit(ParseTree child): visit the child provided in argument.
- Object visitchildren(ParseTree parent): visit all the children of the node provided in argument.

Grun ANTLR4 also provides a way to show the ASTs in a graphical interface. This is a very useful feature to understand and visualize the overall operation of the parsing methods.

1.3 Conclusion on ANTLR4

ANTLR4 appears to be a powerful tool that fits perfectly the interpreter needs. Now it provides an efficient way to parse the submissions written in Oz and it provides adaptable mechanisms to visit the ASTs that result from this parsing. The adaptability of those mechanisms is the key idea that allows the interpreter to simulate the execution of the submissions in JAVA. The interpreter is discussed in the next section.

2 The interpreter

The parser generated by ANTLR4 is able to parse any submissions written in Oz. The final step is to visit the AST produced through the interfaces provided by ANTLR and described in the previous section. The objective is twofold and consists of performing both *syntactic* and *semantic* analysis to outperform the previous grading tools. The interpreter is responsible for those tasks. It is discussed in this section.

2.1 Syntactic analysis

The first step before *semantic* analysis is to identify the potential syntax errors. INGINIOUS usually returns the syntax errors provided by the MOZART compiler. Unfortunately, those feedback are inaccurate and lack precision for beginning students. ANTLR4 provides interesting features to walk the AST in order to identify those syntax errors. Those features are provided by the interface *OzBetaListener* which was first implemented by *OzBetaBaseListener* as described in the previous section. The idea is to override the entering events of the main syntactic rules such as *expression*,

nestConStat and *statement* to name but a few. Once triggered, each event should verify the correctness and the completeness of its syntax according to the related syntactic rule. Some events should also keep up to date the count of some parameters: if statements, case statements, declared variables,...

Fig 4.12 provides a complete example based on the implementation of the *nested declaration* entering event. This syntactic rule defines the existing syntactic ways to declare a procedure or a function. It takes as input a *OzBetaParser.NestDecContext*. This context stores the informations related to the node in the AST. Fig 4.13 shows one potential structure for this context. In this case, the context is a procedure called "JustReturn" that simply returns the argument received in input.

Let's focus on the event implementation:

- First, it verifies if the first child corresponds to the string "proc" or "fun". According to the syntactic rule, the first child should indeed be one of those tokens.
- The second child should be an opening bracket. If the student forgot this character, then the interpreter will deliver the 54th marker of error. The markers can be found in the file called *markers.txt*. The 54th corresponds to the following one: "A " { " is missing in your procedure/function declaration". This is a first example of precise and accurate feedback on a syntax error.
- The procedure/definition signature should be closed with a closing bracket. Otherwise, the interpreter will output the 55th marker: "A " } " is missing in your procedure/function declaration.".

2.2 Semantic analysis

Once the potential syntax errors have been identified, it is finally possible to perform the *semantic* analysis. This final step consists of simulating the execution of the submission by walking the AST a second time. Indeed, each sentence has a meaning in the language and the interpreter should apply this meaning to each one of the parsed sentences in order to simulate the execution of a submission. As explained previously, the interface *OzBetaVisitor* provides another way to visit the ASTs. The class *OzBetaBaseVisitor* was described earlier as one possible implementation of this interface. Through those methods, it is possible to visit the children of a syntactic rule. Those children correspond to its sub-rules. For example, the previously observed *nestConStat* has two sub-rules: *declarationPart* and *statement* as described in fig 4.9. Those sub-rules have their own sub-rules that can be visited as well. Each visitor returns an object according to its implementation. *visitInteger* returns only a JAVA integer but *visitExpression* can return a boolean, a record, a list or even a cell, for example. Those visitors have been overridden such that the interpreter is now able to simulate the execution, in JAVA, of the parsed submissions.

```
1     @Override public void enterNestDec(OzBetaParser.NestDecContext ctx) {
2         if( ctx.getChild(0).getText().equals("proc") ||
3             ctx.getChild(0).getText().equals("fun")) {
4
5             String procName = ctx.getChild(2).getText();
6
7             if(!ctx.getChild(1).getText().equals("{")) {
8                 procName = ctx.getChild(1).getText();
9                 CustomErrorListener.addError(ctx,
10                OzInterpretor.vectorsArray.get(53));
11            }
12
13            else if(!ctx.getChild(ctx.getChildCount()-3).getText().equals
              ("}")) {
14                CustomErrorListener.addError(ctx,
15                OzInterpretor.vectorsArray.get(54));
16            }
17        }
18    }
```

Figure 4.12: Implementation of the nested declaration entering event

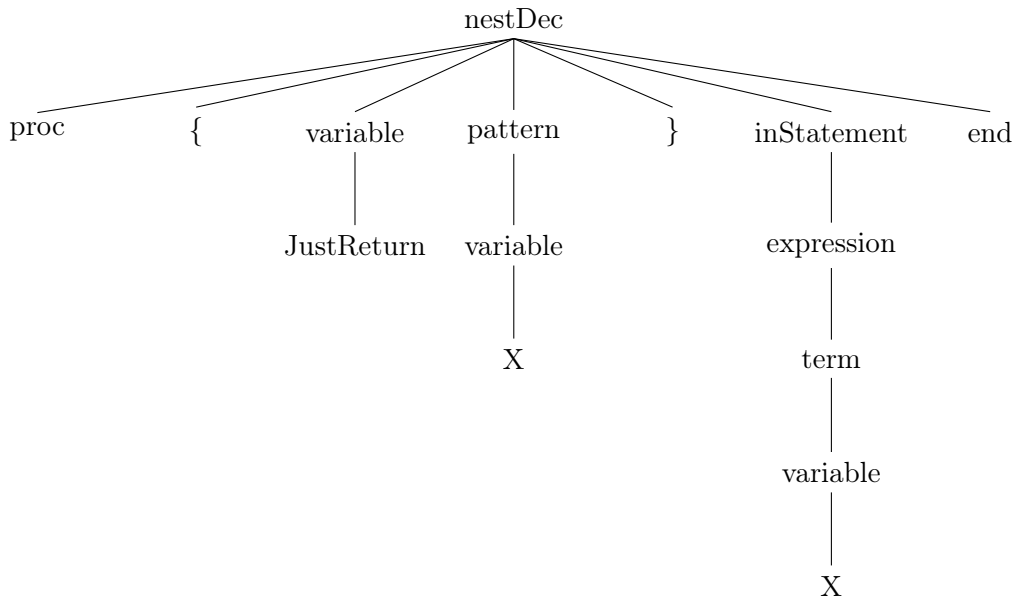


Figure 4.13: Tree representation of a potential NestDecContext

Fig 4.14 provides the complete code of the *visitInteger* method. This corresponds to the visitor of the *integer* rule as it was overridden by the interpreter. This method receives as input a *OzBetaParser.IntegerContext* which contains the informations related to the

node in the AST. Let's focus on the implementation of the method:

- First, the upper element of the stack is peeked and the environment is extracted from this element. The stack stores tuples and each tuple is a pair of element. The first element corresponds to a *ParseTree*, a part of the AST that is being simulated by the interpreter. The second element corresponds to the current environment, the binding between the identifiers and the variables that have been declared until this point of execution. In Oz, the identifiers correspond to the name of the variable in the code. For more informations on the environment, please refer to the course LOUV1.1X explained in appendix **??**.
- The first child of an *IntegerContext* is the value itself which is stored as a String. The 6th line shows how to retrieve this value from the context.
- Then, a try-catch statement tries to parse this extracted String value into an integer. If the parsing is possible, the value is returned and the execution of the integer rule has been completely simulated. Note that the student might have declared a negative value. Those values began with a "~" character in Oz. In this case, the statement tries to parse the extracted String value without the special character. If the parsing is successful, the negative integer is returned.

```
1     @Override public Object visitInteger(OzBetaParser.IntegerContext ctx)
       {
2         Tuple oldTuple = (Tuple) stack.peek();
3         Map<String,String> environment = new HashMap<String,String>(
4         oldTuple.getEnvironment());
5
6         String myInt = ctx.getChild(0).getText();
7
8         try{
9             if(myInt.charAt(0)==('~')) {
10                return -1*(Integer.parseInt(
11                myInt.substring(1,myInt.length())));
12             }
13         else      return    Integer.parseInt(myInt);
14        }
15        catch(NumberFormatException e) {
16        }
17
18        return myInt;
19    }
```

Figure 4.14: Modified visitorInteger method in the interpreter

This piece of code explains the overall operation of the interpreter. It simulates the execution of each rule in order to reveal the semantic mistakes. If an issue has been

detected by the semantic analysis, the interpreter will provide the corresponding marker as a feedback. In this example, no marker can be provided as feedback because the integer rule is the most simple one within the grammar. However, some visitor methods may contain several hundred lines of code to cover all the potential executions of a rule. Those visitors may detect and return a dozen of different markers. For clarity and length, we did not provide them as example.

2.3 Markers as feedback providers

The objective of submissions analysis was to provide a list of common mistakes in order to find efficient ways to provide a better feedback to the students. The idea at that time was to refine those common mistakes into markers. Those markers were designed to be directly implemented into the interpreter. They should be displayed as soon as a mistake has been detected in a submission. Therefore, those markers stand as the primary way to provide feedback to the students.

ErrorListener is a generic interface provided by ANTLR4 to report the parsing errors revealed by its generated parsers. This interface is generic because it is not related to a particular grammar.

BaseErrorListener is a first example of implementation for the interface *ErrorListener*. This class is able to output rather generic errors from the parsing errors.

CustomErrorListener extends *BaseErrorListener* to provide more accurate feedback and fit to the interpreter needs. This class is made of two ArrayLists of String that contains the reported mistakes. The first ArrayList stores *error* messages. An error is a problem that prevents the program from running correctly. The second ArrayList stores *hint* messages. Those hints correspond to a series of good practices that promote students' learning. Those ArrayLists are filled with the markers content during the interpretation of a submission. At the end of it, they are emptied and provided as feedback.

CustomErrorListener has 3 essential methods:

1. **addError**: both syntactic and semantic analysis described previously report their errors to this method. It takes four arguments: (1) the statement where the error occurred, (2) the marker content related to the error, (3) the variables involved in the error and (4) an optional message. Some auxiliary methods exist to report an error with less arguments if required.

2. **addHint**: the good practices that should be provided during both syntactic and semantic analysis are reported to this method. It takes the four same arguments and also exists with less arguments if required.

3. **syntaxError**: unfortunately, it was not possible to predict all the potential syntax errors. If no marker exists for a syntax error, the parser reports it to this method

that provides a generic feedback with the line and the column where the error occurred.

Fig 4.15 provides a complete example of error reporting. The first part of the figure is a submission in which multiple-assignment is performed. The next piece of code corresponds to the part of the interpreter that reports the error to the *CustomErrorListener*. The first argument is the context, the node in the AST which corresponds to the part of the execution in which the error occurred. In this case, the node corresponds to "X=3". The second argument corresponds to the first marker of error and the third argument to the variable involved in the issue (in this case, "X"). The last part of figure 4.15 provides the complete feedback output by the *CustomErrorListener*.

```
1    local X in
2      X = 2
3      X = 3
4    end
```

```
1    String [] variables = {ctx.getChild(0).getText()};
2    CustomErrorListener.addError(ctx,vectorsArray.get(1),variables);
```

ERROR: You can not change the value of the variable because it was already assigned to one. A variable can only be assigned to one value in Oz. Use cell with " := " for multiple assignments. The variable(s) concerned: [X]. Error found in "X = 3".

Figure 4.15: Example of error reporting

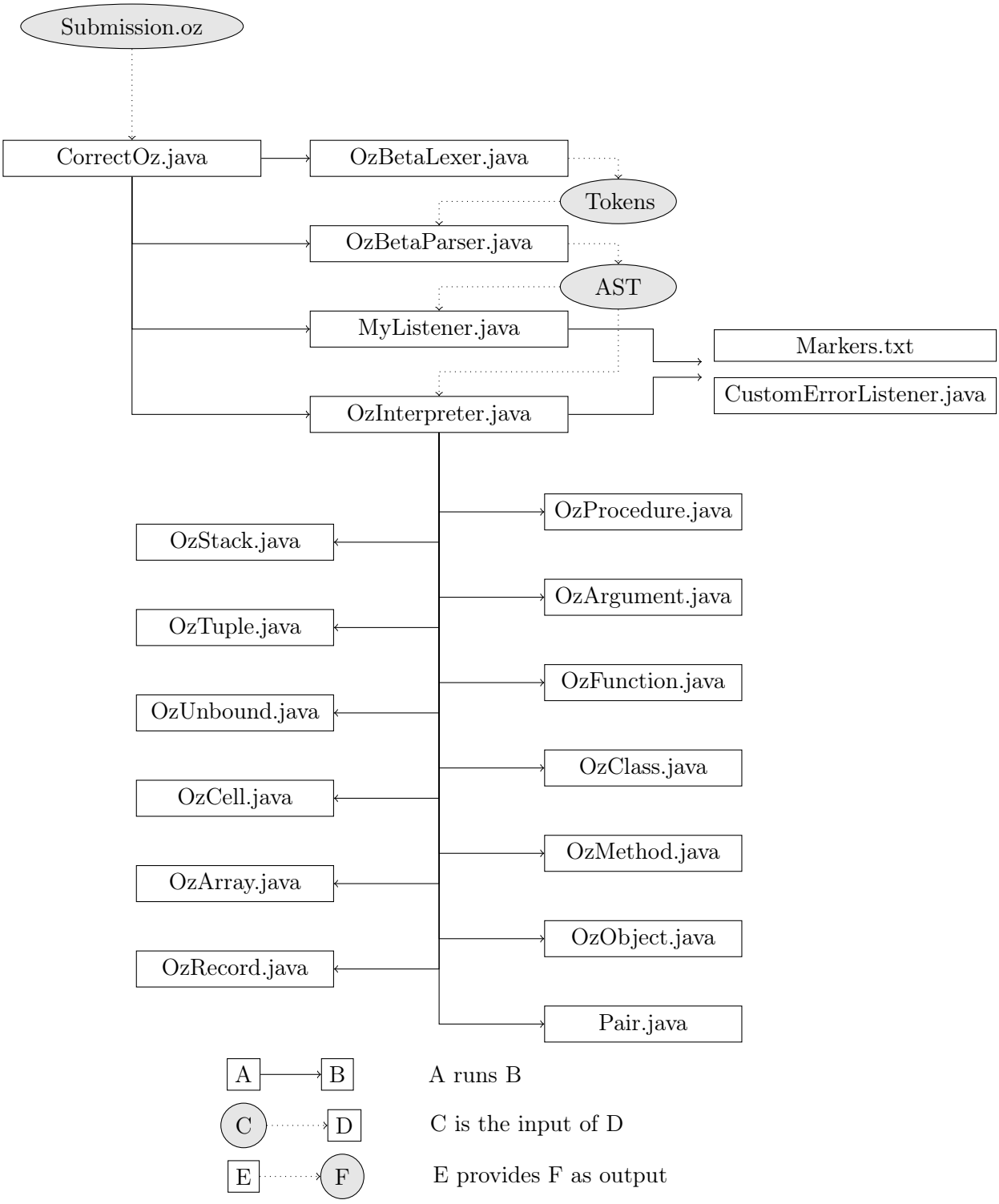


Figure 4.16: Architecture of the interpreter

2.4 Implementation

This section describes the overall architecture of the interpreter. It also discusses the main implementation choices that have not been addressed previously. Fig 4.16 provides a first overview of the presented architecture.

CorrectOz is the core file of CorrectOz. It receives the instructions to start the grading of a submission and forwards those instructions to the other classes. This file is responsible for parsing the input submission through the *OzBetaLexer* and the *OzBetaParser* generated by ANTLR4. Then, it is responsible for running both syntactic and semantic analysis and printing the selected feedback.

Markers The markers of error described in chapter 3, section 4 are listed into a unique textual file. Therefore, it is trivial to add or modify the error messages, e.g. for translation purpose or future improvements. Each marker should simply start with a unique integer followed by the message that should be provided as feedback. *CorrectOz* is responsible for loading the list of markers before starting the syntactic and semantic analysis. Then, those markers can be added or modified between two interpretations.

SyntaxVerifier extends *OzBetaBaseListener* provided by ANTLR4. It is responsible for supervising the syntactic analysis. Therefore, it will mostly report syntax errors and a few hints to improve the submission. Its implementation has been discussed in subsection 2.1.

OzInterpreter extends *OzBetaBaseVisitor* provided by ANTLR4. It is responsible for simulating the execution of a submission. In order to perform this task, it uses many other classes that reflect the implementation of the data structures and concepts in Oz. Its overall implementation has been discussed in subsection 2.2.

CustomErrorListener extends *BaseErrorListener* which implements *ErrorListener* provided by ANTR4. The class collects and manages the errors provided by both syntactic and semantic analysis. Its implementation has been described in subsection 2.3.

OzStack extends the so-called *Stack* data structure provided by JAVA. The stack is an essential concept to implement because of tail recursive functions that are studied during LOUV1.1X. As explained previously, the stack stores tuples. It extends the JAVA stack with the ability to get the size of the data structure at any moment of execution. It is pushed, popped or peeked according to the *OzInterpreter* needs to simulate the execution of a submission.

OzTuple is a data structure implemented to fit the interpreter needs. Each tuple stored by the stack is a pair of element (*ParseTree*, environment). As explained previously, the *ParseTree* corresponds to the part of the AST that is being simulated

by the interpreter. This data structure is entirely provided and handled by ANTLR4. The environment corresponds to the binding between the identifiers and the variables that have been declared until this point of execution. Those bindings are stored inside a *HashMap* data structure. Fig 4.17 provides an example of environment with 3 declared identifiers: X, Y and Reverse. The variables x0, y3 and reverse0 correspond to the *store variables* that are explained in the next paragraph.

$$\{X \mapsto x0, Y \mapsto y3, Reverse \mapsto reverse0\}$$

Figure 4.17: Example of HashMap environment

Store The store is another *HashMap* data structure. It binds each store variable to its corresponding value. The triplet {identifier $\mapsto$ store variable $\mapsto$ value} is an essential concept in Oz. Fig 4.18 describes a potential store related to the environment provided in fig 4.17. In this example, the variable x0 is bound to the value 1 which means the identifier X has the value 1 in the related code. The identifier Reverse corresponds to a function definition because it is bound to the store variable reverse0 which is itself bound to the function definition. The store is entirely initialized and managed by *OzInterpreter* and it is not defined in a separate file.

$$\{x0 \mapsto 1, y0 \mapsto \text{'|'}(1 : a2 : list2), reverse0 \mapsto fun(reserve) < body > end\}$$

Figure 4.18: Example of HashMap store related to fig 4.17

OzUnbound is a simple data structure instantiated when an unbound variable should be declared in the store. This data structure is defined in a separate file because it represents a Oz-based concept.

OzCell implements a data structure that supports multiple-assignment in Oz. In JAVA, this feature is, of course, trivial. However, the class is defined in its own file because it represents a Oz-based concept.

OzArray represents the implementation of an Array in Oz. The data structure was implemented for differentiation purpose between JAVA Array and Oz Array.

OzRecord implements an essential data structure in Oz. This concept is called *record*. Each one has a label and a set of values. The label is the name of the record and each value is stored in a pair (feature, value). In the interpreter, a record object is made of two data structures: a String to store its label and a *HashMap* to store the binding between the features and the values.

Pair is a short class that implements the concept of a pair: any two *Objects* grouped within the same data structure.

OzProcedure The interpreter should be able to provide an accurate feedback when a submission involves an issue with a procedure call. Therefore, an *OzProcedure* class has been implemented to address these potential issues. A procedure has a String name and an *ArrayList* of *Arguments*. The *OzProcedure* class is able to provide a feedback on the number of times each procedure is called at run time. Therefore, the interpreter can provide a feedback on two kinds of mistakes:

1. **If an overflow occurs:** for example, when a recursive call keeps being applied to the entire list instead of its tail. Then, no progress can be achieved by the recursive function and the student should be aware that his submission has an overflow.

2. **If a value should be stored:** instead of performing multiple calls to a function with the same arguments, the student should store the value returned by the first call in a variable in order to avoid recomputing this result afterwards.

OzArgument The *ArrayList* of arguments stored by a *OzProcedure* is filled with objects of type *OzArgument*. Each of these arguments has four attributes: (1) the name of the related procedure, (2) its index among the procedure arguments, (3) its String representation and (4) its value. Fig 4.19 provides an example of a procedure call with 3 arguments. This procedure is stored as an *OzProcedure* object called "Sum" that has an *ArrayList* of three *OzArguments*. In this array, the first argument has the following attributes: (1) procedure name = "Sum", (2) index = 1, (3) String representation = "A" and (3) value = 1.

```
1    local A B Result in
2      A = 1
3      B = 2
4      {Sum A B Result}
5    end
```

Figure 4.19: Example of a procedure call with 3 arguments

OzFunction is a class that represents the notion of function. In Oz, a function is defined by a body and the environment in which it is declared. The environment allows to keep track of the *free* identifiers, the identifiers declared outside the function. The body corresponds to a *ParseTree* such that it is possible to visit this tree and simulate the execution of the function. As explained previously, the environment is implemented with a *HashMap* that binds the identifiers to their store variables.

OzClass is a JAVA class that represents the notion of class in Oz. Each class has three attributes: (1) a name, (2) an *ArrayList* of fields and (3) a *Map* between the methods and their name.

OzMethod is a JAVA class that represents the methods stored by the *Map* of a *OzClass*. Each method is made of two *ParseTrees*, one for its signature and one for its body. This allows the interpreter to visit those trees in order to simulate easily the execution of the method.

OzObject is a JAVA class that represents an instance of a *OzClass*. Each instance has two attributes: (1) a String name that corresponds to the instantiated class and (2) a *HashMap* that binds the object attributes to their value.

Garbage collection *OzIntepreter* has the ability to perform a "Mark-and-Sweep" garbage collection. First, this algorithm marks the identifiers that are referenced by, at least, one environment on the stack. As a recall, the stack is made of tuples and each tuple consists of a pair (*ParseTree*, environment). Then, the algorithm removes the unreferenced variables from the store such that a store variable is referenced if and only if it is bound to a marked identifier.

Oz pre-defined functions *OzIntepreter* would not be complete if it did not implement some Oz pre-defined functions such as *Browse* or *Append* that are frequently called by the students during LOUV1.1X and LOUV1.2X. Those functions have been re-defined in JAVA and are called instead of the existing Oz functions by the interpreter. Unfortunately, some Oz functions might be missing.

3 Run CorrectOz

The full command line to run CorrectOz is provided in fig 4.20. This command can appear rather messy the first time. Indeed, many arguments can be provided in order to output the most precise and accurate feedback as possible. For clarity purpose, those arguments will be described one by one in the following list. Note that every argument has a default value, you should only provide the arguments for which you want to change the value. As an example, Fig 4.21 shows the command line for an exercise using lists, three functions, seven variables and for which the use of the *Append* function is forbidden.

- **FileName**: the path that leads to the Oz submission.
- **-d debug**: *true* if and only if debug prints are required. *False* by default.
- **-nVar NbrVar**: the number of variables required to succeed the exercise. *Maximum* integer value by default.
- **-nProc nbrProc**: the number of procedures required to succeed the exercise. *Maximum* integer value by default.
- **-nFun NbrFun**: the number of functions required to succeed the exercise. *Maximum* integer value by default.

- **-nCall NbrCall**: the number of procedure/function calls required to succeed the exercise. *Maximum* integer value by default.
- **-nThread NbrThread**: the number of threads required to succeed the exercise. *Maximum* integer value by default.
- **-nClass NbrClass**: the number of classes required to succeed the exercise. *Maximum* integer value by default.
- **-nIf NbrIf**: the number of if-statements required to succeed the exercise. *Maximum* integer value by default.
- **-nCase NbrCase**: the number of case-statements required to succeed the exercise. *Maximum* integer value by default.
- **-nilCase NilCase**: *true* if and only if at least one "nil" pattern matching should be performed. *False* by default.
- **-leafCase LeafCase**: *true* if and only if at least one "leaf" pattern matching should be performed. *False* by default.
- **-list List**: *true* if and only if at least one *list* data structure should be used. *False* by default.
- **-bottom Bottom**: *true* if and only if at least one "bottom" atom should be used. *False* by default.
- **-stackSize StackSize**: the maximum size that can be reached by the stack at run time. *Maximum* integer value by default.
- **-tailRec TailRec**: *true* if the functions should be tail recursive. *False* by default.
- **-extProc ExtProc**: list of the functions that cannot be used by the student. Each function should be separated by a comma and the entire list should be surrounded by brackets. No function is forbidden by default.
- **-cell Cell**: *true* if and only if a *cell* data structure should be used. *False* by default.

```
java CorrectOz [FileName] -d [debug] -nVar [NbrVar] -nProc [NbrProc] -
nFun [NbrFun] -nCall [NbrProcCall] -nThread [NbrThread] -nClass [
NbrClass] -nIf [NbrIf] -nCase [NbrCase] -nilCase [NilCase] -leafCase [
LeafCase] -list [List] -bottom [Bottom] -stackSize [StackSize] -tailRec
 [TailRec] -extProc [ExtProc] -cell [Cell]
```

Figure 4.20: Full command line to run the tool

```
java CorrectOz list.oz -d false -nVar 2 -nFun 3 -nCall 7 -nilCase true
-list true -extProc [Append]
```

Figure 4.21: Example of command line

4 Conclusion on CorrectOz

Implementing an interpreter to simulate the execution of a complete language is a tough and challenging task that requires time, application and rigour. ANTLR4 provided the right tools to start with. This parser generator has deeply influenced the architecture and the implementation choices of the interpreter. CorrectOz is now completely functional to interpret and to analyse almost any kind of Oz submissions. The next chapter discusses how it can be integrated into the current grader, INGInious.

Chapter 5

Integration into INGInious

At this point, the implemented tool is fully able to simulate the execution of a submission and to provide a feedback based on the common mistakes observed previously. CorrectOz was implemented with the idea of a future integration into INGInious. This integration is discussed in this chapter. First, the actual configuration, the different files and their usefulness for both Louv1.1x and Louv1.2x are presented. Then, the execution environment provider Docker is described. The files provided to INGInious and some exercises are also presented to better understand the integration of CorrectOz in the grader. Finally, a short conclusion ends the discussion.

1 Actual configuration

Before even trying to integrate CorrectOz into INGInious, it is essential to understand how the grader is currently configured for both Louv1.1x and Louv1.2x courses. Here is a complete description of the current configuration. The complete files can be found on our git given in appendix **??** .

task.sh contains the script that will be executed when a student sends his submission to the grader. First, it inserts the code into a pre-defined code used to perform unitary tests. Then, it instantiates a complete Docker environment to run the grading process. This process consists of compiling and executing the complete code. Once completely executed, the grader outputs the results in two different files : *out.txt* and *err.txt*. The first one contains the output of the code if no compilation nor runtime errors were found. Otherwise these errors are reported in the second file. Finally, it provides a feedback according to the content of both files.

task.oz corresponds to the pre-defined code in which the submission should be inserted.

insert_input.py is responsible for writing the submission into the pre-defined code on behalf of the main file *task.sh*.

compil_check.py contains the verifications that are performed in case of compilation error.

running_check.py contains the verifications that are performed in case of runtime error.

feedback.py provides the feedback to the student by analysing the content of *err.txt* and *out.txt*. In case of reported error, it starts the corresponding verification mechanism: *compil_ check.py* if a compilation error occurred, or *running_ check.py* if a runtime error occurred. The key idea for CorrectOz is to replace those verifications in order to provide a better feedback in case of error.

2 Configuration for CorrectOz

INGInious' developers provide a virtual machine along with the grading tool. This VM contains only the basic tools for running the initial configuration. Unfortunately, the configuration described in section 1 is specific to Louv1.1x and Louv1.2x. It requires the installation of a complete Mozart environment and the addition of the files discussed in the previous section. The time was missing to set up such a configuration. This section describes how the tool was integrated *alone* into INGInious. The last section discusses how the current configuration could be added later into this environment.

2.1 Docker

Docker provides the execution environments required by INGInious to grade securely each submission. Indeed, INGInious cannot trust the students' inputs and should therefore execute their submissions into separate containers. Each one of those containers should provide the right dependencies required to run the submissions. This is exactly what Docker is achieving. It is described as "[a tool that] allows you to package an application with all of its dependencies into a standardized unit for software development" [13]. In other words, Docker is a tool that automates the deployment of other tools inside a software container.

Set up the environment Docker provides a configuration file to setup the environment in which a submission will be graded. *CorrectOz* only needs Java and the full ANTRL4 library to simulate the execution of a submission. However, INGInious also needs Python to run some tasks. Fig 5.1 provides the complete configuration file required to set up an environment that will support the execution of the interpreter.

2.2 Files

In addition to the configuration file, Docker also needs the implementation of the interpreter and some files to know *how* to run the submissions inside the configured

```
FROM ingi/inginious-c-default
RUN yum -y install java-1.8.0-openjdk*
RUN yum -y install yum-utils
RUN yum-builddep -y python
RUN yum -y install epel-release
RUN yum - y install python34
RUN curl -O http://www.antlr.org/download/antlr-4.5complete.jar
RUN export CLASSPATH=" .:/antlr-4.5-complete.jar:$CLASSPATH "
RUN alias antlr4='java -Xmx500P -cp " /antlr-4.5-complete.jar:
$CLASSPATH "
org;antlr.v4.Tool'
```

Figure 5.1: Docker configuration file

containers. This section describes each file provided to INGInious in addition to the Docker configuration file. Fig 5.2 provides an overview of the interactions between those files.

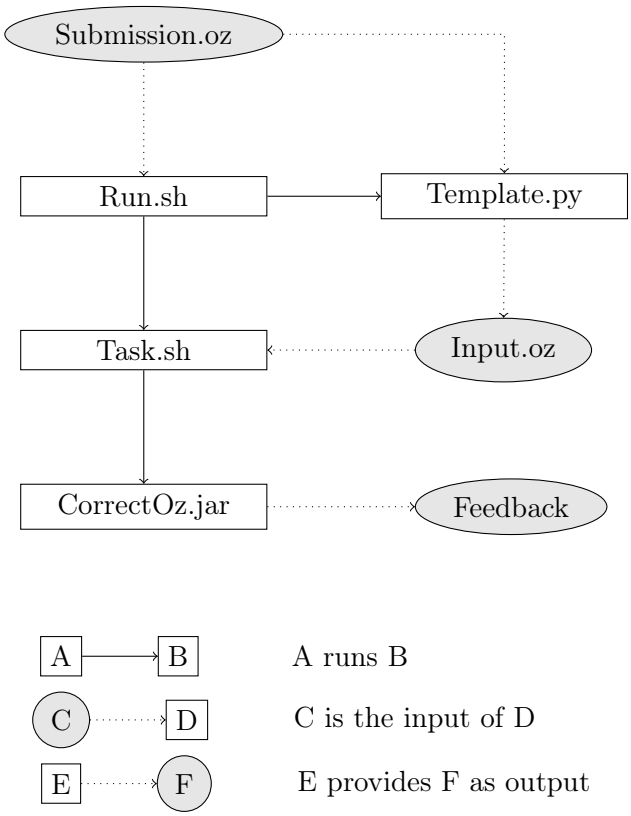


Figure 5.2: Interactions between the files provided to INGInious

CorrectOz is a ".jar" archive that groups the entire architecture presented in chapter 4, subsection 2.4, without the file *markers.txt*. This corresponds to the implemented interpreter that will simulate the execution of each Oz submission provided to INGInious.

Markers is the textual file presented in chapter 4, subsection 2.4 that stores the markers of error. It is separated from the previously described archive to ease its update. Indeed, even if this file is modified, the entire archive does not need to be re-uploaded on INGInious.

Template is a Python file that includes the pre-defined code in which the submissions are integrated before being graded. Each exercise has its own template. This template often contains pre-defined functions and unitary tests. Each one should implement the line "@@problem_id@@" which will be replaced by the students' submissions to form the entire code that should be interpreted. Notice that "problem_id" should itself be replaced by the identifier of the exercise. Fig 5.3 provides the template file for the second exercise of Louv1.1x called *BrowseX*. In this case, the second line will be replaced by the students' submissions.

```
1  local BrowseFormula in
2     proc {BrowseFormula}
3          @@BrowseX@@
4     end
5   {BrowseFormula}
6  end
```

Figure 5.3: template.py for *BrowseX* exercise

Task is a *bash* script that prepares the environment for the interpretation of a submission, then runs this interpretation. Each exercise has its own task file according to its requirements. Fig 5.4 provides an example of task that starts the interpretation of the file *input.oz* with 4 arguments: (1) debug prints on false, (2) two variables required, (3) one procedure required and (4) five calls to this procedure also required.

Run is the main *bash* script that starts the whole mechanism of grading when a submission is sent to the Docker environment. Fig 5.5 provides its implementation. First, the script integrates the submission into the pre-defined template through the command *parsetemplate* provided by INGInious. This command creates a new file called "input.oz" that contains the complete code to interpret. Then, the script runs

```
#!/bin/sh

export PATH=/usr/local/mozart/bin:$PATH
export CLASSPATH="/antlr-4.5-complete.jar:$CLASSPATH"
alias antlr4='java -Xmx500M -cp
"/usr/local/lib/antlr-4.5-complete.jar:$CLASSPATH" org.antlr.v4.Tool'

java -jar CorrectOz.jar input.oz -d false -nVar 2 -nFun1 -nCall 5
```

Figure 5.4: Example of task.sh

the *task* file related to the exercise. Finally, it displays the feedback provided by the interpreter.

```
#!/bin/sh

parsetemplate --output input.oz template.py
output=$(/task/task.sh)
if [ "$output" = "===== HINTS =====" ]; then
   # The student succeeded
   feedback --result success --feedback "You solved this difficult task
   !"
elif [[ "$output" == *"ERRORS"* ]]; then
   # The student failed
   feedback --result failed --feedback "$output"

else   feedback --result success --feedback "You solved this exercise
but you could have done better: $output"
fi
```

Figure 5.5: run.sh

2.3 Some exercises

This section provides some examples to show how the previously presented files can be implemented. As a reminder, each exercise has its own *task* and *template* files. A task starts the interpretation of a submission with the arguments required by the exercise and a template is a pre-defined code in which the submission is integrated.

2.3.1 CalledOnlyOnce

In this exercise, the students are asked to return three times the result of a pre-defined function while they should not call this function more than once. Therefore, in addition

to the usual feedback that could be provided by the interpreter, the number of procedure/function call should also be monitored. Indeed, CorrectOz should verify that this pre-defined method is called only once in the students' submissions. This monitoring can be performed by the interpreter by passing to it the argument "-nCall 2" which means that two calls to a procedure/function (SlowAdd and Delay) are allowed in the submission. In this exercise, the students do not need to define their own function because all they need to achieve is a unique call to the pre-defined function. Then, the argument "-nFun 1" can also be passed to the interpreter. This limits the number of function definitions to one. In fact, because there is already one function defined in the *template* file, this means that the student should not define any additional one. Moreover, the final submission should not be implemented with more than 2 variables. This limitation can be applied by passing the argument "-nVar 2". Fig 5.6 provides the command line stored in the *task* file to run correctly the interpreter and fig 5.7 provides the complete implementation of the *template* file.

```
java -jar CorrectOz.jar input.oz -d false -nVar 2 -nFun 1 -nCall 2
```

Figure 5.6: Command line for exercise CalledOnlyOnce

```
1  local X SlowAdd in
2    fun {SlowAdd X Y}
3      {Delay 1000}
4      X+Y
5    end
6
7    @@CalledOnlyOnce@@
8  end
```

Figure 5.7: Template file for exercise CalledOnlyOnce

2.3.2 Sum

In this exercise, the students have to implement the body of the *Sum* function. This function was already presented in chapter 3, subsubsection 3.3.1 and consists of computing the sum of the square of the N first integers. However this function should be tail recursive. Then, CorrectOz should monitor this property. The *task* file should pass the argument "-tailRec true" in order to achieve this monitoring. Moreover, the students should use the concept of accumulator which means that they should at least define one if-statement. Then, the argument "-nIf 1" should also be passed as an argument. Other arguments can be passed as well. The argument "-nVar 3" is used because the final submission should not contain more than three variables (in fact, they are already declared

in the template: MainSum, R and Sum). But, also, the argument "-Fun 2" is provided because the students are not supposed to define any additional function to *MainSum* and *Sum* that have already been declared. The *template* is provided by fig 5.9 and the command line of the *task* file by fig 5.8.

```
java -jar CorrectOz.jar input.oz -nVar 3 -nFun 2 -nIf 1 -tailRec true
```

Figure 5.8: Command line for exercise Sum

```
1  local MainSum R in
2    fun {MainSum N}
3      local Sum in
4        fun {Sum N Acc}
5          @@Sum@@
6        end
7        {Sum N 0}
8      end
9    end
10   R = {MainSum 8}
11 end
```

Figure 5.9: Template file for exercise Sum

2.3.3 Append

This is the first exercise in which the students are asked to handle the concept of list. In this case, they have to implement their own version of the Oz function *Append*. Therefore, it should be forbidden to use this function directly from Oz. The argument "-extProc [Append]" can be used to achieve this goal. Once again, many other arguments can be passed to provide a better feedback. For example, the argument "-list" should be *true* because the students have to use a list in this exercise. The arguments "-nCase 1" and "-nilCase true" should also be passed to the interpreter because handling a list always involves at least one case-statement with a "nil" pattern matching in Oz. Fig 5.10 provides the complete command line run by the *task* file and fig 5.11 provides the complete implementation of the *template* file.

```
  java -jar CorrectOz.jar input.oz -d false -nVar 2 -nFun 1 -nCase 1
-nilCase true -list true -extProc [Append] -tailRec true
```

Figure 5.10: Command line for exercise Append

```
1   local AppendLists R in
2     fun{AppendLists L1 L2}
3       @@Append@@
4     end
5   R = {AppendLists [123] [4 5 6]}
6   end
```

Figure 5.11: Template file for exercise Append

3 Integration of the actual configuration

CorrectOz itself is not able to confirm that a submission is correct. Indeed, there is no way to check if the results of the execution are the expected ones. This verification is performed by the actual configuration of INGInious for both courses. This configuration provides the required unit tests to achieve this verification. As stated previously, CorrectOz was not integrated into the actual configuration but to a completely new one because time was missing to configure both environments. This section describes how both environments could be grouped together.

The first step would be to install and configure the Mozart compiler into the new environment. Indeed, INGInious needs to compile and execute Oz files to perform the unit tests. Afterwards, there would be only a few lines to modify. As stated previously, the key idea is to replace the runtime and compilation verifications provided by the current configuration with CorrectOz which will provide better feedback. This is the goal achieved by the file *feedback.py*. In this file, the calls to the current verifications should be replaced by a call to CorrectOz such as described in fig 5.12 at lines 11 and 18.

```
# 1) Check if there is a runtime error.
# 2) If there is no runtime error, check the answer.
# 3) If there is no runtime error, and if there is no answer, check if there is a compilation error.
# 2 is before 3 because otherwise, warnings could block the process (it would say there is a compilation error,
# even if the code is correct but raises warnings.)
with open("errR.txt", "r") as errR: # 1)
    errRs = errR.read()
    if not errRs == "":
         os.system("java -jar CorrectOz.jar task.oz -d false")
    error = 0
    with open("out.txt","r") as out: # 2)
        outs = out.read()
        if not outs == "":
            checkStdout(outs)
        else: # 3)
            os.system("java -jar CorrectOz.jar task.oz -d false")
```

Figure 5.12: Modified feedback.py to integrate CorrectOz to the current configuration

4 Conclusion on the integration

From a student's perspective, INGInious is a great tool. It has a nice and user-friendly interface and it grades the submission rather quickly. Moreover, the tool reveals to be as practical and convenient for a developer as for a student. Indeed, INGInious provides a complete documentation for installing, configuring, extending or even integrating the tool inside another environment [14]. During its development, a special attention was paid to developers and teachers eager to use it.

CorrectOz could perfectly fit INGInious with a bit more time. First, the grading tool should perform its own unitary testings. Then, if the submission is unsuccessful, CorrectOz should run the submission and provide the complete feedback to help the student to improve his implementation. One last remaining improvement would be to clarify some statements of exercises and both Louv1.1x and Louv1.2x would see their overall quality of learning improving in the upcoming years.

Chapter 6

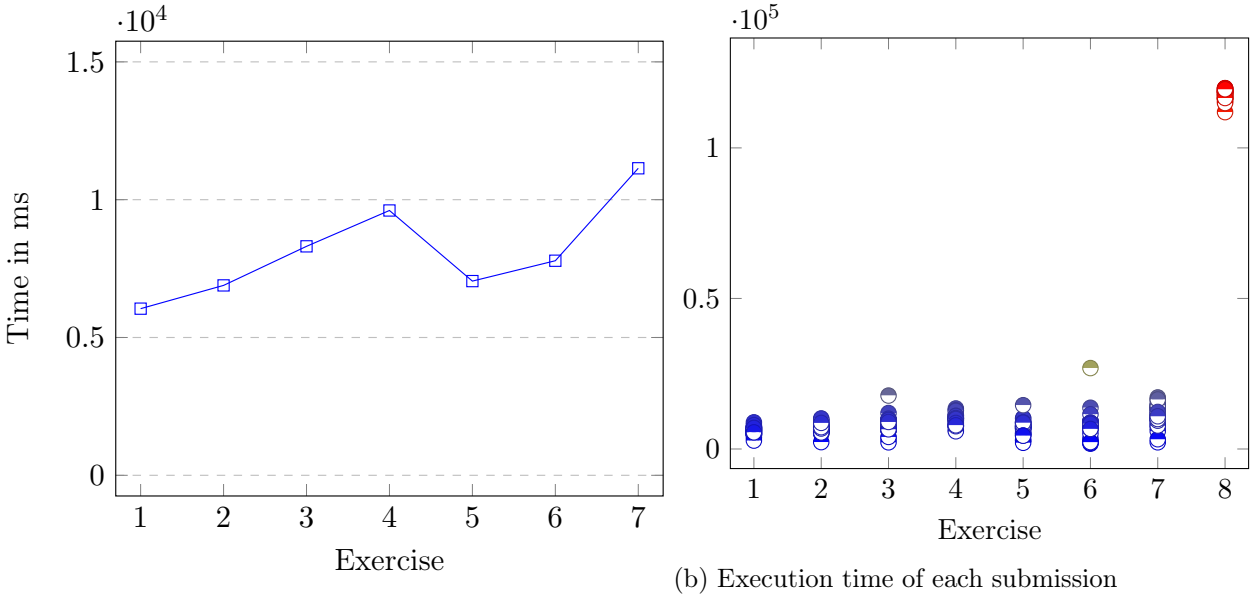
Evaluation

A software solution should not be approved before an intensive phase of testing. This chapter discusses the tests performed to assess the final quality of CorrectOz. Those tests have been run on several hundreds of submissions to evaluate two important features: (1) the execution time required to provide a feedback and (2) the correctness and the quality of the feedback provided.

1 Execution time

The execution time of a software is very important for the final users. No one likes to wait for an answer. Therefore, CorrectOz should provide a feedback in a limited amount of time. In the worst cases, the interpreter should output a feedback in less than 30 seconds. The tests have been run on height different exercises: (1) *CalledOnlyOnce*, (2) *Sum*, (3) *Infix*, (4) *FromImplicitToExplicit*, (5) *Append*, (6) *IsPrime*, (7) *IsBalanced* and (8) *Expressions*. These exercise have been carefully selected because they require the implementation of different data structures or the monitoring of different properties. For each exercise, 31 random submissions have been tested.

Fig 6.1a provides the average time required to interpret the submissions related to each exercise. This figure shows that it takes on average 6 to 11 seconds to provide a feedback. However, the figure also omits to display the results for the last exercise, called *Expressions*. It is the 5th exercise of Louv1.2x and requires the implementation of five classes. It appears that CorrectOz has some troubles to simulate the execution of a submission that implements the concept of class. Fig 6.1b provides the complete results that allow to compute the average time described by fig 6.1a for each exercise. On this figure, it can be observed that the execution time of the last exercise oscillates between 113 and 118 seconds. For clarity purpose, this result was not displayed on the first graph. In terms of execution time, CorrectOz provides efficient results except for exercises that requires the implementation of classes.



(a) Average execution time per exercise

(b) Execution time of each submission per exercise

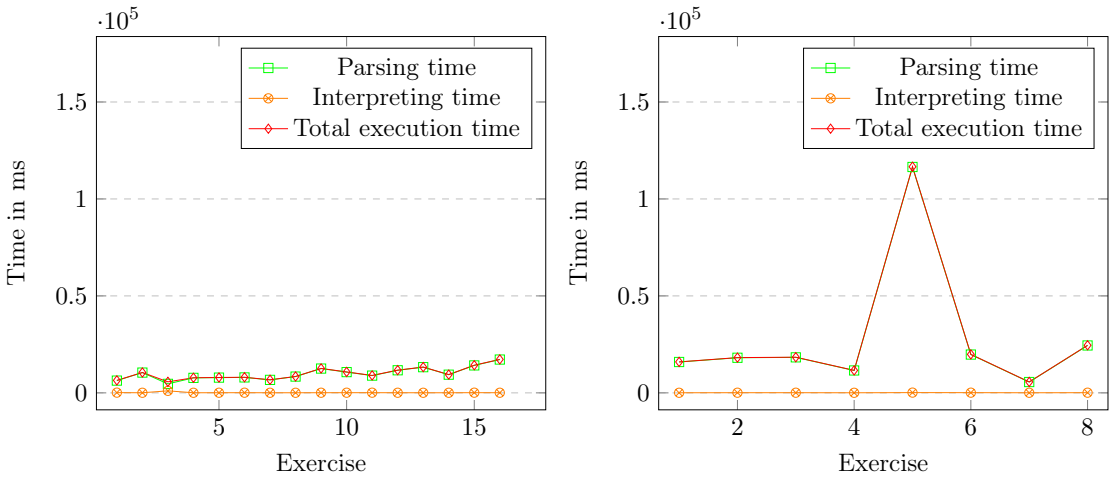
Figure 6.1

Comparison between parsing and interpreting time

Fig 6.2a and fig 6.2b provide a comparison between the execution time required to parse a submission and the execution time required to interpret a submission for each exercise of both courses. Both figures confirm the fact that the total execution time depends entirely upon the parser slowness. The interpreting time is completely insignificant. A quick comparison between both figures also shows that the parsing time increases for the exercises of LOUV1.2X. Mainly for the 5th exercise which, as stated previously, requires the implementation of classes. Therefore, the execution time issues are related to the parser generated by ANTLR4. This parser represents the main limitation of CorrectOz and a future improvement of the grammar, or even the entire parser, could be realized.

2 Correctness

The quality of CorrectOz also lies in its ability to provide a precise and accurate feedback. Therefore, it is essential to control the quality of the feedback provided. Unfortunately, it is not possible to confirm their accuracy without the verification of a human. Therefore, this part of the evaluation is very time consuming. Two kinds of tests have been performed to evaluate the correctness: (1) a handmade statistical analysis of the feedback, and (2) a comparison between the feedback provided by INGINIOUS and the feedback provided by the tool.



(a) Comparison of the execution times for Louv1.1x exercises

(b) Comparison of the execution times for Louv1.2x exercises

Figure 6.2

2.1 Statistics about correctness

The first evaluation procedure consists of comparing the feedback provided by CorrectOz with the actual mistakes for each submission tested. To draw table 6.1, the feedback provided for 248 submissions have been evaluated with this procedure. These submissions correspond to the ones previously tested to observe the execution time. The resulting feedback have been classified into different categories. First, a feedback is considered as *irrelevant* if the student's mistake was not found or if the reported error is too generic. A feedback can also be considered as a *false positive*. There are two kinds of false positives: the *strong* ones, and the *weak* ones. A *strong false positive* happens when CorrectOz does not find the correct mistake and provides an inaccurate feedback. The problem with these feedback is that they mislead the student towards a wrong mistake. In these cases, CorrectOz does more harm than good. Fortunately, there are only a few strong false positives. Finally, the *weak false positives* happen when an issue involves another issue which is itself reported by CorrectOz. Fig 6.3 provides an example of such a *weak false positive*. In this case, the student forgot to "end" the if-statement. The first feedback provided by the grader is accurate. Unfortunately, the missing "end" makes it hard to parse correctly the if-statement. Therefore, the interpreter also warns the student that he should provide a "else" clause to his conditional statement. This hint is not necessarily bad, but it should have been avoided. The *weak false positives* can still be considered as "relevant".

In conclusion, the feedback appear most of the time relevant and helpful for the students. Indeed, there are 80% relevant feedback according to table 6.1. However, there remain some misunderstandings that cannot be efficiently reported. These understanding

mistakes are often specific to one exercise and it was not possible to cover them all.

# Submissions tested:	248
# Irrelevant feedback:	42
# Weak false positives:	29
# Strong false positives:	5

Table 6.1: Statistics about correctness

Submission:

```
1   local Fact S in
2       fun {Fact N}
3         local
4           fun {FactList L N Acc}
5             if N==1 then L|Acc
6             else {FactList L|Acc N-1 N*Acc}
7           end
8         in
9           {Browse {FactList nil N 1}}
10        end
11      end
12    S = {Fact 3}
13  end
```

Provided feedback:

```
ERROR:  A " end " statement is missing in the if statement. [if <
condition> then <statement> (else <statement>) end].
Error found in " if N == 1 then L | Acc else { FactList L | Acc N - 1 N *
 Acc }".

HINT:  An " else " statement is missing in your if structure. it is
better to have a default case.
Error found in " if N == 1 then L | Acc else { FactList L | Acc N - 1 N *
 Acc }".
```

Figure 6.3: Example of weak false positive feedback

2.2 Comparison between INGInious and the tool

This subsection discusses the comparison between the feedback provided by INGInious and the ones provided by CorrectOz. This discussion intends to show in another way how

much the feedback provided by CorrectOz are improved in terms of quality. We compared the feedback given by CorrectOz with those given by the current tests for 56 different submissions among those presented in appendix **??**. The results are showed in table 6.2 and reveal that CorrectOz is more efficient in two-thirds of the cases. Five submissions are presented and discussed below. Because of length issue, we cannot provide more comparisons in this paper. More examples are provided in appendix **??**.

# Submissions tested:	56
# more relevant feedback:	38
# equivalent feedback :	18

Table 6.2: Comparison of current feedback with CorrectOz's feedback

2.2.1 Submission 1: Fact

The first submission corresponds to the same one as described in fig 6.3 in which the student forgot an "end" keyword to close the if-statement. Both graders detect that the keyword is missing as shown below. However, the error is revealed more precisely by the tool than INGInious. Finally, both graders provide a *weak false positive* feedback because of the missing keyword. INGInious thinks that the student tried to redefine *Fact* and the tool thinks that he forgot to write an "else" clause to the conditional statement.

Submission:

```
1   local Fact S in
2       fun {Fact N}
3         local
4           fun {FactList L N Acc}
5             if N==1 then L|Acc
6             else {FactList L|Acc N-1 N*Acc}
7           end
8         in
9           {Browse {FactList nil N 1}}
10        end
11      end
12    S = {Fact 3}
13  end
```

INGInious feedback:

```
Did you write the signature of Fact? Be careful, you are asked to provide
 the body only.
The message "Parse error" often means that you have forgotten a closing
bracket, a "end", etc. Or maybe, there are too much brackets, "end", etc
.! Take a look at the error line. The line may be incorrect because if an
 end is missing, for instance, it looks too far away for the error.

The error given by the compiler is:

Mozart Compiler 2.0.0-alpha.0+build.4091.61fe075-dirty (Mon, 14 Jul 2014
20:27:57 +0200) playing Oz 3

%%% feeding file exercise.oz

%************************* parse error ***********************
%**
%** Parse error
%**
%** in file "exercise.oz", line  20, column 1
%** ------------------ rejected (1 error)
```

CorrectOz feedback:

```
ERROR :  An " end " statement is missing in the if statement. [if <
condition>then <statement> (else <statement>) end]. Error found in " if N
 == 1 then L | Acc else { FactList L | Acc N - 1 N * Acc }".

HINT:  An " else " statement is missing in your if structure. it is
better to have a default case. Error found in " if N == 1 then L | Acc
else { FactList L | Acc N - 1 N * Acc }".
```

2.2.2 Submission 2: Fact

This submission provides a second example for the exercise *Fact*. In this case, the student provides the entire function declaration for *Fact*. However, this function is already defined in the *template* of the exercise and the students are only asked to implement the body of the function. As shown below, CorrectOz reveals the error in two different ways. First, it recalls to the student that he should not use any "declare" statement. Then, it explains that there is already a definition for the function *Fact*. Finally, CorrectOz provides a useful hint to improve the implementation because the variable *Acc1* is indeed never used. Notice that INGInious also highlights the correct mistake but the grader lacks precision in comparison with CorrectOz.

Submission:

```
1   local Fact S in
2     fun {Fact N}
3       declare
4       fun {Fact N}
5         local Aux Aux1 in
6           fun {Aux1 N Acc1 Acc2}
7             if N == 0 then Acc2
8             else {Aux1 N-1 ({Aux N 1 Acc2}) Acc2 } end
9           end
10          local Aux in
11            fun {Aux N Acc Acc2}
12              if N == 0 then Acc|Acc2
13              else {Aux N-1 N * Acc Acc2} end
14            end
15          end
16          {Aux N 1 nil}
17        end
18      end
19    end
20    S = {Fact 3}
21  end
```

INGInious feedback:

```
Did you write the signature of Fact? Be careful, you are asked to provide
 the body only.
The message "Parse error" often means that you have forgotten a closing
bracket, a "end", etc. Or maybe, there are too much brackets, "end", etc
.! Take a look at the error line. The line may be incorrect because if an
 end is missing, for instance, it looks too far away for the error.

The error given by the compiler is:

Mozart Compiler 2.0.0-alpha.0+build.4091.61fe075-dirty (Mon, 14 Jul 2014
20:27:57 +0200) playing Oz 3

%%% feeding file exercise.oz
%************************** parse error ***********************
%**
%** Parse error
%**
%** in file "exercise.oz", line  2, column 1
%** ------------------ rejected (1 error)
```

CorrectOz feedback:

```
ERROR :  You should use a local instead of declare.  Error found in "
declare".

ERROR:  The procedure/variable already exists. Maybe the procedure you
want to implement is already implemented. Otherwise, change the name of
your procedure. The variable(s) concerned:[Fact].
Error found in " fun { Fact N } local Aux Aux1 in fun { Aux1 N Acc1 Acc2
} if N == 0 then Acc2 else { Aux1 N - 1 ( { Aux N 1 Acc2 } ) Acc2 } end
end local Aux in fun { Aux N Acc Acc2 } if N == 0 then Acc | Acc2 else {
Aux N - 1 N * Acc Acc2 } end end end { Aux N 1 nil } end end".

HINT : The variable Acc1 is not used. It is probably useless.
```

2.2.3 Submission 3: Fact

Here is a third submission for the exercise *Fact*. In this submission, the student redefines the *Fact* function with 2 arguments. As shown below, INGInious feedback is rather inaccurate. It states that the function is used with a wrong arity. This will not help the student to understand that he should not have redefined it. Moreover, CorrectOz provides a better feedback. First, it advises the student that the function *Fact* is already declared and that he should modify the name of his own function. Then, it provides two useful hints for the next submission: (1) the function is not yet tail recursive and (2) the input provided to the function does not match any pattern in the case-statement. In this case, the tool is really a game changer for the student.

Submission:

```
1     local Fact S in
2       fun {Fact N}
3         fun {Fact N Acc}
4           if N<2 then Acc
5           else {Fact N-1 N*Acc} end
6         end
7       in
8         case N of nil then nil
9         [] H|T then {Fact H 1}|{Fact T}
10        end
11      end
12      S = {Fact 3}
13    end
```

INGInious feedback:

```
It seems you used "{Fact T _}" with a wrong arity (wrong number of
arguments for instance).
Maybe this tip can help: if you use functions, do not forget to "handle"
the returned value.

The error given by the compiler is:

Mozart Compiler 2.0.0-alpha.0+build.4091.61fe075-dirty (Mon, 14 Jul 2014
20:27:57 +0200) playing Oz 3

%********************* static analysis error ******************
%**
%** illegal arity in application
%**
%** Arity found:          2
%** Expected:             3
%** Application (names):  {Fact T _}
%** Application (values): {<P/3> _<optimized> _<optimized>}
%** in file "exercise.oz", line  7, column 31
%** ------------------ rejected (1 error)
```

CorrectOz feedback:

```
ERROR:  The procedure/variable already exists. Maybe the procedure you
want to implement is already implemented. Otherwise, change the name of
your procedure. The variable(s) concerned:[Fact]. Error found in " fun {
Fact N Acc } if N < 2 then Acc else { Fact N - 1 N * Acc } end end".

HINT: Those functions are not tail recursive: [Fact]

HINT:  The case/if statement does not match any pattern for the
expression. You should have an else statement. The variable(s) concerned
:[N (3 )]. Error found in " case N of nil then nil [] H | T then { Fact H
 1 } | { Fact T } end".
```

2.2.4 Submission 4: Append

The fourth selected example is a submission for the exercise *Append*. It consists of implementing the so-called *Append* function initially provided by Oz. In this case, the student provides a bad recursive call. He is supposed to call recursively the function with the tail of the first list but he calls it with the entire list once again. Therefore, his submission will run forever without any progress. This is the exact feedback provided by INGInious. The grader detects that the recursive function does not terminate but does not provide any additional information. Fortunately, CorrectOz provides more feedback.

First, it detects which function is running infinitely. Then, it provides three hints: (1) warns the student that his function is not tail recursive, (2) warns him that the procedure call is applied several times with the same arguments and (3) warns him that the variable "T" is not used. These three hints are essential to understand that the mistake lies in the recursive call.

Submission:

```
1     local AppendLists R in
2       fun {AppendLists L1 L2}
3         case L1 of nil then L2
4         [] H|T then H|{AppendLists L1 L2}
5         end
6       end
7       R={AppendLists [1] [2]}
8     end
```

INGInious feedback:

```
There is a runtime error! It seems a part of your code generate a large
amount of memory allocation. This may happen if one of your recursive
functions does not terminate.

The error given by the emulator is:

FATAL: The active memory (95325008) after a GC is over the maximal heap
size threshold: 95325000
Terminated VM 1
Test failed Error

Your code does not pass all tests. More precisely: There is an infinite
loop with the call: {AppendLists [a b c] nil}
```

CorrectOz feedback:

```
ERROR:  Your program runs infinitely, it can be from one of those parts
of the program:
[ fun { AppendLists L1 L2 } case L1 of nil then L2 [] H | T then H | {
AppendLists L1 L2 } end end].
You call Argument "L2" with value "[ 2 ]" at position "2" in method "
AppendLists" 307 times. Maybe you forgot to update it.
You call Argument "L1" with value "[ 1 ]" at position "1" in method "
AppendLists" 307 times. Maybe you forgot to update it.
```

```
HINT: Those functions are not tail recursive: [AppendLists]

HINT:  You call the procedure several times with the same argument(s).
You could maybe store the result in a variable to avoid to call several
times the function. Or maybe did you forget to update the argument ? The
arguments are: [Argument "L1" with value "[ 1 ]" at position "1" in
method "AppendLists", Argument "L2" with value "[ 2 ]" at position "2" in
 method "AppendLists"].
The variable(s) concerned:[AppendLists]. Error found in " { AppendLists
L1 L2 }".

HINT: The variable T is not used. It is probably useless.
```

2.2.5 Submission 5 : IsPrime

The fifth and last example is a submission for exercise *IsPrime*. It consists of returning "true" if the input integer is a prime number, "false" otherwise. In this case, the function is declared with a starting lowercase character which is strictly forbidden in Oz. This syntax error may seem easy to reveal, nevertheless INGInious' feedback notably lacks precision. CorrectOz provides four different ways to understand the error and always states exactly where the error was found. The main improvement is that it recalls the syntax rule for declaring a variable to the student.

Submission

```
1     local R Prime in
2       fun {Prime N}
3         local isPrime in
4           fun {isPrime N Acc}
5             if N == Acc then true
6             else
7               if (N mod 2) == 0 then false
8               else {IsPrime N Acc + 1}
9               end
10            end
11          end
12          {IsPrime N 1}
13        end
14      end
15      R = {Prime 7}
16    end
```

INGInious feedback:

```
The variable "IsPrime" in your code has not been introduced/declared.
The variable "IsPrime" in your code has not been introduced/declared.
At line(s): it seems you used an expression (did you try to return
something?) instead of a statement (for instance X = 42).
The error given by the compiler is:

Mozart Compiler 2.0.0-alpha.0+build.4091.61fe075-dirty (Mon, 14 Jul 2014
20:27:57 +0200) playing Oz 3

%%% feeding file exercise.oz

%************************* syntax error ************************
%**
%** expression at statement position
%**
%** in file "exercise.oz", line  1, column 9

%********************* binding analysis error ******************
%**
%** variable IsPrime not introduced
%**
%** in file "exercise.oz", line  5, column 22

%********************* binding analysis error ******************
%**
%** variable IsPrime not introduced
%**
%** in file "exercise.oz", line  9, column 4
%** ------------------ rejected (3 errors)
```

CorrectOz feedback:

```
ERROR :  A variable should always start with an upper case The variable(s
) concerned :[isPrime]. Error found in " isPrime".

ERROR :  A variable should always start with an upper case.
Error found in " fun { isPrime N Acc } if N == Acc then true else if ( N
mod 2 ) == 0 then false
else { IsPrime N Acc + 1 } end end end".

ERROR :  The variable is not declared. The variable(s) concerned :[
IsPrime]. Error found in " IsPrime".

ERROR :  The procedure you want to call is not declared. Maybe did you
mean one of these : {isPrime, Prime}.
The variable(s) concerned :[IsPrime]. Error found in " { IsPrime N 1 }".
```

3 Conclusion on evaluation

The evaluation of CorrectOz can be considered successful. First, because the execution time is on average short enough unless classes are implemented in the submission. This is a well-known issue that could be improved in future works. Secondly, because the quality of the feedback provided is improved by CorrectOz. Indeed, the feedback appear better, or at least, more precise than the feedback provided by INGInious. Yet, some improvements could be implemented to reveal some mistakes very specifically related to one exercise or another.

Chapter 7

Conclusion

The implementation of an interpreter is a difficult and rigorous task. This is particularly true when the final objective is not only to develop an interpreter, but to turn it into a complete feedback provider. CorrectOz is the result of several months of research and development. CorrectOz takes codes written in Oz language as input and delivers an intelligent feedback about the syntax and runtime errors encountered in the submissions. These feedback are classified into two categories, hints and errors, depending on the nature of the mistake. Many steps were needed to have this final product. We first produced a parser by using ANTLR4, a powerful tool to produce a parser and a lexer from an EBNF grammar. Afterwards, we implemented an interpreter to execute the code parsed by ANTLR4. In parallel to the implementation of the interpreter, we analysed several hundreds of submissions to spot the recurrent errors and bad habits of the students registered on the MOOCs. The analysis of these codes allowed us to list these errors as markers that will be used in our interpreter. We ended with tens of markers that encompass the relevant set of possible errors. Once CorrectOz was able to cover the exercise of the first MOOC, the final step was the integration into INGInious, the feedback provider tool used for the two MOOCs of Prof. Peter Van Roy. The integration into INGInious was really straightforward thanks to the extensive documentation.

The implemented tool proved to be efficient in terms of execution time. It was also established that CorrectOz provides more helpful feedback than the current tests performed on INGInious.

However, some issues were encountered along the way. Some have been fixed successfully, but a few challenges still remain. Those issues are discussed in the final section. They are the intangible evidence that a software solution should never cease to evolve. This is a key requirement, to stick to the customers' needs, over and over again.

1 Future work

The project was ambitious and grouped many different notions of computer science such as the creation of an interpreter, the adaptation of a parser and even the integration of the solution into an existing tool. Even though CorrectOz appears to be efficient for many applications, some of them could be improved and extended. This final section discusses these improvements.

1.1 Complete the supported semantic

So far, CorrectOz does not cover all the semantic rules provided by Oz. The threads still need to be implemented as well as a few unusual structures and a lot of pre-defined functions. These features only need to be implemented in the interpreter because the grammar and the parser are already able to recognize them.

1.2 Integrate unitary testing

CorrectOz does not check the final output provided by a submission. It focuses only on improving the quality of the feedback. Therefore, it is currently necessary to integrate it into the INGInious environment used by the students. Indeed the grader provides the unit tests needed to approve or disapprove the correct operation of a submission for each exercise. If the submission does not provide the expected answers, then CorrectOz should be run on the submission to spot the programming mistakes. The integration into INGInious has already been discussed but CorrectOz is still not coupled with the unit tests provided by the grader. This is feature, which was already discussed in chapter 5, section 3, requires resolution.

1.3 Improve the syntax verifier

Unfortunately, CorrectOz does not always behave as expected. Some feedback appear to be irrelevant. At the beginning of the development, the focus was oriented towards the interpreter itself. However, many mistakes revealed themselves even before the interpretation of the submission. Therefore, the focus later moved towards the *SyntaxVerifier*. This part of the tool could be improved to detect many understanding issues related to Oz more quickly.

1.4 Avoid weak false positives

The weak false positives, as explained in chapter 6, section 2.1, are false positives that are caused by a previously detected error in the submission. The source of the error is provided as feedback, but other erroneous markers are also provided. The problem is that CorrectOz keeps looking for new mistakes even if one has already been detected. Therefore, one way to avoid such weak false positives would be to stop CorrectOz right after the first encountered error. However, such a practice could increase the number of strong false positives, since it will depend on the order in which the tests are performed.

If the first verification is performed on the erroneous mistake caused by the actual error, it will display an erroneous feedback and then stop the execution. Another solution could be to identify the errors that can lead to weak false positives and to refine them in order to avoid such behaviours.

1.5 Execution time issues with the parser

The evaluation proved that the final solution had some troubles with the simulation of submissions implementing the concept of *class*. The parser generated by ANTLR4 appears to be the source of the problem. It also becomes slower when the length of the submissions grows. Fortunately, the programming exercises in LOUV1.1X and LOUV1.2X do not require lengthy answers. A potential extension would be to improve the parser and these execution timing issues should be solved efficiently.

Chapter 8

Abbreviations

ANTLR : ANother Tool for Language Recognition

AST : Abstract Syntax Tree

EBNF : Extended Backus-Naur Form

MIT : Massachusetts Institute of Technology

MOOC : Massive Open On-line Course

UCL : Université Catholique de Louvain

VM : Virtual Machine

Bibliography

[1] edX. Louv1.1x Home Page. `https://courses.edx.org/courses/course-v1:LouvainX+Louv1.1x+3T2015/info`.

[2] edX. Louv1.2x Home Page. `https://courses.edx.org/courses/course-v1:LouvainX+Louv1.2x+4T2015/info`.

[3] Ken Masters. A Brief Guide to Understanding MOOCs. *The Internet Journal of Medical Education*, 1(2), 2011.

[4] Loufti Nuaymi. MOOC : « Apprendre n'importe où, n'importe quand, tout ce qu'on voudrait ». *L'Obs avec Rue89*, 2013.

[5] edX. About Us. `https://www.edx.org/about-us`.

[6] Isabelle Decoster. Enseignement UCL : Plus de 50 000 étudiants inscrits aux cours UCL sur edX. *Service de presse et communication - Université Catholique de Louvain*, 2014.

[7] Olivier Eggermont. L'enseignement online entre dans les universités belges. *La Libre*, 2014.

[8] Guillaume Derval, Anthony Gégo, Pierre Reinbold, Benjamin Frantzen, and Peter Van Roy. Automatic Grading of Programming Exercises in a MOOC Using the INGInious Platform. *EMOOCS 2015 (Third MOOC European Stakeholders Summit)*, Mons, Belgium, May 18-20, 2015.

[9] ANTLR4. About the ANTLR Parser Generator. `http://www.antlr.org/about.html`.

[10] Peter Van Roy and Seif Haridi. *Concepts, Techniques, and Models of Computer Programming*. The MIT Press, 1st edition, 2004.

[11] Terence Parr. *The Definitive ANTLR 4 Reference*. Pragmatic Bookshelf, 2nd edition, 2013.

[12] Campbel Bill, Lyer Swami, and Akbal-Delibas Bahar. *Introduction to Compiler Construction in a Java World*. CRC Press, 2012.

[13] Docker. What is Docker? https://www.docker.com/what-docker.

[14] the INGInious team. Inginious' documentation. http://inginious.readthedocs.io/en/latest/index.html.

[15] Antlr 4: A case study. http://mattjquinn.com/2014/01/19/antlr4-case-study.html.

[16] Olivier Zeigermann. Code generation cambridge 2013: Introduction to parsing with antlr4. https://docs.google.com/presentation/d/1XS_VIdicCQVonPK6AGYkWTp-3VeHfGuD2l8yNMpAfuQ/edit?pli=1#slide=id.p, 2013.

[17] Terence Parr. Antlr4. https://vimeo.com/59285751.

[18] German National Research Center For Information Technology Fraunhofer Institure For Computer Architecture and Software Technology. The catalog of compiler construction tools. http://catalog.compilertools.net/.

[19] Jack Crenshaw. Let's build a compiler. http://compilers.iecc.com/crenshaw/, 1988-1995.

[20] Pierre Bouilliez. Glass cat – a tool for interactive visualization of the execution of oz programs in the pythia platform. *Université Catholique de Louvain*, 2014.

[21] Spivak Ruslan. Let's build a simple interpreter. https://ruslanspivak.com/lsbasi-part1/, 2015.

Rue Archimède, 1 bte L6.11.01, 1348 Louvain-la-Neuve **www.uclouvain.be/epl**